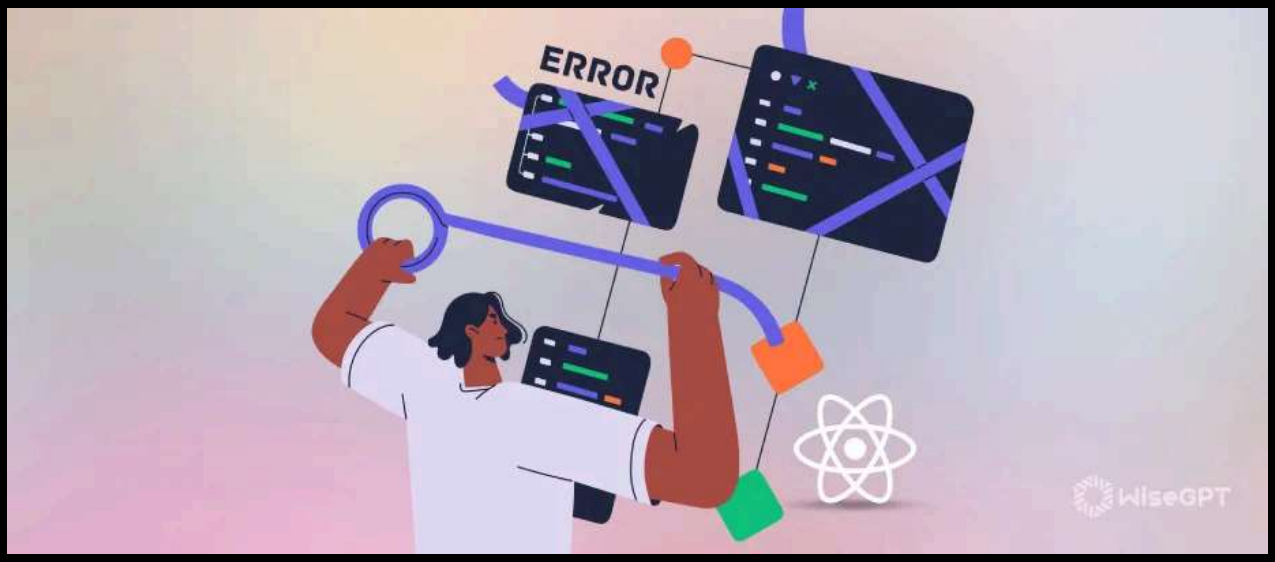
EDUCATION

TypeScript Error Handling: Common Pitfalls and How to Avoid Them


Rakesh Purohit
Developer Advocate

JAN 23, 2024



Error handling is a critical aspect of software development, and TypeScript, being a superset of JavaScript, offers a robust set of features to manage and handle errors effectively. Understanding how to leverage TypeScript's error-handling mechanisms is essential for developers to write resilient and reliable applications.

In TypeScript, the try...catch statement is a fundamental construct for catching and handling exceptions during code execution.

This mechanism is similar to that found in JavaScript but has additional features beneficial for type checking and catching errors at compile time.

TypeScript's approach to error handling not only helps in managing runtime errors but also assists in preventing potential issues during

development by enforcing type safety.

This ensures that errors are caught early in the development process, reducing the likelihood of bugs entering production.

As we delve deeper into TypeScript's error-handling capabilities, we'll explore how to throw errors, handle them, and create custom error classes to improve the clarity and maintainability of our code. We'll also address common mistakes and share best practices to help you avoid pitfalls and handle errors like a pro.

By the end of this blog, you'll have a solid understanding of TypeScript's error-handling patterns and how to apply them effectively in your projects. Whether you're a beginner or an experienced developer, mastering error handling in TypeScript is a valuable skill that will enhance the quality of your code and the robustness of your applications.

Understanding TypeScript's Error Object

When an error occurs in TypeScript, it is represented by an error object. This object is an instance of the Error class or a subclass, such as SyntaxError, TypeError, or ReferenceError.

The error object typically contains two main properties: name, which indicates the type of error, and message, which provides a human-readable description of the error.

TypeScript enhances error handling by allowing developers to specify the catch clause's error parameter type. This feature, introduced in TypeScript 4.0, enables developers to narrow down the error type within the catch block, providing more control over error handling.

The error object in TypeScript can be extended to create custom error classes. This is particularly useful when you want to add context or behavior to the errors in your application.

By creating a custom error class, you can include extra properties or methods to help debug or provide the user with more detailed information.

Throwing Errors in TypeScript Functions

Throwing errors intentionally in your TypeScript code is a way to signal that something unexpected has occurred. You use the throw statement followed by an error object to throw an error. This can be an instance of the built-in Error class or a custom error class you've defined to represent specific error conditions.

```
function calculateSquareRoot(number: number): number {
  if (number < 0) {
    throw new Error("Cannot calculate the square root of a negat
  }
  return Math.sqrt(number);
}

```

In the above example, we throw a new error with a descriptive message when the input is invalid. This pattern is typical in functions that need to validate their arguments and cannot proceed with execution when the statements do not meet specific criteria.

Custom error classes can be instrumental when handling specific kinds of errors differently. By extending the base Error class, you can create a hierarchy of error types that can be used to provide more granular control over error handling in your application.

```
class ValidationError extends Error {
  constructor(message: string) {
    super(message);
    this.name = "ValidationError";
  }
}

```

With a custom error class like ValidationError, you can throw more descriptive errors that can be caught and handled separately from other types of errors.

