Standard Standard Ecma-XXX

1st Edition / 1 June 2025

Sockets API

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Ecma International
Rue du Rhone 114 CH-1204 Geneva
Tel: +41 22 849 6000
Fax: +41 22 849 6001
Web: https://www.ecma-international.org

Table of Contents

page

Introduction

This document defines an API for establishing TCP connections in Non-Browser JavaScript runtime environments. Existing standard APIs are reused as much as possible, for example `ReadableStream` and `WritableStream` are used for reading and writing from a socket. Some options are inspired by the existing Node.js `net.Socket` API.

Contributing to this Specification

This version:
https://sockets-api.proposal.wintercg.org/

Issue Tracking:
GitHub

Editors:
Dominik Picheta (Cloudflare)

Ethan Arrowood

James M Snell (Cloudflare)

1. Concepts

1.1. Socket

A *socket* represents a TCP connection, from which you can read and write data. A socket begins in a *connected* state (if the socket fails to connect, an error is thrown). While in a *connected* state, the socket's `ReadableStream` and `WritableStream` can be read from and written to respectively.

A socket becomes *closed* when its `close()` method is called. A socket configured with `allowHalfOpen: false` will close itself when it receives a FIN or RST packet in its read stream.

1.2. Connect

The connect method here is defined in a `sockets` module only for initial implementation purposes. It is imagined that in a finalized standard definition, the connect would be exposed as a global or within a binding object

A socket can be constructed using a *connect* method defined in a `sockets` module (early implementations may use `vendor:sockets` for the module name), or defined on a binding object.

The connect method is the primary mechanism for creating a socket instance. It instantiates a socket with a resource identifier and some configuration values. It should synchronously return a socket instance in a *pending* state (or an error should be thrown). The socket will asynchronously *connect* depending on the implementation.

1.3. Binding Object

A binding object in this context is essentially just an object that exposes a connect method conformant with this specification. It is anticipated that a runtime may have any number of such objects. This is an area where there is still active discussion on how this should be defined.

The *binding object* defines extra socket `connect` options. The options it contains can modify the behaviour of the `connect` invoked on it. Some of the options it can define:

- TLS settings
- The HTTP proxy to use for the socket connection

The binding object is the primary mechanism for runtimes to introduce unique behavior for the connect method. For example, in order to support more TLS settings, a runtime may introduce a `TLSSocket` interface that extends from `Socket`. Thus, the binded `connect()` method could then utilize additional properties and configuration values that are controlled by the new `TLSSocket` interface.

```
const tls_socket = new TLSSocket({ key: '...', cert: '...' });
tls_socket.connect("example.com:1234");
```

Additionally, the binding object does not necessarily have to be an instance of a class, nor does it even have to be JavaScript. It can be any mechanism that exposes the **`connect()`** method. Cloudflare achieves this through environment bindings.

2. Socket

2.1. Using a socket

A basic example of using connect with an echo server.

```
const socket = connect({ hostname: "my-url.com", port: 43 });

const writer = socket.writable.getWriter();
await writer.write("Hello, World!\r\n");

const reader = socket.readable.getReader();
const result = await reader.read();

console.log(Buffer.from(result.value).toString()); // Hello, World!
```

2.2. The `Socket` class

The **`Socket`** class is an instance of the socket concept. It should not be instantiated directly (**`new Socket()`**), but instead created by calling **`connect()`**. A constructor for **`Socket`** is intentionally not specified, and is left to implementors to create.

```
[Exposed=*]
dictionary SocketInfo {
  DOMString remoteAddress = null;
  DOMString localAddress = null;
  DOMString alpn = null;
};

[Exposed=*]
interface Socket {
  readonly attribute ReadableStream readable;
  readonly attribute WritableStream writable;

  readonly attribute Promise<SocketInfo> opened;

  readonly attribute Promise<undefined> closed;
  Promise<undefined> close(optional any reason);

  [NewObject] Socket startTls();
};
```

The terms **`ReadableStream`** and **`WritableStream`** are defined in [WHATWG-STREAMS].

2.3. Attributes

2.3.1. readable

The `readable` attribute is a `ReadableStream` which receives data from the server the socket is connected to.

The below example shows typical `ReadableStream` usage to read data from a socket:

```
import { connect } from 'sockets';
const socket = connect("google.com:80");

const reader = socket.readable.getReader();

while (true) {
  const { value, done } = await reader.read();
  if (done) {
    // the ReadableStream has been closed or cancelled
    break;
  }
  // In many protocols the `value` needs to be decoded to be used:
  const decoder = new TextDecoder();
  console.log(decoder.decode(value));
}

reader.releaseLock();
```

The ReadableStream currently is defined to operate in non-byte mode, that is the **`type`** parameter to the ReadableStream constructor is not set. This means the stream's controller is `ReadableStreamDefaultController`. This, however, should be discussed and may be made configurable. It is reasonable, for instance, to assume that sockets used for most TCP cases would be byte-oriented, while sockets used for messages (e.g. UDP) would not.

2.3.2. writable

The `writable` attribute is a `WritableStream` which sends data to the server the socket is connected to.

