Failure is your Domain

Ben Johnson

June 4, 2018

Go's paradox is that error handling is core to the language yet the language doesn't prescribe how to handle errors. Community efforts have been made to improve and standardize error handling but many miss the centrality of errors within our application's domain. That is, your errors are as important as your `Customer` and `Order` types.

An error also must serve the different goals for each of its consumer roles—the application, the end user, and the operator. This post explores the purpose of errors for each of these consumers within our application and how we can implement a simple but effective strategy that satisfies each role's needs.

This post expands on many ideas about application domain & project design from Standard Package Layout [1] so it is helpful to read that first.

1 Why We Err

Errors, at their core, are simply a way of explaining why things didn't go how you wanted them to. Go splits errors into two groups—`panic` and `error`. A `panic` occurs when you don't expect something to go wrong such as accessing invalid memory. Typically, a panic is unrecoverable so our application fails catastrophically and we simply notify an operator to fix the bug.

An `error`, on the other hand, is when we expect something could go wrong. That's the focus of this post.

1.1 Types of errors

We can divide `error` into two categories—*well-defined errors* & *undefined errors*.

A well-defined error is one that is specified by the API such as an `IsNotExist` error returned from `os.Open()`. These allow us to manage our application flow because we know what to expect and can work with them on a case-by-case basis.

An undefined error is one that is undocumented by the API and therefore we are unable to thoughtfully handle it. This can occur from poor documentation but it can also occur when APIs we depend on add additional errors conditions after we've integrated our code with them.

2 Who Consumes Our Errors

The tricky part about errors is that they need to be different things to different consumers of them. In any given system, we have at least 3 consumer roles—*the application, the end user, & the operator.*

2.1 Application role

Your first line of defense in error handling is your application itself. Your application code can recover from error states quickly and without paging anyone in the middle of the night. However, application error handling is the least flexible and it can only handle well-defined error states.

An example of this is your web browser receiving a `301` redirect code and navigating you to a new location. It's a seamless process that most users are oblivious to. It's able to do this because the HTTP specification has well-defined *error codes.*

2.2 End user role

If your application is unable to handle the error condition then hopefully your end user can resolve the issue. Your end user can see an error state such as `"Your debit card is declined"` and is flexible enough to resolve it (i.e. deposit money in their bank account).

Unlike the application role, the end user needs a *human-readable message* that can provide context to help them resolve the error.

These users are still limited to well-defined errors since revealing undefined errors could compromise the security of your system. For example, a `postgres` error may detail query or schema information that could be used by an attacker. When confronted with an undefined error, it may be appropriate to simply tell the user to contact technical support.

2.3 Operator role

Finally, the last line of defense is the system operator which may be a developer or an operations person. These people understand the details of the system and can work with any kind of error.

In this role, you typically want to see as much information as possible. In addition to the error code and human-readable message, a *logical stack trace* can help the operator understand the program flow.

3 Our Baseline Application Error Type

Given our understanding that we need error codes, human-readable messages, & logical stack trace we can construct a simple error type to handle most of our application's errors such as the struct in Listing 1.

Listing 1: Error type definition

```
package myapp

// Error defines a standard application error.
type Error struct {
    // Machine-readable error code.
    Code    string

    // Human-readable message.
    Message string

    // Logical operation and nested error.
    Op      string
    Err     error
}
```

Note: This approach extends the implementation described in Error handling in Upspin [2] but with some important differences that we'll discuss later.

This is our baseline error type for our application. The `Code` and `Message` fields provide communication to our application and end user roles, respectively. The `Op` and `Err` fields allow us to chain errors together so that we can build the logical stack trace for our operator.

One important detail that is easy to miss is that our error is defined in our root package—`myapp.Error`. This is important because it becomes part of our domain language. It also avoids the stutter problem that's caused when defining an `errors` subpackage (e.g. `errors.Error`).

Every application is unique though so this can be extended further based on your application or business use case. You may also need to define additional specific error types but this works surprisingly well for most error scenarios.

