@push.rocks/smartstatus

A TypeScript library for managing HTTP status information, with detailed status classes.

- readme.md for @push.rocks/smartstatus
- changelog.md for @push.rocks/smartstatus

readme.md for @push.rocks/smartstatus

A TypeScript library for managing HTTP status information, with detailed status classes.

Install

To install `@push.rocks/smartstatus`, use npm (or yarn, or pnpm) by running the following command in your terminal:

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

Ensure you have TypeScript and a package to work with TypeScript in your project. If not, you might want to install TypeScript and ts-node (for development purposes) to your project:

```
npm install typescript ts-node --save-dev
```

Usage

The `@push.rocks/smartstatus` library provides a structured and comprehensive way to handle HTTP status codes in TypeScript, enhancing your ability to manage HTTP responses effectively. Below, we outline an extensive set of scenarios demonstrating how you can leverage this module's capabilities to streamline and enrich your response handling from server-side applications to API endpoints.

Getting Started

To start using `@push.rocks/smartstatus`, ensure you import the module into your TypeScript files. Here's a basic import statement you can use:

```
import * as smartstatus from '@push.rocks/smartstatus';
```

For more specific imports, such as obtaining particular HTTP status code classes, you may use:

```
import { HttpStatus, Status404, Status200 } from '@push.rocks/smartstatus';
```

Retrieving a Specific Status

The library allows you to retrieve any HTTP status code with its corresponding information quickly. You can do this by using either a class representation for known statuses or using a method that fetches the status by its status code string.

Example: Fetching 404 Not Found Status

Let's say you wish to work with the HTTP 404 "Not Found" status. You can retrieve its details as follows:

```
import { HttpStatus } from '@push.rocks/smartstatus';

const notFoundStatus = HttpStatus.getHttpStatusByString('404');
console.log(notFoundStatus.code); // 404
console.log(notFoundStatus.text); // Not Found
console.log(notFoundStatus.description); // The requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible.
```

The `getHttpStatusByString` function dynamically returns an instance of the respective status class with fields like HTTP code, textual representation, and an insightful description of the status.

Handling Errors Using HTTP Statuses

One of the most valuable features of `@push.rocks/smartstatus` is enriching error handling protocols in web applications or APIs. Let's illustrate how smartstatus can be used with an Express.js framework to enhance HTTP response management, specifically handling different HTTP error statuses.

Example: Implementing with Express.js

Imagine you are developing a web service, and a specific operation fails due to an authorization issue. Here's how you might use smartstatus to handle this error gracefully:

```
import express from 'express';
import { HttpStatus } from '@push.rocks/smartstatus';

const app = express();
```

```
app.get('/secure/data', (req, res) => {
  // Simulating an authorization failure condition
  const userIsAuthorized = false;

  if (!userIsAuthorized) {
    const unauthorizedStatus = HttpStatus.getHttpStatusByString('401');
    res.status(unauthorizedStatus.code).json({
      error: unauthorizedStatus.text,
      message: unauthorizedStatus.description,
    });
  } else {
    res.status(200).send('Secure Data');
  }
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

In this express app endpoint, if the user is not authorized to access the resource, it responds with a 401 Unauthorized status. This status code is enriched with a detailed message derived from smartstatus, providing a helpful explanation to API consumers.

Extending with Custom Statuses

While `@push.rocks/smartstatus` covers standard HTTP statuses, the library also allows you to define and manage custom status codes tailored to specific application needs. This is particularly useful for internal APIs requiring bespoke statuses.

Example: Creating a Custom HTTP Status

Let's say you want to create a unique status for indicating a special condition encountered by your application, such as a "Processing Error" that is represented by code 499. Define a custom status class as follows:

```
import { HttpStatus } from '@push.rocks/smartstatus';

class Status499 extends HttpStatus {
  constructor() {
    super({
      code: 499,
```

```
      text: 'Processing Error',
      description:
        'The server encountered a processing error which prevented it from completing the
request.',
    });
  }
}

// Register the custom status for use
HttpStatus.addStatus('499', Status499);

// Usage of the custom status
const processingErrorStatus = HttpStatus.getHttpStatusByString('499');
console.log(`${processingErrorStatus.code} ${processingErrorStatus.text}`); // 499 Processing
Error
```

This custom status can then be used in the same way as predefined statuses in the library, allowing you to handle unique responses efficiently.

Comprehensive Examples Across HTTP Status Ranges

The `smartstatus` library classifies statuses across different ranges (1xx, 2xx, 3xx, 4xx, and 5xx). Here, we will demonstrate how to handle each of these categories to provide robustness in response management.

Informational Responses (1xx)

Status codes in the 1xx range inform the client about the progress of the request. These are mostly used in scenarios involving continuation or switching protocols.

