Photo by Nathan Dumlao on Unsplash

Expecting the Unexpected—Best practices for Error handling in Angular

Michael Karén Follow
Jan 29 · 7 min read

> *"To expect the unexpected shows a thoroughly modern intellect."—Oscar Wilde*

In this article, I write about **centralizing error handling** in Angular. I discuss some of the more common topics such as:

- client-side errors
- server-side errors
- user notification
- tracking errors

I present some code snippets during the way and lastly provide a link to the full example.

• • •

Whom should we blame for errors?

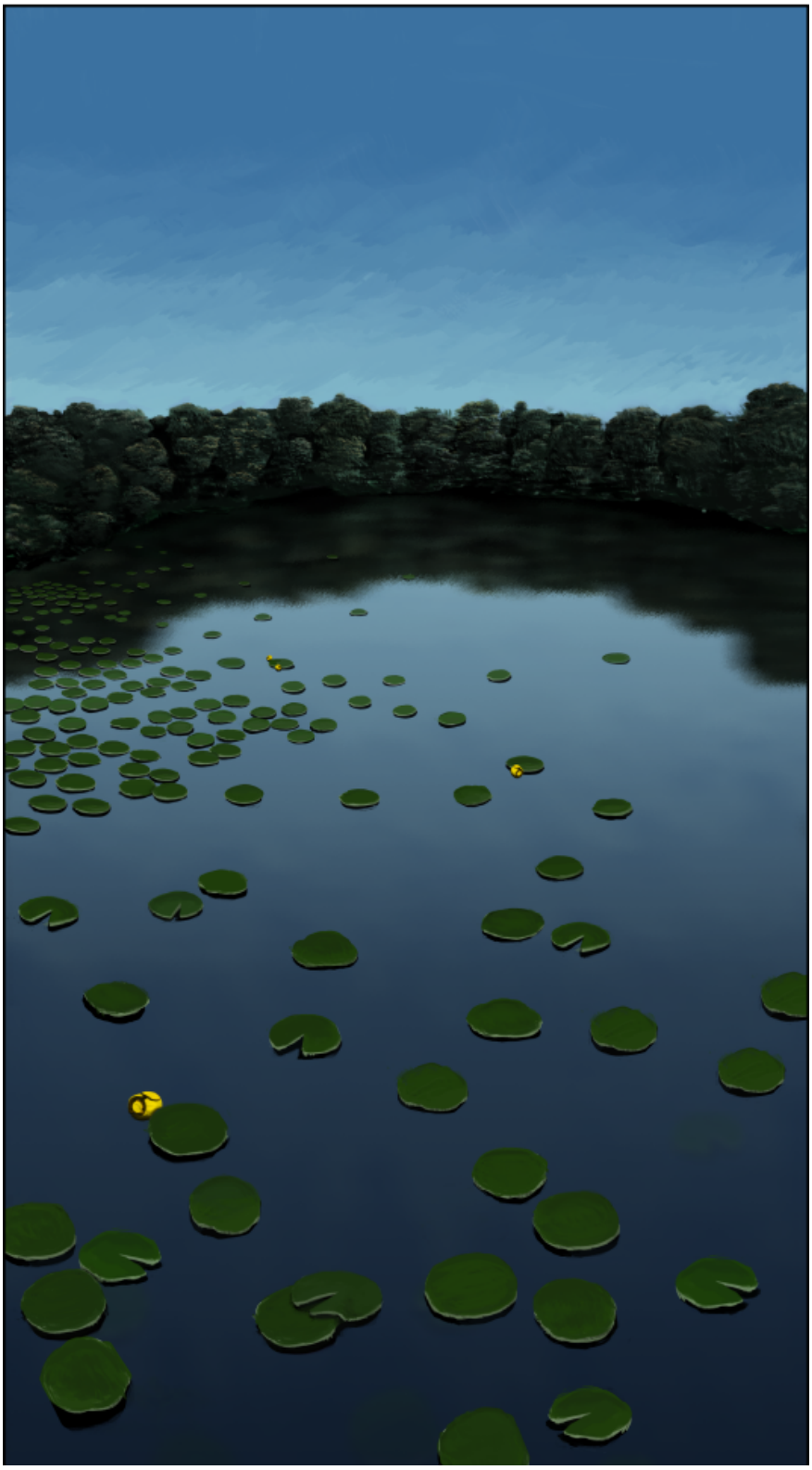
Why do we have errors in our applications? Why can't we write code from specifications that always works?

