HTML5

W3C

- The main international standards organization for the world wide web
- Was founded in 1994
- Any one can join (after approval)
- A member pays annual fee
  - Big player ~70000$
  - Small player ~8000$
- As of 2014 has 384 member
- Is criticized for
  - Being dominated by the big players
  - Slow pace

W3C Ratification Process

- Working Draft (WD)
- Candidate Recommendation (CR)
- Proposed Recommendation (PR)
- W3C Recommendation (REC)
- Working Group Note (NOTE)

W3C Vision – Year 2002

- XHTML is the future, not HTML
- XHTML 2.0 is not backward compatible with HTML 4.01 and XHTML 1.1
  - XForms instead of HTML Forms
  - XFrames instead of HTML frames
- Mozila and Opera presented a paper for HTML next generation but were denied
- As result WHATWG was born

WHATWG

- A community of people interested in evolving HTML and related technologies
- Founded by individuals from Apple, Firefox and Opera at 2004
- On 9 May 2007 W3C decided to adopt WHATWG's HTML5
- XHTML 2.0 is dead !!!

What is HTML5?

- A snapshot of WHATWG specification that is managed by W3C
- Offers new elements, new attributes and APIs
- De facto features are now documented
- Is designed so that old browser can safely ignore new features
- A better platform for building complex web application

HTML5 Status

- As of September 2014 HTML5 is considered Proposed Recommendation (PR)
  - The criteria – Two 100% complete and fully interoperable implementations
- WHATWG continues its work on HTML5 as a "living standard"
  - Never completed
  - Always updated and improved
  - New features are added but old functionality is not removed

What's new?

- Depends on who you ask ☺
- The W3C specification is stable
- But we as developers are interested in capabilities not specifications
- A standard is useful only if implemented widely
- Support for a feature may vary between browsers vendors
- You should always check for current feature support before using it
  - http://caniuse.com/

Interesting Features

Semantic Tags	Content Editable	New Input Types	Placeholder
FORM Validation	Local Storage	Autofocus	Videos and Audio Tags
SVG	Geolocation	Indexed DB	Web Sockets
Web Workers	History API	Drag & Drop	Offline Application
Web Font	XMLHttpRequest	DOM Selection	Canvas 2D
Media Capture	Web Messaging	File API	

History API

- Access to the browser's history is offered through the history object
  - Long before HTML5
  - Limited write operations
- Effectively as if the user pressed the back and forward buttons

```
window.history.back();      // go back
window.history.forward();   // go forward
window.history.go(3);       // go 3 pages forward

window.history.length       // number of history entries
```

- Full page reload ☹

History API – HTML5

- New APIs
  - history.pushState
  - history.replaceState
  - window.onpopstate
- The new API allows the developer to programmatically change the URL address
- No page reload ☺
- IE10+, Good browser support

history.pushState

- Suppose http://myws.com/admin/logins executes the following script

```
window.history.pushState({}, null, "state1");
```

- URL bar changes to http://mywebsite.com/admin/state1
- Browser **does not load** state1 from the server
- You may use absolute path

```
window.history.pushState({}, null, "/state1");
```

- URL changes to http://mywebsite.com/state1

pushState Parameters

- state – Any JavaScript object
  - Is associated with the new entry
  - Can be extracted later when the user navigates back/forward to this new entry
  - The browser holds a copy not a reference
- title – Not used
- url – The browser's URL
  - Must be of the same origin
  - If omitted the current URL is used

popstate Event

- Is dispatched to the window object when the active history entry changes

```
$(window).bind("popstate", function (e) {
     console.log("popstate: " + e.originalEvent.state);
});
```

- If the entry was created by pushState/replaceState the event's state property contains a copy of the original state
- Current state is also available through

```
console.log("Current state is: " + history.state);
```

