

@push.rocks/smartcli

A library for easily creating observable CLI tasks with support for commands, arguments, and options.

- [readme.md for @push.rocks/smartcli](#)
- [docs/index.md for @push.rocks/smartcli](#)
- [changelog.md for @push.rocks/smartcli](#)

readme.md for @push.rocks/smartcli

A library for easily creating observable CLI tasks with support for commands, arguments, and options.

Install

To install the package, simply run:

```
npm install @push.rocks/smartcli --save
```

This command will add @push.rocks/smartcli as a dependency in your package.json and download it into your node_modules folder.

Usage

The @push.rocks/smartcli library was created to simplify the development of command-line tools using a reactive programming paradigm. Built with TypeScript and ESM syntax in mind, this library lets you define and handle CLI commands, arguments, and options easily. It leverages observables to allow asynchronous operations and reactive logic in your CLI commands, making it ideal for tasks like file manipulation, network calls, or interactive command processing.

In this Usage section, we will explore an extensive range of practical examples and in-depth explanations, guiding you through the entire process—from setting up a basic CLI to handling advanced asynchronous commands with observables. We'll discuss how to define commands, set up aliases, handle options, integrate asynchronous tasks, and manage errors gracefully. In doing so, you will gain a complete understanding of all that smartcli has to offer, supported by multiple comprehensive TypeScript examples.

1. Getting Started: Setting Up Your CLI

Before you start using smartcli, ensure that you are working in a TypeScript project environment. If you haven't set up TypeScript in your project yet, initialize it with the following commands:

```
npm install typescript @types/node --save-dev
npx tsc --init
```

Once your environment is ready, you can start by creating a new TypeScript file (e.g., cli.ts) where you will import smartcli and set up your command-line application.

Below is a minimal example demonstrating how to instantiate smartcli, create a basic command, and initiate command parsing. For this example, let's create a simple CLI tool that responds to the command "greet" by printing a greeting message.

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { Subject } from '@push.rocks/smartrx';

// Create an instance of Smartcli
const cli = new Smartcli();

// Define a command where the user types "greet"
const greetCommand = cli.addCommand('greet');

// Subscribe to the greet command's output, providing your logic
greetCommand.subscribe((argv: any) => {
  // Check if there is a "name" option available; if not, default to "World"
  const name = argv.name || 'World';
  console.log(`Hello, ${name}!`);
});

// Define a standard task if no command is provided (this will act as a fallback)
cli.standardCommand().subscribe((argv: any) => {
  console.log('No command specified. Showing default help usage:');
  console.log('Usage: cli greet --name [YOUR_NAME]');
});

// Optionally, add version support
cli.addVersion('1.0.0');

// Optionally, add a help command to provide custom help information
cli.addHelp({
```

```

helpText: `
SmartCLI Help - Usage Information:
  • greet: Prints a greeting message. Use --name to specify your name.
  • -v, --version: Prints the current version.
  • Without a command, the default task shows a help message.
`,
});

// Start parsing command-line arguments using smartcli
cli.startParse();

```

In the above script:

- We import the Smartcli class from the package.
- We create a new instance of Smartcli.
- We add a command named “greet” that will be executed when the user types “greet” in the CLI.
- We subscribe to the command’s observable, which fires when the command is matched.
- We add a standard command to serve as a fallback when no specific command is provided.
- We configure version and help output.
- Finally, we call startParse() to begin processing command-line inputs.

2. Defining Multiple Commands and Handling Options

smartcli enables you to define multiple commands that your CLI can execute. Imagine you want a CLI tool that supports several functionalities such as “install”, “uninstall”, and “update”. Each command may need to handle its own set of parameters and options. Let’s see how to set up multiple commands:

```

#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';

const cli = new Smartcli();

// Command: install
const installCommand = cli.addCommand('install');
installCommand.subscribe((argv: any) => {

```

```

// You can access additional command line options as properties on argv, e.g. --package
const packageName = argv.package;
if (!packageName) {
  console.log('Please specify the package to install using --package option.');
```

```

  return;
}
console.log(`Installing package: ${packageName} ...`);
// Add your asynchronous installation process here (e.g., calling external APIs or running
tasks)
});

// Command: uninstall
const uninstallCommand = cli.addCommand('uninstall');
uninstallCommand.subscribe((argv: any) => {
  const packageName = argv.package;
  if (!packageName) {
    console.log('Please specify the package to uninstall using --package option.');
```

```

    return;
  }
  console.log(`Uninstalling package: ${packageName} ...`);
  // Include uninstallation logic, such as cleaning up dependencies
});

// Command: update
const updateCommand = cli.addCommand('update');
updateCommand.subscribe((argv: any) => {
  console.log('Updating all packages...');
  // Execute update logic, possibly involving asynchronous operations
});

// A standard task, in case no command is provided
cli.standardCommand().subscribe((argv: any) => {
  console.log('No specific command provided. Please use one of the following: install,
uninstall, update.');
```

```

});