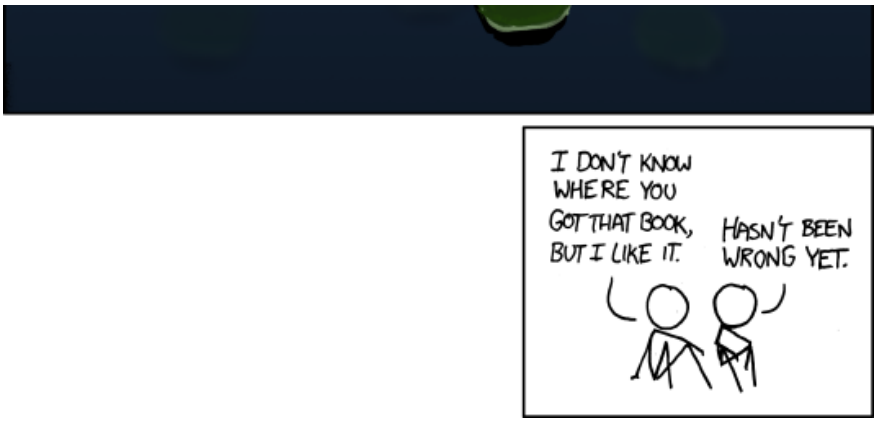
Ultimately, human beings create software, and we are prone to make mistakes. Some reasons behind errors could be:

1. Complexity of application
2. Communication between stakeholders
3. Developer mistakes
4. Time pressure
5. Lack of testing

This list could go on and on. With this in mind, the time comes when the **unexpected happens, and an error is thrown.**


"ERROR: -41"?
THAT'S HELPFUL. IT DOESN'T EVEN SAY WHICH PROGRAM IT'S FROM!
-41? I'LL LOOK IT UP...
IT SAYS -41 IS: "SIT BY A LAKE."





https://xkcd.com/1024/

. . .

Catch them if you can

To catch synchronous exceptions in the code, we can add a `try/catch` block. If an error is thrown inside `try` then we `catch` it and handle it. If we don't do this, the script execution stops.

```
try {
  throw new Error('En error happened');
}
catch (error) {
  console.error('Log error', error);
}
```

Understandably, this becomes unsustainable very fast. We can't try to catch errors everywhere in the code. We need global error handling.

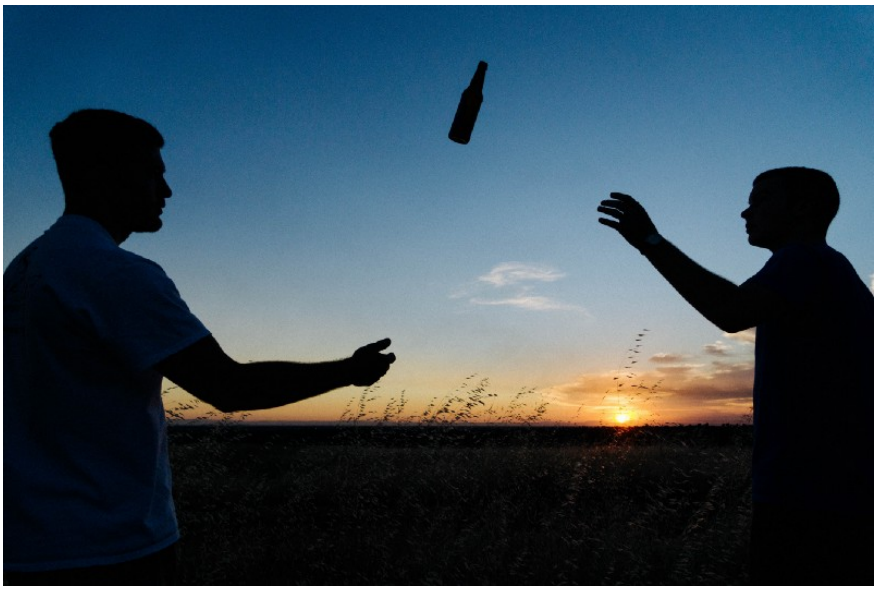


Photo by Wil Stewart on Unsplash

Catch'em all

Fortunately, Angular provides a hook for centralized exception handling with ErrorHandler.

