# HTML 5 APIs

André Restivo

# Index

# Introduction

# HTML5 Web APIs

In addition to describing HTML5 elements, the HTML 5 specification also introduces a set of *application programming interfaces* (APIs), that can be used by developers in their websites.

Most of these APIs are still under **heavy development**. Some have been discontinued, might be discontinued in the future, or heavily changed or never even see the light of day.

W3 APIs Current Status | MDN List of Web APIs

But first we must talk about *Promises*...

# Promises

# Promises

A *promise* represents the eventual result of an asynchronous operation.

A *promise* may be in one of 3 possible states: **fulfilled**, **rejected**, or **pending**.

A *promise* can be used to return asynchronously from an synchronous function. This can save us from Callback Hell.

```
let promise = new Promise(function(resolve, reject) {
  /* Very long operation */
  if (ok)
    resolve(result)
  else
    reject(error)
})
```

# Consuming

When the *promise resolves*, or is *rejected*, we can use *then* and *catch* to consume it:

```
promise.then(function(result){
  alert('Ok')
}).catch(function(error)){
  alert('Error')
}
```

# Example

Transforming a **synchronous** *XMLHttpRequest* into a *promise*:

```
function getSomeData() {
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest()
    request.open("get", "getdata.php", false) // synchronous
    request.send()
    if (request.status === 200) resolve(request.response)
    else reject(request.statusText)
  })
}
```

```
getSomeData().then(alert).catch(alert)
```

# Chaining

*Promises* can be chained, making successive calls to asynchronous functions much easier:

```
getSomeData()
  .then(function(result){
    return getMoreDataBasedOn(result) // also returns a promise
  })
  .then(function(result){
    return getEvenMoreDataBasedOn(result)
  })
  .then(function(result){
    alert(result)
  })
  .catch(alert) // catches any error in the chain
```

# Async Functions

An **async** function is a function which operates asynchronously, using an **implicit** *Promise* to return its result.

An **async** function always returns a *promise*. If the code returns a *non-promise*, then JavaScript automatically wraps it into a resolved *promise* with that value.

```
async function getSomeData() {
  let request = new XMLHttpRequest()
  request.open("get", "getdata.php", false) // synchronous
  request.send()
  if (request.status === 200)
    return request.response
  else
    throw(new Error('Error getting data'))
}
```

# Await

The keyword *await* makes JavaScript wait until a *promise* settles and returns its result.

```
async function doSomething() {
  let result = await getSomeData() // returns a promise
  console.log(result)
}
```

The keyword *await* can only be used inside *async* functions.

# Promise.all

The *Promise.all(<promises>)* method returns a single *Promise* that resolves when all of the *promises* in the argument have resolved. It rejects with the reason of the first *promise* that rejects.

```
async function doSomething() {
  let promise1 = getSomeData()
  let promise2 = getEvenMoreData()
  Promise.all([promise1, promise2]).then(function(results) {
    console.log(values)
  })
}
```

The result is an array with the results of each one of the *promises*

# Permissions

# Permissions

Some of the new HTML 5 APIs provide powerful features and thus require an API to **query** the current permissions state, and **request/revoke** permissions by the users to use those features.

Currently, the HTML 5 standard describes permissions for the following APIs:

```
geolocation, notifications, push, midi, camera, microphone, speaker,
device-info, background-sync, bluetooth, persistent-storage,
ambient-light-sensor, accelerometer gyroscope, magnetometer, clipboard.
```

# Permission State

To read the current permission state for a certain feature we must use the *navigator.permissions* interface:

```
navigator.permissions.query({ name: "geolocation" }).then(({ state }) => {
  switch (state) {
    case "granted":
      break;
    case "prompt":
      break;
    case "denied":
      break;
  }
})
```

The result can be **granted** (the user already granted us the permission), **denied** (the user denied the permission) or **prompt** (the use hasn't been asked yet).

# Clipboard

# Clipboard API

The Clipboard API allows developers to:

- **Read** data from the clipboard
- **Change** the clipboard data
- **Detect** when the clipboard data changed
- **Intercept** clipboard actions and modify them

W3 Specification | MDN

# Examples

Reading from the clipboard:

```
navigator.clipboard.readText().then(
  function (copiedText) {
    /* ... */
  }
)
```

Writing to the clipboard:

```
navigator.clipboard.writeText(someValue)
```

Intercepting and changing copy events:

```
document.addEventListener('copy', (e) => {
  e.preventDefault();
  e.clipboardData.setData('text/plain', 'Something else!')
})
```

# GeoLocation

# GeoLocation

The GeoLocation API allows a web app to ask the browser for it's current location. This can be done using several different methods (GPS, IP address, 3G triangulation, ...)

To get the current location of the user, we must use the *getCurrentPosition* function of the *navigator.geolocation* API.

