Web Streams API

> Stability: 1 - Experimental.

An implementation of the WHATWG Streams Standard.

Overview

The WHATWG Streams Standard (or "web streams") defines an API for handling streaming data. It is similar to the Node.js Streams API but emerged later and has become the "standard" API for streaming data across many JavaScript environments.

There are three primary types of objects

- `ReadableStream` - Represents a source of streaming data.
- `WritableStream` - Represents a destination for streaming data.
- `TransformStream` - Represents an algorithm for transforming streaming data.

Example `ReadableStream`

This example creates a simple `ReadableStream` that pushes the current `performance.now()` timestamp once every second forever. An async iterable is used to read the data from the stream.

```
import {
  ReadableStream
} from 'node:stream/web';

import {
  setInterval as every
} from 'node:timers/promises';

import {
  performance
} from 'node:perf_hooks';

const SECOND = 1000;

const stream = new ReadableStream({
  async start(controller) {
    for await (const _ of every(SECOND))
      controller.enqueue(performance.now());
  }
});
```

```
for await (const value of stream)
  console.log(value);
```

```
const {
  ReadableStream
} = require('stream/web');

const {
  setInterval: every
} = require('timers/promises');

const {
  performance
} = require('perf_hooks');

const SECOND = 1000;

const stream = new ReadableStream({
  async start(controller) {
    for await (const _ of every(SECOND))
      controller.enqueue(performance.now());
  }
});

(async () => {
  for await (const value of stream)
    console.log(value);
})();
```

API

Class: `ReadableStream`

`new ReadableStream([underlyingSource [, strategy]])`

- `underlyingSource` {Object}
  - `start` {Function} A user-defined function that is invoked immediately when the `ReadableStream` is created.
    - `controller` {ReadableStreamDefaultController|ReadableByteStreamController}
    - Returns: `undefined` or a promise fulfilled with `undefined`.
  - `pull` {Function} A user-defined function that is called repeatedly when the `ReadableStream` internal queue is not full. The operation may be sync or async. If async, the function will not be called again until the previously returned promise is fulfilled.
    - `controller` {ReadableStreamDefaultController|ReadableByteStreamController}
    - Returns: A promise fulfilled with `undefined`.
  - `cancel` {Function} A user-defined function that is called when the

      `ReadableStream` is canceled.
        * `reason` {any}
        * Returns: A promise fulfilled with `undefined`.
    - `type` {string} Must be `'bytes'` or `undefined`.
    - `autoAllocateChunkSize` {number} Used only when `type` is equal to `'bytes'`.
- `strategy` {Object}
    - `highWaterMark` {number} The maximum internal queue size before backpressure is applied.
    - `size` {Function} A user-defined function used to identify the size of each chunk of data.
        * `chunk` {any}
        * Returns: {number}

`readableStream.locked`

- Type: {boolean} Set to `true` if there is an active reader for this {ReadableStream}.

The `readableStream.locked` property is `false` by default, and is switched to `true` while there is an active reader consuming the stream's data.

`readableStream.cancel([reason])`

- `reason` {any}
- Returns: A promise fulfilled with `undefined` once cancelation has been completed.

`readableStream.getReader([options])`

- `options` {Object}
    - `mode` {string} `'byob'` or `undefined`
- Returns: {ReadableStreamDefaultReader|ReadableStreamBYOBReader}

```
import { ReadableStream } from 'node:stream/web';

const stream = new ReadableStream();

const reader = stream.getReader();

console.log(await reader.read());
```

```
const { ReadableStream } = require('stream/web');

const stream = new ReadableStream();

const reader = stream.getReader();
```

```
reader.read().then(console.log);
```

Causes the `readableStream.locked` to be `true`.

`readableStream.pipeThrough(transform[, options])`

- `transform` {Object}
  - `readable` {ReadableStream} The `ReadableStream` to which `transform.writable` will push the potentially modified data is receives from this `ReadableStream`.
  - `writable` {WritableStream} The `WritableStream` to which this `ReadableStream`'s data will be written.
- `options` {Object}
  - `preventAbort` {boolean} When `true`, errors in this `ReadableStream` will not cause `transform.writable` to be aborted.
  - `preventCancel` {boolean} When `true`, errors in the destination `transform.writable` is not cause this `ReadableStream` to be canceled.
  - `preventClose` {boolean} When `true`, closing this `ReadableStream` will no cause `transform.writable` to be closed.
  - `signal` {AbortSignal} Allows the transfer of data to be canceled using an {AbortController}.
- Returns: {ReadableStream} From `transform.readable`.

