readme.md for @push.rocks/smartshell

Execute shell commands with superpowers in Node.js

npm version License: MIT

⚠ Security Notice

IMPORTANT: Please read the Security Guide below for critical information about command execution and input handling. Always use `execSpawn` methods for untrusted input.

Why smartshell? □□

Tired of wrestling with Node.js child processes? Meet `@push.rocks/smartshell` - your promise-based shell command companion that makes executing system commands feel like a breeze. Whether you're building automation scripts, CI/CD pipelines, or need fine-grained control over shell execution, smartshell has got you covered.

□ Key Features

- □□ **Promise-based API** - Async/await ready for modern codebases
- □□ **Silent execution modes** - Control output verbosity
- □□ **Streaming support** - Real-time output for long-running processes
- □□ **Interactive commands** - Handle user input programmatically
- □□ **Secure execution** - Shell-free methods for untrusted input
- ⚡ **Smart execution modes** - Strict, silent, or streaming
- □□ **Pattern matching** - Wait for specific output patterns
- □□ **Environment management** - Custom env vars and PATH handling
- □□ **Memory protection** - Built-in buffer limits prevent OOM
- □ **Timeout support** - Automatic process termination
- □□ **PTY support** - Full terminal emulation (optional)
- □□ **Cross-platform** - Windows, macOS, and Linux ready

- □□ **TypeScript first** - Full type safety and IntelliSense

Installation □□

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

# Using yarn
yarn add @push.rocks/smartshell

# Using pnpm (recommended)
pnpm add @push.rocks/smartshell

# Optional: For PTY support (terminal emulation)
pnpm add --save-optional node-pty
```

Quick Start □□♂

```
import { Smartshell } from '@push.rocks/smartshell';

// Create your shell instance
const shell = new Smartshell({
  executor: 'bash' // or 'sh' for lighter shells
});

// Run a simple command
const result = await shell.exec('echo "Hello, World!"');
console.log(result.stdout); // "Hello, World!"
console.log(result.signal); // undefined (no signal)
console.log(result.stderr); // "" (no errors)
```

Security-First Execution □□

Secure Command Execution with execSpawn

When dealing with untrusted input, **always use execSpawn methods** which don't use shell interpretation:

```
// □ DANGEROUS with untrusted input
const userInput = "file.txt; rm -rf /";
await shell.exec(`cat ${userInput}`); // Command injection!

// □ SAFE with untrusted input
await shell.execSpawn('cat', [userInput]); // Arguments are properly escaped
```

execSpawn Family Methods

```
// Basic secure execution
const result = await shell.execSpawn('ls', ['-la', '/home']);

// Streaming secure execution
const streaming = await shell.execSpawnStreaming('npm', ['install']);
await streaming.finalPromise;

// Interactive secure execution
const interactive = await shell.execSpawnInteractiveControl('cat', []);
await interactive.sendLine('Hello');
interactive.endInput();
await interactive.finalPromise;
```

Production Features □□

Resource Management

Prevent memory issues with built-in buffer limits:

```
const result = await shell.exec('large-output-command', {
  maxBuffer: 10 * 1024 * 1024, // 10MB limit
  onData: (chunk) => {
    // Process chunks as they arrive
    console.log('Received:', chunk.toString());
  }
});
```

Timeout Support

Automatically terminate long-running processes:

```
try {
  const result = await shell.execSpawn('long-process', [], {
    timeout: 5000 // 5 second timeout
  });
} catch (error) {
  console.log('Process timed out');
}
```

Debug Mode

Enable detailed logging for troubleshooting:

```
const result = await shell.exec('command', {
  debug: true // Logs process lifecycle events
});
```

Custom Environment

Control the execution environment precisely:

```
const result = await shell.execSpawn('node', ['script.js'], {
  env: {
    NODE_ENV: 'production',
    PATH: '/usr/bin:/bin',
    CUSTOM_VAR: 'value'
```

```
  }
});
```

Interactive Control □□

Programmatic Input Control

Send input to processes programmatically:

```
const interactive = await shell.execInteractiveControl('cat');

// Send input line by line
await interactive.sendLine('Line 1');
await interactive.sendLine('Line 2');

// Send raw input without newline
await interactive.sendInput('partial');

// Close stdin
interactive.endInput();

// Wait for completion
const result = await interactive.finalPromise;
```

Passthrough Mode

Connect stdin for real keyboard interaction:

