Go 언어 & Gin 네트워크 프로그래밍

1. 왜 백엔드에 Go 언어인가?

압도적인 성능

가상 머신(VM) 없이 기계어로 직접 컴파일되어 C/C++에 준하는 매우 빠른 실행 속도를 제공합니다.

메모리 효율

수십 MB의 적은 메모리 점유율로도 서버 구동이 가능하여 클라우드 인프라 유지 비용을 획기적으로 줄여줍니다.

강력한 동시성

고루틴(Goroutine)을 통해 수만 개의 트래픽 요청을 비동기적으로 가볍고 안전하게 처리합니다.

2. 설치 및 환경 셋팅

- **공식 패키지 다운로드:** golang.org 에 접속하여 운영체제에 맞는 패키지를 설치합니다.
- **설치 확인:** 터미널을 열고 go version을 입력하여 go1.26.0 windows/amd64와 같이 출력되는지 점검합니다.
- **IDE 설정:** Visual Studio Code 설치 후 공식 'Go' 확장 프로그램을 설치합니다. 자동 포맷팅(gofmt)을 활용해 협업 효율을 높입니다.
- **작업 영역:** 원하는 위치에 프로젝트 폴더를 생성하고 터미널을 통해 해당 경로로 진입하여 개발을 시작합니다.

3. 프로젝트 초기화 (go mod)

의존성 관리 시스템

Node.js의 npm처럼 Go의 패키지와 버전을 관리합니다.

- go mod init [모듈명]: 프로젝트를 초기화합니다.
- go.mod: 외부 라이브러리 목록이 저장됩니다.
- go.sum: 패키지의 무결성 해시를 보장하여 공급망 공격(변조)을 방어합니다.

```
$ mkdir secure-go-api
$ cd secure-go-api # Go 모듈 초기화
$ go mod init secure-go-api go: creating new go.mod:
module secure-go-api
```

4. 첫 번째 코드 (Hello World)

프로그램 진입점

실행 가능한 Go 프로그램은 반드시 main 패키지에 main() 함수를 가져야 합니다.

- 문장 끝에 세미콜론(;)을 사용하지 않습니다.
- fmt 패키지를 임포트하여 표준 입출력을 다룹니다.
- go run main.go 명령어로 즉시 컴파일 후 실행합니다.

```
package main import
"fmt" func main() {
// 콘솔에 문자열 출력
fmt.Println("Hello, Go Backend!")
}
```

5. 변수 선언 (Variables)

정적 타입과 타입 추론

Go는 강타입 언어지만, 강력한 타입 추론을 통해 코드를 간결하게 유지합니다.

- var 키워드를 이용한 명시적 선언.
- := (짧은 선언)를 사용하여 var와 타입을 모두 생략 (실무 표준).
- **주의:** 선언하고 사용하지 않은 변수가 있으면 **컴파일 에러**가 발생합니다. (메모리 낭비 원천 차단)

```
func main() {
// 1. 명시적 선언
var name string = "Alice"
// 2. 타입 추론 (타입 생략)
var email = "admin@test.com"
// 3. 짧은 선언 (함수 내에서만 사용 가능)
role := "admin"
isActive := true
}
```

6. 상수 (Constants)

불변의 데이터

실행 중 절대 변경되어서는 안 되는 값을 선언할 때 사용합니다.

- 상수는 짧은 선언(:=)을 사용할 수 없습니다.
- 대문자로 시작하면 외부 패키지에서 접근 가능합니다.
- 비밀키, 환경설정 키, API URL 고정값 등에 주로 사용됩니다.

```
package main
import "fmt"
// 전역 레벨 상수 (Exported)
const SecretKey = "super_secret_123!"
func main() {
const pi = 3.14159
// pi = 3.14
// 값 변경 시 컴파일 에러 발생
fmt.Println(SecretKey, pi)
}
```

7. 기본 데이터 타입 (Data Types)

- **bool:** 논리형 타입으로 true 또는 false 값을 가집니다.
- **string:** 문자열 타입입니다. 반드시 쌍따옴표("")를 사용하여 선언합니다.
- **int / int64:** 정수형 타입입니다. OS 아키텍처에 따라 크기가 동적으로 결정됩니다.
- **float64:** 실수형(부동소수점) 타입입니다.
- **byte:** uint8의 별칭입니다. 파일 I/O나 네트워크 바이트 스트림 처리에 필수적입니다.
- **rune:** int32의 별칭입니다. 유니코드 글자 하나(예: 한글)를 완벽히 표현할 때 사용합니다.

8. 제로 값 (Zero Values)

초기화 없는 변수의 안전성

변수를 선언하고 값을 넣지 않으면 메모리에 쓰레기값을 두지 않고, 타입에 맞는 **기본값**을 자동 할당합니다.

