1. **Explicit Error Handling:** In Go, functions that can encounter an error return an error value, and it is expected that the caller will check and handle this error immediately. This explicit error handling ensures that errors are not overlooked.

There is no try catch error handling provision in Go to check for errors instead you use the if checks to check if a function call returned an error or not.

function in Go can return multiple values

if the function is returning an error it should be the last return value,

return the zero value for all other return values   if there are any.

**2. Adding Context to Errors:** To make errors more informative, I use the fmt.Errorf function to add context to the error messages. This helps in diagnosing issues when they are logged or returned to the client.

**Example Explanation:** If an API endpoint needs to fetch data from a database, I wrap any errors returned from the database query with additional context using fmt.Errorf. This might include details like the operation that failed and any relevant parameters.

**3. Using Custom Error Types:** For more complex error handling scenarios, I define custom error types that implement the error interface. This allows me to provide more detailed information about the error and handle specific error cases more precisely.

**Example Explanation:** In an API, if I need to differentiate between different kinds of errors (e.g., validation errors vs. database errors), I create custom error types. For instance, a ValidationError type could include information about which field is invalid and why.

A computer screen shot of a program code

Description automatically generated A screenshot of a computer error

Description automatically generated A screenshot of a computer program

Description automatically generated

**4. Sentinel Errors:** For common error conditions, I use sentinel errors, which are predefined error values. These sentinel errors can be compared directly using equality operators.

**Example Explanation:** In an API, if there are common error conditions such as "resource not found," I define sentinel errors like ErrNotFound. This makes it easy to handle these errors consistently across different parts of the application.

**5. Propagating Errors:** I propagate errors up the call stack by returning them from functions and adding context as needed. This approach ensures that the caller has all the information necessary to handle the error appropriately.

**Example Explanation:** When an error occurs in a lower-level function, such as a database query, I return the error to the API handler function. The handler can then decide whether to return a specific HTTP status code, log the error, or both.

**6. Graceful Error Handling and User Feedback:** In the context of an API, it’s crucial to convert internal errors into appropriate HTTP responses. This means mapping different types of errors to relevant HTTP status codes and returning meaningful error messages to the client.

**Example Explanation:** In my API handler functions, I translate internal errors into HTTP responses. For example, if a database query returns an ErrNotFound error, I convert this into a 404 Not Found response. If a validation error occurs, I return a 400 Bad Request response with details about the validation failure.

**Summary:**

* **Explicit Error Handling:** Always check and handle errors immediately.
* **Adding Context:** Use fmt.Errorf to provide additional context.
* **Custom Error Types:** Define custom error types for complex scenarios.
* **Sentinel Errors:** Use predefined errors for common conditions.
* **Propagating Errors:** Return errors up the call stack with added context.
* **Graceful Error Handling:** Map errors to appropriate HTTP responses in an API context.