The Web Bluetooth Series

Martin Woolley, Bluetooth SIG

Part 2 - Foundations

Introduction

In the first part of this series, I presented my thoughts on the Web Bluetooth API, its potential and its significance and argued for Web Bluetooth becoming a fully-fledged W3C standard, implemented in all the browsers.

In this second part, I want to start us on a path which will deliver a solid technical grounding for developers wishing to use Web Bluetooth. To start with, I'll cover the prerequisite, foundational knowledge that you'll need.

JavaScript

Web Bluetooth is a JavaScript API and therefore to use it, you need to know at least basic JavaScript, preferably a little more than that. Much of what you'll use JavaScript for will be familiar and simple. You'll interact with the Document Object Model (DOM), listen for and respond to UI events and so on.

There are numerous resources for learning JavaScript out there on the web, so if you really are a complete beginner, with no prior experience of JavaScript then my recommendation is that you start out by finding a suitable JavaScript tutorial and complete it. Come back and resume once you're comfortable with the subject.

HTML and CSS

JavaScript can be used on servers but our focus here is JavaScript in the browser. Consequently, to use Web Bluetooth you'll also need to know a bit of HTML and if you want the HTML to look pretty, a bit of CSS. Once again, you're advised to search for a suitable tutorial in each of these topics if you have no prior knowledge.

Asynchronous Patterns

Experienced web developers know that synchronous JavaScript code can be problematic and impact user experience. JavaScript is essentially single threaded and runs in the thread that the browser uses for various other, important things. Therefore, if you execute a relatively long running JavaScript function, you risk delaying other code which relies on that thread, like screen painting. So, popular web developer wisdom decrees that wherever possible, asynchronous coding patterns such as events and call back functions should be used. 命令

There are a few approaches to implementing asynchronous code available to the JavaScript developer, in part depending on the version of ECMAScript being coded for. ECMAScript is the language specification that JavaScript is based upon. JavaScript is said to be a dialect of ECMAScript.

Web Bluetooth works with asynchronous programming constructs known as promises, which were introduced in ECMAScript version 6.

Rather than send you off to read about promises elsewhere, we'll explore them right here, right now[1].

Promises, Promises

The concept of a promise is not specific to JavaScript. It comes from the world of concurrent software and has characteristics that make them better to use than other approaches to asynchronous, concurrent execution.

Conceptually, promises separate the timing of events from the outcome or result which the event communicates.

This is interesting. Imagine we had used JavaScript events to communicate something like a button being pressed. There are two parts to this pattern. The event needs to be fired and something needs to listen for that event and call a specified function when it happens. If the event gets fired before there's something actively listening for it, then the event is lost. The event exists at a moment in time. The event itself (the button being pressed) and the time it occurred are tightly coupled, in fact inseparable concepts in event driven programming.

With promises, if an event occurs before there is a function available to respond to it, the event is not lost. When a function is later made available for events of this type, if an event has already occurred, the newly supplied callback function will be called. This is helpful in web applications where timing the construction and availability of different, dependent parts of the web application can sometimes be tricky.

Promises also offer a stricter contract regarding the states and outcomes they can produce. They can only succeed or fail once and cannot switch states from success to failure or vice versa.

There's a collection of formal terminology associated with promises. The formality of promises is the source of that stricter usage contract and what makes them appealing to use.

The outcome, or "fate" of a promise is "resolved" or "unresolved". There are two ways in which a promise can be resolved or, to put it another way, two states a resolved promise can be in, either "fulfilled" if there was a positive outcome or "rejected" if there was a negative outcome, such as an error. The state of a promise which is not yet resolved is said to be "pending".

