Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Readings:	Lecture material
Discussions:	Discussion 2 postings end Tuesday, April 2 at 6:00 AM ET
Assignments:	Assignment 2 due Tuesday, April 2 at 6:00 AM ET
Live Classroom:	Wednesday, March 27 from 7:00 - 9:00 PM ET

HTML5 Drag and Drop

Introduction

HTML5 provides native Drag and Drop capability for web applications. The user selects an element and drags it onto a target element. Any element in the web page can be made draggable and event handlers associated with the drag source and the drag target process the drag and drop operation. The following figure shows the basic user action when dragging from the source onto the target.

Specifying Draggable Items

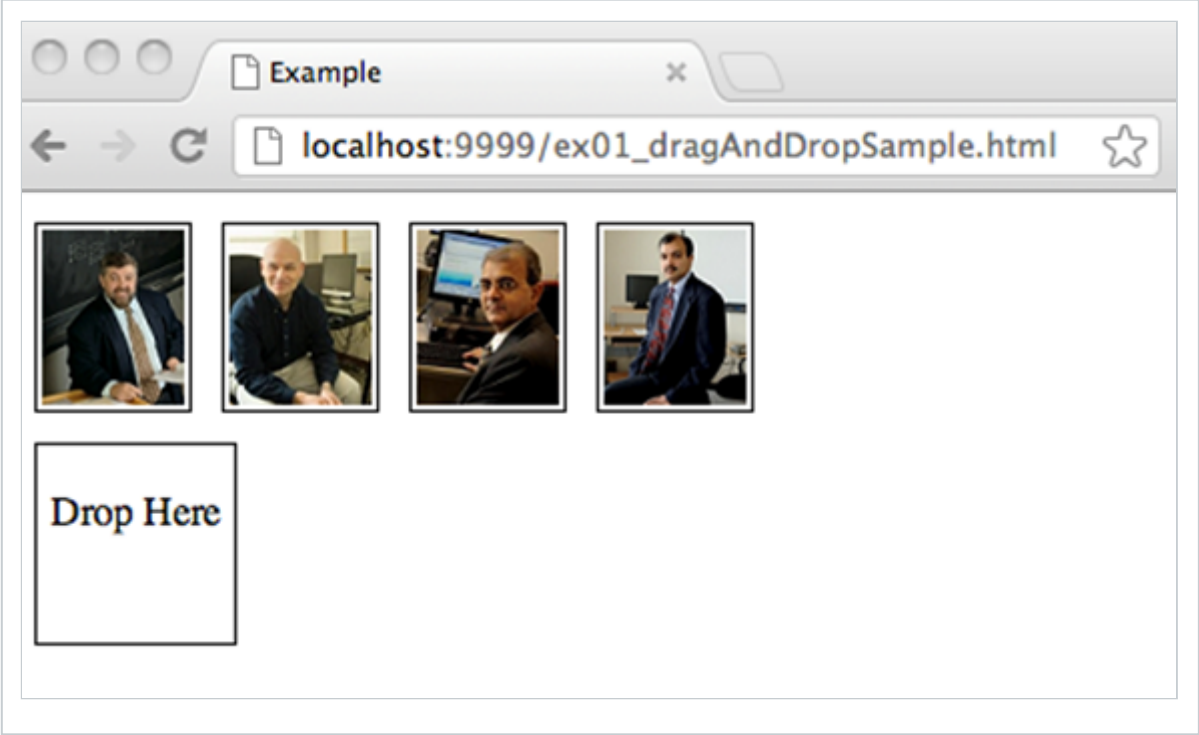
The HTML elements in the document that can be dragged are specified through the *draggable* attribute for each element. When the *draggable* attribute of an element is set to *true*, the element can be dragged. When set to *false*, the element cannot be dragged. The value can also be specified as *auto*, in which case, the browser determines whether the element may be dragged or not. Links and image elements are *draggable* by default.

In the example shown below, the *draggable* attribute of the *img* elements is explicitly set to *true*. The *div* (id = “src”) section will be used as the source of the drag operations. The elements will be dropped into the *div* section (id = “target”).

ex01_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>

        <div id="target">
            <p>
                Drop Here
            </p>
        </div>
    </body>
</html>
```

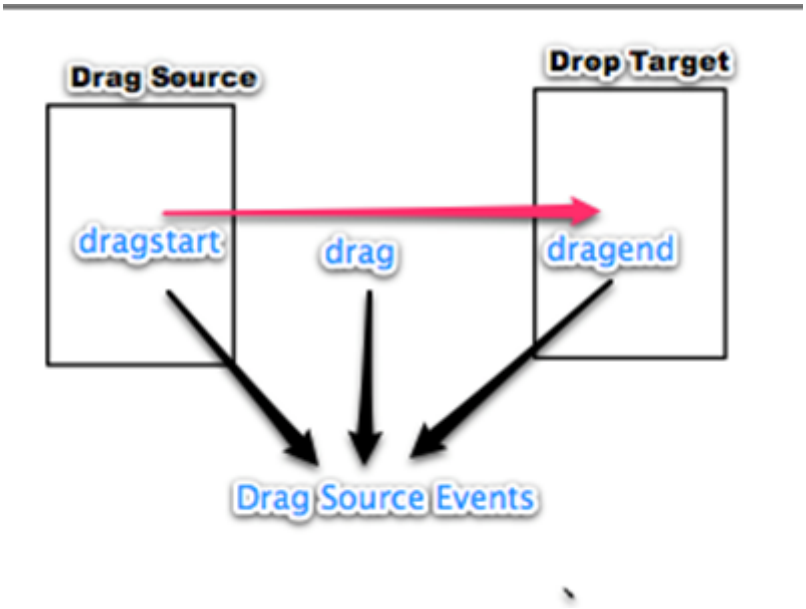
The above *html* is rendered as shown below.



When the user drags any of the images into the target, nothing happens. The drop is rejected. Event handlers need to associated with the various drag and drop events to make the drag and drop successful.

Specifying Source Event Handlers

When an element is selected by clicking with the mouse and then dragged, the drag process starts at that moment. The *dragstart, drag, and dragend* events are associated with the drag source element and the registered event handlers will take the necessary steps for the drag operation. The following figure shows the events associated with the drag source element.



The *dragstart* event is fired when the user selects an element which is *draggable* with the mouse click and starts the dragging operation. The *dragstart* event is fired on the drag source. As the user is dragging the source element

around the web page, the *drag* event is repeatedly fired on the drag source. When the user releases the mouse, the *dragend* event is fired on the drag source indicating that the drag operation has ended. Whether the source element is actually dropped on the target or not is controlled by the event handlers of the drop target.

The following example includes the *JavaScript* code for handling the drag source events.

ex02_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex02_dragAndDropSample.js"></script>

        <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>

        <div id="target">
            <p id="msg">Drop Here</p>
        </div>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The code only handles the drag source events and hence the drop operation is not yet implemented. The code below registers the event handlers for *ondragstart*, *ondrag,* and *ondragend*. The *target* property of the event in the handlers is the drag source element. When the drag operation starts, the *dragged* style is added which reduces the opacity of the dragged source. When the drag operation ends, the *dragged* style is removed.

ex02_dragAndDropSample.js

```
1  window.onload = init;
2
3  var  src, msg;
4
5  function init() {
6      src = document.getElementById("src");
7      msg = document.getElementById("msg");
8
9      // Add event handlers
10     src.ondragstart = dragStartHandler;
11     src.ondragend   = dragEndHandler;
12     src.ondrag      = dragHandler;
13 }
14
15 function dragStartHandler(e) {
16     e.target.classList.add("dragged");
17 }
18
19 function dragEndHandler(e) {
20     e.target.classList.remove("dragged");
21     msg.innerHTML = "Drop Here";
22 }
23
24 function dragHandler(e) {
25     msg.innerHTML = "Dragging " + e.target.id;
26 }
```

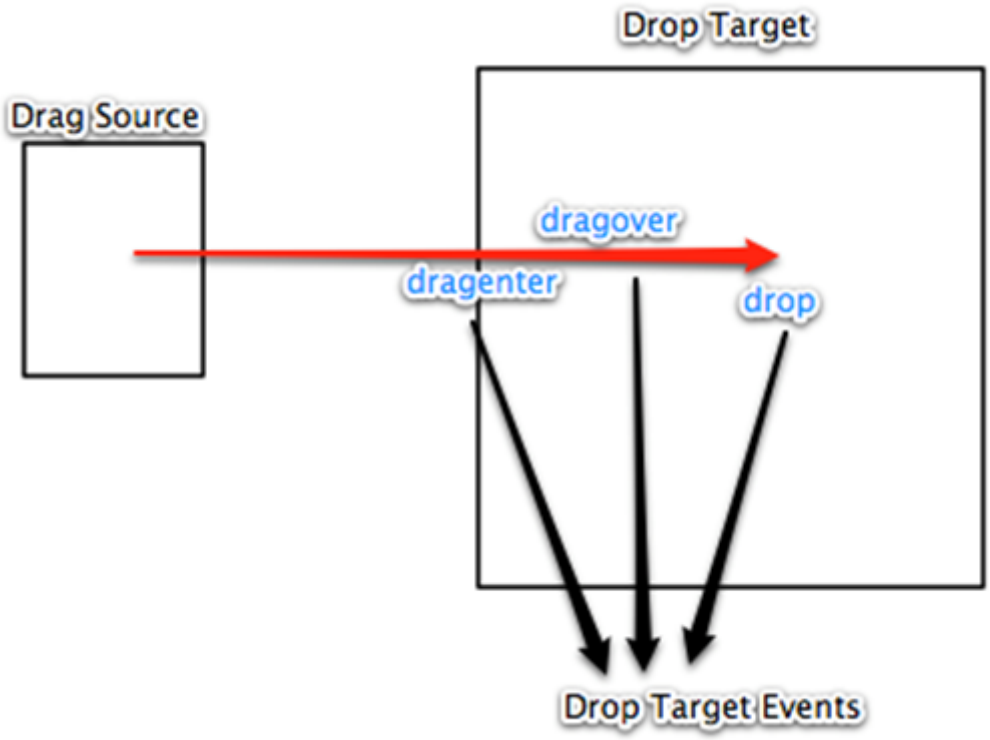
The CSS file associated with the example is shown below.

styles.css

```
#target, #src > img {
    border: thin solid black;
    padding: 2px;
    margin: 4px;
}

#target {
    height: 75px;
    width: 75px;
    text-align: center;
    display: table;
}

.dragged {opacity: 0.5;}
```

Specifying Target Event Handlers

In the previous example, when the source element is dragged over the target, the drag operation is rejected by default and hence when the drag ends, the source element is not dropped over the target. The events *dragenter, dragover,* and *drop* are associated with the drop target. The event handles for these events must explicitly accept the source element for it to be dropped on the target. The following figure shows the events associated with the drop target element.



The *dragenter* event is fired when the element being dragged enters the area of new element on the page. The event is fired on the new element. This new element could be the drop target or an intermediate element in between. The event handler may decide whether to accept the element being dropped. The default is not to accept the element. The *dragover* event is fired frequently at regular intervals as the mouse dragging the source element moves over an element during the dragging phase. This event is fired on the current target of the mouse. The *drop* event is fired on the target element when the user releases the mouse on the target. The event handler for this event will handle the code for the drop operation.

The *dragenter* and *dragover* events are illustrated with the example shown below.

ex03_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex03_dragAndDropSample.js"></script>

        <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>

        <div id="target">
            <p id="msg">Drop Here</p>
        </div>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The code handles the drag source events and the two drop target events (except the *drop*) and hence the drop operation is not yet implemented. The code below registers the event handlers for *ondragstart*, *ondrag,* and *ondragend* on the drag source, and the *ondragenter and ondragover* on the drop target.

ex03_dragAndDropSample.js

```
1  window.onload = init;
2
3  var  src, target, msg;
4
5  function init() {
6      src = document.getElementById("src");
7      target = document.getElementById("target");
8      msg = document.getElementById("msg");
9
10     // Add event handlers for the source
11     src.ondragstart = dragStartHandler;
12     src.ondragend = dragEndHandler;
13     src.ondrag = dragHandler;
14
15     // Add event handlers for the target
16     target.ondragenter = dragEnterHandler;
17     target.ondragover = dragOverHandler;
18 }
```

The code for the event handlers is shown below. The *target* property of the event in the first three handlers is the drag source element. However, the *target* property of the event in the last two handlers is the drop target element. The browser's default behavior is not to accept the element being dropping on the target. The *preventDefault* method of the event is used to tell the browser that the target is a valid target for dropping the element.

ex03_dragAndDropSample.js

```
19
20  function dragStartHandler(e) {
21      e.target.classList.add("dragged");
22  }
23
24  function dragEndHandler(e) {
25      e.target.classList.remove("dragged");
26      msg.innerHTML = "Drop Here";
27  }
28
29  function dragHandler(e) {
30      msg.innerHTML = "Dragging " + e.target.id;
31  }
32
33  function dragEnterHandler(e) {
34      msg.innerHTML = "Drag Entering " + e.target.id;
35      e.preventDefault();
36  }
37
38  function dragOverHandler(e) {
39      msg.innerHTML = "Drag Over " + e.target.id;
40      e.preventDefault();
41  }
```

Specifying Target Drop Event Handler

In order to accept the element being dropped, the *ondrop* event handler on the drop target needs to be implemented. The previous example is modified to take case of this scenario.

The HTML file is shown below. The *msg* paragraph element is separated from the *target* div element.

ex04_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex04_dragAndDropSample.js"></script>

        <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>
        <p/>
        <div id="target">

        </div>

        <p id="msg">Drop over the above target</p>

    </body>
</html>
```

The associated *JavaScript* code is shown below. The code handles the drag source events and the drop target events. The code below registers the event handlers for *ondragstart*, *ondrag,* and *ondragend* on the drag source, and the *ondragenter, ondragover* and *ondrop* on the drop target. The *sourceId* variable will be used to keep track of the *id* of the drag source.

ex04_dragAndDropSample.js

```
1  window.onload = init;
2
3  var src, target, msg;
4  var sourceId;
5
6  function init() {
7      src = document.getElementById("src");
8      target = document.getElementById("target");
9      msg = document.getElementById("msg");
10
11      // Add event handlers for the source
12     src.ondragstart = dragStartHandler;
13     src.ondragend = dragEndHandler;
14     src.ondrag = dragHandler;
15
16     // Add event handlers for the target
17     target.ondragenter = dragEnterHandler;
18     target.ondragover = dragOverHandler;
19     target.ondrop = dropHandler;
20 }
```

The *ondragstart* event handler is used to store the *id* of the drag source into the *sourceId* variable. When the drag ends, the *dragged* style class is removed from the elements which would the drag source and the drop target.

ex04_dragAndDropSample.js

```
21
22  function dragStartHandler(e) {
23      sourceId = e.target.id;
24      e.target.classList.add("dragged");
25  }
26
27  function dragEndHandler(e) {
28      msg.innerHTML = "Drag Ended";
29      var elems = document.querySelectorAll(".dragged");
30      for(var i = 0; i < elems.length; i++) {
31          elems[i].classList.remove("dragged");
32      }
33  }
34
35  function dragHandler(e) {
36      msg.innerHTML = "Dragging " + e.target.id;
37  }
```

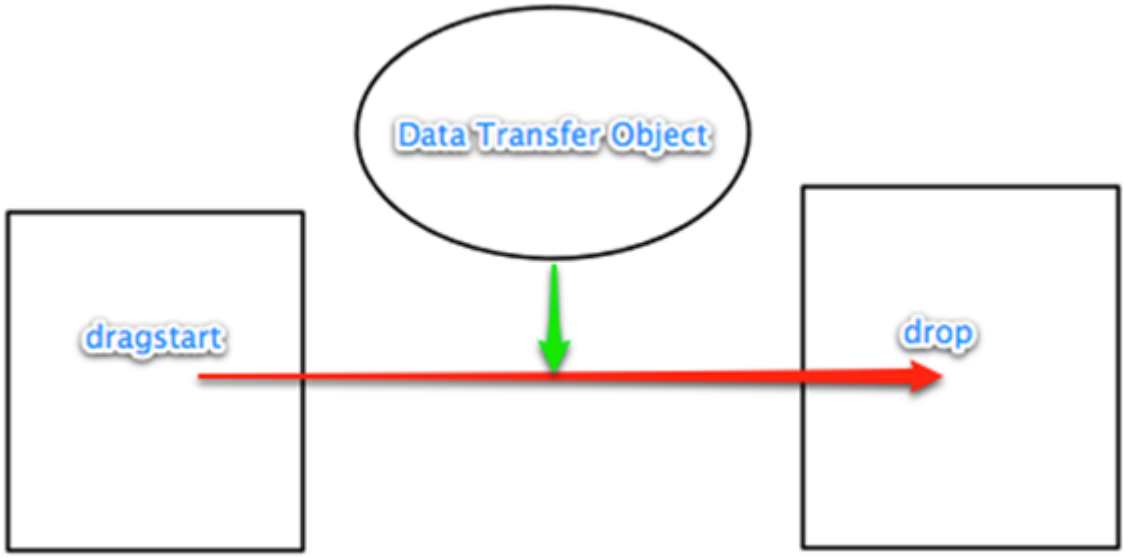
The *ondrop* event handler is now responsible for handling the drag source being dropped on the target. In this example, the element that corresponds to *sourceId* is cloned and added to the drop *target*. The *preventDefault* method informs the browser not to do the default operation.

ex04_dragAndDropSample.js

```
38
39  function dragEnterHandler(e) {
40      msg.innerHTML = "Drag Entering " + e.target.id;
41      e.preventDefault();
42  }
43
44  function dragOverHandler(e) {
45      msg.innerHTML = "Drag Over " + e.target.id;
46      e.preventDefault();
47  }
48
49  function dropHandler(e) {
50      var sourceElement = document.getElementById(sourceId);
51      var newElement = sourceElement.cloneNode(false);
52      target.innerHTML = "";
53      target.appendChild(newElement);
54      e.preventDefault();
55  }
```

Drag Data Store

In the examples seen so far, the drag source and the drop target are coupled to each other. The *ondragstart* event handler identifies the drag source and stores the information in a global variable. The *ondrop* event handler looks at the information stored in the global variable and takes the necessary action. To decouple the source and the target. the *drag data store* defines the data of the underlying drag and drop operation as one or more *data transfer* objects. The *data transfer* object is set during the *drag start* and accessed during the *drop* phase of the drag and drop operation as shown in the figure below.



The *data transfer* object is used to set the drag data and get the drop data during the drag and drop operation. The object is accessed through the *dataTransfer* property of the *event* argument of the *ondragstart* and *ondrop* event handlers. The following methods and properties are applicable for the *data transfer* object:

- ***setData*(format, data)** — registers one data transfer items of the specified MIME format during the *dragstart* event.
- ***getData*(format)** — retrieves the registered data transfer item of the specified MIME format.
- ***types*** — returns an array of the registered MIME formats.
- ***items*** — returns a list of all the registered data transfer items along with their associated formats.
- ***files*** — returns a list of all the files associated with the drop, if any.
- ***clearData*([format])** — clears all the registered data transfer items. If the MIME format is specified, only the registered item of that type is cleared.
- ***setDragImage*(element, x, y)** — specifies that an *existing img* element will be used as the drag image. The *x* and *y* values specify the relative position of the mouse with respect to the image.
- ***addElement*(element)** — an alternative way to specify the drag image. This method automatically generates the image based on the current rendering of the specified element.
- ***effectAllowed*** — possible values for this property are *none, copy, copyLink. copyMove, link, linkMove, move,* and *all*. Only the specified operations are allowed.

- ***dropEffect*** — possible values for this property are *none, copy, link,* and *move*. Specifies the type of current operation.

Data Transfer Example

The previous example is modified to make use of the *data* transfer object. The HTML file is shown below.

ex05_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex05_dragAndDropSample.js"></script>

        <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>
        <p/>
        <div id="target">

        </div>

         <p id="msg">Drop over the target</p>

    </body>
</html>
```

The associated *JavaScript* code is shown below. The code handles the drag source events and the drop target events. The code below registers the event handlers for *ondragstart*, *ondrag,* and *ondragend* on the drag source, and the *ondragenter, ondragover* and *ondrop* on the drop target.

ex05_dragAndDropSample.js

```
1  window.onload = init;
2
3  var src, target, msg;
4
5  function init() {
6      src = document.getElementById("src");
7      target = document.getElementById("target");
8      msg = document.getElementById("msg");
9
10     // Add event handlers for the source
11     src.ondragstart = dragStartHandler;
12     src.ondragend = dragEndHandler;
13     src.ondrag = dragHandler;
14
15     // Add event handlers for the target
16     target.ondragenter = dragEnterHandler;
17     target.ondragover = dragOverHandler;
18     target.ondrop = dropHandler;
19
20 }
```

The *ondragstart* event handler is now modified to make use of the *dataTransfer* property of the drag event. The *id* of the drag source is set as the data transfer item and is associated with the *Text* (plain/text) MIME type.

ex05_dragAndDropSample.js

```
21
22  function dragStartHandler(e) {
23      e.dataTransfer.setData("Text", e.target.id);
24      e.target.classList.add("dragged");
25  }
26
27  function dragEndHandler(e) {
28      msg.innerHTML = "Drop over the target";
29      var elems = document.querySelectorAll(".dragged");
30      for(var i = 0; i < elems.length; i++) {
31          elems[i].classList.remove("dragged");
32      }
33  }
34
35  function dragHandler(e) {
36      msg.innerHTML = "Dragging " + e.target.id;
37  }
```

Similarly, the *ondrop* event handler is modified to retrieve the *id* of the element being dragged. The MIME datatype *Text* (plain/text) is used with the *getData* method of the *dataTransfer* object to retrieve of *id* of the element set by the drag start handler.

ex05_dragAndDropSample.js

```
38
39  function dragEnterHandler(e) {
40      msg.innerHTML = "Drag Entering " + e.target.id;
41      e.preventDefault();
42  }
43
44  function dragOverHandler(e) {
45      msg.innerHTML = "Drag Over " + e.target.id;
46      e.preventDefault();
47  }
48
49  function dropHandler(e) {
50      var sourceId = e.dataTransfer.getData("Text");
51      var sourceElement = document.getElementById(sourceId);
52      var newElement = sourceElement.cloneNode(false);
53      target.innerHTML = "";
54      target.appendChild(newElement);
55      e.preventDefault();
56  }
```

Filtering

The previous example is modified to make use of the *data* transfer object and filter what the source elements that can be dropped on the target. The HTML file is shown below.

ex06_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex06_dragAndDropSample.js"></script>

         <div id="src">
            <img draggable="true" id="ltc" src="f1.png"
                alt="Prof. Chitkushev"/>
            <img draggable="true" id="temkin" src="f2.png"
                alt="Prof. Temkin"/>
            <img draggable="true" id="kanabar" src="f3.png"
                alt="Prof. Kanabar"/>
            <img draggable="true" id="kalathur" src="f4.png"
                alt="Prof. Kalathur"/>
        </div>
        <p/>
        <div id="target">

        </div>

         <p id="msg">Drop over the target</p>

    </body>
</html>
```

The associated *JavaScript* code is shown below. The code handles the drag source events and the drop target events. The code below registers the event handlers for *ondragstart*, *ondrag,* and *ondragend* on the drag source, and the *ondragenter, ondragover* and *ondrop* on the drop target.

ex06_dragAndDropSample.js

```
1  window.onload = init;
2
3  var src, target, msg;
4  var sourceId;
5
6  function init() {
7      src = document.getElementById("src");
8      target = document.getElementById("target");
9      msg = document.getElementById("msg");
10
11     // Add event handlers for the source
12     src.ondragstart = dragStartHandler;
13     src.ondragend = dragEndHandler;
14     src.ondrag = dragHandler;
15
16     // Add event handlers for the target
17     target.ondragenter = dragEnterHandler;
18     target.ondragover = dragOverHandler;
19     target.ondrop = dropHandler;
20 }
```

The *ondragstart* event handler is now modified to make use of the *dataTransfer* property of the drag event. The *id* of the drag source is set as the data transfer item and is associated with the *Text* (plain/text) MIME type. The *sourceId* is also explicitly set to the same value as the filtering doesn't work in some of the browsers.

ex06_dragAndDropSample.js

```
22  function dragStartHandler(e) {
23      e.dataTransfer.setData("Text", e.target.id);
24      sourceId = e.target.id;     // explicitly for some browsers
25      e.target.classList.add("dragged");
26  }
27
28  function dragEndHandler(e) {
29      msg.innerHTML = "Drag ended";
30      var elems = document.querySelectorAll(".dragged");
31      for(var i = 0; i < elems.length; i++) {
32          elems[i].classList.remove("dragged");
33      }
34  }
35
36  function dragHandler(e) {
37      msg.innerHTML = "Dragging " + e.target.id;
38  }
```

The *ondragenter* and *ondragover* event handlers are modified to check if the element being dragged is a valid choice. The *dataTransfer* object is not available in some browsers for these two events, hence, the explicit need for the *id* of the source.

ex06_dragAndDropSample.js

```
function dragEnterHandler(e) {
    console.log("Drag Entering " + e.target.id +
            " source is " + e.dataTransfer.getData("Text") );

    var id = e.dataTransfer.getData("text") || sourceId;
    if (id == "kalathur") {
        e.preventDefault();
    }
}

function dragOverHandler(e) {
    console.log("Drag Over " + e.target.id +
             " source is " + e.dataTransfer.getData("Text")) ;

    var id = e.dataTransfer.getData("text") || sourceId;
    if (id == "kalathur") {
        e.preventDefault();
    }
}

function dropHandler(e) {
    console.log("Drop on " + e.target.id +
             " source is " + e.dataTransfer.getData("Text")) ;

    var id = e.dataTransfer.getData("text") || sourceId;
    var sourceElement = document.getElementById(id);
    var newElement = sourceElement.cloneNode(false);
    target.innerHTML = "";
    target.appendChild(newElement);
    e.preventDefault();
}
```

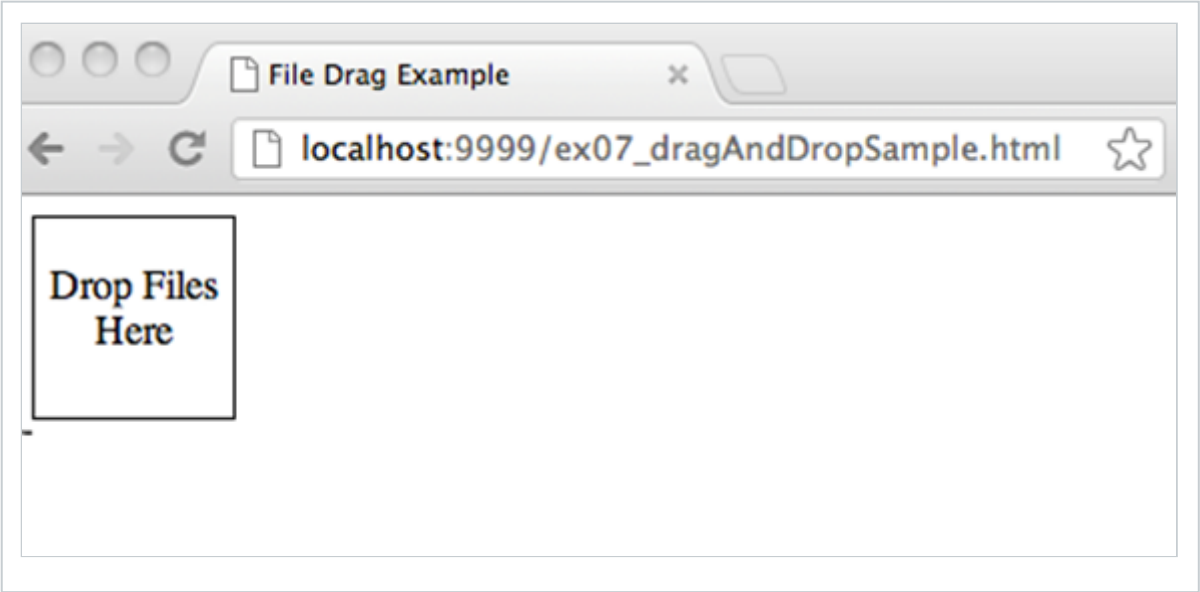
Dragging Files

The *files* property of the *data transfer* object returns the list of the files being dropped onto the target, if any. The following example demonstrates the drag and drop of files from the desktop to the web application.

ex07_dragAndDropSample.html

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>File Drag Example</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <script src="ex07_dragAndDropSample.js"></script>

        <div id="target">
            <p id="msg">Drop Files Here</p>
        </div>

        <table id="fileData" border="1">
        </table>

    </body>
</html>
```

The above HTML is rendered as shown below.



The associated *JavaScript* code is shown below. The code handles only the drop target events and registers the event handlers for *ondragenter, ondragover* and *ondrop* on the drop target.

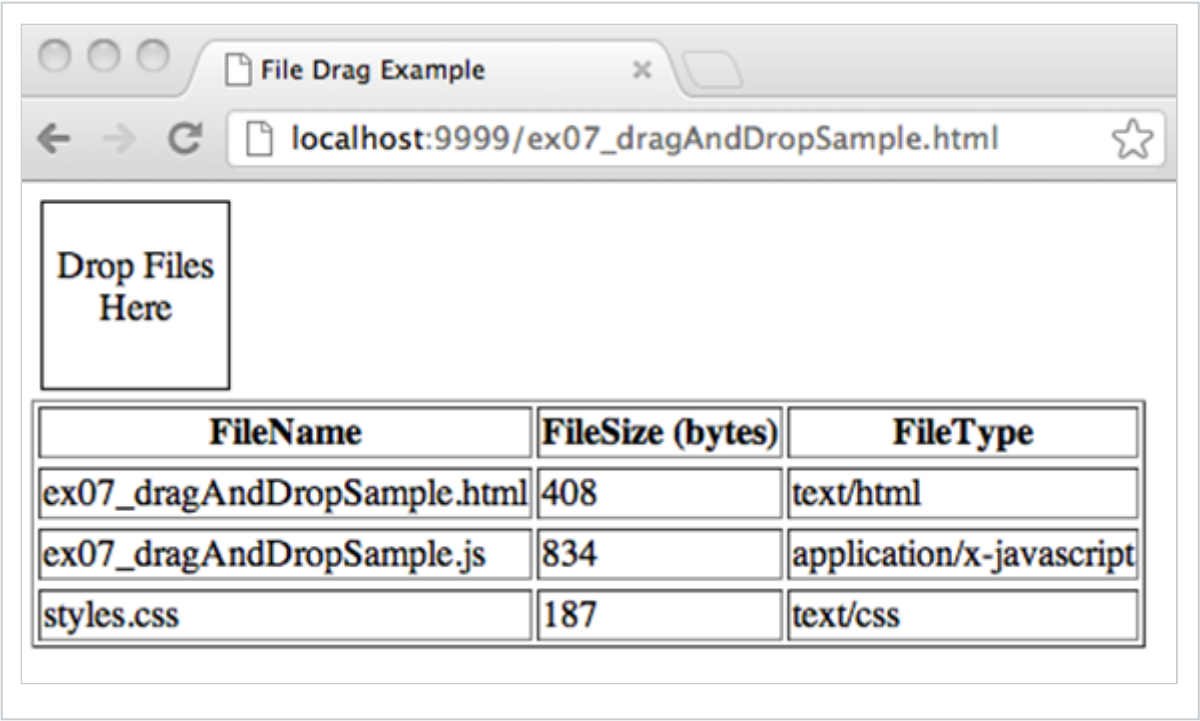
ex07_dragAndDropSample.js

```
1  window.onload = init;
2
3  var target;
4
5  function init() {
6      target = document.getElementById("target");
7
8      // Add event handlers
9      target.ondragenter = handleDrag;
10     target.ondragover = handleDrag;
11     target.ondrop = handleDrop;
12 }
13
14 function handleDrag(e) {
15     e.preventDefault();
16 }
17
18 function handleDrop(e) {
19     var fileList = e.dataTransfer.files;
20     var tableHTMLElem = document.getElementById("fileData");
21     tableHTMLElem.innerHTML =
22      "<tr><th>FileName</th><th>FileSize (bytes)</th><th>FileType</th></tr>";
23
24     for(var i = 0; i < fileList.length; i++) {
25         var tableRow = "<tr><td>" +
26             fileList[i].name + "</td><td>" +
27             fileList[i].size + "</td><td>" +
28             fileList[i].type + "</td></tr>";
29         tableHTMLElem.innerHTML += tableRow;
30     }
31     e.preventDefault();
32 }
```

In the *ondrop* event handler, the *files* property of the *dataTransfer* object gives access to the properties of the files being dropped. The file items have the following properties:

- ***name —*** the name of the file
- ***size*** — the size of the file (in bytes)
- ***type*** — the MIME type of the file
- ***lastModifiedDate*** — last modified date of the file.

The following figure shows the result after dropping the example's files onto the drop target.



Geolocation

Introduction

Many web applications rely on the current location of the user to provide location centric services. HTML5 now includes the *Geolocation* JavaScript API that allows the web application to request the current location through the browser. The browser alerts the user when such an action occurs whether permission is allowed or not. If the user allows permission, the API calls retrieve the current location. The location is specified in terms of the *latitude* and *longitude*.

> The Record Latitude and Longitude Coordinates interactive website shows the location values for any location on the world map.

How it Works

The *Geolocation* API provides the methods, which enable the web application to acquire the current location. The API provides the capability to acquire the location once (*getCurrentPosition*), or repeatedly track the client's position (*watchPosition*). These are asynchronous calls. The callback function specified will be invoked when the location is available. If there is an error during the process, the optional error handler may be specified to handle those cases. The *navigator.geolocation* property provides the *Geolocation* interface if it is implemented for the browser. The API as specified in the HTML5 specification [2] is shown below.

```
[NoInterfaceObject]
interface Geolocation {
  void getCurrentPosition(PositionCallback successCallback,
                          optional PositionErrorCallback errorCallback,
                          optional PositionOptions options);

  long watchPosition(PositionCallback successCallback,
                     optional PositionErrorCallback errorCallback,
                     optional PositionOptions options);

  void clearWatch(long watchId);
};

callback PositionCallback = void (Position position);

callback PositionErrorCallback = void (PositionError positionError);
```

When the location information is available, the callback function's argument provides the details about the *Position*. The *Position* interface has the *coords* and *timestamp* property as shown below.