State Serialization

- pushState serializes the state object
- popstate event deserializes it back
- You loose any metadata on the object
  - For example, prototype chaining
- Originally, some browser used JSON serialization
  - Cyclic references cause a runtime error
- Latest browsers use the “Structured clone algorithm”
  - Allows any structured graph to be serialized

History API Notes

- popstate event is not triggered on page load
- popstate event is not triggered when calling pushState/replaceState
- You probably want to wrap all details under navigate function
- There is no way to clear browser's history
- Use polyfills
  - HistoryJS - https://github.com/browserstate/history.js/
  - Backbone Router - http://backbonejs.org/#Router

Web Storage

- Passing data/state from one page to another is a common web task
- Common solutions
  - Query string – Limited storage
  - Session state – Does not scale well
  - Cookie – Limited storage
  - Hidden field – Reset upon GET request
  - URL – Limited storage

Web Storage API

- Designed to provide a larger, more secure, easier to use alternative to cookies
- Allows for key/value pairs to be stored and retrieved
- Is great for building offline web application
  - Web storage is accessible even when there is no internet connectivity
- Offers the following objects
  - sessionStorage
  - localStorage

Storage Interface

- Both sessionStorage and localStorage implements the same API
  - getItem(key)
  - setItem(key, value)
  - removeItem(key)
  - clear
- Key and value are strings !!!
- When passing object as a key or as a value it will first be converted to a string using toString

sessionStorage

- Not related to server side session management
- Each browser window/tab holds a sessionStorage object per origin
- Should live as long as the window/tab is alive
- When duplicating a tab the sessionStorage should be duplicated too
  - No sharing

```
sessionStorage.setItem("data", data);
```

```
var data = sessionStorage.getItem("data");
```

Serialization

- Web storage support only strings (key & value)
- Therefore, data must serialized/deserialized
- Can use JSON.stringify & JSON.parse
  - Cannot save cyclic references
  - Prototype is lost
  - String representation is expensive

```
var str = JSON.stringify(data);
localStorage.setItem("data", str);
```

```
var str = localStorage.getItem("data");
var data = JSON.parse(str);
```

localStorage

- A Storage object per origin
- Storage should not be cleared by browser
  - Only on rare conditions
- Different windows with same origin share the same loadStorage object
  - Browser should protect against concurrent access
- Exception is thrown under Incognito mode

```
localStorage.setItem("data", data);
```

```
var data = localStorage.getItem("data");
```

Disk Space

- Browser should limit the total amount
  - Specification does not mention the limit itself
  - 5MB is common
- Browser may prompt the user when quota is reached
  - Only supported on opera
- On some browsers the administrator can change the limit
  - Not from Java Script

storage Event

- Is fired when calling setItem/removeItem/clear
- Contains the following
  - key
  - oldValue
  - newValue
  - url – The url of the window that changes the data
- Event should not be fired on the window that changes the storage
  - IE disagrees

```
window.addEventListener("storage", function (e) {
     console.log(e);
}, false);
```

Web Storage Notes

- Updating a single record may be inefficient
  - Deserialize all data into memory
  - Fix memory
  - Serialize back
- API is blocking
  - UI is blocked while saving big data
- Should be careful when application is running under shared host
  - Wix.com

Indexed DB

- Web storage does not deal well with large data
  - Cannot update single record
  - Blocking API
- Indexed DB
  - Transactional database
  - Lets you store and retrieve objects
  - Objects are indexed → Better search performance
  - No schema
  - Asynchronous API

Browser Support

- Desktop
  - Firefox, Chrome, Safari for desktop
  - IE11 – Not complete
- Mobile
  - Android 4.4+
  - iPhone 8+ (8 is not released yet)
  - IE Mobile 10+ (Strange …)

Storage Limits

- Storage is limited per origin
- Firefox
  - No limit
  - Asks user after 50MB
- Chrome
  - No limit
- IE
  - 250MB limit
  - Asks user after 10MB