```
import { Status100 } from '@push.rocks/smartstatus';

// Handling 100 Continue status
const continueStatus = new Status100();
console.log(continueStatus.code, continueStatus.text, continueStatus.description);
```

Successful Responses (2xx)

The 2xx successes indicate that the client's request was received, understood, and accepted. Utilizing these codes effectively can aid in confirming successful API operations.

```
import { Status200, Status201 } from '@push.rocks/smartstatus';

// 200 OK status for successfully processed requests
const okStatus = new Status200();
console.log(okStatus.code, okStatus.text, okStatus.description);

// 201 Created status for resource creation
const createdStatus = new Status201();
console.log(createdStatus.code, createdStatus.text, createdStatus.description);
```

Redirection Messages (3xx)

Statuses in the 3xx range indicate redirection, offering guidance on how the client can access different resources or follow up with another request.

```
import { Status301, Status302 } from '@push.rocks/smartstatus';

// Permanent and Temporary redirects
const movedPermanentlyStatus = new Status301();
console.log(movedPermanentlyStatus.text); // Moved Permanently

const foundStatus = new Status302();
console.log(foundStatus.text); // Found
```

Client Error Responses (4xx)

This class of status codes is intended to inform the client that the error seems to have been caused by the client, such as a malformed request.

```
import { Status400, Status404 } from '@push.rocks/smartstatus';

// Using 400 Bad Request for malformed requests
const badRequestStatus = new Status400();
console.log(badRequestStatus.description);

// Handling 404 Not Found for missing resources
const notFoundStatus = new Status404();
console.log(notFoundStatus.description);
```

Server Error Responses (5xx)

5xx indicates server-side errors, signaling the server is aware it encountered an error or is otherwise incapable of performing the request.

```
import { Status500, Status503 } from '@push.rocks/smartstatus';

// Internal server error handling
const internalServerErrorStatus = new Status500();
console.log(internalServerErrorStatus.text);

// Service unavailable handling with a 503 response
const serviceUnavailableStatus = new Status503();
console.log(serviceUnavailableStatus.description);
```

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.

changelog.md for @push.rocks/smartstatus

2024-12-25 - 1.1.1 - fix(smartstatus)

Remove all tslint:disable-next-line comments for max-classes-per-file rule

- Code cleanup by removing unnecessary tslint comments for disabling max-classes-per-file rule.

2024-12-25 - 1.1.0 - feat(ci)

Set up GitHub Actions workflows for CI/CD

- Replaced GitLab CI with GitHub Actions for both tag and non-tag pushes.
- Implemented npm audit for security checks on both production and development dependencies.
- Enabled npm test and build steps as part of the CI workflow.

2024-05-29 - 1.0.12 - Documentation & Configuration

Minor updates focusing on project configuration and descriptive metadata.

- Updated project description.
- Updated tsconfig multiple times for improved TypeScript configurations.
- Adjusted npmextra.json configuration for host consistency.

2021-08-19 to 2024-04-14 - 1.0.11 to 1.0.12 - Core Fixes & Maintenance

A range of updates focused on core functionality and essential maintenance fixes across versions.

- Multiple core updates distributed across versions 1.0.7 to 1.0.12.

2019-01-02 to 2021-08-19 - 1.0.5 to 1.0.10 - Core Updates

Core functionality was reviewed and updated multiple times to ensure stability and improved performance.

2017-04-06 to 2019-01-02 - 1.0.0 to 1.0.4 - Initial Development & Feature Implementations

The initial stages of development including major feature implementations across status codes and basic framework setup.

- Initial setup completed with basic framework and configuration.
- Implemented handling of 1xx to 5xx range of status codes.
- Added continuous integration and initial testing protocols.