```
[NoInterfaceObject]
interface Position {
  readonly attribute Coordinates coords;
  readonly attribute DOMTimeStamp timestamp;
};
```

The *Coordinates* interface provides the location details. The *latitude*, *longitude,* and *accuracy* properties are supported by all implementations. The rest of the properties are implementation dependent.

```
[NoInterfaceObject]
interface Coordinates {
  readonly attribute double latitude;
  readonly attribute double longitude;
  readonly attribute double? altitude;
  readonly attribute double accuracy;
  readonly attribute double? altitudeAccuracy;
  readonly attribute double? heading;
  readonly attribute double? speed;
};
```

The *latitude* and *longitude* attributes are specified in decimal degrees while the *accuracy* is specified in meters.

Browser Support

Browser support for *Geolocation* capability can be checked programmatically using *navigator.geolocation* object as shown below.

ex01_checkGeolocationSupport.html

```
<!DOCTYPE html>
<html>
    <body>
        <script type="text/javascript">
            if (navigator.geolocation)
                document.write("Geolocation support is available.")
            else
                document.write("Geolocation support is not available.")
        </script>
    </body>
</html>
```

Getting Current Position

The *getCurrentPosition* method of the *Geolocation* object is used to obtain the current location of the client. The functionality is illustrated through the following example that provides the capability for requesting the current location. The HTML file has the placeholders for displaying the *latitude*, *longitude*, *accuracy*, and *timestamp* of the location data.

ex02_geoLocationSample.html

```
<!doctype html>
<html>
    <head>
        <title>Geolocation Sample</title>
        <meta charset="utf-8">
        <script src="ex02_geoLocationSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
        <header>
            <h2>Geolocation Example</h2>
        </header>
        <section>
            <article>
                <button id="checkButton">Get Location</button>
                <p/>
                <div id="status">
                    <p id="latitude">Latitude:</p>
                    <p id="longitude">Longitude:</p>
                    <p id="accuracy">Accuracy:</p>
                    <p id="timestamp">Timestamp:</p>
                </div>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handler for the button that the user clicks for requesting the current location.

ex02_geoLocationSample.js

```
window.onload = init;

// register the event handler for button

function init() {
    var checkButton = document.getElementById("checkButton");
    checkButton.onclick = getLocation;
}
```

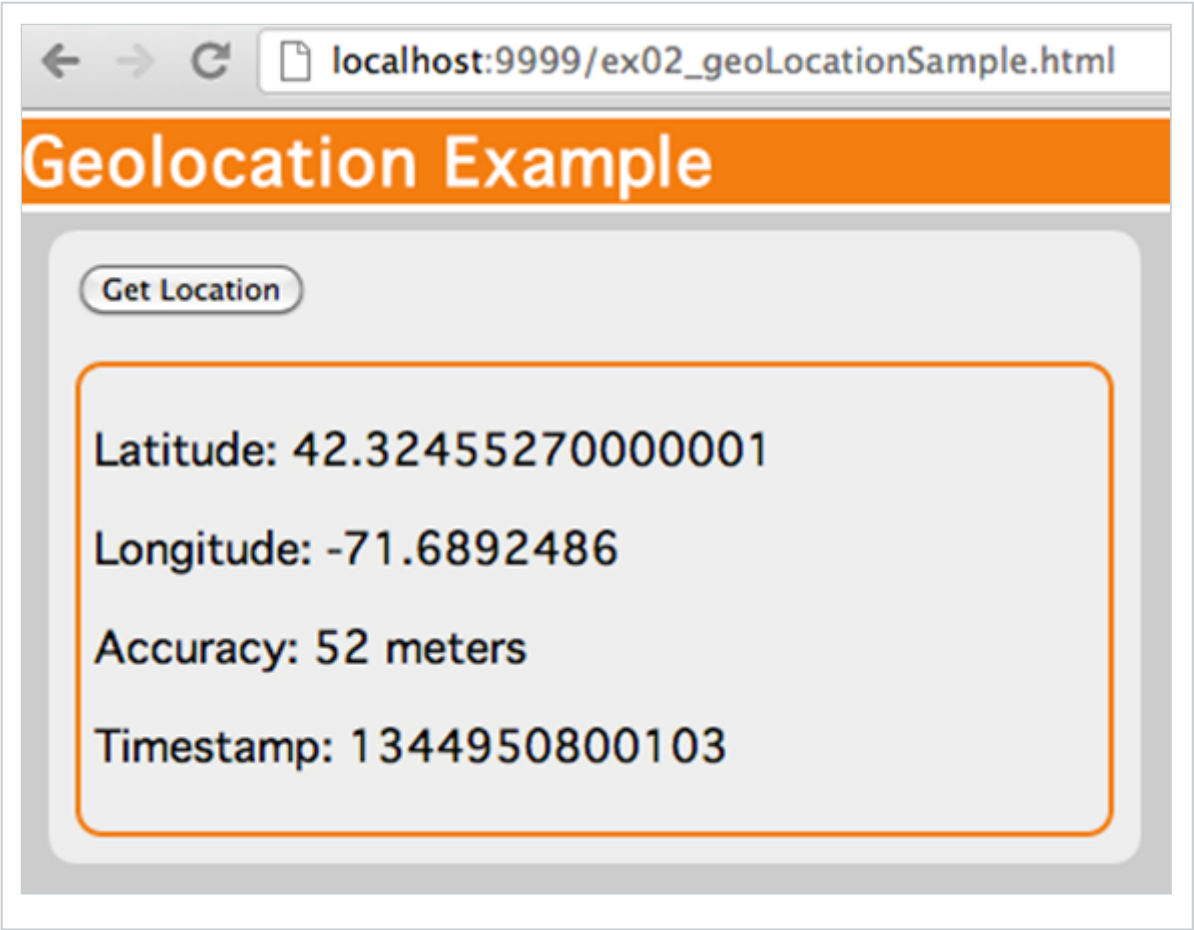
The *getLocation* event handler invokes the *getCurrentPosition* method of the *Geolocation* object accessed through *navigator.geolocation* property. The asynchronous call immediately returns. When the location details are available,

the specified callback function *displayLocation* will be invoked. The location details are accessed through the *position* argument object passed to this function. The *latitude, longitude,* and *accuracy* details of the current location are accessed through the *coords* property of the *position* argument. The *timestamp* property provides the time when the location data was obtained. The values are populated in the HTML document as shown below.

ex02_geoLocationSample.js

```
10 function getLocation() {
11
12     // asynchronous call with callback function specified
13     navigator.geolocation.getCurrentPosition(displayLocation);
14
15 }
16
17 // callback function on success
18 function displayLocation(position) {
19
20     var latitude = position.coords.latitude;
21     var longitude = position.coords.longitude;
22     var accuracy = position.coords.accuracy;
23     var timestamp = position.timestamp;
24
25     document.getElementById("latitude").innerHTML =
26             "Latitude: " + latitude;
27     document.getElementById("longitude").innerHTML =
28             "Longitude: " + longitude;
29     document.getElementById("accuracy").innerHTML =
30             "Accuracy: " + accuracy + " meters";
31     document.getElementById("timestamp").innerHTML =
32             "Timestamp: " + timestamp;
33 }
```

The above example is rendered as shown below. Clicking on the button retrieves the current location information and is shown in the web page.



Error Handling

The above example illustrated the scenario when the location information was returned successfully. However, the following error scenarios may arise:

- The user denies permission to the browser for accessing the location information
- The location provider cannot determine the position of the device
- A timeout occurs before the location information is available

For the application to handle the above error conditions, an error callback handler may optionally be specified as the second argument of the *getCurrentPosition* method. The above scenarios are illustrated through the following example. The HTML file has the placeholders for displaying the *latitude*, *longitude*, *accuracy*, and *timestamp* of the location data. If there is an error, the error message is displayed instead.

ex03_geoLocationSample.html

```
1  <!doctype html>
2  <html>
3      <head>
4          <title>Geolocation Sample</title>
5          <meta charset="utf-8">
6          <script src="ex03_geoLocationSample.js"></script>
7          <link rel="stylesheet" href="html5.css">
8      </head>
9      <body>
10         <header>
11             <h2>Geolocation Example</h2>
12         </header>
13         <section>
14             <article>
15                 <button id="checkButton">Get Location</button>
16                 <p/>
17                 <div id="status">
18                     <p id="latitude">Latitude:</p>
19                     <p id="longitude">Longitude:</p>
20                     <p id="accuracy">Accuracy:</p>
21                     <p id="timestamp">Timestamp:</p>
22                 </div>
23             </article>
24         </section>
25     </body>
26 </html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handler for the button that the user clicks for requesting the current location. The event handler invokes the *getCurrentPosition* method with two arguments. When the location details are available, the specified callback function *displayLocation* will be invoked. When there is an error, the error callback function *handleError* is invoked as shown below.

ex03_geoLocationSample.js

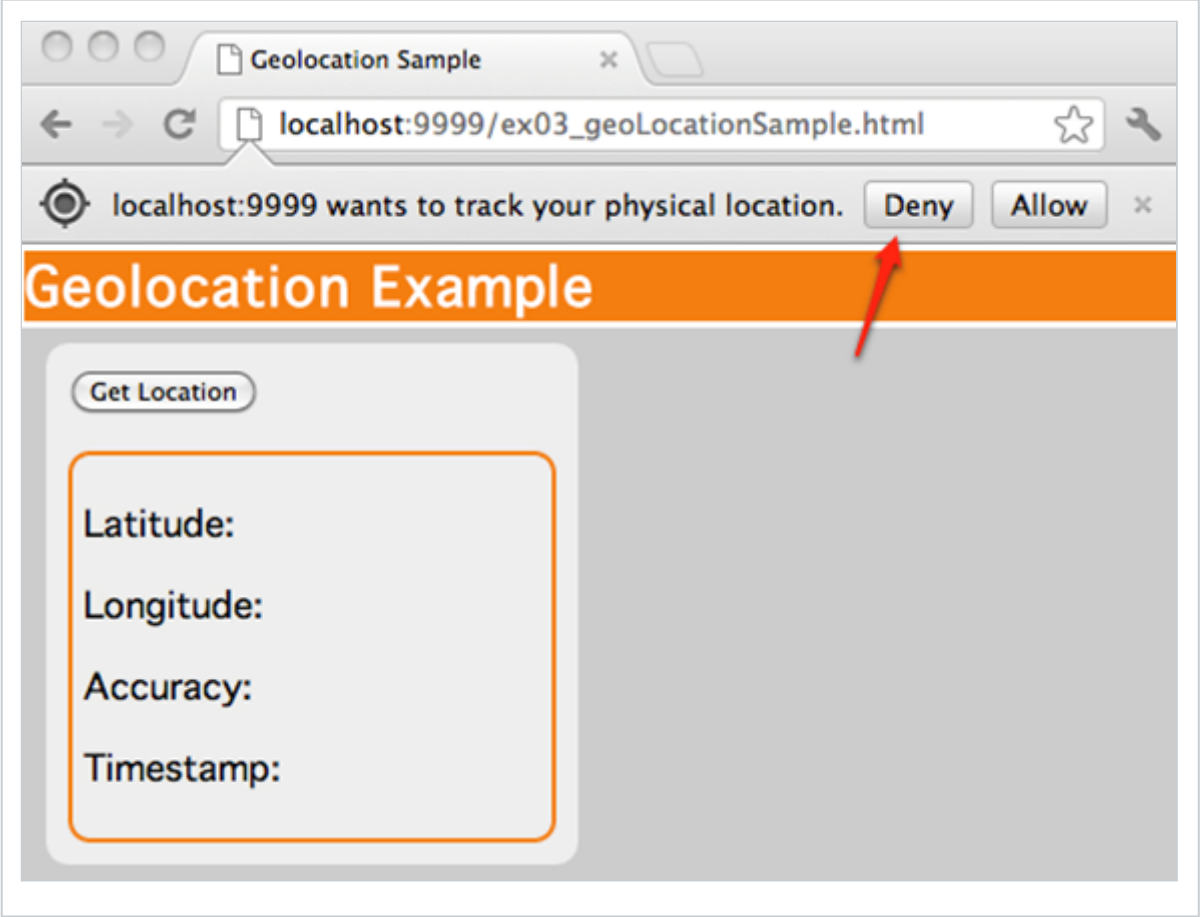
```
1  window.onload = init;
2
3  // register the event handler for button
4
5  function init() {
6      var checkButton = document.getElementById("checkButton");
7      checkButton.onclick = getLocation;
8  }
9
10 function getLocation() {
11     // asynchronous call with callback  success
12     // and error functions specified
13
14     navigator.geolocation.getCurrentPosition(
15         displayLocation, handleError);
16 }
```

The success callback handler and the error callback handler are shown below. If the location information is obtained successfully, the details are shown in the web application. If there is an error, the error code is accessed through the error handler's argument. The error codes 1, 2, and 3 correspond to the error scenarios described above.

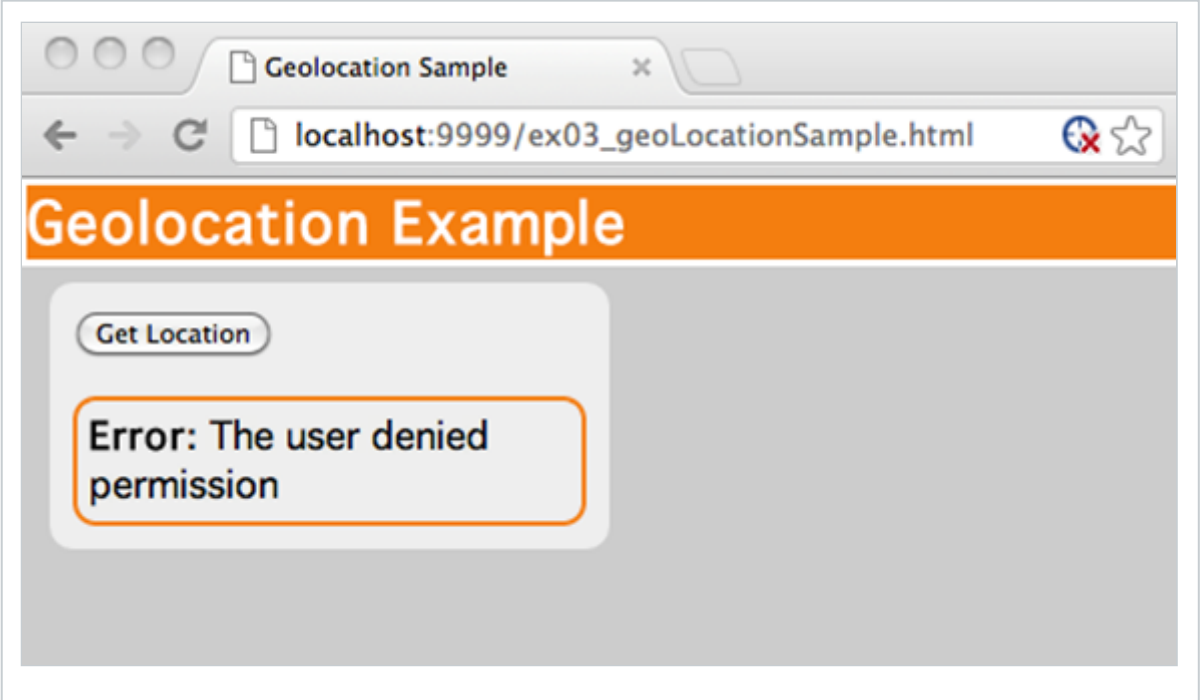
ex03_geoLocationSample.js

```
// callback function on success
function displayLocation(position) {

    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var accuracy = position.coords.accuracy;
    var timestamp = position.timestamp;

    document.getElementById("latitude").innerHTML =
            "Latitude: " + latitude;
    document.getElementById("longitude").innerHTML =
            "Longitude: " + longitude;
    document.getElementById("accuracy").innerHTML =
            "Accuracy: " + accuracy + " meters";
    document.getElementById("timestamp").innerHTML =
            "Timestamp: " + timestamp;
}

// callback function on error
function handleError(error) {
    switch(error.code) {
        case 1:
            updateStatus("The user denied permission");
            break;
        case 2:
            updateStatus("Position is unavailable");
            break;
        case 3:
            updateStatus("Timed out");
            break;
    }
}

function updateStatus(message) {
    document.getElementById("status").innerHTML =
        "<strong>Error</strong>: " + message;
}
```

The above example is rendered as shown below. Clicking on the button retrieves the current location information and shown in the web page.



When the user denies permission for the location, the error condition is handled as shown below.



In the following scenario, disconnecting the Internet connectivity simulates the unavailability of the position information.

The *timeout* scenario is illustrated in a subsequent example when a timeout option is specified.

Position Options

The *getCurrentPosition* method takes an optional third argument that controls the behavior of the call. The *PositionOptions* interface provides the following three optional properties—*enableHighAccuracy* (default value is *false*), *timeout* (default value is *infinity*), and *maximumAge* (default value is 0).

```
[Callback, NoInterfaceObject]
interface PositionOptions {
  attribute boolean enableHighAccuracy;
  attribute long timeout;
  attribute long maximumAge;
};
```

Setting the *enableHighAccuracy* option to *true* enables the application to get the best possible accuracy. This may result in a slower response time, or an increase in the power consumption of the device. The *timeout* option specifies the maximum time (in milliseconds) allowed for the results once the *getCurrentPosition* method is invoked. If the result is not available, the error handler is invoked with the error code set to the timeout case. The *maximumAge* option specifies that a cached location may be used if the time elapsed since it is acquired is within the specified time (in milliseconds). By default, a new position is acquired for each call.

The above options are illustrated with the following example.

ex04_geoLocationSample.html

```
<!doctype html>
<html>
    <head>
        <title>Geolocation Sample</title>
        <meta charset="utf-8">
        <script src="ex04_geoLocationSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
        <header>
            <h2>Geolocation Example</h2>
        </header>
        <section>
            <article>
                <button id="checkButton">Get Location</button>
                <p/>
                <div id="status">
                    <p id="latitude">Latitude:</p>
                    <p id="longitude">Longitude:</p>
                    <p id="accuracy">Accuracy:</p>
                    <p id="timestamp">Timestamp:</p>
                </div>
            </article>
        </section>
    </body>
</html>
```

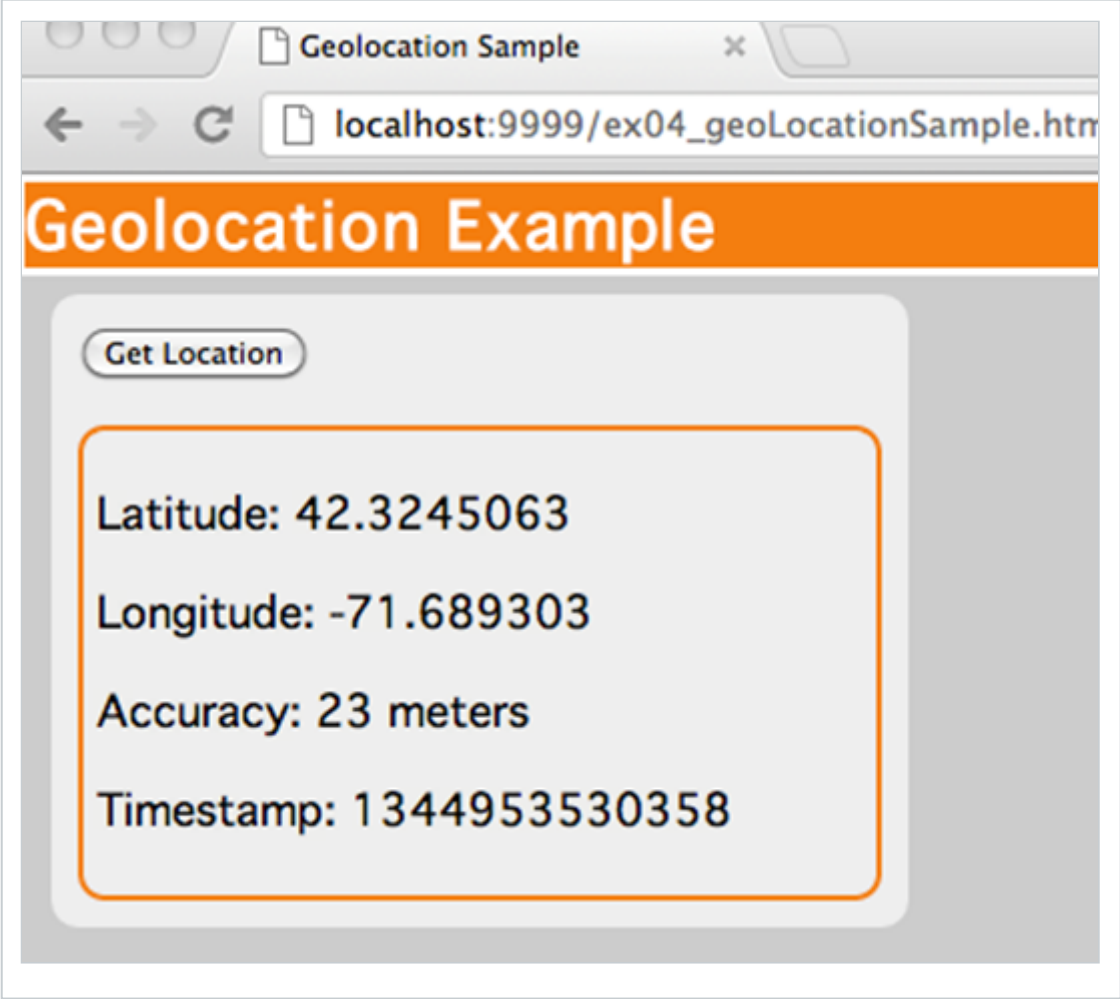
A portion of the relevant *JavaScript* code is shown below. The following snippet adds the event handler for the button that the user clicks for requesting the current location.

The *getLocation* event handler invokes the *getCurrentPosition* method with the three arguments – the success handler, the error handler, and the position options. The example shows the options *enableHighAccurary* (true), *timeout* (1000), and *maximumAge* (0).

ex04_geoLocationSample.js

```
1  window.onload = init;
2
3  // register the event handler for button
4
5  function init() {
6      var checkButton = document.getElementById("checkButton");
7      checkButton.onclick = getLocation;
8  }
9
10 function getLocation() {
11     // asynchronous call with callback success,
12     // error functions and options specified
13
14     var options = { enableHighAccuracy: true,
15                     timeout: 1000,
16                     maximumAge: 0
17                   };
18
19     navigator.geolocation.getCurrentPosition(
20         displayLocation, handleError, options);
21 }
```

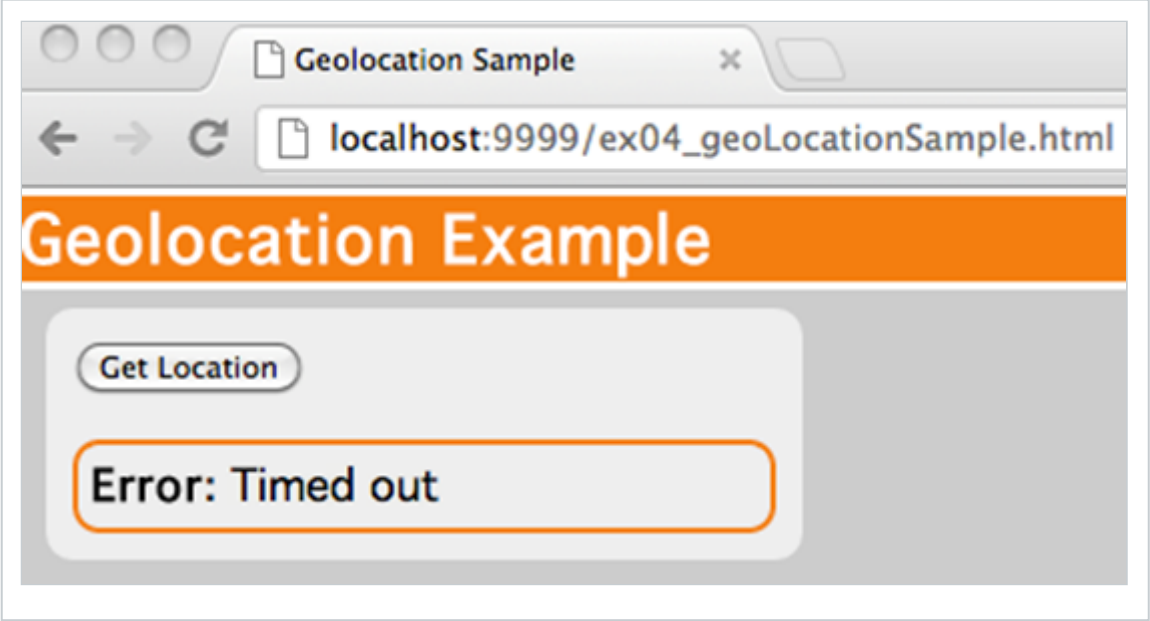
The above example is rendered as shown below. Clicking on the button retrieves the current location information and shown in the web page.



To demonstrate the *timeout* scenario, set the value to, say 100, as shown below.

```
var options = { enableHighAccuracy: true,
                timeout: 100,
                maximumAge: 0
              };
```

Clicking on the button on the web page to retrieve the current location information results in the error condition as shown below.



Tracking the Position

The *getCurrentPosition* method used in the previous examples retrieves the location information only once. On the other hand, the *watchPostion* method provides the functionality to continuously track the location of the client as its position changes. The *watchPosition* method takes the same type of arguments as the *getCurrentPosition* method and asynchronously starts the tracking operation. However, the *watchPosition* method returns a *watchId* value that can used to cancel the tracking through the *clearWatch* method.

The tracking functionality is illustrated with the following example. The HTML file of the sample is shown below.

ex05_geoLocationSample.html

```
<!doctype html>
<html>
    <head>
        <title>Geolocation Sample</title>
        <meta charset="utf-8">
        <script src="ex05_geoLocationSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
        <header>
            <h2>Geolocation Example</h2>
        </header>
        <section>
            <article>
                <button id="startButton">Start</button>
                <button id="stopButton">Stop</button>
                <p/>
                <div id="status">
                    <p id="counter">Update#:</p>
                    <p id="latitude">Latitude:</p>
                    <p id="longitude">Longitude:</p>
                    <p id="accuracy">Accuracy:</p>
                    <p id="timestamp">Timestamp:</p>
                </div>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons that the user clicks for starting and stopping the tracking of the current location.

ex05_geoLocationSample.js

```
1  window.onload = init;
2
3  var watchId = null;
4  var counter = 0;
5
6  // register the event handlers for the buttons
7
8  function init() {
9      var startButton = document.getElementById("startButton");
10     var stopButton = document.getElementById("stopButton");
11
12     startButton.onclick = startTrackingLocation;
13     stopButton.onclick = stopTrackingLocation;
14
15 }
```

The *startTrackingLocation* event handler invokes the *watchPosition* method of the *Geolocation* object accessed through *navigator.geolocation* property. The call immediately returns with the *watchId*. When the location details are available, the specified callback function *displayLocation* will be invoked. Similarly, the callback function is repeatedly invoked as the location of the client changes. The *stopTrackingLocation* event handler invokes the *clearWatch* method with the associated *watchId*.

ex05_geoLocationSample.js

```
17  function startTrackingLocation() {
18
19      // asynchronous call with callback success,
20      // error functions and options specified
21
22      var options = {
23          enableHighAccuracy : true,
24          timeout : 5000,
25          maximumAge : 0
26      };
27
28      // start the position tracking
29      watchId = navigator.geolocation.watchPosition(
30              displayLocation, handleError, options);
31  }
32
33  function stopTrackingLocation() {
34
35      if (watchId) {
36          // stop the position tracking
37          navigator.geolocation.clearWatch(watchId);
38      }
39  }
```

Whenever the location details are available, the *displayLocation* function is invoked which updates the web application with the location details.

ex05_geoLocationSample.js

```
41  // called repeatedly for updating the position
42
43  function displayLocation(position) {
44
45      var latitude = position.coords.latitude;
46      var longitude = position.coords.longitude;
47      var accuracy = position.coords.accuracy;
48      var timestamp = position.timestamp;
49
50      counter++;
51
52      document.getElementById("counter").innerHTML =
53              "Update#: " + counter;
54      document.getElementById("latitude").innerHTML =
55              "Latitude: " + latitude;
56      document.getElementById("longitude").innerHTML =
57              "Longitude: " + longitude;
58      document.getElementById("accuracy").innerHTML =
59              "Accuracy: " + accuracy + " meters";
60      document.getElementById("timestamp").innerHTML =
61              "Timestamp: " + timestamp;
62  }
```

Integrating with Google Map API

The Geolocation functionality can easily be integrated with the Google Maps JavaScript API. The Google Maps API allows the mapping of the location coordinates of latitude and longitude. To make use of the Google Maps API, the associated JavaScript is included in the web application.

> Check out the API documentation Google Maps JavaScript API V3 Reference .

