4

HTML5 APIs

While the language spec (ECMA-262) changes once in years, new HTML5 APIs sneak to the language almost with every browser update. Already available APIs are quite numerous. Yet we focus in this chapter on those that make use to reconsider the entire development process. We’ll learn how benefit from multithreading by using Web workers, how to build application from reusable independent Web Components, how store and search considerably huge amounts of data in client-side and how establish bi-directional communication with server.

In this chapter we will be covering following topics:

* Storing data in web-browser
* Boosting performance with JavaScript Workers
* Creating our first web component
* Learning to use server-to-browser communication channels

# Storing data in web-browser

Amongst HTML5 features there are a few intended for storing data on client-side: Web Storage, IndexedDB, and FileSystem API. We benefit from these technologies when:

* We want to cache client-side data so to make them fetchable without extra HTTP requests
* We have a significant amount of local data in the web application We want our application to work off-line

Let’s take a look at them

## Web Storage API

In past we had the only mechanism to keep the application state and it was HTTP Cookies. Besides unfriendly API, Cookies have a few flaws. They generally have a maximum size of about 4 KB. So we simply cannot store any decent amount of data. Cookies don’t really fit when the application state is being changed in different tabs. Cookies are vulnerable for Cross Site Scripting attacks.

Now we have an advanced API called Web Storage. It provides greater storage capacity (5-25MB depending on the browser) and doesn’t attach any data to the HTTP request headers. There two JavaScript built-in objects implementing this interface: localStorage and sessionStorage. The first is used as persistent data storage and the second for keeping the data during a session.

Storage API is very simple to use:

var storage = isPersistent ? localStorage : sessionStorage;

storage.setItem( "foo", "Foo" );

console.log( storage.getItem( "foo" ) );

storage.removeItem( "foo" );

Or by using getters/setters for convenience:

storage.foo = "Foo";

console.log( storage.foo );

delete storage.foo;

If we want to iterate through the storage we can use storage.length and storage.key():

var i = 0, len = storage.length, key;

for( ; i < len; i++ ) {

key = storage.key( i );

storage.getItem( key );

}

As you see the Web Storage API is much more developer-friendly comparing to Cookies. It’s also more powerful. One of the most common real-life examples where we need storage is shopping cart. When designing the application we have to keep in mind that user while making their choices often open pages with product details in multiple tabs or windows. So we should take care of storage synchronization across all the open pages. Fortunately whenever we update the local storage, “storage” event is fired on window object. So we can subscribe a handler for this event to update shopping cart with the actual data. A simple code illustrating this example may look like that:

<html>

<head>

<title>Web Storage</title>

</head>

<body>

<div>

<button data-bind="btn">Add to cart</button>

<button data-bind="reset">Reset</button>

</div>

<output data-bind="output">

</output>

<script>

var output = document.querySelector( "[data-bind=\"output\"]" ),

btn = document.querySelector( "[data-bind=\"btn\"]" ),

reset = document.querySelector( "[data-bind=\"reset\"]" ),

storage = localStorage,

/\*\*

\* Read from the storage

\* @return {Arrays}

\*/

get = function(){

// From the storage we receive either JSON string or null

return JSON.parse( storage.getItem( "cart" ) ) || [];

},

/\*\*

\* Append an item to the cart

\* @param {Object} product

\*/

append = function( product ) {

var data = get();

data.push( product );

// WebStorage accepts simple objects, so we pack the object into JSON string storage.setItem( "cart", JSON.stringify( data ) );

},

/\*\* Re-render list of items \*/

updateView = function(){

var data = get();

output.innerHTML = "";

data && data.forEach(function( item ){

output.innerHTML += [ "id: ", item.id, "<br />" ].join( "" );

});

};

this.btn.addEventListener( "click", function(){

append({ id: Math.floor(( Math.random() \* 100 ) + 1 ) });

updateView();

}, false );

this.reset.addEventListener( "click", function(){

storage.clear();

updateView();

}, false );

// Update item list when a new item is added in another window/tab

window.addEventListener( "storage", updateView, false );