Connects this {ReadableStream} to the pair of {ReadableStream} and {WritableStream} provided in the `transform` argument such that the data from this {ReadableStream} is written in to `transform.writable`, possibly transformed, then pushed to `transform.readable`. Once the pipeline is configured, `transform.readable` is returned.

Causes the `readableStream.locked` to be `true` while the pipe operation is active.

```
import {
  ReadableStream,
  TransformStream,
} from 'node:stream/web';

const stream = new ReadableStream({
  start(controller) {
    controller.enqueue('a');
  },
});

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

const transformedStream = stream.pipeThrough(transform);

for await (const chunk of transformedStream)
  console.log(chunk);
```

```
const {
  ReadableStream,
  TransformStream,
} = require('stream/web');

const stream = new ReadableStream({
  start(controller) {
    controller.enqueue('a');
  },
});

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

const transformedStream = stream.pipeThrough(transform);

(async () => {
  for await (const chunk of transformedStream)
    console.log(chunk);
})();
```

`readableStream.pipeTo(destination, options)`

- `destination` {WritableStream} A {WritableStream} to which this `ReadableStream`'s data will be written.
- `options` {Object}
  - `preventAbort` {boolean} When `true`, errors in this `ReadableStream` will not cause `transform.writable` to be aborted.
  - `preventCancel` {boolean} When `true`, errors in the destination `transform.writable` is not cause this `ReadableStream` to be canceled.
  - `preventClose` {boolean} When `true`, closing this `ReadableStream` will no cause `transform.writable` to be closed.
  - `signal` {AbortSignal} Allows the transfer of data to be canceled using an {AbortController}.

- Returns: A promise fulfilled with `undefined`

Causes the `readableStream.locked` to be `true` while the pipe operation is active.

`readableStream.tee()`

- Returns: {ReadableStream[]}

Returns a pair of new {ReadableStream} instances to which this `ReadableStream`'s data will be forwarded. Each will receive the same data.

Causes the `readableStream.locked` to be `true`.

`readableStream.values([options])`

- `options` {Object}
  - `preventCancel` {boolean} When `true`, prevents the {ReadableStream} from being closed when the async iterator abruptly terminates. **Defaults**: `false`

Creates and returns an async iterator usable for consuming this `ReadableStream`'s data.

Causes the `readableStream.locked` to be `true` while the async iterator is active.

```
import { Buffer } from 'node:buffer';

const stream = new ReadableStream(getSomeSource());

for await (const chunk of stream.values({ preventCancel: true }))
  console.log(Buffer.from(chunk).toString());
```

Async Iteration The {ReadableStream} object supports the async iterator protocol using `for await` syntax.

```
import { Buffer } from 'buffer';

const stream = new ReadableStream(getSomeSource());

for await (const chunk of stream)
  console.log(Buffer.from(chunk).toString());
```

The async iterator will consume the {ReadableStream} until it terminates.

By default, if the async iterator exits early (via either a `break`, `return`, or a `throw`), the {ReadableStream} will be closed. To prevent automatic closing of the {ReadableStream}, use the `readableStream.values()` method to acquire the async iterator and set the `preventCancel` option to `true`.

The {ReadableStream} must not be locked (that is, it must not have an existing active reader). During the async iteration, the {ReadableStream} will be locked.

Transferring with `postMessage()` A {ReadableStream} instance can be transferred using a {MessagePort}.

```
const stream = new ReadableStream(getReadableSourceSomehow());

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => {
  data.getReader().read().then((chunk) => {
    console.log(chunk);
  });
};

port2.postMessage(stream, [stream]);
```

Class: `ReadableStreamDefaultReader`

By default, calling `readableStream.getReader()` with no arguments will return an instance of `ReadableStreamDefaultReader`. The default reader treats the chunks of data passed through the stream as opaque values, which allows the {ReadableStream} to work with generally any JavaScript value.

`new ReadableStreamDefaultReader(stream)`

- `stream` {ReadableStream}

Creates a new {ReadableStreamDefaultReader} that is locked to the given {ReadableStream}.

`readableStreamDefaultReader.cancel([reason])`

- `reason` {any}
- Returns: A promise fulfilled with `undefined`.

Cancels the {ReadableStream} and returns a promise that is fulfilled when the underlying stream has been canceled.