The above mapping functionality is illustrated through the following example. The HTML file has the placeholders for displaying the map location in addition to the *latitude*, *longitude*, *accuracy*, and *timestamp* of the location data.

ex06_geoLocationSample.html

```
<!doctype html>
<html>
    <head>
        <title>Geolocation Sample</title>
        <meta charset="utf-8">
        <script src="http://maps.google.com/maps/api/js?sensor=false"></script>
        <script src="ex06_geoLocationSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
        <header>
            <h2>Geolocation Example</h2>
        </header>
        <section>
            <article>
                <button id="checkButton">Get Location</button>
                <p/>
                <div id="status">
                    <p id="latitude">Latitude:</p>
                    <p id="longitude">Longitude:</p>
                    <p id="accuracy">Accuracy:</p>
                    <p id="timestamp">Timestamp:</p>
                </div>
                <p></p>
                <div id="map">
                </div>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handler for the button that the user clicks for requesting the current location.

ex06_geoLocationSample.js

```
1  window.onload = init;
2
3  // register the event handler for the button
4
5  function init() {
6      var checkButton = document.getElementById("checkButton");
7      checkButton.onclick = getLocation;
8  }
9
10 function getLocation() {
11     // asynchronous call with callback success,
12     // error functions and options specified
13
14     var options = {
15         enableHighAccuracy : true,
16         timeout : 5000,
17         maximumAge : 0
18     };
19
20     navigator.geolocation.getCurrentPosition(
21         displayLocation, handleError, options);
22 }
```

When the location information is available, the callback function *displayLocation* displays the location details and delegates the mapping functionality to the *showOnMap* function.

ex06_geoLocationSample.js

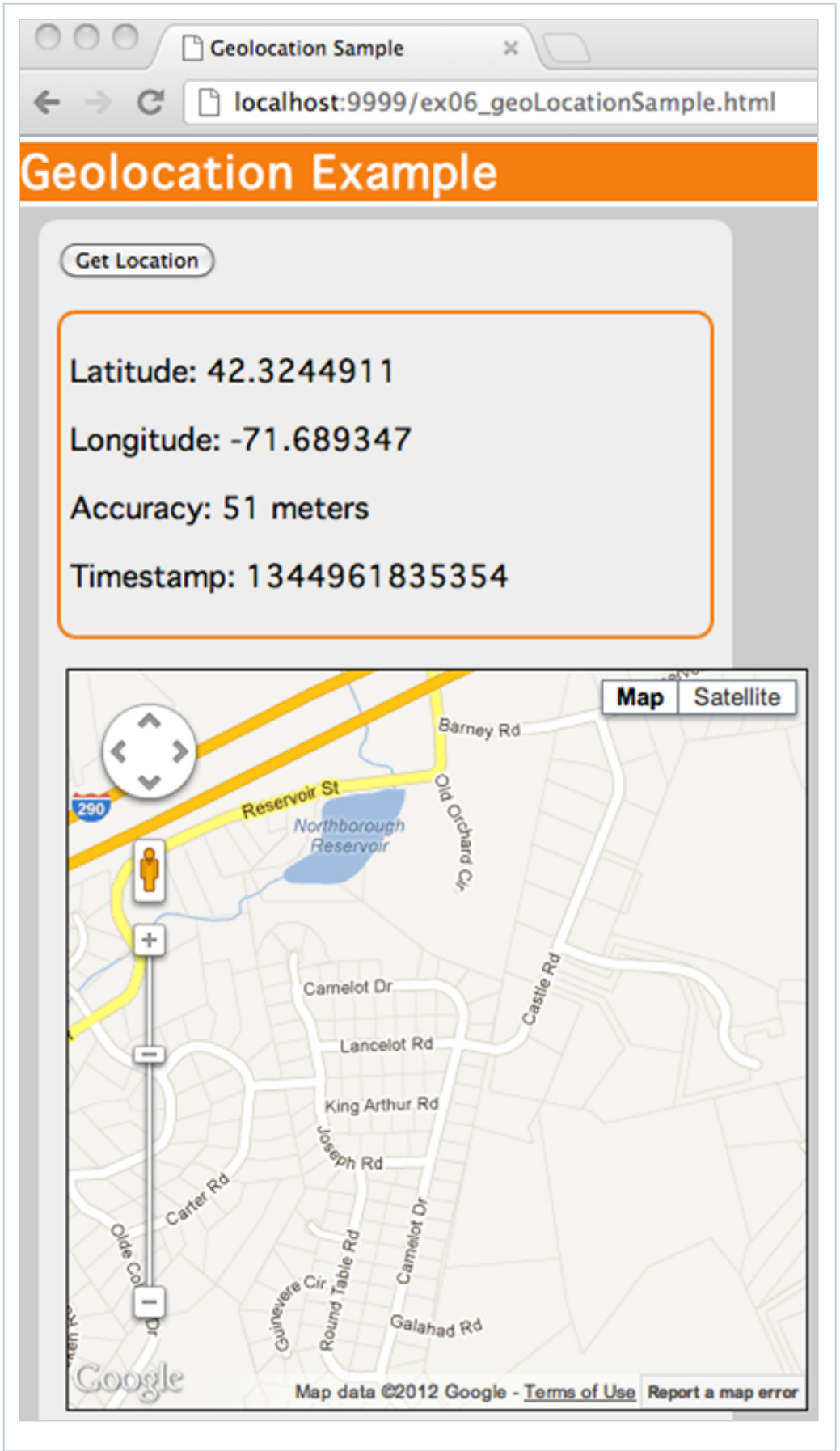
```
24 function displayLocation(position) {
25
26     var latitude = position.coords.latitude;
27     var longitude = position.coords.longitude;
28     var accuracy = position.coords.accuracy;
29     var timestamp = position.timestamp;
30
31     document.getElementById("latitude").innerHTML =
32             "Latitude: " + latitude;
33     document.getElementById("longitude").innerHTML =
34             "Longitude: " + longitude;
35     document.getElementById("accuracy").innerHTML =
36             "Accuracy: " + accuracy + " meters";
37     document.getElementById("timestamp").innerHTML =
38             "Timestamp: " + timestamp;
39
40     // Show the google map with the position
41     showOnMap(position.coords);
42 }
```

The *showOnMap* function shown below maps the latitude and longitude data and creates the *Map* object to be displayed on the web page.

ex06_geoLocationSample.js

```
64 // initialize the map and show the position
65 function showOnMap(pos) {
66
67     var googlePosition =
68         new google.maps.LatLng(pos.latitude, pos.longitude);
69
70     var mapOptions = {
71         zoom: 15,
72         center: googlePosition,
73         mapTypeId: google.maps.MapTypeId.ROADMAP
74     };
75
76     var mapElement = document.getElementById("map");
77     map = new google.maps.Map(mapElement, mapOptions);
78 }
```

The above example is rendered as shown below. Clicking on the button retrieves the current location information and is shown in the web page along with the map. The map is centered on the current location.



Adding Marker on the Map

In the previous example, the map was displayed and centered on the current location. However, there is no visible marker to identify the current location on the map. The above example is modified to add the *Marker* object at the current location along with the options to be displayed when the user clicks on the marker.

ex07_geoLocationSample.js

```
64  // initialize the map and show the position
65  function showOnMap(pos) {
66
67      var googlePosition =
68          new google.maps.LatLng(pos.latitude, pos.longitude);
69
70      var mapOptions = {
71          zoom: 15,
72          center: googlePosition,
73          mapTypeId: google.maps.MapTypeId.ROADMAP
74      };
75
76      var mapElement = document.getElementById("map");
77      map = new google.maps.Map(mapElement, mapOptions);
78
79      // add the marker to the map
80      var title = "Location Details";
81      var content = "Lat: " + pos.latitude +
82                    ", Long: " + pos.longitude;
83
84      addMarker(map, googlePosition, title, content);
85  }
```

The *JavaScript* code to add the *Marker* object is shown below. The *clickable* option is set to true. When the user clicks on the marker, a window pops up with the specified *title* and *content*.

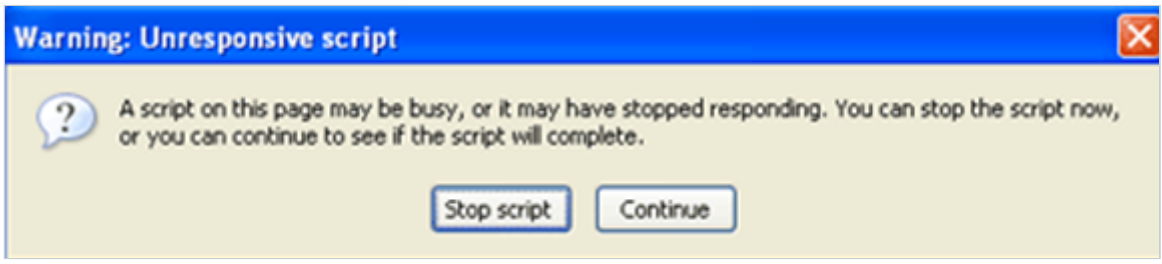
ex07_geoLocationSample.js

```
87  // add position marker to the map
88  function addMarker(map, latlongPosition, title, content) {
89
90      var options = {
91          position: latlongPosition,
92          map: map,
93          title: title,
94          clickable: true
95      };
96      var marker = new google.maps.Marker(options);
97
98      var popupWindowOptions = {
99          content: content,
100         position: latlongPosition
101     };
102
103     var popupWindow = new google.maps.InfoWindow(popupWindowOptions);
104
105     google.maps.event.addListener(marker, 'click', function() {
106         popupWindow.open(map);
107     });
108 }
```

Web Workers

Introduction

JavaScript is a single-threaded execution model. As a result, multiple scripts cannot run at the same time. Also, if the current script is a computationally intensive one, user will not be able interact with the web application. This usually results in the browser popping up an unresponsive script message as shown below.



To overcome these limitations, the *Web Workers* API provides the capability to create background scripts in the web application. This allows the computationally intensive scripts to be run as separate threads in the background without blocking the user interaction with the application or with other scripts.

How it Works

Whenever there is a need for some computation that should not interfere the UI of the web application, a new Web Worker is spawned by the main script of the web application. An event listener listens for messages from the Web Worker and acts on the data as they are received. Similarly, the main script communicates with the worker by sending the input messages to be worked upon. The main script can spawn any number of Web Workers as desired. Similarly, a Web Worker can spawn other Web Workers if the task needs to be broken down into simpler tasks.

There are two types of web workers—dedicated workers and shared workers. The Web Workers described above are dedicated workers. The web page that creates the Web Worker exclusively communicates between them to accomplish the required task. Shared Web Workers, on the other hand, are shared by multiple web pages originating from the same origin. Each page will have their own unique port object through which they communicate with the shared worker.

The JavaScript code executed by the Web Worker has limitations on what it can access. The worker code cannot access the *window*/DOM and hence will not be able to read or modify the main HTML document. Global variables and JavaScript functions defined in the main page are also not accessible.

Web Workers run in an isolated thread and hence the JavaScript code the Web Worker runs has to be in a separate file by itself. However, the worker code can also be specified inline and this approach is only used for quick development/testing.

Browser Support

Checking the *Worker* property of the global *window* object helps us determine whether the browser support for Web Workers is available or not. If the Web Worker API is not supported, the *Worker* property will be *undefined*. The following sample shows if the support exists or not.

ex01_checkOfflineSupport.html

```
<!DOCTYPE html>
<html>
    <body>
        <script type="text/javascript">
            if (typeof(Worker) != "undefined")
                document.write("Web Worker support is available.")
            else
                document.write("Web Worker support is not available.")
        </script>
    </body>
</html>
```

Dedicated Web Workers

The basic functionality of the dedicated Web Worker is illustrated in the following example. Before exploring the example, the essential steps involved are as follows:

1. The main web page creates a Web Worker whose code is specified in the file, say *workerTask.js*.

```
var worker = new Worker("workerTask.js");
```

The browser spawns a new thread that downloads and executes the specified code asynchronously.

2. If communication is required from the main web page to the Web Worker, the *postMessage* method is used which accepts string or JSON data.

```
worker.postMessage(data);
```

3. To listen for messages from the Web Worker, register the *onmessage* event handler for the *message* event.

```
worker.addEventListener("message", handler1, false);
```

4. In the *workerTask.js* file, listen for messages from the main page by registering the *onmessage* event handler.

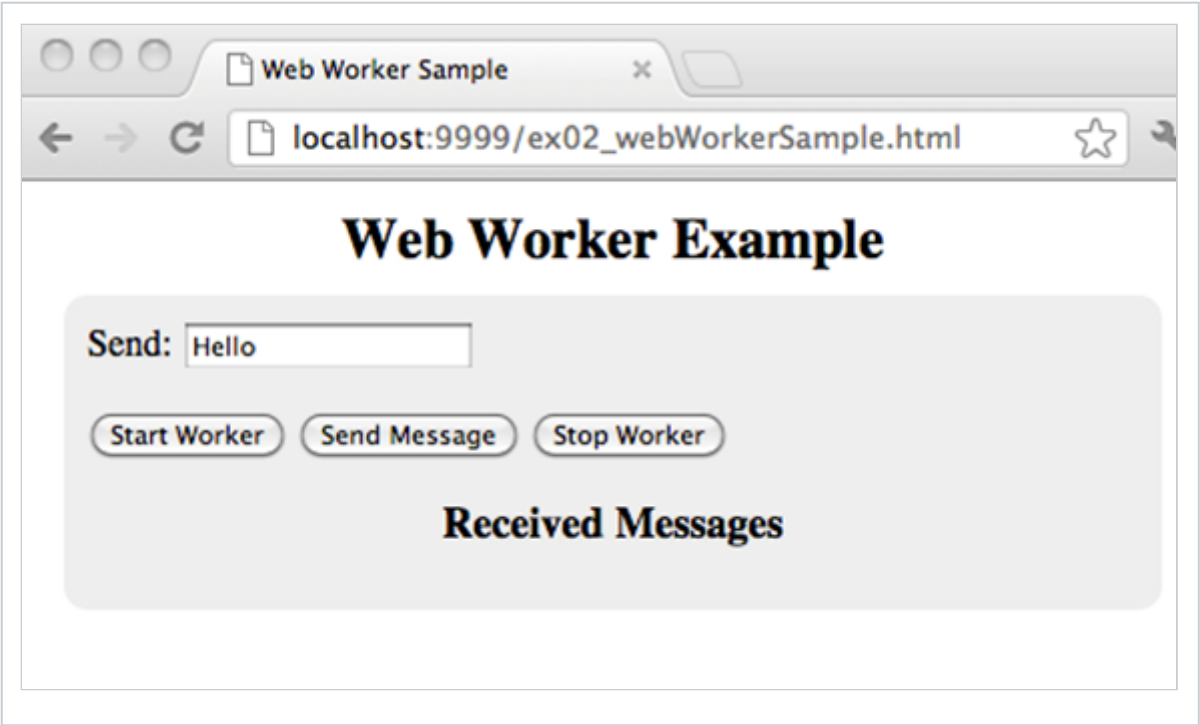
```
self.addEventListener("message", handler2, false);
```

The above functionality is illustrated through the following example that provides the capabilities for starting the Web Worker, sending messages to the Web Worker, and terminating the Web Worker.

ex02_webWorkerSample.html

```
<!doctype html>
<html>
    <head>
        <title>Web Worker Sample</title>
        <meta charset="utf-8">
        <script src="ex02_webWorkerSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Web Worker Example</h2>
      </header>

        <section>
            <article>
                Send:   <input type="text" id="msg" value="Hello">
                <p/>
                <button id="startButton">Start Worker</button>
                <button id="sendButton">Send Message</button>
                <button id="stopButton">Stop Worker</button>
                <p/>
                <h3>Received Messages</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The above HTML renders as shown below. The messages received from the Web Worker are also displayed as *line items*.



The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons. The global variable *myWorker* keeps track of the Web Worker.

ex02_webWorkerSample.js

```
1  window.onload = init;
2
3  // the worker
4  var myWorker;
5
6  function init() {
7      var startButton = document.getElementById("startButton");
8      startButton.onclick = startWorker;
9      var sendButton = document.getElementById("sendButton");
10     sendButton.onclick = sendMessageToWorker;
11     var stopButton = document.getElementById("stopButton");
12     stopButton.onclick = stopWorker;
13 }
```

When the user clicks the *Start* button, the *startWorker* function shown below checks that the Web Worker has not yet been created, then creates the new *Worker* with the specified worker code, *echoWorker.js*. The *onmessage* event handler, *handlerReceipt*, responds to the messages from the Web Worker. The content received is accessed through the *data* property of the event.

ex02_webWorkerSample.js

```
14
15  // start the Web Worker and register the event handler
16  function startWorker(e) {
17      if (myWorker == null) {
18          myWorker = new Worker("echoWorker.js");
19          myWorker.addEventListener("message", handleReceipt, false);
20      }
21  }
22
23  // Handle messages received from the Web Worker
24  function handleReceipt(event) {
25      var itemsList = document.getElementById("items");
26      var item = document.createElement("li");
27      item.innerHTML = event.data;
28      items.appendChild(item);
29  }
```

When the user clicks on the *Send* button, the following function uses the *postMessage* method of the Web Worker object to send the user specified data, as shown in the snippet below. Similarly, when the user clicks on the *Stop* button, the *terminate* method kills the Web Worker.

ex02_webWorkerSample.js

```
30
31  // send message to the Web Worker
32  function sendMessageToWorker(e) {
33      var data = document.getElementById("msg").value;
34      if (myWorker != null) {
35          myWorker.postMessage(data);
36      }
37   }
38
39  // terminate the Web Worker
40  function stopWorker(e) {
41      if (myWorker != null) {
42          myWorker.terminate();
43          myWorker = null;
44      }
45  }
```

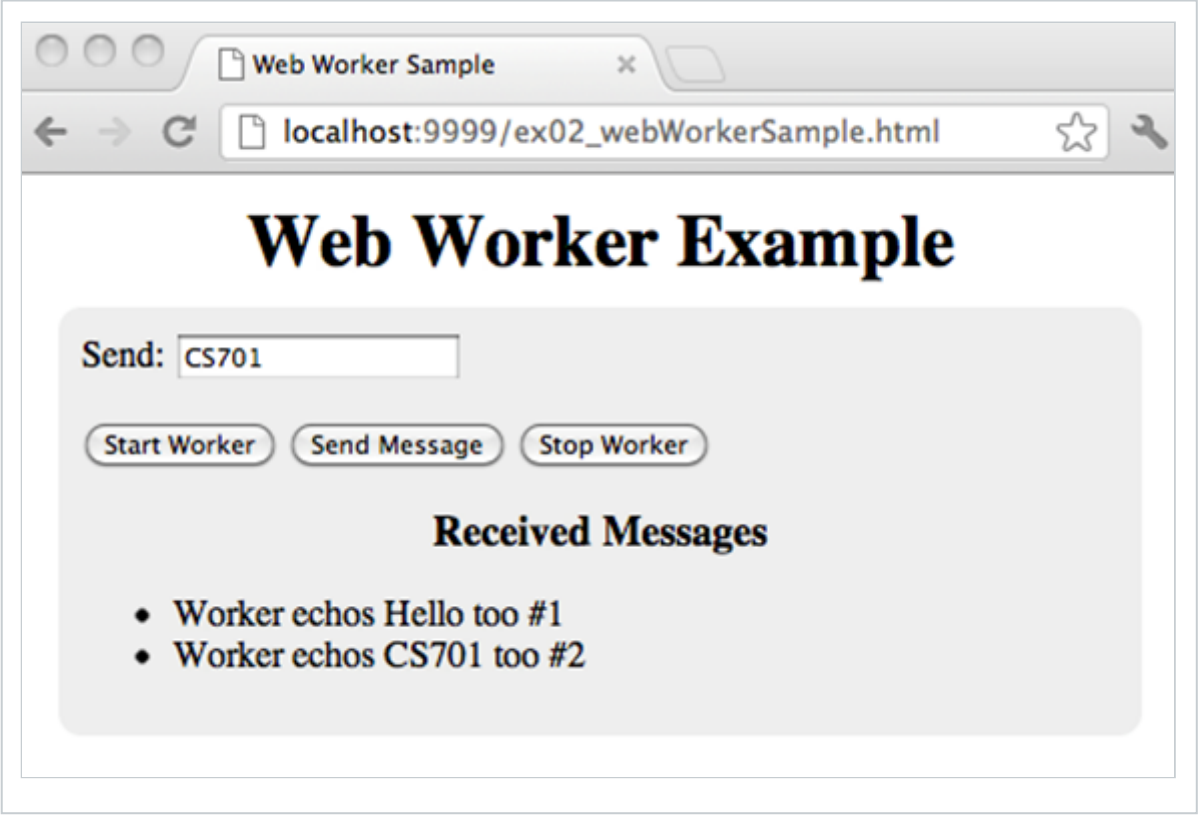
The Web Worker's code is shown in the example below. The *onmessage* event handler of the Web Worker (accessed with *self*) uses its *postMessage* method to respond to the data received from the Web page. The event's

data property is the message received from the Web page. The worker echo's the same message with additional content at the beginning and at the end.

echoWorker.js

```
1  self.onmessage = messageHandler;
2  // or,
3  //addEventListener("message", messageHandler, true);
4
5  var counter = 1;
6
7  function messageHandler(e) {
8    postMessage("Worker echos " + e.data +
9                 " too #" + counter);
10   counter++;
11 }
```

The following demo shows a sample communication between the web page and the worker after starting the worker, and sending the messages *Hello* and *CS701*.



JSON Sample

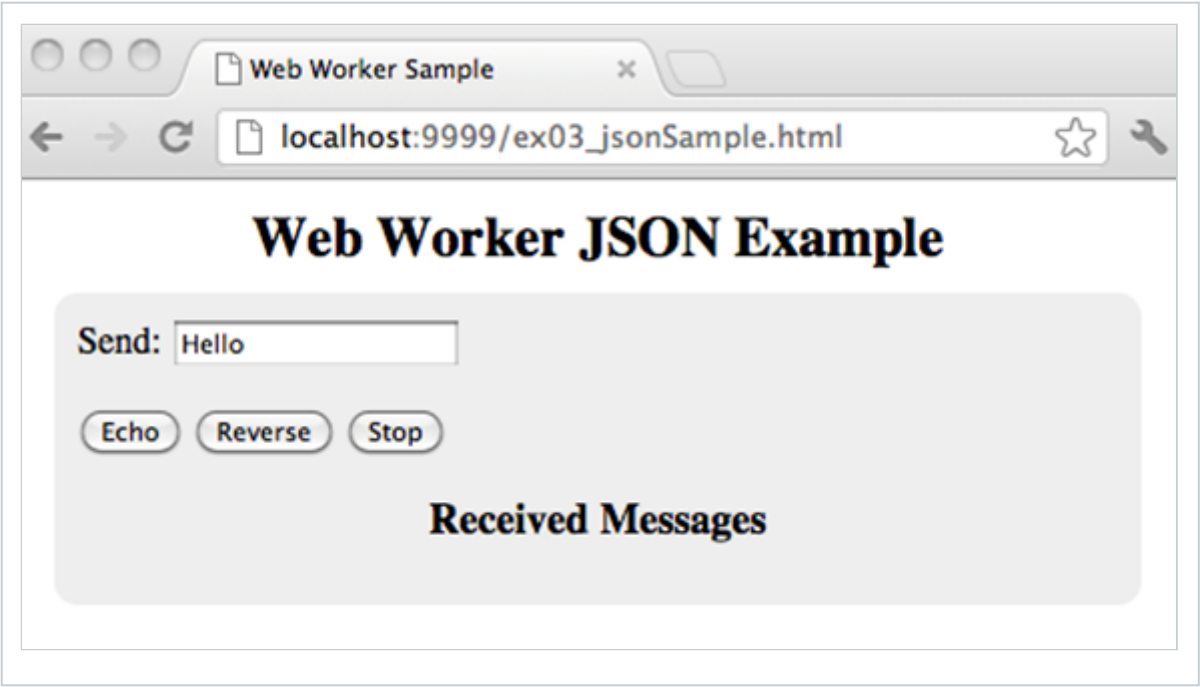
The communication between the web page and the worker is not restricted to string data. JavaScript objects can also be used in a similar fashion. The web page posts the data in JSON format. The data is serialized and sent to the worker. The worker deserializes the received data back to the JSON object that the worker processes. Similarly, the worker can also do the same. In the example shown below, the Web page sends JSON data while the worker sends string data.

The web page used for this example provides three scenarios of communication with the Web Worker—echoing the data, reversing the data, and commanding the worker to stop.

ex03_jsonSample.html

```
<!doctype html>
<html>
    <head>
        <title>Web Worker Sample</title>
        <meta charset="utf-8">
        <script src="ex03_jsonSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Web Worker JSON Example</h2>
      </header>

        <section>
            <article>
                Send:   <input type="text" id="msg" value="Hello">
                <p/>
                <button id="echoButton">Echo</button>
                <button id="revButton">Reverse</button>
                <button id="stopButton">Stop</button>
                <p/>
                <h3>Received Messages</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The above *html* is rendered as shown below.



The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons.

ex03_jsonSample.js

```
1  window.onload = init;
2
3  function init() {
4      var echoButton = document.getElementById("echoButton");
5      echoButton.onclick = sendEchoToWorker;
6      var revButton = document.getElementById("revButton");
7      revButton.onclick = sendReverseToWorker;
8      var stopButton = document.getElementById("stopButton");
9      stopButton.onclick = sendStopToWorker;
10 }
```

The Web Worker *jsonWorker.js* is started and the *onmessage* event handler handles the messages received from the Web Worker as shown below. The received message is added as a *line item* to the web page.

ex03_jsonSample.js

```
// the worker
var myWorker = new Worker("jsonWorker.js");
myWorker.onmessage = handleReceipt;

// Handle messages received from the Web Worker
function handleReceipt(event) {
    var itemsList = document.getElementById("items");
    var item = document.createElement("li");
    item.innerHTML = event.data;
    items.appendChild(item);
}
```

The following code shows the actions of the button clicks. The *cmd* and *message* names are used in the JSON format to simultaneously send the action command and the required input data. The *cmd* name uses the three values *echo*, *reverse*, and *stop* for the required actions to be processed by the Web Worker as shown below

ex03_jsonSample.js

```
// send messages to the Web Worker
function sendEchoToWorker(e) {
    var data = document.getElementById("msg").value;
    myWorker.postMessage({'cmd': 'echo', 'message': data});
}

function sendReverseToWorker(e) {
    var data = document.getElementById("msg").value;
    myWorker.postMessage({'cmd': 'reverse', 'message': data});
}

function sendStopToWorker(e) {
    var data = document.getElementById("msg").value;
    myWorker.postMessage({'cmd': 'stop', 'message': data});
}
```

The Web Worker functionality is shown below. When a message is received from the web page, the *cmd* property is examined and the appropriate action is taken on the value associated with the *message* property. The action for *echo* is straightforward. For the *reverse* action, the received message is reversed and the appropriate response is posted to the web page. For the *stop* command, the *close* method of the Web Worker terminates the worker itself.

jsonWorker.js

```
self.onmessage = messageHandler;
// or,
//addEventListener("message", messageHandler, true);

var counter = 1;

function messageHandler(e) {
    var data = e.data;
    switch (data.cmd) {
        case 'echo' :
            self.postMessage("Worker echos: " + data.message +
                " #" + counter);
            break;
        case 'reverse' :
            var reversed = data.message.split("").reverse().join("");
            self.postMessage("Worker reverses: " + reversed +
                " #" + counter);
            break;
        case 'stop' :
            self.postMessage("Worker stopped: " + data.message +
                " #" + counter);
            self.close(); // terminates the worker
            break;
    }
  counter++;
}
```

The following demo shows a sample communication between the web page and the worker for the message *Hello* and the three commands *echo*, *reverse*, and *stop*.

Inline Workers (Optional)

In the examples covered so far, the Web Worker code is defined in its own *JavaScript* file and the Web Worker is created using the respective file as the argument. Inline Web Worker code comes handy to create the workers on the fly, or to avoid multiple *JavaScript* files. The following example shows the *html* web page with the Web Worker defined inline in the *script* section.

For the inline Web Worker JavaScript code, the *type* of the *script* is *JavaScript/worker* so that the browser does not parse the code. The *id* attribute will be used to extract the content of the Web Worker. The *echo* example is shown below utilizing the inline Web Worker functionality.

ex04_inlineWorkerSample.html

```
<!doctype html>
<html>
    <head>
        <title>Web Worker Sample</title>
        <meta charset="utf-8">
        <script src="ex04_inlineWorkerSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Inline Web Worker Example</h2>
      </header>

  <script id="worker1" type="JavaScript/worker">
    var counter = 1;
    self.onmessage = function(e) {
        self.postMessage("Worker echos " + e.data + " too #" +
                         counter);
        counter++;
    }
  </script>

        <section>
            <article>
                Send:   <input type="text" id="msg" value="Hello">
                <p/>
                <button id="startButton">Start Worker</button>
                <button id="sendButton">Send Message</button>
                <button id="stopButton">Stop Worker</button>
                <p/>
                <h3>Received Messages</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons.

ex04_inlineWorkerSample.js

```
1  window.onload = init;
2
3  // the worker
4  var myWorker;
5
6  function init() {
7      var startButton = document.getElementById("startButton");
8      startButton.onclick = startWorker;
9      var sendButton = document.getElementById("sendButton");
10     sendButton.onclick = sendMessageToWorker;
11     var stopButton = document.getElementById("stopButton");
12     stopButton.onclick = stopWorker;
13 }
```

For accessing the Web Worker code, the *text content* of the *script* is accessed and the *Blob* object is created. A new object URL is created which is tied to the script content in the window. The contents of the Web Worker script are represented by the text of the created URL. The web page creates the Web Worker with the URL and the *onmessage* event handler is registered to handle the messages received from the worker.

ex04_inlineWorkerSample.js

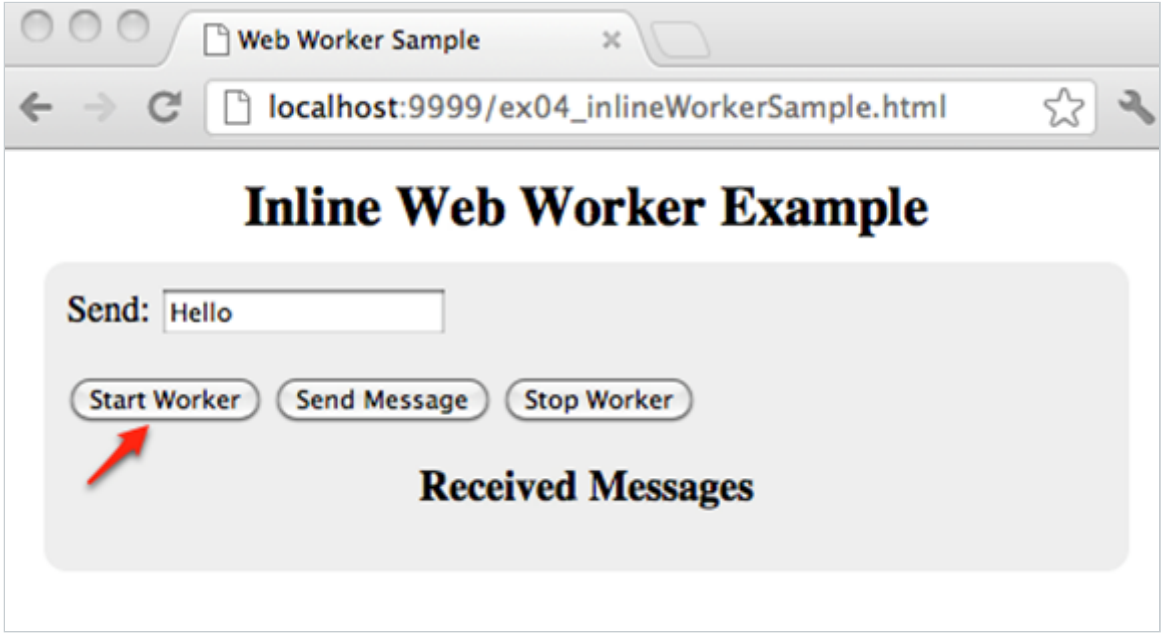
```
15 // start the Web Worker and register the event handler
16
17 function startWorker(e) {
18     if(myWorker == null) {
19         var content = document.querySelector('#worker1').textContent;
20         var bb = new Blob([content]);
21
22         // create a URL string which can be used to reference
23         // data stored in a DOM File / Blob object.
24
25         var objUrl = (window.webkitURL || window.URL);
26         var blobUrl = objUrl.createObjectURL(bb);
27         console.log(blobUrl);
28
29         myWorker = new Worker(blobUrl);
30         myWorker.addEventListener("message", handleReceipt, false);
31     }
32 }
```

The following code snippet shows the event handler that processes the received data from the Web Worker and adds a line item in the web page. The remaining two functions handle sending the message to the Web Worker and terminating the worker, respectively.

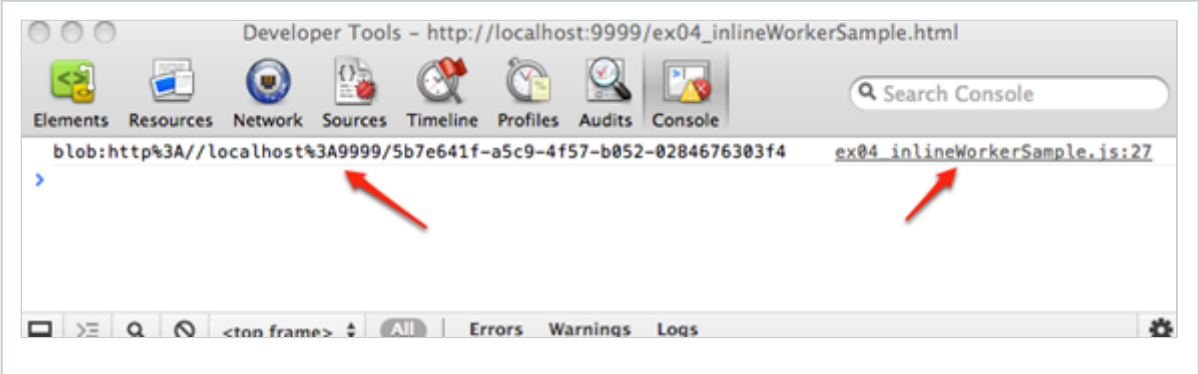
ex04_inlineWorkerSample.js

```
34  // Handle messages received from the Web Worker
35  function handleReceipt(event) {
36      var itemsList = document.getElementById("items");
37      var item = document.createElement("li");
38      item.innerHTML = event.data;
39      items.appendChild(item);
40  }
41
42  // send message to the Web Worker
43  function sendMessageToWorker(e) {
44      var data = document.getElementById("msg").value;
45      if (myWorker != null) {
46          myWorker.postMessage(data);
47      }
48   }
49
50  // terminate the Web Worker
51  function stopWorker(e) {
52      if (myWorker != null) {
53          myWorker.terminate();
54          myWorker = null;
55      }
56  }
```

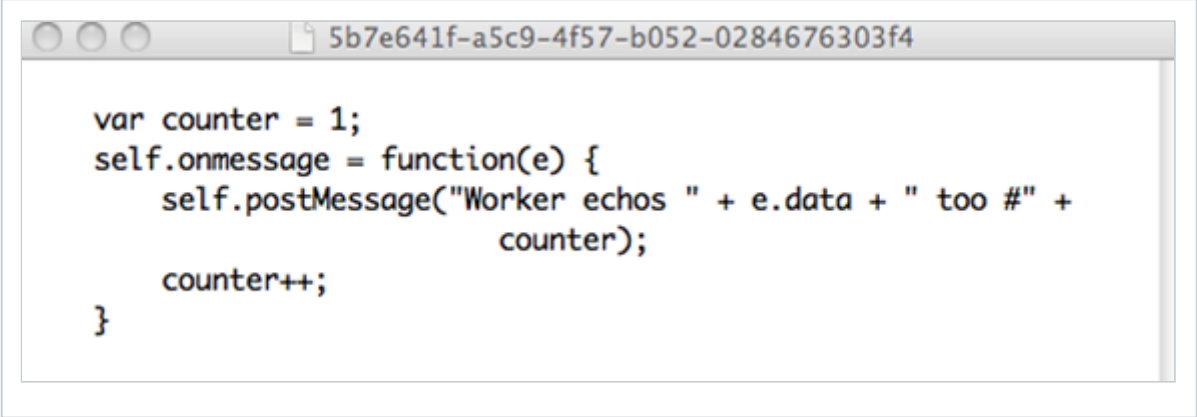
When the web page is opened in the browser and the *Start* button is clicked, the Web Worker is created.



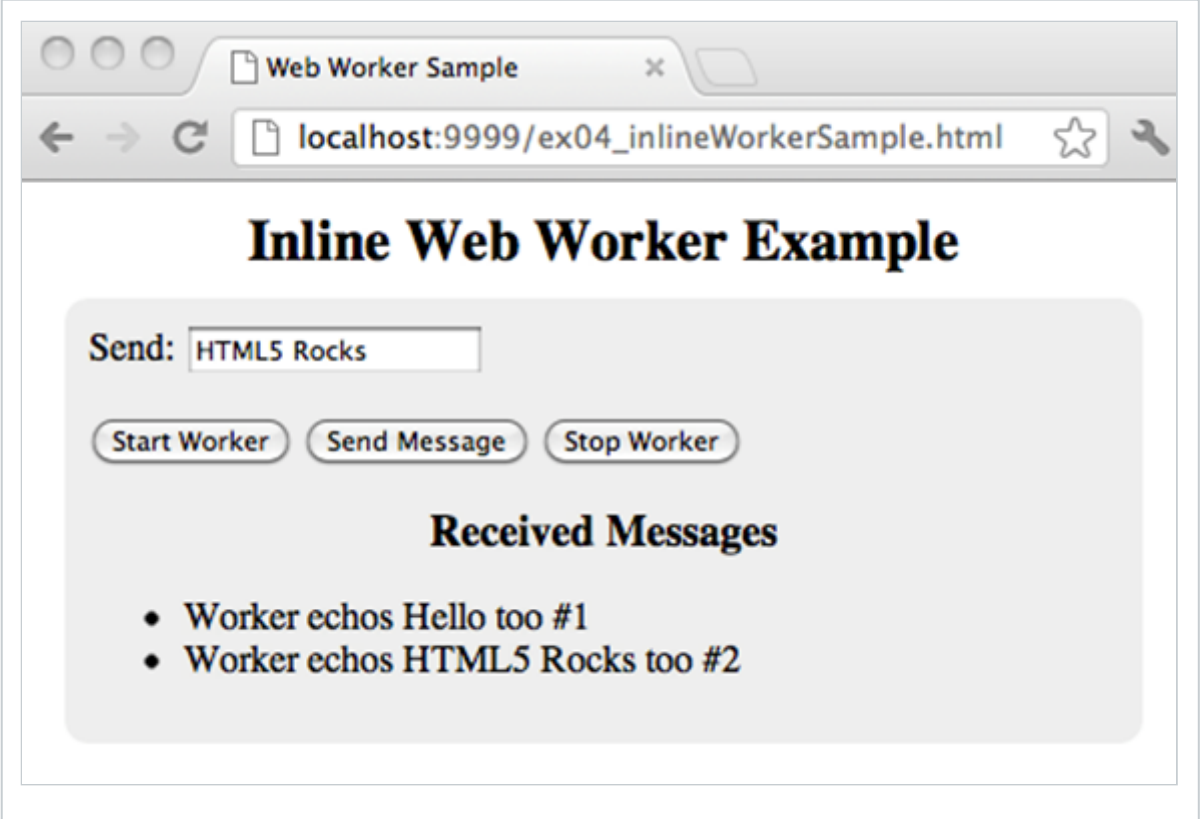
The URLobject that was created for the inline Web Worker code is shown below.