The below example shows typical `WritableStream` usage to write data to a socket:

```
import { connect } from 'sockets';
const socket = connect("google.com:80");

const writer = socket.writable.getWriter();
const encoder = new TextEncoder();
writer.write(encoder.encode("GET / HTTP/1.0\r\n\r\n"));
```

2.3.3. opened

The `opened` attribute is a promise that is resolved when the socket connection has been successfully established, or is rejected if the connection fails. For sockets which use secure-transport, the resolution of the `opened` promise indicates the completion of the secure handshake.

The `opened` promise resolves a `SocketInfo` dictionary that optionally provides details about the connection that has been established.

By default, the `opened` promise is `marked as handled`.

2.3.4. closed

The `closed` attribute is a promise which can be used to keep track of the socket state. It gets resolved under the following circumstances:

- the `close()` method is called on the socket
- the socket was constructed with the `allowHalfOpen` parameter set to `false`, the ReadableStream is being read from, and the remote connection sends a FIN packet (graceful closure) or a RST packet

The current Cloudflare Workers implementation behaves as described above, specifically the ReadableStream needs to be read until completion for the `closed` promise to resolve, if the ReadableStream is not read then even if the server closes the connection the `closed` promise will not resolve.

Whether the promise should resolve without the ReadableStream being read is up for discussion.

It can also be rejected with a SocketError when a socket connection could not be established under the following circumstances:

- The address/port combo requested is blocked
- A transient issue with the runtime

Cancelling the socket's ReadableStream and closing the socket's WritableStream does not resolve the `closed` promise.

2.4. Methods

2.4.1. close(optional any reason)

The `close()` method closes the socket and its underlying connection. It returns the same promise as the `closed` attribute.

When called, the `ReadableStream` and `WritableStream` associated with the `Socket` will be canceled and aborted, respectively. If the `reason` argument is specified, the `reason` will be passed on to both the `ReadableStream` and `WritableStream`.

If the `opened` promise is still pending, it will be rejected with the `reason`.

2.4.2. startTls()

The `startTls()` method enables opportunistic TLS (otherwise known as StartTLS) which is a requirement for some protocols (primarily postgres/mysql and other DB protocols).

In this `secureTransport` mode of operation the socket begins the connection in plain-text, with messages read and written without any encryption. Then once the `startTls` method is called on the socket, the following shall take place:

- the original socket is closed, though the original connection is kept alive
- a secure TLS connection is established over that connection
- a new socket is created and returned from the `startTls` call

Here is a simple code example showing usage of the `startTls()` method:

```
import { connect } from 'sockets';
let sock = connect("google.com:443", { secureTransport: "starttls" });
// ... some code here ...
// We want to StartTLS at this point.
let tlsSock = sock.startTls();
```

The original readers and writers based off the original socket will no longer work. You must create new readers and writers from the new socket returned by **`startTls`**.

The method must fail with an SocketError if:

- called on an existing TLS socket
- the **`secureTransport`** option defined on the **`Socket`** instance is not equal to **`"starttls"`**.

2.5. SocketError

Arguably, this should be a type of **`DOMException`** rather than **`TypeError`**. More discussion is necessary on the form and structure of socket-related errors.

SocketError is an instance of **`TypeError`**. The error message should start with **`"SocketError: "`**.

An **`"connection failed"`** SocketError.

```
throw new SocketError('connection failed');
```

Should result in the following error: **`Uncaught SocketError [TypeError]: SocketError: connection failed`**.

3. connect

```
[Exposed=*]
dictionary SocketAddress {
  DOMString hostname;
  unsigned short port;
};

typedef (DOMString or SocketAddress) AnySocketAddress;

enum SecureTransportKind { "off", "on", "starttls" };

[Exposed=*]
dictionary SocketOptions {
  SecureTransportKind secureTransport = "off";
  boolean allowHalfOpen = false;
  DOMString sni = null;
  DOMString[] alpn = [];
};

[Exposed=*]
interface Connect {
  Socket connect(AnySocketAddress address, optional SocketOptions opts);
};
```

The `connect()` method performs the following steps:

1. New `Socket` instance is created with each of its attributes initialised immediately.
2. The socket's `opened` promise is set to a new promise. Set *opened*.[[PromiseIsHandled]] to true.
3. The socket's `closed` promise is set to a new promise. Set *closed*.[[PromiseIsHandled]] to true.
4. The created `Socket` instance is returned immediately in a *pending* state.
5. A connection is established to the specified `SocketAddress` asynchronously.
6. Once the connection is established, set *info* to a new `SocketInfo`, and Resolve *opened* with *info*. For a socket using secure transport, the connection is considered to be established once the secure handshake has been completed.
7. If the connection fails for any reason, set *error* to a new SocketError and reject the socket's `closed` and `opened` promises with *error*. Also, the `readable` is canceled with *error* and the `writable` is aborted with *error*.
8. The instance's `ReadableStream` and `WritableStream` streams can be used immediately but may not actually transmit or receive data until the socket is fully opened.