> *The default implementation of* `ErrorHandler` *prints error messages to the* `console` .

We can modify this behavior by creating a class that implements the ErrorHandler:

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

@Injectable()
export class GlobalErrorHandler implements ErrorHandler

  handleError(error) {
    // your custom error handling logic
```

Then, we provide it in our root module to change default behavior in our application. Instead of using the default ErrorHandler class we are using our class.

```
@NgModule({
  providers: [{provide: ErrorHandler, useClass: GlobalE
})
```

So now we only have one place where to change the code for error handling.

• • •

Photo by NeONBRAND on Unsplash

Client-side errors

On the client side, when something unexpected happens, a JavaScript Error is thrown. It has two important properties that we can use.

1. message—Human-readable description of the error.
2. stack—Error stack trace with a history (call stack) of what files were 'responsible' of causing that Error.

Typically, the message property is what we show the user if we don't write our error messages.

Photo by imgix on Unsplash

Server-side errors

On the server-side, when something goes wrong, a HttpErrorResponse is returned. As with the JavaScript error, it has a message property that we can use for notifications.

It also returns the status code of the error. These can be of different types. If it starts with a four (4xx), then the client did something unexpected. For example, if we get the status 400 (Bad Request), then the request that the client sent was not what the server was expecting.

Statuses starting with five (5xx) are server errors. The most typical is the 500 Internal Server Error, a very general HTTP status code that means something has gone wrong on the **server**, but the server could not be more specific on what the exact problem is.

> *With different kinds of errors, it is helpful with a service that parses messages and stack traces from them.*

Error service

In this service, we add the logic for parsing error messages and stack traces from the server and client. This example is very simplistic. For more advanced use cases we could use something like stacktrace.js.

The logic in this service depends on what kind of errors we receive from our backend. It also depends on what kind of message we want to show to our users.

Usually, we don't show the stack trace to our users. However, if we are not in a production environment, we might want to show the stack trace to the testers. In that scenario, we can set a flag that shows the stack trace.

```
import { Injectable } from '@angular/core';
import { HttpErrorResponse } from '@angular/common/htt

@Injectable({
    providedIn: 'root'
})
export class ErrorService {

    getClientMessage(error: Error): string {
        if (!navigator.onLine) {
            return 'No Internet Connection';
        }
        return error.message ? error.message : error.t
    }

    getClientStack(error: Error): string {
        return error.stack;
    }

```

. . .

HttpInterceptor

HttpInterceptor was introduced with Angular 4.3.1. It provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

There are two use cases that we can implement in the interceptor.

First, we can **retry the HTTP call** once or multiple times before we throw the error. In some cases, for example, if we get a timeout, we can continue without throwing the exception.

For this, we use the retry operator from RxJS to resubscribe to the observable.

More advanced examples of this sort of behavior:

- Retry an observable sequence on error based on custom criteria
- Power of RxJS when using exponential backoff

We can then check the status of the exception and see if it is a **401 unauthorized error**. With token-based security, we can try to **refresh the token**. If this does not work, we can redirect the user to the login page.

```
import { Injectable } from '@angular/core';
import {
  HttpEvent, HttpRequest, HttpHandler,
  HttpInterceptor, HttpErrorResponse
} from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';

@Injectable()
export class ServerErrorInterceptor implements HttpInt

  intercept(request: HttpRequest<any>, next: HttpHandl

    return next.handle(request).pipe(
      retry(1),
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
```

Here we retry once before we check the error status and rethrow the error. Refreshing security tokens is outside the scope of this article.

We also need to provide the interceptor we created.

```
providers: [
  { provide: ErrorHandler, useClass: GlobalErrorHandler
  { provide: HTTP_INTERCEPTORS, useClass: ServerErrorIr
]
```

Notifications

For notifications, I'm using Angular Material Snackbar.

