Angular Error Handling With SOLID Principles

Elias Martinez Follow
Feb 1 · 9 min read ★

Server logging is so important to detect application problems and there are many solutions already out there that help you out to capture, analyze and some also go beyond even offers you to tell the exact commit and who could be the best candidate (git blame, is that you?) who can work and fix it (Sentry-coff-coff). That's pretty cool, handy or you put the nice adjective.

Photo by Markus Spiske on Unsplash

However, we all have been in that situation where the company we works for doesn't want to pay for such solutions because of or whatever reason is. I sort of was that guy who had to look for feasible alternative, in the end, if we do it properly, a combination of your own code with an open-source stack can be a great solution that just hit the point and fulfills the requirements.

The requirements

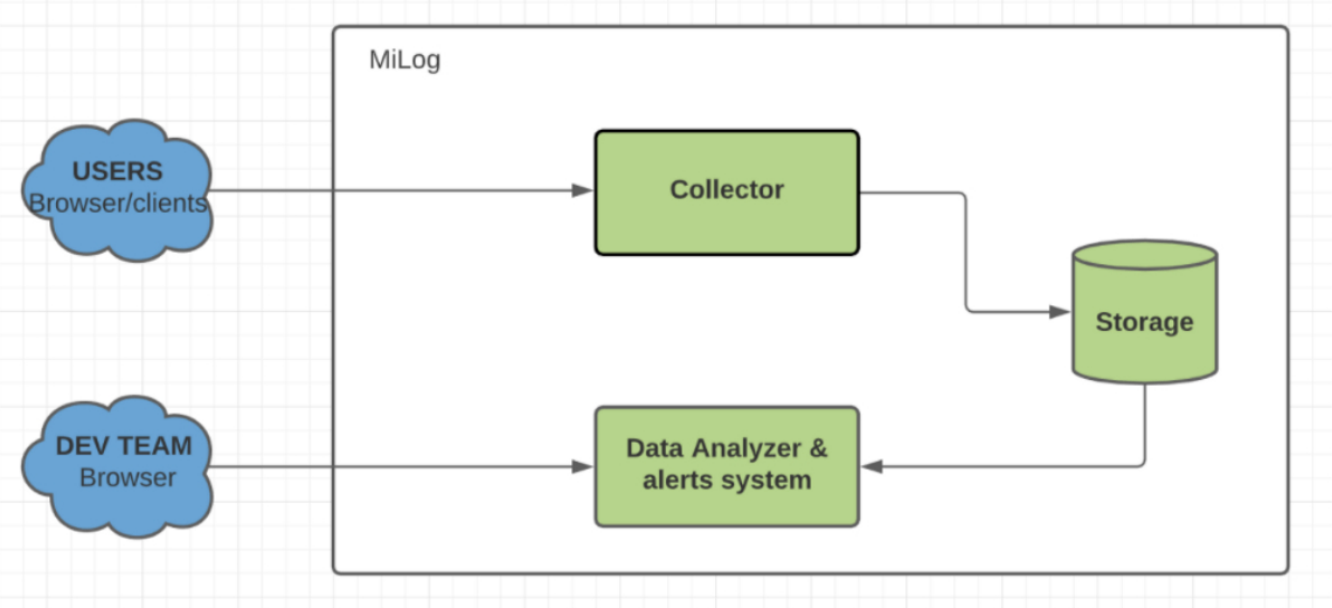
- To have a mechanism to **capture all errors occurring** in the browser.
- To have a utility to log/send events, errors, or messages of different importance levels (you know it, the classic log levels: info, warn, debug, error, etc).
- As a system, we required components capable of **collect & store "log messages/events"**
- As a team, we should be able to see all the captured "logs" that occurred in the last 3 months so the team can analyze errors in different ways (reactive, preventive, monitoring).
- As a system, it is required to have alert messages triggered realtime to inform the support team about "critical errors" occurring in the application

The overall Architecture

In a big big picture, the solution can be composed of two main macro-components:

components:

1. The **Collector, Storage, and Dashboard** for the messages
2. The **browser utilities to capture and drop/send the logs/messages to the collector**