`readableStreamDefaultReader.closed`

- Type: {Promise} Fulfilled with `undefined` when the associated {ReadableStream} is closed or rejected if the stream errors or the reader's lock is released before the stream finishes closing.

`readableStreamDefaultReader.read()`

- Returns: A promise fulfilled with an object:
  - `value` {ArrayBuffer}
  - `done` {boolean}

Requests the next chunk of data from the underlying {ReadableStream} and returns a promise that is fulfilled with the data once it is available.

`readableStreamDefaultReader.releaseLock()` Releases this reader's lock on the underlying {ReadableStream}.

Class: `ReadableStreamBYOBReader`

The `ReadableStreamBYOBReader` is an alternative consumer for byte-oriented {ReadableStream}'s (those that are created with `underlyingSource.type` set equal to `'bytes'` when the `ReadableStream` was created).

The `BYOB` is short for "bring your own buffer". This is a pattern that allows for more efficient reading of byte-oriented data that avoids extraneous copying.

```
import {
  open
} from 'node:fs/promises';

import {
  ReadableStream
} from 'node:stream/web';

import { Buffer } from 'node:buffer';

class Source {
  type = 'bytes';
  autoAllocateChunkSize = 1024;

  async start(controller) {
    this.file = await open(new URL(import.meta.url));
    this.controller = controller;
  }

  async pull(controller) {
    const view = controller.byobRequest?.view;
    const {
      bytesRead,
    } = await this.file.read({
      buffer: view,
      offset: view.byteOffset,
      length: view.byteLength
    });

    if (bytesRead === 0) {
      await this.file.close();
      this.controller.close();
    }
    controller.byobRequest.respond(bytesRead);
  }
}

const stream = new ReadableStream(new Source());

async function read(stream) {
  const reader = stream.getReader({ mode: 'byob' });

  const chunks = [];
  let result;
  do {
    result = await reader.read(Buffer.alloc(100));
    if (result.value !== undefined)
      chunks.push(Buffer.from(result.value));
  } while (!result.done);

  return Buffer.concat(chunks);
}

const data = await read(stream);
console.log(Buffer.from(data).toString());
```

`new ReadableStreamBYOBReader(stream)`

- `stream` {ReadableStream}

Creates a new `ReadableStreamBYOBReader` that is locked to the given {ReadableStream}.

`readableStreamBYOBReader.cancel([reason])`

- `reason` {any}
- Returns: A promise fulfilled with `undefined`.

Cancels the {ReadableStream} and returns a promise that is fulfilled when the underlying stream has been canceled.

`readableStreamBYOBReader.closed`

- Type: {Promise} Fulfilled with `undefined` when the associated {ReadableStream} is closed or rejected if the stream errors or the reader's lock is released before the stream finishes closing.

`readableStreamBYOBReader.read(view)`

- `view` {Buffer|TypedArray|DataView}
- Returns: A promise fulfilled with an object:
    - `value` {ArrayBuffer}
    - `done` {boolean}

Requests the next chunk of data from the underlying {ReadableStream} and returns a promise that is fulfilled with the data once it is available.

Do not pass a pooled {Buffer} object instance in to this method. Pooled `Buffer` objects are created using `Buffer.allocUnsafe()`, or `Buffer.from()`, or are often returned by various `fs` module callbacks. These types of `Buffer`s use a shared underlying {ArrayBuffer} object that contains all of the data from all of the pooled `Buffer` instances. When a `Buffer`, {TypedArray}, or {DataView} is passed in to `readableStreamBYOBReader.read()`, the view's underlying `ArrayBuffer` is *detached*, invalidating all existing views that may exist on that `ArrayBuffer`. This can have disastrous consequences for your application.

`readableStreamBYOBReader.releaseLock()` Releases this reader's lock on the underlying {ReadableStream}.

Class: `ReadableStreamDefaultController`

Every {ReadableStream} has a controller that is responsible for the internal state and management of the stream's queue. The `ReadableStreamDefaultController` is the default controller implementation for `ReadableStream`s that are not byte-oriented.

`readableStreamDefaultController.close()` Closes the {ReadableStream} to which this controller is associated.

`readableStreamDefaultController.desiredSize`

- Type: {number}

Returns the amount of data remaining to fill the {ReadableStream}'s queue.

`readableStreamDefaultController.enqueue(chunk)`

- `chunk` {any}

Appends a new chunk of data to the {ReadableStream}'s queue.

`readableStreamDefaultController.error(error)`

- `error` {any}

Signals an error that causes the {ReadableStream} to error and close.

