@push.rocks/smartfile-interfaces

Provides TypeScript interfaces for interacting with @pushrocks/smartfile, focusing on virtual directory objects.

readme.md for @push.rocks/smartfile-interfaces

an interface package for @pushrocks/smartfile

Install

To install `@push.rocks/smartfile-interfaces`, use npm:

```
npm install @push.rocks/smartfile-interfaces --save
```

This command will add `@push.rocks/smartfile-interfaces` as a dependency to your project and download it to the `node_modules` directory.

Usage

Using `@push.rocks/smartfile-interfaces` in your TypeScript project is straightforward. This package provides TypeScript interfaces to enhance type checking and autocompletion when working with file-related operations, especially in projects that also use `@pushrocks/smartfile`.

First, ensure your TypeScript project is set up properly to use ESM syntax. You might need to include `"type": "module"` in your `package.json` or use the `.mjs` extension for your module files.

Basic Example

Once you've included `@push.rocks/smartfile-interfaces` in your project, you can start using the interfaces to define types for file operations. For instance, if you're working with an object representing a virtual directory structure to be passed around in your application, you might want to use the `VirtualDirTransferableObject` interface for type safety.

```
import type { VirtualDirTransferableObject } from '@push.rocks/smartfile-interfaces';

// Example of a virtual directory object implementing the interface
const virtualDir: VirtualDirTransferableObject = {
  files: [
    'path/to/your/file1.txt',
    'path/to/your/file2.jpg',
    // Add more file paths as needed
  ],
};

console.log(virtualDir);
```

This simple code sample demonstrates how to use the `VirtualDirTransferableObject` interface to ensure your virtual directory objects conform to a predetermined structure, providing clearer code documentation and enhanced type safety.

Advanced Usage

In a more complex application, interfaces from `@push.rocks/smartfile-interfaces` can be used in conjunction with other types and interfaces to facilitate more sophisticated file handling operations. For example, you might be creating a function responsible for processing virtual directories, filtering out specific files, or performing other actions. Leveraging TypeScript's type system can significantly reduce bugs and development time.

```
import type { VirtualDirTransferableObject } from '@push.rocks/smartfile-interfaces';

function processVirtualDir(virtualDir: VirtualDirTransferableObject) {
  // Implement logic to process the virtual directory
  // For example, filtering out non-text files
  const textFiles = virtualDir.files.filter(file => file.endsWith('.txt'));

  return {
    ...virtualDir,
    files: textFiles,
  };
}

// Example usage
const initialVirtualDir: VirtualDirTransferableObject = {
```

```
  files: [
    'path/to/your/file1.txt',
    'path/to/your/important.pdf',
    'path/to/your/note.txt',
  ],
};

const processedDir = processVirtualDir(initialVirtualDir);
console.log('Processed virtual directory:', processedDir);
```

This advanced example showcases how to use the interface to create more reliable and maintainable code for file processing tasks in a TypeScript project.

Notes

Remember, when using TypeScript, leveraging interfaces not only provides type safety but also enhances code documentation and developer experience through better autocompletion and error checking.

As you integrate `@push.rocks/smartfile-interfaces` into larger projects, consider how these interfaces interact with other parts of your application and external libraries. The goal is to achieve a cohesive system where data structures and operations are well-defined and understood across your codebase.

For further information and more detailed documentation, you can refer to the official TypeScript documentation on modules and namespaces, as well as consult the `@pushrocks/smartfile` documentation for additional context on how these interfaces might be used in conjunction with smartfile operations.

License and Legal Information

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.