readme.md for @push.rocks/smartrequest

A modern, cross-platform HTTP/HTTPS request library for Node.js, Bun, Deno, and browsers with a unified API, supporting form data, file uploads, JSON, binary data, streams, and unix sockets.

Install

```
# Using npm
npm install @push.rocks/smartrequest --save

# Using pnpm
pnpm add @push.rocks/smartrequest

# Using yarn
yarn add @push.rocks/smartrequest
```

Key Features

- □□ **Modern Fetch-like API** - Familiar response methods (`.json()`, `.text()`, `.arrayBuffer()`, `.stream()`)
- □□ **Cross-Platform** - Works in Node.js, Bun, Deno, and browsers with a unified API
- □□ **Unix Socket Support** - Connect to local services like Docker (Node.js, Bun, and Deno)
- □□ **Form Data & File Uploads** - Built-in support for multipart/form-data
- □□ **Pagination Support** - Multiple strategies (offset, cursor, Link headers)
- ⚡ **Keep-Alive Connections** - Efficient connection pooling in Node.js
- □□ **TypeScript First** - Full type safety and IntelliSense support
- □□ **Zero Magic Defaults** - Explicit configuration following fetch API principles
- □□ **Streaming Support** - Stream buffers, files, and custom data without loading into memory
- □□ **Highly Configurable** - Timeouts, retries, headers, rate limiting, and more

Architecture

SmartRequest features a multi-layer architecture that provides consistent behavior across platforms:

- **Core Base** - Abstract classes and unified types shared across implementations
- **Core Node** - Node.js implementation using native http/https modules with unix socket support
- **Core Bun** - Bun implementation using native fetch with unix socket support via `unix` option
- **Core Deno** - Deno implementation using fetch with unix socket support via HttpClient proxy
- **Core Fetch** - Browser implementation using the Fetch API
- **Core** - Dynamic runtime detection and implementation selection using @push.rocks/smartenv
- **Client** - High-level fluent API for everyday use

The library automatically detects the runtime environment (Deno, Bun, Node.js, or browser) and loads the appropriate implementation, ensuring optimal performance and native feature support for each platform.

Usage

`@push.rocks/smartrequest` provides a clean, type-safe API inspired by the native fetch API but with additional features needed for modern applications.

Basic Usage

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Simple GET request
async function fetchUserData(userId: number) {
  const response = await SmartRequest.create()
    .url(`https://jsonplaceholder.typicode.com/users/${userId}`)
    .get();

  // Use the fetch-like response API
  const userData = await response.json();
  console.log(userData); // The parsed JSON response
```

```
}

// POST request with JSON body
async function createPost(title: string, body: string, userId: number) {
  const response = await SmartRequest.create()
    .url('https://jsonplaceholder.typicode.com/posts')
    .json({ title, body, userId })
    .post();

  const createdPost = await response.json();
  console.log(createdPost); // The created post
}
```

Direct Core API Usage

For advanced use cases, you can use the Core API directly:

```
import { CoreRequest } from '@push.rocks/smartrequest';

async function directCoreRequest() {
  const request = new CoreRequest('https://api.example.com/data', {
    method: 'GET',
    headers: {
      Accept: 'application/json',
    },
  });

  const response = await request.fire();
  const data = await response.json();
  return data;
}
```

Setting Headers and Query Parameters

```
import { SmartRequest } from '@push.rocks/smartrequest';

async function searchRepositories(query: string, perPage: number = 10) {
  const response = await SmartRequest.create()
```

```
    .url('https://api.github.com/search/repositories')
    .header('Accept', 'application/vnd.github.v3+json')
    .query({
      q: query,
      per_page: perPage.toString(),
    })
    .get();

  const data = await response.json();
  return data.items;
}
```

Handling Timeouts and Retries

```
import { SmartRequest } from '@push.rocks/smartrequest';

async function fetchWithRetry(url: string) {
  const response = await SmartRequest.create()
    .url(url)
    .timeout(5000) // 5 seconds timeout
    .retry(3) // Retry up to 3 times on failure
    .get();

  return await response.json();
}
```

Setting Request Options

Use the `options()` method to set any request options supported by the underlying implementation:

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Set various options
const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .options({
    keepAlive: true, // Enable connection reuse (Node.js)
    timeout: 10000, // 10 second timeout
```

```
    hardDataCuttingTimeout: 15000, // 15 second hard timeout
    // Platform-specific options are also supported
  })
  .get();
```