Class: `ReadableByteStreamController`

Every {ReadableStream} has a controller that is responsible for the internal state and management of the stream's queue. The `ReadableByteStreamController` is for byte-oriented `ReadableStreams`.

`readableByteStreamController.byobRequest`

- Type: {ReadableStreamBYOBRequest}

`readableByteStreamController.close()` Closes the {ReadableStream} to which this controller is associated.

`readableByteStreamController.desiredSize`

- Type: {number}

Returns the amount of data remaining to fill the {ReadableStream}'s queue.

`readableByteStreamController.enqueue(chunk)`

- `chunk`: {Buffer|TypedArray|DataView}

Appends a new chunk of data to the {ReadableStream}'s queue.

`readableByteStreamController.error(error)`

- `error` {any}

Signals an error that causes the {ReadableStream} to error and close.

Class: `ReadableStreamBYOBRequest`

When using `ReadableByteStreamController` in byte-oriented streams, and when using the `ReadableStreamBYOBReader`, the `readableByteStreamController.byobRequest` property provides access to a `ReadableStreamBYOBRequest` instance that represents the current read request. The object is used to gain access to the `ArrayBuffer`/`TypedArray` that has been provided for the read request to fill, and provides methods for signaling that the data has been provided.

`readableStreamBYOBRequest.respond(bytesWritten)`

- `bytesWritten` {number}

Signals that a `bytesWritten` number of bytes have been written to `readableStreamBYOBRequest.view`.

`readableStreamBYOBRequest.respondWithNewView(view)`

- `view` {Buffer|TypedArray|DataView}

Signals that the request has been fulfilled with bytes written to a new `Buffer`, `TypedArray`, or `DataView`.

`readableStreamBYOBRequest.view`

- Type: {Buffer|TypedArray|DataView}

Class: `WritableStream`

The `WritableStream` is a destination to which stream data is sent.

```
import {
  WritableStream
} from 'node:stream/web';

const stream = new WritableStream({
  write(chunk) {
    console.log(chunk);
  }
});

await stream.getWriter().write('Hello World');
```

`new WritableStream([underlyingSink[, strategy]])`

- `underlyingSink` {Object}
  - `start` {Function} A user-defined function that is invoked immediately when the `WritableStream` is created.
    - `controller` {WritableStreamDefaultController}
    - Returns: `undefined` or a promise fulfilled with `undefined`.
  - `write` {Function} A user-defined function that is invoked when a chunk of data has been written to the `WritableStream`.
    - `chunk` {any}
    - `controller` {WritableStreamDefaultController}
    - Returns: A promise fulfilled with `undefined`.
  - `close` {Function} A user-defined function that is called when the `WritableStream` is closed.
    - Returns: A promise fulfilled with `undefined`.

    - `abort` {Function} A user-defined function that is called to abruptly close the `WritableStream`.
        * `reason` {any}
        * Returns: A promise fulfilled with `undefined`.
    - `type` {any} The `type` option is reserved for future use and *must* be undefined.
- `strategy` {Object}
    - `highWaterMark` {number} The maximum internal queue size before backpressure is applied.
    - `size` {Function} A user-defined function used to identify the size of each chunk of data.
        * `chunk` {any}
        * Returns: {number}

`writableStream.abort([reason])`

- `reason` {any}
- Returns: A promise fulfilled with `undefined`.

Abruptly terminates the `WritableStream`. All queued writes will be canceled with their associated promises rejected.

`writableStream.close()`

- Returns: A promise fulfilled with `undefined`.

Closes the `WritableStream` when no additional writes are expected.

`writableStream.getWriter()`

- Returns: {WritableStreamDefaultWriter}

Creates and creates a new writer instance that can be used to write data into the `WritableStream`.

`writableStream.locked`

- Type: {boolean}

The `writableStream.locked` property is `false` by default, and is switched to `true` while there is an active writer attached to this `WritableStream`.

Transferring with postMessage() A {WritableStream} instance can be transferred using a {MessagePort}.

```
const stream = new WritableStream(getWritableSinkSomehow());

const { port1, port2 } = new MessageChannel();
```

```
port1.onmessage = ({ data }) => {
  data.getWriter().write('hello');
};

port2.postMessage(stream, [stream]);
```

Class: `WritableStreamDefaultWriter`

`new WritableStreamDefaultWriter(stream)`

- `stream` {WritableStream}

Creates a new `WritableStreamDefaultWriter` that is locked to the given `WritableStream`.