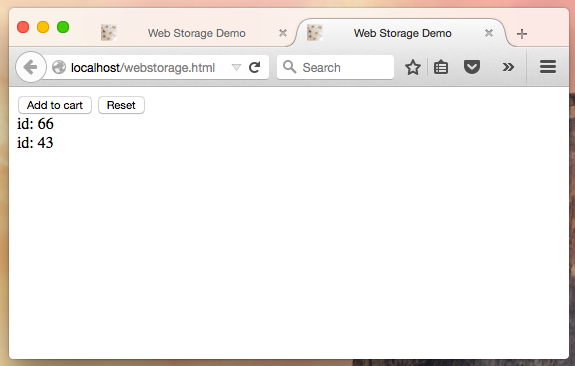
updateView();

</script>

</body>

</html>

To see it in action we have to open this HTML in two or more tabs. Now when we click Add to cart button we have the list of ordered items updated in every tab. As you probably noticed we can also clean up the cart by pressing Reset button. It calls storage.clear method and empties the list. If you fancy to use here sessionStorage instead of localStorage, I have to warn it won’t work. SessionStorage is isolated for every tab or window so we cannot communicate across them this way. But we could run this example with sessionStorage if we had the page loaded in different frame, but on the same window though.



1. Insert Image 04-01-01.png

## IndexedDB

Web Storage serves well when we have to store considerably small amount of data (megabytes). However if we need structured data in much greater amounts and we want performance searches through this data using indices we shall use IndexedDB API. The idea of an API for storing data in databases in a browser isn’t new. A few years ago Google and partners were actively advocating a standard candidate called Web SQL Database. This spec has failed to make it through W3C Recommendation though. Now we have IndexedDB API instead that is widely-supported already and provides a significant performance boost (asynchronous API, robust search due to indexed keys).

However the API of IndexedDB is pretty complex. It’s also quite hard to read because of large amounts of nested callbacks:

/\*\*

\* @type {IDBOpenDBRequest}  
 \* Syntax: indexedDB.open( DB name, DB version );

\*/

var request = indexedDB.open( "Cem", 2 );

/\*\* Report error \*/

request.onerror = function() {

alert( "Opps, something went wrong" );

};

/\*\*

\* Create DB

\* @param {Event} e

\*/

request.onupgradeneeded = function ( e ) {

var objectStore;

if ( e.oldVersion ) {

return;

}

// define schema

objectStore = e.currentTarget.result.createObjectStore( "employees", { keyPath: "email" });

objectStore.createIndex( "name", "name", { unique: false } );

// Populate objectStore with test data

objectStore.add({ name: "John Dow", email: "john@company.com" });

objectStore.add({ name: "Don Dow", email: "don@company.com" });

};

/\*\*

\* Find a row from the DB

\* @param {Event} e

\*/

request.onsuccess = function( e ) {

var db = e.target.result,

req = db.transaction([ "employees" ]).objectStore( "employees" ).get( "don@company.com" );

req.onsuccess = function() {

console.log( "Employee matching `don@company.com` is `" + req.result.name + "`" );

};

};

In this sample we create a request for opening DB. If the DB doesn’t exist or its version is changed, upgradeneeded event fires. In the function subscribed on this event we can define the schema by declaring object stores and their indices. So if we need to update the schema of the existing DB, we can increment the version number, upgradeneeded will fire again and the listener called to update the schema. As soon as we defined the schema we can populate object store with sample data. When the request opening DB is complete, we request the record matching email don@company.com. When the request is done we get in console

Employee matching `don@company.com` is `Don Dow`

Pretty tangled, isn’t it? The API makes me think of a wrapper. The best I know called Dexie (http://www.dexie.org). Just compare how easy to solve the same task with interface it exposes:

<script src="./Dexie.js"></script>

<script>

var db = new Dexie( "Cem" );

// Define DB

db.version( 3 )

.stores({ employees: "name, email" });

// Open the database

db.open().catch(function( err ){

alert( "Opps, something went wrong: " + err );

});

// Populate objectStore with test data

db.employees.add({ name: "John Dow", email: "john@company.com" });

db.employees.add({ name: "Don Dow", email: "don@company.com" });

// Find an employee by email

db.employees

.where( "email" )

.equals( "don@company.com" )

