readme.md for @push.rocks/smartstream

A TypeScript-first library for creating and manipulating Node.js and Web streams with built-in backpressure handling, async transformations, and seamless Node.js ↔ Web stream interoperability.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Install

```
pnpm install @push.rocks/smartstream
```

The package ships with two entry points:

Entry Point	Import Path	Environment
Node.js (default)	`@push.rocks/smartstream`	Node.js — full stream utilities, duplex, intake, wrappers, and Node↔Web helpers
Web	`@push.rocks/smartstream/web`	Browser & Node.js — pure Web Streams API (`WebDuplexStream`)

Usage

All examples use ESM / TypeScript syntax.

□□ Importing

```
// Node.js — full API
import {
  SmartDuplex,
  StreamWrapper,
  StreamIntake,
  createTransformFunction,
  createPassThrough,
  nodewebhelpers,
} from '@push.rocks/smartstream';

// Web — browser-safe, zero Node.js dependencies
import { WebDuplexStream } from '@push.rocks/smartstream/web';
```

□□ SmartDuplex — The Core Stream Primitive

`SmartDuplex` extends Node.js `Duplex` with first-class async support, built-in backpressure management, and a clean functional API. Instead of overriding `_transform` or `_write` manually, you pass a `writeFunction` that receives each chunk along with a `tools` object.

Basic Transform

```
import { SmartDuplex } from '@push.rocks/smartstream';

const upperCaser = new SmartDuplex<Buffer, Buffer>({
  writeFunction: async (chunk, tools) => {
    // Return a value to push it downstream
    return Buffer.from(chunk.toString().toUpperCase());
  },
});

readableStream.pipe(upperCaser).pipe(writableStream);
```

Using `tools.push()` for Multiple Outputs

The `writeFunction` can emit multiple chunks per input via `tools.push()`:

```
const splitter = new SmartDuplex<string, string>({
  objectMode: true,
  writeFunction: async (chunk, tools) => {
    const words = chunk.split(' ');
    for (const word of words) {
      await tools.push(word);
    }
    // Returning nothing — output was already pushed
  },
});
```

Final Function

Run cleanup or emit final data when the writable side ends:

```
const aggregator = new SmartDuplex<number, number>({
  objectMode: true,
  writeFunction: async (chunk, tools) => {
    runningTotal += chunk;
    // Don't emit anything per-chunk
  },
  finalFunction: async (tools) => {
    return runningTotal; // Emitted as the last chunk
  },
});
```

Truncating a Stream Early

Call `tools.truncate()` inside `writeFunction` to signal that no more data should be read:

```
const limiter = new SmartDuplex<string, string>({
  objectMode: true,
  writeFunction: async (chunk, tools) => {
    if (chunk === 'STOP') {
      tools.truncate();
      return;
    }
    return chunk;
  },
```

```
});
```

Creating from a Buffer

```
const stream = SmartDuplex.fromBuffer(Buffer.from('hello world'));
stream.on('data', (chunk) => console.log(chunk.toString())); // "hello world"
```

Creating from a Web ReadableStream

Bridge the Web Streams API into a Node.js Duplex:

```
const response = await fetch('https://example.com/data');
const nodeDuplex = SmartDuplex.fromWebReadableStream(response.body);

nodeDuplex.pipe(processTransform).pipe(outputStream);
```

Getting Web Streams from SmartDuplex

Convert a `SmartDuplex` into Web `ReadableStream` + `WritableStream` pair:

```
const duplex = new SmartDuplex({
  writeFunction: async (chunk, tools) => {
    return transform(chunk);
  },
});

const { readable, writable } = await duplex.getWebStreams();
```

Debug Mode

Pass `debug: true` and `name` to get detailed internal logs:

```
const stream = new SmartDuplex({
  name: 'MyStream',
  debug: true,
  writeFunction: async (chunk, tools) => chunk,
});
```