Opening the URL in the browser shows the contents of the Web Worker as shown below.



The responses received from the worker for the input messages *Hello* and *HTML5 Rocks* are as shown below.



Shared Workers (Optional)

The Web Workers seen so far are dedicated to the web page that created the Web Worker. On the other hand, a shared Web Worker is similar to the dedicated Web Worker in functionality except that it can be shared by multiple web pages from the same domain. Each web page creates the *SharedWorker* by specifying the worker's *JavaScript* file. Web pages from the same domain share the same worker. Shared Web Workers introduce the notion of *ports* to communicate with the correct web page that sends the message.

The following *html* page demonstrates the functionality and provides two buttons, one to start the shared Web Worker and the other to send messages to that worker.

ex05_sharedWorkerSample.html

```
<!doctype html>
<html>
    <head>
        <title>Web Worker Sample</title>
        <meta charset="utf-8">
        <script src="ex05_sharedWorkerSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Shared Web Worker Example</h2>
      </header>

        <section>
            <article>
                Send:   <input type="text" id="msg" value="Hello">
                <p/>
                <button id="startButton">Start Worker</button>
                <button id="sendButton">Send Message</button>
                <p/>
                <h3>Received Messages</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons.

ex05_sharedWorkerSample.js

```
window.onload = init;

// the worker
var myWorker;

function init() {
    var startButton = document.getElementById("startButton");
    startButton.onclick = startWorker;
    var sendButton = document.getElementById("sendButton");
    sendButton.onclick = sendMessageToWorker;
}
```

The following code snippet shows the creation of the *SharedWorker* by the web page. Communicating with the shared worker is done through the *port* attribute of the *SharedWorker* object. The *onmessage* event handler on the

port object handles the messages received to this web page from the shared worker as shown below. Similarly, for sending messages to the *shared worker*, the *postMessage* function is invoked on the *port* attribute of the *SharedWorker* object.

ex05_sharedWorkerSample.js

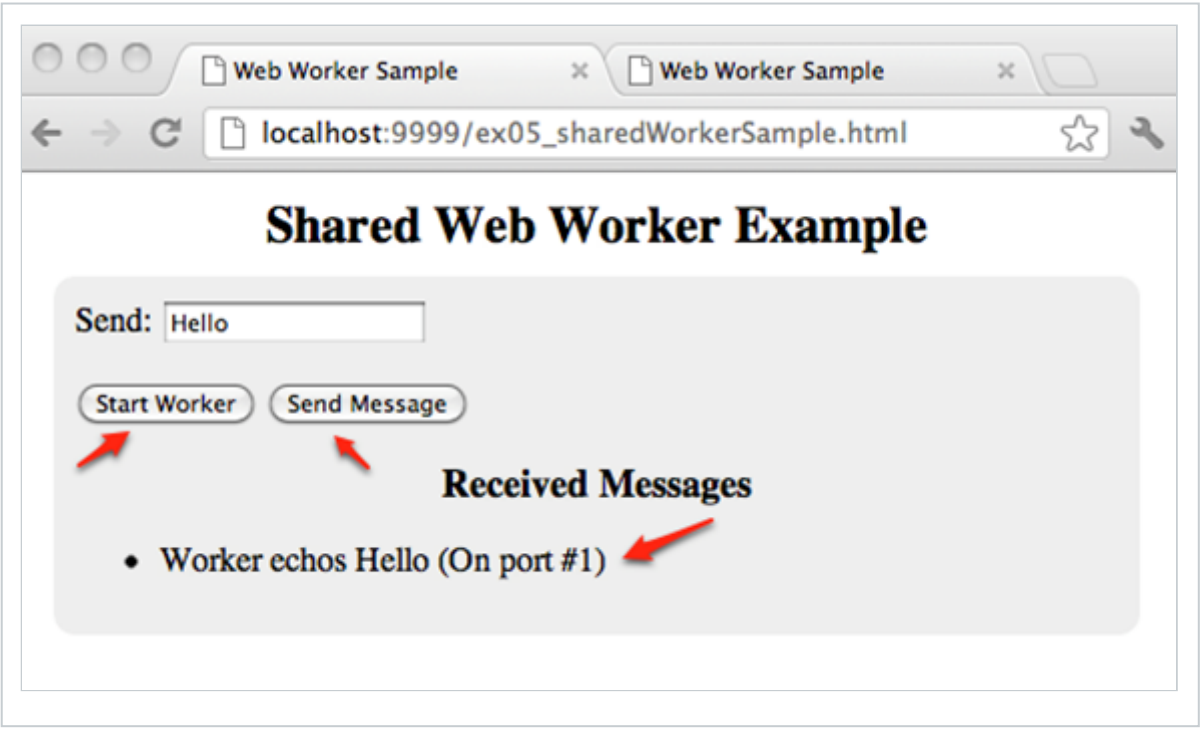
```
12
13  // start the Shared Web Worker and register the event handler
14  function startWorker(e) {
15      if (myWorker == null) {
16          myWorker = new SharedWorker("sharedWorker.js");
17          myWorker.port.addEventListener("message",
18                      handleReceipt, false);
19          myWorker.port.start();
20      }
21  }
22
23  // Handle messages received from the Web Worker
24  function handleReceipt(event) {
25      var itemsList = document.getElementById("items");
26      var item = document.createElement("li");
27      item.innerHTML = event.data;
28      items.appendChild(item);
29  }
30
31  // send message to the Web Worker
32  function sendMessageToWorker(e) {
33      var data = document.getElementById("msg").value;
34      if (myWorker != null) {
35          myWorker.port.postMessage(data);
36      }
37   }
```

The *Shared Web Worker* handles new client connections through the *onconnect* event handler.When a client connects to the shared worker, the *port* object that handles the client connection is retrieved and the *onmessage* event handler is associated as shown below.

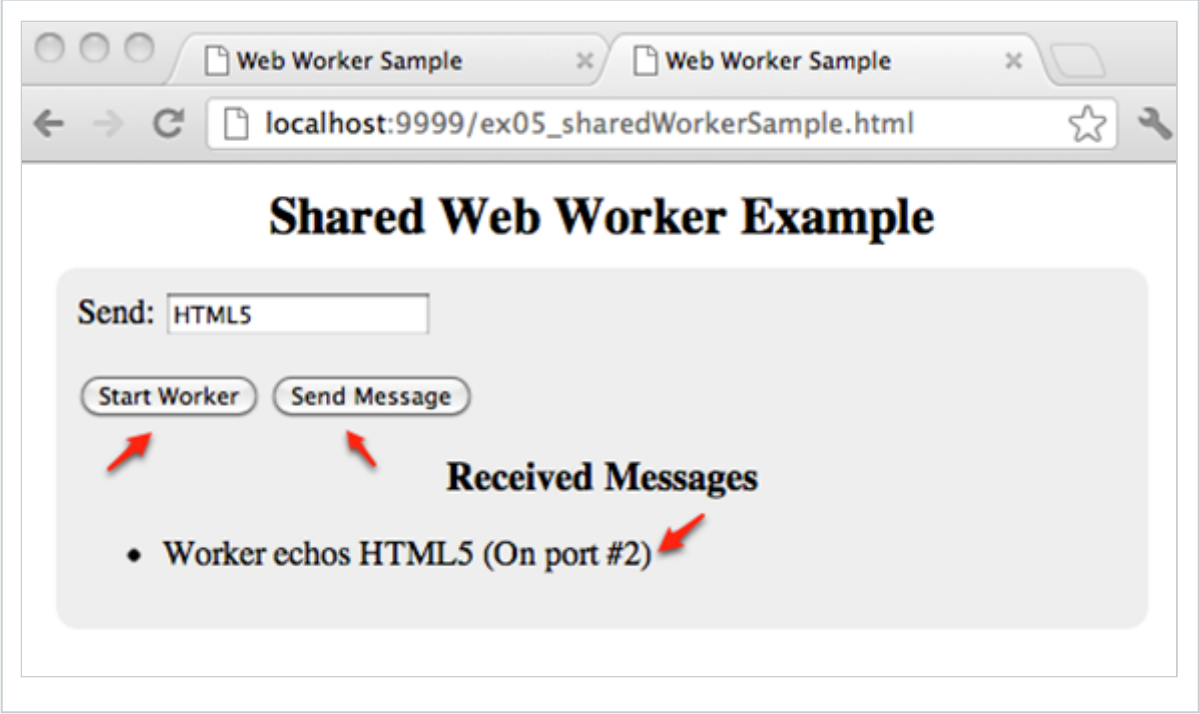
sharedWorker.js

```
1
2  var connections = 0; // count active connections
3  self.onconnect = connectionHandler;
4
5  function connectionHandler(e) {
6     var port = e.ports[0];
7     connections++;
8     port.addEventListener("message", function(e) {
9         port.postMessage("Worker echos " + e.data +
10         " (On port #" + connections + ")");
11
12     });
13    port.start();
14 }
```

To demonstrate the shared Web Worker functionality, open the web page in multiple tabs/windows. In each tab/window, click the *Start* button and send the message to the shared worker. The fist tab/window communication is shown below.



Similarly, the second tab/window's interaction is shown below.



Case Study – Broadcast Messages by Shared Worker (Optional)

Since the shared Web Worker handles all client connections from web pages from the same domain, the worker can be used for broadcasting messages to all the web pages.
In the following example, the connected clients are maintained in the *viewer's* associative array. Each *port* object is given a name *Client1, Client2, etc.* The *port* object is associated with the corresponding name in the associative array.

Whenever a message is received by the shared Web Worker, the response is posted to all the clients as shown in the *for* loop of the *onmessage* event handler below.

sharedBroadcastWorker.js | ex06_broadcastWorkerSample.html

```
1
2  var connections = 0; // count active connections
3  var viewers = {}; // store all connections
4  self.onconnect = connectionHandler;
5
6  function connectionHandler(e) {
7     var port = e.ports[0];
8     connections++;
9     port._name = "Client" + connections;
10    viewers[port._name] = port;
11
12    port.addEventListener("message", function(e) {
13        // Broadcast the message to each client
14        for (var viewer in viewers)
15        {
16          viewers[viewer].postMessage(e.target._name + ": " + e.data);
17        }
18
19     });
20
21    port.start();
22 }
```

To demo the above functionality, open the *ex06_broadcastWorkerSample* in multiple tabs/windows and communicate with the worker. The response appears in all the tabs/windows.

Loading External Scripts

External JavaScript files or libraries can be loaded into a Web Worker through the *importScripts* function as shown below.

```
importScripts('externalScript1.js');
importScripts('externalScript2.js');
…
```

The following shortcut can also be used to load multiple scripts using a single *importScripts* function with the script file names separated by a comma.

```
importScripts('externalScript1.js', 'externalScript2.js');
```

Setting a Timer Interval

A Web Worker can use the JavaScript timing API functions *setTimeout, setInterval, clearTimeout,* and *clearInterval*. The *setTimeout* method waits for the specified number of milliseconds and then executes the specified function once:

*setTimeout(*function, milliSeconds);

The *setInterval* method, on the other hand, executes the specified function repeatedly at the specified time interval.

setInterval(function, milliSeconds);

Web Storage

Introduction

HTML5 Web storage enables applications to store data on the client side that can be used across web requests. Before this, cookies were used by the web applications to store information on the client side. The cookies can be persistent (when an expiry date is set), or can be used only for the lifetime of the browser's session. However, with cookies, the web application cannot distinguish the cases when the user opens multiple browser windows or multiple tabs and works with the same web application. Also, cookies are limited in size and the browser imposes a limit on the number of cookies that can be stored for each domain. Each cookie has a name and a string value associated with it. If structured data has to be associated with a cookie, the web application will have to encode and decode the data. Also, with every browser request, the entire cookies for the web site are transmitted to and from the web server.

To overcome these shortcomings, HTML5 provides local storage known as Web Storage API for storing structured data on the client side. In this lecture, we will look into two such mechanisms, *sessionStorage* and *localStorage*. Both are based on the concept of storing the data on the client side using string name-value pairs.

How it Works

The first scenario is for working with web applications where the user can work on different transactions in multiple windows/tabs at the same time. The session storage API is provided for handling these scenarios. Web sites can add data to the session storage, and it will be accessible to any web page for the same web site opened in the same window/tab. Different windows/tabs have their own session storage object associated with them. Hence, data doesn't leak from one window/tab to the other. This is not possible with a cookie based web application. The values in the session storage are destroyed when the browser window's session ends.

For persistent storage, the local storage API is used to store data on the client side. There is no expiration date like the cookies. Each domain will have its own local storage object created on the client side. The data stored in the local storage is accessible for all web pages and scripts from that domain.

Browser Support

The storage database for a web site is accessed through the *window* object by checking for the existence of *window.sessionStorage* or *window.localStorage* as shown in the sample below.

ex01_checkWebStorageSupport.html

```
<!DOCTYPE html>
<html>
    <body>
        <script type="text/javascript">
            if(window.sessionStorage)
                document.writeln("sessionStorage support is available.")
            else
                document.writeln("sessionStorage support is not available.")

            if(window.localStorage)
                document.writeln("localStorage support is available.")
            else
                document.writeln("localStorage support is not available.")
        </script>
    </body>
</html>
```

Storage API

Both the *sessionStorage* and *localStorage* implement the *Storage* interface shown below.

```
interface Storage {
  readonly attribute unsigned long length;
  DOMString? key(unsigned long index);
  getter DOMString getItem(DOMString key);
  setter creator void setItem(DOMString key, DOMString value);
  deleter void removeItem(DOMString key);
  void clear();
};
```

The storage is based on named key-value pairs. The *setItem* method is used to add a new item to the storage using the specified key and the specified value. If the key already exists in the storage, the specified value replaces the existing value. The *getItem* returns the current value associated with the specified key. If the specified key doesn't exist in the storage, the method returns *null*. The *length* attribute returns the number of key-value pairs associated currently with the storage. The *key* method returns the name of the key located in the storage for the specified index (zero-based). If the index is greater than or equal to the length, the method returns null. The *removeItem* method removes the associated entry from the storage for the specified key. If the key doesn't exist, the method does nothing. The *clear* method removes all the associated key-value entries from the storage.

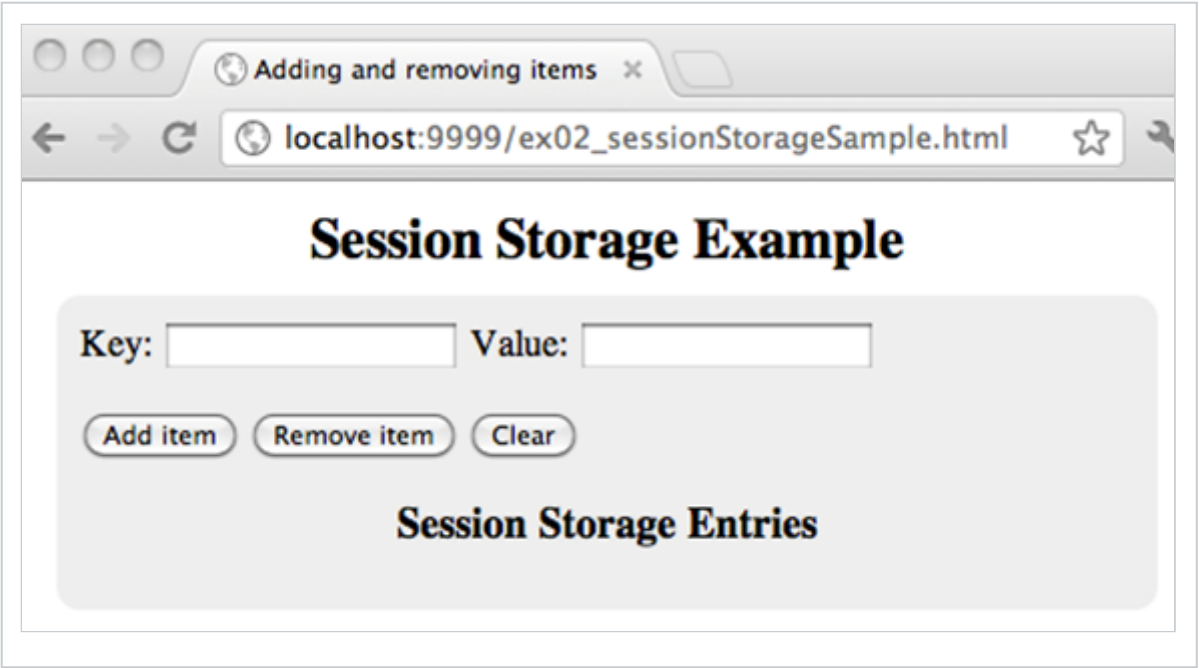
Session Storage

The session storage capabilities are illustrated with the following example. Two input fields are provided representing the key and the value. With the help of button clicks. items are added, removed, or cleared from the *sessionStorage* of the window.

ex02_sessionStorageSample.html

```
<!doctype html>
<html>
    <head>
        <title>Adding and removing items</title>
        <meta charset="utf-8">
        <script src="ex02_sessionStorageSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Session Storage Example</h2>
      </header>

        <section>
            <article>
                Key:   <input type="text" id="key">
                Value: <input type="text" id="value">
                <p/>
                <button id="addButton">Add item</button>
                <button id="remButton">Remove item</button>
                <button id="clrButton">Clear</button>
                <p/>
                <h3>Session Storage Entries</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The above *html* is rendered as shown below. The existing session storage entries are shown in the web page as well.



The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons. The *updateDOMItems* function shown later shows the current session storage entries in the web page.

ex02_sessionStorageSample.js

```
1  window.onload = init;
2
3  function init() {
4      var addButton = document.getElementById("addButton");
5      addButton.onclick = addItem;
6      var remButton = document.getElementById("remButton");
7      remButton.onclick = removeItem;
8      var clrButton = document.getElementById("clrButton");
9      clrButton.onclick = clearItems;
10
11     updateDOMItems();
12 }
```

The following snippet shows the event handlers for adding an item to the session storage, removing an item, and clearing all the items. The user input values are retrieved for the key and the associated value. The *setItem* method on the *sessionStorage* stores the key-value pair. If the key already exists, the new value overwrites the existing value. Similarly, the *removeItem* method removes the specified key and the associated entry from the *sessionStorage.* The *clear* method removes all entries for this window from the *sessionStorage*.

ex02_sessionStorageSample.js

```
13
14  function addItem(e) {
15      var key = document.getElementById("key").value;
16      var value = document.getElementById("value").value;
17
18      // add item to session storage
19      window.sessionStorage.setItem(key, value);
20
21      updateDOMItems();
22  }
23
24  function removeItem(e) {
25      var key = document.getElementById("key").value;
26
27      // remove item from session storage
28      window.sessionStorage.removeItem(key);
29
30      updateDOMItems();
31  }
32
33  function clearItems() {
34      // clear session storage
35      window.sessionStorage.clear();
36
37      updateDOMItems();
38  }
```

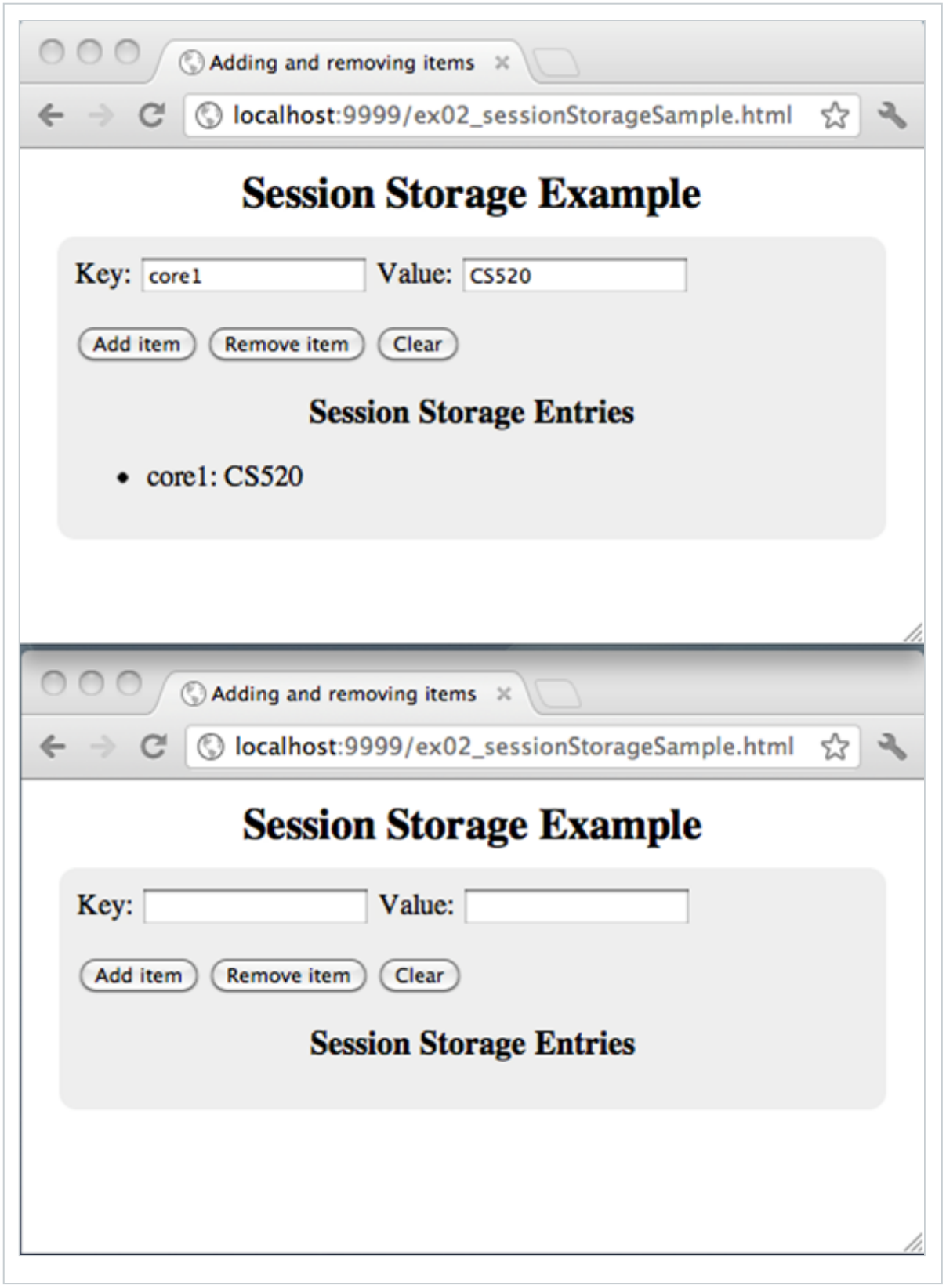
The *updateDOMItems* function clears the entries listed in the web page, iterates over the keys in the *sessionStorage*, and adds a new *line item* to the web page as shown below.

ex02_sessionStorageSample.js

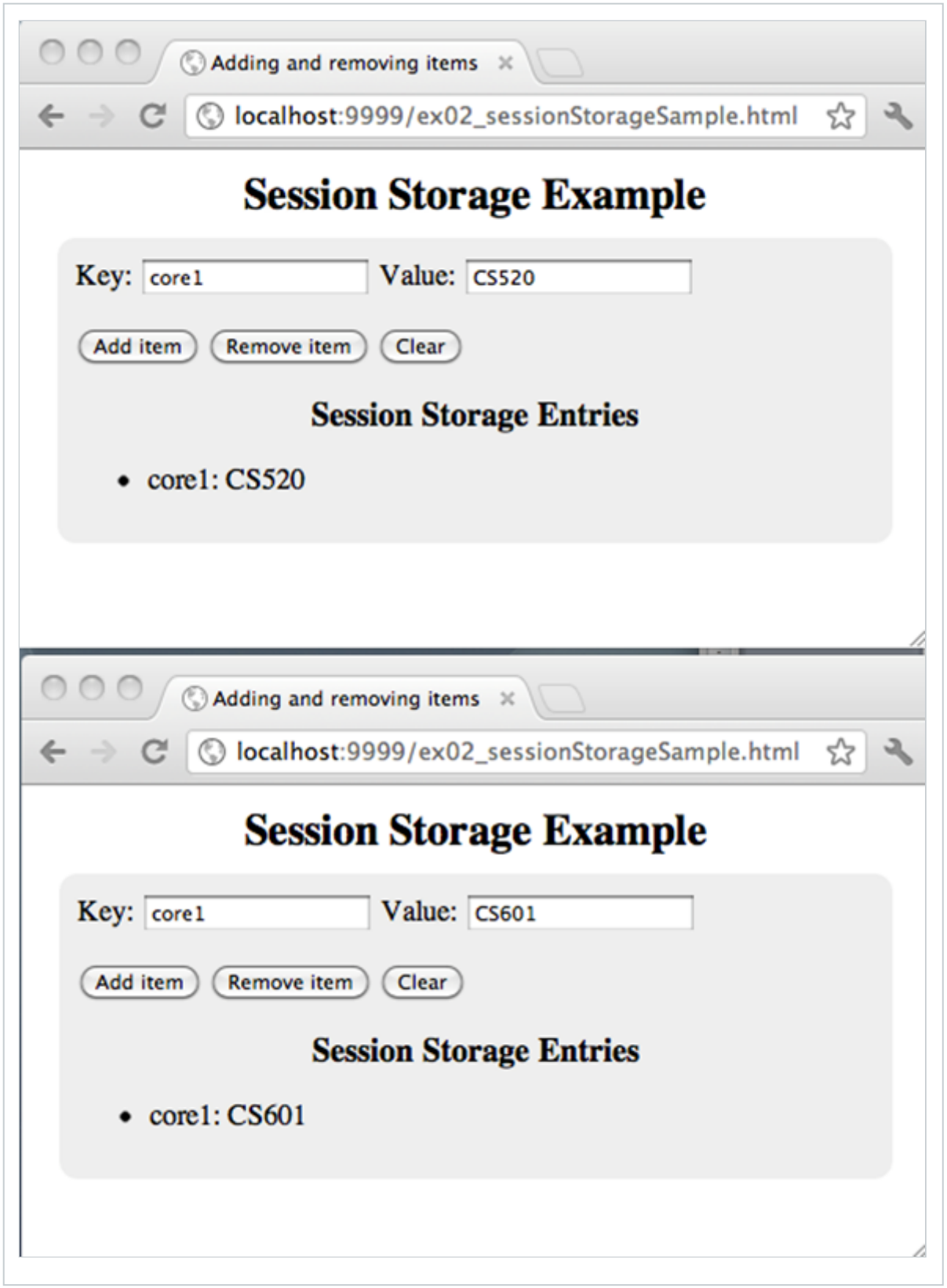
```
39
40  function updateDOMItems() {
41      var itemsList = document.getElementById("items");
42      // Clear shown list
43      itemsList.innerHTML = '';
44      // Add current items to the list
45      for (key in window.sessionStorage) {
46          addItemToDOM(key, sessionStorage[key]);
47      }
48  }
49
50  function addItemToDOM(key, value) {
51      var items = document.getElementById("items");
52      // create a line item and add to the end of the list
53      var item = document.createElement("li");
54      item.innerHTML = key + ": " + value;
55
56      items.appendChild(item);
57  }
```

Session Storage Demo

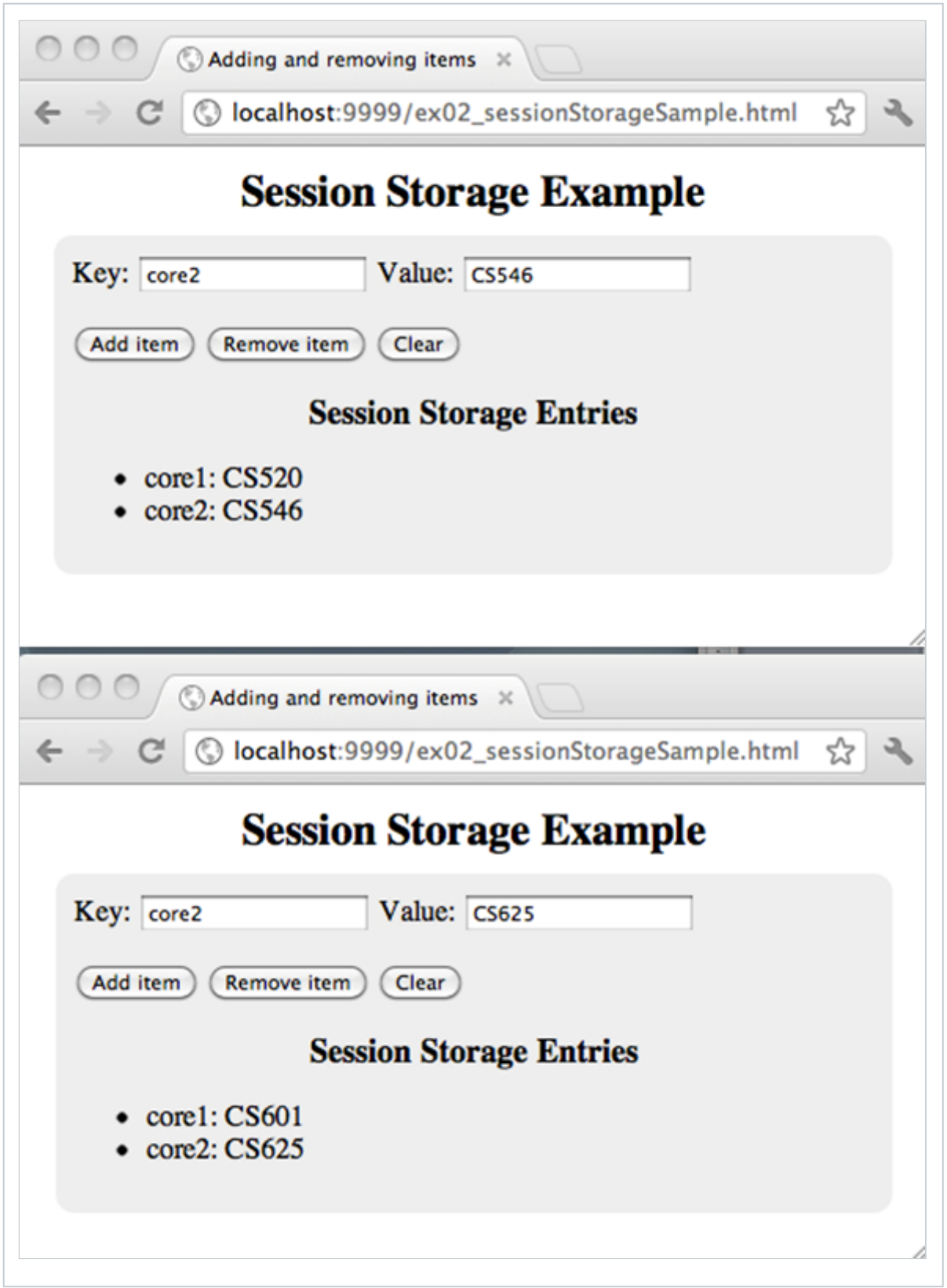
The uniqueness of the *sessionStorage* object for each window/tab of the browser is demonstrated using the above example with two windows opened at the same time. In the first figure shown below, a new key-value pair is added in the first window.



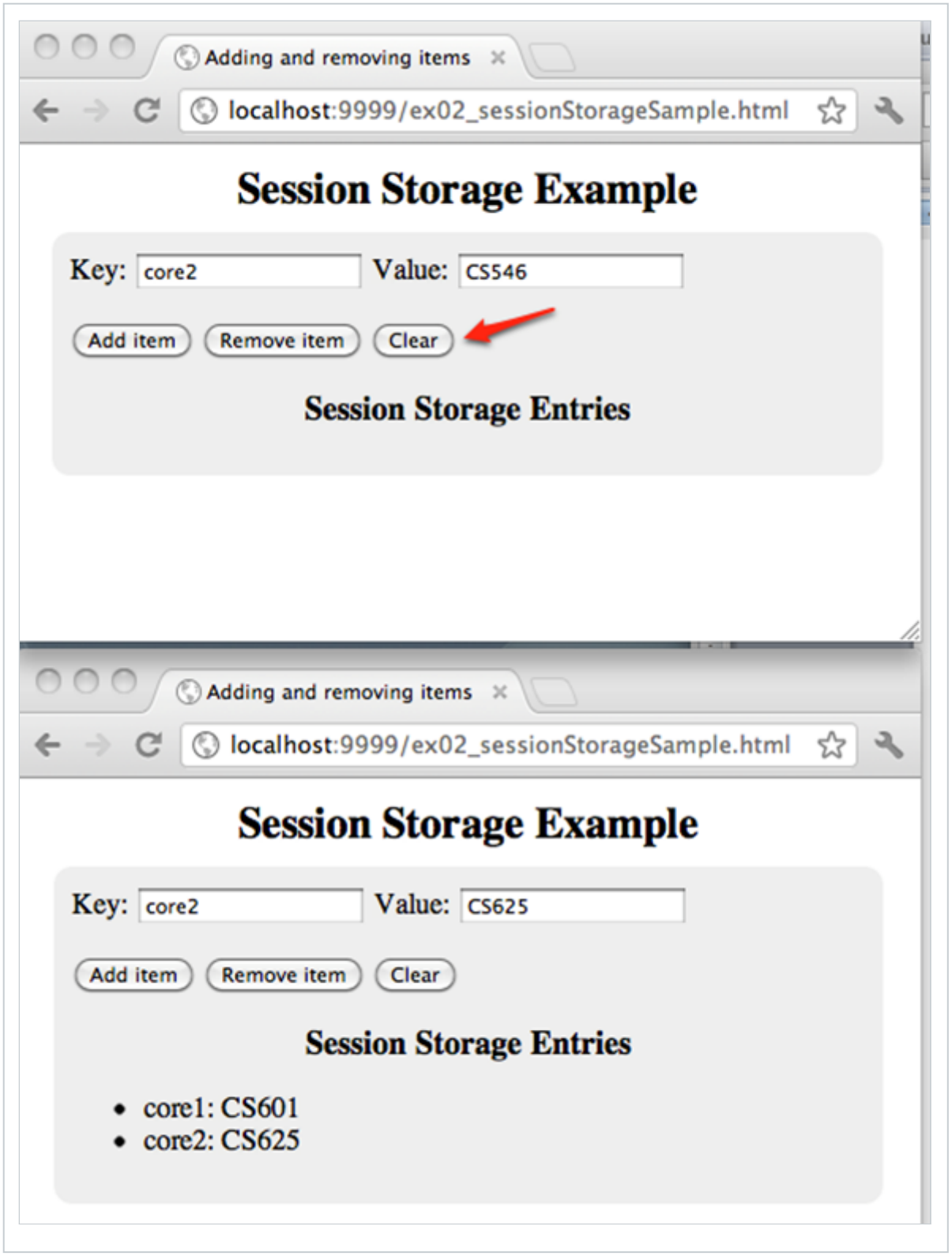
In the next step, a new key-value pair is added in the second window. From the resulting list shown, it is obvious that the values of the first window are not affecting the values in the second window.