.each(function( employee ){

console.log( "Employee matching `don@company.com` is `" + employee.name + "`" );

});

</script>

## FileSystem API

Well, in a web application we can store key value pairs with Web Storage and we can create and use indexedDB. Something is still missing. Desktop applications can read and write files and directories. That is what we often need in a web application capable of running offline. FileSystem API allows us to create, read and write to a user local filesystem in application scope. Let’s take up an example:

window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;

/\*\*

\* Read file from a given FileSystem

\* @param {DOMFileSystem} fs

\* @param {String} file

\*/

var readFile = function( fs, file ) {

console.log( "Reading file " + file );

// Obtain FileEntry object

fs.root.getFile( file, {}, function( fileEntry ) {

fileEntry.file(function( file ){

// Create FileReader

var reader = new FileReader();

reader.onloadend = function() {

console.log( "Fetched content: ", this.result );

};

// Read file

reader.readAsText( file );

}, console.error );

}, console.error );

},

/\*\*

\* Save file into a given FileSystem and run onDone when ready

\* @param {DOMFileSystem} fs

\* @param {String} file

\* @param {Function} onDone

\*/

saveFile = function( fs, file, onDone ) {

console.log( "Writing file " + file );

// Obtain FileEntry object

fs.root.getFile( file, { create: true }, function( fileEntry ) {

// Create a FileWriter object for the FileEntry

fileEntry.createWriter(function( fileWriter ) {

var blob;

fileWriter.onwriteend = onDone;

fileWriter.onerror = function(e) {

console.error( "Writing error: " + e.toString() );

};

// Create a new Blob out of the text we want into the file.

blob = new Blob([ "Lorem Ipsum" ], { type: "text/plain" });

// Write into the file

fileWriter.write( blob );

}, console.error );

}, console.error );

},

/\*\*

\* Run when FileSystem initialized

\* @param {DOMFileSystem} fs

\*/

onInitFs = function ( fs ) {

const FILENAME = "log.txt";

console.log( "Opening file system: " + fs.name );

saveFile( fs, FILENAME, function(){

readFile( fs, FILENAME );

});

};

window.requestFileSystem( window.TEMPORARY, 5\*1024\*1024 /\*5MB\*/, onInitFs, console.error );

First of all we request a local file system (requestFileSystem) that’s sandboxed to the application. With the first argument we state whether the file system should be persistent. If we go with window.TEMPORARY, the data can be removed by browser (e.g. when more space needed). Otherwise (window.PERSISTENT) cannot be cleaned without explicit user confirmation. The second argument specifies how much of space we allocate for the file system. Then go onSuccess and onError callbacks. When the file system is created we receive a reference to the FileSystem object. The object has property fs.root where is keeps DirectoryEntry bound to the root file system directory. The DirectoryEntry object has methods DirectoryEntry.getDirectory, DirectoryEntry.getFile, DirectoryEntry.removeRecursevly and DirectoryEntry.createReader. In the example we write into the current (root) directory, so we simply use DirectoryEntry.getFile to open a file of a given name. On success we receive FileEntry representing the open file. The object has a few properties

(FileEntry.fullPath, FileEntry.isDirectory, FileEntry.isFile, FileEntry.name) and methods FileEntry.file and FileEntry.createWriter. The first method returns File object, which can be used to read file content and the second we use to write in the file. By the time the operation is complete we read from the file. For that we create a FileReader object and make it reading our File object as text.

# Boosting performance with JavaScript workers

JavaScript is a single-threaded environment. So multiple scripts cannot really run simultaneously. Yes, we use setTimeout(), setInterval(), XMLHttpRequest and event handlers to run tasks asynchronously. So we gain non-blocking execution, but that doesn’t mean concurrency. However by using web workers we can run one or more scripts in background independently of the UI scripts. Web workers are long running scripts that are not interrupted by blocking UI events. Web workers utilize multithreading, so we can benefit from multi-core CPUs.

Well, where can we use web workers? Anywhere where we do processor-intensive calculations and don’t want them blocking the UI thread. It can be graphics, web-games, crypto and Web I/O. We cannot manipulate the DOM from a web worker directly, but we have access XMLHttpRequest, Web Storage, IndexedDB, FileSystem API, Web Sockets and other features.