Database

- Has one or more object stores
- Has a name
- Has current version – Initially 0
- There may be multiple connections to a given database
- Each origin has an associated set of databases
- Is described by the IDBDatabase interface

Open Database

- Should specify a version – Default is 1
- Starts with no object store
- Can delete a whole database

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

request.onsuccess = function (e) {
     db = e.target.result;

     console.log("current version: " + db.version);
}

request.onerror = function (e) {
     console.log("open db error: " + e.target.error.message);
}

request.onupgradeneeded = function (e) {
     console.log("DB Upgrade is needed");
}
```

Object Store

- Has a unique name
- Has a list of records
- Has a list of indexes
- No schema
- Each record consists of a key and a value
- The list is sorted according to key
- Key is unique
- Is described by the IDBObjectStore interface

Create Object Store

- Creating new object store is only allowed during upgradeneeded event

```
request.onupgradeneeded = function (e) {
     console.log("DB Upgrade is needed");

     console.log("old version: " + e.oldVersion);
     console.log("new version: " + e.newVersion);

     var db = e.target.result;

     if (e.newVersion == 1) {
         console.log("Creating object store: contacts");

         var objectStore = db.createObjectStore("contacts",
                                { keyPath: "id", autoIncrement: true });

         console.log("Creating indexes");

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

IDBObjectStore

- Properties
  - name
  - keypath
  - autoIncrement
  - indexNames
  - transaction
- Methods
  - get
  - add
  - put
  - delete
  - clear
  - index
  - createIndex/deleteIndex

Key

- Must be: number, string, Date or Array (not sparse)
- Sort order is according to language natural sort
- Can be
  - Extracted from the record itself: Use keypath

```
var objectStore = db.createObjectStore("contacts", { keyPath: "id" });
```

  - Generated by key generator: Use autoIncrement

```
var objectStore = db.createObjectStore("contacts", { autoIncrement: true });
```

  - Explicitly specified as part of the insertion

Read Single Record

- Use the get method
- Must specify the key to look for

```
var tran = db.transaction(["contacts"], "readonly");
tran.oncomplete = function () { }
tran.onerror = function (e) { }

var contacts = tran.objectStore("contacts");

var key = 1001;
var request = contacts.get(key);

request.onsuccess = function (e) {
     var contact = e.target.result;
     if (contact) {
         console.log("Item found: " + contact.name);
     }
     else {
         console.log("Item not found");
     }
}

request.onerror = function () {console.error("get error"); }
```

Read Range of Records

- Open a cursor
- Must call continue to get the next record

```
var contacts = getStore("contacts", "readonly");

var request = contacts.openCursor();

request.onsuccess = function (e) {
     var cursor = e.target.result;
     if (cursor) {
          var key = cursor.key;
          var value = cursor.value;
          console.log(key + ": " + value.id + ", " + value.name);

          cursor.continue();
     }
     else {
          console.log("No more entries");
     }
}

request.onerror = function () {
     console.log("error");
}
```

Read using an Index

- Retrieve the index and open a cursor

```
var range = IDBKeyRange.bound("Ori", "Roni", false, false);

var contacts = getStore("contacts", "readonly");
var index = contacts.index("name");

var request = index.openCursor(range, "prev");

request.onsuccess = function (e) {
     var cursor = e.target.result;
     if (cursor) {
         var key = cursor.key;
         var value = cursor.value;
         console.log(key + ": " + value.name + ", " + value.email);

         cursor.continue();
     }
     else {
         console.log("No more entries");
     }
}

request.onerror = function () { console.log("error"); }
```

Insert

- Must create a readwrite transaction
- Use add method
- Watch for success and completion of the transaction