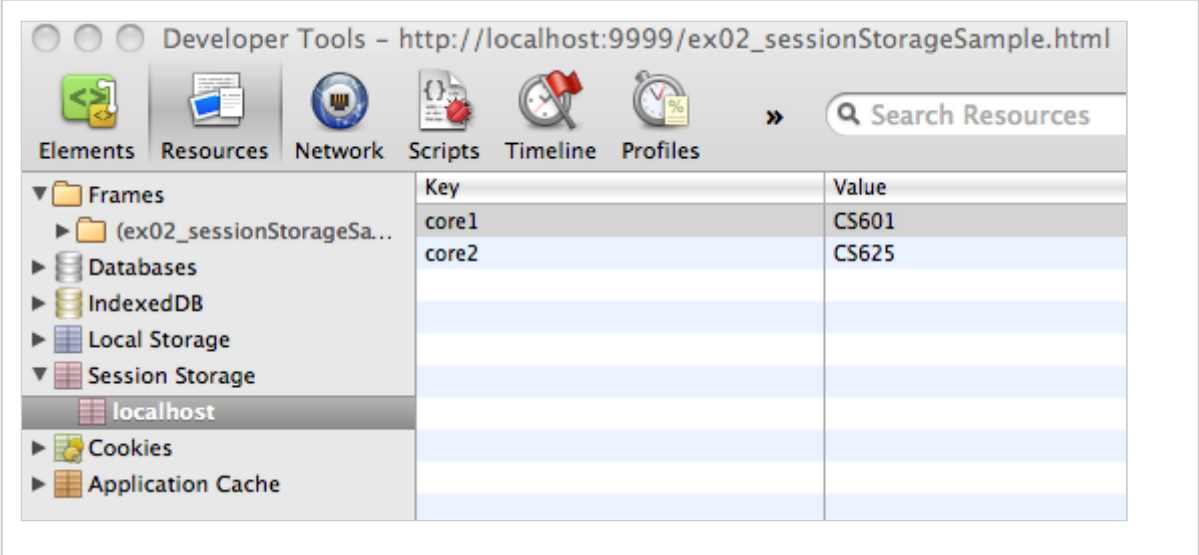
In the next step, a new key-value pair is added for both the windows as shown in the figure below.



In the next step, the entries in the *sessionStorage* for the first window are cleared.



The entries in the *sessionStorage* for the second window are viewed in the developer tools as shown below.



Local Storage

The local storage on the client is unique for each domain and hence the web applications from the same domain share the local storage. Data can be shared amongst these web applications. The following example will be used to illustrate the *localStorage* capabilities. In addition to the *add*, *remove*, and *clear* buttons, a *refresh* button is also provided.

ex03_localStorageSample.html

```
<!doctype html>
<html>
    <head>
        <title>Adding and removing items</title>
        <meta charset="utf-8">
        <script src="ex03_localStorageSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Local Storage Example</h2>
      </header>

        <section>
            <article>
                Key:   <input type="text" id="key">
                Value: <input type="text" id="value">
                <p/>
                <button id="addButton">Add item</button>
                <button id="remButton">Remove item</button>
                <button id="clrButton">Clear</button>
                <button id="refreshButton">Refresh</button>

                <p/>
                <h3>Local Storage Entries</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* code is shown below. The following snippet adds the event handlers for the buttons. The *updateDOMItems* function illustrated later shows the current local storage entries in the web page.

ex03_localStorageSample.js

```
window.onload = init;

function init() {
    var addButton = document.getElementById("addButton");
    addButton.onclick = addItem;
    var remButton = document.getElementById("remButton");
    remButton.onclick = removeItem;
    var clrButton = document.getElementById("clrButton");
    clrButton.onclick = clearItems;
    var refreshButton = document.getElementById("refreshButton");
    refreshButton.onclick = refreshItems;

    updateDOMItems();
}
```

The following snippet shows the event handlers for adding an item to the local storage, removing an item, and clearing all the items. The user input values are retrieved for the key and the associated value. The *setItem* method on the *localStorage* stores the key-value pair. If the key already exists, the new value overwrites the existing value. Similarly, the *removeItem* method removes the specified key and the associated entry from the *localStorage.* The *clear* method removes all entries for the web domain from the *localStorage*.

ex03_localStorageSample.js

```
15
16  function addItem(e) {
17      var key = document.getElementById("key").value;
18      var value = document.getElementById("value").value;
19
20      // add item to local storage
21      window.localStorage.setItem(key, value);
22
23      updateDOMItems();
24  }
25
26  function removeItem(e) {
27      var key = document.getElementById("key").value;
28
29      // remove item from local storage
30      window.localStorage.removeItem(key);
31
32      updateDOMItems();
33  }
34
35  function clearItems() {
36      // clear session storage
37      window.localStorage.clear();
38
39      updateDOMItems();
40  }
41
42  function refreshItems() {
43       updateDOMItems();
44  }
```

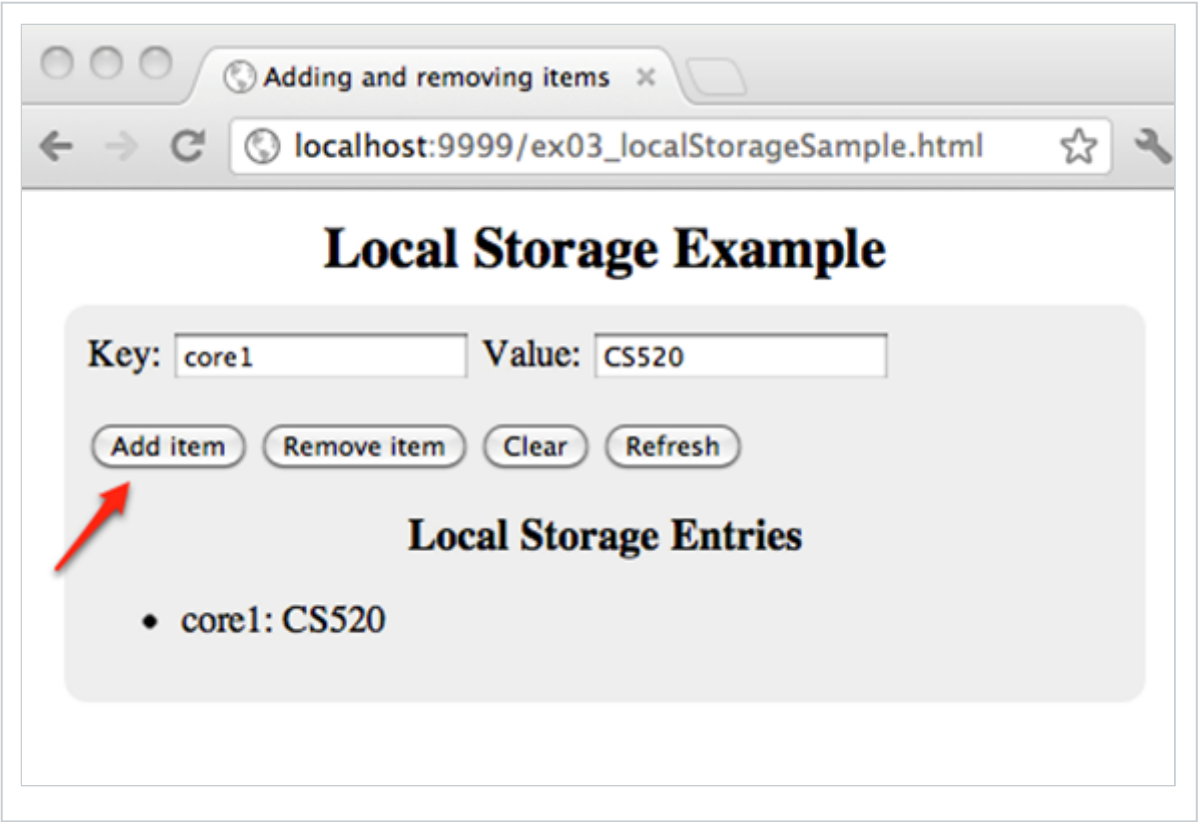
The *updateDOMItems* function clears the entries listed in the web page, iterates over the keys in the *localStorage*, and adds a new *line item* to the web page as shown below.

ex03_localStorageSample.js

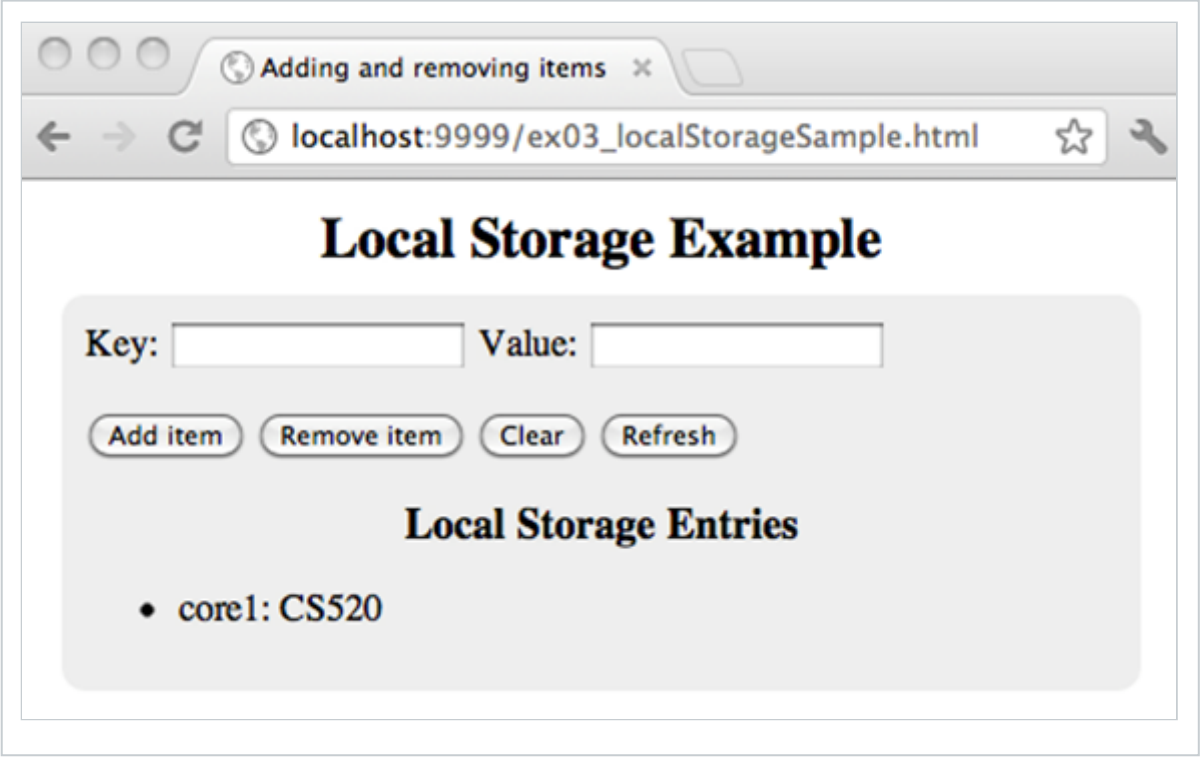
```
45
46  function updateDOMItems() {
47      var itemsList = document.getElementById("items");
48      // Clear shown list
49      itemsList.innerHTML = '';
50      // Add current items to the list
51      for (key in window.localStorage) {
52          addItemToDOM(key, localStorage[key]);
53      }
54  }
55
56  function addItemToDOM(key, value) {
57      var items = document.getElementById("items");
58      // create a line item and add to the end of the list
59      var item = document.createElement("li");
60      item.innerHTML = key + ": " + value;
61
62      items.appendChild(item);
63  }
```

Local Storage Demo

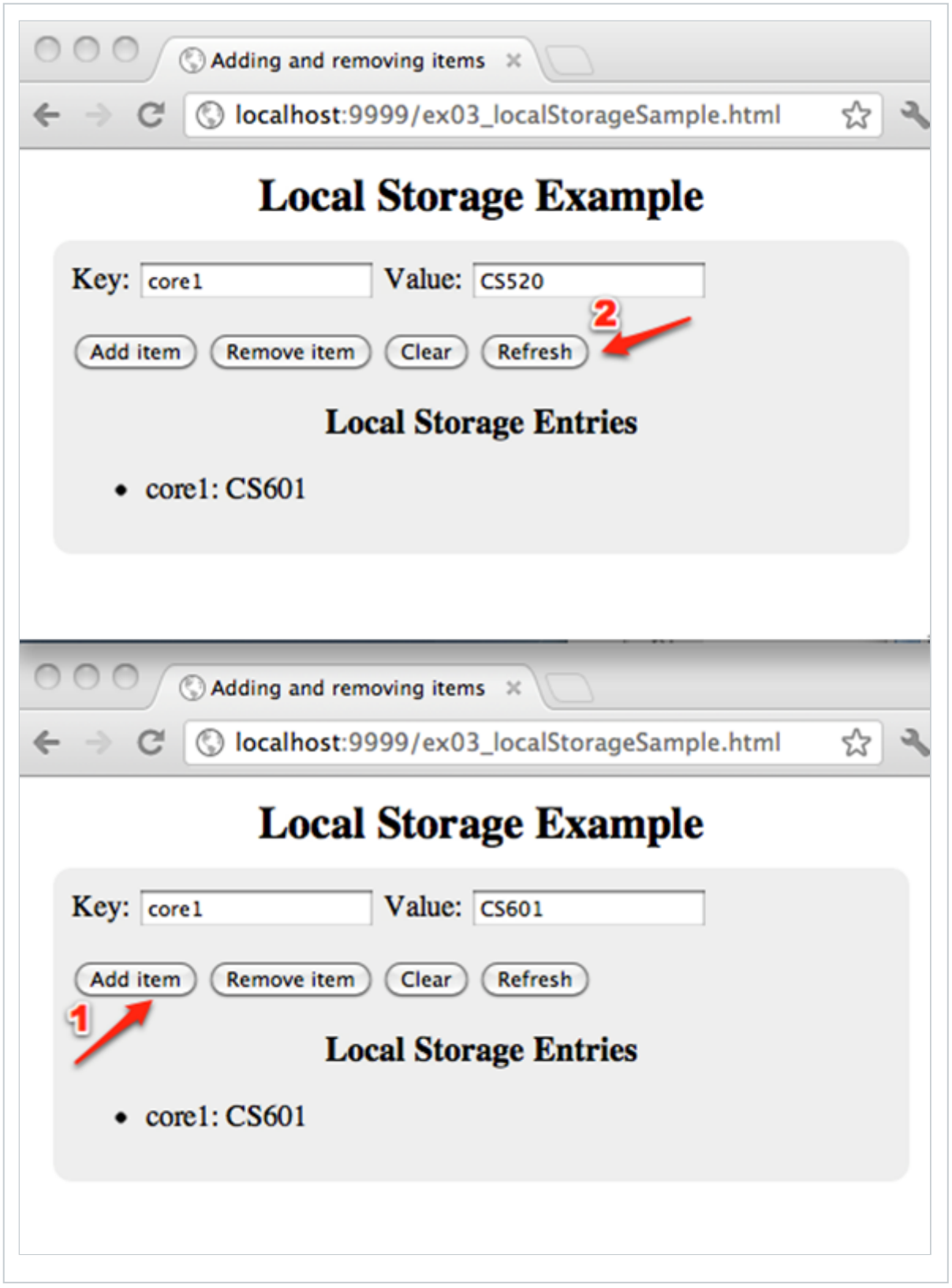
The usefulness of the *localStorage* object for each web domain is demonstrated using the above example with two windows opened at the same time. In the first figure shown below, a new key-value pair is added in the first window with key as *core1* and the value as *CS520*.



Now, open a new browser window referring to the same web page. The added item in the first window is now visible in the second window that lists the entries when the web page is loaded.

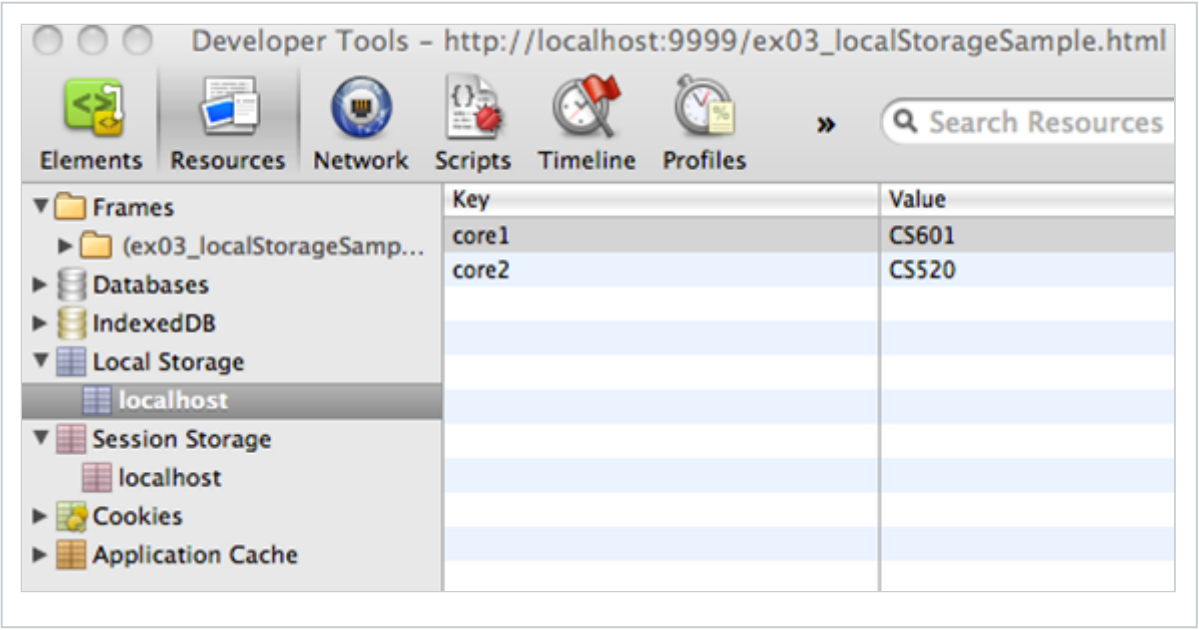


In the next step, using the second window, the same key *core1* is used with a different value *CS601* and item is added to the *localStorage*. The first window is then refreshed which lists the new value for the key *core1*.



In the next step, a new pair is added in the first window with the key *core2* and the associated value *CS520.* A refresh of the second window lists both the entries for the keys *core1* and *core2*.

The entries in the *localStorage* for the web page are viewed in the developer tools as shown below.



Storage Event Handling (Optional)

In the previous example, an explicit refresh has to be done in the second window for the changes to be visible made by the first window. The *storage* event is fired whenever the *localStorage* object changes. Any web page from the same domain that subscribes to this event will be notified of the changes—adding/updating an item, removing and item, and clearing all the items. Changes in the local storage can then be automatically reflected in the other web pages from the same domain.

The following example is used to demonstrate the *storage* event. The web page lists the event and its properties whenever there are changes to the *localStorage*.

ex04_storageEvents.html

```
<!doctype html>
<html>
    <head>
        <title>Storage Log</title>
        <meta charset="utf-8">
        <script src="ex04_storageEvents.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2> Storage Log</h2>
      </header>

        <section>
            <article>
                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

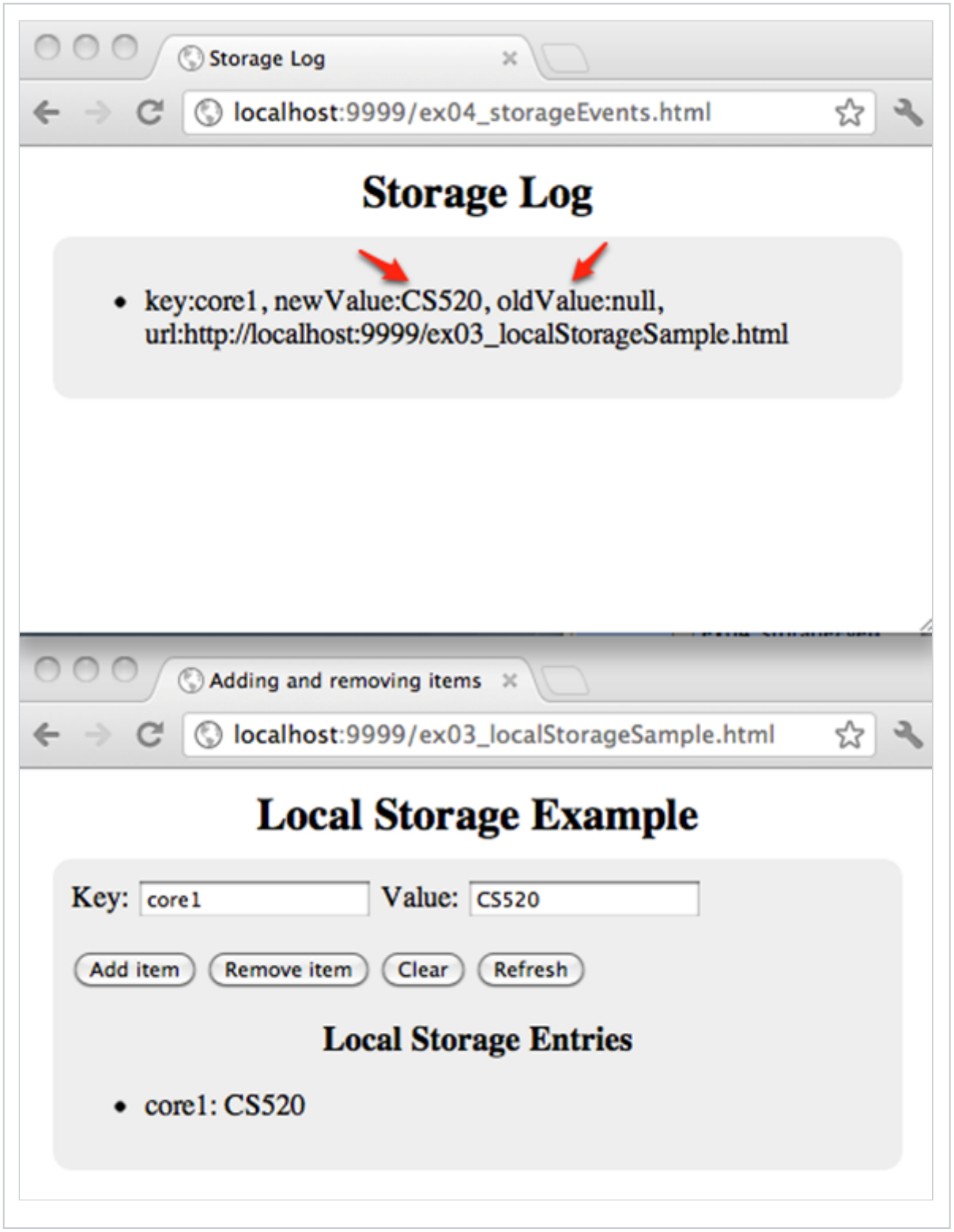
The associated *JavaScript* shown below registers the handler for the *storage* event for the window. The *storage* event's properties provide the information about the associated key, the old value, the new value, and the *URL* of the web application that caused a change in the local storage.

ex04_storageEvents.js

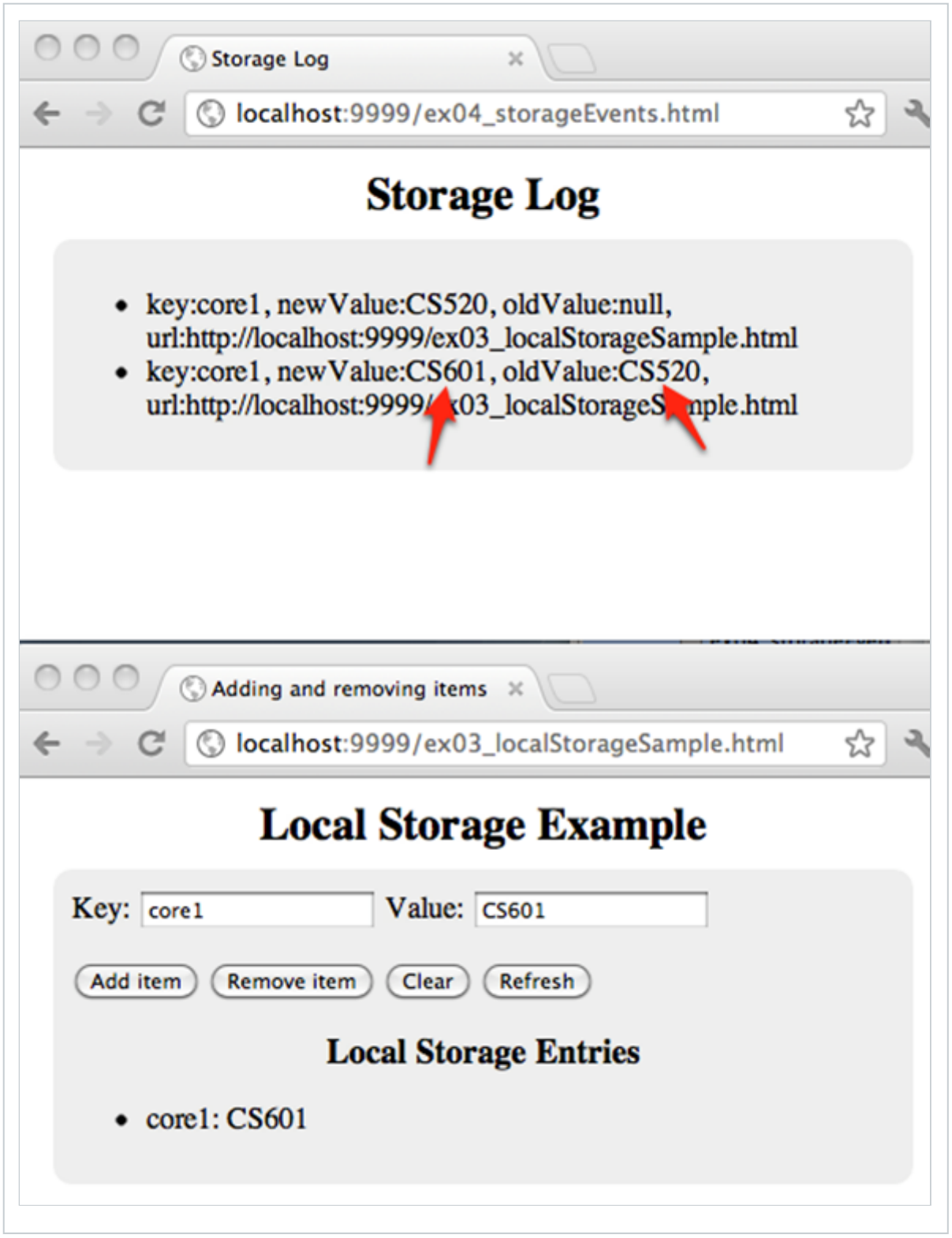
```
// add listeners on storage events
window.addEventListener("storage", displayStorageEvent, false);
// or
// window.onstorage = displayStorageEvent;

// display new storage events
function displayStorageEvent(e) {

    var items = document.getElementById("items");
    // create a line item and add to the end of the list
    var item = document.createElement("li");
    item.innerHTML = "key:" + e.key +
        ", newValue:" + e.newValue +
        ", oldValue:" + e.oldValue +
        ", url:" + e.url;

    items.appendChild(item);
}
```

Storage Event Handling Demo

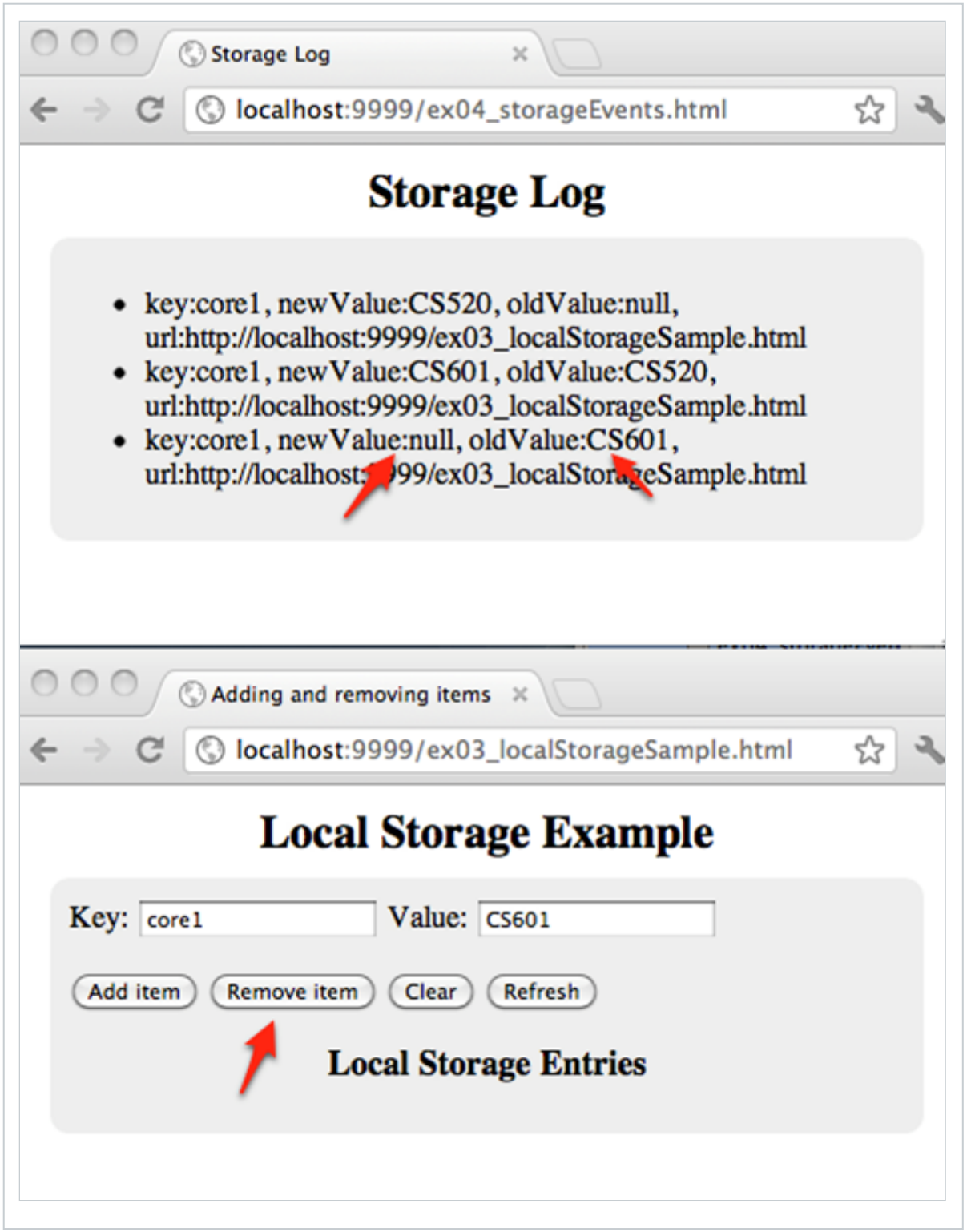
The local storage sample (*ex03_localStorageSample*.html) from the previous section will be used to make modifications to the local storage. The web page, *ex04_storageEvents.html*, registers for the *storage* event. In the figure shown below, a new key, *core1*, with the value, *CS520*, is added to the local storage. The event is handled in the web page and the associated information is displayed as shown below.



In the next step, the key *core1* is updated with a new value, *CS601*. The change is propagated to the event handler that lists both the old value and the new value as shown below.



In the next step, the item with the key *core1* is removed from the local storage. The event handler shows the new entry with the new value as null and the existing value as the old value.



