

Interact with Bluetooth devices on the Web



By François Beaufort

Dives into Chromium source code

What if I told you websites could communicate with nearby Bluetooth devices in a secure and privacy-preserving way? This way, heart rate monitors, singing lightbulbs, turtles and flying grumpy cats could interact directly with a website.

Until now, the ability to interact with bluetooth devices has been possible only for native apps. The Web Bluetooth API aims to change this and brings it to web browsers as well. Alongside efforts like Physical Web, people can walk up to and interact with devices straight from the web. Check out this drone controlled from a web app video to get a sense of how that would work.

Before we start

This article assumes you have some basic knowledge of how Bluetooth Low Energy (BLE) and the Generic Attribute Profile (GATT) work.

Even though the Web Bluetooth API specification is not finalized yet, the Chrome Team is actively looking for enthusiastic developers (I mean you) to try out this work-in-progress API and give feedback on the spec and feedback on the implementation.

A subset of the Web Bluetooth API is available in Chrome 56 for Chrome OS, Chrome for Android M, and Mac. This means you should be able to request and connect to nearby Bluetooth devices, read/write Bluetooth characteristics, receive GATT Notifications, know when a Bluetooth device gets disconnected, and even read and write to Bluetooth descriptors.

On Linux, you still have to go to `chrome://flags/#enable-experimental-web-platform-features`, enable the highlighted flag, and restart Chrome for now.

Available for Origin Trials

In order to get as much feedback as possible from developers using the Web Bluetooth API in the field, we've previously added this feature in Chrome 53 as an origin trial for Chrome

OS, Android M, and Mac.

The trial has successfully ended in January 2017.

Security Requirements

To understand the security tradeoffs, I recommend the [Web Bluetooth Security Model](#) post from Jeffrey Yasskin, a software engineer on the Chrome team, working on the Web Bluetooth API specification.

HTTPS Only

Because this experimental API is a powerful new feature added to the Web, Google Chrome aims to make it available only to [secure contexts](#). This means you'll need to build with TLS in mind.

Note: We care deeply about security, so you will notice that new Web capabilities require HTTPS. The Web Bluetooth API is no different, and is yet another good reason to get HTTPS up and running on your site.

During development you'll be able to interact with Web Bluetooth through `http://localhost` by using tools like the [Chrome Dev Editor](#) or the handy `python -m SimpleHTTPServer`, but to deploy it on a site you'll need to have HTTPS set up on your server. I personally enjoy [GitHub Pages](#) for demo purposes.

To add HTTPS to your server you'll need to get a TLS certificate and set it up. Be sure to check out the [Security with HTTPS article](#) for best practices there. For info, you can now get free TLS certificates with the new Certificate Authority [Let's Encrypt](#) [↗](#).

User Gesture Required

As a security feature, discovering Bluetooth devices with `navigator.bluetooth.requestDevice` must be triggered by [a user gesture](#) such as a touch or a mouse click. We're talking about listening to [pointerup](#), `click`, and `touchend` events.

```
button.addEventListener('pointerup', function(event) {  
  // Call navigator.bluetooth.requestDevice  
});
```



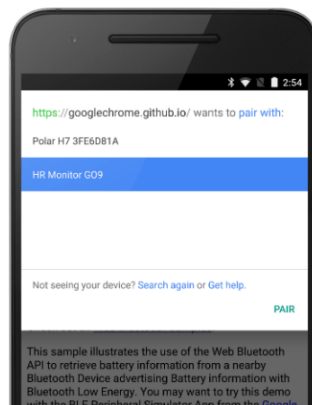
Get into the Code

The Web Bluetooth API relies heavily on JavaScript Promises. If you're not familiar with them, check out this great Promises tutorial. One more thing, `() => {}` are simply ECMAScript 2015 Arrow functions – they have a shorter syntax compared to function expressions and lexically bind the `this` value.

Request Bluetooth Devices

This version of the Web Bluetooth API specification allows websites, running in the Central role, to connect to remote GATT Servers over a BLE connection. It supports communication among devices that implement Bluetooth 4.0 or later.

When a website requests access to nearby devices using `navigator.bluetooth.requestDevice`, Google Chrome will prompt user with a device chooser where they can pick one device or simply cancel the request.



The `navigator.bluetooth.requestDevice` function takes a mandatory Object that defines filters. These filters are used to return only devices that match some advertised Bluetooth GATT services and/or the device name.