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {...}
tran.onerror = function (e) {...}
tran.onabort = function () {...}

var contacts = tran.objectStore("contacts");
var request = contacts.add({id: 1001, name: "Ori",email: "ori@gmail.com"});

request.onsuccess = function (e) {
     var key = e.target.resolve;};

request.onerror = function (e) {
     console.error("add error " + name + ": " + e.target.error.message);
};
```

No commit ?

- indexedDB supports auto committed transaction
- There is no method named commit ☺
- There is a method named abort
- So, when is transaction committed ?
  - When there are no pending change requests on the current transaction
  - And the thread returns to the browser's message loop
- Therefore,
  - You cannot postponed transaction commit
  - Once a transaction finishes any manipulation on it causes a runtime error

Concurrent Transaction

- Two readwrite transactions with overlapping scope block each other
  - Two read transaction do not block
  - Concurrent read and write transaction do not block
- The first created transaction must be completed and only then the second can be completed too
- This does not mean that your code blocks ☺
  - The onXXX are postponed until the transaction can be completed

Transaction events

- onerror is raised for every single operation that fails during the transaction
  - Use e.target.error for more details
- onabort is raised only once
- oncomplete is raised only once

```
var tran = db.transaction(["contacts"], “readwrite");

tran.oncomplete = function () {
     console.log("tran complete");
}

tran.onerror = function (e) {
     console.error("tran error: " + e.target.error.message);
}

tran.onabort = function () {
     console.error("tran abort");
}
```

Aborting Transaction

- Is aborted automatically if one of the change operation fails
- Can be aborted manually using IDBTransaction.abort
- Can throw an exception from one of the change onsuccess handlers
- Once oncomplete is raised the transaction is considered completed and cannot be aborted

Update

- There is no strict update method
- The put method updates or inserts an object

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {}
tran.onerror = function (e) {}

var contacts = tran.objectStore("contacts");

var contact = { id: 1001, name: "XXX“ };

var request = contacts.put(contact);

request.onsuccess = function () {
     console.log("put success");
};

request.onerror = function () {
     console.log("put error");
};
```

Delete

- Must retrieve the object key
- Use the delete method

```
var tran = db.transaction(["contacts"], "readwrite");

tran.oncomplete = function () {...}
tran.onerror = function (e) {...}

var contacts = tran.objectStore("contacts");

var request = contacts.delete(1);

request.onsuccess = function () {
     console.log("delete success");
};

request.onerror = function () {
     console.log("delete error");
};
```

Summary

- indexedDB offers rich API for dealing with data
- Great for storing heterogeneous objects
- Quite complex API
  - With respect to the simplicity of localStorage
- Good desktop support
- No iPhone support (yet)
  - Can be implemented using Web SQL

File API

- A standard way to interact with local files
- Offers the following objects
  - File - A reference to a file
  - FileList - A collection of File objects
  - Blob - Maniulate File's data
  - FileReader – Read file asynchronously
  - URL Scheme – Binary data inside URL
- IE10+

Getting Started

- Web script cannot access a file on the file system
- However, it can
  - Use a form file input
    - User browses for the file
    - File is accessible through the input element
  - Drag & Drop
    - User drags and drops a file on the page
    - File is accessible through the dataTransfer property of the event object

Using input field

- Use multiple attribute to allow multi file selection

```
<input type="file" multiple />
```

- Can hide the input and then programmatically trigger the dialog open

```
input[type=file] {
     display: none;
}
```

```
$(".browse").click(function () {
     fileInput.trigger("click");
});
```

- Use files property on the input element

```
$.each(fileInput[0].files, function () {
     var file = this;
     console.log(file.name + ": " + file.size);
});
```

File Object

- The files property of the input element is of type FileList
  - Which is a collection of File objects
- Each File object offers the following API
  - type – Mime type
  - name – Short file name (without path)
  - size – File size in bytes
  - lastModifiedDate – Modification date
  - slice – Returns a blob object that represents the specified byte range

FileReader

- Provides methods to read a File or Blob object into memory
- Asynchronous API
- Fires progress events
- Supported formats: Text, DataURL, binary
- Can read the whole file content or multiple slices
- IE10+

