|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385329) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385329) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385329) |
|  | |
| 18. Some More HTML5 APIs  18.1. Web Storage Overview  *Copyright (c) 2014, Rula Khayrallah*  The HTML5 Web Storage API allows web applications to store **data locally on the user’s device**.  For instance a web app may store certain user preferences, or progress through a lesson plan, or scores in a game.  This kind of storage is **specific to a single origin**.  The origin is defined as the combination of scheme, such as http, domain, such as foothill.edu, and port number such as 8080 for our own node server.  A script associated with one origin cannot read data stored by a script from a different origin.  For example the web pages  [http://foothill.edu/counseling/](http://foothill.edu/counseling/" \t "_blank) and <http://foothill.edu/aid/> can share the data stored in web storage because they have the same origin: they have the same scheme (http), the same domain (foothill.edu) and the same port number (default for both).  However  [https://myetudes.org/portal](https://myetudes.org/portal" \t "_blank) and <http://foothill.edu/counseling/>  do not have the same origin.  They have different schemes (http vs https) and different domains (myetudes.org vs foothill.edu).  As a result they each have their own web storage and they can’t read the data stored by the other.  It is also important to note that**the data stored through the web storage API is browser specific**.  For example when we access a web app with Chrome, and then access it again with Firefox, any data stored through the web storage API during the first visit is not accessible during the second visit.  The Web Storage API is easy to use. It is more secure and faster than using cookies.  It allows us to store large amounts of data without a negative impact on performance.  **The web storage API includes the localStorage and sessionStorage objects.**  Before we go into the details of each of these objects, it is important to note that the data here is stored as plain text, **unencrypted**.  **As a result, this should never be used to store sensitive information** such as credit card numbers or passwords.  The data is easily accessible to anyone who has access to that device (and that can be physical access or software access through spyware).  18.2. The localStorage Object  *Copyright (c) 2014, Rula Khayrallah*  We have access to the localStorage object through the property of the window object:  window.localStorage (or simply localStorage).  It is technically a 'Storage object'  but it works the same way as a regular JavaScript object.  The data is stored in name:value pairs, just like a regular JavaScript object.  Let’s illustrate the use of localStorage object with a simple example.  Consider the following modified HTML file for our word guessing name where we invite the player to enter their name.  We'll call the modified HTML file guesssave.html.  guessave.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>Guessing Game</title>  <link rel = "stylesheet" type = "text/css" href = "guess.css" media = "all">  </head>  <body>  **<p>Who is playing?</p>**  **<input id = "playername" type="text">**  <p>Guess a Letter</p>  <p id = "display" class = "letters"></p>  <input id = "guess" type="text" maxlength='1' class = "letters" autofocus>  <div>  <input id = "guessbutton" type="button" value="GUESS">  </div>  <p>Wrong Letters</p>  <p id = "wrong" class = "letters wrong"> </p>  <progress id = "indicator" value="0" max="10">  </progress>  <div>  <input id = "restart" type="button" value="RESTART">  </div>  <script defer src="../scripts/guess.js"></script>  </body>  </html>  Now let’s open the HTML source document in Firefox and enable the Firebug console.  Enter a name under the prompt: Who’s playing?  Then enter the following at the Console prompt  >>>var player = document.getElementById("playername").value;  The variable player now contains the string "Alice" or any other name you entered.  To save the player’s name in the localStorage object, we can simply write:  >>>localStorage["lastplayer"] = player;  or:  >>>localStorage.lastplayer = player;  We are basically creating a "lastplayer" property of the localStorage object.  We can also use the setItem() property of the localStorage object to get the same result as above:  >>>localStorage.setItem("lastplayer", player);  We can now query the localStorage object by typing the following at the console prompt:  >>> localStorage  1 item in Storage lastplayer="Alice"  We can use any of the following:  >>> localStorage.getItem("lastplayer");  // use the getItem() method of the localStorage object  "Alice"  >>> localStorage["lastplayer"];  "Alice"  >>> localStorage.lastplayer;  "Alice"  All values in localStorage are currently stored as strings. When we store a number, it is implicitly converted to a string.  Let’s say for example that we want to store the player’s score.  At the Firebug console prompt, enter:  >>> localStorage.setItem("score", 1);  Now let’s query the localStorage object:  >>> localStorage;  2 items in Storage lastplayer="Alice", score="1"  >>> localStorage.score;  "1"  Note that 1 is stored as the string "1".  If for example we need to retrieve the score and update it, we must convert to a number first (otherwise we get a string concatenation instead of an addition).  localStorage["score"] = Number(localStorage["score"]) + 5;  >>> localStorage["score"];  "6"  The following will not work.  