Working with Different Response Types

The API provides a fetch-like interface for handling different response types:

```
import { SmartRequest } from '@push.rocks/smartrequest';

// JSON response (default)
async function fetchJson(url: string) {
  const response = await SmartRequest.create().url(url).get();

  return await response.json(); // Parses JSON automatically
}

// Text response
async function fetchText(url: string) {
  const response = await SmartRequest.create().url(url).get();

  return await response.text(); // Returns response as string
}

// Binary data
async function downloadImage(url: string) {
  const response = await SmartRequest.create()
    .url(url)
    .accept('binary') // Optional: hints to server we want binary
    .get();

  const buffer = await response.arrayBuffer();
  return Buffer.from(buffer); // Convert ArrayBuffer to Buffer if needed
}

// Streaming response (Web Streams API - cross-platform)
async function streamLargeFile(url: string) {
  const response = await SmartRequest.create().url(url).get();
```

```
  // Get a web-style ReadableStream (works everywhere)
  const stream = response.stream();

  if (stream) {
    const reader = stream.getReader();

    try {
      while (true) {
        const { done, value } = await reader.read();
        if (done) break;
        console.log(`Received ${value.length} bytes of data`);
      }
    } finally {
      reader.releaseLock();
    }
  }
}

// Convert to Node.js stream if needed (Node.js only)
async function streamWithNodeApi(url: string) {
  const response = await SmartRequest.create().url(url).get();

  // Convert web stream to Node.js stream
  import { Readable } from 'stream';
  const webStream = response.stream();
  const nodeStream = Readable.fromWeb(webStream);

  nodeStream.on('data', (chunk) => {
    console.log(`Received ${chunk.length} bytes of data`);
  });

  return new Promise((resolve, reject) => {
    nodeStream.on('end', resolve);
    nodeStream.on('error', reject);
  });
}
```

Response Object Methods

The response object provides these methods:

- `json<T>(): Promise<T>` - Parse response as JSON
- `text(): Promise<string>` - Get response as text
- `arrayBuffer(): Promise<ArrayBuffer>` - Get response as ArrayBuffer
- `stream(): ReadableStream<Uint8Array> | null` - Get web-style ReadableStream (cross-platform)
- `raw(): Response | http.IncomingMessage` - Get the underlying platform response object

Each body method can only be called once per response, similar to the fetch API.

Important: Always Consume Response Bodies

You should always consume response bodies, even if you don't need the data.
Unconsumed response bodies can cause:

- Memory leaks as data accumulates in buffers
- Socket hanging with keep-alive connections
- Connection pool exhaustion

```
// □ BAD - Response body is not consumed
const response = await SmartRequest.create()
  .url('https://api.example.com/status')
  .get();

if (response.ok) {
  console.log('Success!');
}
// Socket may hang here!

// □ GOOD - Response body is consumed
const response = await SmartRequest.create()
  .url('https://api.example.com/status')
  .get();

if (response.ok) {
  console.log('Success!');
}
await response.text(); // Consume the body even if not needed
```

In Node.js, SmartRequest automatically drains unconsumed responses to prevent socket hanging, but it's still best practice to explicitly consume response bodies. When auto-drain occurs, you'll see a console log: `Auto-draining unconsumed response body for [URL] (status: [STATUS])`.

You can disable auto-drain if needed:

```
// Disable auto-drain (not recommended unless you have specific requirements)
const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .autoDrain(false) // Disable auto-drain
  .get();

// Now you MUST consume the body or the socket will hang
await response.text();
```

Advanced Features

Form Data with File Uploads

```
import { SmartRequest } from '@push.rocks/smartrequest';
import * as fs from 'fs';

async function uploadMultipleFiles(
  files: Array<{ name: string; path: string }>,
) {
  const formFields = files.map((file) => ({
    name: 'files',
    value: fs.readFileSync(file.path),
    filename: file.name,
    contentType: 'application/octet-stream',
  }));

  const response = await SmartRequest.create()
    .url('https://api.example.com/upload')
    .formData(formFields)
    .post();
```

```
  return await response.json();
}
```