FileReader API

- Read whole file
  - readAsArrayBuffer
  - readAsText
  - readAsDataUrl
- readyState attribute – EMPTY/LOADING/DONE
- Should wait for load event
- Result is stored under event.target.result
- Concurrent reads are not allowed
  - Exception is thrown

FileReader Events

- loadstart
- progress
- abort
- error
- load
- loadend

Read Text

- Use readAsText
- You may specify an encoding

```
var fileReader = new FileReader();
fileReader.readAsText(file, "windows-1255");

fileReader.addEventListener("progress", function (e) {
    progress.text(e.target.result.length);
});

 fileReader.addEventListener("load", function (e) {
    var str = e.target.result;
     result.text(str);
});
```

Read Data Url

- Use readAsDataUrl
- You may embed the result inside an img tag

```
var fileReader = new FileReader();

fileReader.readAsDataURL(file);

fileReader.addEventListener("load", function (e) {
     //
     //  load is fired when read operation completed successfully
     //
     var str = e.target.result;
     result.text(str);
     $("img").attr("src", str);
});
```

Read ArrayBuffer

- Need to wrap the ArrayBuffer inside a Typed array object in order to access its content

```
fileReader = new FileReader();

fileReader.readAsArrayBuffer(file);

fileReader.addEventListener("load", function (e) {
     var bufArr = e.target.result;

     var byteArr = new Uint8Array(bufArr);

     console.log("load: " + byteArr.length);

     for (var i = 0; i < 1000; i++) {
         console.log("    byteArr[" + i + "]: " + byteArr[i]);
     }
});
```

Slicing

- Reading a big file into memory is problematic
  - Firefox may even crash (file > 1GB)
- Should read slices instead of whole file

```
function readNextBuffer() {
     if (pos >= file.size) {
        return;
     }

     fileReader = new FileReader();
     fileReader.readAsArrayBuffer(file.slice(pos, pos + bufSize));

     fileReader.addEventListener("load", function (e) {
         pos += e.target.result.byteLength;

         var percentCompleted = Math.round(pos / file.size * 10000) / 100;

         progress.text(percentCompleted + "%");

         readNextBuffer();
     });
}
```

Summary

- Cannot access file system directly
- User should browse to the file or drop it
- File metadata is accessible using a File object
- Reading a file content is done using a FileReader object
- Content can be read as: Text, DataUrl, ArrayBuffer

Form Validation

- Validate user data without Java Script
- Is supported through a list of HTML attributes
- Can control element styling through CSS
- Can query validation status using JavaScript API
- IE10+
- No iPhone support ☹

Pattern Attribute

- All input elements can be associated with the pattern attribute
  - Text area is not supported
- The attribute expects a case sensitive Regular Expression as its value
- Empty value is not validated against the pattern
  - Use required attribute

```
<input type="text" pattern="(ab)*" />
```

Validation Lifecycle

- On page load
  - pseudo-class :invalid is attached to failed elements
- During typing
  - pseudo-class :invalid is attached/detached according to element's value
- During form submission
  - invalid DOM event is raised for every failed element
  - A popup message is displayed
  - submit event is not fired

More Validation Attributes

- required

```
<input type="text" required />
```

- maxLength – Usually is enforced during typing

```
<input type="text" maxlength="10" />
```

- min & max & step – Only for number field

```
<input type="number" min="1000" max="1200" step="2" />
```

- type – email & url

```
<input type="email" />
<input type="url" />
```

Notes

- The developer is responsible for setting CSS styling according to :invalid class
- Usually, only first failure is displayed to the user
- Message content changes between browsers
- Every browser styles its popup messages differently
- Messages are localized according to browser locale
  - Not page locale

Localizing Validation Messages

- Listen to invalid DOM event
- Reveal the reason of the failure
  - See next slides
- Register custom validation message using setCustomValidity

```
<input type="text" required class="name" />
```

```
$(".name").bind("invalid", function () {
    this.setCustomValidity("ערך חובה");
});
```