At any point during the creation of the `Socket` instance, **`connect`** may throw a SocketError. One case where this can happen is if the input address is incorrectly formatted.

The implementation may consider blocking connections to certain hostname/port combinations which can pose a threat of abuse or security vulnerability.

For example, port 25 may be blocked to prevent abuse of SMTP servers and private IPs can be blocked to avoid connecting to private services hosted locally (or on the server's LAN).

3.1. `SocketOptions` dictionary

`secureTransport` **member**
: The secure transport mode to use.

  `off`
  : A connection is established in plain text.

  `on`
  : A TLS connection is established using default CAs

  `starttls`
  : Initially the same as the **`off`** option, the connection continues in plain text until the `startTls()` method is called

`alpn` **member**
: The Application-Layer Protocol Negotiation list to send, as an array of strings. If the server agrees with one of the protocols specified in this list, it will return the matching protocol in the `info` property. May be specified if and only if `secureTransport` is **`on`** or **`starttls`**.

`sni` **member**
: The Server Name Indication TLS option to send as part of the TLS handshake. If specified, requests that the server send a certificate with a matching common name. May be specified if and only if `secureTransport` is **`on`** or **`starttls`**.

`allowHalfOpen` **member**
: This option is similar to that offered by the Node.js **`net`** module and allows interoperability with code which utilizes it.

  false
  : The WritableStream- and the socket instance- will be automatically closed when a FIN packet is received from the remote connection.

  true
  : When a FIN packet is received, the socket will enter a "half-open" state where the ReadableStream is closed but the WritableStream can still be written to.

3.2. `SocketInfo` dictionary

`remoteAddress` member
: Provides the hostname/port combo of the remote peer the `Socket` is connected to, for example `"example.com:443"`. This value may or may not be the same as the address provided to the `connect()` method used to create the `Socket`.

`localAddress` member
: Optionally provides the hostname/port combo of the local network endpoint, for example `"localhost:12345"`.

`alpn` property
: If the server agrees with one of the protocols specified in the `alpn` negotiation list, returns that protocol name as a string, otherwise `null`.

3.3. `AnySocketAddress` type

`SocketAddress` dictionary
: The address to connect to. For example `{ hostname: "google.com", port: 443 }`.

`hostname`
: A connection is established in plain text.

`port`
: A TLS connection is established using default CAs

`DOMString`
: A hostname/port combo separated by a colon. For example `"google.com:443"`.

References

Normative References

[WEBGPU]
Kai Ninomiya; Brandon Jones; Jim Blandy. *WebGPU*. URL: https://gpuweb.github.io/gpuweb/

[WEBIDL]
Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

[WHATWG-STREAMS]
Adam Rice; et al. *Streams Standard*. Living Standard. URL: https://streams.spec.whatwg.org/

Index

Terms defined by this specification

- Connect, in § 3
- connect, in § 1.2
- connect(address), in § 3
- connect(address, opts), in § 3
- hostname, in § 3
- localAddress, in § 2.2
- "off", in § 3
- "on", in § 3
- opened, in § 2.2
- port, in § 3
- readable, in § 2.2
- remoteAddress, in § 2.2
- secureTransport, in § 3
- SecureTransportKind, in § 3
- sni, in § 3
- Socket, in § 2.2
- socket, in § 1.1
- SocketAddress, in § 3
- SocketError, in § 2.5
- SocketInfo, in § 2.2
- SocketOptions, in § 3
- "starttls", in § 3
- startTls(), in § 2.2
- writable, in § 2.2

Terms defined by reference

- [WEBGPU] defines the following terms:
  - info
- [WEBIDL] defines the following terms:
  - DOMException
  - DOMString
  - NewObject
  - Promise
  - TypeError
  - a new promise
  - any
  - boolean
  - resolve
  - undefined
  - unsigned short
- [WHATWG-STREAMS] defines the following terms:
  - ReadableStream
  - ReadableStreamDefaultController
  - WritableStream

IDL Index

```
[Exposed=*]
dictionary SocketInfo {
  DOMString remoteAddress = null;
  DOMString localAddress = null;
  DOMString alpn = null;
};

[Exposed=*]
interface Socket {
  readonly attribute ReadableStream readable;
  readonly attribute WritableStream writable;

  readonly attribute Promise<SocketInfo> opened;

  readonly attribute Promise<undefined> closed;
  Promise<undefined> close(optional any reason);

  [NewObject] Socket startTls();
};

[Exposed=*]
dictionary SocketAddress {
  DOMString hostname;
  unsigned short port;
};

typedef (DOMString or SocketAddress) AnySocketAddress;

enum SecureTransportKind { "off", "on", "starttls" };

[Exposed=*]
dictionary SocketOptions {
  SecureTransportKind secureTransport = "off";
  boolean allowHalfOpen = false;
  DOMString sni = null;
  DOMString[] alpn = [];
};

[Exposed=*]
interface Connect {
  Socket connect(AnySocketAddress address, optional SocketOptions opts);
};
```

Copyright & Software License

Fax: +41 22 849 6001

Web: https://ecma-international.org/

Copyright Notice

Software License