It performs string concatenation  >>> localStorage["score"] = localStorage["score"] + 5;  "15"  **The data stored in the localStorage object is permanent.  It has no expiration date and it does not disappear when the browser is closed.**  We can close our browser completely, then reopen it and reload our page and then query the localStorage again and the information we saved is still there.  >>> localStorage  2 items in Storage lastplayer="Alice", score="6"  However if we open the same page in Chrome, we don’t have access to the same localStorage.  To see that, open the same quesssave.html page in Chrome.  To access the console in Chrome you can select the JavaScript console from the Tools menu.  Querying the localStorage object in Chrome shows an empty object because **each storage object is specific to the browser.**    We said that the data stored in the localStorage object is permanent.  This does not mean that we can never delete it if we choose to.  The removeItem() method will remove a given item from the localStorage object.  Let’s go back to our web page in Firefox and type:  >>> localStorage.removeItem("lastplayer");  Then query our localStorage object:  >>> localStorage  1 item in Storage score="6"  We can use the clear() method to remove all items in the localStorage object.  >>> localStorage.clear();  >>> localStorage  0 items in Storage  18.3. The sessionStorage Object  *Copyright (c) 2014, Rula Khayrallah*  The sessionStorage object is similar to the localStorage object.  We can access it through the property of the window object:  window.sessionStorage (or simply sessionStorage).  We can manipulate it, adding items, accessing items and removing items just like the localStorage object.  **The difference is in lifetime and scope.**  Let’s consider scope first.  The sessionStorage object is specific not just to the browser (Chrome vs Firefox) but to a specific tab within the browser.  To illustrate this, let’s first run the same scenario as in the previous section but this time save our data in sessionStorage.  We first open our source HTML document guesssave.html in Firefox, enter a player name and then go to the Firebug console.  We enter the following at the Console prompt  >>>var player = document.getElementById("playername").value;  The variable player now contains the string "Alice" or any other name you entered.  To save the player’s name in the **sessionStorage** object, we can simply write:  >>>sessionStorage["lastplayer"] = player;  Or:  >>>sessionStorage.setItem("lastplayer", player);  We can now query the sessionStorage object by typing the following at the console prompt:  >>> sessionStorage  1 item in Storage lastplayer="Alice"  Or:  >>> sessionStorage.getItem("lastplayer")  // use the getItem() method of the session Storage object  "Alice"  All values in sessionStorage are also stored as strings. When we store a number, it is implicitly converted to a string.  >>> sessionStorage.setItem("score", 1);  >>> sessionStorage.score;  "1"  In order to retrieve the score and update it, we must convert it to a number first.  sessionStorage["score"] = Number(sessionStorage["score"]) + 5;  >>> sessionStorage["score"];  "6"  Now let’s open a new tab in Firefox and in that tab open the same web page guesssave.html.  At the console prompt in that second tab, type:  >>> sessionStorage  0 items in Storage  **The sessionStorage that we defined in the first tab is only accessible from that tab.**  Now let’s close both tabs and then reopen guesssave.html in Firefox.  >>> sessionStorage  0 items in Storage  **The sessionStorage object  is cleared when the corresponding window is closed.**  18.4. The Application Cache  *Copyright (c) 2014, Rula Khayrallah*  In general, caching refers to the practice of storing recently accessed data so that it can be easily and quickly accessed later.  Browsers usually cache web pages on the user’s device.  For example, when we visit a web page, the browser may cache the source HTML document, the images, stylesheets and JavaScript files used.  When at a later time we go back to revisit that same page (say with the back button), the browser can get a copy of all the files needed from the local device. Browsers have some rules to determine if the copy on the local device is 'fresh'.  This kind of caching results in a faster user response and a reduced load on the server and network.  The Application Cache, introduced in HTML5 deals with a slightly different aspect of caching.  **It allows us to have web applications available offline, when the device does not have internet access.**  The application cache is different from the browser’s cache. While the browser’s cache has a size limit and keeps the most recently used data only, the application cache is permanent.  Applications stored there remain there until they are explicitly deleted.  To be available offline, web applications have to install themselves in the application cache, use localStorage to store any associated data, and then synchronize with the server when they detect that the device is online again.  Let’s go back to our simple canvas drawing example from module 10.3. Our goal is to make that simple app available when we are offline. To emulate offline, we’ll initially serve our app using our node server server.js, then stop the server and access the app.  We’ll need to access it through our localhost:8080 url NOT through the file scheme.  Here's our drawing app source document:  draw.