For instance, requesting Bluetooth devices advertising the Bluetooth GATT Battery Service is this simple:

```
navigator.bluetooth.requestDevice({ filters: [{ services: ['battery_service']  
.then(device => { /* ... */ })  
.catch(error => { console.log(error); });
```

If your Bluetooth GATT Service is not on the list of [the standardized Bluetooth GATT services](#) though, you may provide either the full Bluetooth UUID or a short 16- or 32-bit form.

```
navigator.bluetooth.requestDevice({  
  filters: [{  
    services: [0x1234, 0x12345678, '99999999-0000-1000-8000-00805f9b34fb']  
  }]  
})  
.then(device => { /* ... */ })  
.catch(error => { console.log(error); });
```

You can also request Bluetooth devices based on the device name being advertised with the `name` filters key, or even a prefix of this name with the `namePrefix` filters key. Note that in this case, you will also need to define the `optionalServices` key to be able to access some services. If you don't, you'll get an error later when trying to access them.

```
navigator.bluetooth.requestDevice({  
  filters: [{  
    name: 'Francois robot'  
  }],  
  optionalServices: ['battery_service']  
})  
.then(device => { /* ... */ })  
.catch(error => { console.log(error); });
```

Finally, instead of `filters` you can use the `acceptAllDevices` key to show all nearby Bluetooth devices. You will also need to define the `optionalServices` key to be able to access some services. If you don't, you'll get an error later when trying to access them.

```
navigator.bluetooth.requestDevice({  
  acceptAllDevices: true,  
  optionalServices: ['battery_service']  
})  
.then(device => { /* ... */ })  
.catch(error => { console.log(error); });
```

Caution: This may result in a bunch of unrelated devices being shown in the chooser and energy being wasted as there are no filters. Use it with caution.

Connect to a Bluetooth Device

So what do you do now that you have a **BluetoothDevice** returned from **navigator.bluetooth.requestDevice**'s Promise? Let's connect to the Bluetooth remote GATT Server which holds the service and characteristic definitions.

```
navigator.bluetooth.requestDevice({ filters: [{ services: ['battery_service'] }]}
  .then(device => {
    // Human-readable name of the device.
    console.log(device.name);

    // Attempts to connect to remote GATT Server.
    return device.gatt.connect();
  })
  .then(server => { /* ... */ })
  .catch(error => { console.log(error); });
```

Read a Bluetooth Characteristic

Here we are connected to the GATT Server of the remote Bluetooth device. Now we want to get a Primary GATT Service and read a characteristic that belongs to this service. Let's try, for instance, to read the current charge level of the device's battery.

In the example below, **battery_level** is the standardized Battery Level Characteristic.

```
navigator.bluetooth.requestDevice({ filters: [{ services: ['battery_service'] }]}
  .then(device => device.gatt.connect())
  .then(server => {
    // Getting Battery Service...
    return server.getPrimaryService('battery_service');
  })
  .then(service => {
    // Getting Battery Level Characteristic...
    return service.getCharacteristic('battery_level');
  })
  .then(characteristic => {
    // Reading Battery Level...
    return characteristic.readValue();
  })
  .then(value => {
    console.log('Battery percentage is ' + value.getUint8(0));
  })
  .catch(error => { console.log(error); });
```

If you use a custom Bluetooth GATT characteristic, you may provide either the full Bluetooth UUID or a short 16- or 32-bit form to `service.getCharacteristic`.

Note that you can also add a `characteristicvaluechanged` event listener on a characteristic to handle reading its value. Check out [Read Characteristic Value Changed Sample](#) to see how to optionally handle upcoming GATT notifications as well.



```
...
.then(characteristic => {
  // Set up event listener for when characteristic value changes.
  characteristic.addEventListener('characteristicvaluechanged',
    handleBatteryLevelChanged);

  // Reading Battery Level...
  return characteristic.readValue();
})
.catch(error => { console.log(error); });

function handleBatteryLevelChanged(event) {
  let batteryLevel = event.target.value.getUint8(0);
  console.log('Battery percentage is ' + batteryLevel);
}
```

Write to a Bluetooth Characteristic

Writing to a Bluetooth GATT Characteristic is as easy as reading it. This time, let's use the Heart Rate Control Point to reset the value of the Energy Expended field to 0 on a heart rate monitor device.