4 Error Management by Role

Our `Error` type is a good start but we need to add some simple functionality to make it usable. Let's walk through various scenarios from the perspective of each of our consumer roles. In these following examples we'll assume an application, `myapp`, that has a `UserService` interface defined in its domain as shown in Listing 2.

Let's look and see how each of these roles can manage its error state.

4.1 Application role

Our application role is typically concerned with simple error codes. For example, if our program attempts to fetch a `User` by ID and it receives a `"not found"` error, it could reattempt by searching by an e-mail address.

Listing 2: User service interface

```
package myapp

// UserService represents a service for managing users.
type UserService interface {
    // Returns a user by ID.
    FindUserByID(ctx context.Context, id int) (*User, error)

    // Creates a new user.
    CreateUser(ctx context.Context, user *User) error
}
```

Listing 3: Error codes

```
// Application error codes.
const (
    ECONFLICT   = "conflict"   // action cannot be performed
    EINTERNAL   = "internal"   // internal error
    EINVALID    = "invalid"    // validation failed
    ENOTFOUND   = "not_found"  // entity does not exist
)
```

4.1.1 Defining our error codes

While it's tempting to build fine-grained error codes, it's much easier to manage more generic codes. There are several systems that define generic codes from which we can draw inspiration but two of my favorites are HTTP [3] & gRPC [4].

While these specifications contain numerous error codes, I try to start from a more humble set of codes and expand as needed. I find the codes in Listing 3 to be a good, simple starting point.

4.1.2 Translating codes to our domain

These error codes are specific to our application so when we are interacting with external libraries we must translate those errors to our domain's error codes. For example, if our application implements our `UserService` in Postgres as shown in Listing 4, we will need to translate a `sql.ErrNoRows` error to an `ENOTFOUND` code. Our domain model has no knowledge of `sql.ErrNoRows` and it would break down if we also implement `UserService` with a non-SQL database.

This allows us to return an `ENOTFOUND` error for our application to operate on independent of the implementation of `UserService`.

Listing 4: Postgres implementation of FindUser

```
package postgres

// FindUserByID returns a user by ID. Returns ENOTFOUND if user does
    not exist.
func (s *UserService) FindUserByID(ctx context.Context, id int) (*myapp
    .User, error) {
    var user myapp.User
    if err := s.QueryRowContext(ctx, `
        SELECT id, username
        FROM users
        WHERE id = $1
    `,
        id
    ).Scan(
        &user.ID,
        &user.Username,
    ); err == sql.ErrNoRows {
        return nil, &myapp.Error{Code: myapp.ENOTFOUND}
    } else if err {
        return nil, err
    }
    return &user, nil
}
```

4.1.3 Working with error codes effectively

At this point, however, we have two issues. First, our function returns `error` instead of `*myapp.Error` so we'll need to type assert whenever we want to access `Error.Code`. That's annoying. Second, our `Error.Err` field allows us to nest errors so our top-level error may not contain the error code and we'll need to recursively search for it.

We can solve both issues with a simple function that accomplishes the following:

1. Returns no error code for `nil` errors.
2. Searches the chain of `Error.Err` until a defined `Code` is found.
3. If no code is defined then return an internal error code (`EINTERNAL`).

The implementation can be found in Listing 5. We can now apply this in our calling code as shown in Listing 6.

4.2 End user role

Our end users expect actionable, human-readable messages. These can have additional constraints such as branding tone or internationalization but we'll just focus on the basics.

Listing 5: Recursive error code handling function

```
// ErrorCode returns the code of the root error, if available.
    Otherwise returns EINTERNAL.
func ErrorCode(err error) string {
    if err == nil {
        return ""
    } else if e, ok := err.(*Error); ok && e.Code != "" {
        return e.Code
    } else if ok && e.Err != nil {
        return ErrorCode(e.Err)
    }
    return EINTERNAL
}
```

Listing 6: Error code usage

```
user, err := userService.FindUserByID(ctx, 100)
if myapp.ErrorCode(err) == myapp.ENOTFOUND {
    // retry another method of finding our user
} else if err != nil {
    return err
}
```

4.2.1 Example usage

A perfect example of end user messaging is for field validation. In Listing 7 we check to ensure that new users in our `UserService` have a username and it is unique.