So let’s see what these web workers are in practice. By and large we register an existing web worker in the main script and communicate to web worker using PostMessage API (https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage).

**index.html**

<html>

<body>

<script>

"use strict";

// Register worker

var worker = new Worker( "./foo-worker.js" );

// Subscribe for worker messages

worker.addEventListener( "message", function( e ) {

console.log( "Result: ", e.data );

}, false );

console.log( "Starting the task..." );

// Send a message to worker

worker.postMessage({

command: "loadCpu",

value: 2000

});

</script>

</body>

</html>

**foo-worker.js**

"use strict";

var commands = {

/\*\*

\* Emulate resource-consuming operation

\* @param {Number} delay in ms

\*/

loadCpu: function( delay ) {

var start = Date.now();

while (( Date.now() - start ) < delay );

return "done";

}

};

// Workers don't have access to the window object.   
// To access global object we have to use self object instead.

self.addEventListener( "message", function( e ) {

var command;

if ( commands.hasOwnProperty( e.data.command ) ) {

command = commands[ e.data.command ];

return self.postMessage( command( e.data.value ) );

}

self.postMessage( "Error: Command not found" );

}, false );

Here in index.html we request the web worker (“foo-worker.js”), subscribe for worker messages and request it to load the CPU for 2000ms, what represents a resource-consuming process. The worker receives the message and checks for a function specified in command property. If it exists, the workers pass message value to the function and replies with the return value.

Note that despite of launching so expensive process by starting up index.html the main thread stays non-blocked. Nonetheless it reports in console when the process is complete. But if you try to run loadCpu function with in the main script, the UI freezes and most probably result in script-timeout error. Now consider this: if you call loadCpu asynchronously (e.g. with setTimeout) the UI still hangs. The only safe way to deal with processor-sensitive operations is to hand them over web workers.

Web workers can be dedicated and shared. A dedicated worker is accessible by only script, the one where we call the worker. Shared workers can be accessed from multiple scripts, even running in different windows. That makes the API a bit different:

**index.html**

<script>

"use strict";

var worker = new SharedWorker( "bar-worker.js" );

worker.port.onmessage = function( e ) {

console.log( "Worker echoes: ", e.data );

};

worker.onerror = function( e ){

console.error( "Error:", e.message );

};

worker.port.postMessage( "Hello worker" );

</script>

**bar-worker.js**

"use strict";

onconnect = function( e ) {

var port = e.ports[ 0 ];

port.onmessage = function( e ) {

port.postMessage( e.data );

};

port.start();

};

Example worker simply echoes the received message. If it did some effective computation we would be able to command it from different script on different pages.

These examples show use of web workers for concurrent computation. What about unloading the main thread from some of Web I/O operations? For example, we are requested to report specified UI events to a remote BI server (Business Intelligence Server, here used to receive statistical data). That is none of core functionality so it would be great to keep any loads these requests produce out of the main thread. So we can go with a web worker. However a worker gets available only after it’s loaded. Normally it happens very fast, but I still want to be sure no BI events lost because of worker were unavailable. What can I do is to embed web worker code into HTML and register web worker by data URI:

<script data-bind="biTracker" type="text/js-worker">

"use strict";

// Here shall go you BI endpoint

const REST\_METHOD = "http://www.telize.com/jsonip";

/\*\*

\* @param {Map} data - BI request params

\* @param {Function} resolve

\*/

var call = function( data, resolve ) {

var xhr = new XMLHttpRequest(),

params = data ? Object.keys( data ).map(function( key ){

return key + "=" + encodeURIComponent( data[ key ] );

}).join( "&" ) : "";

xhr.open( "POST", REST\_METHOD, true );

xhr.addEventListener( "load", function() {

if ( this.status >= 200 && this.status < 400 ) {

return resolve( this.response );

}

console.error( "BI tracker - bad request " + this.status );

}, false );

xhr.addEventListener( "error", console.error, false );

xhr.responseType = "json";

xhr.setRequestHeader( "Content-Type", "application/x-www-form-urlencoded" );

xhr.send( params );