Controlling Messages Styling

- Can't really do that. Instead,
- Disable validation using novalidate attribute
  - No invalid event
  - No popup messages
  - :invalid pseudo-class is still attached
- Listen to submit event
- Call checkValidity on the form element
  - invalid event is fired
  - Check the return value
- Display validation messages the way you like

Controlling Messages Styling


```
<form novalidate>
     Name:
     <input type="text" required class="name" />

     <br />
     E-Mail:
     <input type="email" value="not valid email" />

     <br />
     <br />
     <input type="submit" value="Submit" />
</form>
```

```
$(".name").bind("invalid", function () {
     this.setCustomValidity("ערך חובה");
});

$("form").bind("submit", function () {
    if (!this.checkValidity()) {
         var invalidInputs = $("input:invalid");
         if (invalidInputs.length) {
             alert(invalidInputs[0].validationMessage);
         }
         return false;
     }
     return true;
});
```

Constraint Validation API

- Every input element has a property named validity
- Is an object of type ValidityState which supports
  - valid – True when element fails validation
  - valueMissing – required validation failed
  - patternMismatch – pattern validation failed
  - rangeOverflow/rangeUnderflow – min/max failed
  - stepMismatch – Out of possible step values
  - tooLong – maxLength validation failed. Probably never
  - typeMismatch – type email or url validation failed
  - customError – setCustomValidity was called

Summary

- FORM validation is easy
- No need to write JavaScript
- Just throw some attributes
- Probably, not suited to high-end web applications
  - No cross browser compatibility
  - Limited control over styling
  - Is not a hard task to accomplish with plain old JavaScript

Web Worker

- Traditionally,
  - JavaScript is single threaded
  - Running long computation means UI is blocked
- With HTML5 Web Worker support
  - The developer can create multiple web workers
  - Each represent a background thread
  - Long computation no longer blocks the UI
    - The developer is responsible for spawning the worker
  - What about race condition ?

Create a Worker

- Construct a new object using the Worker constructor
- Specify the URL of the script to be executed
  - Same origin policy applies
- Optionally, listen to the message event in case you want to receive messages from the worker

```
var worker = new Worker("/Scripts/Task.js");

worker.addEventListener("message", function (e) {
    ...
});
```

Thread Safety

- Web worker allows for parallel execution
- However, the worker is executed under a new global execution context
  - No sharing of global variables
- It can communicate using special channel
- All non thread safe components are unavailable when running under a worker
  - DOM
  - window
  - document

Passing Data

- Use postMessage method to send the actual data
  - From both ends
- Messages are serialized/deserialized
- This means you get a copy of the original object
- No sharing of the references
- Browsers use the structured clone algorithm
  - Cyclic references are allowed ☺

Passing Data

App.js

```
var worker = new Worker("/Scripts/Task.js");

worker.postMessage(10);

worker.addEventListener("message", function (e) {
     console.log("Result: " + e.data);
});
```

Task.js

```
self.addEventListener("message", function (e) {
     self.postMessage(e.data * 2);
});
```

Single Thread Model

- A single worker represents a single thread
- Like the browser "main" thread a web worker is not interruptible
- When a worker runs some JavaScript code it cannot process incoming messages
- Only when code finishes and returns to the browser's message loop the next queued message is processed

Terminate a Worker

- Use terminate method on the Worker object
  - Kills the worker immediately
  - The worker has no chance to complete current work

```
worker.terminate();
```

- Use close method from the Worker itself
  - Post a message from the main page
  - Close the worker

```
self.close();
```

  - close method returns and only later Worker will be closed

Unhandled Exception

- Unhandled exceptions may be tracked using the error event type
- The event can be monitored from both the worker and its creator
- Unhandled exception does not kill the worker

```
self.addEventListener("error", function (e) {
     console.log(e.message);
});
```

```
worker = new Worker("/Scripts/Task.js");

worker.addEventListener("error", function (e) {
     console.log(e.message);
});
```