□□ StreamWrapper — Pipeline Composition

`StreamWrapper` takes an array of streams, pipes them together, attaches error listeners on all of them, and returns a `Promise` that resolves when the pipeline finishes:

```
import { StreamWrapper } from '@push.rocks/smartstream';
import fs from 'fs';

const pipeline = new StreamWrapper([
  fs.createReadStream('./input.txt'),
  new SmartDuplex({
    writeFunction: async (chunk) => Buffer.from(chunk.toString().toUpperCase()),
  }),
  fs.createWriteStream('./output.txt'),
]);

await pipeline.run();
console.log('Pipeline complete!');
```

Error handling is automatic — if any stream in the array errors, the returned promise rejects:

```
pipeline.run()
  .then(() => console.log('Done'))
  .catch((err) => console.error('Pipeline failed:', err));
```

You can also listen for custom events across all streams:

```
pipeline.onCustomEvent('progress', () => {
  console.log('Progress event fired');
});
```

□□ StreamIntake — Dynamic Data Injection

`StreamIntake` is a `Readable` stream that lets you programmatically push data into a pipeline. It operates in object mode by default and provides a reactive observable (`pushNextObservable`) for demand-driven data production.

```
import { StreamIntake, SmartDuplex } from '@push.rocks/smartstream';

const intake = new StreamIntake<string>();
```

```
// Pipe through a transform
intake
  .pipe(new SmartDuplex({
    objectMode: true,
    writeFunction: async (chunk) => {
      console.log('Processing:', chunk);
      return chunk;
    },
  }))
  .on('data', (data) => console.log('Output:', data));

// Push data whenever it's ready
intake.pushData('Hello');
intake.pushData('World');
intake.signalEnd(); // Signal end-of-stream
```

Demand-driven Production with Observable

`pushNextObservable` emits whenever the stream is ready for more data — perfect for throttled or event-driven producers:

```
const intake = new StreamIntake<number>();

let counter = 0;
intake.pushNextObservable.subscribe(() => {
  if (counter < 100) {
    intake.pushData(counter++);
  } else {
    intake.signalEnd();
  }
});

intake.pipe(consumer);
```

Creating from Existing Streams

Wrap a Node.js `Readable` or a Web `ReadableStream`:

```
// From Node.js Readable
const intake = await StreamIntake.fromStream<Buffer>(fs.createReadStream('./data.bin'));
```

```
// From Web ReadableStream
const response = await fetch('https://example.com/stream');
const intake = await StreamIntake.fromStream<Uint8Array>(response.body);
```

⚡ Utility Functions

`createTransformFunction`

Quickly create a `SmartDuplex` from a simple async mapping function:

```
import { createTransformFunction } from '@push.rocks/smartstream';

const doubler = createTransformFunction<number, number>(async (n) => n * 2);

intakeStream.pipe(doubler).pipe(outputStream);
```

`createPassThrough`

Create an object-mode passthrough stream (useful as an intermediary or tee point):

```
import { createPassThrough } from '@push.rocks/smartstream';

const passThrough = createPassThrough();
source.pipe(passThrough).pipe(destination);
```

□□ WebDuplexStream — Pure Web Streams API

`WebDuplexStream` extends `TransformStream` and works in both browsers and Node.js. Import it from the `/web` subpath for zero Node.js dependencies.

```
import { WebDuplexStream } from '@push.rocks/smartstream/web';

const stream = new WebDuplexStream<number, number>({
  writeFunction: async (chunk, { push }) => {
    push(chunk * 2); // Push transformed data
  },
```

```
});

const writer = stream.writable.getWriter();
const reader = stream.readable.getReader();

// Write
await writer.write(5);
await writer.write(10);
await writer.close();

// Read
const { value } = await reader.read(); // 10
const { value: v2 } = await reader.read(); // 20
```

From a Uint8Array

```
const stream = WebDuplexStream.fromUInt8Array(new Uint8Array([1, 2, 3]));
const reader = stream.readable.getReader();
const { value } = await reader.read(); // Uint8Array [1, 2, 3]
```