};

/\*\*

\* Subscribe to window.onmessage event

\*/

onmessage = function ( e ) {

call( e.data, function( data ){

// respond back

postMessage( data );

})

};

</script>

<script type="text/javascript">

"use strict";

window.biTracker = (function(){

var blob = new Blob([ document.querySelector( "[data-bind=\"biTracker\"]" ).textContent ], {

type: "text/javascript"

}),

worker = new Worker( window.URL.createObjectURL( blob ) );

worker.onmessage = function ( oEvent ) {

console.info( "Bi-Tracker responds: ", oEvent.data );

};

return worker;

}());

// Let's test it

window.biTracker.postMessage({ page: "#main" });

</script>

By handing over Web I/O to a worker we can also get additional control over it. For example in reaction on network status change (ononline, onoffline events and navigator.online property available to workers) we can respond to application either with actual call results or cached ones. In other words we can make our application working offline. In fact there is special type of JavaScript workers called Service Worker. Service workers inherit from Shared Workers and acts as a proxy between web application and network (<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Social_API/Service_worker_API_reference>).

# Creating our first web component

You are likely familiar with HTML5 video element (http://www.w3.org/TR/html5/embedded-content-0.html#the-video-element). By placing a single element in your HTML you get a widget that runs a video. The element accepts a number of attributes to set up the player. If you want to enhance it you can use its public API and subscribe listeners on its events (http://www.w3.org/2010/05/video/mediaevents.html). So we reuse that element whenever we need a player and only customize it for project-relevant “look & feel”. If we only had enough of these elements to pick one every time we need a widget on a page... It’s not the right way to include any widget that one may need into HTML spec. However the API to create custom elements like video is already there. We can really define an element, package the compounds (JavaScript, HTML, CSS, images and so on) and then just link it from the consuming HTML. In other words, we can create an independent and reusable web component, that we then use by placing the corresponding custom element (<my-widget />) in our HTML. We can re-style the element and if needed and we can utilize the element API and events. For example, if you need a date picker, you can take an existing web component, let’s say http://component.kitchen/components/x-tag/datepicker. All what we have to do is to download the component sources (e.g. by using bower package manager) and link to it from our HTML code:

<link rel="import" href="bower\_components/x-tag-datepicker/src/datepicker.js">

and declare it in the HTML:

<x-datepicker name="2012-02-02"></x-datepicker>

It is supposed to go smoothly in latest versions of Chrome, but won’t probably work in other browsers. Running a web component requires a number of new technologies unlocked in client browser such as Custom Elements, HTML Imports, Shadow DOM and Templates. JavaScript templates we examined in “Chapter 1: Diving into JavaScript core”. Custom Element API allows us to define a new HTML elements, their behavior and properties. Shadow DOM encapsulates DOM sub-tree required by a custom element. And support of HTML imports assumes that by a given link the user-agent enables a web-component by including its HTML on a page. We can use a polyfill (http://webcomponents.org/) to ensure support for all of the required technologies in all the major browsers:

<script src="./bower\_components/webcomponentsjs/webcomponents.min.js"></script>

Do you fancy on writing your own web components? Let’s do it. Our component acts similar to HTML details/summary. When one clicks on summary the details show up. So we create x-details.html where we put component styles and JavaScript with component API:

**x-details.html**

<style>

.x-details-summary {

font-weight: bold;

cursor: pointer;

}

.x-details-details {

transition: opacity 0.2s ease-in-out, transform 0.2s ease-in-out;

transform-origin: top left;

}

.x-details-hidden {

opacity: 0;

transform: scaleY(0);

}

</style>

<script>

"use strict";

/\*\*

\* Object constructor representing x-details element

\* @param {Node} el

\*/

var DetailsView = function( el ){

this.el = el;

this.initialize();

},

// Creates an object based in the HTML Element prototype

element = Object.create( HTMLElement.prototype );

/\*\* @lend DetailsView.prototype \*/

Object.assign( DetailsView.prototype, {

/\*\*

\* @constracts DetailsView

\*/

initialize: function(){

this.summary = this.renderSummary();

this.details = this.renderDetails();

this.summary.addEventListener( "click", this.onClick.bind( this ), false );

this.el.textContent = "";

this.el.appendChild( this.summary );

this.el.appendChild( this.details );

