# readme.md for @push.rocks/smartaction

a module for managing actions. What was done? What needs to be done? How often should it be done? What's currently being done?

## Install

To start using `@push.rocks/smartaction` in your project, you need to install it using npm or yarn. Run one of the following commands in your project root:

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

or if you prefer yarn:

```
yarn add @push.rocks/smartaction
```

## Usage

`@push.rocks/smartaction` is designed to help manage actions within your application. It can keep track of actions' states, frequency, and dependencies, making it easier to handle complex workflows. Below, we'll dive into how you can integrate `@push.rocks/smartaction` into your TypeScript projects.

## Setting Up

First, make sure you import the necessary classes from the package at the beginning of your TypeScript file.

```
import { Action, ActionStore } from '@push.rocks/smartaction';
```

# Creating Actions

Actions are the building blocks of your workflow management. Each action represents a task or a series of tasks that need to be performed.

```
const myFirstAction = new Action({
  name: 'initialSetup',
  description: 'Performs the initial setup tasks',
});
```

In the example above, we create a simple action called `initialSetup`. The constructor takes an object where you can specify various properties of the action, such as its name and description.

# Managing Actions with ActionStore

While `Action` instances represent the individual tasks, `ActionStore` is where you manage them collectively. It allows you to add, retrieve, and execute actions based on your criteria.

```
import { ActionStore } from '@push.rocks/smartaction';

const myActionStore = new ActionStore();

// Adding an action to the store
myActionStore.addAction(myFirstAction);

// Executing an action by name
await myActionStore.executeAction('initialSetup');
```

`ActionStore` provides a range of methods to work with the actions, such as adding new actions to the store, executing them, and checking their status.

# Advanced Usage: Dependencies and Scheduling

`@push.rocks/smartaction` supports more sophisticated use cases, including action dependencies and scheduling.

For instance, if you have actions that should only run after certain other actions have completed, you can set up these dependencies explicitly.

```
const prepareDatabaseAction = new Action({
  name: 'prepareDatabase',
  description: 'Prepares the database for use',
});

const loadDataAction = new Action({
  name: 'loadData',
  description: 'Loads initial data into the database',
  dependencies: ['prepareDatabase'], // This action depends on the 'prepareDatabase' action
});

myActionStore.addAction(prepareDatabaseAction);
myActionStore.addAction(loadDataAction);

// When executed, `loadDataAction` will wait for `prepareDatabaseAction` to complete
await myActionStore.executeAction('loadData');
```

Scheduling actions allows you to control when actions should be executed, potentially on a recurring basis. While `@push.rocks/smartaction` provides a foundation for action management, scheduling might require integration with other libraries or your custom code, depending on your project's needs.

# Conclusion

`@push.rocks/smartaction` is a powerful module for managing actions within your applications. From simple task execution to handling complex dependencies and scheduling, it can help streamline your workflow management process. As demonstrated, integrating `@push.rocks/smartaction` into your TypeScript application is straightforward, and it can significantly enhance your project's capability to manage and execute actions efficiently.

Remember, the examples provided here are just starting points. Depending on your application's complexity and requirements, you might find yourself extending classes or contributing new features to handle your specific use cases better.

For more information and advanced configurations, please refer to the official documentation and explore the TypeScript declaration files to understand all available options and methods. Happy coding!

# 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.

---

Revision #3
Created 2026-03-28 11:09:58 UTC by foss.global Team
Updated 2026-03-28 12:16:36 UTC by foss.global Team