Contextual Yet Non-Descriptive Errors

Proposal for Automatically Generated Error Codes

Guillaume Lethuillier
glthr.com

October 24, 2023 · updated November 14, 2026

Abstract

I propose computing contextual error codes as bitmasks: each error-item has a stable power-of-two identifier, selected items are combined with OR, and engineers recover them with AND. The code represents a set of circumstances, complements a readable error message and can be decoded without looking up a log. It obscures implementation details rather than keeping them secret, and its capacity depends on the code's bit width.

Developers must carefully balance the amount of information provided in error messages displayed to end-users. Disclosing too much information through these errors can lead to security breaches. On the other hand, using generic error messages can make it more challenging for support teams to identify and resolve the underlying issues.

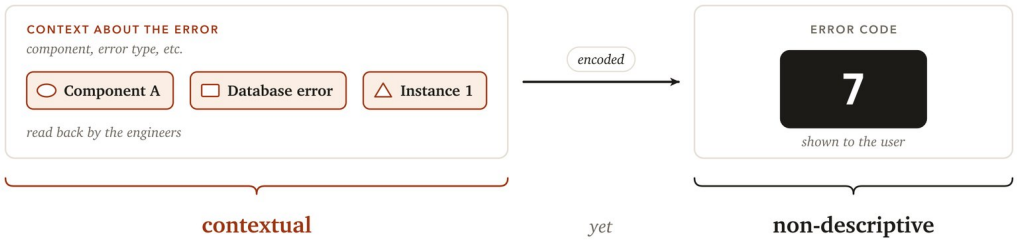
A feasible solution is creating public-facing error messages that map to internal (potentially confidential) error objects, such as iOS update and restore error codes. These error codes are not very descriptive (or not descriptive at all), but they offer context to the engineers who can map them to their actual meaning—in terms of actual implementation. This manual and static mapping requires developers to maintain it when new categories or combinations of errors arise, which can be prone to errors and incompleteness.

To avoid the need for manual error code mapping, I propose **a dynamic mapping that generates unique and deterministic error codes on the fly, encapsulating the context of an error in a single number (an error code)**.

Programmers will recognize the mechanism: it is a set of bit flags, which Go writes `1 << iota`. Structured error codes are not new either: a Windows `HRESULT` packs a severity bit, a facility and a code into 32 bits, and an SQL `SQLSTATE` is a two-character class followed by a three-character condition. What differs here is the use: there, each code is assigned and listed in a catalogue; here, only the error-items are listed, and the code of any combination is computed when the error occurs.

These dynamic error codes are contextual (they offer clues about the context of the errors), yet non-descriptive (they say nothing about that context to anyone who does not hold the list of error-items).

Cite as: Guillaume Lethuillier, "Contextual Yet Non-Descriptive Errors", glthr.com, October 24, 2023. https://glthr.com/contextual-yet-non-descriptive-errors
Interactive figures, animations and videos of the web page are shown here as still images or addresses.



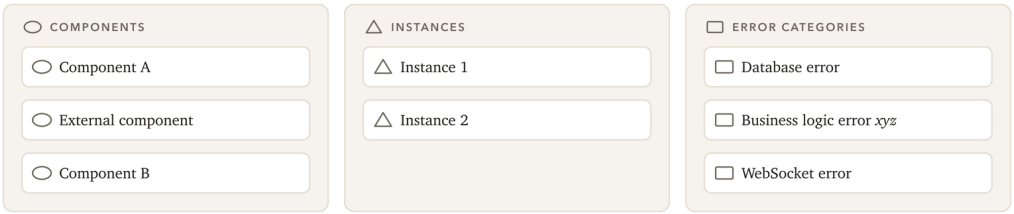
Identification of "Error-Items"

To begin with, I *define* a set of miscellaneous items—"error-items"—that can help identify the root cause of bugs, such as components, instances of the application (when the application knows their characteristics by the time an error object is generated), categories of errors, and so on. The developers arbitrarily define these error-items based on their perceived usefulness in debugging a specific issue. The purpose of these items is to quickly pinpoint the root cause of a bug; in other words, *to characterize the context of an error.*

Let's use a toy example to illustrate this idea.

A small application consists of two main components (or packages, or microservices, *etc.*), which can be instantiated several times. Additionally, I identify three categories of errors: WebSocket errors, some business logic errors, and database errors.

