ERROR HANDLING IN GO

Error Handling in Go: A Complete Guide

Error handling is the process of anticipating, identifying, and managing errors or exceptions that can occur during the execution of a program or system, ensuring that the application remains stable and user-friendly

Error handling in Go is **explicit** and follows a **multiple return value** pattern rather than exceptions (like in Java/Python).

◇ **Basics of Error Handling in Go :** The error type is a built-in interface in Go:

```
type error interface {
   Error() string
}
```

✅ **Example: Function Returning an Error**

```
package main
import (
	"errors"
	"fmt"
)
func divide(a, b int) (int, error) {
	if b == 0 {
		return 0, errors.New("cannot divide by zero")
	}
	return a / b, nil
}
func main() {
	result, err := divide(10, 0)
	if err != nil {
		fmt.Println("Error:", err)
	} else {
		fmt.Println("Result:", result)
	}
```

```
}
```

✅ **Output:** Error: cannot divide by zero

◇ Using fmt.Errorf() for Formatted Errors

fmt.Errorf() allows formatted error messages.

✅ **Example:**

```
import "fmt"

func getFile(name string) error {
	return fmt.Errorf("file %s not found", name)
}
```

◇ Using errors.Is() and errors.As() for Wrapped Errors

Go 1.13+ supports **error wrapping**.

✅ **Example: Wrapping & Unwrapping Errors**

```
import (
	"errors"
	"fmt"
)
var ErrNotFound = errors.New("item not found")

func findItem(id int) error {
	return fmt.Errorf("DB error: %w", ErrNotFound) // Wrap error
}
func main() {
	err := findItem(42)
	// Check if the error is of type ErrNotFound
	if errors.Is(err, ErrNotFound) {
		fmt.Println("Handling not found error:", err)
	}
```

```
}
```

✅ **Output:** Handling not found error: DB error: item not found

---

◇ panic and recover (Handling Critical Errors)

- panic stops execution and unwinds the stack.
- recover catches panics to prevent crashes.

✅ **Example:**

```
package main

import "fmt"

func safeDivide(a, b int) {
	defer func() {
		if r := recover(); r != nil {
			fmt.Println("Recovered:", r)
		}
	}()
	if b == 0 {
		panic("division by zero!")
	}
	fmt.Println(a / b)
}
func main() {
	safeDivide(10, 0)
	fmt.Println("Program continues after panic recovery.")
}
```

✅ **Output:**

```
Recovered: division by zero!
Program continues after panic recovery.
```

◇ Custom Errors with struct and Error() Method

✅ **Example:**

```
type MyError struct {
	Code    int
	Message string
}
func (e MyError) Error() string {
	return fmt.Sprintf("Error %d: %s", e.Code, e.Message)
}
func doSomething() error {
	return MyError{404, "Resource Not Found"}
}
func main() {
	err := doSomething()
	fmt.Println(err)
}
```

✅ **Output:**

```
Error 404: Resource Not Found
```

🔹 Summary: When to Use What?

Method	Usage
errors.New()	Simple error creation
fmt.Errorf()	Formatted error messages
errors.Is()	Check specific errors
errors.As()	Type assertion for custom errors
panic/recover	Critical error handling

Final Thoughts

- **Use errors for normal failures.**
- **Reserve panic for unrecoverable conditions (e.g., database corruption).**
- **Use errors.Is() for error comparison.**

REAL-WORLD EXAMPLE OF ERROR HANDLING IN GO

Real-World Example: API Call with Error Handling

Scenario:

You're building a Go-based **REST API client** that fetches user data. The API might:

- Return a **404 (Not Found)**
- Have **network issues**
- Return **invalid JSON**

We'll handle these cases properly!

---

1 Full Code with Error Handling

```
package main
import (
	"encoding/json"
	"errors"
	"fmt"
	"io"
	"net/http"
	"time"
)

// Define a custom error type
type APIError struct {
	Code    int
	Message string
```

```
}

func (e APIError) Error() string {
	return fmt.Sprintf("API Error - Code: %d, Message: %s", e.Code, e.Message)
}

// User struct to hold API response
type User struct {
	ID    int    `json:"id"`
	Name  string `json:"name"`
	Email string `json:"email"`
}

// fetchUser makes an API call and handles errors
func fetchUser(userID int) (*User, error) {
	url := fmt.Sprintf("https://jsonplaceholder.typicode.com/users/%d", userID)

	// Create an HTTP client with timeout
	client := &http.Client{Timeout: 5 * time.Second}
	resp, err := client.Get(url)

	// Handle network errors
	if err != nil {
		return nil, fmt.Errorf("network error: %w", err)
	}
	defer resp.Body.Close()

	// Handle API response errors
	if resp.StatusCode == http.StatusNotFound {
		return nil, APIError{Code: 404, Message: "User not found"}
```

```
	} else if resp.StatusCode != http.StatusOK {
		return nil, APIError{Code: resp.StatusCode, Message: "Unexpected API
error"}
	}

	// Read and parse response body
	body, err := io.ReadAll(resp.Body)
	if err != nil {
		return nil, fmt.Errorf("failed to read response: %w", err)
	}

	var user User
	err = json.Unmarshal(body, &user)
	if err != nil {
		return nil, fmt.Errorf("JSON parsing error: %w", err)
	}

	return &user, nil
}

// Main function
func main() {
	user, err := fetchUser(1)
	if err != nil {
		if errors.As(err, &APIError{}) {
			fmt.Println("Handled API error:", err)
		} else {
			fmt.Println("Unhandled error:", err)
		}
	} else {
		fmt.Printf("User found: %+v\n", user)
```

```
        }
}
```

🔥 **Explanation of Error Handling in Code**

✅ **Network Error Handling**

- fmt.Errorf("network error: %w", err) → Wraps network errors for debugging.

✅ **API Error Handling**

- Custom APIError struct with Error() method.
- Uses errors.As(err, &APIError{}) to check for API-related errors.

✅ **JSON Parsing Error Handling**

- json.Unmarshal() can fail if the response format is incorrect.
- We wrap it using fmt.Errorf("JSON parsing error: %w", err).

✅ **Graceful Fallback in main()**

- Uses if errors.As(err, &APIError{}) {} to **differentiate API vs. unexpected errors**.

Output Scenarios

✅ **Success Case**

User found: {ID:1 Name:"Leanne Graham" Email:"leanne@example.com"}

❌ **API Error (User Not Found)**

Handled API error: API Error - Code: 404, Message: User not found

❌ **Network Failure**

Unhandled error: network error: Get "https://jsonplaceholder.typicode.com/users/1": dial tcp: lookup failed

❌ **Invalid JSON**

Unhandled error: JSON parsing error: unexpected end of JSON input

---

Summary

- ✅ **Wrap errors using fmt.Errorf("%w") for better debugging.**
- ✅ **Use errors.As() for structured error handling.**
- ✅ **Always handle network, API, and JSON parsing errors gracefully.**