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>Let's Draw</title>  </head>  <body>  <h2> Just click inside the box </h2>  <canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">  </canvas>  <script defer src="../scripts/draw.js"></script>  </body>  </html>  The corresponding client side JavaScript file draw.js contains the following:  draw.js  function drawSquare(event) {  // Our function takes the event object as a parameter  var myCanvas=document.getElementById("myCanvas");  // obtain the coordinates with respect to the canvas  var x = event.clientX - myCanvas.offsetLeft;  var y = event.clientY - myCanvas.offsetTop;  // to access the canvas, we need its context  var myContext = myCanvas.getContext("2d");  // set the color to red  myContext.fillStyle="#FF0000";  // draw a 10 by 10 square starting at the click event position  myContext.fillRect(x,y,10,10);  };  document.getElementById("myCanvas").addEventListener("click", drawSquare, false);  To enable application cache, we must first create a manifest file.  **A manifest file is a text file that includes the resources that our application needs to run.  These resources include source HTML documents, images, stylesheets and JavaScript files used.**The manifest file tells the browser what to cache and what to never cache.  The manifest file has to start with the line:  CACHE MANIFEST  It may have up to three sections:  CACHE:  - Files listed under this header will be cached after they are downloaded for the first time.  This is the section that we need here.  In order for an application to be available offline, its associated html source,  JavaScript files (including jquery if applicable), style sheets, any images or other resources needed must be listed in this section.  Note that the section header (CACHE:) may be omitted.  Files listed immediately after the CACHE MANIFEST line are assumed to belong to the CACHE section.  NETWORK: - Files listed under this header require a connection to the server, and will never be cached.  FALLBACK: - This section specifies fallback pages the browser should use if a resource is inaccessible. Each entry in this section lists two URIs—the first is the resource, the second is the fallback.  Our focus here is the CACHE section.  Let’s create our manifest file.  In our case, our drawing application only needs the following resources:  draw.html  ../scripts/draw.js  So we create our manifest file draw.appcache as shown below.  The recommended file extension for a manifest file is "appcache".  We'll save our manifest file in the html folder.  draw.appcache  CACHE MANIFEST  draw.html  ../scripts/draw.js  Note that we omitted  the CACHE: section header in the above example.  We could have also written:  CACHE MANIFEST  CACHE:  draw.html  ../scripts/draw.js  The next step is to associate our manifest with the application.  To do that we include the manifest attribute in the main source document's <html> tag:  draw.html  <!DOCTYPE html>  **<html manifest="draw.appcache">**  <head>  <meta charset="utf-8">  <title>Let's Draw</title>  </head>  <body>  <h2> Just click inside the box </h2>  <canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">  </canvas>  <script  defer src="../scripts/draw.js"></script>  </body>  </html>  There is one more detail about the manifest file that we need to address:  it must be served with content type “text/cache-manifest”.  We will use our node server from section 17.5, server.js.  This server checks the extension of the requested resource and serves it with the corresponding content type header.  This way our manifest file will be served with the correct type: "text/cache-manifest".  We can now start our server from the node command prompt (or terminal window on a Mac) by typing:  node server.js  Then go to Firefox and write the following in the address bar:  [http://localhost:8080/html/draw.html](http://localhost:8080/drawapp.html)  Our simple drawing application page is displayed.  Now let’s go back to the node command prompt window or terminal window and stop the server by closing the window or pressing control C.  Then we go back to our Firefox window, close it completely then reopen it.  Now try to access calculator.html by typing the following in the address bar:  [http://localhost:8080/html/calculator.html](http://localhost:8080/calculator.html)  We get an error message 'Unable to connect' because the server is no longer running.  However if we try to access our drawing application by typing the following:  [http://localhost:8080/html/draw.html](http://localhost:8080/drawapp.html)  We can see that the drawing application is loaded successfully from the application cache even though our server is not running (or if we are offline).  Clearing the cache and checking the cache:  In Firefox, to see the list of the files stored in the offline cache, you can do the following:  Type:  about:cache  in the address bar.  Then under Offline cash device, click on List Cache Entries.  You can see the list of the files stored in the offline cache.  You can clear the application cache as follows.  From the menu, select History - > Clear Recent History then make sure Offline Website Data is selected and click on Clear Now.  18.5. History Management  *Copyright (c) 2014, Rula Khayrallah*  We have access to the browsing history associated with a given window through the history property window.history or simply history.  This property refers to an object that includes entries for the urls 'visited'.  We can simply type history at the Firebug prompt:  >>> history  6 history entries  Note that we don't have access to the stored entries, just their number.  