`writableStreamDefaultWriter.abort([reason])`

- `reason` {any}
- Returns: A promise fulfilled with `undefined`.

Abruptly terminates the `WritableStream`. All queued writes will be canceled with their associated promises rejected.

`writableStreamDefaultWriter.close()`

- Returns: A promise fulfilled with `undefined`.

Closes the `WritableStream` when no additional writes are expected.

`writableStreamDefaultWriter.closed`

- Type: {Promise} Fulfilled with `undefined` when the associated {WritableStream} is closed or rejected if the stream errors or the writer's lock is released before the stream finishes closing.

`writableStreamDefaultWriter.desiredSize`

- Type: {number}

The amount of data required to fill the {WritableStream}'s queue.

`writableStreamDefaultWriter.ready`

- type: A promise that is fulfilled with `undefined` when the writer is ready to be used.

`writableStreamDefaultWriter.releaseLock()` Releases this writer's lock on the underlying {ReadableStream}.

`writableStreamDefaultWriter.write([chunk])`

- `chunk`: {any}
- Returns: A promise fulfilled with `undefined`.

Appends a new chunk of data to the {WritableStream}'s queue.

Class: `WritableStreamDefaultController`

The `WritableStreamDefaultController` manage's the {WritableStream}'s internal state.

`writableStreamDefaultController.abortReason`

- Type: {any} The `reason` value passed to `writableStream.abort()`.

`writableStreamDefaultController.error(error)`

- `error` {any}

Called by user-code to signal that an error has occurred while processing the `WritableStream` data. When called, the {WritableStream} will be aborted, with currently pending writes canceled.

`writableStreamDefaultController.signal`

- Type: {AbortSignal} An `AbortSignal` that can be used to cancel pending write or close operations when a {WritableStream} is aborted.

Class: `TransformStream`

A `TransformStream` consists of a {ReadableStream} and a {WritableStream} that are connected such that the data written to the `WritableStream` is received, and potentially transformed, before being pushed into the `ReadableStream`'s queue.

```
import {
  TransformStream
} from 'node:stream/web';

const transform = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk.toUpperCase());
  }
});

await Promise.all([
  transform.writable.getWriter().write('A'),
  transform.readable.getReader().read(),
]);
```

`new TransformStream([transformer[, writableStrategy[, readableStrategy]]])`

- `transformer` {Object}
  - `start` {Function} A user-defined function that is invoked immediately when the `TransformStream` is created.
    - `controller` {TransformStreamDefaultController}
    - Returns: `undefined` or a promise fulfilled with `undefined`
  - `transform` {Function} A user-defined function that receives, and potentially modifies, a chunk of data written to `transformStream.writable`, before forwarding that on to `transformStream.readable`.
    - `chunk` {any}
    - `controller` {TransformStreamDefaultController}
    - Returns: A promise fulfilled with `undefined`.
  - `flush` {Function} A user-defined function that is called immediately before the writable side of the `TransformStream` is closed, signaling the end of the transformation process.
    - `controller` {TransformStreamDefaultController}
    - Returns: A promise fulfilled with `undefined`.
  - `readableType` {any} the `readableType` option is reserved for future use and *must* be `undefined`.
  - `writableType` {any} the `writableType` option is reserved for future use and *must* be `undefined`.
- `writableStrategy` {Object}
  - `highWaterMark` {number} The maximum internal queue size before backpressure is applied.
  - `size` {Function} A user-defined function used to identify the size of each chunk of data.
    - `chunk` {any}
    - Returns: {number}
- `readableStrategy` {Object}
  - `highWaterMark` {number} The maximum internal queue size before backpressure is applied.
  - `size` {Function} A user-defined function used to identify the size of each chunk of data.
    - `chunk` {any}
    - Returns: {number}

`transformStream.readable`

- Type: {ReadableStream}

`transformStream.writable`

- Type: {WritableStream}

Transferring with postMessage() A {TransformStream} instance can be transferred using a {MessagePort}.

```
const stream = new TransformStream();

const { port1, port2 } = new MessageChannel();

port1.onmessage = ({ data }) => {
  const { writable, readable } = data;
  // ...
};

port2.postMessage(stream, [stream]);
```

Class: `TransformStreamDefaultController`

The `TransformStreamDefaultController` manages the internal state of the `TransformStream`.

`transformStreamDefaultController.desiredSize`

- Type: {number}

The amount of data required to fill the readable side's queue.

`transformStreamDefaultController.enqueue([chunk])`

- `chunk` {any}

Appends a chunk of data to the readable side's queue.