Data Production with `readFunction`

Supply data into the stream from any async source:

```
const stream = new WebDuplexStream<string, string>({
  readFunction: async (tools) => {
    await tools.write('chunk 1');
    await tools.write('chunk 2');
    tools.done(); // Signal end
  },
  writeFunction: async (chunk, { push }) => {
    push(chunk.toUpperCase());
  },
});

const reader = stream.readable.getReader();
// reads "CHUNK 1", "CHUNK 2"
```

□□ Node ↔ Web Stream Converters

The `nodewebhelpers` namespace provides bidirectional converters between Node.js and Web Streams:

```
import { nodewebhelpers } from '@push.rocks/smartstream';
```

Function	From	To
`createWebReadableStreamFromFile(path)`	File path	Web `ReadableStream<Uint8Array>`
`convertWebReadableToNodeReadable(webStream)`	Web `ReadableStream`	Node.js `Readable`
`convertNodeReadableToWebReadable(nodeStream)`	Node.js `Readable`	Web `ReadableStream`
`convertWebWritableToNodeWritable(webWritable)`	Web `WritableStream`	Node.js `Writable`
`convertNodeWritableToWebWritable(nodeWritable)`	Node.js `Writable`	Web `WritableStream`

Example: Serve a File as a Web ReadableStream

```
const webStream = nodewebhelpers.createWebReadableStreamFromFile('./video.mp4');

// Use with fetch Response, service workers, etc.
return new Response(webStream, {
  headers: { 'Content-Type': 'video/mp4' },
});
```

Example: Convert Between Stream Types

```
import fs from 'fs';
import { nodewebhelpers } from '@push.rocks/smartstream';

// Node → Web
const nodeReadable = fs.createReadStream('./data.bin');
const webReadable = nodewebhelpers.convertNodeReadableToWebReadable(nodeReadable);

// Web → Node
const nodeReadable2 = nodewebhelpers.convertWebReadableToNodeReadable(webReadable);
nodeReadable2.pipe(fs.createWriteStream('./copy.bin'));
```

□□ Backpressure Handling

`SmartDuplex` uses a `BackpressuredArray` internally, bounded by `highWaterMark` (default: 1). When the downstream consumer is slow, the stream automatically pauses the upstream producer until space is available — no manual bookkeeping required.

```
const slow = new SmartDuplex({
  name: 'SlowConsumer',
  objectMode: true,
  highWaterMark: 1,
  writeFunction: async (chunk, tools) => {
    await new Promise((resolve) => setTimeout(resolve, 200));
    return chunk;
  },
});

const fast = new SmartDuplex({
  name: 'FastProducer',
  objectMode: true,
  writeFunction: async (chunk, tools) => {
    return chunk; // Instant processing
  },
});

// Backpressure is handled automatically between fast → slow
fast.pipe(slow).on('data', (d) => console.log(d));

for (let i = 0; i < 100; i++) {
  fast.write(`chunk-${i}`);
}
fast.end();
```

□□ Real-World Example: Processing Pipeline

```
import fs from 'fs';
import { SmartDuplex, StreamWrapper } from '@push.rocks/smartstream';

// Read → Transform → Filter → Write
const pipeline = new StreamWrapper([
  fs.createReadStream('./access.log'),
  new SmartDuplex({
    writeFunction: async (chunk) => {
      // Parse each line
      return chunk.toString().split('\n');
    },
  }),
  new SmartDuplex({
    objectMode: true,
    writeFunction: async (lines: string[], tools) => {
      // Filter and push matching lines
      for (const line of lines) {
        if (line.includes('ERROR')) {
          await tools.push(line + '\n');
        }
      }
    },
  }),
  fs.createWriteStream('./errors.log'),
]);

await pipeline.run();
console.log('Error extraction complete');
```

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #3
Created 2026-03-28 11:12:38 UTC by foss.global Team
Updated 2026-03-28 12:19:24 UTC by foss.global Team