This is to make sure that a script running on one web page does not have access to a user's browsing history.  There are methods defined on the history object that allow us to navigate back and forward and these methods have the same effect as pressing the browser Back and Forward buttons.  We can try the following from the Firebug console:  >>>history.back()  // this loads the previous web page  >>>history.forward() // this takes us forward one page  >>>history.go(-3)  // this takes us 3 pages back  >>> history.go(2) // this takes us forward 2 pages  The issue we face is that today, **web applications dynamically change the web page content without loading a new web page**.    We still want to allow the user to use the Back and Forward buttons to navigate between these dynamically created application states.  HTML5 introduced an API that allows us to implement seamless history management in a dynamic application.  Let's see how to do that with a simple web page that is modified based on user input (without a page reload).  For the sake of simplicity, we will not include any Ajax communication with the server in this example.  The same approach can be used when we have Ajax communication with the server.  We’ll call our source document historydemo.html.  historydemo.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2>History Management with Dynamic Pages</h2>  <p>Click on one of the buttons to change the page dynamically. </p>  <input id = "first" type="button" value="FIRST">  <input id = "second" type="button" value="SECOND">  <p id="response"></p>  <script defer src="../scripts/historydemo.js"></script>  </body>  </html>  Let's write a  first version of our script historydemo.js without taking into account any history management:  historydemo.js  // This function takes a parameter, line, and writes the corresponding  // text to the element with id: response.  function displayIt (line) {      // store the text corresponding to the two options in an object      // with properties first and second.      var lineContent = {first: "First Line", second: "Second line"};      // display the corresponding line in the html element with id response      document.getElementById("response").textContent = lineContent[line];  }  // this function handles click events on both buttons.  function update(event){      // determine which button was clicked from the event.target      displayIt(event.target.id);  }    // Add event listeners to call update() when a button is clicked  document.getElementById("first").addEventListener("click", update);  document.getElementById("second").addEventListener("click", update);  Once we have saved historydemo.html and historydemo.js, we can open historydemo.html with Firefox.  Let’s press on the First and Second buttons a few times to change the web change.  What happens when we use the browser’s back button <- to go back to the previous page?  We go back to whatever initial page we had before we loaded historydemo.html.  We can’t navigate between the dynamic states of historydemo.html.  We can modify our JavaScript code to use the **pushState()** method and the **popstate**event as follows:  The pushState() method allows us to **add an entry to the history object.**  The syntax is:  history.pushState(stateObject, title, URL)  **stateObject** is an object that contains all the information needed to restore the current state of the document. Any object that can be converted to a string with JSON.stringify() will work here.  When the user navigates back or forth from one history entry to the other, a copy of the stateObject associated with the current history entry becomes available through **history.state**.  **The title** parameter is currently ignored by major browsers.  It refers to a title to be associated with that state.  **The URL** parameter is optional.  It allows us to associate **each state with a unique url.**  This URL will appear in the browser’s location bar.  If an absolute url is specified, **it must have the same origin as the current url** (the url of the corresponding static web page).  Going back to our historydemo.html web page, there are two dynamic states that we need to associate with that page:   * The state right after the user has clicked the FIRST button. * The state right after the user has clicked the SECOND button.   We modify our update() function in the historydemo.js file to push a state object corresponding to each of these two states:  function update(event){      var stateObject;  // our state object will simply contain a string      displayIt(event.target.id);      // determine which button was clicked from the event.target      // and push the corresponding state object.      if (event.target.id === "first") {          // identify the state corresponding to FIRST button  **stateObject = "first";**  **history.pushState(stateObject,"First Line", "#first");**      } else {          // identify the state corresponding to SECOND button  **stateObject = "second";**  **history.pushState(stateObject,"Second Line", "#second");**      }  }  Note that we just picked a string here to identify our state ("first" or "second").  We could have picked 1 or 2 or a more complex object…  The next step is for our code **to generate the correct web page when the user navigates back or forth to a given state.**  **The popstate event is triggered on the window object when the user navigates back or forth.