```
import { Injectable} from '@angular/core';
import { MatSnackBar } from '@angular/material/snack-b

@Injectable({
  providedIn: 'root'
})
export class NotificationService {

  constructor(public snackBar: MatSnackBar) { }

  showSuccess(message: string): void {
    this.snackBar.open(message);
  }

```

With this, we have simple notifications for the user when errors occur.

We can handle server-side and client-side errors differently. Instead of notifications, we could show an error page.

Error Message

Error messages matter and should, therefore, have some meaning to help the user to move along. By showing “An error occurred” we are not telling the user what the problem is or how to resolve it.

In comparison, if we instead show something like “Sorry, you are out of money.” then the user knows what the error is. A bit better but it does not help them to resolve the error.

An even better solution would be to tell them to transfer more money and give a link to a money transfer page.

> *Remember that error handling is not a substitute for bad UX.*

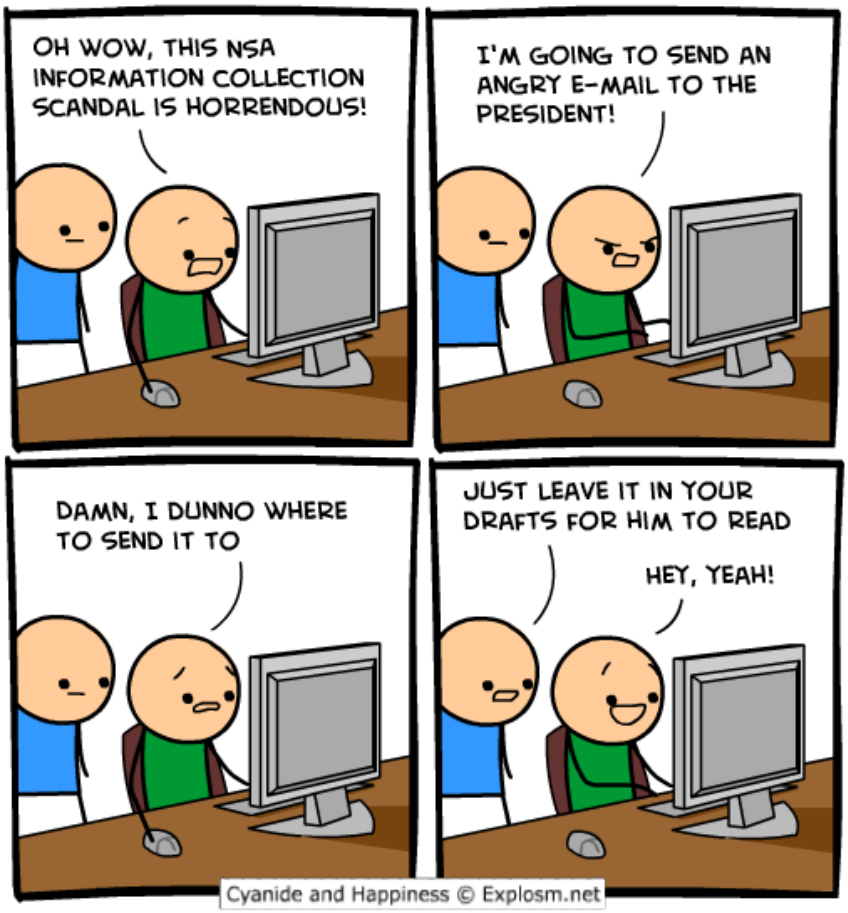
What I mean by this is that you should not have any expected errors. If a user can do something that throws an error, then fix it!

> *Don't let an error through just because you created a nice error message for it.*

Logging

If we don't log errors, then only the user who runs into them knows about them. Saving the information is necessary to be able to troubleshoot the problem later.

When we have decided to store the data we need also choose how to save it. More on that later.



Cyanide & Happiness

> *Where should we save the data?*

With centralized error handling, we don't have to feel too sorry for leaving the decision for later. We only have one place to change our code now. For now, let's log the message to the console.

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

@Injectable({
    providedIn: 'root'
})
export class LoggingService {

  logError(message: string, stack: string) {
    // Send errors to be saved here
```

Error Tracking

Ideally, you want to identify bugs in your web application before users encounter them. Error tracking is the process of **proactively identifying issues and fixing them** as quickly as possible.

So, we can't just sit back and expect users to report errors to us. Instead, we should be proactive by logging and monitoring errors.