// Start parsing user input
cli.startParse();

```

In this example:

- Multiple commands are registered using `addCommand()`.
 - Each command's subscriber receives a parsed `argv` object containing options defined by the user.
 - We leverage the `argv` object to extract key options such as `package` (`--package`) that are necessary for the respective operation.
 - The standard command notifies the user if no valid command is entered.
-

3. Using Command Aliases

A common requirement for CLI applications is to provide short aliases for commands. With `smartcli`, you can set up aliases so that users can type shorthand commands instead of the full command names. Here's how to define and use command aliases:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';

const cli = new Smartcli();

// Add a primary command "install"
const installCommand = cli.addCommand('install');
installCommand.subscribe((argv: any) => {
  console.log('Executing installation process...');
  // Installation logic here
});

// Add an alias, so "i" will also trigger the "install" command
cli.addCommandAlias('install', 'i');

// You could define more aliases if needed:
cli.addCommandAlias('uninstall', 'uni');
cli.addCommandAlias('update', 'upd');

// Define other commands similarly
const uninstallCommand = cli.addCommand('uninstall');
uninstallCommand.subscribe((argv: any) => {
  console.log('Executing uninstallation process...');
});

const updateCommand = cli.addCommand('update');
```

```
updateCommand.subscribe((argv: any) => {
  console.log('Executing update process...');
});

// Start parsing command-line arguments
cli.startParse();
```

Key points regarding aliases:

- The `addCommandAlias` method maps a shorter or alternative command name to the primary command.
- Internally, when the command line input is parsed, `smartcli` checks for both the registered command and its aliases.
- This feature improves user experience by allowing shorter commands for frequently used operations.

4. Accessing Command-Line Options

`smartcli` is built on top of `yargs-parser`, making it easy to retrieve options specified on the command line. For example, you may have a command that requires options like `--config`, `--verbose`, or custom flags. The `getOption` method can help you retrieve these options anywhere in your script. Consider the following example:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';

const cli = new Smartcli();

// Define a command "deploy" that uses options
const deployCommand = cli.addCommand('deploy');
deployCommand.subscribe((argv: any) => {
  // Access options directly via the parsed argv object:
  const configFile = cli.getOption('config');
  const isVerbose = cli.getOption('verbose');

  console.log('Starting deployment...');
  if (configFile) {
    console.log(`Using config file: ${configFile}`);
  }
});
```

```
if (isVerbose) {
  console.log('Verbose mode is enabled.');
```



```
  // Your deployment logic could be asynchronous; use observables or promises as needed
});
```



```
// Start parsing the CLI arguments
cli.startParse();
```

In this scenario, the `getOption` method abstracts the direct use of `yargs-parser`, letting you focus on the application logic rather than argument parsing intricacies. Whether your CLI requires configuration files or toggles for debug modes, `getOption` enables a clean and efficient way to extract these parameters.

5. Integrating Asynchronous Operations with Observables

One of the most powerful aspects of `@push.rocks/smartcli` is its ability to integrate asynchronous tasks using observables. By leveraging the RxJS observable pattern, you can easily chain actions, manage asynchronous flows, and even handle errors gracefully during command execution.

Imagine a scenario where you have to fetch data from an external API or perform lengthy file I/O operations before the CLI outputs its result. In such cases, you can integrate asynchronous libraries or directly use RxJS to handle the asynchronous commands. Here's an example demonstrating asynchronous behavior:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { of, throwError } from 'rxjs';
import { delay, catchError, tap, switchMap } from 'rxjs/operators';

const cli = new Smartcli();

// Define a command 'fetch-data' that simulates asynchronous data retrieval
const fetchCommand = cli.addCommand('fetch-data');
fetchCommand.subscribe((argv: any) => {
  console.log('Starting the data fetching process...');
```

```

// Simulate a network request using an observable that emits a value after a delay
of({ data: 'Sample data from API' })
  .pipe(
    delay(2000),
    tap((response) => {
      console.log('Data received:', response.data);
    }),
    switchMap((response) => {
      // Further process the data asynchronously if needed
      return of(response.data + ' processed');
    }),
    tap((processedData) => {
      console.log('Processed Data:', processedData);
    }),
    catchError((error) => {
      console.error('An error occurred during data fetching:', error);
      return throwError(() => new Error('Data fetch failed'));
    })
  )
  .subscribe({
    next: (result) => {
      console.log('Asynchronous operation complete with result:', result);
    },
    error: (err: any) => {
      console.error('Error in observable chain:', err);
    },
  });
});

// Start parsing CLI input
cli.startParse();

```

In this example:

- We simulate an asynchronous API call using RxJS's `of()` combined with `delay()`.
- The observable chain processes the emitted value by tapping into the data, then further transforming it.
- Errors in the asynchronous pipeline are caught using `catchError` to allow for graceful degradation.
- Finally, the subscriber is notified when the asynchronous task completes.

This pattern is especially useful if your CLI commands perform operations that cannot be completed synchronously, such as interacting with remote servers, processing large datasets, or performing computationally expensive tasks.

6. Combining Multiple Features for a Complex CLI Application

Developers often need to build CLI tools that incorporate many of the features described above into a single cohesive application. Below, we walk through a comprehensive example that assembles multiple commands, handles aliases, incorporates asynchronous tasks, provides detailed help and versioning information, and gracefully handles unknown inputs.

Imagine we are developing a CLI tool to manage a simple task list. The CLI supports commands for adding a task, listing tasks, removing a task, and updating a task's status. For demonstration purposes, we will simulate these tasks rather than connecting to a real database. We will also support an alias for the "list" command and implement both synchronous and asynchronous operations.