This function receives two functions that are called, respectively, if the location was acquired or access to the user location was denied. It also receives a third parameter containing information about the request.

W3 Specification | MDN

# Example

```
let geoSettings = {
  enableHighAccuracy: true,
  maximumAge        : 30000,
  timeout           : 20000
};

navigator.permissions.query({name:'geolocation'}).then(function(result) {
  if (result.state == 'granted') {
    navigator.geolocation.getCurrentPosition(revealPosition,
                                             positionDenied,
                                             geoSettings);
  } else if (result.state == 'prompt') {
    navigator.geolocation.getCurrentPosition(revealPosition,
                                             positionDenied,
                                             geoSettings);
  } else if (result.state == 'denied') {
    console.log('Already denied')
  }
  result.onchange = function() {
    console.log(result.state)
  }
})
```

# Example

Getting the coordinates:

```
function revealPosition(position) {
  console.log(`Latitude: ${position.coords.latitude}`)
  console.log(`Longitude: ${position.coords.longitude}`)
}

function positionDenied() {
  console.log('GeoLocation denied')
}
```

# Web Storage

# Web Storage

The Web Storage API allows web apps to store data in the browser.

Web storage is **more secure** than *cookies*, and it allows for large amounts of data to be stored locally. It also doesn't rely on the **server** — like *session variables* do.

Unlike cookies — that are sent to the server using HTTP headers — data stored using the Web Storage API never leaves the browser.

W3 Specification | MDN

# Storages

The Web Storage API can be accessed using two different interfaces:

- Window.localStorage - stores data with no expiration date, and gets cleared only through JavaScript, or clearing the Browser Cache.
- Window.sessionStorage - data is different for each tab/window and is cleared when the browser is closed.

Either way, data is private for each *server* / *protocol*.

# Storing, Retrieving and Removing

There are several different ways to store data but using the *setItem* method is recommended:

```
localStorage.theme = 'dark'
localStorage['theme'] = 'dark'
localStorage.setItem('theme', 'dark')
```

There are also several different ways to retrieve data but using the *getItem* method is recommended:

```
console.log(localStorage.theme)
console.log(localStorage['theme'])
console.log(localStorage.getItem('theme')
```

To delete data from storage, we can use the *removeItem* method:

```
storage.removeItem('theme')
```

# Storage Events

When a storage changes, it fires a *storage* event. This event is received only in other windows from the same origin and only for the *localStorage*.

```
window.addEventListener('storage', function(e) {
  console.log(`${e.url} changed ${e.key} from ${e.oldValue} to ${e.newValue}`)
})
```

This can be used to synchronize data between different windows/tabs.

# Web Sockets

# Web Sockets

The web has always relied on the proven **client/server**, request/response paradigm.

XMLHTTPRequests and Ajax has started a small revolution but all communication was still **initiated by the client**.

This means that when data changes on the server, the client only knows about it if the **user asks** for new data or if the data is **periodically polled** from the server.

# Long Polling

One solution to this problem is called **long polling**.

When using long polling, the browser opens an HTTP connection to the server which **keeps it open** waiting for any changes to announce.

Although some problems, like keeping a server process occupying resources for each client, have already been solved, and in several different ways, HTTP connections aren't suited for **low latency** applications (games for example).

# Web Sockets

Web Sockets allow opening a two-way interactive communication session between the browser and the server.

This allows the server to push updates instantaneously to the browser instead of waiting for the browser to ask if the data changed.

W3 Specification | MDN

# The Server

A WebSocket server is a **TCP application** listening on any port of a server that follows a specific protocol.

The protocol comprises several steps:

- **Handshake** (request / response)
- Keeping track of **clients**
- Exchanging **data** frames
- Checking if **connected** (ping and pongs)
- **Closing** the connection

But we're not going to discuss the protocol in detail. It is recommended that you use a proven solution. For PHP, Ratchet is the most known and used Web Sockets library.

# The Client

On the client side, things are a little bit easier.

To connect to a Web Socket server we just have to create a socket:

```
let conn = new WebSocket('ws://localhost:8080')
```

When the connection is established, a *open* event is fired:

```
conn.addEventListener('open', function() {
  console.log("Connection established!");
})
```

And every time a message is received from the server a *message* event is fired:

```
conn.addEventListener('message', function(e) {
  console.log(e.data)
})
```

# Other APIs

# Other APIs

- **IndexedDB** is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data.
- The **Web Crypto** is an interface allowing a script to use cryptographic primitives in order to build systems using cryptography: digest, sign, verify, hash, encrypt, decrypt...
- The **Pointer Lock** gives you access to raw mouse movement, locks the target of mouse events to a single element, eliminates limits on how far mouse movement can go in a single direction, and removes the cursor from view (demo).
- **Page Visibility** provides events you can watch for to know when a document becomes visible or hidden.
- **Web Notifications** - Why does this even exist?!? No, I don't want your darn notifications!