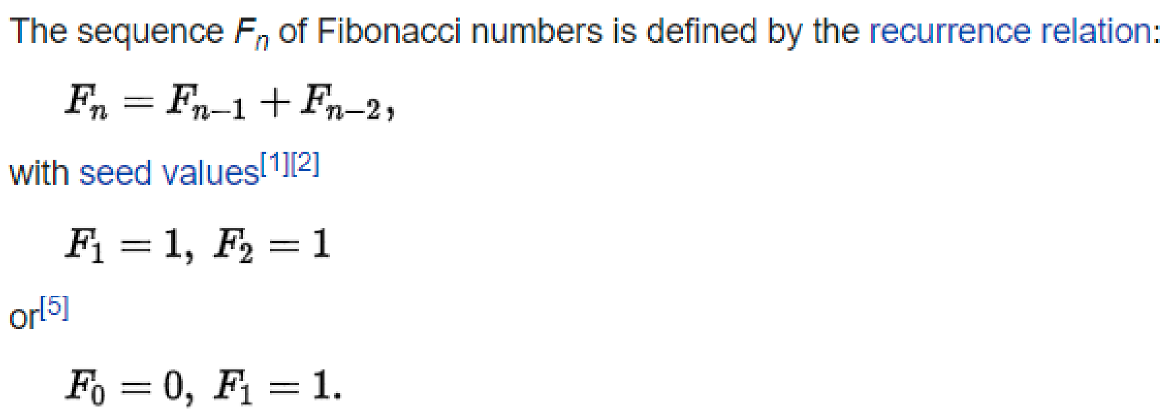
You'll find a good summary definition of key terms, here: https://github.com/domenic/promises-unwrapping/blob/master/docs/states-and-fates.md

[1] with thanks to Fatboy Slim

Coding

Let’s now move away from the purely conceptual and look at promises in JavaScript, what they consist of, how they’re used and how you create them. It’s conceivable that you’ll find yourself using APIs which utilise promises but never have to create one yourself, but in my experience, once you start working with promise based APIs, you’ll find this naturally leads to you creating promises elsewhere in your code.

To learn about promises, we’ll implement a Fibonacci number calculator using first a synchronous approach and then using a promise-based asynchronous approach. According to Wikipedia,



The sequence F_n of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$

with seed values[1][2]

$$F_1 = 1,\ F_2 = 1$$

or[5]

$$F_0 = 0,\ F_1 = 1.$$

Fibonacci Calculator - Synchronous

The synchronous approach is very simple, but risks blocking the main browser thread for a long time, depending on the value of n supplied to the function.

The main functions involved are these:

```
function calculate() {
  N = document.getElementById("fib_arg").value;
  if (nIsValid()) {
    start = Date.now();
    F();
  } else {
    log("ERROR: Value for N is invalid (" + N + "). N must be a positive
integer >= 2 and <= 45");
  }
  elapsed = Date.now() - start;
  log("Finished - " + elapsed + "ms");
}

function fibby(n) {
  if (n <= 2) {
    return 1;
  } else {
    return fibby(n - 1) + fibby(n - 2);
  }
}

function F() {
  n = parseInt(N);
  document.getElementById("result").innerHTML = fibby(n);
}
```

The calculate() function is called when a button is clicked. The variable N is extracted from the HTML form and validated. We then call function F() and calculate the Fibonacci number using a recursive function fibby(n) which runs in the same thread as the button click handler.

The full code as a JsFiddle is right here if you'd like to explore.

```
<iframe width="100%" height="300"
src="//jsfiddle.net/bluetooth_mdw/8yzdL082/embedded/"
allowfullscreen="allowfullscreen" frameborder="0"></iframe>
```

Promise use and syntax

There are two alternative syntaxes for using promises. Promises are objects and in either case, the promise object's then() function is passed two functions, one to execute when the promise has fulfilled (i.e. completed successfully) and the other for if the promise was rejected. In either case, parameters may be passed to the functions.

Note that functions which return promises are often said to be "thenable" and this property means promise execution can be chained together.

Here's the first syntactic variant:

```
function_which_returns_a_promise().then(
    function(result) {

        // code to execute when the promised was fulfilled

    },
    function(err) {

        // code to execute when the promised was rejected

    });
```

And here's the second:

```
function_which_returns_a_promise
.then(_ => {

       // code to execute when the promised was fulfilled

})
.catch(error => {

       // code to execute when the promised was rejected

});
```

The "=>" operand is known as an Arrow Function which as you can see, has a shorter syntax.

The "_" character in the second version means that no parameter is being passed to the function to be executed when the promise has fulfilled.