From the point of view of the definition of error details, this application contains a mixed bag of 8 error-items. I arbitrarily identified them as useful pieces of information that, when combined, can provide a relevant context to debug future bugs. I do not need to anticipate all combinations of items (or, to put it in another way, all possible error contexts): *this approach does it for me.*



Then, to represent each item in the error code, I start by assigning a unique number—a power of two—to each of them. These numbers are their identifiers.

Using binary manipulation, I create a one-hot vector for each item by left-shifting one bit by the position of the item in the list of error-items defined for the application (`1 << index`; in a Go list of constants, `iota` is this index). The first item will be represented as `0b1` (1), the second as `0b10` (2), the third as `0b100` (4), and so on.

```
type errorItem int64

const (
  ComponentA errorItem = 1 << iota // 1
```

```
  DatabaseError                    // 2
  Instance1                        // 4
  // ... list of error-items ...
)
```

These identifiers can alternatively be generated arithmetically, from the index i of each item:

$$id_i = 2^i$$

The error-items are now labelled.

INDEX	ERROR-ITEM	IDENTIFIER	BINARY
0	⬭ Component A	1	0000000**1**
1	▭ Database error	2	000000**1**0
2	△ Instance 1	4	00000**1**00
3	▭ Business logic error *xyz*	8	0000**1**000
4	⬭ External component	16	000**1**0000
5	▭ WebSocket error	32	00**1**00000
6	⬭ Component B	64	0**1**000000
7	△ Instance 2	128	**1**0000000

IDENTIFIER

`1 << i`

$= 2^i$

a power of two
one bit per item

This auto-generated mapping between error-items and their identifiers can be easily extended whenever a new feature is added to the application or when it becomes apparent that a new item needs to be defined. This can be done just by adding the new item to the list of constants, ensuring that all new items are properly represented in the error code.

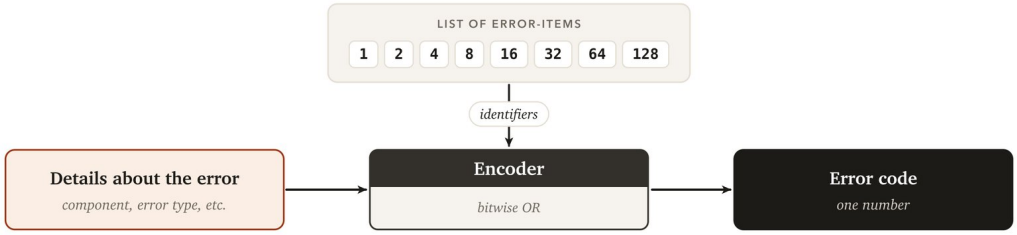
It should be noted that, for the system to be future-proof, obsolete items have to be disabled—for instance by using a special enum—instead of being removed, and identifiers of new items have to be greater than previous ones.

Code already issued	Meant	Read after adding "Cache error" first
7	Component A, Database error, Instance 1	Cache error, Component A, Database error
33	Component A, WebSocket error	Cache error, External component
208	External component, Component B, Instance 2	Business logic error xyz, WebSocket error, Component B

Added first, a new item moves every identifier up: the codes already issued name the wrong items.

Each error-item takes one bit of the code, so the size of the code bounds the number of items: with the `int64` used above, an application can define up to 64 error-items (63 if the code must stay positive, as the last bit is the sign bit). The toy example needs only 8.

Encoding the Error Context



When an error occurs in the application, its error code is computed through a bitwise `OR` operation on all selected error-items (basically, the items that compose the context of the error). This operation generates a unique number representing the union of items—as they are unique powers of two.

Let's consider an example.

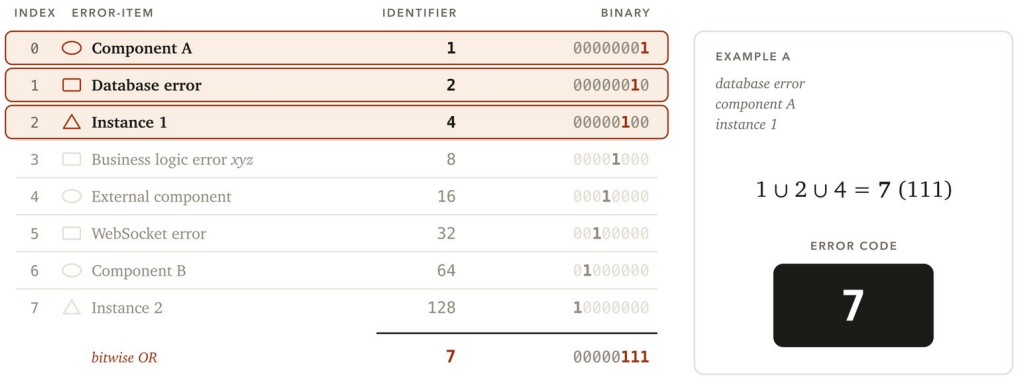
Suppose there are two items having identifiers `2` and `16`:

```
n := 2 | 16
fmt.Printf("%d <=> %b (%b | %b)\n", n, n, 2, 16)
// 18 <=> 10010 (10 | 10000)
```