I promise there is no magic here. It's all explained in the [Heart Rate Control Point Characteristic page](#).





```
navigator.bluetooth.requestDevice({ filters: [{ services: ['heart_rate'] }] })
.then(device => device.gatt.connect())
.then(server => server.getPrimaryService('heart_rate'))
.then(service => service.getCharacteristic('heart_rate_control_point'))
.then(characteristic => {
  // Writing 1 is the signal to reset energy expended.
  var resetEnergyExpended = Uint8Array.of(1);
  return characteristic.writeValue(resetEnergyExpended);
})
.then(_ => {
  console.log('Energy expended has been reset.');
```

Receive GATT Notifications

Now, let's see how to be notified when the Heart Rate Measurement characteristic changes on the device:

```
navigator.bluetooth.requestDevice({ filters: [{ services: ['heart_rate'] }] })
  .then(device => device.gatt.connect())
  .then(server => server.getPrimaryService('heart_rate'))
  .then(service => service.getCharacteristic('heart_rate_measurement'))
  .then(characteristic => characteristic.startNotifications())
  .then(characteristic => {
    characteristic.addEventListener('characteristicvaluechanged',
                                   handleCharacteristicValueChanged);
    console.log('Notifications have been started.');
```



```
  })
  .catch(error => { console.log(error); });

function handleCharacteristicValueChanged(event) {
  var value = event.target.value;
  console.log('Received ' + value);
  // TODO: Parse Heart Rate Measurement value.
  // See https://github.com/WebBluetoothCG/demos/blob/gh-pages/heart-rate-sensor/
}
```

The Notifications Sample will show you to how to stop notifications with `stopNotifications()` and properly remove the added `characteristicvaluechanged` event listener.



Get disconnected from a Bluetooth Device

To provide a better user experience, you may want to show a warning message if the `BluetoothDevice` gets disconnected to invite the user to reconnect.

```
navigator.bluetooth.requestDevice({ filters: [{ name: 'Francois robot' }] })
  .then(device => {
    // Set up event listener for when device gets disconnected.
    device.addEventListener('gattserverdisconnected', onDisconnected);

    // Attempts to connect to remote GATT Server.
    return device.gatt.connect();
  })
  .then(server => { /* ... */ })
  .catch(error => { console.log(error); });

function onDisconnected(event) {
```



```

    let device = event.target;
    console.log('Device ' + device.name + ' is disconnected.');
```

You can also call `device.gatt.disconnect()` to disconnect your web app from the Bluetooth device. This will trigger existing `gattserverdisconnected` event listeners. Note that it will NOT stop bluetooth device communication if another app is already communicating with the Bluetooth device. Check out the [Device Disconnect Sample](#) and the [Automatic Reconnect Sample](#) to dive deeper.

Warning: Bluetooth GATT attributes, services, characteristics, etc. are invalidated when a device disconnects. This means your code should always retrieve (through `getPrimaryService(s)`, `getCharacteristic(s)`, etc.) these attributes after reconnecting.

Read and write to Bluetooth descriptors

Bluetooth GATT descriptors are attributes that describe a characteristic value. You can read and write them to in a similar way to Bluetooth GATT characteristics.

Let's see for instance how to read the user description of the measurement interval of the device's health thermometer.

In the example below, `health_thermometer` is the [Health Thermometer service](#), `measurement_interval` the [Measurement Interval characteristic](#), and `gatt.characteristic_user_description` the [Characteristic User Description descriptor](#).

```

navigator.bluetooth.requestDevice({ filters: [{ services: ['health_thermometer']
  .then(device => device.gatt.connect())
  .then(server => server.getPrimaryService('health_thermometer'))
  .then(service => service.getCharacteristic('measurement_interval'))
  .then(characteristic => characteristic.getDescriptor('gatt.characteristic_user_de
  .then(descriptor => descriptor.readValue())
  .then(value => {
    let decoder = new TextDecoder('utf-8');
    console.log('User Description: ' + decoder.decode(value));
  })
  .catch(error => { console.log(error); });
```