},

/\*\*

\* Render summary element

\*/

renderSummary: function(){

var div = document.createElement( "a" );

div.className = "x-details-summary";

div.textContent = this.el.dataset.summary;

return div;

},

/\*\*

\* Render details element

\*/

renderDetails: function(){

var div = document.createElement( "div" );

div.className = "x-details-details x-details-hidden";

div.textContent = this.el.textContent;

return div;

},

/\*\*

\* Handle summary on click

\* @param {Event} e

\*/

onClick: function( e ){

e.preventDefault();

if ( this.details.classList.contains( "x-details-hidden" ) ) {

return this.open();

}

this.close();

},

/\*\*

\* Open details

\*/

open: function(){

this.details.classList.toggle( "x-details-hidden", false );

},

/\*\*

\* Close details

\*/

close: function(){

this.details.classList.toggle( "x-details-hidden", true );

}

});

// Fires when an instance of the element is created

element.createdCallback = function() {

this.detailsView = new DetailsView( this );

};

// Expose method open

element.open = function(){

this.detailsView.open();

};

// Expose method close

element.close = function(){

this.detailsView.close();

};

// Register the custom element

document.registerElement( "x-details", {

prototype: element

});

</script>

Further in JavaScript code we create an element based on generic HTML element (Object.create( HTMLElement.prototype )). Here we could inherit from a complex element (e.g. video) if needed. We register a custom element ‘x-details’ by using earlier created one as prototype. With element.createdCallback we subscribe a handler that will be called when a custom element created. There we attach our View to the element to enhance it with the functionality we intend for it. Now we can use the component in HTML:

<!DOCTYPE html>

<html>

<head>

<title>X-DETAILS</title>

<!-- Importing Web Component's Polyfill -->

<!-- uncomment for non-Chrome browsers

script src="./bower\_components/webcomponentsjs/webcomponents.min.js"></script-->

<!-- Importing Custom Elements -->

<link rel="import" href="./x-details.html">

</head>

<body>

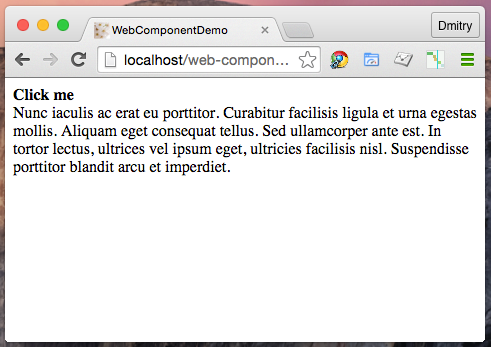
<x-details data-summary="Click me">

Nunc iaculis ac erat eu porttitor. Curabitur facilisis ligula et urna egestas mollis. Aliquam eget consequat tellus. Sed ullamcorper ante est. In tortor lectus, ultrices vel ipsum eget, ultricies facilisis nisl. Suspendisse porttitor blandit arcu et imperdiet.

</x-details>

</body>

</html>



Insert Image 04-02-01.png

# Learning to use server-to-browser communication channels

By using XHR or Fetch API we can request a state from the server. This is a one-way communication. If we want real-time we need it also in the opposite direction. For example we may want user notifications (“your post has been liked”, “new comment”, “new private message”) to pop up as soon as the corresponding records changed in the DB. The server-side has connection to the DB, so we expect the server to notify the client. In past to receive these events on the client we were using tricks known under umbrella term COMET (hidden iframe, long polling, tag long polling and others). Now we can go with native JavaScript APIs.