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { of, timer } from 'rxjs';
import { delay, tap, switchMap } from 'rxjs/operators';

// Let's simulate an in-memory task list
interface Task {
  id: number;
  description: string;
  completed: boolean;
}

let tasks: Task[] = [];

// Instantiate Smartcli
const cli = new Smartcli();

// Command: add - to add a new task to the task list
const addCommand = cli.addCommand('add');
addCommand.subscribe((argv: any) => {
```

```

const taskDesc = argv.description || 'Untitled task';
const newTask: Task = { id: tasks.length + 1, description: taskDesc, completed: false };
tasks.push(newTask);
console.log(`Task added: [${newTask.id}] ${newTask.description}`);
});

// Command: list - list all tasks
const listCommand = cli.addCommand('list');
listCommand.subscribe((argv: any) => {
  if (tasks.length === 0) {
    console.log('No tasks available. ');
    return;
  }
  console.log('Task List:');
  tasks.forEach((task) => {
    console.log(`- [${task.id}] ${task.description} ${task.completed ? '(completed)' : ''}`);
  });
});

// Add an alias for the list command
cli.addCommandAlias('list', 'ls');

// Command: remove - delete a task by its ID
const removeCommand = cli.addCommand('remove');
removeCommand.subscribe((argv: any) => {
  const taskId = Number(argv.id);
  if (!taskId) {
    console.log('Please provide a valid task id using --id flag. ');
    return;
  }
  const taskIndex = tasks.findIndex(task => task.id === taskId);
  if (taskIndex === -1) {
    console.log(`Task with id [${taskId}] not found.`);
    return;
  }
  const removedTask = tasks.splice(taskIndex, 1)[0];
  console.log(`Task removed: [${removedTask.id}] ${removedTask.description}`);
});

// Command: complete - mark a task as completed

```

```

const completeCommand = cli.addCommand('complete');
completeCommand.subscribe((argv: any) => {
  const taskId = Number(argv.id);
  if (!taskId) {
    console.log('Please provide a valid task id using --id flag.');
```

```

    return;
  }
  const task = tasks.find(task => task.id === taskId);
  if (!task) {
    console.log(`Task with id [${taskId}] not found.`);
    return;
  }
  // Simulate an asynchronous operation (e.g., updating a remote server) with RxJS timer
  timer(1000)
    .pipe(
      tap(() => {
        task.completed = true;
        console.log(`Task [${task.id}] marked as completed.`);
      })
    )
    .subscribe();
});

// Command: update - update an existing task's description
const updateCommand = cli.addCommand('update');
updateCommand.subscribe((argv: any) => {
  const taskId = Number(argv.id);
  const newDesc = argv.description;
  if (!taskId || !newDesc) {
    console.log('Usage: update --id [taskId] --description "New Description"');
    return;
  }
  const task = tasks.find(task => task.id === taskId);
  if (!task) {
    console.log(`Task with id [${taskId}] not found.`);
    return;
  }
  // Simulate a delay to represent asynchronous update (like saving to a database)
  of(null)
    .pipe(
```

```

    delay(500),
    tap(() => {
        task.description = newDesc;
        console.log(`Task [${task.id}] updated to: ${task.description}`);
    })
)
.subscribe();
});

// Standard command to handle cases where no specific command is provided
cli.standardCommand().subscribe((argv: any) => {
    console.log('No command provided. Available commands: add, list (ls), remove, complete,
update.');
```

console.log('For help, try: --help');

```

});

// Add version information to support -v or --version flags
cli.addVersion('2.0.0');
```

// Add help text to provide an overview of available commands and options

```

cli.addHelp({
    helpText: `
Task Manager CLI Help:
Commands:
    add      : Add a new task. Use --description "Your task here" to set the task details.
    list (ls) : List all tasks.
    remove   : Remove a task by id. Use --id [taskID] to specify which task to remove.
    complete : Mark a task as completed. Use --id [taskID] to specify which task.
    update   : Update a task's description. Use --id [taskID] and --description "New
description".
Global Options:
    --version, -v: Show the CLI version.
    --help      : Display this help information.
`,
});

// Start parsing command-line input
cli.startParse();
```

This layered example demonstrates multiple aspects:

- Each command is defined with its distinct business logic.
- Asynchronous operations are incorporated using RxJS operators such as timer, delay, and of to simulate delays.
- Aliases (such as ls for list) enhance usability.
- The help function provides a descriptive guide for end users.
- Standard commands act as fallbacks, ensuring that users always receive informative feedback.

7. Developing an Interactive CLI with Error Handling and Feedback

In many practical applications, commands might reject invalid input or encounter errors during asynchronous operations. smartcli allows you to integrate error handling as part of your observable pipelines. This design is especially beneficial when real-world issues occur (e.g., network errors, missing arguments, or unrecognized commands).

Consider an extension of the “fetch-data” command where errors are handled meticulously. In the following example, we simulate fetching external data with built-in error handling:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { of, throwError } from 'rxjs';
import { delay, catchError, tap, switchMap } from 'rxjs/operators';

const cli = new Smartcli();

const fetchDataCommand = cli.addCommand('fetch-data');
fetchDataCommand.subscribe((argv: any) => {
  console.log('Initiating data fetch...');

  // Simulate a network fetch which may fail
  const simulateNetworkCall = Math.random() > 0.5
    ? of({ data: 'Fetched data successfully!' }).pipe(delay(1500))
    : throwError(() => new Error('Network Error: Unable to fetch data')).pipe(delay(1500));

  simulateNetworkCall
    .pipe(
      tap((response: any) => {
        console.log('Data received:', response.data);
      }),
    ),
});
```

```

switchMap((response: any) => {
  // Continue processing the data after successful fetch
  return of(`Data processed: ${response.data.toUpperCase()}`);
}),
tap((processedData: string) => {
  console.log(processedData);
}),
catchError((error: Error) => {
  console.error('Encountered an error during fetch:', error.message);
  // Optionally, provide recovery or fallback logic here
  return of('Default data after error handling');
})
)
.subscribe({
  next: (result: any) => {
    console.log('Final result:', result);
  },
  error: (err: Error) => {
    console.error('Error in subscription:', err.message);
  },
});
});

// Define a standard fallback command in case no specific command is provided
cli.standardCommand().subscribe(() => {
  console.log('No command provided. For fetching data, use: fetch-data');
});

// Start parsing the command
cli.startParse();

```

In this example, we simulate a network call that may either succeed or fail. Notice:

- We use `throwError` to simulate a network error.
- The `catchError` operator intercepts errors, logs the error message, and allows us to provide fallback data.
- This example demonstrates reactive error handling in `smartcli`, ensuring that your CLI tool remains robust even in adverse conditions.