Now that we've read the user description of the measurement interval of the device's health thermometer, let's see how to update it and write a custom value.

```

navigator.bluetooth.requestDevice({ filters: [{ services: ['health_thermometer']
  .then(device => device.gatt.connect())
```



```

.then(server => server.getPrimaryService('health_thermometer'))
.then(service => service.getCharacteristic('measurement_interval'))
.then(characteristic => characteristic.getDescriptor('gatt.characteristic_user_de
.then(descriptor => {
  let encoder = new TextEncoder('utf-8');
  let userDescription = encoder.encode('Defines the time between measurements.');
```

Samples, Demos and Codelabs

All [Web Bluetooth samples](#) below have been successfully tested. To enjoy these samples to their fullest, I recommend you install the [BLE Peripheral Simulator Android App](#) which simulates a BLE peripheral with a Battery Service, a Heart Rate Service, or a Health Thermometer Service.

Beginner

- [Device Info](#) - retrieve basic device information from a BLE Device.
- [Battery Level](#) - retrieve battery information from a BLE Device advertising Battery information.
- [Reset Energy](#) - reset energy expended from a BLE Device advertising Heart Rate.
- [Characteristic Properties](#) - display all properties of a specific characteristic from a BLE Device.
- [Notifications](#) - start and stop characteristic notifications from a BLE Device.
- [Device Disconnect](#) - disconnect and get notified from a disconnection of a BLE Device after connecting to it.
- [Get Characteristics](#) - get all characteristics of an advertised service from a BLE Device.
- [Get Descriptors](#) - get all characteristics' descriptors of an advertised service from a BLE Device.

Combining multiple operations

- [GAP Characteristics](#) - get all GAP characteristics of a BLE Device.
- [Device Information Characteristics](#) - get all Device Information characteristics of a BLE Device.

- [Link Loss](#) - set the Alert Level characteristic of a BLE Device (readValue & writeValue).
- [Discover Services & Characteristics](#) - discover all accessible primary services and their characteristics from a BLE Device.
- [Automatic Reconnect](#) - reconnect to a disconnected BLE device using an exponential backoff algorithm.
- [Read Characteristic Value Changed](#) - read battery level and be notified of changes from a BLE Device.
- [Read Descriptors](#) - read all characteristic's descriptors of a service from a BLE Device.
- [Write Descriptor](#) - write to the descriptor "Characteristic User Description" on a BLE Device.

Check out our [curated Web Bluetooth Demos](#) and [official Web Bluetooth Codelabs](#) as well.

Libraries

- [web-bluetooth-utils](#) is a npm module that adds some convenience functions to the API.
- A Web Bluetooth API shim is available in [noble](#), the most popular Node.js BLE central module. This allows you to webpack/browserify noble without the need for a WebSocket server or other plugins.
- [angular-web-bluetooth](#) is a module for [Angular](#) that abstracts away all the boilerplate needed to configure the Web Bluetooth API.
- [<platinum-bluetooth>](#) is a set of [Polymer](#) elements to discover and communicate with nearby Bluetooth devices based on the Web Bluetooth API. For instance, here's how to read battery level from a nearby bluetooth device advertising a Battery service:

```
<platinum-bluetooth-device services-filter=['battery_service']>
  <platinum-bluetooth-service service='battery_service'>
    <platinum-bluetooth-characteristic characteristic='battery_level'>
    </platinum-bluetooth-characteristic>
  </platinum-bluetooth-service>
</platinum-bluetooth-device>
```



```
var bluetoothDevice = document.querySelector('platinum-bluetooth-device');
var batteryLevel = document.querySelector('platinum-bluetooth-characteristic');

bluetoothDevice.request()
  .then(_ => batteryLevel.read())
```