Storage Event Example – Simultaneous Updates (Optional)

The *storage* event can be utilized to propagate the changes made in one web page to all the other web pages opened from the same domain. The local storage example is now modified to handle the *storage* event. The *html* page of the web application is shown below.

ex05_localStorageSample.html

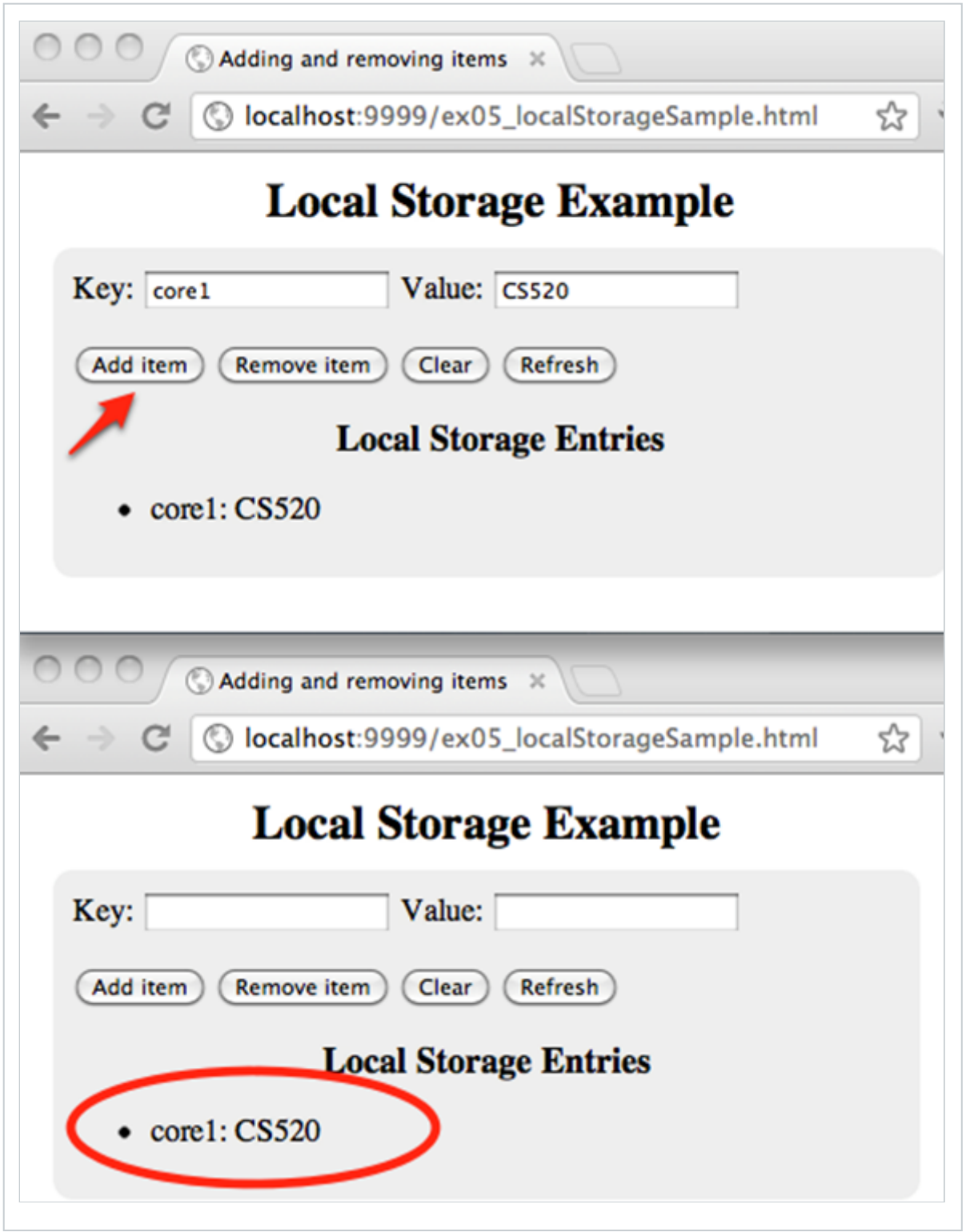
```
<!doctype html>
<html>
    <head>
        <title>Adding and removing items</title>
        <meta charset="utf-8">
        <script src="ex05_localStorageSample.js"></script>
        <link rel="stylesheet" href="html5.css">
    </head>
    <body>
      <header>
        <h2>Local Storage Example</h2>
      </header>

        <section>
            <article>
                Key:   <input type="text" id="key">
                Value: <input type="text" id="value">
                <p/>
                <button id="addButton">Add item</button>
                <button id="remButton">Remove item</button>
                <button id="clrButton">Clear</button>
                <button id="refreshButton">Refresh</button>

                <p/>
                <h3>Local Storage Entries</h3>

                <ul id="items"></ul>
            </article>
        </section>
    </body>
</html>
```

The associated *JavaScript* has an additional function registered as the event handler for the *storage* event as shown below. Whenever there is a change in the local storage, the event handler lists all the current entries in the *localStorage* in the web page.

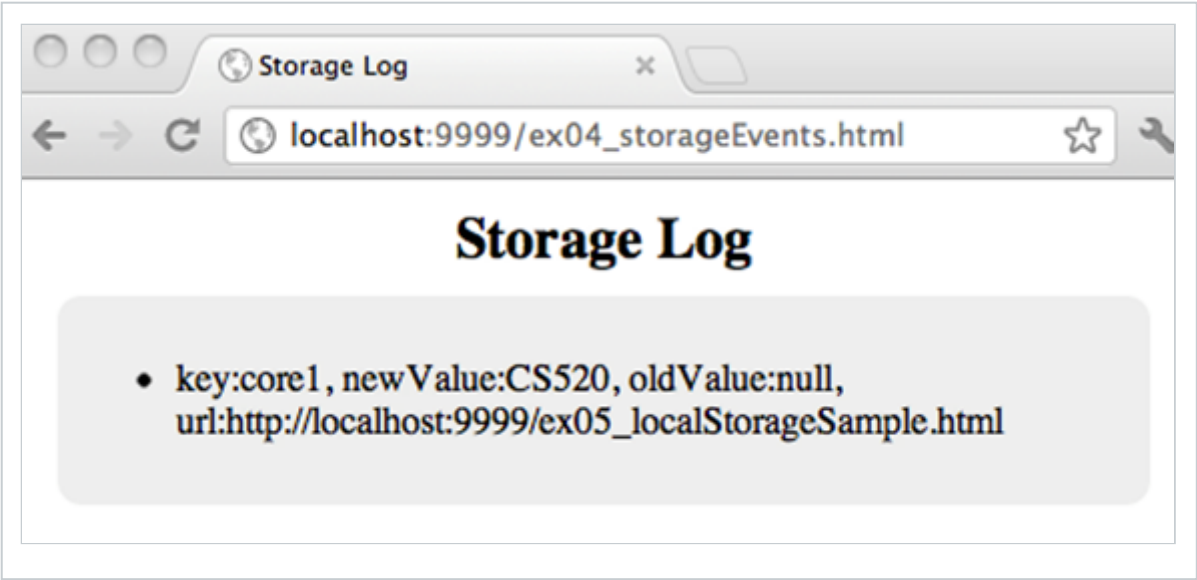
ex05_localStorageSample.js

```
45
46  function updateDOMItems() {
47      var itemsList = document.getElementById("items");
48      // Clear shown list
49      itemsList.innerHTML = '';
50      // Add current items to the list
51      for (key in window.localStorage) {
52          addItemToDOM(key, localStorage[key]);
53      }
54  }
55
56  function addItemToDOM(key, value) {
57      var items = document.getElementById("items");
58      // create a line item and add to the end of the list
59      var item = document.createElement("li");
60      item.innerHTML = key + ": " + value;
61
62      items.appendChild(item);
63  }
64
65  window.onstorage = function(e) {
66      updateDOMItems();
67  }
```

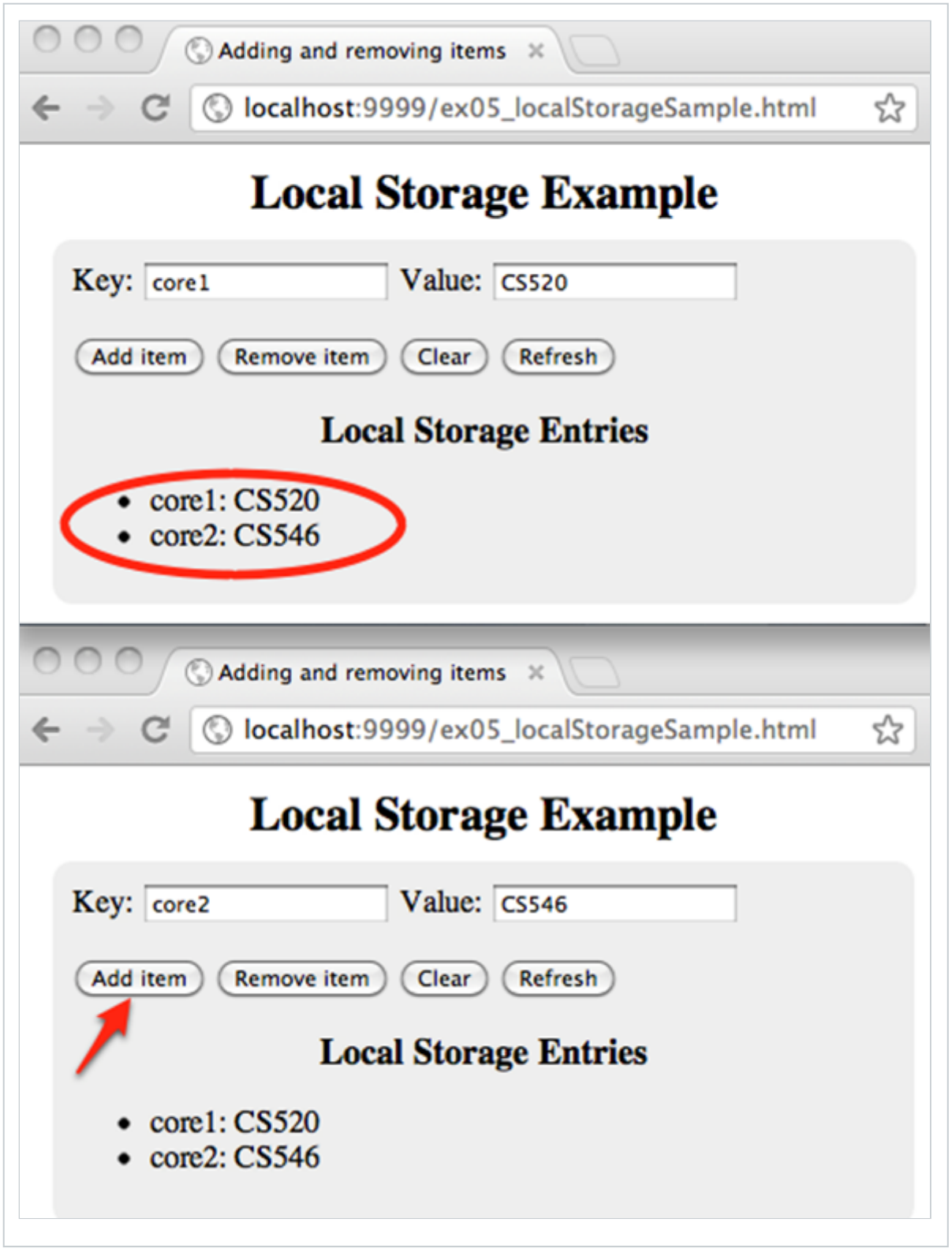
In order to demonstrate the simultaneous updates, open two browser windows of the above web application. In the first window, add a new entry to the local storage. The update will be immediately visible in the second window. Also, keep the *ex04_storageEvents.html* web page open in a new browser window.



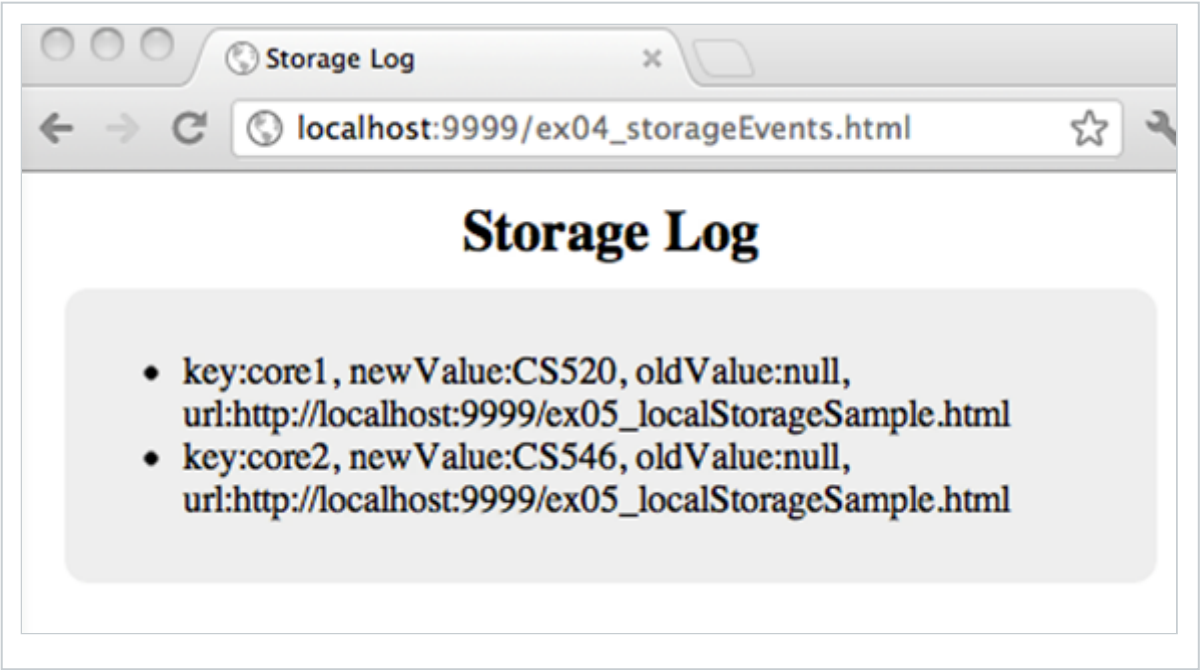
The log entry shows the changes to the *localStorage* as illustrated below.



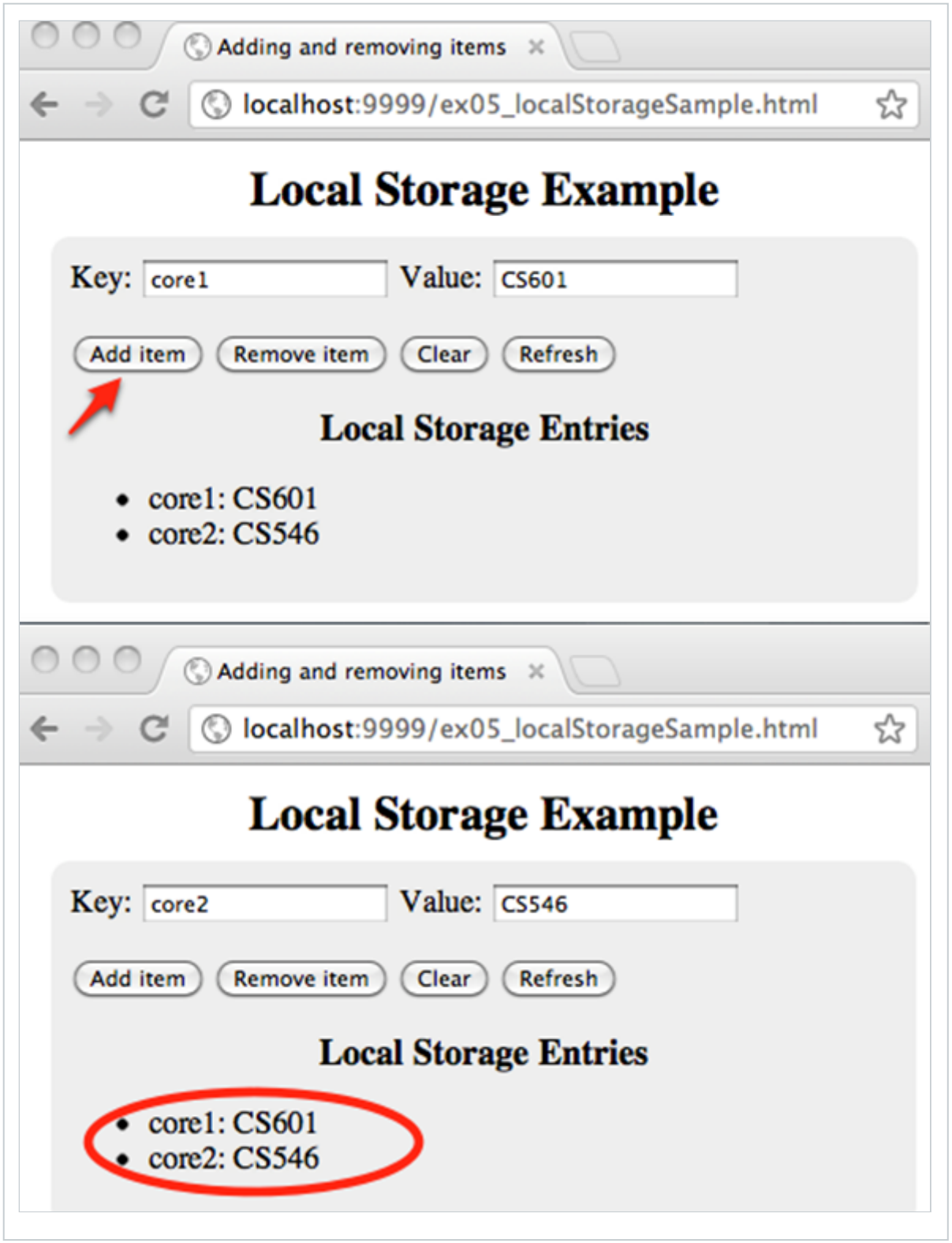
In the next step, add a new entry in the second browser window. The changes are immediately reflected in the first window as shown below.



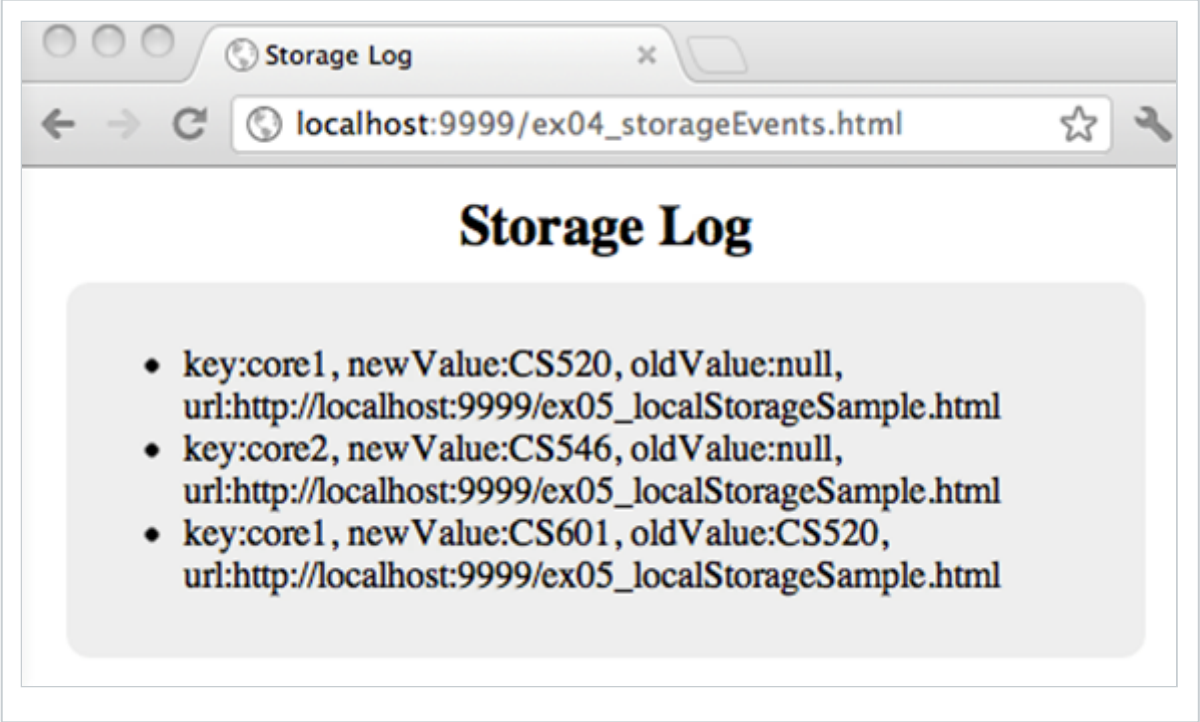
The storage log shown below lists the new entry as well.



In the next step, the value associated with the key, *core1*, is modified from *CS520* to *CS601* in the first window. Again, changes are immediately reflected in the second window.



The storage log lists the old value and the new value as well.



Storing Objects

In order to store JavaScript objects in the *sessionStorage* or the *localStorage*, the objects are first converted to strings in JSON format as follows.

```
window.localStorage.setItem(key, JSON.stringify(object));
```

When the value is retrieved using the getItem, the string representation is converted back to object notation as follows:

```
var myObject = JSON.parse(window.localStorage.getItem(key));
```

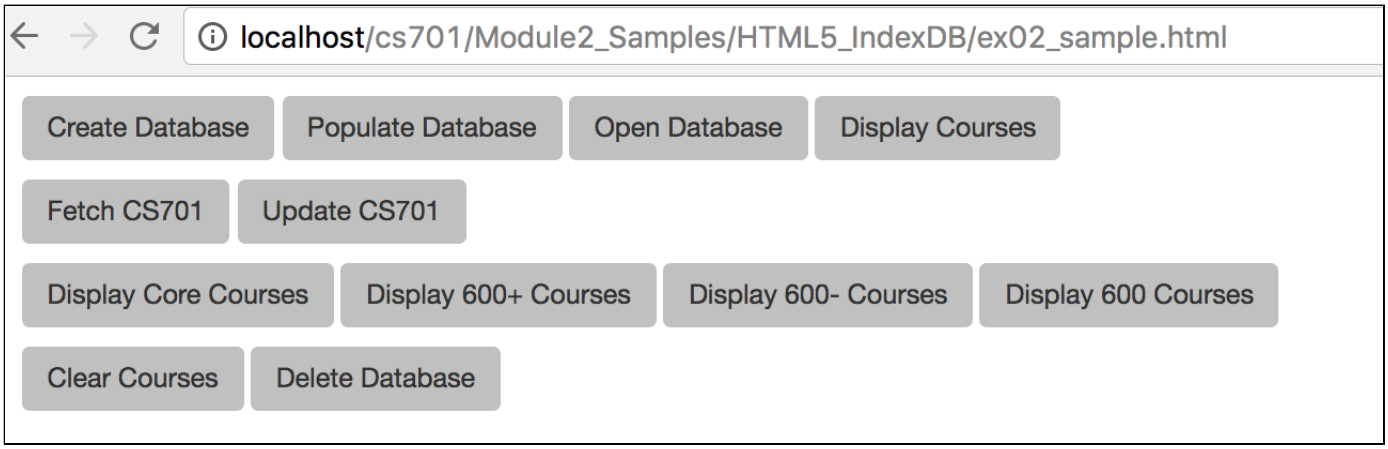
IndexedDB

The IndexedDB API allows client-side storage of structured data. IndexedDB is a transactional database system and unlike the traditional relational databases, it uses the JavaScript object-oriented database notation. The API also provides the indexing capability to allow high performance search of indexed data. The browser's support for IndexedDB functionality can be checked through the *indexedDB* property of the *window* object.

ex01_checkIndexedDBSupport.html

```
<!DOCTYPE html>
<html>
  <body>
    <script type="text/javascript">
      if (!window.indexedDB)
        document.write("IndexedDB support is not available.")
      else
        document.write("IndexedDB support is  available.")
    </script>
  </body>
</html>
```

The various features of the IndexedDB API are demonstrated using the following example. The application manipulates the course objects in the IndexedDB database. It has functionality for creating the database, populating the database with course objects, displaying the courses, fetching and updating a course, display courses using index features, and deleting the data from the database.

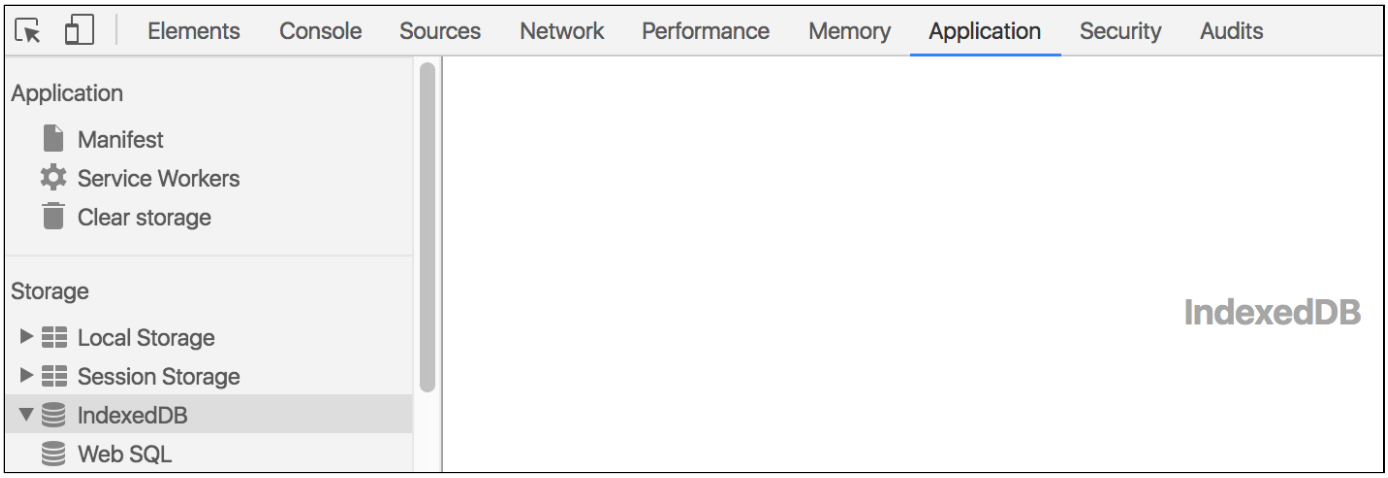


The HTML for the application shown below ties the various buttons to the respective functions in the associated JavaScript file.

ex02_sample.html

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>HTML 5 Application - IndexedDb</title>
  <script type="text/javascript" src="ex02_sample.js"></script>
  <link rel="stylesheet" href="bootstrap.css">
</head>
<body>
  <div class="container">
    <p>
    <div class="row">
      <button onclick="createDb()" class="btn">Create Database</button>
      <button onclick="populateDb()" class="btn">Populate Database</button>
      <button onclick="openDb()" class="btn">Open Database</button>
      <button onclick="displayDb()" class="btn">Display Courses</button>
    </div>
    <p>
    <div class="row">

      <button onclick="fetchCourse('cs701')" class="btn">Fetch CS701</button>
      <button onclick="updateCourse('cs701')" class="btn">Update CS701</button>
    </div>
    <p>
    <div class="row">

      <button onclick="displayCoreCourses()" class="btn">Display Core Courses</button>
      <button onclick="displayCourses1()" class="btn">Display 600+ Courses</button>
      <button onclick="displayCourses2()" class="btn">Display 600- Courses</button>
      <button onclick="displayCourses3()" class="btn">Display 600 Courses</button>
    </div>
    <p>
    <div class="row">

      <button onclick="clearAllCourses()" class="btn">Clear Courses</button>
      <button onclick="deleteDb()" class="btn">Delete Database</button>
    </div>
    <p>
      <div id="result"></div>
</body>
</html>
```

The initial view of the IndexedDB using the Chrome browser is shown below. No databases exist in the IndexedDB storage.



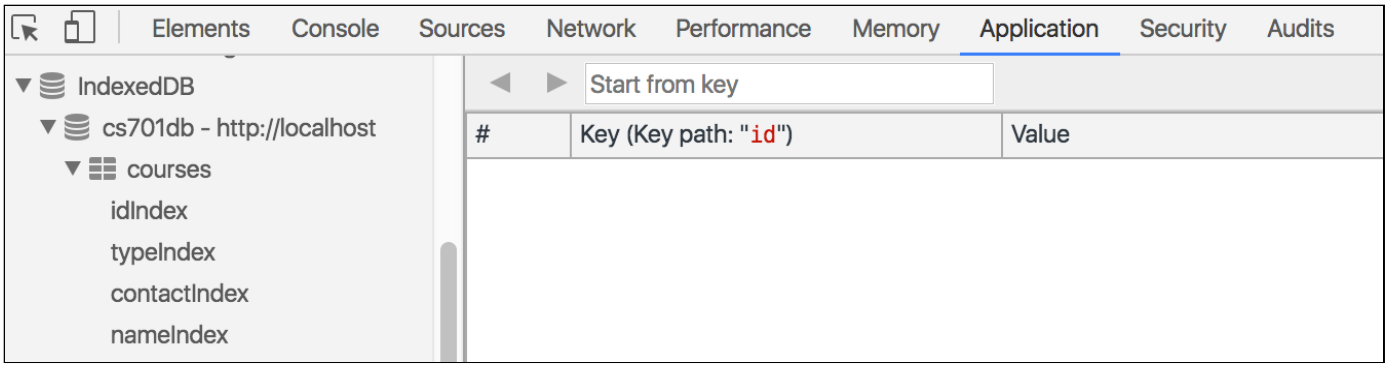
All operations using the IndexedDB API are done asynchronously so that the application never blocks for the actions to complete. The appropriate callbacks are used to check the results or errors. In this application, a database named *cs701db* is created. Within the database, an object store named *courses* will be used to store the course objects. Each course is a JavaScript object with the properties *id, name*, *type*, and *contact*.

The function *createDb* shown below creates a request to open the specified database for the specified version. If the database does not exist, or if the specified version is a new version, the *upgradeneeded* event is triggered and the associated event handler is called. In this case, the object store is created using the course *id* property as the key. Indexes are also created for the course properties to facilitate indexing. The *success* event is fired if the operation is successful. In this case, the reference to the database is stored in the variable *db*. This variable is used in the rest of the functions.

ex02_sample.js

```
1   const DB_NAME = 'cs701db';
2   const DB_VERSION = 1;
3   const DB_STORE_NAME = 'courses';
4
5   var db;
6
7   function createDb() {
8     console.log("createDb ...");
9
10    var request = window.indexedDB.open(DB_NAME, DB_VERSION);
11
12    request.onsuccess = function (evt) {
13      db = request.result;
14      console.log("createDb DONE");
15      logMessage("createDb DONE");
16    };
17
18    request.onerror = function (evt) {
19      console.error("createDb ERROR:", evt.target.errorCode);
20      logMessage("createDb ERROR: " + evt.target.errorCode);
21    };
22
23    request.onupgradeneeded = function (evt) {
24      console.log("createDb.onupgradeneeded ...");
25
26      var store = evt.currentTarget.result.createObjectStore(
27        DB_STORE_NAME, { keyPath: 'id'});
28
29      store.createIndex('idIndex',
30                        'id',       { unique: true });
31      store.createIndex('nameIndex',
32                        'name',     { unique: false });
33      store.createIndex('typeIndex',
34                        'type',     { unique: false });
35      store.createIndex('contactIndex',
36                        'contact',  { unique: false });
37    };
38  }
```

The schema of the *courses* object store within the *cs701db* database is shown below.



The following data is used for populating the database.

ex02_sample.js

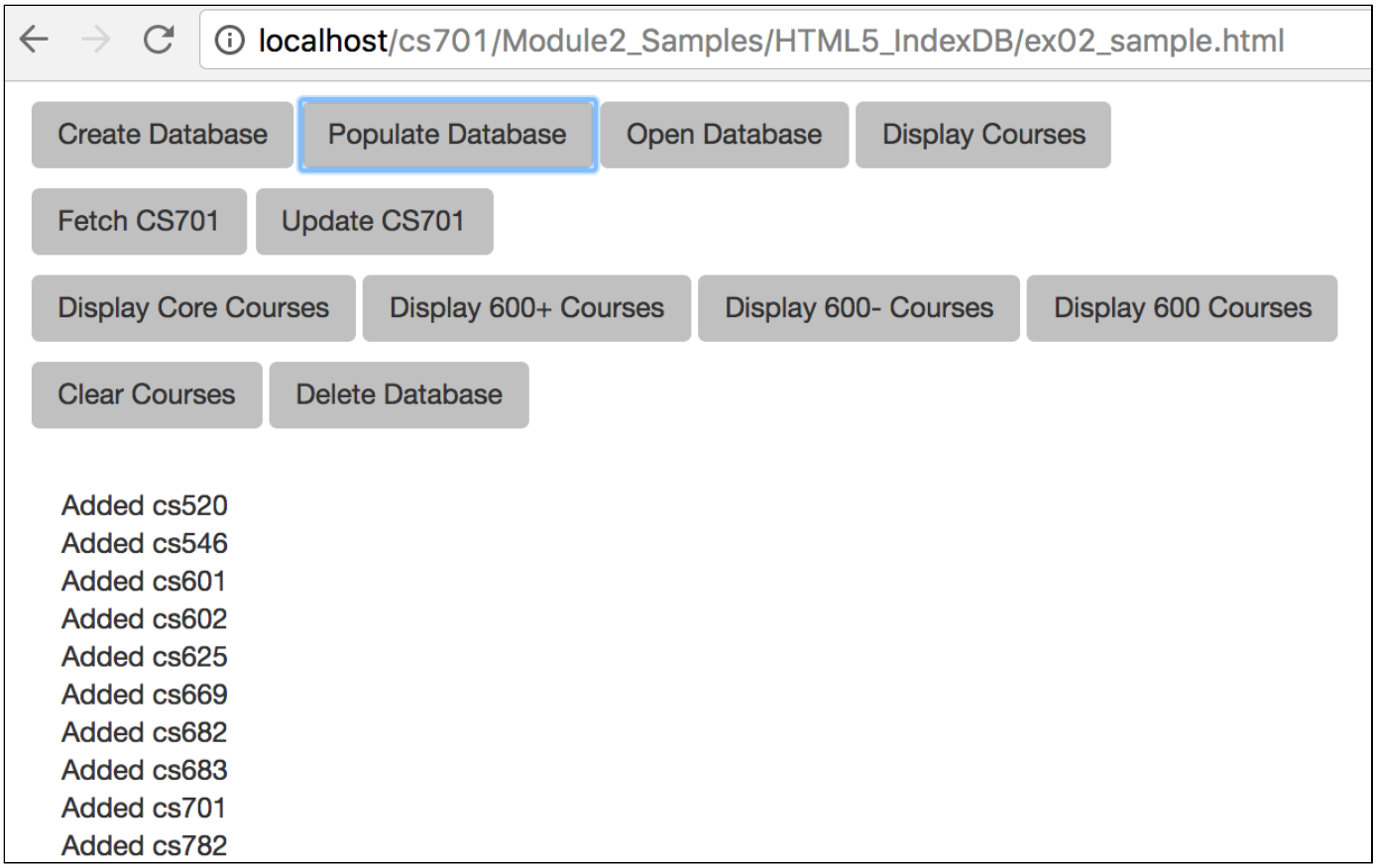
```
40    var courses = [
41     {"id": "cs520", "type": "core", "contact": "kalathur",
42          "name": "Information Structures"},
43     {"id": "cs546", "type": "elective", "contact": "temkin",
44          "name": "Quantitative Methods for Information Systems"},
45     {"id": "cs601", "type": "elective", "contact": "kalathur",
46          "name": "Web Application Development"},
47     {"id": "cs602", "type": "elective", "contact": "kalathur",
48          "name": "Server-Side Web Development"},
49     {"id": "cs625", "type": "core", "contact": "day",
50          "name": "Business Data Comm and Networks"},
51     {"id": "cs669", "type": "core", "contact": "rschudy",
52          "name": "Database Design and Impl for Business"},
53     {"id": "cs682", "type": "core", "contact": "kanabar",
54          "name": "Information Systems Analysis and Design"},
55     {"id": "cs683", "type": "elective", "contact": "braude",
56          "name": "Mobile Application Development"},
57    {"id": "cs701", "type": "elective", "contact": "kalathur",
58          "name": "Rich Internet Appl Development"},
59     {"id": "cs782", "type": "core", "contact": "kanabar",
60          "name": "IT Strategy and Management"}
61  ];
```

The *populateDb* function creates the *readwrite* transaction object using the reference to the database connection and acquire the reference to the object store using the transaction. For each course in the data, the asynchronous *add* method on the object store stores new records to the object store. The *success* event is fired when the record is successfully added to the store.

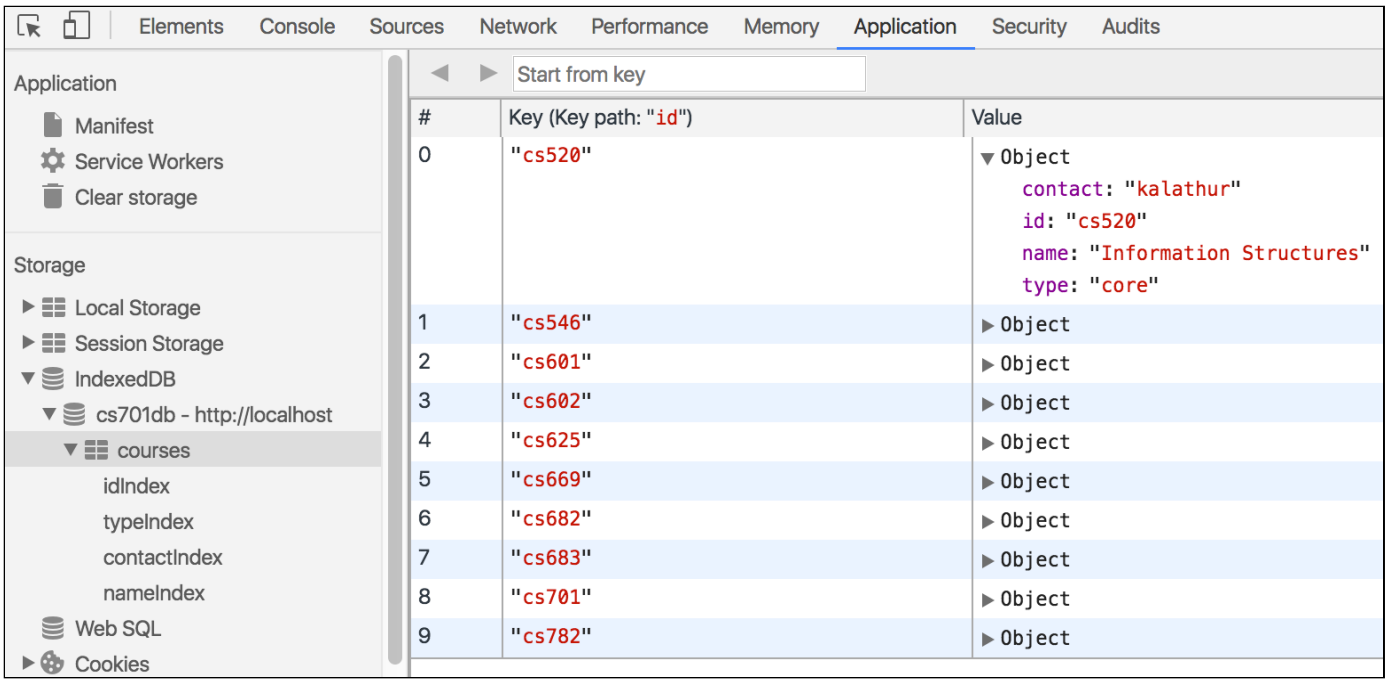
ex02_sample.js