8. Advanced Integration: Combining Observables, Configuration Files, and Environment Variables

Many CLI applications require dynamically reading configurations from files or based on environment variables. Combining smartcli's reactive capabilities with Node's file system and environment handling can create truly adaptive CLI applications.

Let's build an example where a command "configure" reads a configuration file, uses environment variables, and then applies those settings to the CLI's operation. In this scenario, we assume that the configuration is stored in a JSON file. The command will read the file asynchronously, merge it with any options passed via the CLI, and then print out the resultant configuration.

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { promises as fs } from 'fs';
import path from 'path';
import { from } from 'rxjs';
import { switchMap, tap, catchError } from 'rxjs/operators';

const cli = new Smartcli();

// Command: configure
const configureCommand = cli.addCommand('configure');
configureCommand.subscribe((argv: any) => {
  // Determine the configuration file path (default or provided via --config flag)
  const configFile = argv.config || path.join(process.cwd(), 'config.json');

  console.log(`Attempting to load configuration from: ${configFile}`);

  // Read the configuration file asynchronously using fs.promises
  from(fs.readFile(configFile, 'utf-8'))
    .pipe(
      switchMap((fileContent: string) => {
        // Parse the JSON content of the configuration file
        const fileConfig = JSON.parse(fileContent);
        console.log('Configuration loaded from file:', fileConfig);
        // Merge file configurations with additional options from command line, such as
```

overriding values via --option flags

```
    const finalConfig = { ...fileConfig, ...argv };
    return from(Promise.resolve(finalConfig));
  }),
  tap((finalConfig) => {
    console.log('Final merged configuration:', finalConfig);
    // Here you might dynamically adjust your CLI behavior based on finalConfig values
  }),
  catchError((err: Error) => {
    console.error('Error loading configuration file:', err.message);
    return from(Promise.resolve({ error: 'default configuration activated' }));
  })
)
.subscribe((result) => {
  console.log('Configuration process completed with result:', result);
});

// Standard fallback command
cli.standardCommand().subscribe(() => {
  console.log('No command specified. Please use one of the available commands (e.g. configure,
add, list, remove, complete, update).');
});

// Start parsing commands
cli.startParse();
```

In this use case:

- We leverage Node's `fs.promises` to read a configuration file asynchronously.
- The configuration file is merged with runtime arguments provided by the user.
- We employ RxJS operators to manage asynchronous operations and error handling.
- The final configuration is then outputted or used to alter further application behavior.

9. Testing Your CLI Application

For robust CLI applications, automated tests are essential. `smartcli`'s design promotes testability by exposing its observable commands directly. You can simulate command inputs and verify that the respective command callbacks are executed correctly.

Below is an example test scenario using a testing framework (such as tap or jest) to verify that commands are being correctly registered and executed:

```
// test/cli.test.ts
import { Smartcli } from '@push.rocks/smartcli';
import { Subject } from '@push.rocks/smartrx';

// Test to verify that a command executes as expected
const cli = new Smartcli();
let commandExecuted = false;

// Register a test command "testcmd"
const testCommand = cli.addCommand('testcmd');
testCommand.subscribe((argv: any) => {
  commandExecuted = true;
  console.log('Test command executed with:', argv);
});

// Simulate passing the command as if from process.argv
// For testing, you can modify process.argv temporarily
process.argv.splice(2, 0, 'testcmd', '--sample', 'value');

// Start parsing and executing the test command
cli.startParse();