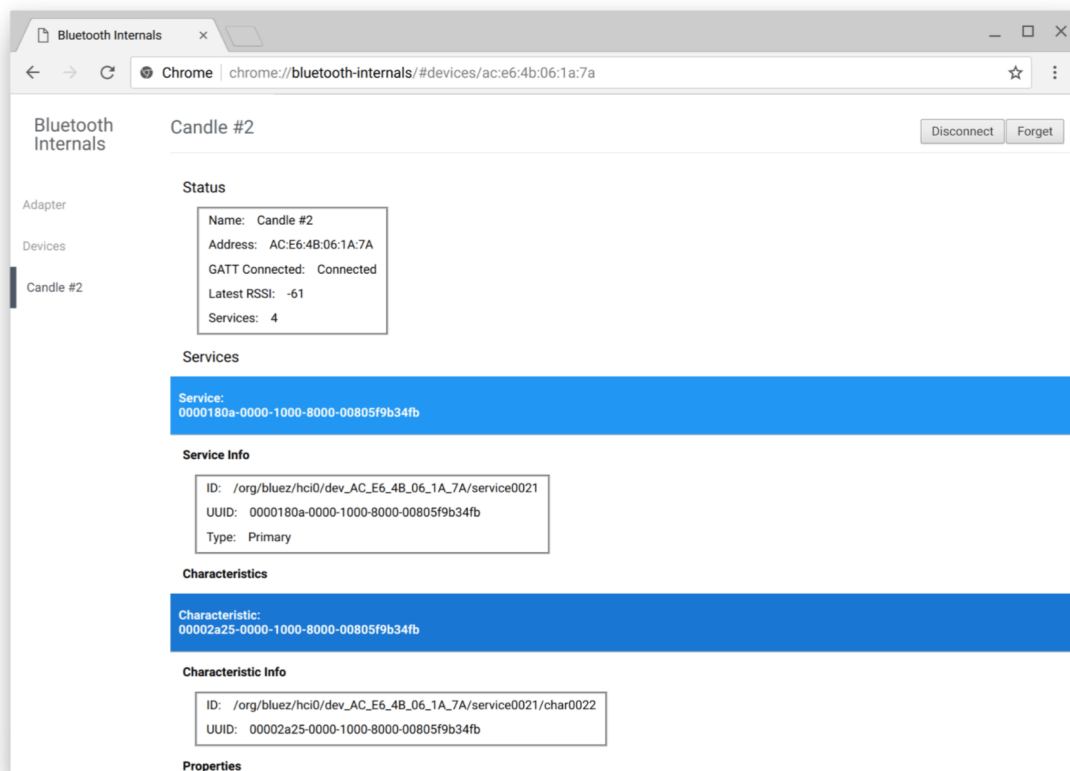
```
.then(value => {  
  console.log('Battery percentage is ' + value.getUint8(0));  
})  
.catch(error => { console.log(error); });
```

Tools

- [Get Started with Web Bluetooth](#) is a simple Web App that will generate all the JavaScript boilerplate code to start interacting with a Bluetooth device. Enter a device name, a service, a characteristic, define its properties and you're good to go.
- If you're already a Bluetooth developer, the [Web Bluetooth Developer Studio Plugin](#) will also generate the Web Bluetooth JavaScript code for your Bluetooth device.

Dev Tips

A "Bluetooth Internals" page is available in Chrome at `chrome://bluetooth-internals` so that you can inspect everything about nearby Bluetooth devices: status, services, characteristics, and descriptors.



I would also recommend you check out the official ["How to file Web Bluetooth bugs"](#) page as debugging Bluetooth can be hard sometimes.

What's next

Check the [browser and platform implementation status](#) first to know which parts of the Web Bluetooth API are currently being implemented.

Though it's still incomplete, here's a sneak peek of what to expect in the near future:

- [Scanning for nearby BLE advertisements](#) will happen with `navigator.bluetooth.requestLEScan()`.
- [Specifying the Eddystone upgrade](#) will allow a website opened from a Physical Web notification, to communicate with the device that advertised its URL.
- A new `serviceadded` event will track newly discovered Bluetooth GATT Services while `serviceremoved` event will track removed ones. A new `servicechanged` event will fire when any characteristic and/or descriptor gets added or removed from a Bluetooth GATT Service.

At the time of writing, Chrome OS, Android M, Linux, and Mac are the most advanced platforms. Windows 8.1+ and iOS will be supported as much as feasible by the platforms.

Resources

- Stack Overflow: <https://stackoverflow.com/questions/tagged/web-bluetooth>
- Web Bluetooth Community: <https://plus.google.com/communities/108953318610326025178>
- Chrome Feature Status: <https://www.chromestatus.com/feature/5264933985976320>
- Implementation Bugs: <https://crbug.com/?q=component:Blink>Bluetooth>
- Web Bluetooth Spec: <https://webbluetoothcg.github.io/web-bluetooth>
- Spec Issues: <https://github.com/WebBluetoothCG/web-bluetooth/issues>
- BLE Peripheral Simulator App: <https://github.com/WebBluetoothCG/ble-test-peripheral-android>

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.