**Error Handling With Angular**

Error handling is an important aspect of writing reliable and maintainable software. In Angular, there are several ways to handle errors, depending on the type and nature of the error.

One way to handle errors is to use try-catch blocks. Try-catch blocks allow you to catch and handle exceptions that are thrown during the execution of your code. For example:

try {

// Code that might throw an exception

} catch (error) {

// Code to handle the exception

}

JavaScript

You can also use the async-await pattern to handle errors in asynchronous code. This pattern allows you to write asynchronous code that looks and behaves like synchronous code, making it easier to read and debug. For example:

async function example() {

try {

const result = await someAsyncFunction();

// Do something with the result

} catch (error) {

// Code to handle the error

}

}

JavaScript

In addition to try-catch blocks and async-await, Angular provides a global error handler service called the ErrorHandler. The ErrorHandler is a service that intercepts uncaught errors and logs them to the console. You can customize the ErrorHandler service by providing your own implementation.

import { ErrorHandler } from '@angular/core';

export class MyErrorHandler implements ErrorHandler {

handleError(error: any) {

// Custom error handling logic

}

}

JavaScript

You can also use the HTTP\_INTERCEPTORS provider to intercept and handle errors that occur during HTTP requests. This can be useful for handling HTTP errors such as 404 (Not Found) or 500 (Internal Server Error).

import { HttpInterceptor, HttpRequest, HttpHandler, HttpErrorResponse } from '@angular/common/http';

import { Injectable } from '@angular/core';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Injectable()

export class HttpErrorInterceptor implements HttpInterceptor {

intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(request).pipe(

catchError((error: HttpErrorResponse) => {

// Handle the error and return a new observable

return throwError(error);

})

);

}

}

JavaScript

Finally, you can use the @Error decorator to handle errors in Angular components. The @Error decorator allows you to specify a method that will be called whenever an error is thrown inside the component.

import { Error } from '@angular/compiler';

@Component({

// ...

})

export class MyComponent {

@Error('myErrorHandler')

myMethod() {

// Code that might throw an error

}

myErrorHandler(error: any) {

// Code to handle the error

}

}

JavaScript

I hope this information helps! Let me know if you have any questions.

Error Handling With Angular

* [Author Profile PhotoMohit Ande](https://www.c-sharpcorner.com/members/mohit-ande)
* 2y
* 9.8k
* 2
* 5

[100](https://www.c-sharpcorner.com/members/rank.aspx?page=points-table)

* [Article](https://www.c-sharpcorner.com/articles/)

Error handling is an important aspect of writing reliable and maintainable software. In Angular, there are several ways to handle errors, depending on the type and nature of the error.

One way to handle errors is to use try-catch blocks. Try-catch blocks allow you to catch and handle exceptions that are thrown during the execution of your code. For example:

try {

// Code that might throw an exception

} catch (error) {

// Code to handle the exception

}

JavaScript

You can also use the async-await pattern to handle errors in asynchronous code. This pattern allows you to write asynchronous code that looks and behaves like synchronous code, making it easier to read and debug. For example:

async function example() {

try {

const result = await someAsyncFunction();

// Do something with the result

} catch (error) {

// Code to handle the error

}

}

JavaScript

In addition to try-catch blocks and async-await, Angular provides a global error handler service called the ErrorHandler. The ErrorHandler is a service that intercepts uncaught errors and logs them to the console. You can customize the ErrorHandler service by providing your own implementation.

import { ErrorHandler } from '@angular/core';

export class MyErrorHandler implements ErrorHandler {

handleError(error: any) {

// Custom error handling logic

}

}

JavaScript

You can also use the HTTP\_INTERCEPTORS provider to intercept and handle errors that occur during HTTP requests. This can be useful for handling HTTP errors such as 404 (Not Found) or 500 (Internal Server Error).

import { HttpInterceptor, HttpRequest, HttpHandler, HttpErrorResponse } from '@angular/common/http';

import { Injectable } from '@angular/core';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Injectable()

export class HttpErrorInterceptor implements HttpInterceptor {

intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

return next.handle(request).pipe(

catchError((error: HttpErrorResponse) => {

// Handle the error and return a new observable

return throwError(error);

})

);

}

}

JavaScript

Finally, you can use the @Error decorator to handle errors in Angular components. The @Error decorator allows you to specify a method that will be called whenever an error is thrown inside the component.

import { Error } from '@angular/compiler';

@Component({

// ...

})

export class MyComponent {

@Error('myErrorHandler')

myMethod() {

// Code that might throw an error

}

myErrorHandler(error: any) {

// Code to handle the error

}

}

JavaScript

I hope this information helps! Let me know if you have any questions.