```
63  function populateDb() {
64
65    var result = document.getElementById("result");
66    result.innerHTML = "";
67
68    if (!db)
69      return;
70
71    var transaction = db.transaction(DB_STORE_NAME, "readwrite");
72    var store = transaction.objectStore(DB_STORE_NAME);
73
74    courses.forEach(function (course) {
75
76      var request = store.add({
77        "id":       course.id,
78        "type":     course.type,
79        "contact":  course.contact,
80        "name":     course.name
81      });
82
83      request.onsuccess = function(e) {
84        console.log("Added", course.id);
85        result.innerHTML = result.innerHTML + "<br/>" +
86                                    "Added " +  course.id;
87      };
88
89      request.onerror = function(e) {
90        console.log("Error", e.target.error.message);
91        result.innerHTML = result.innerHTML + "<br/>" +
92          "Error " +  e.target.error.message
93      };
94    });
95
96  }
```

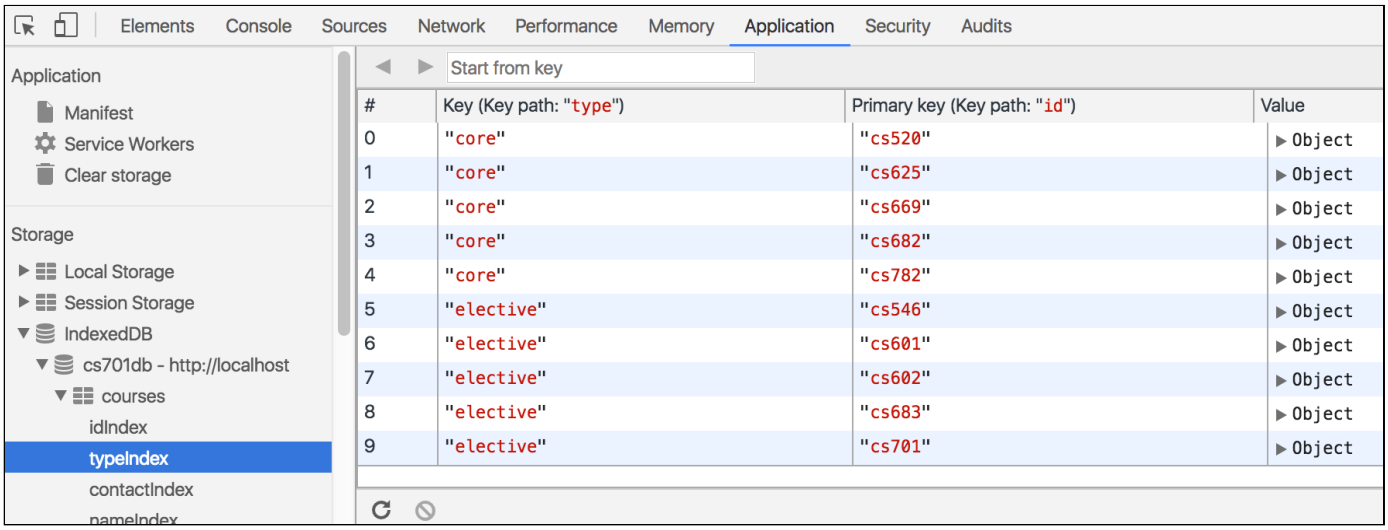
The output of the application when the button is clicked is shown below.



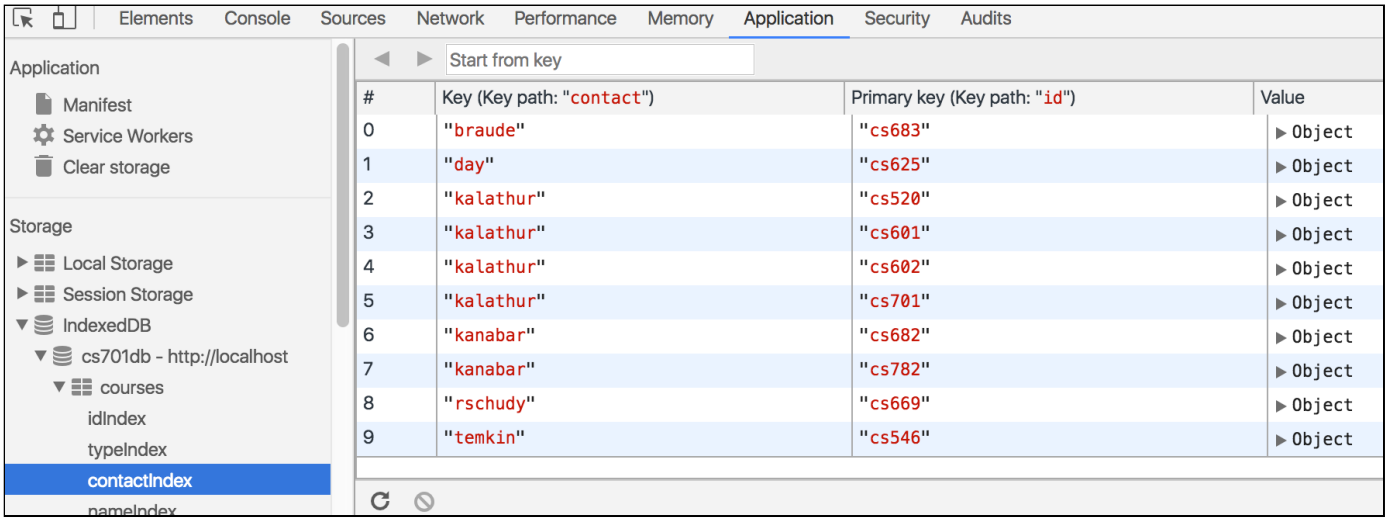
The data in the browser's storage appears as shown below. The *key* is the *id* property of each object. Each object has the properties of the corresponding course.



The index view of the data is shown below. The *type* property of the course data has two values: *core* and *elective*. The objects associated with the corresponding *type* are shown below.



Similarly, the objects associated with the corresponding *contact* are shown below.



The *openDb* function is useful to get a reference to the database connection when the application is refreshed. The *success* event handlers stores the result of the *open* result in the variable *db*.

ex02_sample.js

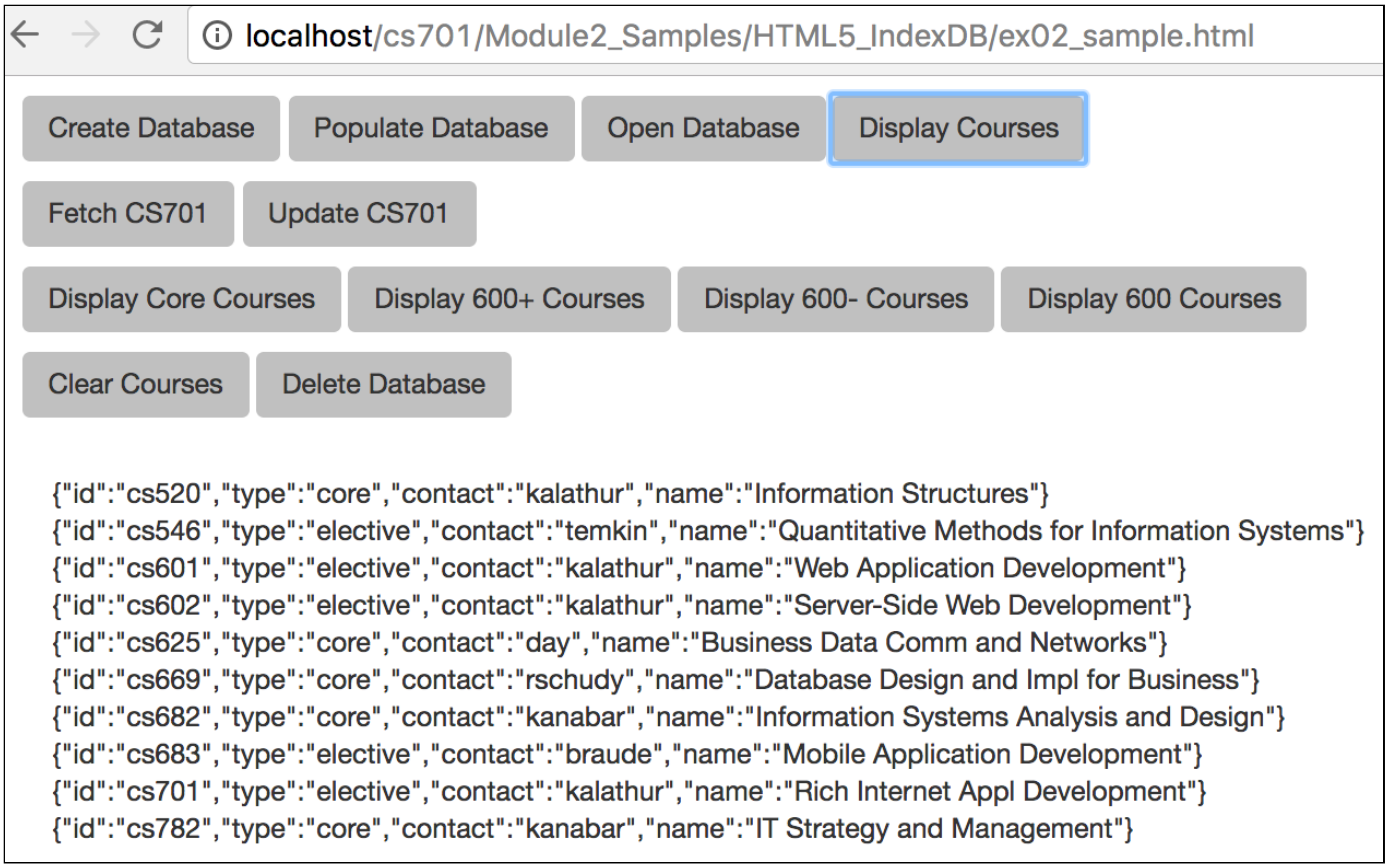
```
97
98    function openDb() {
99      console.log("openDb ...");
100     var openRequest = window.indexedDB.open(DB_NAME);
101
102     openRequest.onerror = function(e) {
103       console.log("Database error: " + e.target.errorCode);
104     };
105
106     openRequest.onsuccess = function(event) {
107       db = openRequest.result;
108     };
109
110   }
```

The *displayDb* function gets the reference to the object store through the transaction object. The default transaction mode is the *read* mode. The *openCursor* method is used to iterate over the data. The default direction travel mode is *next*. The *value* property of the cursor object returns each course object.

ex02_sample.js

```
112   function displayDb() {
113     var result = document.getElementById("result");
114     result.innerHTML = "";
115
116     if (db) {
117       try {
118         var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
119
120         var request = store.openCursor();
121
122         request.onsuccess = function(evt) {
123           var cursor = evt.target.result;
124           if (cursor) {
125             var course = cursor.value;
126             var jsonStr = JSON.stringify(course);
127             result.innerHTML = result.innerHTML + "<br/>" + jsonStr;
128             cursor.continue();
129           }
130         };
131
132         request.onerror = function (evt) {
133           console.error("displayDB ERROR:", evt.target.errorCode);
134         };
135
136       } catch (e) {
137         console.log(e);
138         logMessage(e);
139       }
140     }
141
142   }
```

The output of the application showing the JSON representation of each course in the object store is shown below.

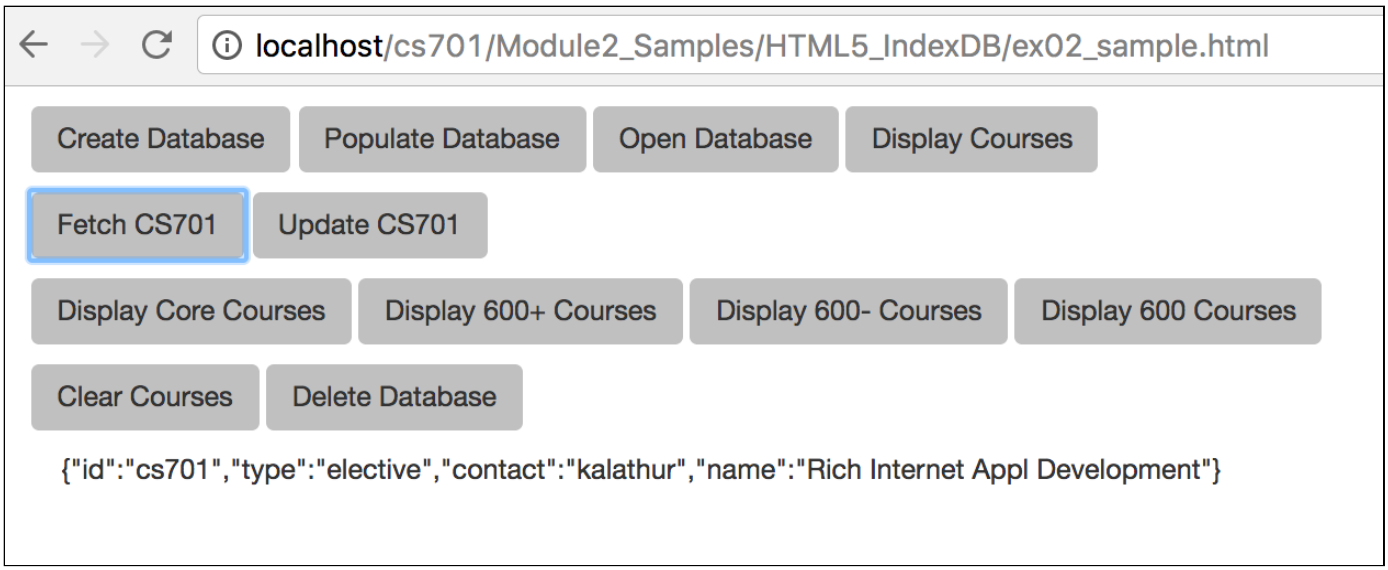


The function *fetchCourse* uses the *get* method on the object store to access the object with the specified key. The success event handler is invoked when the result is available. If the specified key value exists in the data, the corresponding object is the returned through the *result* property of the event's target object.

ex02_sample.js

```
144   function fetchCourse(courseId) {
145   try {
146     var result = document.getElementById("result");
147     result.innerHTML = "";
148
149     if (db) {
150       var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
151       store.get(courseId).onsuccess = function(event) {
152         var course = event.target.result;
153         if (course == null) {
154           result.innerHTML = "Course not found";
155         }
156         else {
157           var jsonStr = JSON.stringify(course);
158           result.innerHTML = jsonStr;
159         }
160       };
161     }
162   }
163   catch(e){
164     console.log(e);
165   }
166 }
```

The output of the application fetching the course with *id* value of *cs701* is shown below.

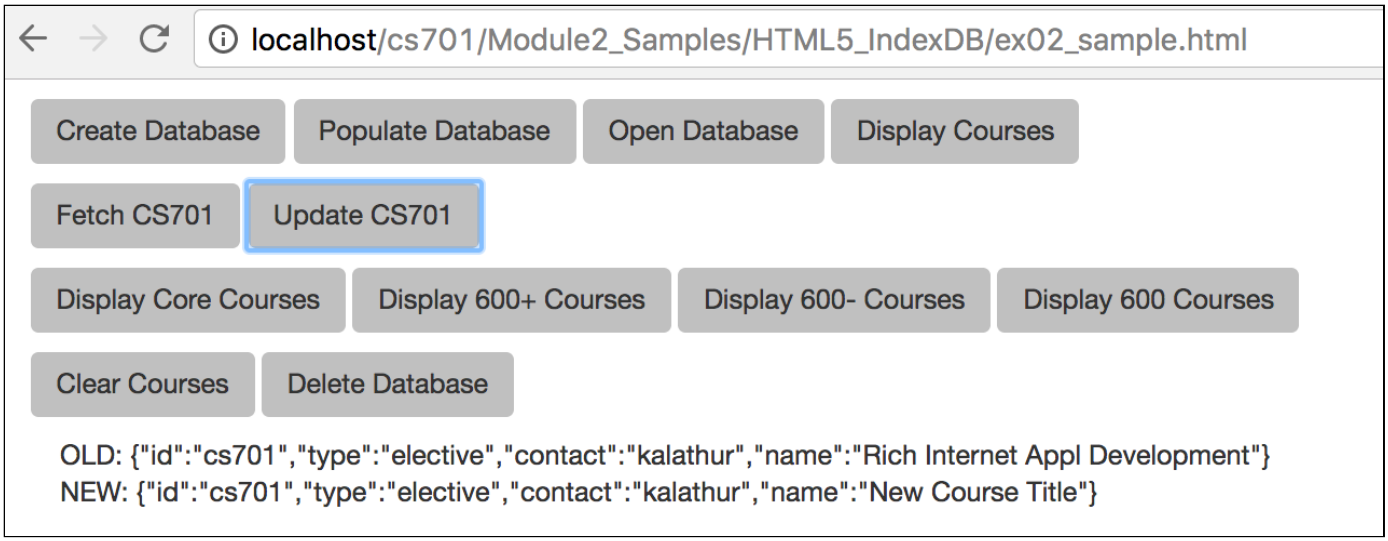


For updating an object, the transaction is created in *readwrite* mode and using the reference to the object store, the *get* method is first used to access the object. After the course object is modified, the *put* method is used to replace the current object with the new object of the same key.

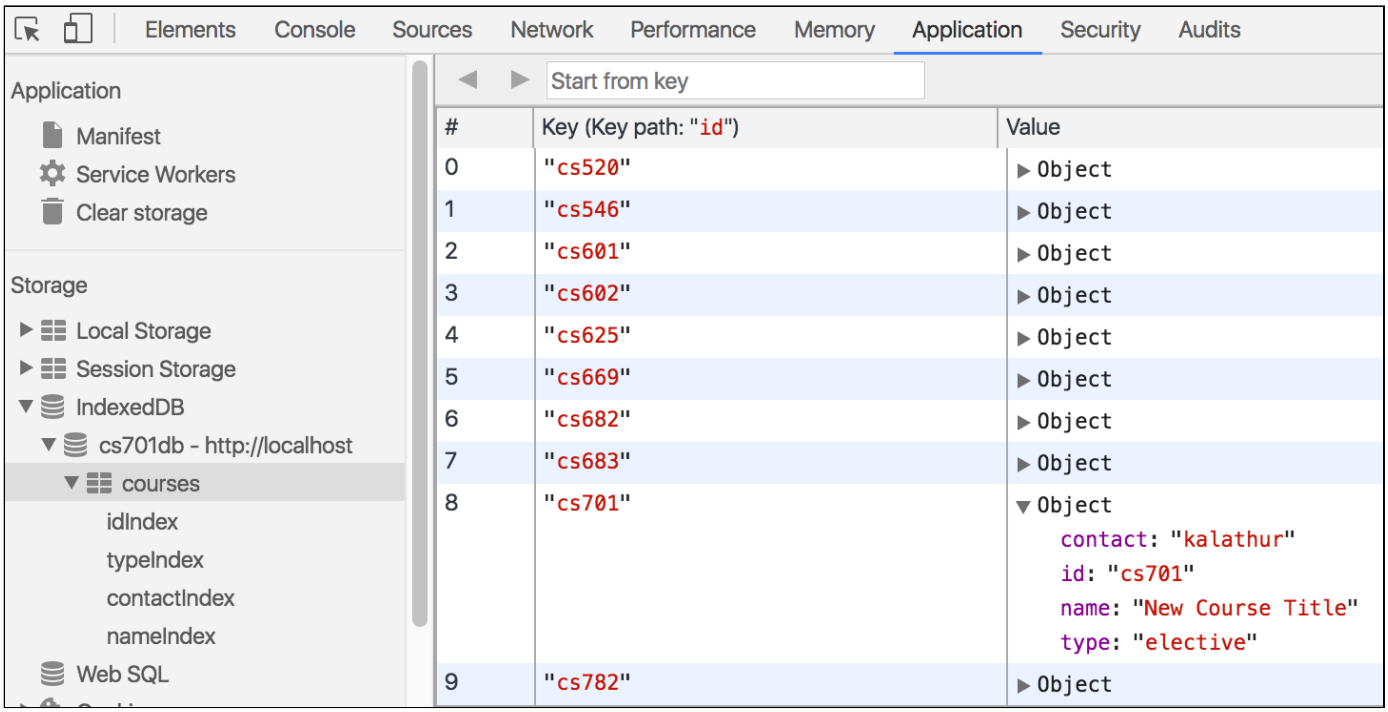
ex02_sample.js

```
function updateCourse(courseId) {
  try {
    var result = document.getElementById("result");
    result.innerHTML = "";

    var jsonStr;
    var course;

    if (db) {

      var transaction = db.transaction(DB_STORE_NAME, "readwrite");
      var store = transaction.objectStore(DB_STORE_NAME);

      store.get(courseId).onsuccess = function(event) {
        course = event.target.result;
        // current value
        jsonStr = "OLD: " + JSON.stringify(course);
        result.innerHTML = jsonStr;

        // update record
        course.name = "New Course Title";

        var request = store.put(course);

        request.onsuccess = function(e) {
          console.log("Updated");
        };

        request.onerror = function(e) {
          console.log(e.value);
        };

        // fetch record again
        store.get(courseId).onsuccess = function(event) {
          course = event.target.result;
          jsonStr = "<br/>NEW: " + JSON.stringify(course);
          result.innerHTML = result.innerHTML  + jsonStr;
        };
      };
    }
  }
  catch(e){
    console.log(e);
  }
```

The result of updating the *cs701* course with the new title is shown below.



The changes are reflected in the storage as shown below.

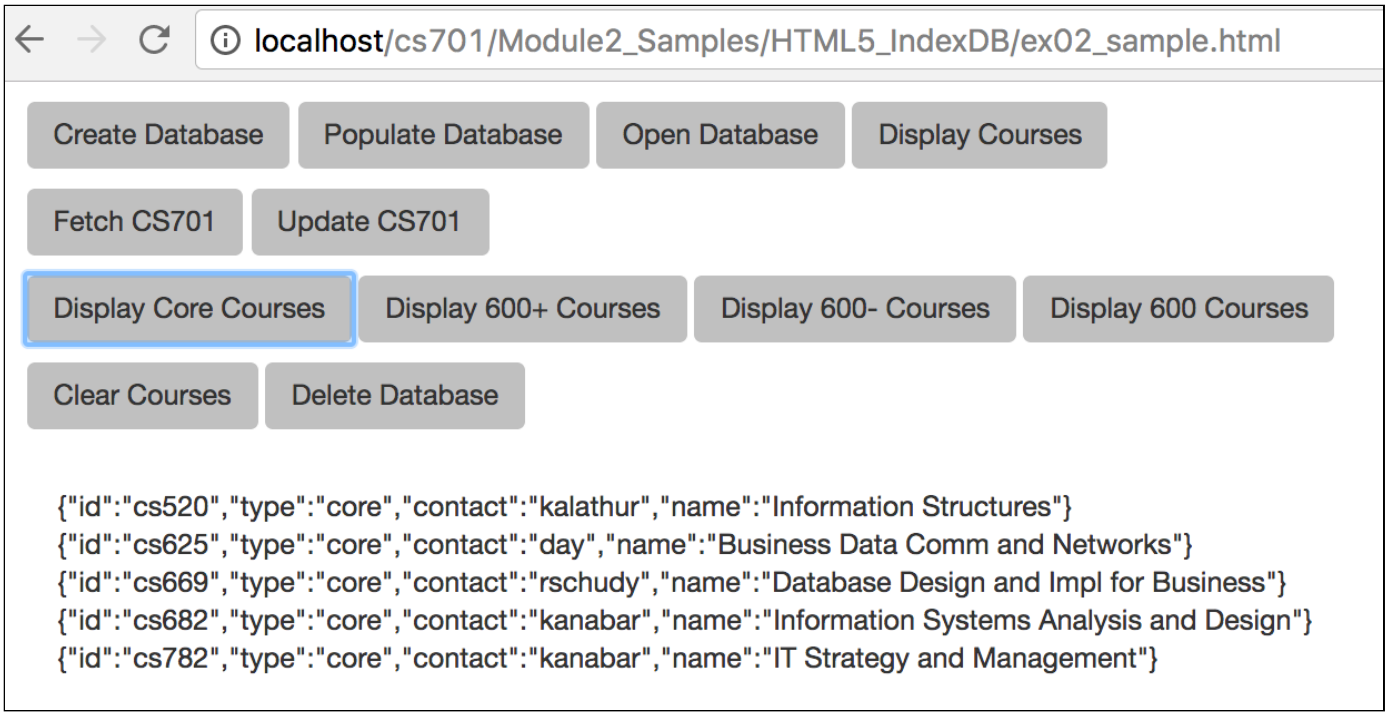


The following functions demonstrate the index searching capabilities of the *IndexedDB* API. In the following function to display the core courses, the *typeIndex* is first accessed using the *index* method of the object sore. The cursor is opened using the *index* object with the specified key range with the only value as *core*. The cursor is now iterated to display all the core courses.

ex02_sample.js

```
215
216   function displayCoreCourses() {
217     try {
218       var result = document.getElementById("result");
219       result.innerHTML = "";
220
221       if (db) {
222
223         var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
224         var index = store.index("typeIndex");
225         var range = IDBKeyRange.only("core");
226
227         index.openCursor(range).onsuccess = function(evt) {
228             var cursor = evt.target.result;
229             if (cursor) {
230               var course = cursor.value;
231               var jsonStr = JSON.stringify(course);
232               result.innerHTML = result.innerHTML + "<br/>" + jsonStr;
233               cursor.continue();
234             }
235         };
236       }
237     }
238     catch(e){
239       console.log(e);
240       logMessage(e);
241     }
242   }
```

The output of the application is shown below.

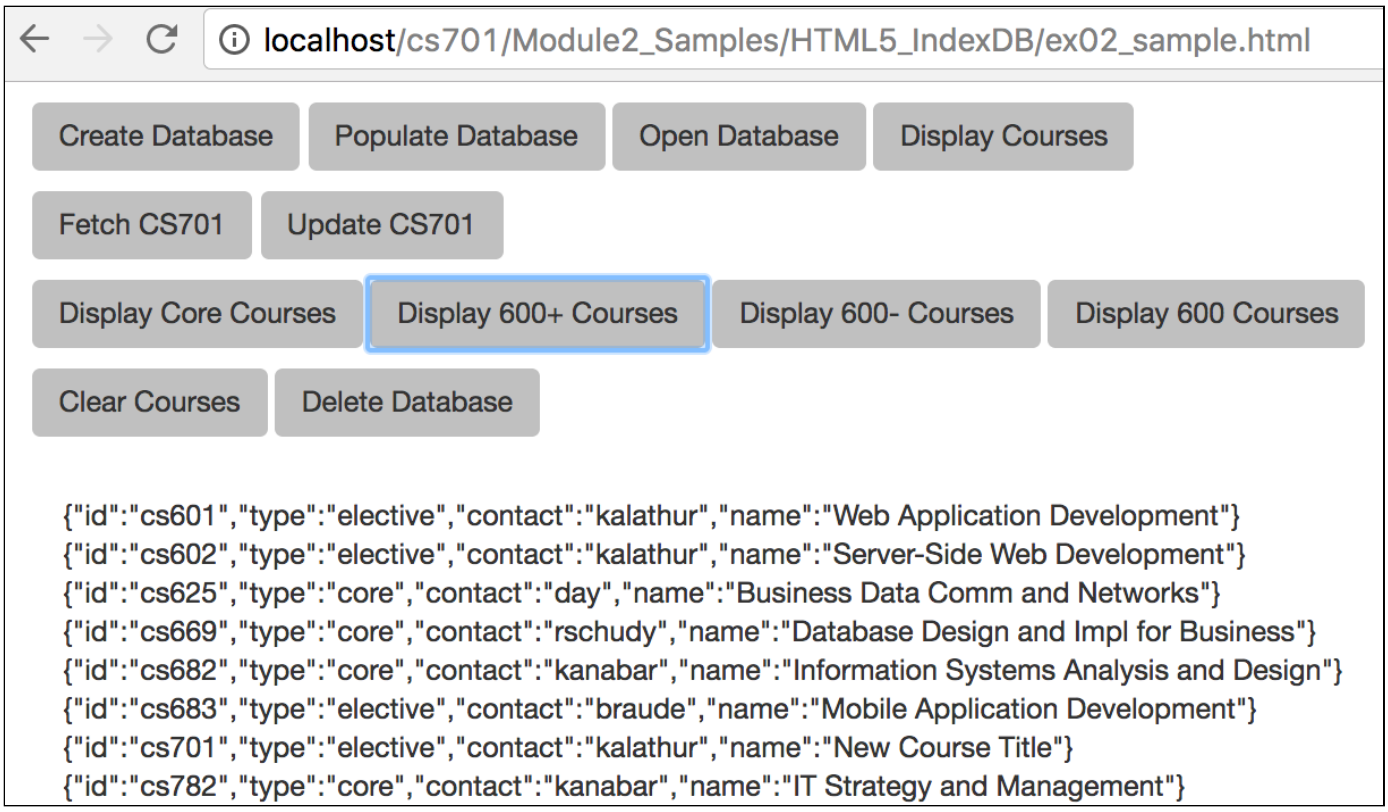


The following function uses the *idIndex* of the object store and species only the lower bound value for the index. All the results with the key value greater than or equal to specified value are included in the results.

ex02_sample.js