Big picture logging Architecture — Elias Martinez

Concrete components

The Collector Storage & Dashboard

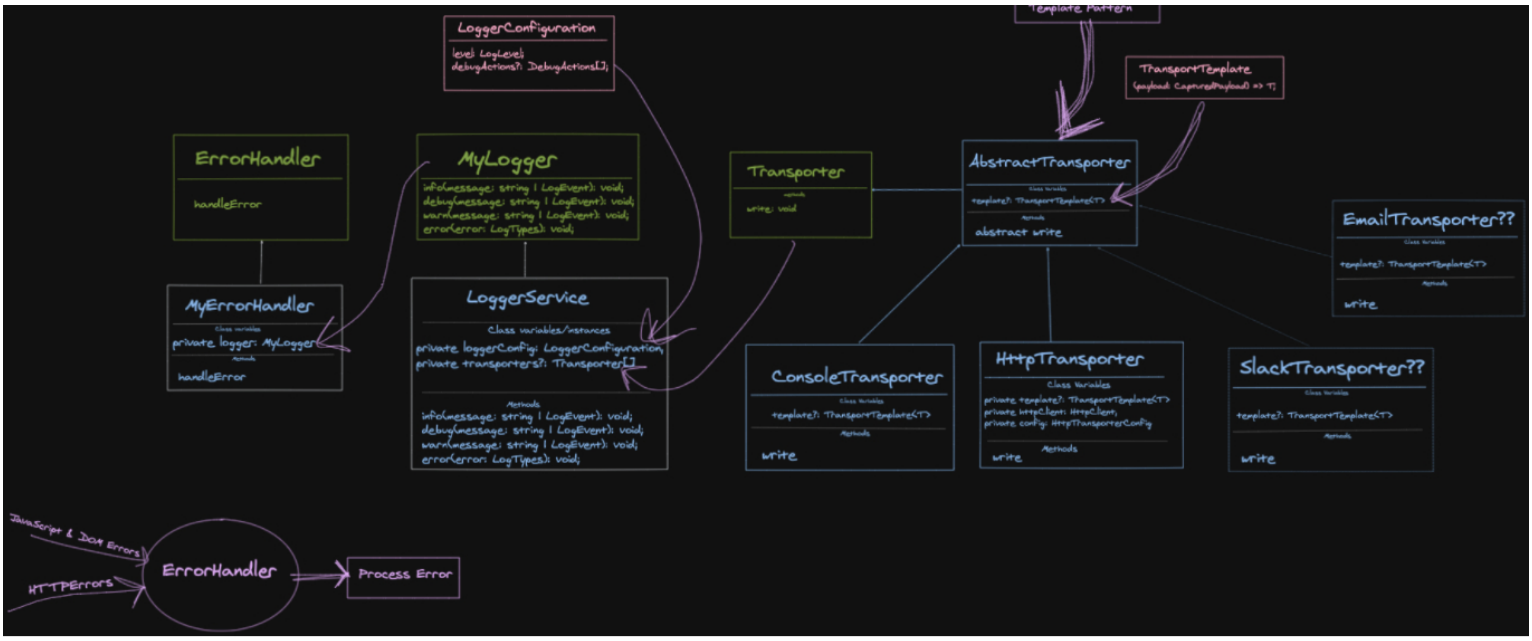
The collector we did was a simple Go server which the only task was to receive a JSON formatted message and drop it to the std.out file (will make more sense in the next paragraph)

The storage and dashboard component we can provide by setting up an EKL stack (more precise Elastic Search, Kibana, LogStash & FileBeat) where **FileBeat streams** the **message logged from the Go server** and **push to a LogStash pipe** to slice each attribute and **finally drop an index-able message to ElasticSearch**. Finally, by the nature of **Kibana**, **we can set up queries, dashboards, and alerts** to be sent when considered "critical" messages (defined completely by you) comes in by the nature of occurrence or amount of occurrences.
In this article **we will focus on the frontend** part, for the backend part maybe in the future I can provide the details on how to implement it.

. . .

The Client-Side Big Picture Parts (or Js components)

Here is what we will need for the recipe:



Whuuut!!... Easy my friend, let's "de-structure" ;) this thing:

1. ErrorHandler

The task of this guy is to listen to **ALL errors occurring in the application**; fortunately Angular provides an interface to use; another framework/libraries I guess they have their own ways, but in theory, we all should be able to implement an onerror callback.

```
window.onerror = funcRef;
```

Screenshot from Mozilla

2. Logger

The **Logger** is another service that can receive via dependency injection all the instantiated and configured "transporters" and use them to write on each different log level messages and implementation.