Import Scripts

- All non DOM related API is accessible under web worker: Date, Math, JSON and others
- But what about your own custom code?
  - Use importScripts
- Is executed synchronously and returns only after all scripts were executed
- Url is relative to worker's script

```
importScripts("Logger.js");

Logger.message("Web worker is running and using Logger module");
```

Maximum # of Web Workers

- Specification does not mention a limit
  - Firefox has dom.workers.maxPerDomain setting which is 20 by default
  - Chrome crashes when trying to spawn 1000 web workers
  - IE has a limit of 25
- Usually, no error is reported when reaching the limit but rather the worker is queued until a previous worker is closed

Summary

- At last we have threads
- Long computation may be refactored into a background web worker
- UI may become more responsive
- However, NO DOM manipulation is allowed
  - Which means long DOM related computation still blocks the UI

Web Socket Protocol

- Enables two-way communication between browser and server
- Does not rely on opening multiple HTTP connections
- Replacement for older techniques like long polling and forever frame
- A simple abstract over TCP socket
- A totally new application protocol
  - No HTTP headers
- Managed by IETF

Web Socket API

- A JavaScript API
- Is used by the browser to initiate a Web Socket communication with the server
- Is all around the WebSocket class
  - send
  - close
  - readyState
  - onopen, onmessage, onclose, onerror
- Managed by W3C

Getting Started

- Create a new WebSocket object
- Specify URL and sub protocols (Optional)
  - Must use ws or wss protocols
- Monitor open and error events

```
var ws = new WebSocket("ws://localhost:5481/socket/connect");

ws.onerror = function (e) {
     console.log("ERROR");
     console.log(e);
}

ws.onopen = function (e) {
     console.log("OEPN");
     console.log(e);
}
```

The Handshake

- Upon a WebSocket object creation the browser sends an Upgrade request to the server

```
GET http://localhost:5481/socket/connect HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:5481
Origin: http://localhost:5481
Sec-WebSocket-Key: rHeA9NhKxIuFX85mxpTOfQ==
Sec-WebSocket-Version: 13
```

- Server must respond with appropriate headers

Server Response


```
HTTP/1.1 101 Switching Protocols
Cache-Control: private
Upgrade: Websocket
Server: Microsoft-IIS/8.0
Sec-WebSocket-Accept: JLSwLlhPkGnb4qOJOnYzl957T7I=
Connection: Upgrade
```

- After a successful handshake, the data transfer part starts
- This is a two-way communication channel where each side can, independently from the other, send data at will

Send and Receive Messages

- WebSocket object supports
  - send – Can only be used after onopen event was fired
  - onmessage

```
var ws = new WebSocket("ws://localhost:5481/api/socket/connect");

ws.onclose = function (e) {
     ...
}

ws.onopen = function (e) {
    ws.send(message);
}

ws.onmessage = function (e) {
     console.log("MESSAGE: " + e.data);
}
```

Framing

- WebSocket is message based protocol
- The data being sent by the client is considered a message
- A message consists of multiple frames
- Each frame has slight overhead over the original payload
  - 2 bytes – FIN + Opcode + Payload Length + More
  - 4 bytes – Masking key
  - Above is true only for messages <= 125 bytes

Sub Protocol

- Client may specify a list of sub protocols
- Server must response with exactly one matched sub protocol
- WebSocket object contains a property named protocol which holds server selection

```
var ws = new WebSocket("ws://localhost:5481/api/socket/connect", ["myproto1", "myproto2"]);

ws.onopen = function (e) {
     console.log("OPEN: " + e.target.protocol);

     ws.send(message);
}
```

Summary

- Web Sockets brings “realtime-ness” to your web pages
- It is smarter and more efficient then just using polling
- However, you need both modern server and modern browser
- Consider using Web Socket Polyfills like SignalR