Iterable Streams

James Snell · Web Engines Hackfest 2026

WinterTC proposal · https://github.com/WinterTC55/iter-streams

Who really wants a new streams API?

Literally, no rational person ever.

Hi, I’m James, and I’m talking about a new streams API.

Web Streams have a complexity and performance challenge

"One of the most frustrating parts … is how he believes that his implementations' performance problems are fundamental, and arise from the design decisions in the standard. This betrays a naïve mindset wherein implementers can get good performance out of the box, just by transcribing steps from specification text into JavaScript code."

Domenic Denicola, https://domenic.me/streams-standard/

Users expect to (and do) write code the way the standard allows them to.

We have to meet users where they are.

Streams aren’t hard to implement; they are hard to optimize given how they are used.

They have a lot of moving parts

ReadableStream, WritableStream, TransformStream, ReadableStreamDefaultController, ReadableByteStreamController, WritableStreamDefaultController, TransformStreamDefaultController, ReadableStreamBYOBRequest, WritableStreamDefaultWriter, ReadableStreamDefaultReader, ReadableStreamBYOBReader, ByteLengthQueuingStrategy, CountQueuingStrategy.

140 Abstract operations. 60+ internal slots.

They are both push and pull, sometimes at the same time

Users can, and do, use the API in ways that make optimization quite difficult.

```
const rs = new ReadableStream({
  async pull(controller) {
    controller.enqueue( ... );
  }
});
```

```
let controller;
const rs = new ReadableStream({
  start(c) {
    controller = c;
  }
});
controller.enqueue('push');  // push!
const reader = rs.getReader();
await reader.read(); // pull!
```

Things like fetch can be optimized easily and well:

```
const response = await fetch(url);

return new Response(response.pipeThrough(new DecompressionStream('gzip')));
```

Because everything is hidden.

In Browsers, at least, users can’t observe what’s happening under the hood.

That’s not true in Node.js and other servers, where observability and diagnostics are an important use case.

It’s easier to optimize what the user cannot see.

This is far more difficult:

```
const rs = new ReadableStream({
  type: 'bytes',
  async pull(controller) {
    // Is it a BYOB or default read?
    if (controller.pullRequest) {
    } else {
      controller.enqueue( ... )
    }
  }
});

return new Response(response.pipeThrough(new TransformStream({
  async transform(chunk, controller) {
    controller.enqueue(performTransform(chunk));
  }
})));
```

Everything is exposed and observable.

Or, even worse…

```
let controller;
const rs = new ReadableStream({
  start(c) {
    controller = c;
  }
});

// Use controller outside of the pull cycle
controller.enqueue( ... );
controller.enqueue( ... )
```

We don’t just have to accept complexity

What if the API were designed to be more optimizable?

Not "optimize around the spec" but "the spec is the optimization"

Iteration protocols instead of custom reader/writer locking

`for-await`

Batched yields: `Uint8Array[]`
instead of `Uint8Array`

Pull-through transforms that compose without intermediate state machines

Byte-budget backpressure with enforced policies

Bytes only – no polymorphic value/byte source distinction

Iterable Streams

Core idea: streams are iterables

An async stream is an `AsyncIterable<Uint8Array[]>`.

A sync stream is an `Iterable<Uint8Array[]>`.

There is no stream class.

```
// This is the entire consumption API
for await (const chunks of readable) {
  for (const chunk of chunks) {
    process(chunk);
  }
}
```

- Engines already optimize async iteration surprisingly well
- No reader, no API level locking
- `break` and `return` work naturally for cancellation

Why `Uint8Array[]`?

Batching amortizes the per-iteration async cost.

```
Web Streams:   1 chunk  → 1 promise → 1 microtask tick
Iterable:      N chunks → 1 promise → 1 microtask tick
```

The inner array is the batching mechanism. When the buffer has 10 chunks ready, the consumer gets all 10 in a single yield rather than 10 separate reads.

We should probably make this possible in Web Streams too.

The `Stream` namespace

No classes. A namespace of factory functions and utilities.

```
Stream.push()          // Push stream: writer + async iterable pair
Stream.from()          // Normalize strings, buffers, generators, etc.
Stream.duplex()        // Bidirectional channel pair (like socketpair)

Stream.pull()          // Lazy pull-through transform chain
Stream.pipeTo()        // Pipe source → transforms → writer destination

Stream.bytes()         // Collect as Uint8Array
Stream.text()          // Collect as string (UTF-8)

Stream.broadcast()     // Push-model: one writer, many consumers
Stream.share()         // Pull-model: shared source, many consumers

Stream.merge()         // Interleave multiple async sources
Stream.ondrain()       // Wait for backpressure to clear
```

Push streams

`Stream.push()` returns a bonded `{ writer, readable }` pair. The writer produces; the readable is an `AsyncIterable<Uint8Array[]>`.

```
const { writer, readable } = Stream.push({
  budget: 65536,
  backpressure: 'strict'
});

// Writer is just an interface. Not a specific class.
(async () => {
  await writer.write('Hello ');
  await writer.write('World!');
  await writer.end();
})();

// Consumer
console.log(await Stream.text(readable)); // "Hello World!"
```

No controller, no underlying source callback, no `enqueue()`.

The writer is the intake. The iterable is the outlet.

Pull pipelines

Transforms are lazy. Nothing executes until the consumer iterates.

```
// Build a pipeline — no data flows yet
const output = Stream.pull(source, compress, encrypt);

// Data flows on demand
for await (const chunks of output) { /* ... */ }

// Or pipe to a destination
const bytesWritten = await Stream.pipeTo(
  source, compress, encrypt, destinationWriter
);
```

Transforms are just functions:

```
// Stateless: (chunks, { signal }) => result
const upper = (chunks) => chunks?.map(c => encode(decode(c).toUpperCase()));

// Stateful: { transform: (source, { signal }) => AsyncIterable }
const lineBuffer = {
  async *transform(source) { /* accumulate lines across chunks */ }
};
```

No `TransformStream`, no controller, no `flush()` callback.