```
// User can type directly
await shell.execPassthrough('vim file.txt');
```

PTY Support - Full Terminal Emulation □□

Smartshell provides two modes for executing interactive commands:

1. **Pipe Mode (Default)** - Fast, simple, no dependencies
2. **PTY Mode** - Full terminal emulation for advanced interactive programs

When to Use Each Mode

Use Pipe Mode (Default) When:

- Running simple commands that read from stdin
- Using tools like `cat`, `grep`, `sed`, `awk`
- Running basic scripts that don't need terminal features
- You want maximum performance and simplicity
- You don't want native dependencies

Use PTY Mode When:

- Running commands that require a real terminal:
  - Password prompts (`sudo`, `ssh`, `su`)
  - Interactive editors (`vim`, `nano`, `emacs`)
  - Terminal UIs (`htop`, `less`, `more`)
  - Programs with fancy prompts (`bash read -p`)
  - Tab completion and readline features
- You need terminal features:
  - ANSI colors and escape sequences
  - Terminal size control
  - Signal handling (Ctrl+C, Ctrl+Z)
  - Line discipline and special key handling

Installing PTY Support

PTY support requires the optional `node-pty` dependency:

```
# Install as optional dependency
pnpm add --save-optional node-pty

# Note: node-pty requires compilation and has platform-specific requirements
# - On Windows: Requires Visual Studio Build Tools
# - On macOS/Linux: Requires Python and build tools
```

PTY Usage Examples

```
// Use PTY for commands that need terminal features
const ptyInteractive = await shell.execInteractiveControlPty(
  "bash -c 'read -p \"Enter name: \" name && echo \"Hello, $name\"'"
);
await ptyInteractive.sendLine('John');
const result = await ptyInteractive.finalPromise;
// With PTY, the prompt "Enter name: " will be visible in stdout

// Streaming with PTY for real-time interaction
const ptyStreaming = await shell.execStreamingInteractiveControlPty('vim test.txt');
await ptyStreaming.sendInput('i'); // Enter insert mode
await ptyStreaming.sendInput('Hello from PTY!');
await ptyStreaming.sendInput('\x1b'); // ESC key
await ptyStreaming.sendInput(':wq\r'); // Save and quit
```

PTY vs Pipe Mode Comparison

Feature	Pipe Mode	PTY Mode
Dependencies	None	node-pty
Terminal Detection	`isatty()` returns false	`isatty()` returns true
Prompt Display	May not show	Always shows
Colors	Often disabled	Enabled
Signal Handling	Basic	Full (Ctrl+C, Ctrl+Z, etc.)
Line Ending	`\n`	`\r` (carriage return)
EOF Signal	Stream end	`\x04` (Ctrl+D)

Common PTY Patterns

```
// Password input (PTY required)
const sudo = await shell.execInteractiveControlPty('sudo ls /root');
await sudo.sendLine('mypassword');
const result = await sudo.finalPromise;

// Interactive REPL with colors
const node = await shell.execStreamingInteractiveControlPty('node');
await node.sendLine('console.log("PTY supports colors!")');
```

```
await node.sendLine('.exit');

// Handling terminal colors
const ls = await shell.execInteractiveControlPty('ls --color=always');
const result = await ls.finalPromise;
// result.stdout will contain ANSI color codes
```

PTY Fallback Strategy

Always provide a fallback for when PTY isn't available:

```
try {
  // Try PTY mode first
  const result = await shell.execInteractiveControlPty(command);
  // ...
} catch (error) {
  if (error.message.includes('node-pty')) {
    // Fallback to pipe mode
    console.warn('PTY not available, using pipe mode');
    const result = await shell.execInteractiveControl(command);
    // ...
  }
}
```

Advanced Pattern Matching □□

Enhanced execAndWaitForLine

Wait for patterns with timeout and auto-termination:

```
// Wait with timeout
await shell.execAndWaitForLine(
  'npm start',
  /Server listening on port/,
  false, // silent
  {
```

```
    timeout: 30000,        // 30 second timeout
    terminateOnMatch: true // Kill process after match
  }
);
```

Core Concepts □□

The Smartshell Instance

The heart of smartshell is the `Smartshell` class. Each instance maintains its own environment and configuration:

```
const shell = new Smartshell({
  executor: 'bash', // Choose your shell: 'bash' or 'sh'
  sourceFilePaths: ['/path/to/env.sh'], // Optional: source files on init
});
```

Execution Modes □□

Standard Execution

Perfect for general commands where you want to see the output:

```
const result = await shell.exec('ls -la');
console.log(result.stdout); // Directory listing
console.log(result.exitCode); // 0 for success
console.log(result.signal); // Signal if terminated
console.log(result.stderr); // Error output
```

Silent Execution

Run commands without printing to console - ideal for capturing output:

```
const result = await shell.execSilent('cat /etc/hostname');
// Output is NOT printed to console but IS captured in result
console.log(result.stdout);  // Access the captured output here
```

Strict Execution

Throws an error if the command fails - great for critical operations:

```
try {
  await shell.execStrict('critical-command');
  console.log('□ Command succeeded!');
} catch (error) {
  console.error('□ Command failed:', error.message);
  // Error includes exit code or signal information
}
```

Streaming Execution

For long-running processes or when you need real-time output:

```
const streaming = await shell.execStreaming('npm install');

// Access the child process directly
streaming.childProcess.stdout.on('data', (chunk) => {
  console.log('□□Installing:', chunk.toString());
});

// Control the process
await streaming.terminate(); // SIGTERM
await streaming.kill();      // SIGKILL
await streaming.keyboardInterrupt(); // SIGINT

// Wait for completion
await streaming.finalPromise;
```

Real-World Examples □□

Build Pipeline

```
const shell = new Smartshell({ executor: 'bash' });

// Clean build directory
await shell.execSilent('rm -rf dist');

// Run TypeScript compiler
const buildResult = await shell.execStrict('tsc');

// Run tests
await shell.execStrict('npm test');

// Build succeeded!
console.log('□ Build pipeline completed successfully');
```

CI/CD with Security

```
async function deployApp(branch: string, untrustedTag: string) {
  const shell = new Smartshell({ executor: 'bash' });

  // Use execSpawn for untrusted input
  await shell.execSpawnStrict('git', ['checkout', branch]);
  await shell.execSpawnStrict('git', ['tag', untrustedTag]);

  // Safe to use exec for hardcoded commands
  await shell.execStrict('npm run build');
  await shell.execStrict('npm run deploy');
}
```

Docker Compose Helper

```
const shell = new Smartshell({ executor: 'bash' });

// Start services and wait for readiness
console.log('□□Starting Docker services...');
```

```
await shell.execAndWaitForLine(
  'docker-compose up',
  /All services are ready/,
  false,
  {
    timeout: 60000,
    terminateOnMatch: false // Keep running after match
  }
);

// Run migrations
await shell.execStrict('docker-compose exec app npm run migrate');
console.log('□ Environment ready!');
```

Development Server with Auto-Restart

```
import { SmartExecution } from '@push.rocks/smartshell';

const shell = new Smartshell({ executor: 'bash' });
const devServer = new SmartExecution(shell, 'npm run dev');

// Watch for file changes and restart
fs.watch('./src', async () => {
  console.log('□□Changes detected, restarting...');
  await devServer.restart();
});
```

API Reference □□

Smartshell Class

Method	Description	Security
`exec(command)`	Execute with shell	⚠ Vulnerable to injection
`execSpawn(cmd, args)`	Execute without shell	□ Safe for untrusted input
`execSilent(command)`	Execute without console output	⚠ Vulnerable to injection

Method	Description	Security
`execStrict(command)`	Execute, throw on failure	⚠ Vulnerable to injection
`execStreaming(command)`	Stream output in real-time	⚠ Vulnerable to injection
`execSpawnStreaming(cmd, args)`	Stream without shell	Safe for untrusted input
`execInteractive(command)`	Interactive terminal mode	⚠ Vulnerable to injection
`execInteractiveControl(command)`	Programmatic input control	⚠ Vulnerable to injection
`execSpawnInteractiveControl(cmd, args)`	Programmatic control without shell	Safe for untrusted input
`execPassthrough(command)`	Connect stdin passthrough	⚠ Vulnerable to injection
`execInteractiveControlPty(command)`	PTY with programmatic control	⚠ Vulnerable to injection
`execStreamingInteractiveControlPty(command)`	PTY streaming with control	⚠ Vulnerable to injection
`execAndWaitForLine(cmd, regex, silent, opts)`	Wait for pattern match	⚠ Vulnerable to injection

Result Interfaces