Streaming Request Bodies

SmartRequest provides multiple ways to stream data in requests, making it easy to upload large files or send real-time data without loading everything into memory:

```
import { SmartRequest } from '@push.rocks/smartrequest';
import * as fs from 'fs';
import { Readable } from 'stream';

// Stream a Buffer directly (works everywhere)
async function uploadBuffer() {
  const buffer = Buffer.from('Hello, World!');

  const response = await SmartRequest.create()
    .url('https://api.example.com/upload')
    .buffer(buffer, 'text/plain')
    .post();

  return await response.json();
}

// Stream using web ReadableStream (cross-platform!)
async function uploadWebStream() {
  const stream = new ReadableStream({
    start(controller) {
      const data = new TextEncoder().encode('Stream data');
      controller.enqueue(data);
      controller.close();
    },
  });

  const response = await SmartRequest.create()
    .url('https://api.example.com/upload')
    .stream(stream, 'text/plain')
    .post();
```

```
  return await response.json();
}

// Stream a file using Node.js streams (Node.js only)
async function uploadLargeFile(filePath: string) {
  const fileStream = fs.createReadStream(filePath);

  const response = await SmartRequest.create()
    .url('https://api.example.com/upload')
    .stream(fileStream, 'application/octet-stream')
    .post();

  return await response.json();
}

// Stream data from any readable source (Node.js only)
async function streamData(dataSource: Readable) {
  const response = await SmartRequest.create()
    .url('https://api.example.com/stream')
    .stream(dataSource)
    .post();

  return await response.json();
}

// Send Uint8Array (works everywhere)
async function uploadBinaryData() {
  const data = new Uint8Array([72, 101, 108, 108, 111]); // "Hello"

  const response = await SmartRequest.create()
    .url('https://api.example.com/binary')
    .buffer(data, 'application/octet-stream')
    .post();

  return await response.json();
}
```

Streaming Methods

- **`.buffer(data, contentType?)`** - Stream a Buffer or Uint8Array directly

    - `data`: Buffer (Node.js) or Uint8Array (cross-platform) to send
    - `contentType`: Optional content type (defaults to 'application/octet-stream')
    - □ Works everywhere (Node.js, Bun, Deno, browsers)
- **`.stream(stream, contentType?)`** - Stream from ReadableStream or Node.js stream
    - `stream`: Web ReadableStream (cross-platform) or Node.js stream (Node.js only)
    - `contentType`: Optional content type
    - □ Web ReadableStream works everywhere (Node.js, Bun, Deno, browsers)
    - ⚠ Node.js streams only work in Node.js (automatically converted to web streams in Bun/Deno)

These methods are particularly useful for:

- Uploading large files without loading them into memory
- Streaming real-time data to servers
- Proxying data between services
- Implementing chunked transfer encoding

Unix Socket Support (Node.js, Bun, and Deno)

SmartRequest supports unix sockets across all server-side runtimes with a unified API:

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Connect to a service via Unix socket (works on Node.js, Bun, and Deno)
async function queryViaUnixSocket() {
  const response = await SmartRequest.create()
    .url('http://unix:/var/run/docker.sock:/v1.24/containers/json')
    .get();

  return await response.json();
}

// Alternative: Use socketPath option (works on all server runtimes)
async function queryWithSocketPath() {
  const response = await SmartRequest.create()
    .url('http://localhost/version')
    .options({ socketPath: '/var/run/docker.sock' })
    .get();
```

```
  return await response.json();
}
```

Runtime-Specific Unix Socket APIs

Each runtime implements unix sockets using its native capabilities:

Bun:

```
import { CoreRequest } from '@push.rocks/smartrequest/core_bun';

// Bun uses the native `unix` fetch option
const response = await CoreRequest.create('http://localhost/version', {
  unix: '/var/run/docker.sock'
});
```

Deno:

```
import { CoreRequest } from '@push.rocks/smartrequest/core_deno';

// Deno uses HttpClient with unix socket proxy
const client = Deno.createHttpClient({
  proxy: { url: 'unix:///var/run/docker.sock' }
});

const response = await CoreRequest.create('http://localhost/version', {
  client
});

// Clean up when done
client.close();
```

Node.js:

```
import { CoreRequest } from '@push.rocks/smartrequest/core_node';

