@push.rocks/smartguard

A library for creating and managing validation guards.

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

readme.md for @push.rocks/smartguard

A TypeScript library for creating and managing validation guards, aiding in data validation and security checks.

Install

To install `@push.rocks/smartguard`, run the following command in your terminal:

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

This will add `@push.rocks/smartguard` to your project's dependencies.

Usage

`@push.rocks/smartguard` provides a robust and easy way to validate data by using guards. Guards are functions that return a boolean value indicating whether the data meets certain criteria. This package is highly beneficial for input validation, security checks, or any scenario where data needs to conform to specific rules or patterns.

Basics

At the core of `@push.rocks/smartguard` are two main classes: `Guard` and `GuardSet`. A `Guard` represents a single rule or validation step, while a `GuardSet` allows you to combine multiple `Guard` instances and evaluate them together.

Creating a Guard

A `Guard` is an object that encapsulates a validation rule. You define a guard by providing a function that takes an input and returns a Promise, resolving to a boolean value indicating if the input meets the criteria.

```
import { Guard } from '@push.rocks/smartguard';

const isStringGuard = new Guard<string>(async (data) => {
  return typeof data === 'string';
});
```

In the example above, we define a simple guard that checks if the input is a string.

Using GuardSets for Composite Validations

When you have multiple validation rules, you can combine them using `GuardSet`. This allows you to evaluate all guards on a piece of data and only pass if all guards return true.

```
import { Guard, GuardSet } from '@push.rocks/smartguard';

const isStringGuard = new Guard<string>(async (data) => {
  return typeof data === 'string';
});

const isNotEmptyGuard = new Guard<string>(async (data) => {
  return data.length > 0;
});

const stringValidationSet = new GuardSet<string>([isStringGuard, isNotEmptyGuard]);

// Now you can use stringValidationSet.executeGuardsWithData(data) to validate your data
```

Executing Guards

To execute a guard or a set of guards against data, you use the `execGuardWithData` method for a single guard, or `execGuardsWithData` method for a `GuardSet`.

```
const isValidString = await isStringGuard.execGuardWithData('Hello World!');
console.log(isValidString); // true

const areValidStrings = await stringValidationSet.executeGuardsWithData('Hello World!');
```

```
console.log(areValidStrings.every(result => result)); // true if all validations passed
```

Advanced Usage: Custom Guard Functions

Guards can perform any asynchronous operation inside their validation function, making them incredibly versatile. For instance, you could call an API to validate an address, check if a username already exists in a database, or even integrate with third-party validation services.

```
import { Guard } from '@push.rocks/smartguard';
import { someApiRequestFunction } from './myApiFunctions';

const isValidAddressGuard = new Guard<string>(async (address) => {
  const response = await someApiRequestFunction(address);
  return response.isValid;
});
```

Integrating with Express Middleware

`@push.rocks/smartguard` can easily integrate with frameworks like Express by utilizing guards within middleware functions. This allows you to perform validations before a request reaches your route handlers.

```
import express from 'express';
import { Guard } from '@push.rocks/smartguard';

const app = express();
const isAuthorizedUserGuard = new Guard<express.Request>(async (req) => {
  // your logic here, return true if authorized
  return req.headers.authorization === 'Bearer some-token';
});

app.use(async (req, res, next) => {
  const isAuthorized = await isAuthorizedUserGuard.execGuardWithData(req);
  if (!isAuthorized) {
    res.status(403).send('Unauthorized');
    return;
  }
  next();
});
```

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

In the example above, we use a guard to check if a request has a valid authorization header. This demonstrates how `@push.rocks/smartguard` can be seamlessly integrated into existing server applications to enforce security or input validations.

Combining Guards with `GuardSet`

One of the strengths of `@push.rocks/smartguard` is its ability to combine multiple guards into a `GuardSet`. This is particularly useful when you need to validate data against several criteria. For example, to validate a string that must be non-empty and start with a specific prefix:

```
import { Guard, GuardSet } from '@push.rocks/smartguard';

const isStringGuard = new Guard<string>(async (data) => {
  return typeof data === 'string';
});

const isNotEmptyGuard = new Guard<string>(async (data) => {
  return data.length > 0;
});

const startsWithPrefixGuard = new Guard<string>(async (data) => {
  return data.startsWith('prefix');
});

const combinedValidationSet = new GuardSet<string>([isStringGuard, isNotEmptyGuard,
startsWithPrefixGuard]);

const validationResults = await combinedValidationSet.executeGuardsWithData('prefix: Valid
String');
console.log(validationResults.every(result => result)); // true if all validations passed
```

Integration with Other Libraries

To demonstrate the versatility and integration capabilities of `@push.rocks/smartguard`, let's integrate it with another popular library, `@push.rocks/smartrequest`, for validating API response data.

```
import { Guard } from '@push.rocks/smartguard';
import { smartrequest } from '@push.rocks/smartrequest';

const validApiResponseGuard = new Guard(async (url: string) => {
  const response = await smartrequest.request(url, { method: 'GET' });
  return response.status === 200;
});

const isValidResponse = await
validApiResponseGuard.execGuardWithData('https://example.com/api/data');
console.log(isValidResponse); // true if the API response status is 200
```

