Sistema de Gestión de Libros Electrónicos en Go

Objetivo General del Programa

Desarrollar una aplicación de consola en el lenguaje Go que permita a los usuarios:

- Registrar nuevos libros electrónicos.
- Listar todos los libros almacenados.
- Buscar libros por título y autor.
- Actualizar la información de un libro.
- Eliminar libros del sistema con confirmación.

Todo esto usando principios de la Programación Orientada a Objetos, manejo de errores y diseño modular.

Bloque de Importación

```
import (
    "bufio"
    "errors"
    "fmt"
    "os"
    "strconv"
    "strings"
)
```

Explicación:

- `bufio`: permite leer texto de entrada (desde consola) de forma eficiente.
- `errors`: proporciona herramientas para crear y devolver errores personalizados.
- `fmt`: ofrece funciones para imprimir y formatear texto en la consola.
- `os`: permite acceder a recursos del sistema operativo, como la entrada estándar (`os.Stdin`).
- `strconv`: convierte cadenas de texto a tipos numéricos y viceversa (como convertir el año ingresado a `int`).
- `strings`: contiene funciones útiles para trabajar con texto, como `ToLower` o `Contains`.

Estructura `Libro` y Encapsulación

```
type Libro struct {
    ID        int
    titulo    string
    autor     string
    categoria string
    anio      int
}
```

Explicación:

- Define la estructura de un libro con atributos.
- Los campos `titulo`, `autor`, `categoria` y `anio` están **encapsulados** porque inician con minúscula (no accesibles desde fuera del paquete).

Para acceder a estos campos de manera segura, se utilizan métodos públicos:

```
func (l Libro) Titulo() string         { return l.titulo }
func (l *Libro) SetTitulo(t string)    { l.titulo = t }
```

- `Titulo()` devuelve el valor del campo privado `titulo`.
- `SetTitulo(t)` modifica el valor del campo `titulo` de forma controlada.

Interfaz `ServicioLibro`

```
type ServicioLibro interface {
    Listar() []Libro
    Buscar(titulo, autor string) []Libro
    Agregar(libro Libro) error
    Actualizar(id int, libro Libro) error
    Eliminar(id int) error
}
```

Explicación: Define un contrato que debe cumplir cualquier servicio que gestione libros. Este patrón desacopla la lógica de negocio del almacenamiento. Así, podrías reemplazar la implementación actual en memoria por una en base de datos sin modificar el resto del programa.

Implementación en Memoria: `MemoriaLibros`

```
type MemoriaLibros struct {
    datos    []Libro
    ultimoID int
}
```

Explicación:

- `datos` es un slice que simula una base de datos.

- `ultimoID` se usa para asignar identificadores únicos automáticamente.

Ejemplo de función `Agregar()`:

```
func (m *MemoriaLibros) Agregar(libro Libro) error {
    if libro.titulo == "" || libro.autor == "" {
        return errors.New("título y autor son obligatorios")
    }
    m.ultimoID++
    libro.ID = m.ultimoID
    m.datos = append(m.datos, libro)
    return nil
}
```

- Valida que los campos obligatorios no estén vacíos.
- Asigna un nuevo ID automáticamente.
- Agrega el libro al slice.
- Devuelve `nil` si todo fue correcto o un `error` si hay problemas.

Menú Interactivo (`main()`)

```
fmt.Println("1. Listar libros")
fmt.Println("2. Buscar libros")
fmt.Println("3. Registrar nuevo libro")
```

- Muestra opciones al usuario.
- Se usa `bufio.NewScanner(os.Stdin)` para capturar lo que escribe el usuario.

```
scanner := bufio.NewScanner(os.Stdin)
scanner.Scan()
opcion := scanner.Text()
```

- Luego se utiliza un `switch` para ejecutar la acción correspondiente:

```
switch opcion {
case "1":
    libros.Listar()
case "3":
    libros.Agregar(...)
```

Función `leerLibro()`

```
func leerLibro(scanner *bufio.Scanner) Libro {
    fmt.Print("Título: ")
    scanner.Scan()
    t := scanner.Text()
    ...
```

```
    return Libro{titulo: t, autor: a, categoria: c, anio: año}
}
```

Explicación:

- Solicita los datos del libro.
- Lee cada valor del usuario (título, autor, etc.).
- Construye y devuelve una instancia del tipo `Libro`.

Manejo de errores

Las operaciones que podrían fallar (como agregar, actualizar o eliminar) devuelven errores explícitos.

```
if err := libros.Agregar(libro); err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Libro registrado correctamente.")
}
```

Este patrón permite manejar errores de forma clara y robusta, sin que el programa se detenga inesperadamente.

Futura Actualización Web

El diseño del programa facilita su migración a plataforma web:

- La lógica de negocio ya está separada.
- Las funciones como `Agregar` o `Listar` pueden adaptarse como handlers HTTP.
- Se pueden usar templates HTML en lugar de la consola.
- El almacenamiento puede evolucionar hacia una base de datos (por ejemplo, MySQL).
- Su estructura permite crecer hacia una solución web profesional con mínima reestructuración.