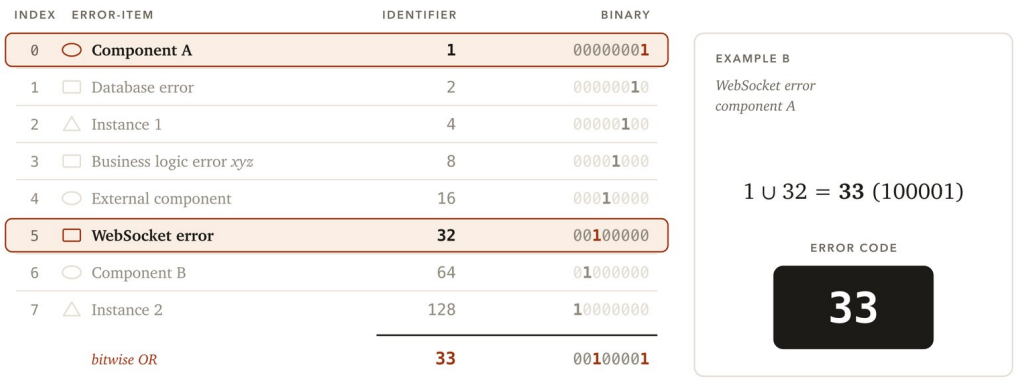
The result is a unique number (`18`) representing the union of the two singleton sets.

Here are additional illustrations of this idea, based on the toy example.

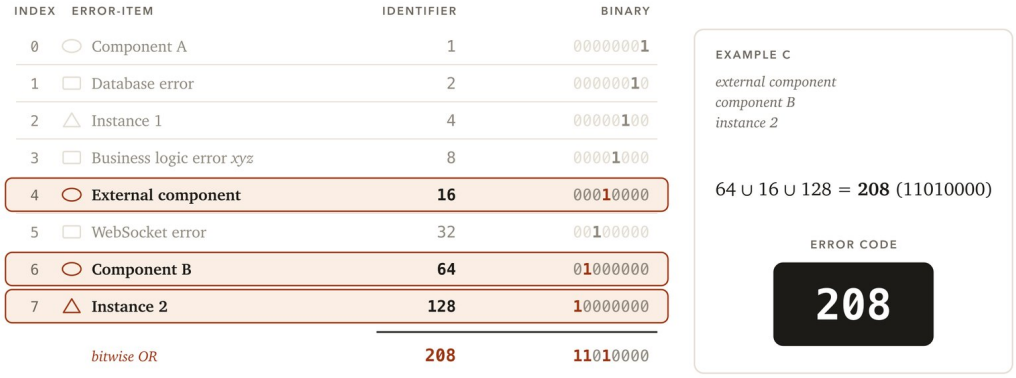
A) A database error returned by component A that occurred in the first instance of the application ⇒ `Component A` ∪ `Instance 1` ∪ `Database Error` ⇒ `7`



B) A WebSocket error returned by component A ⇒ `Component A` ∪ `WebSocket Error` ⇒ `33`

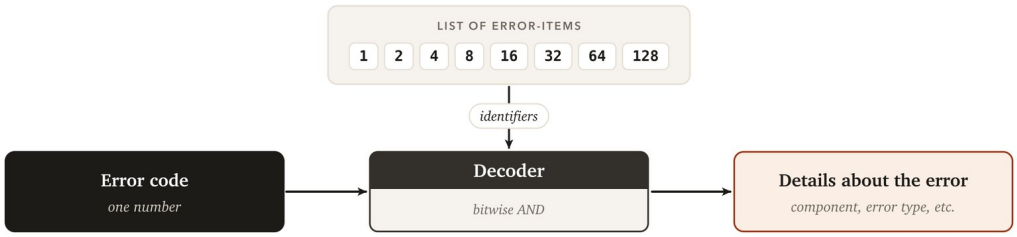


C) An error returned by an external component that occurred in component B of the second instance of the application ⇒ `Component B` ∪ `External component` ∪ `Instance 2` ⇒ 208



This operation is trivial: it simply corresponds to adding the identifiers of the error-items. What makes this process useful is that this addition can be *reversed* to extract its terms. In other words, **the contextual information transformed by the addition can be reconstructed ex-post**.