**  We can add an event listener for the popstate event as follows:  window.addEventListener("popstate", generateState);  Before we define the generateState function, remember that when the user navigates back or forth from one history entry to the other, a copy of the stateObject associated with the current history entry becomes available through **history.state**.  // Generate the web page corresponding to the state in history.state  // If no state is available, restore the original web page  function generateState(){      if (history.state) {         displayIt(history.state);      } else {          document.getElementById("response").textContent = "";      }  }  There is one last special case we need to cover.  When a page (with dynamic states) reloads, it may have a non-null state object without a popstate event being triggered.  To cover this special case, we add the following to our JavaScript code:  if (history.state) {      generateState();  }    Putting it all together, we now have:  historydemo.js  // This function takes a parameter, line, and writes the corresponding  // text to the element with id: response.  function displayIt(line){      // store the text corresponding to the two options in an object      // with properties first and second.      var lineContent = {first: "First Line", second: "Second line"};      // display the corresponding line in the html element with id response      document.getElementById("response").textContent = lineContent[line];  }    function update(event){      var stateObject;  // our state object will simply contain a string      displayIt(event.target.id);      // determine which button was clicked from the event.target      // and push the corresponding state object      if (event.target.id === "first") {          // identify the state corresponding to FIRST button          stateObject = "first";          history.pushState(stateObject,"First Line", "#first");      } else {          // identify the state corresponding to SECOND button          stateObject = "second";          history.pushState(stateObject,"Second Line", "#second");      }  }    // Generate the web page corresponding to the state in history.state.  // If no state is available, restore the original web page.  function generateState(){      if (history.state) {         displayIt(history.state);      } else {          document.getElementById("response").textContent = "";      }  }    // Add event listeners to call update() when a button is clicked  document.getElementById("first").addEventListener("click", update);  document.getElementById("second").addEventListener("click", update);  // Add even listener to handle popstate event on the window  window.addEventListener("popstate", generateState);  // Handle the special case of reload  if (history.state) {      generateState();  }  To see how it works, we can now load historydemo.html in our browser, click on FIRST and SECONDS a few times to generate different dynamic pages then navigate with the browser back and forward buttons (<- and ->).  As you do that note that the correct dynamic page is loaded and that the corresponding url is displayed in the address bar:  [file:///somepath/historydemo.html#first](file:///\\somepath\historydemo.html#first)  [file:///somepath/historydemo.html#second](file:///\\somepath\historydemo.html#second)  where somepath depends on your configuration.  18.6. Geolocation  *Copyright (c) 2014, Rula Khayrallah*  The HTML5 Geolocation API gives us access to the geographical position of a user with their permission.  This allows us to generate directions, maps, and local information such as weather or local shopping and dining options.  The geographical position is available through the property: navigator.geolocation.  Here is a very simple example of how to use it.  We start with the source document geodemo.html.  geodemo.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2>Where in the world are we?</h2>  <p>Click on the button to find out. </p>  <input id = "findme" type="button" value="FINDME">  <p>Latitude:  <span id="latitude"></span></p>  <p>Longitude: <span id="longitude"></span></p>  <script defer src="../scripts/geodemo.js"></script>  </body>  </html>  We would like to display the latitude and longitude of the user's position when the user clicks on the FINDME button.  We use the **getCurrentPosition()** method to access the position. This method is **asynchronous.**  It takes a callback function.  We invoke it as shown in our scipt geodemo.js below.  geodemo.js  // Show the latitude and the longitude corresponding to the given position  function showPosition(position)    {        document.getElementById("latitude").textContent = position.coords.latitude;        document.getElementById("longitude").textContent = position.coords.longitude;    }  document.getElementById("findme").addEventListener("click", function (){  **navigator.geolocation.getCurrentPosition(showPosition);**  });  Let’s open the web page geodemo.html in Firefox and click on the FINDME button:  The browser first asks for our permission to share our location.  If we give our permission, then our position will be available to the script.  watchPosition() is another method that keeps returning the updated position as the user moves and until we invoke clearPosition().  It allows us to track the user’s movement.  Note that the position is more accurate for devices with GPS.  18.7. Web Workers  *Copyright (c) 2014, Rula Khayrallah*  Client side JavaScript is essentially single threaded.  If a function takes too long to perform a certain task the web page becomes unresponsive to user input.  