// After parsing, verify if commandExecuted is true
setTimeout(() => {
  if (commandExecuted) {
    console.log('Test passed: command was executed.');
```

This test demonstrates:

- Creating an instance of smartcli.
- Adding a command and subscribing to its observable output.
- Simulating command-line input by adjusting process.argv.
- Validating that the command logic executes as expected.

10. Best Practices and Recommendations

Now that we have explored various usage scenarios of smartcli, here are some best practices and recommendations to help you maximize the benefits of this library in your CLI development:

1. Modularize Your Commands

Organize your CLI application by separating commands into different modules or files. This aids maintainability and scalability, especially when dealing with a large number of functions.

2. Utilize Observables for Async Processing

Embrace the power of observables to manage asynchronous operations in a declarative manner. Utilize RxJS operators like `switchMap`, `delay`, and `catchError` to handle asynchronous workflows and potential errors.

3. Consistent Command Naming and Alias Management

Design a consistent naming scheme for your commands and define intuitive aliases to improve the overall user experience. This can be especially useful when commands are likely to be used repeatedly.

4. Offer Comprehensive Help and Feedback

Use the `addHelp` method to provide detailed usage instructions. A well-documented CLI helps end users understand available commands, options, and expected behaviors. Always consider fallback or default commands to guide users when unknown commands are entered.

5. Include Version Information

The `addVersion` method is a convenient way to ensure that users can quickly check which version of your CLI they are using, facilitating troubleshooting and support.

6. Integrate Error Handling Throughout

In a production CLI application, robust error handling can make the difference between a good user experience and a frustrating one. Incorporate error handling in observable pipelines to handle network failures or invalid user input gracefully.

7. Leverage Environment Variables and Configuration Files

For flexible CLI behavior, consider dynamically loading configuration from files or environment variables. This improves the usability of your CLI tool in different deployment environments.

8. Test Thoroughly

Since smartcli decouples command registration and execution, automated testing becomes straightforward. Write unit tests to simulate various command inputs and verify that your commands respond correctly. This helps to catch regressions and ensures a high-quality tool.

11. Integrating with Other Reactive Libraries and Tools

smartcli's design encourages integration with other reactive and asynchronous libraries. You can blend smartcli with external modules to create advanced workflows, such as chaining CLI commands with external APIs, handling streams of data, or integrating real-time logging utilities.

For instance, if your CLI needs to process a stream of events from a WebSocket or external feed, you can subscribe to those events within a command callback and process the data reactively. Here's an abstract example of integrating a WebSocket stream within a CLI command:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';
import { websocket } from 'rxjs/webSocket';
import { tap, catchError } from 'rxjs/operators';

const cli = new Smartcli();

const streamCommand = cli.addCommand('stream-data');
streamCommand.subscribe((argv: any) => {
  const socketUrl = argv.url || 'wss://example.com/socket';
  console.log(`Connecting to ${socketUrl}...`);

  // Create a WebSocket subject that acts as an observable stream
  const socketSubject = websocket(socketUrl);

  socketSubject.pipe(
    tap((message: any) => console.log('Message received:', message)),
    catchError((err: any) => {
      console.error('Error in WebSocket stream:', err);
      throw err;
    })
  ).subscribe({
    next: (msg: any) => {
      // process messages as they arrive
    },
    error: (err: any) => {
      console.error('WebSocket error:', err);
    },
  },
```

```
    complete: () => {
      console.log('WebSocket connection closed.');
```

```
    }
  });
});

// Fall back standard command if no argument provided
cli.standardCommand().subscribe(() => {
  console.log('No command provided. To stream data, use: stream-data --url [WS_URL]');
```

```
});

// Start the CLI argument parsing
cli.startParse();
```

This example is an outline of how you might integrate real-time data streams with smartcli to build a highly interactive command-line tool. By combining reactive streams with smartcli's command management, you can create advanced workflows and event-driven CLI applications.

12. Debugging and Logging

Debugging command-line applications can sometimes be challenging. smartcli integrates well with logging libraries, such as [@push.rocks/smartlog](#), to provide developers with useful output during command execution.

In your commands, consider logging key events, such as:

- When a command is received.
- The start and end of an asynchronous process.
- Errors or warnings when parsing options.
- Informational logs to indicate the progress of long-running tasks.

Below is a brief example illustrating logging within a command:

```
#!/usr/bin/env node
import { Smartcli } from '@push.rocks/smartcli';

const cli = new Smartcli();

const debugCommand = cli.addCommand('debug');
debugCommand.subscribe((argv: any) => {
  console.log('Debug command initiated...');
  console.log('Parsed arguments:', argv);
```

```
// Simulate an asynchronous task with logging
setTimeout(() => {
  console.log('Debug task completed successfully.');
```

```
  }, 1000);
});

cli.standardCommand().subscribe(() => {
  console.log('For debugging, use: debug with appropriate flags.');
```

```
});

cli.startParse();
```

Using proper logging throughout your CLI can significantly ease troubleshooting and enable you to better understand the flow of execution.

13. Comprehensive Walkthrough and Further Customization

Let's take a step-by-step walkthrough of a final, comprehensive example. In this example, we will create a CLI tool that not only supports multiple commands and asynchronous operations but also demonstrates how to work with configuration files, environment variables, reactive streams, debugging logs, and error handling. This walkthrough is designed to illustrate a real-world application where smartcli is brought to its fullest potential.

1. Initialization and Setup

Create a file called cli.ts and include the following content at its top:

```
#!/usr/bin/env node

import { Smartcli } from '@push.rocks/smartcli';
import { promises as fs } from 'fs';
import path from 'path';
import { from, timer } from 'rxjs';
import { delay, tap, switchMap, catchError } from 'rxjs/operators';

const cli = new Smartcli();

// Set global version and help text
```

```

cli.addVersion('3.0.0');
cli.addHelp({
  helpText: `
Comprehensive Task CLI:
  • add      : Add a new task. Use --description to set details.
  • list (ls) : List tasks.
  • remove   : Remove a task with --id.
  • complete : Complete a task with --id.
  • config   : Load configuration from a JSON file.
  • debug    : Run in debug mode.
`,
});

```

2. Adding Commands

Split your command definitions into separate sections for clarity. For every command, register the command, subscribe to its observable, and implement your business logic:

- For delayed operations and asynchronous processing, use RxJS observables with proper delays and error handling.
- For configuration loading, use a combination of fs.promises and observables to merge file-based settings and command-line arguments.