// Node.js uses native socketPath option
const response = await CoreRequest.create('http://localhost/version', {
  socketPath: '/var/run/docker.sock'
});
```

Pagination Support

The library includes built-in support for various pagination strategies:

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Offset-based pagination (page & limit)
async function fetchAllUsers() {
  const client = SmartRequest.create()
    .url('https://api.example.com/users')
    .withOffsetPagination({
      pageParam: 'page',
      limitParam: 'limit',
      startPage: 1,
      pageSize: 20,
      totalPath: 'meta.total',
    });

  // Get first page with pagination info
  const firstPage = await client.getPaginated();
  console.log(`Found ${firstPage.items.length} users on first page`);
  console.log(`Has more pages: ${firstPage.hasNextPage}`);

  if (firstPage.hasNextPage) {
    // Get next page
    const secondPage = await firstPage.getNextPage();
    console.log(`Found ${secondPage.items.length} more users`);
  }

  // Or get all pages at once (use with caution for large datasets)
  const allUsers = await client.getAllPages();
  console.log(`Retrieved ${allUsers.length} users in total`);
}

// Cursor-based pagination
async function fetchAllPosts() {
  const allPosts = await SmartRequest.create()
    .url('https://api.example.com/posts')
    .withCursorPagination({
```

```
      cursorParam: 'cursor',
      cursorPath: 'meta.nextCursor',
      hasMorePath: 'meta.hasMore',
    })
    .getAllPages();

  console.log(`Retrieved ${allPosts.length} posts in total`);
}

// Link header-based pagination (GitHub API style)
async function fetchAllIssues(repo: string) {
  const paginatedResponse = await SmartRequest.create()
    .url(`https://api.github.com/repos/${repo}/issues`)
    .header('Accept', 'application/vnd.github.v3+json')
    .withLinkPagination()
    .getPaginated();

  return paginatedResponse.getAllPages();
}
```

Keep-Alive Connections (Node.js)

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Enable keep-alive for better performance with multiple requests
async function performMultipleRequests() {
  // Note: keepAlive is NOT enabled by default
  const response1 = await SmartRequest.create()
    .url('https://api.example.com/endpoint1')
    .options({ keepAlive: true })
    .get();

  const response2 = await SmartRequest.create()
    .url('https://api.example.com/endpoint2')
    .options({ keepAlive: true })
    .get();

  // Connections are pooled and reused when keepAlive is enabled
```

```
  return [await response1.json(), await response2.json()];
}
```

Rate Limiting (429 Too Many Requests) Handling

The library includes built-in support for handling HTTP 429 (Too Many Requests) responses with intelligent backoff:

```
import { SmartRequest } from '@push.rocks/smartrequest';

// Simple usage - handle 429 with defaults
async function fetchWithRateLimitHandling() {
  const response = await SmartRequest.create()
    .url('https://api.example.com/data')
    .handle429Backoff() // Automatically retry on 429
    .get();

  return await response.json();
}

// Advanced usage with custom configuration
async function fetchWithCustomRateLimiting() {
  const response = await SmartRequest.create()
    .url('https://api.example.com/data')
    .handle429Backoff({
      maxRetries: 5, // Try up to 5 times (default: 3)
      respectRetryAfter: true, // Honor Retry-After header (default: true)
      maxWaitTime: 30000, // Max 30 seconds wait (default: 60000)
      fallbackDelay: 2000, // 2s initial delay if no Retry-After (default: 1000)
      backoffFactor: 2, // Exponential backoff multiplier (default: 2)
      onRateLimit: (attempt, waitTime) => {
        console.log(`Rate limited. Attempt ${attempt}, waiting ${waitTime}ms`);
      },
    })
    .get();

  return await response.json();
```

```
}

// Example: API client with rate limit handling
class RateLimitedApiClient {
  private async request(path: string) {
    return SmartRequest.create()
      .url(`https://api.example.com${path}`)
      .handle429Backoff({
        maxRetries: 3,
        onRateLimit: (attempt, waitTime) => {
          console.log(
            `API rate limit hit. Waiting ${waitTime}ms before retry ${attempt}`,
          );
        },
      });
  }

  async fetchData(id: string) {
    const response = await this.request(`/data/${id}`).get();
    return response.json();
  }
}
```

The rate limiting feature:

- Automatically detects 429 responses and retries with backoff
- Respects the `Retry-After` header when present (supports both seconds and HTTP date formats)
- Uses exponential backoff when no `Retry-After` header is provided
- Allows custom callbacks for monitoring rate limit events
- Caps maximum wait time to prevent excessive delays

Platform-Specific Features

Browser-Specific Options

When running in a browser, you can use browser-specific fetch options:

```
const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .options({
    credentials: 'include', // Include cookies
    mode: 'cors', // CORS mode
    cache: 'no-cache', // Cache mode
    referrerPolicy: 'no-referrer',
  })
  .get();
```

Node.js-Specific Options

When running in Node.js, you can use Node-specific options:

```
import { Agent } from 'https';

const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .options({
    agent: new Agent({ keepAlive: true }), // Custom agent
    socketPath: '/var/run/api.sock', // Unix socket
  })
  .get();
```

Bun-Specific Options

When running in Bun, you can use Bun-specific options:

```
const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .options({
    unix: '/var/run/api.sock', // Unix socket (Bun's native option)
    keepAlive: true, // Keep-alive support
  })
  .get();

// Bun uses web streams natively
const streamResponse = await SmartRequest.create()
```

```
  .url('https://api.example.com/data')
  .get();

const webStream = streamResponse.stream(); // □ Use web streams in Bun
```

Deno-Specific Options

When running in Deno, you can use Deno-specific options:

```
// Custom HttpClient for advanced configuration
const client = Deno.createHttpClient({
  proxy: { url: 'unix:///var/run/api.sock' }
});

const response = await SmartRequest.create()
  .url('https://api.example.com/data')
  .options({
    client, // Custom Deno HttpClient
  })
  .get();

// Remember to clean up clients when done
client.close();

// Deno uses web streams natively
const streamResponse = await SmartRequest.create()
  .url('https://api.example.com/data')
  .get();

const webStream = streamResponse.stream(); // □ Use web streams in Deno
```

Complete Example: Building a REST API Client

Here's a complete example of building a typed API client:

```typescript
import { SmartRequest, type ICoreResponse } from '@push.rocks/smartrequest';

interface User {
  id: number;
  name: string;
  email: string;
}

interface Post {
  id: number;
  title: string;
  body: string;
  userId: number;
}

class BlogApiClient {
  private baseUrl = 'https://jsonplaceholder.typicode.com';

  private async request(path: string) {
    return SmartRequest.create()
      .url(`${this.baseUrl}${path}`)
      .header('Accept', 'application/json');
  }

  async getUser(id: number): Promise<User> {
    const response = await this.request(`/users/${id}`).get();
    return response.json<User>();
  }

  async createPost(post: Omit<Post, 'id'>): Promise<Post> {
    const response = await this.request('/posts').json(post).post();
    return response.json<Post>();
  }

  async deletePost(id: number): Promise<void> {
    const response = await this.request(`/posts/${id}`).delete();

    if (!response.ok) {
      throw new Error(`Failed to delete post: ${response.statusText}`);
```

```
    }

    // Consume the body
    await response.text();
  }

  async getAllPosts(userId?: number): Promise<Post[]> {
    const client = this.request('/posts');

    if (userId) {
      client.query({ userId: userId.toString() });
    }

    const response = await client.get();
    return response.json<Post[]>();
  }
}

// Usage
const api = new BlogApiClient();
const user = await api.getUser(1);
const posts = await api.getAllPosts(user.id);
```

Error Handling

```
import { SmartRequest } from '@push.rocks/smartrequest';

async function fetchWithErrorHandling(url: string) {
  try {
    const response = await SmartRequest.create()
      .url(url)
      .timeout(5000)
      .retry(2)
      .get();

    // Check if request was successful
    if (!response.ok) {
```

```
      throw new Error(`HTTP ${response.status}: ${response.statusText}`);
    }

    // Handle different content types
    const contentType = response.headers['content-type'];

    if (contentType?.includes('application/json')) {
      return await response.json();
    } else if (contentType?.includes('text/')) {
      return await response.text();
    } else {
      return await response.arrayBuffer();
    }
  } catch (error) {
    if (error.code === 'ECONNREFUSED') {
      console.error('Connection refused - is the server running?');
    } else if (error.code === 'ETIMEDOUT') {
      console.error('Request timed out');
    } else if (error.name === 'AbortError') {
      console.error('Request was aborted');
    } else {
      console.error('Request failed:', error.message);
    }
    throw error;
  }
}
```

Migrating from Earlier Versions

From v4.x to v5.x

Version 5.0 completes the transition to modern web standards by removing Node.js-specific streaming APIs:

Breaking Changes

1. `.streamNode()` **Method Removed**

- The `.streamNode()` method has been removed from all response objects
- Use the cross-platform `.stream()` method instead, which returns a web `ReadableStream<Uint8Array>`
- For Node.js users who need Node.js streams, convert using `Readable.fromWeb()`