3. Transporters

Transporters are an interface whose only task is to "write" a message in a particular way. It's a Template pattern where the implementation can vary between different targets from the simple console.log, HTTP, slack, sockets, etc. this is what would make the framework flexible to integrate with other targets.

Ok, Where is my money?

Photo by Melanie Pongratz on Unsplash

Let's see the details, the meaty part, the bull balls, the under the hood, etc.

The ErrorHandler Details

As the picture stands, the ErrorHandler is sort of a proxy for all errors occurring in the application (either Javascript or HTTP errors) to handle them in something like a funnel/hub. Angular provides OOB with his class:

```
class ErrorHandler {   handleError(error: any): void   }
```

And the way to implement it is too easy as just override the provider with yours, like:

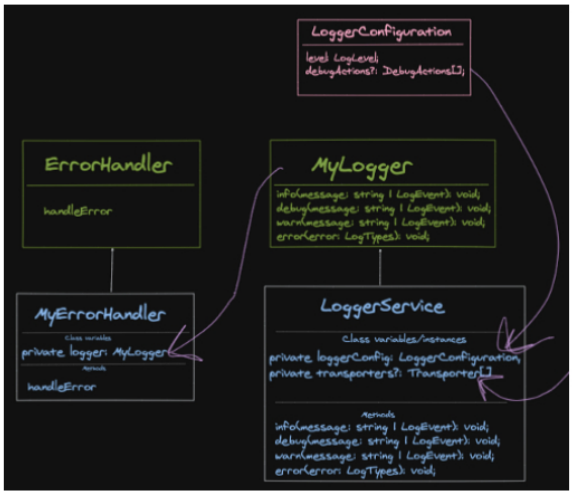
```
class MyErrorHandler implements ErrorHandler {
  handleError(error) {
   // do something with the exception
  }
}

@NgModule({
    providers: [{provide: ErrorHandler, useClass: MyErrorHandler}]
  }
) class AppModule {}
```

Now taking the advantage of Dependency injection **we can** easily **inject our** custom **Logger** that will be explained in more detail next. So in the end, our ErrorHandler might look like this:

```
@Injectable()
export class MyErrorHandler implements ErrorHandler {

  constructor(@Inject(MyLoggerToken) private logger: MyLogger) {}

  handleError(error: Error | HttpErrorResponse): void {
    this.logger.error(error);
  }

}
```

Here, what we are doing is basically the injection of a given logger component. The implementation is unknown yet, whatever we inject as the Logger only knows what it can do, to log info, debug, warn or error messages.

Now “The LoggerService” details



The Logger is maybe the more complex component because the way we implement will define how useful or flexible it behaves, but since we are providing an interface/contract it can be later be replaced for something “better” if you manage to improve it. Another important component are the Transporters which we will see the details later so let’s focus only on the Logger mechanics.

The mechanics of the logger is simply as to say “it needs to drop messages of different levels”, so we need to define the interface among other utilities and interfaces

```
/**
* Logger Injection token
*/
export const MyLoggerToken =
  new InjectionToken<MyLogger>('MyLogger');

/**
* Allowed Log input types to be received by a MyLogger
*/
export type LogTypes = string | Error | HttpErrorResponse;

/**
* Logger Interface. Implementations should implement and expose the
following methods: info, debug, warn, error
*/
export interface MyLogger {
  info(message: string): void;
  debug(message: string): void;
  warn(message: string): void;
  error(error: LogTypes): void;
}
```

So the job of the logger is to drop log messages somewhere depending if the configured log level is higher than what it’s being tried to be logged. Therefore we need to define Log levels.

```
export enum LogLevel {
 Info = 4,
 Debug = 3,
 Warn = 2,
 Error = 1,
}
```

Once we defined the log levels we need to somehow tell the logger what log level is permitted to be logged. for that, we can define an injectable object like **LoggerConfiguration**

```
/*
* LoggerConfiguration Token
*/
export const LoggerConfigurationToken =
  new InjectionToken<LoggerConfiguration>('LoggerConfiguration');

/**
* LoggerConfiguration object interface
*/
export interface LoggerConfiguration {
  level: LogLevel;
}
```