A `null` flush signal is part of the source iterable.

Backpressure: byte budgets

The writer's internal buffer has a **byte budget**.

```
budget: 16KB

Write 1: 4KB    buffered: 0→4KB      under budget → accepted (true)
Write 2: 4KB    buffered: 4→8KB      under budget → accepted (true)
Write 3: 4KB    buffered: 8→12KB     under budget → accepted (true)
Write 4: 10KB   buffered: 12→22KB    under budget → accepted (overshoots, that's OK)
Write 5: any    buffered: 22KB       over budget  → policy decides
```

Four backpressure policies

Policy	Available	Pending	Over budget
`strict` (default)	byte budget	1 operation	writeSync: `false`; write: pending or reject
`unbounded`	byte budget	unlimited	writeSync: `false`; write: always pending
`drop-oldest`	byte budget	none	evict oldest until under budget
`drop-newest`	byte budget	none	discard incoming

The try-fallback pattern

Synchronous fast path with async fallback.

```
// Try sync first — zero promise overhead when buffer has space
if (!writer.writeSync(chunk)) {
  // Budget exhausted. Fall back to async.
  await writer.write(chunk);
}
```

For sync pipelines (`pipeToSync`), there is no async fallback. If `writeSync` returns `false` , the pipe throws — the destination can't keep up.

```
// Fully synchronous pipeline — no promises at all
const bytes = Stream.pipeToSync(source, transform, syncWriter);
```

Multi-consumer patterns

Replacing `tee()` with bounded alternatives

The tee() problem

```
tee() branches:

  Source ──► Branch A (fast) ──► consumer A
         └─► Branch B (slow) ──► consumer B

  Branch A reads 1GB while Branch B has read 10KB.
  tee() must buffer 1GB - 10KB ≈ 1GB for Branch B.
  This is baked into the spec. There is no bound.
```

Implementers are told they can implement it differently if they want. But differences are often observable.

Iterable Streams replaces `tee()` with two explicit patterns:

- `broadcast()` – push model: writer pushes to all consumers
- `share()` – pull model: consumers pull from shared source

Both require an explicit byte budget.

Broadcast and Share

Broadcast (push)

```
const { writer, broadcast } =
  Stream.broadcast({ budget: 131072 });

const c1 = broadcast.push();
const c2 = broadcast.push(decompress);

// Write once, deliver to all
await writer.write(data);
await writer.end();

await Promise.all([
  Stream.text(c1),
  Stream.text(c2)
]);
```

Share (pull)

```
const shared = Stream.share(
  source,
  { budget: 131072 }
);

const c1 = shared.pull();
const c2 = shared.pull(decompress);

// Consumers pull on demand
const [raw, decompressed] =
  await Promise.all([
    Stream.bytes(c1),
    Stream.bytes(c2),
  ]);
```

Backpressure governed by the **slowest consumer**. When the byte budget is exhausted, the policy decides: strict rejects, unbounded waits, drop-oldest evicts, drop-newest discards.

Engine optimization opportunities

What this design enables

What engines can exploit

- **Batched iteration is the default** — The `Uint8Array[]` yield isn't an optimization bolt-on. It's the protocol. Engines don't need to detect batching opportunities.
- **No reader aliasing** — No locked reader object holding references to shared internal state. The async iterator is the only interface. Simpler lifetime analysis.
- **Pull-through composes** — A pipeline `pull(source, t1, t2, t3)` is a single composed async iterable. No intermediate TransformStream state machines. An engine could potentially inline the entire chain.
- **Sync fast path is spec-level** — `writeSync` / `pipeToSync` / `bytesSync` are separate entry points. Engines don't need to detect "this async operation completed synchronously" — the caller chose the sync path explicitly.
- **Byte budget maps to system flow control** — QUIC, HTTP/2, TCP: all use byte-based windows. The byte budget model means the stream's backpressure and the transport's flow control speak the same language.
- **Bytes only** — No polymorphic `ReadableStream<string | Uint8Array | any>`. Every chunk is `Uint8Array`. Monomorphic dispatch, no type guards.

Interop: Web Streams aren't going away

Web Streams objects work with this API out of the box.

```
// ReadableStream implements Symbol.asyncIterator
// Stream.from() accepts any async iterable
const readable = Stream.from(existingReadableStream);

// Pipe from Web Streams through iterable transforms to a writer
await Stream.pipeTo(existingReadableStream, transform, writer);
```

- `ReadableStream` is an `AsyncIterable` — `Stream.from()` normalizes it automatically
- Each chunk is wrapped into a single-element `Uint8Array[]` batch
- No special-casing of `ReadableStream` in the spec — it just works via the iteration protocol
- This is also how Node.js streams interop — they're async iterables too
- It's trivial to have `ReadableStream` also produce `AyncIterable<Uint8Array[]>` batches if we want

Can’t we just improve Web Streams?

Sure, if there’s enough interest and consensus to do so.

Browser use cases can’t be the only consideration

"The WHATWG’s focus is on standards implementable in web browsers"

Specification status

- **Proposal to WinterTC (ECMA TC55)** – not yet adopted as a standard
- **Reference implementation**: TypeScript, 273 tests, published on npm (`new-streams`)
- **Implementation**: Node.js Experimental `import stream from 'node:stream/iter'`
- **Only DOM dependency**: AbortSignal – will migrate to TC39 cancellation if/when it ships

Thank you

https://github.com/WinterTC55/iter-streams