Fibonacci Calculator - Asynchronous using promises

Here’s an alternative implementation of our Fibonacci number calculator using promises:

```
function calculate() {
  N = document.getElementById("fib_arg").value;
  if (nIsValid()) {
    start = Date.now();
    F().then(function(result) {
      // fulfilled - Fibonacci number has been calculated
      document.getElementById("result").innerHTML = result;
      elapsed = Date.now() - start;
      log("Finished - " + elapsed + "ms");
    }, function(err) {
      // rejected - something went wrong
      console.log("Error: " + err);
    });
  } else {
    log("ERROR: Value for N is invalid (" + N + "). N must be a positive
integer >= 2 and <= 45");
  }
}

function fibby(n) {
  if (n <= 2) {
    return 1;
  } else {
    return fibby(n - 1) + fibby(n - 2);
  }
}

function F() {
  return new Promise(function(resolve, reject) {
    n = parseInt(N);
    result = fibby(n);
    resolve(result);
  });
```

The code for calculating the Fibonacci number itself is identical in each of the two patterns, synchronous and asynchronous.

Here’s the full solution in JsFiddle:

```
<iframe width="100%" height="300"
src="//jsfiddle.net/bluetooth_mdw/n2jgmjz1/embedded/"
allowfullscreen="allowfullscreen" frameborder="0"></iframe>
```

Creating Promises

Promise objects are constructed with one argument; a function with two parameters, the first of which is a function to call when we have a positive outcome and the other, a function to call when there's a negative outcome. The relevant function is called in the body of the promise. The general pattern looks like this:

```
var promise = new Promise(function(resolve, reject) {
  // do whatever this promise is supposed to do and then...

  // I'm using ok as an example boolean which indicates whether the
  // main code completed successfully (ok == true) or not (ok == false)

  if (ok) {
    resolve("Success");
  }
  else {
    reject(Error("Some kind of error occurred"));
  }
});
```

When calling the reject function, whilst it's not mandatory to pass an Error object, it is recommended.

Promises are often returned by functions and have their "then()" function directly invoked as was shown with the Fibonacci number calculator:

```

F().then(function(result) {
      // fulfilled - Fibonacci number has been calculated
      document.getElementById("result").innerHTML = result;
      elapsed = Date.now() - start;
      log("Finished - " + elapsed + "ms");
    }, function(err) {
      // rejected - something went wrong
      console.log("Error: " + err);
    });

function F() {
  return new Promise(function(resolve, reject) {
    n = parseInt(N);
    result = fibby(n);
    resolve(result);
 });
```

Close

This article should have given you a good idea of what you need to know before you can use the Web Bluetooth APIs. You probably already have at least some experience of web development (hence your interest in this series) and if there was anything new here, it's likely to have been promises. I recommend that if promises are new to you, you play around with them in JsFiddle or elsewhere and make sure you're comfortable with using and implementing them. They're key to being able to use Web Bluetooth.

In the next part, we'll dive right into Web Bluetooth and learn about device discovery, connecting to a Bluetooth LE device and how to perform GATT service discovery.

Call to Action

Web Bluetooth is not yet a W3C standard. I think it needs to be. Web Bluetooth is not yet implemented in all browsers either[2]. And it really needs to be in my humble opinion. Right now, you'll find Web Bluetooth in Chrome on most platforms. The "caniuse" URL in the footnote will give you full information.

If, you feel the same as I do, then I invite and encourage you to petition browser implementers to get behind Web Bluetooth and progress it through the W3C standards process. IoT needs this. You need this.

Mozilla Firefox	https://bugzilla.mozilla.org/show_bug.cgi?id=674737	
Microsoft Edge	https://developer.microsoft.com/en-us/microsoft-edge/platform/status/webbluetooth/ https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/9775308-implement-the-web-bluetooth-gatt-client-api	
WebKit (Safari)	https://bugs.webkit.org/show_bug.cgi?id=101034	

The W3C also host a public email list for Web Bluetooth: https://lists.w3.org/Archives/Public/public-web-bluetooth/

[2] See https://caniuse.com/#search=web%20bluetooth