Decoding the Error Codes



To decode the error code, it suffices to perform the reverse operation (permitted by the fact that the identifiers are unique powers of two) by applying the `AND` operation between each identifier and the error code. If the operation results in the identifier

itself—that is: `(errorCode & identifier) == identifier`—this item belongs to the context of the error.

```
errorCode := 18
for _, identifier := range []int{1, 2, 4, 8, 16, 32, 64, 128} {
  belongsToErrorContext := (errorCode & identifier) == identifier
  fmt.Printf("%d: %v\n", identifier, belongsToErrorContext)
  // 1: false
  // 2: true   ✔
  // 4: false
  // 8: false
  // 16: true  ✔
  // 32: false
  // 64: false
  // 128: false
}
```

Because the identifiers are automatically mapped to the error-items, the reconstituted terms of the addition/`OR` operation are translated into a meaningful context.

Discussion

So, what's the point of these contextual yet non-descriptive errors?

These error codes lack helpful information for end-users, making it challenging for them to resolve the issue by themselves (when applicable). A user-friendly, customized error message is *always* preferable.

Furthermore, when investigating a bug, with this approach, engineers need to use a decoder that perfectly matches the encoder's representation of error-items.

I think that these errors can still be helpful for the following reasons.

1. **Complementary debugging device.** This approach provides an additional indicator that *complements* well-thought-out error messages (it does not aim at replacing them), making it easier for developers to identify and resolve issues without leaking details about the internal implementation—just compare `Username cannot be created` vs. `Username cannot be created (11)`. It offers a clue about the context of an error without having the developers examine multiple variables and logs while making it harder for attackers to reverse-engineer sensitive information—though it obscures the details rather than keeping them secret: anyone who holds the list of error-items can decode the code. And whoever can provoke errors at will can learn, code after code, which bit goes with which cause: the code is obfuscated, not secret. Where that matters, encrypt it with a key that stays on the server before showing it. (Decoded against the toy example, `11`, that is `8 | 2 | 1`, names Component A, a database error and the business logic error xyz.)

2. **Simplified error management.** Thanks to this algorithmic mapping, developers do not need to worry about managing error mapping, as this approach takes care of it automatically. This leads to increased flexibility in handling new combinations of existing items. For instance, imagine that the `Database`

`Error` is only used in combination with `Component A` (because this component is the only one using a database). Later, this application is updated so that `Component B` now uses a database. No need for the developers to define a new error-item. It suffices to automatically generate a new error code based on the `Component B` ∪ `Database Error` union. Programmatically, the developer just has to call the function that generates the code with the enums corresponding to component B and the database error type. In this respect, while the error-items have to be defined, the mapping, corresponding to combinations of items, is dynamic.

3. **Simplified logging.** When errors are propagated from one service to another, this approach simplifies logs by only transmitting the error code and potentially a customized error message, rather than duplicating the entire error object. In practice, in the context of a microservices architecture, people reviewing the logs of a service that propagates the error code can quickly identify the relevant log thanks to the contextual information encoded by the error code.

4. **Easy to implement.** Last but not least, the encoder and decoder are based on simple, stateless, and easy-to-understand operations, further simplifying the development process.

Why not a trace ID or a correlation ID, then, as many services already return? Such an ID is only a key: it says nothing by itself, and an engineer has to look it up in the logs, which supposes that the logs are kept long enough and can be reached. A contextual error code carries its context in the code itself: anyone with the decoder can read it from a screenshot sent by a user, without any log lookup. An ID remains the better choice when the whole history of a request matters (the exact time, the parameters, the sequence of calls across services), and nothing prevents an application from showing both.

How to cite this article

> Guillaume Lethuillier, "Contextual Yet Non-Descriptive Errors", glthr.com, October 24, 2023.
> https://glthr.com/contextual-yet-non-descriptive-errors

For BibTeX:

```
@misc{lethuillier2023contextual,
  author = {Lethuillier, Guillaume},
  title  = {Contextual Yet Non-Descriptive Errors},
  year   = {2023},
  month  = oct,
  url    = {https://glthr.com/contextual-yet-non-descriptive-errors},
  note   = {glthr.com}
}
```

This document was typeset with LaTeX from the article published at https://glthr.com/contextual-yet-non-descriptive-errors, which remains the reference version (as updated on November 14, 2026).