## Server-Sent Events

The technology that does provides a way to subscribe to server side events is named respectively - Server-Side Events API (SSE). On the client we register a server stream (EventSource) and subscribe to event coming from it:

var src = new EventSource( "./sse-server.php" );

src.addEventListener( "open", function() {

console.log( "Connection opened" );

}, false);

src.addEventListener( "error", function( e ) {

if ( e.readyState === EventSource.CLOSED ) {

console.error( "Connection closed" );

}

}, false );

src.addEventListener( "foo", function( e ) {

var data = JSON.parse( e.data );

console.log( "Received from the server:", data );

}, false);

Here we subscribe a listener to a specific event “foo”. If you want your callback to invoke on every server event, just use src.onmessage. As for server side we just need to set MIME type text/event-stream and send event payload blocks separated with pairs of newlines.

event: foo\n

data: { time: "date" }\n\n

SSE works via HTTP connection, so we need a web server to create a stream. PHP is considerably simple and widely used server-side language is PHP. Chances are - you are already familiar with its syntax. In the other hand PHP isn’t designed for long-held persistent connection. Yet we can trick it by declaring a loop that makes our PHP script never ending:

<?PHP

set\_time\_limit( 0 );

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

header("Cache-Control: no-cache");

date\_default\_timezone\_set("Europe/Berlin");

function postMessage($event, $data){

echo "event: {$event}", PHP\_EOL;

echo "data: ", json\_encode($data, true), PHP\_EOL, PHP\_EOL;

ob\_end\_flush();

flush();

}

while (true) {

postMessage("foo", array("time" => date("r")) );

sleep(1);

}

You may have seen SSE examples where the server script outputs the data once and terminates the process (e.g. http://www.html5rocks.com/en/tutorials/eventsource/basics/). It’s also a working example, because every time the connection is terminated by server the browser renew the connection. But this way we have no benefits of SSE, that works the same as polling.

Now everything looks ready, so we can run HTML. As we do it we get the following output in the console:

Connection opened

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:54 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:55 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:56 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:57 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:58 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:31:59 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:32:00 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:32:01 +0200"}

Received from the server: Object { time="Tue, 25 Aug 2015 10:32:02 +0200"}

...

## WebSockets

Well, with XHR/Fetch we communicate from client to server. With SSE we do it in the opposite direction. But can we have communication in both ways at once? Another HTML5 goody called WebSockets provides bi-directional, full-duplex client-server communications

The client side looks similar to SEE. We just register WebSocket server, subscribe to its events and send to it ours.

var rtm = new WebSocket("ws://echo.websocket.org");

rtm.onopen = function(){

console.log( "Connection established" );

rtm.send("hello");

};

rtm.onclose = function(){

console.log( "Connection closed" );

};

rtm.onmessage = function( e ){

console.log( "Received:", e.data );

};

rtm.onerror = function( e ){

console.error( "Error: " + e.message );

};

This demo source ws://echo.websocket.org simply echoes any messages sent to it:

Connection established

Received: hello

Crave for something more practical? I believe the most illustrative case would be a chat.

demo.html

<style>

input {

border-radius: 5px;

display: block;

font-size: 14px;

border: 1px solid grey;

margin: 3px 0;

}

button {

border-radius: 5px;

font-size: 14px;

background: #189ac4;

color: white;

border: none;

padding: 3px 14px;

}

</style>

<form data-bind="chat">

<input data-bind="whoami" placeholder="Enter your name">

<input data-bind="text" placeholder="Enter your msg" />

<button type="submit">Send</button>

</form>

<h3>Chat:</h3>

<output data-bind="output">

</output>

<script>

var whoami = document.querySelector( "[data-bind=\"whoami\"]" ),

text = document.querySelector( "[data-bind=\"text\"]" ),

chat = document.querySelector( "[data-bind=\"chat\"]" ),

output = document.querySelector( "[data-bind=\"output\"]" ),

// create ws connection

rtm = new WebSocket("ws://localhost:8001");

rtm.onmessage = function( e ){

var data = JSON.parse( e.data );

output.innerHTML += data.whoami + " says: " + data.text + "<br />";

};

rtm.onerror = function( e ){

console.error( "Error: " + e.message );

};

chat.addEventListener( "submit", function( e ){

e.preventDefault();

if ( !whoami.value ) {

return alert( "You have enter your name" );

}

if ( !text.value ) {

return alert( "You have enter some text" );

}

rtm.send(JSON.stringify({

whoami: whoami.value,

text: text.value

}));

});