Real-World Example: Form Validation

Let's create a real-world example where we use `@push.rocks/smartguard` to validate form data in a Node.js application. Suppose we have a user registration form with fields for `username`, `email`, and `password`.

```
import { Guard, GuardSet } from '@push.rocks/smartguard';

// Guards for individual fields
const isUsernameValid = new Guard<string>(async (username) => {
  return typeof username === 'string' && username.length >= 3;
});

const isEmailValid = new Guard<string>(async (email) => {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return typeof email === 'string' && emailRegex.test(email);
});

const isPasswordStrong = new Guard<string>(async (password) => {
  return typeof password === 'string' && password.length >= 8;
});

// Combining guards using GuardSet
const registrationValidationSet = new GuardSet<{ username: string, email: string, password:
string }>([
  new Guard(async (data) => isUsernameValid.execGuardWithData(data.username)),
  new Guard(async (data) => isEmailValid.execGuardWithData(data.email)),
```

```
  new Guard(async (data) => isPasswordStrong.execGuardWithData(data.password))
]);

// Form data to validate
const formData = {
  username: 'exampleUser',
  email: 'user@example.com',
  password: 'strongpassword123'
};

const formValidationResults = await registrationValidationSet.executeGuardsWithData(formData);
console.log(formValidationResults.every(result => result)); // true if all fields are valid
```

In this example, we used guards to validate each form field. We then combined these guards into a `GuardSet` to validate the entire form data object.

Validating Nested Objects

`@push.rocks/smartguard` can also handle validation of nested objects. Suppose you need to validate a user profile that includes nested address information.

```
interface UserProfile {
  username: string;
  email: string;
  address: {
    street: string;
    city: string;
    postalCode: string;
  };
}

const isStreetValid = new Guard<string>(async (street) => {
  return typeof street === 'string' && street.length > 0;
});

const isCityValid = new Guard<string>(async (city) => {
  return typeof city === 'string' && city.length > 0;
});

const isPostalCodeValid = new Guard<string>(async (postalCode) => {
```

```
  return typeof postalCode === 'string' && /^[0-9]{5}$/.test(postalCode);
});

const isAddressValid = new Guard<UserProfile['address']>(async (address) => {
  const streetValid = await isStreetValid.execGuardWithData(address.street);
  const cityValid = await isCityValid.execGuardWithData(address.city);
  const postalCodeValid = await isPostalCodeValid.execGuardWithData(address.postalCode);
  return streetValid && cityValid && postalCodeValid;
});

const isUsernameValid = new Guard<string>(async (username) => {
  return typeof username === 'string' && username.length >= 3;
});

const isEmailValid = new Guard<string>(async (email) => {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return typeof email === 'string' && emailRegex.test(email);
});

const userProfileValidationSet = new GuardSet<UserProfile>([
  new Guard(async (data) => isUsernameValid.execGuardWithData(data.username)),
  new Guard(async (data) => isEmailValid.execGuardWithData(data.email)),
  new Guard(async (data) => isAddressValid.execGuardWithData(data.address))
]);

const userProfile = {
  username: 'exampleUser',
  email: 'user@example.com',
  address: {
    street: '123 Main St',
    city: 'Anytown',
    postalCode: '12345'
  }
};

const userProfileValidationResults = await
userProfileValidationSet.executeGuardsWithData(userProfile);
console.log(userProfileValidationResults.every(result => result)); // true if user profile is
valid
```

In this example, we created a nested guard structure to validate a user profile object that includes address information. Each nested object is validated individually using its specific guards.

Dynamic Guards

There can be situations when you need to create guards dynamically based on some conditions or input. `@push.rocks/smartguard` allows you to create and use such dynamic guards effortlessly.

```
import { Guard, GuardSet } from '@push.rocks/smartguard';

const createDynamicGuard = (minLength: number) => new Guard<string>(async (data) => {
  return data.length >= minLength;
});

const flexibleLengthGuardSet = (length: number) => new
GuardSet<string>([createDynamicGuard(length)]);

const dynamicGuard = flexibleLengthGuardSet(5);

const isValid = await dynamicGuard.executeGuardsWithData('Hello, world!');
console.log(isValid.every(result => result)); // true because the length of 'Hello, world!' is
more than 5
```

In the example above, we created a dynamic guard based on a minimum length and then evaluated some data against it.