TypeScript's Try Catch Mechanism Explained

The try...catch mechanism in TypeScript catches exceptions during code execution within the try block. When an error is thrown, the control is passed to the catch block, allowing you to handle the error gracefully.

```
try {
  const result = calculateSquareRoot(-1);
  console.log(result);
} catch (error) {
  if (error instanceof Error) {
    console.error(error.message);
  }
}

```

In this example, if the calculateSquareRoot function throws an error, the catch block catches it, and we check if the caught object is an instance of the Error class before logging the error message. This type of checking within the catch block is crucial to ensure we are handling the error correctly and not encountering any unexpected issues.

The catch block can handle different errors differently based on the error type or other custom logic. This flexibility allows developers to write more robust and user-friendly error-handling code.

Custom Error Classes and Error Handling Patterns

Creating custom error classes in TypeScript is a powerful way to enhance error handling. By extending the native Error class, you can add context or behavior to the errors in your application.

Custom error classes are beneficial when you want to differentiate between error types or when you need to attach extra data to the errors being thrown.

```
class DatabaseError extends Error {
  constructor(message: string, public code: number) {
    super(message);
    this.name = 'DatabaseError';
  }
}

```

In the above example, the DatabaseError class includes a custom property code that could represent an error code specific to your application's database errors. This allows for more detailed error handling and can help debug by clearly indicating what went wrong.

When using custom error classes, using the instance operator within your catch blocks is essential to differentiate between error types and handle them accordingly.

```
try {
  // Some database operation that may fail
} catch (error) {
  if (error instanceof DatabaseError) {
    // Handle database errors specifically
    console.error(`Database error (${error.code}): ${error.messa
  } else if (error instanceof Error) {
    // Handle general errors
    console.error(error.message);
  }
}

```

This pattern of checking the instance of the error allows for more precise and targeted error handling, ensuring that each error type is dealt with most appropriately.

Advanced Error Handling: Using the finally Block and Promises

In addition to try-and-catch blocks, TypeScript supports the final block, executed after the try-and-catch blocks, regardless of whether an error was thrown. This is the perfect place to put cleanup code that needs to run no matter what happens in the try block.

```
try {
  // Code that may throw an error
} catch (error) {
  // Error handling logic
} finally {
  // Cleanup code, such as closing resources
```

```
}

```

Handling errors in asynchronous code is another area where TypeScript shines. With the advent of Promises and async/await syntax, error handling in asynchronous operations has become more straightforward.

```
async function fetchData() {
  try {
    const data = await fetch('some-api-endpoint');
    return await data.json();
  } catch (error) {
    // Handle errors that occur during the fetch operation
    console.error('Failed to fetch data:', error);
  }
}

```

In this asynchronous function, the await keyword is used within a try block, allowing the catch block to handle any errors during the fetch operation.

This pattern keeps the asynchronous code clean and easy to read while still providing robust error-handling capabilities.

Common Mistakes and Solutions in TypeScript Error Handling

Error handling in TypeScript can be tricky, and developers often fall into common pitfalls that lead to less reliable code.

One such mistake is not properly checking the type of the error in a catch block, which can result in unexpected behavior if the error is not the type you anticipated.

Another frequent oversight is not rethrowing errors after they have been logged or handled. This can swallow exceptions and make it difficult to trace the flow of errors through the application, especially if you have multiple layers of error handling.

```
try {
  // Code that may throw an error
} catch (error) {
  console.error('An error occurred:', error);
  // Rethrow the error if it needs to be handled further up the
  throw error;
}

```

By rethrowing the error, you allow higher levels of your application to handle the error or log it as needed. This maintains the error's call stack and makes debugging much more accessible.

It's also important to avoid overly broad catch blocks that catch more than they should. This can hide errors and make understanding what's going wrong in your application harder.

Logging and Reporting Errors Effectively in TypeScript Projects

Effective logging and reporting are crucial for monitoring and debugging errors in TypeScript applications. When logging errors, providing as much context as possible, including the error message, stack trace, and any custom properties you've added to your error classes, is essential.

```
function logError(error: Error) {
  console.error(`Error: ${error.message}`);
  if ('stack' in error) {
    console.error(`Stack trace: ${error.stack}`);
  }
  // Additional logging logic, such as sending the error to a mo
}

```

In the above example, we check for the presence of a stack property before logging it, which is a good practice since not all thrown objects will have a stack trace.

For reporting errors, consider integrating with error monitoring services that can capture, track, and alert you to mistakes as they

occur in your application. These services often provide tools for analyzing error trends and can help you respond to issues more quickly.

Following these best practices for logging and reporting errors can create a more maintainable and reliable TypeScript application.

Proper error handling, logging, and reporting enable you to identify and resolve issues quickly, improving your application's overall user experience and stability.

Conclusion

TypeScript's error handling capabilities are robust and provide developers with the tools needed to manage exceptions effectively. By understanding how to throw errors, create custom error classes, and implement try...catch blocks, you can write cleaner, more resilient TypeScript code.

Remember to avoid common mistakes and to log and report errors efficiently to maintain a high-quality codebase. With these practices in place, you'll be well-equipped to tackle any errors that occur in your TypeScript projects.

So, are you ready to take your TypeScript skills to the next level?

DhiWise React Builder empowers you to build production-ready web applications with ease, complete with built-in error handling best practices and automated workflows. Try DhiWise today and experience the power of effortless React TypeScript development.

DhiWise is free to get started with.

Sign up for free

Products

Flutter

React

Android

iOS

Design to code

Figma plugin

Templates

Screen Library

Learning

Documentation

Blog

DhiWise University

Use Cases

Comparison

DhiWise vs Anima

DhiWise vs Appsmith

DhiWise vs FlutterFlow

DhiWise vs Monday Hero

DhiWise vs Retool

DhiWise vs Supernova

DhiWise vs Amplication

DhiWise vs Bubble

DhiWise vs Figma Dev Mode

Company

About Us

Contact Us

Career

Terms of Service

Privacy Policy

DhiWise BETA