```
interface IExecResult {
  exitCode: number;     // Process exit code
  stdout: string;       // Standard output
  signal?: NodeJS.Signals; // Termination signal
  stderr?: string;      // Error output
}

interface IExecResultStreaming {
  childProcess: ChildProcess;  // Node.js ChildProcess instance
  finalPromise: Promise<IExecResult>; // Resolves when process exits
  sendInput: (input: string) => Promise<void>;
  sendLine: (line: string) => Promise<void>;
  endInput: () => void;
  kill: () => Promise<void>;
  terminate: () => Promise<void>;
  keyboardInterrupt: () => Promise<void>;
  customSignal: (signal: NodeJS.Signals) => Promise<void>;
}

interface IExecResultInteractive extends IExecResult {
  sendInput: (input: string) => Promise<void>;
```

```
  sendLine: (line: string) => Promise<void>;
  endInput: () => void;
  finalPromise: Promise<IExecResult>;
}
```

Options Interface

```
interface IExecOptions {
  silent?: boolean;          // Suppress console output
  strict?: boolean;          // Throw on non-zero exit
  streaming?: boolean;       // Return streaming interface
  interactive?: boolean;     // Interactive mode
  passthrough?: boolean;     // Connect stdin
  interactiveControl?: boolean; // Programmatic input
  usePty?: boolean;          // Use pseudo-terminal
  ptyShell?: string;         // Custom PTY shell
  ptyCols?: number;          // PTY columns (default 120)
  ptyRows?: number;          // PTY rows (default 30)
  ptyTerm?: string;          // Terminal type (default 'xterm-256color')
  maxBuffer?: number;        // Max output buffer (bytes)
  onData?: (chunk: Buffer | string) => void; // Data callback
  timeout?: number;          // Execution timeout (ms)
  debug?: boolean;           // Enable debug logging
  env?: NodeJS.ProcessEnv;   // Custom environment
  signal?: AbortSignal;      // Abort signal
}
```

Security Guide

Command Injection Prevention

The standard `exec` methods use `shell: true`, which can lead to command injection vulnerabilities:

```
// □ DANGEROUS - Never do this with untrusted input
const userInput = "file.txt; rm -rf /";
await smartshell.exec(`cat ${userInput}`); // Will execute rm -rf /
```

```
// □ SAFE - Arguments are properly escaped
await smartshell.execSpawn('cat', [userInput]); // Will look for literal filename
```

Best Practices

1. **Always validate input**: Even with secure methods, validate user input
2. **Use execSpawn for untrusted data**: Never pass user input to shell-based methods
3. **Set resource limits**: Use `maxBuffer` and `timeout` for untrusted commands
4. **Control environment**: Don't inherit all env vars for sensitive operations
5. **Restrict signals**: Only allow specific signals from user input

Path Traversal Protection

```
// □ VULNERABLE
const file = userInput; // Could be "../../../etc/passwd"
await shell.exec(`cat ${file}`);

// □ SECURE
const path = require('path');
const safePath = path.join('/allowed/directory', path.basename(userInput));
await shell.execSpawn('cat', [safePath]);
```

Tips & Best Practices □□

1. **Security first**: Use `execSpawn` for any untrusted input
2. **Set resource limits**: Always use `maxBuffer` and `timeout` for untrusted commands
3. **Choose the right executor**: Use `bash` for full features, `sh` for minimal overhead
4. **Use strict mode for critical operations**: Ensures failures don't go unnoticed
5. **Stream long-running processes**: Better UX and memory efficiency
6. **Leverage silent modes**: When you only need to capture output
7. **Handle errors gracefully**: Check both `exitCode` and `signal`
8. **Clean up resources**: Streaming processes should be properly terminated
9. **Control environment**: Don't inherit all env vars for sensitive operations
10. **Enable debug mode**: For development and troubleshooting
11. **Use PTY for terminal UIs**: When programs need real terminal features
12. **Provide fallbacks**: Always handle PTY unavailability gracefully

Environment Customization

Smartshell provides powerful environment management:

```
// Add custom source files
shell.shellEnv.addSourceFiles([
  '/home/user/.custom_env',
  './project.env.sh'
]);

// Modify PATH
shell.shellEnv.pathDirArray.push('/custom/bin');
shell.shellEnv.pathDirArray.push('/usr/local/special');

// Your custom environment is ready
const result = await shell.exec('my-custom-command');
```

Shell Detection

Need to check if a command exists? We export the `which` utility:

```
import { which } from '@push.rocks/smartshell';

try {
  const gitPath = await which('git');
  console.log(`Git found at: ${gitPath}`);
} catch (error) {
  console.log('Git is not installed');
}
```

License and Legal Information

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 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, and any usage must be approved in writing by Task Venture Capital GmbH.

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 11:12:30 UTC by foss.global Team
Updated 2026-03-28 12:19:16 UTC by foss.global Team