**Error Handling and Logging in Angular: Best Practices and Strategies**

**Introduction**

In the dynamic landscape of web development, errors are inevitable. Whether it’s a minor user input mistake or a critical backend failure, handling errors effectively is paramount to delivering a seamless user experience. In Angular, a powerful JavaScript framework, mastering error handling and logging mechanisms can significantly impact your application’s reliability. In this article, we’ll delve into the best practices and strategies for proficient error handling and logging in Angular applications.

**Understanding the Importance**

Before we dive into the technical details, let’s establish why error handling and logging are indispensable aspects of Angular applications. Effective error handling provides users with transparent and actionable information whenever something goes awry. This prevents user frustration and confusion, ultimately leading to a higher level of user satisfaction. Furthermore, robust logging empowers developers to identify and resolve issues within the application, even before users encounter them, expediting the debugging and troubleshooting processes.

**Angular’s Error Handling Mechanisms**

Angular equips developers with a range of mechanisms to gracefully handle errors:

**1. Try-Catch Blocks:** Just like in plain JavaScript, you can utilize try-catch blocks to capture and manage errors within specific sections of your Angular code. This is particularly useful for synchronous operations.

try {

// Your code that might throw an error

} catch (error) {

// Handle the error here

}

**2. Angular ErrorHandler:** Angular presents a built-in **ErrorHandler** class that can be extended to customize global error handling across your application. This centralizes error management and ensures a consistent user experience.

import { ErrorHandler } from '@angular/core';

class CustomErrorHandler implements ErrorHandler {

handleError(error: any): void {

// Handle the error globally

}

}

**3. RxJS catchError Operator:** For asynchronous operations involving observables, the **catchError** operator from the RxJS library is a valuable tool. It enables you to intercept and handle errors elegantly.

import { catchError } from 'rxjs/operators';

someObservable$.pipe(

catchError(error => {

// Handle the error here

throw error; // Rethrow the error or provide a fallback value

})

);

**Logging Strategies**

Efficient logging goes hand in hand with error handling, offering insights into your application’s behavior. Here’s how to approach logging in Angular:

1. **Console Logging:** While beneficial for debugging purposes, excessive **console.log** statements should be minimized in production code. They can clutter the console and potentially impact performance.

console.log('Debugging information'); // Use sparingly in production

**2. Logging Libraries:** Consider integrating logging libraries like **ngx-logger** or **angular2-logger**. These libraries provide features such as log levels, advanced formatting, and compatibility with various log aggregation tools.

import { NGXLogger } from 'ngx-logger';

class MyComponent {

constructor(private logger: NGXLogger) {

this.logger.info('Info log message');

this.logger.error('Error log message');

}

}

**3. Server-Side Logging:** To gain deeper insights into production errors, implement server-side logging. This involves transmitting logs to a server where they can be stored, analyzed, and acted upon.

// Example of sending logs to a server endpoint

function sendLogToServer(logMessage: string) {

// Implement the logic to send the log to the server

}

**User-Friendly Error Messages**

Incorporating user-friendly error messages contributes to a positive user experience, even in the face of errors. Craft error messages that convey meaningful information about the issue and potential solutions. Avoid exposing intricate technical details that might confuse users.

**Error Tracking and Monitoring**

Leverage error tracking and monitoring tools to keep your application’s health in check. Services like Sentry and Rollbar can capture and report errors in real-time, allowing you to proactively address issues before they impact users.

**Handling HTTP Errors**

Given the prominence of API calls in modern web applications, adeptly managing HTTP errors is crucial. Utilize RxJS operators like **retry** and **catchError** to handle HTTP errors gracefully. Provide user feedback when an API call fails, guiding them on appropriate actions.

import { HttpClient } from '@angular/common/http';

import { catchError, retry } from 'rxjs/operators';

class ApiService {

constructor(private http: HttpClient) {}

fetchData() {

return this.http.get('/api/data').pipe(

retry(2), // Retry the request up to 2 times on error

catchError(error => {

// Handle the error and provide user feedback

throw error; // Rethrow the error or provide a fallback value

})

);

}

}

**Conclusion**

Error handling and logging may not steal the limelight, but they are the backbone of a dependable and user-friendly Angular application. By implementing the strategies discussed in this article, you can streamline error detection, offer clear guidance to users, and ensure a smooth application experience, even when things don't go as planned. Always remember, a well-handled error is a chance to enhance the overall user experience and build trust in your application.