With proper config interfaces in place we can go ahead and provide them so they can be injected to our **Logger** service class for later provide it in the application itself ready with all his things up, so let's see the Logger class implementation:

```
import { HttpErrorResponse } from '@angular/common/http';
import { Inject, Injectable, Optional } from '@angular/core';
import { LogMessage, LogTypes, LogLevel } from '../commons/models';
import { ConsoleTransporter } from '../transporters/console-transporter.service';
import { Transporter, TransporterToken, CapturedPayload } from '../transporters';
import { MyLogger, LoggerConfiguration, LoggerConfigurationToken } from './logger.model';

/**
 * Default Logger implementation
 */
@Injectable()
export class LoggerService<T = LogMessage> implements MyLogger {
  private transporters?: Array<Transporter<T>>;

  constructor(
    @Inject(LoggerConfigurationToken) private loggerConfig: LoggerConfiguration,
    @Optional() @Inject(TransporterToken) transporters?: Array<Transporter<T>>
  ) {
    /*
     * Ensure array is received, if no received it means only one
     * transporter has been defined and was not configured as multi
     */
    this.transporters = (Array.isArray(transporters) ? transporters : [transporters]).filter(Boo
    if (!transporters.length) {
      // in case no transporters were defined we provide the console transporter by default
      this.transporters = [new ConsoleTransporter()];
    }
  }

  /**
   * Logs a info message
   * @param message
   */
  info(message: string | T): void {
    this.logWith(LogLevel.Info, message);
  }

  /**
   * Logs debug action messages
   * @param message
   */
  debug(message: string | T): void {
    this.logWith(LogLevel.Debug, message);
  }

  /**
   * Logs warning level messages
   * @param message
   */
  warn(message: string | T): void {
    this.logWith(LogLevel.Warn, message);
  }

  /**
   * Logs error messages
   * @param error
   */
  error(error: string | Error | HttpErrorResponse | T): void {
```

```
59      this.logWith(LogLevel.Error, error);
60    }
61
62    /**
63     * If payload evaluates as should be logged then a macro-task is scheduled
64     * @param logLevel
65     * @param payload
66     */
67    private logWith(logLevel: LogLevel, payload: LogTypes) {
68      if (this.isEventLevelLessThanLogLevel(logLevel, payload)) {
69        // capture payload, timestamp and level
70        const capturedPayload: CapturedPayload = { payload, level: logLevel, timestamp: new Date()
71        // send captured payload to each transporter to be written
72        for (const transport of this.transporters) {
73          this.execute(transport, capturedPayload, logLevel);
74        }
75      }
76    }
77
78    /**
79     * A log-able payload is it which it's log level is less or equal to configured log level
80     * @param logLevel
81     * @param payload
82     */
83    private isEventLevelLessThanLogLevel(logLevel: LogLevel, payload: LogTypes): boolean {
84      return payload != null && this.loggerConfig.level >= logLevel;
85    }
86
87    /**
88     * Queues the logging event as a micro-task
89     * @param transport
90     * @param payload
91     * @param level
92     */
93    private execute(transport: Transporter<T>, capturedPayload: CapturedPayload, level: LogLevel)
94      // send to event queue to minimize the stack processing for the application
95      queueMicrotask(() => transport.write(capturedPayload, level));
96    }
97  }
```

myLoggerService.ts hosted with ❤ by GitHub view raw

This is a simplified/basic implementation of a logger service, we consume the config files, we have implemented all the methods in the MyLogger interface and for each transporter configured we transfer the control to write it in the way it is designed to transport it.

Now let's see the last piece to wire in.

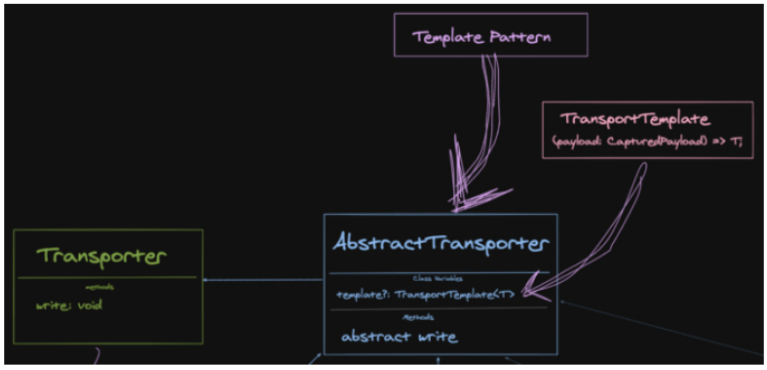
The Transporters

First things first, yes I took inspiration from Winston library. Now, Each transporter is responsible to transport/transfer/write the received payload from the logger (which means it is supposed to be written somewhere) to somewhere/someone else. And this "elsewhere" can be literally whatever (console, HTTP log server, slack channel, email, etc...)