Example for the configuration command:

```

const configCommand = cli.addCommand('config');
configCommand.subscribe((argv: any) => {
  const configFilePath = argv.config || path.join(process.cwd(), 'config.json');
  console.log(`Loading configuration from ${configFilePath}`);

  from(fs.readFile(configFilePath, 'utf-8'))
    .pipe(
      switchMap((content: string) => {
        const parsedConfig = JSON.parse(content);
        console.log('Configuration loaded:', parsedConfig);
        return from(Promise.resolve({ ...parsedConfig, ...argv }));
      }),
      tap((finalConfig) => {
        console.log('Merged configuration:', finalConfig);
      }),
      catchError((err: Error) => {
        console.error('Error loading configuration:', err.message);
        return from(Promise.resolve({ error: true }));
      })
    )
});

```

```
.subscribe((result: any) => {
  console.log('Configuration process complete.', result);
});
});
```

3. Handling Debug Mode and Logging

Create a debug command that logs extensive details about the input:

```
const debugCommand = cli.addCommand('debug');
debugCommand.subscribe((argv: any) => {
  console.log('Debug mode activated.');
```

```
  console.log('Current process arguments:', process.argv);
  console.log('Parsed options:', argv);
  // Insert additional debugging routines as necessary
  timer(500)
    .pipe(
      tap(() => {
        console.log('Debug tasks executed.');
```

```
      })
    )
  .subscribe();
});
```

4. Finalizing with a Standard Command

Always have a fallback standard command that provides instructions if no valid command is given:

```
cli.standardCommand().subscribe(() => {
  console.log('No command provided. Please use one of the available commands: add,
list, remove, complete, config, debug.');
```

```
});
```

5. Starting the CLI Parser

After all command registrations, start the parser which will process the arguments and trigger the appropriate command:

```
cli.startParse();
```

6. Running Your CLI

Make sure your file has proper executable permissions (e.g., `chmod +x cli.ts`) and is referenced in your `package.json`'s `bin` field if you plan on publishing it as a global CLI tool.

This complete walkthrough shows how you can use `smartcli` to build a full-featured, robust CLI application. By dividing functionality into separate commands, handling asynchronous operations

gracefully, and providing detailed help and debugging support, you ensure that your CLI application is both powerful and user-friendly.

14. Summary of Features and Next Steps for Your Project

The `@push.rocks/smartcli` library is a versatile solution for constructing command-line interfaces. Its core features include:

- Defining and subscribing to reactive commands.
- Seamless integration with RxJS for asynchronous and reactive operations.
- Support for command aliases, allowing multiple inputs to trigger the same functionality.
- A simple API to manage command-line options with sophisticated error handling.
- Built-in methods for adding version, help text, and fallback commands.

By exploring and combining these features, you can build robust, interactive, and user-friendly CLI tools that adapt to your application's unique requirements. Whether it's a simple utility or a complex task manager, `smartcli` offers the flexibility and power to develop modern command-line applications using TypeScript and ESM syntax.

15. Additional Considerations and Practical Tips

As you continue developing your CLI application with `smartcli`, consider these additional tips:

- Always validate command-line options to ensure your commands receive the expected input.
- Use descriptive log messages not only for debugging but also to provide users with clear feedback.
- Avoid coupling your command logic with side effects; design your commands as pure functions where possible, managing side effects through subscription callbacks.
- Explore further RxJS operators and patterns to enhance your observable pipelines. The more familiar you become with reactive programming, the more elegant your CLI code will be.
- Consider integrating configuration management libraries or even state management systems if your CLI tool grows in complexity.
- Write comprehensive integration tests that mimic real-world usage scenarios, ensuring that your CLI behaves correctly regardless of the input provided.
- Keep your project modular by refactoring commands into separate files or modules, which not only improves readability but also simplifies maintenance and future enhancements.

By adhering to these recommendations, you can create a CLI tool that is reliable, maintainable, and enjoyable to use.

16. Final Thoughts on smartcli Usage

The extensive usage examples above have demonstrated how smartcli transforms the process of building CLI applications. By unifying command parsing, observable-based asynchronous handling, and modular command definitions within a single, cohesive framework, smartcli becomes an indispensable tool for developers.

Start by modifying the basic examples, then combine more features as your project requirements expand. With its intuitive API and extensive customization capabilities, smartcli enables you to develop professional-grade CLI tools that serve a wide range of use cases—from simple automation scripts to complex interactive applications.

We encourage you to experiment with the various functions outlined in this document. Explore different command structures, integrate external APIs, and test robust error-handling scenarios. The reactive approach provided by smartcli opens up many possibilities to build scalable, responsive, and user-friendly CLI interfaces.

Happy coding with smartcli, and may your command-line applications be as smart and dynamic as the library itself!

This exhaustive guide and the corresponding examples provide you with all the tools and insights you need to leverage @push.rocks/smartcli in your projects. Enjoy developing your next observable CLI task application using TypeScript and ESM syntax!

License and Legal Information

This repository contains open-source code that is licensed under the MIT License. A copy of the MIT License can be found in the [license](#) file within this repository.

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.

docs/index.md for @push.rocks/smartcli

nodejs wrapper for CLI related tasks

Availabililty

[npm](#) [git](#) [git docs](#)

Status for master

[build status](#) [coverage report](#) [npm downloads per month](#) [Dependency Status](#)
[bitHound Dependencies](#) [bitHound Code](#) [TypeScript](#) [node](#) [JavaScript](#) [Style Guide](#)

Usage

For further information read the linked docs at the top of this README.