> *We should know about errors when they happen.*

We could create our solution for this purpose. However, why reinvent the wheel when there are so many excellent services like Bugsnag, Sentry, TrackJs, and Rollbar specializing in this area.

Using one of these front-end error tracking solutions can allow you to record and replay user sessions so that you can see for yourself **exactly what the user experienced.**

> *If you can't reproduce a bug, then you can't fix it.*

In other words, a proper error tracking solution could alert you when an error occurs and provide insights into how to replicate/resolve the issue.

In an earlier article, How to send Errors into Slack in Angular I talked about using Slack to track errors. As an example, we could use it here:

```
import { Injectable } from '@angular/core';
import { SlackService } from './slack.service';

@Injectable({
    providedIn: 'root'
})
export class LoggingService {

    constructor(private slackService: SlackService) {

```

Implementing a more robust solution is outside the scope of this article.

• • •

All together now

Since error handling is essential, it gets loaded first. Because of this, we cant use dependency injection in the constructor for the services. Instead, we have to inject them manually with Injector.

```
import { ErrorHandler, Injectable, Injector } from '@a
import { HttpErrorResponse } from '@angular/common/htt

import { LoggingService } from './services/logging.ser
import { ErrorService } from './services/error.service
import { NotificationService } from './services/notifi

@Injectable()
export class GlobalErrorHandler implements ErrorHandle

    // Error handling is important and needs to be loa
    // Because of this we should manually inject the s
    constructor(private injector: Injector) { }

    handleError(error: Error | HttpErrorResponse) {

        const errorService = this.injector.get(ErrorSe
        const logger = this.injector.get(LoggingServic
        const notifier = this.injector.get(Notificatio

        let message;
        let stackTrace;

        if (error instanceof HttpErrorResponse) {
            // Server Error
            message = errorService.getServerMessage(er
            stackTrace = errorService.getServerStack(e
```

. . .

Conclusion

Error handling is a cornerstone for an enterprise application. In this article, we centralized it by overriding the default behavior of ErrorHandler. We then added several services:

1. Error service to parse messages and stack traces.
2. Notification service to notify users about errors.
3. Logging service to log errors.

We also implemented an interceptor class to:

1. Retry before throwing the error.

2. Check for specific errors and respond accordingly.

With this solution, you can start tracking your errors and hopefully give the users a better experience.

Example code on GitHub. 📜
Run the code on StackBlitz. 🤹‍♂️

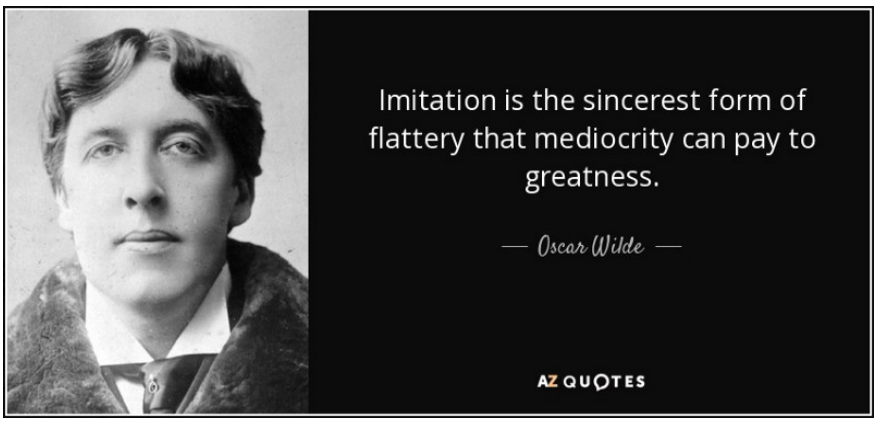
. . .

Call to Action

I always enjoy feedback so please 👏 and 📝 .
Follow me on Twitter and Medium for blog updates.

. . .

I started with a quote from Oscar Wilde so let's end with one as well.



Picture from AZ Quotes

Resources

- Error Handling & Angular by Aleix Suau
- Handling Errors in Javascript: The Definitive Guide by Lukas Gisder-Dubé
- Global Error Handling with Angular 2+ by Austin
- Angular applications—error handling and elegant recovery by Brachi Packter
- The Art of the Error Message by Marina Posniak
- JavaScript Weekly: Graceful Error Handling by Severin Perez