That's it, that simple thing it has to do. merely and simply "write" whatever he gets. The how to write has to be provided as a multi-service

provider. This way we can have multiple providers injected into the **LoggerService**

Now let's define a base template abstract class to save us time while implementing each Transporter. To do so we can implement some sort of Template pattern and let's also introduce one small piece we haven't mentioned before that we can fix with Dependency Injection which is the transport message template (in red)



Having the **AbstractTransporter** implementing the **Transporter** interface will be responsible for registering the transport template which in plain code is just an injected function that will produce a payload of any type based on the **CapturedPayload** interface we implemented in the logger service.

```
import { LogMessage, LogLevel } from '../commons/models';
import { TransportTemplate } from './transporter-template.model';
import { CapturedPayload } from './captured-payload.model.ts';
import { Transporter } from './transporter.model';
/**
 * Base Transporter implementation
 */
export abstract class AbstractTransporter<T = LogMessage> implements Transporter<T> {
  template: TransportTemplate<T>;
  constructor(template: TransportTemplate<T>) {
    this.template = template;
  }

  write(payload: CapturedPayload, level?:LogLevel) {
    const templatedPayload = this.template(payload);
    this.doWrite(templatedPayload, level)
  }

  abstract doWrite(payload: T, level?: LogLevel): void;
}
```

abstract-transporter.ts hosted with ❤ by GitHub view raw

With the *AbstractTransporter* ready to be implemented we can start implementing each transporter specificity, like for example console logger, HTTP transporter, slack transporter, email transporter, socket transporter, or third-party transporter (sentry or any other service).

So let's take a look at a couple of them, Console and Http transporters

ConsoleTransporter

This is probably the basic implementation, we don't even require any template to be consumed or it can be optional.

```
import { LogLevel } from '../commons/models';
import { AbstractTransporter } from './abstract-transporter';
```

```ts
import { AbstractTransporter } from './abstract-transporter';
import { CapturedPayload } from './captured-payload.model.ts';
import { TransportTemplate } from './transporter-template.model';
/**
 * Default ConsoleTransporter template
 */
const defaultConsoleTransporterTemplate = <T>(p: CapturedPayload) => p.payload as T;
/**
 * Console transporter
 */
export class ConsoleTransporter<T> extends AbstractTransporter<T> {
  /**
   * @param template optional if not given, using default (no template, as message comes is writ
   */
  constructor(template?: TransportTemplate<T>) {
    super(template || defaultConsoleTransporterTemplate);
  }
  /**
   * writes payload in console
   */
  doWrite(payload: T, level: LogLevel): void {
    switch (level) {
      case LogLevel.Info:
        return console.info(payload);
      case LogLevel.Debug:
        return console.debug(payload);
      case LogLevel.Warn:
        return console.warn(payload);
      case LogLevel.Error:
        return console.error(payload);
    }
  }
}
```

ConsoleTransporter.ts hosted with ❤ by GitHub view raw

Please notice that what we receive in the “doWrite” method is the result of calling the write method in the logger service a the “execute” method

```ts
queueMicrotask(() => transport.write(capturedPayload, level));
```

And from the next call in the abstract template class

```ts
const templatedPayload = this.template(payload);
this.doWrite(templatedPayload, level)
```

Since the Template is defined while creating the transporters it means that we can have one template for all transporters OR we can have different templates for all transporters, however, all receive the same contract:

```ts
/**
* Final captured event, built-in Logger captures this instance
* before queuing the transport macro task execution, if custom
* Logger implemented it's highly recommended to capture this
* instance before the transporters write processing
*/
export interface CapturedPayload {
  payload: LogTypes;
  level: LogLevel;
  timestamp: Date;
}
```

The HttpTransporter

Finally for this post. Here a simple sample for an HttpTransporter, it extends our AbstractTransporter class and extends few more attributes to be able to work like the httpClient and extra configuration like the endpoint URL or whatever your application needs to perform HTTP calls (cookies management, Bearer token providers what ever it is...) in this

case we only need the endpoint where we want to “write” the payload and the base httpClient, Bearer token (security stuff) is provided in another component of the application.