“ MIT licensed | © [Lossless GmbH](#) | By using this npm module you agree to our [privacy policy](#)

[repo-footer](#)

changelog.md for @push.rocks/smartcli

2025-10-28 - 4.0.19 - fix(license)

Update license files and add local tool settings

- Update LICENSE header to reference Task Venture Capital GmbH as copyright holder
- Add a new license file containing the full MIT license text
- Add .claude/settings.local.json to store local tool permission settings

2025-10-28 - 4.0.18 - fix(smartcli)

Allow passing argv to startParse and improve getUserArgs Deno/runtime handling; update tests and add license

- Smartcli.startParse now accepts an optional testArgv parameter to bypass automatic runtime detection (makes testing deterministic).
- getUserArgs logic refined: always prefer Deno.args when available (handles Deno run and compiled executables reliably) and improve execPath fallback and slicing behavior for Node/Bun/other launchers.
- Tests updated: test/test.node+deno+bun.ts now passes process.argv explicitly to startParse to avoid Deno.args interference in test environments.
- Added MIT LICENSE file and a local .claude/settings.local.json for environment/permission settings.

2025-10-28 - 4.0.17 - fix(license)

Add MIT license and local Claude settings

- Add LICENSE file (MIT) to repository
- Add .claude/settings.local.json with local permissions for tooling

2025-10-28 - 4.0.16 - fix(smartcli.helpers)

Improve CLI argument parsing and Deno runtime detection; use getUserArgs consistently

- Enhance getUserArgs() to prefer Deno.args but detect when process.argv was manipulated (e.g. in tests) and fallback to manual parsing
- Add robust handling of process.execPath / execPath basename and compute correct argv offset for known launchers vs. compiled executables
- Call getUserArgs() (no explicit process.argv) from Smartcli.getOption and Smartcli.startParse to ensure consistent cross-runtime behavior
- Expand readme.hints.md with detailed cross-runtime examples and explanation of Deno.args vs process.argv for compiled executables
- Add local claude settings file for tooling configuration

2025-10-28 - 4.0.15 - fix(smartcli.helpers)

Add robust getUserArgs helper and refactor Smartcli to use it; add MIT license and update documentation

- Add ts/smartcli.helpers.ts: getUserArgs to normalize user arguments across Node.js, Deno (run/compiled), and Bun, with safety checks for test environments
- Refactor Smartcli (ts/smartcli.classes.smartcli.ts) to use getUserArgs in startParse and getOption for correct argument parsing and improved test compatibility
- Update readme.hints.md with detailed cross-runtime CLI argument parsing guidance
- Add LICENSE (MIT) file
- Add .claude/settings.local.json (local settings)

2025-10-28 - 4.0.14 - fix(license)

Add MIT license file

- Add MIT License file to repository to clarify licensing and copyright (Push.Rocks 2015).

2025-10-27 - 4.0.13 - fix(smartcli)

Improve CLI argument parsing, update deps and tests

- Enhance startParse() to filter runtime executables and script paths (node, deno, bun, tsx, ts-node) so commands are detected correctly across runtimes.
- Switch path import to node:path in plugins for ESM compatibility.
- Bump various dependencies and devDependencies (including @push.rocks/lik, @push.rocks/smartlog, @push.rocks/smartpromise, @push.rocks/smartrx, yargs-parser, @git.zone tooling) and add packageManager field.
- Replace / reorganize tests: add/modify test/test.node+deno+bun.ts, adjust test script to use --verbose.
- Add deno.lock and include many resolved npm dependencies.
- Add LICENSE file (MIT).

2025-04-01 - 4.0.12 - fix(docs)

Update documentation with comprehensive usage examples, improved command alias descriptions, and detailed configuration instructions

- Revised readme.md with in-depth examples covering multiple CLI scenarios and RxJS integration
- Updated package.json and npmextra.json descriptions and keywords to reflect enhanced functionality
- Expanded usage guide with additional commands, error handling strategies, and testing guidelines

2024-05-29 - 4.0.11 - general

update description

- Updated the project description

2024-05-28 - 4.0.10 - core

fix(core): update

- Fixed core functionality

2024-04-13 - 4.0.9 - core

fix(core): update

- Improved core update handling

2024-04-12 - 4.0.8 - core / npmextra

- fix(core): update
- update npmextra.json: githost (this change was applied multiple times)

2023-08-19 - 4.0.7 - core

fix(core): update

- Fixed core issues

2023-07-12 - 4.0.6 - core / org

- fix(core): update
- switch to new org scheme (applied twice)