```
243
244   function displayCourses1() {
245     try {
246       var result = document.getElementById("result");
247       result.innerHTML = "";
248
249       if (db) {
250         var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
251         var index = store.index("idIndex");
252
253         var range = IDBKeyRange.lowerBound("cs600");
254
255         index.openCursor(range).onsuccess = function(evt) {
256             var cursor = evt.target.result;
257             if (cursor) {
258               var course = cursor.value;
259               var jsonStr = JSON.stringify(course);
260               result.innerHTML = result.innerHTML + "<br/>" + jsonStr;
261               cursor.continue();
262             }
263         };
264       }
265     }
266     catch(e){
267       console.log(e);
268     }
269   }
```

The output of the application is shown below.

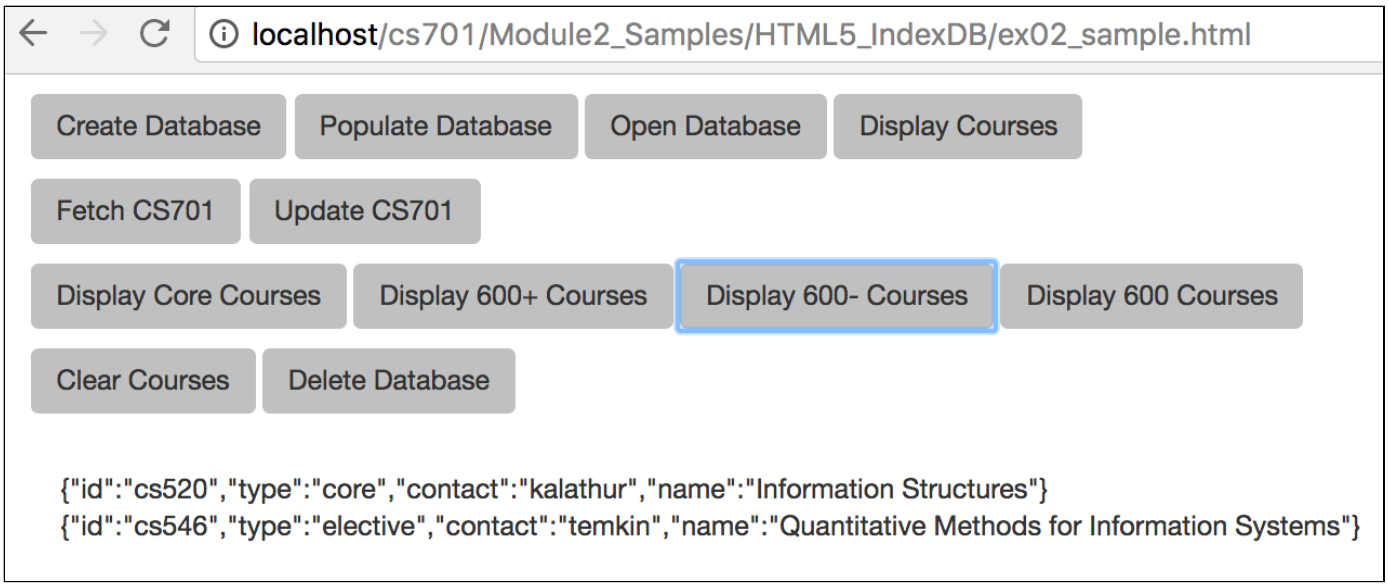


In the following function, only the upper bound is specified for the key range. With the *true* option, all keys less than the specified value are included in the results.

ex02_sample.js

```
271   function displayCourses2() {
272     try {
273       var result = document.getElementById("result");
274       result.innerHTML = "";
275
276       if (db) {
277         var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
278         var index = store.index("idIndex");
279
280         var range = IDBKeyRange.upperBound("cs600", true);
281
282         index.openCursor(range).onsuccess = function(evt) {
283             var cursor = evt.target.result;
284             if (cursor) {
285               var course = cursor.value;
286               var jsonStr = JSON.stringify(course);
287               result.innerHTML = result.innerHTML + "<br/>" + jsonStr;
288               cursor.continue();
289             }
290         };
291       }
292     }
293     catch(e){
294       console.log(e);
295     }
296   }
```

The application displays are courses less than *cs600* as shown below.

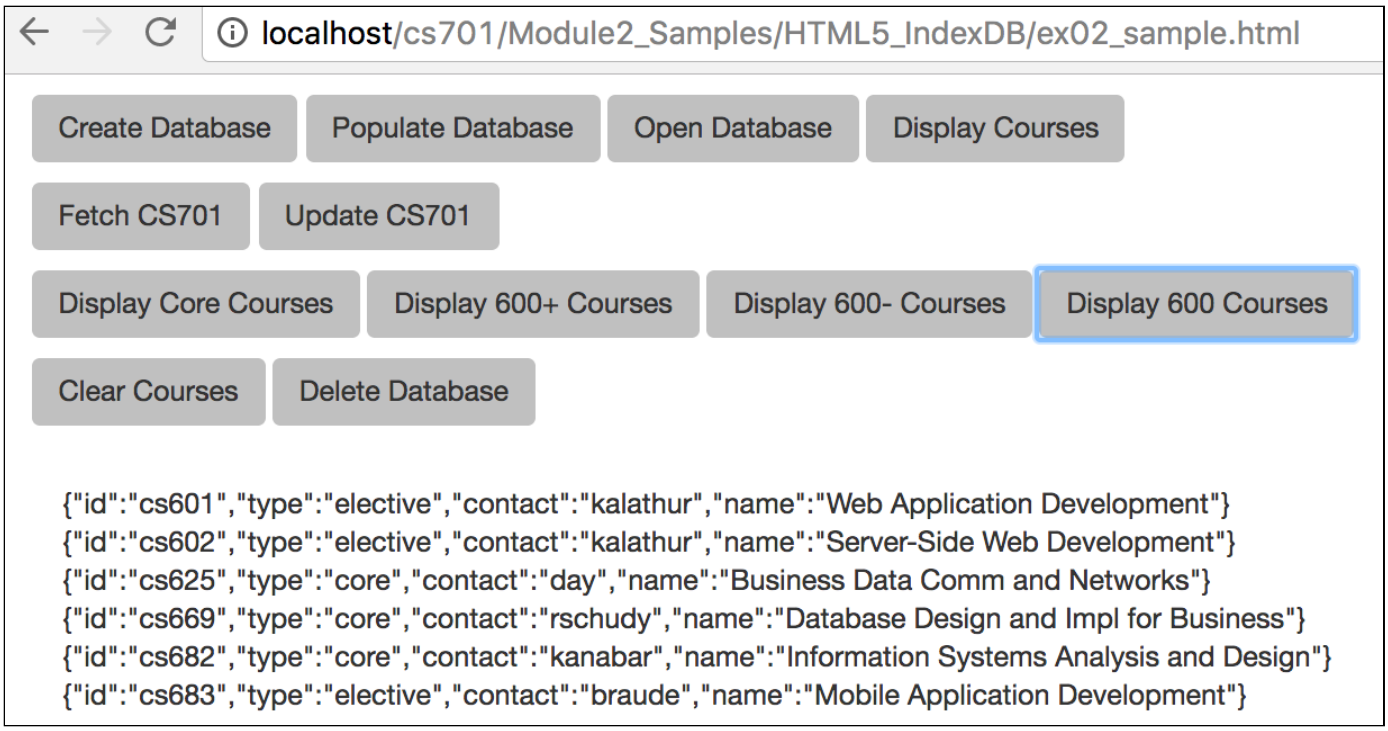


The following function uses the *bound* method is specify both the lower the lower and upper bounds. With the *false* and *true* options, all courses greater than or equal to *cs600* and less than *cs700* are included in the results.

ex02_sample.js

```
298   function displayCourses3() {
299     try {
300       var result = document.getElementById("result");
301       result.innerHTML = "";
302
303       if (db) {
304         var store = db.transaction(DB_STORE_NAME).objectStore(DB_STORE_NAME);
305         var index = store.index("idIndex");
306
307         var range = IDBKeyRange.bound("cs600", "cs700", false, true);
308
309         index.openCursor(range).onsuccess = function(evt) {
310             var cursor = evt.target.result;
311             if (cursor) {
312               var course = cursor.value;
313               var jsonStr = JSON.stringify(course);
314               result.innerHTML = result.innerHTML + "<br/>" + jsonStr;
315               cursor.continue();
316             }
317         };
318       }
319     }
320     catch(e){
321       console.log(e);
322     }
323   }
```

The output of the application showing all 600 level courses is shown below.



The following list shows all the options of the bounds on the index values.

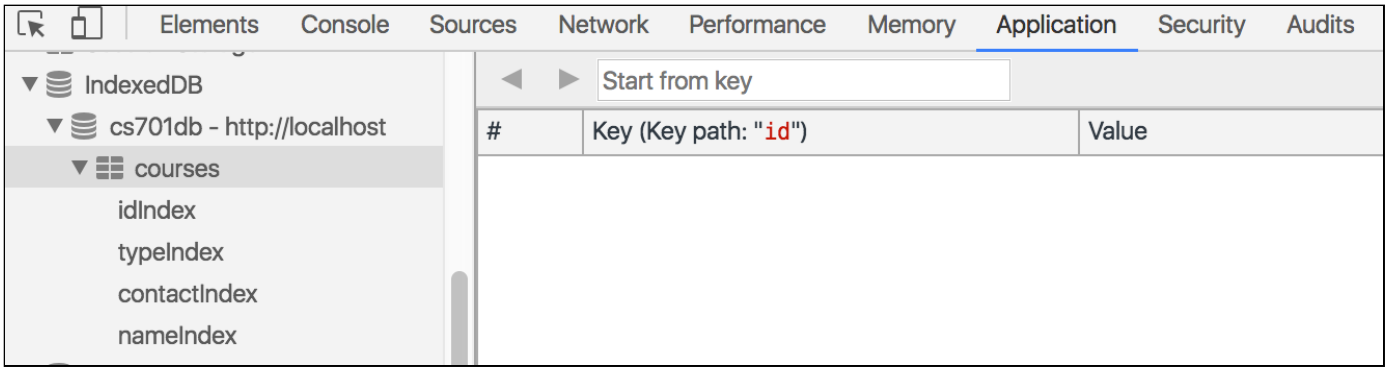
- IDBKeyRange.upperBound(x) All keys <= x
- IDBKeyRange.upperBound(x, true) All keys < x
- IDBKeyRange.lowerBound(y) All keys >= y
- IDBKeyRange. lowerBound (y, true) All keys > y
- IDBKeyRange.bound(x, y) All keys >= x and <= y
- IDBKeyRange.bound(x, y, true, true) All keys > x and < y
- IDBKeyRange.bound (x, y, true, false) All keys > x and <= y
- IDBKeyRange.bound (x, y, false, true) All keys >= x and < y
- IDBKeyRange.only(z) Key with value z

The following function deletes all the records in the object store using the *clear* method.

ex02_sample.js

```
325   function clearAllCourses() {
326   try {
327
328     if (db) {
329
330       var transaction = db.transaction(DB_STORE_NAME, "readwrite");
331       var store = transaction.objectStore(DB_STORE_NAME);
332
333       store.clear().onsuccess = function(event) {
334         logMessage("Courses object store data cleared");
335       };
336     }
337   }
338   catch(e){
339     console.log(e);
340     logMessage(e);
341   }
342 }
```

The resulting storage with no data is shown below.



The database itself can be marked for deletion as shown below.

ex02_sample.js

```
344
345   function deleteDb() {
346     console.log("deleteDb ...");
347     var request = window.indexedDB.deleteDatabase(DB_NAME);
348
349
350     request.onsuccess = function (evt) {
351       db = undefined;
352       console.log("deleteDb DONE");
353       logMessage("deleteDb DONE");
354     };
355
356     request.onerror = function (evt) {
357       console.error("deleteDB ERROR:", evt.target.errorCode);
358       logMessage("deleteDB ERROR:" + evt.target.errorCode);
359     };
360   }
```

The helper function *logMessage* shows results in the application.

ex02_sample.js

```
362   function logMessage(msg) {
363       var result = document.getElementById("result");
364       result.innerHTML = '<b>' + msg + '<b>';
365   }
```

File API

Introduction

The File API allows the manipulation of file objects in web applications. The API includes the following interfaces:

- FileList – represents an array of selected files from the user's file system. The user interface for selection can be via the *input* element or through drag and drop.
- File – provides read-only information about the attributes of a file (name, type, size, and when it was last modified)
- Blob – represents the raw binary data and allows access to ranges of bytes
- FileReader – provides methods to read a *File* or a *Blob.*

Browser Support

The browser's support for the File API can be checked as shown below. The test may be narrowed to include only the features required for the application.

ex01_checkFileSupport.html

```
<!DOCTYPE html>
<html lang="en">
  <body>
    <script type="text/javascript">
        if (window.File && window.FileList &&
                window.FileReader && window.Blob)
            document.write("File System support is available.")
        else
            document.write("File System support is not available.")
    </script>
  </body>
</html>
```

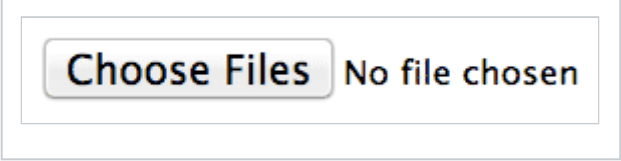
Selecting Files

The *input* element with its *type* as *file* may be used to browse and select files from the user's file system. The *multiple* attribute of the *input* element allows the user to select multiple files. The following *html* is used for the file selection.

ex02_selectingFiles.html

```
<!DOCTYPE html>
<html lang="en">
   <head>
        <title>Selecting Files</title>
        <script src="ex02_selectingFiles.js"></script>
    </head>
  <body>
    <input type="file" id="myfiles"
        name="myfiles[]" multiple />
    <output id="list"></output>
  </body>
</html>
```

The above *html* is rendered as shown below.



The associated *JavaScript* code is shown below. The event handle is fired whenever the user makes a new selection using the *input* element.

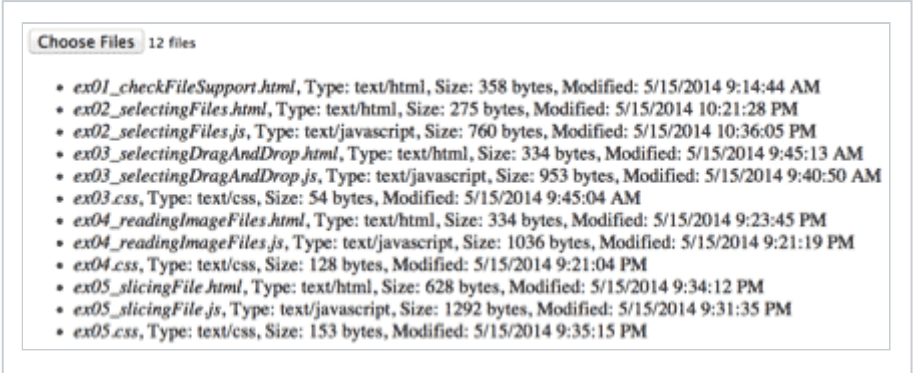
ex02_selectingFiles.js

```
window.onload = init;

function init() {
    var elem = document.getElementById('myfiles');
    elem.addEventListener('change',
            handleFileSelections, false);
}
```

In the event handler, the *files* property of the *input* element provides the *FileList* object representing the selections made by the user. Each *item* in the *FileList* may be accessed using the *index* or through the *item* get function. The properties associated with each *File* object are accessed as shown below.

```
function handleFileSelections(evt) {
    // Get the FileList object
    var files = evt.target.files;
    var output = '';

    // Select each File and its attributes
    for (var i = 0; i < files.length; i++) {
      var f = files.item(i);
      output += '<li><em>' + f.name + '</em>';
      output += ', Type: ' + f.type;
      output += ', Size: ' + f.size + ' bytes';
      output += ', Modified: ' +
        f.lastModifiedDate.toLocaleString() + '</li>';
    }
    // Show the selections
    document.getElementById('list').innerHTML =
      '<ul>' + output + '</ul>';
}
```

A sample output with the example files selected is shown below.



Drag and Drop Selection

Drag and drop provides another alternative for the user to drag the file selections from the desktop/explorer and drop them on the specified target area. The following *html* is used to illustrate the drag and drop.

ex03_selectingDragAndDrop.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>File API Drag and Drop Sample</title>
    <script src="ex03_selectingDragAndDrop.js"></script>
    <link rel="stylesheet" href="ex03.css">
  </head>
  <body>
    <div class="sample">
      <div id="target">
          Drop files here
      </div>
      <output id="list"></output>
    </div>
  </body>
</html>
```

The event handlers, *dragover* and *drop,* are associated with the *target* element in the application.

ex03_selectingDragAndDrop.js

```
window.onload = init;

function init() {
    // Setup the listeners on the target
    var target = document.getElementById('target');
    target.addEventListener('dragover',
            handleDragOver, false);
    target.addEventListener('drop',
            handleFileSelections, false);
}
```

When the drag ends, the default behavior of the browser to open the files is prevented as shown below.

```
function handleDragOver(evt) {
    evt.stopPropagation();
    evt.preventDefault();
    evt.dataTransfer.dropEffect = 'copy';
}
```

The *drop* event handler is used to handle the files that were selected and dropped by the user. The *files* property of the data transfer object refers to the *FileList* object containing the user's selection.

```
function handleFileSelections(evt) {
    evt.stopPropagation();
    evt.preventDefault();

    // Get the FileList object.
    var files = evt.dataTransfer.files;

    var output = '';

    // Select each File and its attributes
    for (var i = 0; i < files.length; i++) {
      var f = files.item(i);
      output += '<li><em>' + f.name + '</em>';
      output += ', Type: ' + f.type;
      output += ', Size: ' + f.size + ' bytes';
      output += ', Modified: ' +
        f.lastModifiedDate.toLocaleString() + '</li>';
    }

    // Show the selections
    document.getElementById('list').innerHTML =
      '<ul>' + output + '</ul>';
}
```

A sample output of the application is shown below. The *type* property of the *File* object is not applicable when the selected item is a folder/directory.

Reading Image Files (Optional)

The *FileReader* object is used for reading the *File*'s content into memory. After the content is loaded, the *onload* event of the reader is fired. The *result* property of the reader provided access to the file data.

The FileReader supports the following asynchronous operations:

- *readAsBinaryString*(File/Blob) – the *result* property contains the data as a binary string, each byte represented as an integer in the range 0..255.
- *readAsText*(File/Blob, {encoding}) – the *result* property contains the data as a text string.

- *readAsDataURL*(File/Blob) – the *result* property contains the data encoded as a data URL.
- *readAsArrayBuffer*(File/Blob) – the *result* property contains the data as an *ArrayBuffer* object.

The following *html* is used for selecting the image files to be read and displayed in the web application.

ex04_readingImageFiles.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Reading Image Files</title>
    <script src="ex04_readingImageFiles.js"></script>
    <link rel="stylesheet" href="ex04.css">
  </head>
  <body>
    <div class="sample">
      <input type="file" id="myfiles"
         name="myfiles[]" multiple />
      <output id="list"></output>
    </div>

  </body>
</html>
```

The associated *JavaScript* code is shown below. When the files are selected, the *change* event handler handles the file selections.

ex04_readingImageFiles.js

```
window.onload = init;

function init() {
     var elem = document.getElementById('myfiles');
     elem.addEventListener('change',
           handleFileSelections, false);
}
```

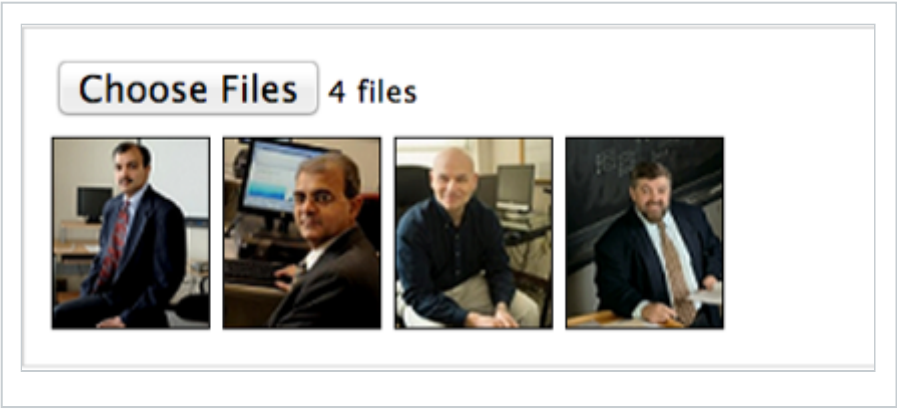
For each of the files selected by the user, if the *type* of the *File* object is not an *image* type, the result of the code is skipped.

```
// Process the selections
 function handleFileSelections(evt) {
     // Get the FileList object
   var files = evt.target.files;

   // Render image files
   for (var i = 0; i < files.length; i++) {
     var f = files[i];
     // If not image file, skip
     if (!f.type.match('image.*')) {
       continue;
     }
```

For each of the *image* files, the *FileReader* object is used to read the contents as a data URL. The *onload* event handler creates the *img* element and appends it to the target area.

```

     // Create a FileReader object
     var reader = new FileReader();

     // Capture the file information using closure (IIFE)
     reader.onload = (function(theFile) {
       return function(e) {
         // Render image.
         var span = document.createElement('span');
         span.innerHTML = '<img src="' + e.target.result +
                   '" title="' + theFile.name + '"/>';
         document.getElementById('list').appendChild(span);
         console.log(e.target); // the FileReader object
       };
     })(f);

     // Read in the image file as a data URL.
     reader.readAsDataURL(f);
   }
 }
```

A sample output is shown below.



Case Study – Reading Chunks of Data (Optional)

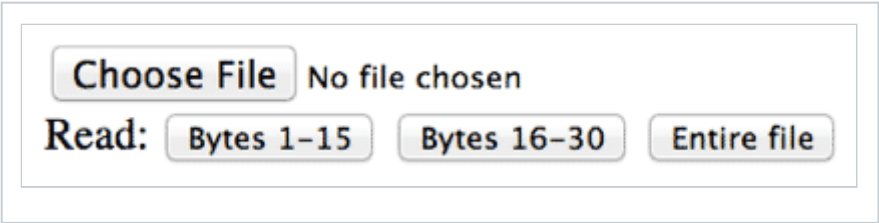
For large files, reading the entire content into memory may not be the best option. Also, portions of the file may be read and sent to the server as and when required. The *File* interface provides the *slice* method that takes the starting byte and ending byte and returns a new *Blob* object. Data can then be read asynchronously using this *Blob* object.

The following example allows the user to optionally read portions of the file or the entire contents.

ex05_slicingFile.html

```
5     <script src="ex05_slicingFile.js"></script>
6     <link rel="stylesheet" href="ex05.css">
7   </head>
8   <body>
9     <input type="file" id="myfile" name="myfile" />
10
11    Read:
12    <span class="readButtons">
13      <button data-start="0" data-end="14">
14        Bytes 1-15
15      </button>
16      <button data-start="15" data-end="29">
17        Bytes 16-30
18      </button>
19      <button>
20        Entire file
21      </button>
22    </span>
23
24    <div id="range"></div>
25    <div id="content"></div>
26  </body>
```

The above application is rendered as shown below:



The associated event handlers for the *input* element and the various buttons are shown below.

ex05_slicingFile.js

```
1  window.onload = init;
2
3  function init() {
4      var elem = document.getElementById('myfile');
5      elem.addEventListener('change', clearOutput, false);
6
7      var target = document.querySelector('.readButtons');
8      target.addEventListener('click', readData, false);
9  }
10
11 function clearOutput(evt) {
12     document.getElementById('content').textContent = '';
13     document.getElementById('range').textContent = '';
14 }
```

The *data-start* and *data-end* attributes specified for the button elements provide the required data for reading portions of the file.

```
16 function readData(evt) {
17     var start = evt.target.getAttribute('data-start');
18     var end = evt.target.getAttribute('data-end');
19     readBlob(start, end);
20 }
```

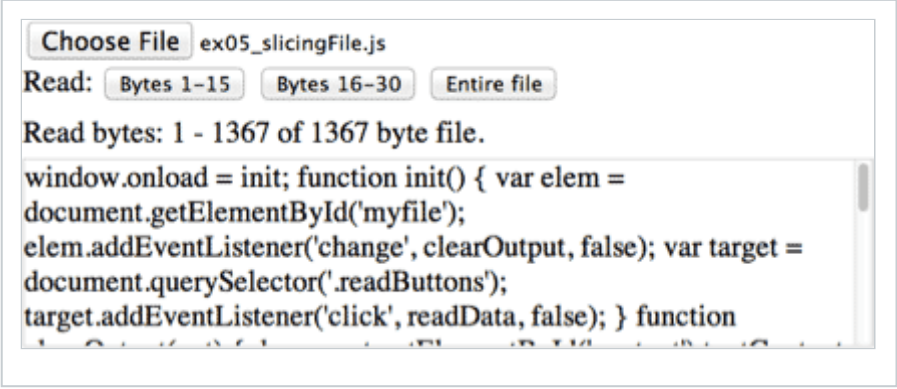
The application is written to read the contents of the single file. The first selection of the input element *files* property is used in this case. If the start and end data is not available, the entire contents of the file are read as shown below.

```
function readBlob(start, end) {

    var files = document.getElementById('myfile').files;
    if (!files.length) {
        alert('Please select a file!');
        return;
    }

    // Select the first file and the positions
    var file = files[0];
    var begin = parseInt(start) || 0;
    var end = parseInt(end) || (file.size - 1);
```

The data is read using the *FileReader* object into the *Blob* created through the *File*'s slice method. The *onload* event handler is fired after the data is read asynchronously.

```
    // Create a FileReader object
    var reader = new FileReader();

    reader.onload = function(evt) {
        document.getElementById('content').textContent =
                evt.target.result;
        document.getElementById('range').textContent =
            'Read bytes: ' +  (begin + 1) +  ' - ' +
              (end + 1) + ' of ' +  file.size + ' byte file.';
    };

    var blob = file.slice(begin, end + 1);
    reader.readAsBinaryString(blob);
}
```

A sample output of the application is shown below.



Server-Sent Events

Introduction

In a typical web application, the client (browser) sends a request to the server to receive some data. The server may have new results to send or if not, send the existing results. This approach is not practical if the server continuously generates data that is needed by the client. With server-sent events, the server can send (push) new data to the browser at any time.

Browser Support

Browser support for *server-sent events* capability can be determined programmatically by checking if the *EventSource* constructor function is defined or not as shown below.

ex01_checkSSESupport.html

```
<!DOCTYPE html>
<html>
  <body>
    <script type="text/javascript">
      if (typeof(EventSource) != "undefined")
        document.write("Server-Sent Event support is available.")
      else
        document.write("Server-Sent Event support is not available.")
    </script>
  </body>
</html>
```

Server Script for Server-Side Events

Various server-side technologies have mechanisms to emit data following the server-side events protocol. A sample script written in PHP is shown below. The MIME type for the data sent by the server is set to *text/event-stream*. The response data from the server using the event stream format should contain a ***data:*** line followed by the message, and followed by two newline **\n** characters to end that stream.

```
data: message\n\n
```

For messages spanning multiple lines, the format is as follows. Each line ends with a single newline character while the last line should end with two newline characters.

```
data: first line\n
data: second line\n
data: third line\n\n
```

The following PHP script sends the *data* messages in a loop every five seconds showing the current count and the current data. When the count reaches 10, it will also send the *Close* message to the browser.

ex02_sse.php

```php
<?php
date_default_timezone_set("America/New_York");
header("Content-Type: text/event-stream\n\n");

$count = 0;
while (1) {
  // Every 5 seconds send an event

  $count++;
  $curDate = date('l jS \of F Y, h:i:s A');

  echo 'data: This is a message #'. $count . ' - ' . $curDate . "\n\n";

  if ($count == 10) {
   echo 'data: Close' . "\n\n";
  }

  ob_end_flush();
  flush();
  sleep(5);
}
?>
```

Web Application using Server-Side Events

A sample web application utilizing the Server-Side events functionality is built in the following example. The web page refers to the corresponding JavaScript file containing the client-side code for interacting with the server script. When data is received from the server, the messages are displayed as line items within the *items* unordered list element.

ex02_SSESample.html

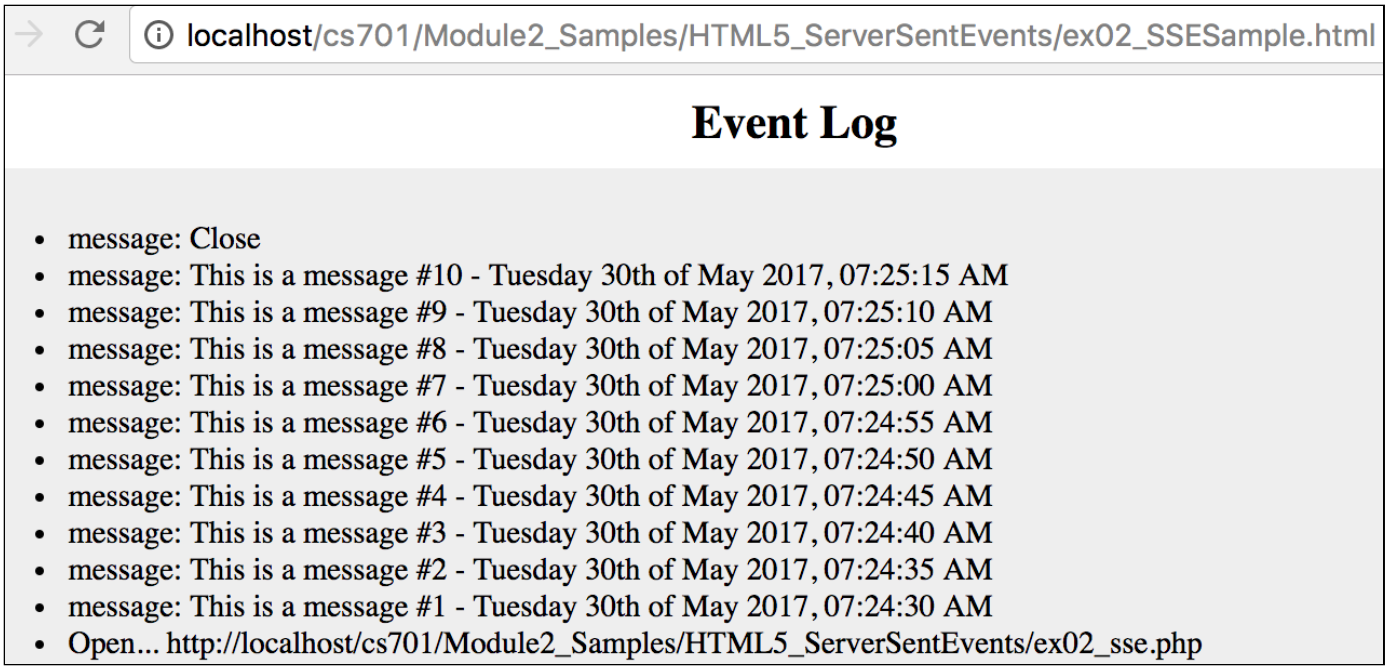
```
<!DOCTYPE html>
<html>
  <head>
    <title>Server-Sent Events</title>
    <meta charset="utf-8">
    <script src="ex02_SSESample.js"></script>
    <link rel="stylesheet" href="html5.css">
  </head>
  <body>
    <header>
      <h2> Event Log</h2>
    </header>

    <section>
        <article>
            <ul id="items"></ul>
        </article>
    </section>
  </body>
</html>
```

The JavaScript code for the application is shown below. The *EventSource* constructor takes the location of the server-side resource and returns the *EventSource* object. The *EventSource* object emits the *open*, *message*, and *error* events. The *onopen* property specifies the event handler function that is called when the *open* event is received when the connection is established to the server script. The *onmessage* property specifies the event handler function that is called when a *message* event is received and the *data* property of the event gives the payload of the message. The *error* event is raised when an *error* event is dispatched by the *EventSource* object. In the following example, when a message is received, it is inserted as the first line item with the DOM element identified by *items*. Also, if the data received from the server is the *Close* message, it is interpreted by the client as closing the *EventSource* object.

ex02_SSESample.js

```
(function () {

  // create the EventSource
  var evtSource = new EventSource("ex02_sse.php");

  // Subscribe to Events

  evtSource.onopen = function(e) {
    log("Open... " + e.target.url);
  }

  evtSource.onerror = function(e) {
    log("Error... " + e.target.url);
  }

  evtSource.onmessage = function(e) {
    log("message: " + e.data);
    if (e.data  == "Close") {
      evtSource.close();
    }
  }

  function log(message) {
   var newElement = document.createElement("li");
   newElement.innerHTML = message;
   items.insertBefore(newElement, items.firstChild);
  }
})();
```

A sample output of the application is shown below. Each client connecting to the server-side script will receive its own set of independent messages.



References

References

1. Working with Drag and Drop, Pro HTML5 Programming, 2nd Edition, Chapter 9.
2. Drag and Drop, HTML Living Standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/dnd.html
3. Using the Web Workers API, Pro HTML5 Programming, 2nd Edition, Chapter 10.
4. Web Workers, HTML Living Standard. http://www.whatwg.org/specs/web-apps/currentwork/multipage/workers.html
5. Eric Bidelman, The Basics of Web Workers, HTML5 Rocks Tutorials. http://www.html5rocks.com/en/tutorials/workers/basics/
6. Using the Storage APIs, Pro HTML5 Programming, 2nd Edition, Chapter 11.
7. Web Storage, HTML5, A vocabulary and associated APIs for HTML and XHTML, W3C, http://www.w3.org/TR/webstorage
8. Micahel Mahemoff, Client-Side Storage, http://www.html5rocks.com/en/tutorials/offline/storage/
9. Using the Geolocation API, Pro HTML5 Programming, 2nd Edition, Chapter 5.
10. Geolocation API Specification W3C Proposed Recommendation, 2012. http://www.w3.org/TR/geolocation-API/
11. Eric Bidelman, Reading files in JavaScript using the File APIs, http://www.html5rocks.com/en/tutorials/file/dndfiles/
12. File API, W3C, http://www.w3.org/TR/FileAPI/
13. Eric Bidelman, Reading files in JavaScript using the File APIs, http://www.html5rocks.com/en/tutorials/file/dndfiles/

14. File API, W3C, http://www.w3.org/TR/FileAPI/
15. IndexedDB API, https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
16. Using Server-Sent Events, https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events