4.2.2 Working with error messages effectively

Our error messages pose similar issues to our error codes above. We need a utility function to extract messages from `error` values. We can implement a function similar to `ErrorCode()` except with the following rules:

1. Returns no error message for `nil` errors.
2. Searches the chain of `Error.Err` until a defined `Message` is found.
3. If no message is defined then return a generic error message.

The implementation is shown in Listing 8. Now we can show this to our users if there is an error as shown in Listing 9.

4.3 Operator role

Finally, we need to be able to provide all this information plus a logical stack trace to our operator so they can debug issues. Go already provides a simple method, `error.Error()`, to print error information so we can utilize that.

Listing 7: Postgres implementation of CreateUser

```
package postgres

// CreateUser creates a new user in the system.
// Returns EINVALID if the username is blank or already exists.
// Returns ECONFLICT if the username is already in use.
func (s *UserService) CreateUser(ctx context.Context, user *myapp.User)
     error {
    // Validate username is non-blank.
    if user.Username == "" {
        return &myapp.Error{Code: myapp.EINVALID, Message: "Username is
             required."}
    }

    // Verify user does not already exist.
    if s.usernameInUse(user.Username) {
        return &myapp.Error{
            Code: myapp.ECONFLICT,
            Message: "Username is already in use. Please choose a
                different username.",
        }
    }

    ...
}
```

Listing 8: Recursive error message handling function

```
// ErrorMessage returns the human-readable message of the error, if
    available.
// Otherwise returns a generic error message.
func ErrorMessage(err error) string {
    if err == nil {
        return ""
    } else if e, ok := err.(*Error); ok && e.Message != "" {
        return e.Message
    } else if ok && e.Err != nil {
        return ErrorMessage(e.Err)
    }
    return "An internal error has occurred. Please contact technical
        support."
}
```

Listing 9: Error message usage

```
if msg := ErrorMessage(err); msg != "" {
    fmt.Printf("ERROR: %s\n", msg)
}
```

Listing 10: Operator error message generation

```
// Error returns the string representation of the error message.
func (e *Error) Error() string {
    var buf bytes.Buffer

    // Print the current operation in our stack, if any.
    if e.Op != "" {
        fmt.Fprintf(&buf, "%s: ", e.Op)
    }

    // If wrapping an error, print its Error() message.
    // Otherwise print the error code & message.
    if e.Err != nil {
        buf.WriteString(e.Err.Error())
    } else {
        if e.Code != "" {
            fmt.Fprintf(&buf, "<%s> ", e.Code)
        }
        buf.WriteString(e.Message)
    }
    return buf.String()
}
```

4.3.1 Logical stack traces

Many operators are familiar with stack traces. They dump a list of every function in the call stack from where an error occurred. You can see this at work when you call `panic()`.

However, many times stack traces can be overwhelming and we only need a small subset of those lines to understand the context of our error. A logical stack trace contains only the layers that we as developers find to be important in describing the program flow. We will accomplish this by using the `Op` and `Err` fields to wrap errors to provide context.

4.3.2 Implementing Error()

Our `myapp.Error.Error()` function should return an error string suitable for operators. There's no definitive standard for how to format this message but I format mine with these goals in mind:

- Show the logical stack trace first. It provides context for the rest of the message. It also allows us to sort error lines to group them together.
- Show `Code` & `Message` at the end.
- Print on a single line so it's easy to grep.

This implementation in Listing 10 assumes that `Err` cannot coexist with `Code` or `Message` on any given error. I find this to work well in practice.

Let's look at how we use this with an example below.