2022-08-07 - 4.0.5 - core

fix(core): update

- Fixed core functionality

2022-08-04 - 4.0.4 - core

fix(core): update

- Improved core update

2022-08-03 - 4.0.3 - core

fix(core): update

- Fixed core issues

2022-08-03 - 4.0.2 - core

fix(core): update

- Updated core handling

2022-08-03 - 4.0.1 - core

fix(core): update

- Fixed core functionality

2022-08-03 - 4.0.0 - core

fix(core): update

- Improved core update handling

2022-08-03 - 3.0.14 - core

BREAKING CHANGE(core): switch to esm

- Switched the project to use ECMAScript modules

2021-04-07 - 3.0.13 - core

fix(core): update

- Fixed core update issues

2021-04-07 - 3.0.12 - core

fix(core): update

- Updated core functionality

2020-05-29 - 3.0.11 - core

fix(core): update

- Fixed core functionality

2020-04-13 - 3.0.10 - core

fix(core): more consistent handling of process.enc.CLI_CALL

- Made process.enc.CLI_CALL handling more consistent

2020-04-13 - 3.0.9 - core

fix(core): now works better with tapbundle tests

- Improved compatibility with tapbundle tests

2020-03-11 - 3.0.8 - core

fix(core): update

- Updated core functionality

2020-03-11 - 3.0.7 - core

fix(core): update

- Fixed core update issues

2018-12-11 - 3.0.6 - core

fix(core): update

- Updated core functionality

2018-09-30 - 3.0.5 - ci

fix(ci): remove obsolete dependencies

- Removed obsolete dependencies from CI configuration

2018-09-30 - 3.0.4 - core

fix(core): update

- Updated core functionality

2018-08-30 - 3.0.3 - structure

fix(structure): remove dist/ dir from git repo

- Removed the dist/ directory from the repository

2018-08-30 - 3.0.2 - dependencies

fix(dependencies): update to latest versions

- Bumped dependency versions to the latest

2018-06-28 - 3.0.1 - core

fix(core): slim down dependencies

- Slimmed down core dependencies

2018-05-03 - 3.0.0 - general

update

- General update

2018-05-03 - 2.0.12 - core / architecture

- change to an all rxjs Subject architecture
- system change
- fix(core): cleanup
- remove package-lock since using yarn

2018-01-27 - 2.0.11 - security

fix(improve security CI step):

- Improved security in the CI step

2018-01-27 - 2.0.10 - core / ci

- fix(core): remove vulnerable paths
- update ci (applied multiple times)

2018-01-27 - 2.0.09 - CI

add security step to CI

- Added an extra security step in the CI process

2017-10-12 - 2.0.08 - compatibility

ensure compatibility with code assertion library

- Ensured compatibility with the code assertion library

2017-10-12 - 2.0.07 - tests / cli

- fix tests and add `.triggerOnlyOnProcessEnvCliCall()`
- fix linting issues

2017-05-07 - 2.0.06 - tasks

fix promise rejection on standard task

- Fixed a promise rejection issue on standard tasks

2017-04-23 - 2.0.05 - tapbundle

use new tapbundle

- Switched to the new tapbundle for testing

2017-04-22 - 2.0.04 - tests

- comment out one test that makes problems due to tap
- update tests

2017-04-22 - 2.0.03 - npmextra

add npmextra.json

- Added npmextra.json for extra configuration

2017-04-22 - 2.0.02 - ci

update ci

- Updated CI configuration

2017-04-22 - 2.0.01 - misc

- update .gitignore
- update to latest standards

2016-12-18 - 2.0.00 - core

fix argvArg for observables

- Fixed the argvArg handling for observables

2016-12-18 - 1.0.16 - triggers

introduce triggers

- Introduced triggers

2016-11-19 - 1.0.15 - triggers / docs

- added .triggerCommandByName
- improve README

2016-11-19 - 1.0.14 - metadata

Update Metadata

- Updated project metadata

2016-11-19 - 1.0.13 - docs

improve README

- Improved the README documentation

2016-11-19 - 1.0.11 - core / tests / docs

- cleanup
- improve README
- update test file

2016-10-14 - 1.0.10 - deps

update deps

- Updated dependencies

2016-10-14 - 1.0.09 - standardJS

implement standardJS

- Implemented standardJS support

2016-09-04 - 1.0.08 - typings

improve typings

- Improved TypeScript typings

2016-09-04 - 1.0.07 - ci

fix ci

- Fixed CI configuration

2016-09-04 - 1.0.06 - base

fix base image

- Fixed the base image used for builds

2016-09-04 - 1.0.05 - interaction

- add page stage
- improve typings and docs
- update smartcli
- Add new file
- start interaction module

2016-08-26 - 1.0.04 - intellisense

improve intellisense

- Improved editor intellisense

2016-06-22 - 1.0.03 - compile

compile fix

- Fixed compilation issues

2016-06-22 - 1.0.02 - updates

- fix
- add getCommandPromise
- update deps and transition from npmts to npmts-g

2016-06-16 - 1.0.01 - tasks

- standard tasks now returns argv
- some cosmetics
- introduce new classes

2016-06-10 - 1.0.00 - version

- fix version return
- return argv to command

2016-06-10 - 0.0.13 - smartcli

- first version with basic functionality
- remove bulk and add some features to Smartcli class
- start restructuring to use a smarter Smartcli class that handles command evaluation for you
- update dependencies

- compile
- add gitlab ci
- start smartcli class
- add class smartcli
- Update README and include commander
- fixed type issue
- fixed test issue
- update deps

2016-04-04 - 0.0.12 - deps

updated deps

- Updated dependencies

2016-04-04 - 0.0.11 - interface

- updated deps
- work in progress (noted twice)
- small interface fix

2015-11-09 - 0.0.10 - travis

improve travis process

- Improved the Travis process

2015-11-09 - 0.0.09 - tests / CLI

- add tests and fix some errors
- add Tests and improve TypeScript organization
- start smarter CLI logic
- fix small comment error (applied twice)

2015-10-14 - 0.0.08 - readme

- improved readme
- updated readme
- added devStatus badge

2015-10-12 - 0.0.07 - various

- improved return objects
- (Minor dependency updates and CI tweaks for beautylog and travis were also applied in this version)

2015-10-06 - 0.0.05 - CLI

- small update
- now handling CLI options

2015-10-05 - 0.0.4 - tests

modified test

2015-10-05 - 0.0.02 - travis / tests

- added travis + tests
- package.json update

2015-10-04 - 0.0.01 - initial

added initial structure

2015-10-04 - unknown - initial

Initial commit

Summary of Omitted Versions

The following versions contained no additional user-facing changes beyond version bumps and are summarized here: 1.0.12, 0.0.6, and 0.0.3.