Photo by Rhys Moult on Unsplash

In this implementation, Let’s use a “batch” processor with a buffer of 500ms to collect all payloads and try to minimize the network calls in case we receive tons of errors (this shouldn’t happen but, we all have horror stories). However, it’s up to you how you implement yours. We are not here to judge the implementation but the overall solution, execution is always debatable right?

```
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';
import { InjectionToken } from '@angular/core';
import { forkJoin, of, Subject } from 'rxjs';
import { buffer, catchError, debounceTime, delay, switchMap, take } from 'rxjs/operators';
import { LogLevel } from '../logger/log-levels.enum';
import { LogMessage } from '../models/log-message.model';
import { logLevelValuesMap } from '../utils/log-levels-values.map';
import { AbstractTransporter } from './abstract-transporter';
import { TransportTemplate } from './transporter-template.model';

export interface HttpTransporterConfig { url: string; }

interface LogBatch<T> { level: LogLevel; payload: T; }

/**
 * Http Transporter
 */
export class HttpTransporter<T = LogMessage> extends AbstractTransporter<T> {
  batcher$ = new Subject<LogBatch<T>>();

  constructor(
    template: TransportTemplate<T>,
    private httpClient: HttpClient,
    private config: HttpTransporterConfig
   ) {
    super(template);

    this.batcher$
      .pipe(
        buffer(this.batcher$.pipe(debounceTime(500))),
        switchMap(batches => forkJoin(this.toHttpRequests(batches)))
      )
      .subscribe();
  }

  private toHttpRequests(batches: Array<LogBatch<T>>) {
    return batches.map(batch =>
      this.makeHttpRequest(this.config, batch.level, batch.payload)
        // make network call and catch errors
        .pipe(
          take(1),
          catchError(error => {
            // just in case LOL
            console.error('Error while logging message', batch.payload, 'Error:', error);
            return EMPTY;
          })
        )
    );
  }

  /**
   * Creates http observable with given params (url, httpMethod, level & payload)
   */
```

```
  private makeHttpRequest(
    { url }: HttpTransporterConfig,
    logLevel: LogLevel,
    payload: T
  ) {
    // build object payload (ensure final message is an object)
    const level = payload['level'] || logLevelValuesMap[logLevel];
    // making sure we send a Object literal, in case we receive primitive payload
    const data: { [string]: any } = { ...this.ensureData(payload), level };
    const headers = { 'Access-Control-Allow-Origin', '*' };
    return this.httpClient.post(url, data, { headers });
  }

  /**
   * Writes/sends payload to configured url and given Http Method
   */
  doWrite(payload: T, level: LogLevel): void {
    this.batcher$.next({ level, payload });
  }

  /**
  * Make sure passed payload is a object literal.
  */
  private ensureData(message: any): { [string]: any } {
    return this.isPrimitive(message) ? { meta: { message } } : message;
  }

  private isPrimitive(message: any): boolean {
    return (
      typeof message === 'string' ||
      typeof message === 'number' ||
      typeof message === 'boolean' ||
      typeof message == null ||
      message instanceof Date ||
      typeof message === 'function'
    );
  }
}
```

HttpTransporter.ts hosted with ❤ by GitHub view raw

All right where do I connect all the pieces? So far we have defined and separated the responsibilities and we got a bunch of open-close components.

The last thing to do is… Provide all the pieces

Yeah, we just need to register the logger, transporters & template and we all can guess where. in the **AppModule/CoreModule.**

Provide the logger

The **Logger** can be provided in different ways, either you create all logger functionality in a module (recommended) or directly in the app module or core module(whatever is your root module).