</script>

Here we have a form with two input fields. The first expects person’s name and the second the chat message. When the form is submitted values of both inputs are sent to WebSocket server. Server response is displayed in the output element. Unlike SSE WebSockets require a special protocol and server implementation to get working. To run the example we will take a simple nodejs-based server implementation nodejs-websocket (https://github.com/sitegui/nodejs-websocket)

**ws.js**

/\*\* @type {module:nodejs-websocket} \*/

var ws = require( "nodejs-websocket" ),

/\*\* @type {Server} \*/

server = ws.createServer(function( conn ) {

conn.on( "text", function ( str ) {

console.log( "Received " + str );

broadcast( str );

});

}).listen( 8001 ),

/\*\*

\* Broadcast message

\* @param {String} msg

\*/

broadcast = function ( msg ) {

server.connections.forEach(function ( conn ) {

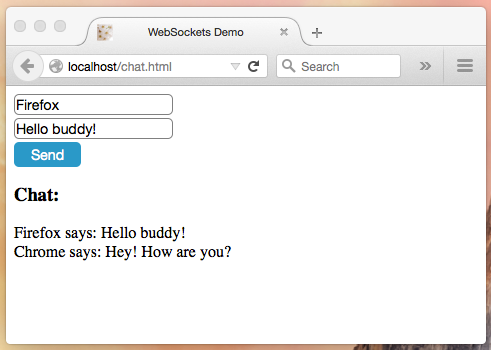
conn.sendText( msg );

});

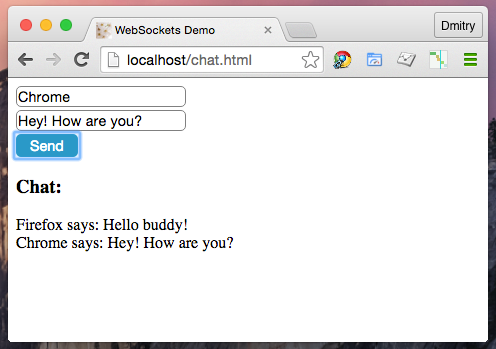
};

The script creates a server available on the port 8001, that listens to WebSocket messages and when any received broadcasts it to all the available connections. We can fire up the server like this:

node ws.js

Now we open our demo chat in two different browsers. When we send a message from one of them the message shows up in both browsers. 

Insert Image 04-03-01.png



Insert Image 04-03-02.png

Note how fast the clients react to the events. Communication through sockets gives irrefutable advantages.

There are a number of WebSocket server implementations for various languages e.g. Socket.IO (http://socket.io) for Node.js, Jetty (http://www.eclipse.org/jetty) for Java, Faye (http://faye.jcoglan.com) fur Ruby, Tornado (http://www.tornadoweb.org) for Python and even one for PHP called Ratchet (http://socketo.me). However I would like to draw your attention to language-agnostic WebSocket daemon - Websocketd (http://websocketd.com/). It’s like CGI (Common Gateway Interface), but for WebSockets. so you can write the server login in your favorite language and then attach your script to the daemon:

websocketd --port=8001 my-script

# Summary

HTML5 provide a number of awesome APIs and some we just examined. Among Browser Storage APIs there are LocalStorage and SessionStorage that extend Cookies relict. Both capable of storing megabytes of data and can be easily synchronize across different browser windows/tabs. IndexedDB allows us to store even greater amounts of data and provides an interface for high-performance searches using indices. We can also use FileSystem API to create and operate a local file system bound to the web-application.

While JavaScript is a single-threaded environment, we still can run scripts in multiple threads. We can register dedicated or shared Web Workers and hand over any processor-intensive operations leaving the main thread and UI unaffected. We also can leverage a special kind of JavaScript Workers - Service Workers as proxy between web application and network. What makes available to control network IO when the browsers switches mode online/offline.

Nowadays we can create own custom advanced elements, that can be easily reused, restyled and enhanced. The assets required to render such elements view such HTML, CSS, JavaScript, images are bundled as Web Components. So we literally we can build the Web now from the components similar to how building are made from bricks.

In past we used tricks known as COMET to exchange events between server and client. Now we can go with Server-Side Events API to subscribe for server events sent over HTTP. We can also use Web Sockets for bi-directional, full-duplex client-server communications.