Validating Complex Data Structures

There can often be a need to validate complex data structures with nested arrays or objects. Using a combination of `Guard` and `GuardSet`, you can effectively handle validations of such complex structures.

```
interface BlogPost {
  title: string;
  content: string;
  tags: string[];
  author: {
    name: string;
    email: string;
  };
```

```
}

const isStringGuard = new Guard<string>(async (data) => {
  return typeof data === 'string';
});

const isNonEmptyStringGuard = new Guard<string>(async (data) => {
  return await isStringGuard.execGuardWithData(data) && data.trim().length > 0;
});

const isStringArrayGuard = new Guard<string[]>(async (data) => {
  return Array.isArray(data) && data.every(item => typeof item === 'string');
});

const isEmailGuard = new Guard<string>(async (data) => {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return typeof data === 'string' && emailRegex.test(data);
});

const isAuthorGuardSet = new GuardSet<BlogPost['author']>([
  new Guard(async (data) => await isNonEmptyStringGuard.execGuardWithData(data.name)),
  new Guard(async (data) => await isEmailGuard.execGuardWithData(data.email))
]);

const isBlogPostGuardSet = new GuardSet<BlogPost>([
  new Guard(async (data) => await isNonEmptyStringGuard.execGuardWithData(data.title)),
  new Guard(async (data) => await isNonEmptyStringGuard.execGuardWithData(data.content)),
  new Guard(async (data) => await isStringArrayGuard.execGuardWithData(data.tags)),
  new Guard(async (data) => await
isAuthorGuardSet.executeGuardsWithData(data.author).then(results => results.every(result =>
result)))
]);

const blogPost: BlogPost = {
  title: 'Introduction to Smart Guard',
  content: 'Smart Guard is a TypeScript library for creating and managing validation
guards...',
  tags: ['typescript', 'validation', 'library'],
  author: {
    name: 'John Doe',
```

```
    email: 'johndoe@example.com'
  }
};

const blogPostValidationResults = await isBlogPostGuardSet.executeGuardsWithData(blogPost);
console.log(blogPostValidationResults.every(result => result)); // true if the blog post is
valid
```

In this example, we created different guards to validate various parts of a complex `BlogPost` object. Notice how we used nested `GuardSet` instances to validate the `author` object.

Asynchronous Validations

`@push.rocks/smartguard` supports asynchronous guard functions, making it possible to perform validations that involve network requests or other asynchronous operations.

```
import { Guard } from '@push.rocks/smartguard';
import { smartrequest } from '@push.rocks/smartrequest';

const isApiKeyValidGuard = new Guard<string>(async (apiKey) => {
  const response = await
smartrequest.request(`https://api.example.com/validate?key=${apiKey}`, { method: 'GET' });
  return response.status === 200;
});

const apiKey = 'some-api-key';
const isApiKeyValid = await isApiKeyValidGuard.execGuardWithData(apiKey);
console.log(isApiKeyValid); // true if the API key is valid
```

In this example, the guard performs an asynchronous API request to validate an API key.

Default Error Handling

When using `@push.rocks/smartguard`, you can take advantage of built-in error handling mechanisms. If a guard fails, it throws an error that you can catch and handle accordingly.

```
import { Guard, passGuardsOrReject } from '@push.rocks/smartguard';

const isNonEmptyStringGuard = new Guard<string>(async (data) => {
  return typeof data === 'string' && data.trim().length > 0;
```

```
});

const validateInput = async (input: string) => {
  try {
    await passGuardsOrReject(input, [isNonEmptyStringGuard]);
    console.log('Input is valid');
  } catch (error) {
    console.error('Validation failed:', error.message);
  }
};

await validateInput(''); // Will print "Validation failed: Guard failed"
await validateInput('Valid input'); // Will print "Input is valid"
```

In this example, we use the `passGuardsOrReject` function to validate an input. If the input is invalid, `passGuardsOrReject` throws an error that is caught and handled in the `catch` block.

Extending Guard Functionalities

Sometimes, you may need to extend or customize the functionalities of a guard to suit specific requirements. `@push.rocks/smartguard` allows you to extend the `Guard` class to create specialized guards.

```
import { Guard } from '@push.rocks/smartguard';

class MinLengthGuard extends Guard<string> {
  constructor(private minLength: number) {
    super(async (data) => {
      return typeof data === 'string' && data.length >= this.minLength;
    });
  }
}

const minLengthGuard = new MinLengthGuard(10);

const isLongEnough = await minLengthGuard.execGuardWithData('Hello, world!');
console.log(isLongEnough); // true because the length of 'Hello, world!' is more than 10
```

In this example, we create a `MinLengthGuard` class that extends `Guard` and validates a string based on its minimum length.

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.

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/smartguard

2024-08-25 - 3.1.0 - feat(core)

Added Guard Handling and Removed npmci from CI

- Refactored Guard and GuardSet classes for better modularization
- Introduced GuardError for detailed error handling
- Updated dependencies versions in package.json
- Removed `npmci` configuration from GitLab CI

2024-05-30 - 3.0.0 to 3.0.2 - Core and API updates

Series of updates and fixes.

- BREAKING CHANGE(api): changed API to be more concise
- fix(core): update

2022-03-21 - 1.0.5 to 2.0.1 - Core updates and new org scheme

Multiple updates including breaking changes and new organizational scheme.

- BREAKING CHANGE(core): updated to esm
- switch to new org scheme

2019-08-07 - 1.0.3 to 1.0.5 - Core updates

Fixes for core components.

- fix(core): update