Let's assume we implemented a module, therefore we can add something like this:

```
@NgModule({
  imports: [HttpClientModule],
  declarations: [],
  exports: [],
})
export class MyLoggerModule {
  static forRoot(moduleConfig: LoggerModuleConfig): ModuleWithProviders {
    /*
     * Return ModuleWithProviders
     */
    return {
      ngModule: MyLoggerModule,
      providers: [
        {
          // used to pass any external information that logger might need
          // to work (urls, tokens, etc, so far only level we are providing)
```

```
          provide: LoggerConfigurationToken,
          useValue: { level: moduleConfig.level },
        },
        {
          /*
          Here we provide our logger by default if logger module is imported.
          and that's it, transporters and template are the ones whe will be
          provided entirely by external source (in case you wrap this in a library)
          we leave open that door but we set the rules with the interfaces.
          */
          provide: MyLoggerToken,
          useClass: Logger,
        },
        // Override default angular error handler with ours
        { provide: ErrorHandler, useClass: MyErrorHandler },
      ],
    };
  }
}
```

logger.module.ts hosted with ❤ by GitHub view raw

Provide the Transporters and template

The rest of the pieces since they can be implemented in varying vast ways and actually needs to fit with every “user” of the module they should be provided in the app module and via factory providers:

```
// ***** IMPORTANT NOTE ****: We use factories because
// Angular don't allows you to execute functions at this stage

/**
 * Http Transporter Factory
 */
export function MyHttpTransporterFactory(httpClient: HttpClient) {
  /*
   * Make Transporter Template and config
   * just remember this fn receives a CapturedPayload and returns whatever
   * you define if your template requires more deps then you also need to
   * provide another factory and include it in the deps array like httpClient
   */
  const template = MyTransportTemplateFn;
  /* any extra config you need */
  const config = { url: FRONTEND_LOGGING_SERVER_URL };
  // create and return the transporter
  return new HttpTransporter(template, httpClient, config);
}

/**
 * Console Transporter Factory
 */
export function MyConsoleTransporterFactory() {
  return new ConsoleTransporter();
  /*
  // template is optional for console transporter but
  // if you want to keep constency also at console log
  // then you can do:
  const template = MyTransportTemplateFn;
  return new ConsoleTransporter(template);
  */
}

const TRANSPORTER_PROVIDERS = [
      // provide Console transporter
  { provide: TransporterToken, useFactory: MyConsoleTransporterFactory, multi: true },
  {
    // provide Http Transporter
    provide: TransporterToken,
    useFactory: MyHttpTransporterFactory,
    multi: true,
    deps: [HttpClient], // <= add as many deps you need
  },
  // keep adding as many transporters you want, you make it crazy
];

// later in the module
@NgModule({
  declarations: [AppComponent],
```

```
51    imports: [/*...exports, etc... and finally*/],
52    provide: [
53       // your other services
54       ...TRANSPORTER_PROVIDERS
55     ]
56    bootstrap: [AppComponent],
57  })
58  export class AppModule {}
59  // ....
```

provide-transporters.module.ts hosted with ❤ by GitHub view raw

That's all folks.

End Thoughts

Photo by Marek Szturc on Unsplash

We should always think and keep present scalability and maintainability of whatever we do. SOLID principles are proven to grant you both; notice that in this solution we applied some SOLID principles, like SRP and OCP by categorizing and segregating the the parts & tasks in such a way we protect the components from knowing from each other, Each component (Logger, Transporters & Message Template) are replaceable & interchangeable.

The solution is not yet perfect but is good enough to provide scalability. In the end, this is yet one of many ways to solve this problem, hope it helps, and happy coding my friends!

. . .

Hey!, Have you heard about object composition?

By The Startup

Get smarter at building your thing. Subscribe to receive The Startup's top 10 most read stories — delivered straight into your inbox, twice a month. Take a look.

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Angular Error Handling Logging JavaScript Web Development

265 7

WRITTEN BY

Elias Martinez

Full-stack software engineer

Follow

Start it up

The Startup

Get smarter at building your thing. Follow to join The Startup's +8 million monthly readers & +740K followers.

Follow

More From Medium

ES6 MVC JavaScript Tutorial: Build a Simple CRUD App

Raja Tamil in The Startup

Environment Variables in Vue

Gaute Meek Olsen

React Native vs React: A Couple First Impression Differences

Alex Foreman

Essentials of MongoDB & JavaScript

Abhishek koserwal

Create Custom Video Controller Using React-Player

Zain Ahmed in GopenSource

02 — Promises, promises (Welcome to JavaScript Async/Await)

Andrew Drummond

Fragments in React

Jhey Tompkins in codeburst

Hosting Custom React Websites via Github Pages

Zoe Millard

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Write a story on Medium.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Start a blog

Medium

About Write Help Legal