The Web Workers API allows us to**run  computationally intensive JavaScript code in a background thread without affecting the performance of the web page.**  We’ll illustrate the use of web workers with a very simple example.  Let’s start with a source document, workerdemo.html  based on calculator.html. To be able to run it from our local server, we'll save it under in NodeServer/html folder.  workerdemo.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2>Web Workers at work</h2>  <h4>This is our old calculator</h2>  <p>Please enter two numbers:  </p>  <input id = "first" type="number">  <br>  <input id = "second" type="number">  <br>  <p> And the answer is:</p>  <p id="answer"></p>  <p>Click on the button to start a web worker in the background.</p>  **<input id = "start" type="button" value="START">**  <p id="result"></p>  <script defer src="../scripts/workerdemo.js"></script>  </body>  </html>  Here we have the simple calculator from our previous example but we also have a button that allows the user to start some computationally intensive task in the background.  For now we’ll assume that the task will be performed in the script compute.js.  Let’s write our main script workerdemo.js.  We'll save it in our NodeServer/scripts folder.  function update() {        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);        // Then  compute the sum      var myAnswer = firstNumber + secondNumber;        // And write it in the 'answer' element      document.getElementById('answer').textContent = myAnswer;  };    **function startWorker() {**  **var worker = new Worker('../scripts/compute.js');  // instantiate worker object**  **worker.addEventListener('message', function(event) {**  **document.getElementById('result').textContent = event.data;**  **});**  **}**    document.getElementById('first').addEventListener('input', update, false);  document.getElementById('second').addEventListener('input', update, false);  document.getElementById('start').addEventListener('click', startWorker, false);  This script is very similar to the add.js script we’ve seen earlier.  It has one new function that is called when the user clicks on the START button.  That function startWorker() starts by **instantiating a new worker object that will execute to code found in the script compute.js.**  **var worker = new Worker('../scripts/compute.js');**  **The main script and the worker script can now communicate via messages:**we add an event listener to our worker object so that when a message is detected we display the data returned in that message to the web page.  For demonstration purposes, we’ll write our compute.js script as follows:  // This web worker performs some computations  // and returns the result  var sum = 0;  for (var i = 0; i < 90000000; i++) {      sum += Math.pow(i, 5)      }  **postMessage(sum)**;  This web worker computes the sum of the first 90,000,000 numbers raised to the power of 5.  It sends a message back to the server with the result.  In practice, a web worker may be performing some complex calculations or image processing…  We'll save compute.js in our NodeServer/scripts folder.  We then start our Node server server.js .  In our browser address bar we type: localhost:8080/html/workerdemo.html.  We get the modified calculator page.  We click on the START button to start the 'compute.js' script in the background then enter some numbers.  The important thing to note here is that**while compute.js is performing the computation in the background (and before the result is displayed under the START button), the web page is responsive to our input and the calculator part of the web page is still working.** This would not have been the case if the calculations were performed as part of the main thread.  18.8. Web Sockets  *Copyright (c) 2014, Rula Khayrallah*  The WebSocket API implements a new communication protocol that enables us to develop**low latency, real-time applications**such as multi-player online games.  Instead of the stateless  (restful)  HTTP requests and responses that we’ve seen so far, web socketsdefine a **persistent bidirectional connection between a client and a server.**  To create a connection, the client script has to instantiate a web socket object and specify a url:  **var connection = new WebSocket ("ws://host:port/resource");**  ws is the protocol here.  We can also use wss for the secure version (corresponding to https).  Then we specify the host, possibly the port number and resource.  Web sockets may share ports with http connections.  Once we have created a connection, we can send data on it as plain text. JSON encoding is very handy here when we have some complex data to transmit.  The latest browsers also allow binary data to be sent.  **connection.send ("Hello Server World!");**  We can also register event handlers on the connection.  **connection.onmessage = function (event) {**  **var  message = event.data; // Get the message**  **//do something with it**  **};**  When we are done we can close the connection:  **connection.close();**  **The important thing to realize here is that once we open the connection, the server can send us messages at any time without waiting for a specific request.**  The web socket API is new and not available yet in all browsers.  socket.io is a library that has become very popular:  it includes a client side library that runs in the browser and a server side library for node.  It primarily uses the web socket  protocol, but when the browser does not support it, it can fall back on other communication methods. |  |