Listing 11: Postgres implementation of CreateUser

```
// CreateUser creates a new user in the system with a default role.
func (s *UserService) CreateUser(ctx context.Context, user *myapp.User)
     error {
    const op = "UserService.CreateUser"

    // Perform validation...

    // Insert user record.
    if err := s.insertUser(ctx, user); err != nil {
        return &myapp.Error{Op: op, Err: err}
    }

    // Insert default role.
    if err := s.attachRole(ctx, user.ID, "default"); err != nil {
        return &myapp.Error{Op: op, Err: err}
    }
    return nil
}
```

Listing 12: User insertion helper function

```
// insertUser inserts the user into the database.
func (s *UserService) insertUser(ctx context.Context, user *myapp.User)
     error {
    const op = "insertUser"
    if _, err := s.db.Exec(`INSERT INTO users...`); err != nil {
        return &myapp.Error{Op: op, Err: err}
    }
    return nil
}
```

4.3.3 Example usage

Returning to our `CreateUser()` function example, suppose we need to create additional roles for our new users. We can utilize the `Op` and `Err` fields in our application `Error` to wrap this nested functionality. This is shown in Listing 11, 12, and 13.

Let's assume we receive a Postgres syntax error inside our `attachRole()` function and Postgres returns the message shown in Listing 14.

Without context, we don't know if this occurred in our `insertUser()` function or our `attachRole()` function. This is very tedious to debug when your API executes 20+ SQL queries. However, because we are wrapping our errors, `Error()` will provide us with a logical stack trace. This lets us narrow down the errant query and make the fix.

Listing 13: Role attachment helper function

```
// attachRole inserts a role record for a user in the database
func (s *UserService) attachRole(ctx context.Context, id int, role
    string) error {
    const op = "attachRole"
    if _, err := s.db.Exec(`INSERT roles...`); err != nil {
        return &myapp.Error{Op: op, Err: err}
    }
    return nil
}
```

Listing 14: Error message with and without context

```
Without Context:
syntax error at or near "INSERT"

With Context:
UserService.CreateUser: attachRole: syntax error at or near "INSERT"
```

5 Comparison with other approaches

This post describes how I work with errors in my applications and it's an approach that has largely been inspired by the work of others. However, this approach does not take any one of these approaches wholesale but rather combines the best of each one. I think it is helpful to discuss where these approaches diverge and why.

5.1 Alternative: Upspin error handling

The Upspin team released a blog post called Error handling in Upspin [2] that inspired much of my approach. They use a subpackage called `errors` with a single concrete `Error` type with an error code (`Kind`) and error wrapping (`Op` & `Err`). They also include a builder function called `errors.E()` for constructing their errors by using type information in the arguments.

I found the error codes & wrapping to be a great, simple implementation but I found issues with the following in practice:

- Moving the `Error` type to the `errors` package removes it from the domain and causes a stuttering name (`errors.Error`).
- The additional types such as `Op` & `Kind` seemed more complicated than they needed to be and seemed to exist for syntactic sugar for the `errors.E()` builder function. Strings would suffice if instantiating `Error` types directly.
- There lacked a separation between end user error messaging and operator error messaging. This makes sense because Upspin's end users are likely

operators but this model doesn't fit for most applications.

5.2 Alternative: pkg/errors

The pkg/errors [5] project is an effort to provide error wrapping as a library. This approach handles error wrapping well and provides much needed context to errors, however, I found the following issues in usage:

- By importing your error handling from a third party, it is external to your application's domain.
- The error wrapping allows developers to obtain root cause error information but still requires verbose type assertions to extract this information. The `ErrorCode()` function from this post simplifies this error information checking to a single line in the caller code.
- It solves the error wrapping problem but does not help with other error handling concerns.

5.3 Feedback

These critiques are not meant to be slams against these other projects. No approach works for every application so I've added these comparisons to help developers make educated decisions about their application design. Please choose the approach that is right for you.

6 Conclusion

Error handling is a critical piece of your application design and is complicated by the variety of different consumer roles that require error information. By considering error codes, error messages, and logical stack traces in our design we can fulfill the needs of each consumer. By integrating our `Error` into our domain, we give all parts of our application a common language to communicate about when unexpected things happen.

References

[1] Standard Package Layout, `https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1`

[2] Error handling in Upspin, `https://commandcenter.blogspot.com/2017/12/error-handling-in-upspin.html`

[3] HTTP/1.1: Status Code Definitions, `https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

[4] gRPC Canonical Error Codes, https://github.com/grpc/grpc-go/blob/v1.12.0/codes/codes.go

[5] pkg/errors, https://github.com/pkg/errors