```
// ❌ Before (v4.x) - Node.js only
const response = await SmartRequest.create().url(url).get();
const nodeStream = response.streamNode();

// ✅ After (v5.x) - Cross-platform
import { Readable } from 'stream';

const response = await SmartRequest.create().url(url).get();
const webStream = response.stream();
const nodeStream = Readable.fromWeb(webStream); // Convert to Node.js stream
```

2. **Request `.raw()` Method Removed**
   - The `.raw(streamFunc)` method has been removed from the SmartRequest client
   - Use `.stream()` with a web `ReadableStream` instead for request body streaming
   - Node.js users can create web streams from Node.js streams using `Readable.toWeb()`

```
// ❌ Before (v4.x) - Node.js only
const response = await SmartRequest.create()
  .url(url)
  .raw((request) => {
    request.write('chunk1');
    request.write('chunk2');
    request.end();
  })
  .post();

// ✅ After (v5.x) - Cross-platform
const stream = new ReadableStream({
  start(controller) {
    controller.enqueue(new TextEncoder().encode('chunk1'));
    controller.enqueue(new TextEncoder().encode('chunk2'));
    controller.close();
  }
});

const response = await SmartRequest.create()
  .url(url)
```

```
  .stream(stream)
  .post();

// Or convert from Node.js stream (Node.js only)
import { Readable } from 'stream';
import * as fs from 'fs';

const nodeStream = fs.createReadStream('file.txt');
const webStream = Readable.toWeb(nodeStream);

const response = await SmartRequest.create()
  .url(url)
  .stream(webStream)
  .post();
```

3. **Response `.raw()` Method Preserved**
   - The `response.raw()` method is still available for accessing platform-specific response objects
   - Returns `http.IncomingMessage` in Node.js or `Response` in other runtimes
   - Use for advanced scenarios requiring access to raw platform objects

```
// □ Still works in v5.x
const response = await SmartRequest.create().url(url).get();
const rawResponse = response.raw(); // http.IncomingMessage or Response
```

Migration Guide

For Response Streaming:

```
// Before (v4.x)
const response = await SmartRequest.create().url(url).get();
const nodeStream = response.streamNode();

nodeStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
});

// After (v5.x) - Option 1: Use web streams directly
const response = await SmartRequest.create().url(url).get();
const webStream = response.stream();
```

```
if (webStream) {
  const reader = webStream.getReader();
  while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    console.log(`Received ${value.length} bytes`);
  }
  reader.releaseLock();
}

// After (v5.x) - Option 2: Convert to Node.js stream (Node.js only)
import { Readable } from 'stream';

const response = await SmartRequest.create().url(url).get();
const webStream = response.stream();
const nodeStream = Readable.fromWeb(webStream);

nodeStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
});
```

For Request Streaming:

Node.js streams are still accepted by the `.stream()` method and automatically converted internally. No changes required for most use cases:

```
// ✅ Still works in v5.x
import * as fs from 'fs';

const fileStream = fs.createReadStream('large-file.bin');
const response = await SmartRequest.create()
  .url('https://api.example.com/upload')
  .stream(fileStream, 'application/octet-stream')
  .post();
```

Benefits:

- ✅ True cross-platform compatibility
- ✅ Modern web standards
- ✅ Cleaner API surface
- ✅ Single streaming approach works everywhere

From v3.x to v4.x

Version 4.0 adds comprehensive cross-platform support:

1. **Multi-Runtime Support**: Now works natively in Node.js, Bun, Deno, and browsers
2. **Unix Sockets Everywhere**: Unix socket support added for Bun and Deno
3. **Web Streams**: Full support for web ReadableStream across all platforms
4. **Automatic Runtime Detection**: No configuration needed - works everywhere automatically

From v2.x to v3.x

Version 3.0 brought significant architectural improvements:

1. **Legacy API Removed**: The function-based API (getJson, postJson, etc.) has been removed. Use SmartRequest instead.
2. **Unified Response API**: All responses now use the same fetch-like interface regardless of platform.
3. **Stream Changes**: The `stream()` method now returns a web-style ReadableStream on all platforms. Use `streamNode()` for Node.js streams.
4. **Cross-Platform by Default**: The library now works in browsers out of the box with automatic platform detection.

License and Legal Information

Trademarks

Company Information

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

For any legal inquiries or if you require 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 11:11:41 UTC by foss.global Team
Updated 2026-03-28 12:18:33 UTC by foss.global Team