`transformStreamDefaultController.error([reason])`

- `reason` {any}

Signals to both the readable and writable side that an error has occurred while processing the transform data, causing both sides to be abruptly closed.

`transformStreamDefaultController.terminate()` Closes the readable side of the transport and causes the writable side to be abruptly closed with an error.

Class: `ByteLengthQueuingStrategy`

`new ByteLengthQueuingStrategy(options)`

- `options` {Object}
  - `highWaterMark` {number}

`byteLengthQueuingStrategy.highWaterMark`

- Type: {number}

`byteLengthQueuingStrategy.size`

- Type: {Function}
  - `chunk` {any}
  - Returns: {number}

Class: `CountQueuingStrategy`

`new CountQueuingStrategy(options)`

- `options` {Object}
  - `highWaterMark` {number}

`countQueuingStrategy.highWaterMark`

- Type: {number}

`countQueuingStrategy.size`

- Type: {Function}
  - `chunk` {any}
  - Returns: {number}

Class: `TextEncoderStream`

`new TextEncoderStream()` Creates a new `TextEncoderStream` instance.

`textEncoderStream.encoding`

- Type: {string}

The encoding supported by the `TextEncoderStream` instance.

`textEncoderStream.readable`

- Type: {ReadableStream}

`textEncoderStream.writable`

- Type: {WritableStream}

Class: `TextDecoderStream`

`new TextDecoderStream([encoding[, options]])`

- `encoding` {string} Identifies the `encoding` that this `TextDecoder` instance supports. **Default:** `'utf-8'`.
- `options` {Object}
  - `fatal` {boolean} `true` if decoding failures are fatal.
  - `ignoreBOM` {boolean} When `true`, the `TextDecoderStream` will include the byte order mark in the decoded result. When `false`, the byte order mark will be removed from the output. This option is only used when `encoding` is `'utf-8'`, `'utf-16be'` or `'utf-16le'`. **Default:** `false`.

Creates a new `TextDecoderStream` instance.

`textDecoderStream.encoding`

- Type: {string}

The encoding supported by the `TextDecoderStream` instance.

`textDecoderStream.fatal`

- Type: {boolean}

The value will be `true` if decoding errors result in a `TypeError` being thrown.

`textDecoderStream.ignoreBOM`

- Type: {boolean}

The value will be `true` if the decoding result will include the byte order mark.

`textDecoderStream.readable`

- Type: {ReadableStream}

`textDecoderStream.writable`

- Type: {WritableStream}

Class: `CompressionStream`

`new CompressionStream(format)`

- `format` {string} One of either `'deflate'` or `'gzip'`.

`compressionStream.readable`

- Type: {ReadableStream}

`compressionStream.writable`

- Type: {WritableStream}

Class: `DecompressionStream`

`new DecompressionStream(format)`

- `format` {string} One of either `'deflate'` or `'gzip'`.

`decompressionStream.readable`

- Type: {ReadableStream}

`decompressionStream.writable`

- Type: {WritableStream}

Utility Consumers

The utility consumer functions provide common options for consuming streams.

They are accessed using:

```
import {
  arrayBuffer,
  blob,
  buffer,
  json,
  text,
} from 'node:stream/consumers';
```

```
const {
  arrayBuffer,
  blob,
  buffer,
  json,
  text,
} = require('stream/consumers');
```

`streamConsumers.arrayBuffer(stream)`

- `stream` {ReadableStream|stream.Readable|AsyncIterator}
- Returns: {Promise} Fulfills with an `ArrayBuffer` containing the full contents of the stream.

`streamConsumers.blob(stream)`

- `stream` {ReadableStream|stream.Readable|AsyncIterator}

- Returns: {Promise} Fulfills with a {Blob} containing the full contents of the stream.

`streamConsumers.buffer(stream)`

- `stream` {ReadableStream|stream.Readable|AsyncIterator}
- Returns: {Promise} Fulfills with a {Buffer} containing the full contents of the stream.

`streamConsumers.json(stream)`

- `stream` {ReadableStream|stream.Readable|AsyncIterator}
- Returns: {Promise} Fulfills with the contents of the stream parsed as a UTF-8 encoded string that is then passed through `JSON.parse()`.

`streamConsumers.text(stream)`

- `stream` {ReadableStream|stream.Readable|AsyncIterator}
- Returns: {Promise} Fulfills with the contents of the stream parsed as a UTF-8 encoded string.