- int, float: 0
- string: "" (빈 문자열)
- bool: false
- 포인터, 맵, 슬라이스: nil

```
func main() {
var count int
var title string
var isActive bool
var data map[string]int
// 출력: 0, "", false, map[]
fmt.Printf("count: %d\ntitle: '%s'\n", count, title)
fmt.Printf("active: %t\ndata: %v\n", isActive, data)
}
```

9. 실습 랩 (Lab 1)

Lab 1

변수와 기초 포맷팅

미션: 유저 프로필 출력기 만들기

이름(string), 나이(int), 관리자여부(bool) 변수를 짧은 선언(:=)으로 만드세요. 그 후 fmt.Printf를 사용하여 한 줄로 콘솔에 깔끔하게 출력하세요.

[힌트]
문자열은 %s, 정수는 %d, 불리언은 %t 포맷 식별자를 사용합니다.

10. 조건문 (If / Else)

간결한 조건식

조건식을 감싸는 괄호 ()를 사용하지 않아 코드가 깔끔합니다.

- 여는 중괄호 {는 반드시 조건식과 같은 줄에 있어야 합니다.
- else if 와 else 역시 닫는 중괄호 }와 같은 줄에서 시작해야 컴파일러가 인식합니다.

```
func main() {
	score := 85
	if score ⩾ 90 {
		fmt.Println("A등급")
	} else if score ⩾ 80 {
		fmt.Println("B등급")
	} else {
		fmt.Println("C등급")
	}
}
```

11. 💡 중요: 사전 실행문 (Pre-statement)

변수 스코프 좁히기

if문 안에서 변수를 초기화하고 바로 조건을 검사하는 Go 특유의 강력하고 안전한 패턴입니다.

- 초기화된 변수는 if 블록 내부에서만 존재합니다.
- 외부 스코프의 변수 오염을 막아주어 에러 처리 시 가장 널리 쓰이는 표준 패턴입니다.

```
func main() {
// 변수 l은 if 블록 안에서만 유효합니다
if l := len("secret"); l < 8 {
      fmt.Printf("비밀번호 길이(%d)가 짧습니다.", l)
}
// fmt.Println(l)
// 🚨 밖에서 호출하면 컴파일 에러!
}
```

12. 에러 처리의 정석 (Error Handling)

명시적 에러 체크

Go는 try-catch 예외 처리 방식이 없습니다. 함수는 정상 결과와 error 객체를 함께 반환합니다.

- 에러를 무시하지 않고 반환 즉시 err != nil 로 검사합니다.
- 문제가 생기면 조기 종료(Early Return)하는 것이 Go 언어의 강력한 보안 철학입니다.

```
import "strconv"
func main() {
	// 다중 반환값과 사전 실행문 결합
	if val, err := strconv.Atoi("123a"); err ≠ nil {
		fmt.Println("변환 실패:", err)
		return
	// 🚨 조기 종료
	}
	else {
		fmt.Println("변환 성공:", val)
	}
}
```

13. 사용자 정의 에러 만들기

errors 패키지 활용

개발자가 직접 비즈니스 로직에 맞는 에러 메시지를 생성하여 반환할 수 있습니다.

- errors.New() 함수를 사용하여 새로운 에러 객체를 만듭니다.
- 함수의 반환 타입에 error 인터페이스를 명시해야 합니다.

```
import (
"errors" "fmt"
)
func validateAge(age int) error {
      if age < 18 {
      // 비즈니스 에러 생성 리턴
            return errors.New("미성년자는 가입할 수 없습니다")
      }
      return nil
      // 정상일 때는 nil 리턴
}
```

14. 유일한 반복문 (For)

단순함의 미학

Go는 while이나 do-while 키워드가 없습니다. 오직 for만 사용하여 모든 반복을 처리합니다.

- 초기화, 조건, 증감을 모두 쓰면 전통적인 for문.
- 조건식만 쓰면 while문.
- 모두 생략하면 무한 루프가 됩니다.

```
for 루프 for i := 0; i < 3; i++ {
      fmt.Println(i)
}
// 2. while 처럼 사용
n := 1
for n < 5 {
      n *= 2
}
// 3. 무한 루프 (웹 리스너 등에 사용) //
for {
      ...
}
```

15. 실습 랩 (Lab 2)

Lab 2

제어문 훈련

미션: 짝수만 더하기

1부터 10까지 1씩 증가하며 반복하는 for문을 작성하세요. 내부에 if문을 사용하여 현재 숫자가 '짝수'일 때만 외부에 선언된 sum 변수에 누적해서 더하세요. 최종 sum 값을 출력하세요.

[힌트]

짝수 조건은 i % 2 == 0 연산자를 활용합니다.

16. 배열과 슬라이스

크기 가변성

데이터의 집합을 다루는 두 가지 방식입니다.

- **배열 (Array):** [3]int 처럼 크기가 고정되어 있어 실무에서는 활용도가 낮습니다.
- **슬라이스 (Slice):** []int 로 선언하며 크기가 동적으로 늘어납니다. (Python의 List)
- append() 함수를 통해 안전하게 요소를 추가합니다.

```
func main() {
	// 슬라이스 초기화 (대괄호 안이 비어있음)
	var slice []int = []int{1, 2}
	// append를 통한 데이터 추가
	// 내부적으로 용량이 부족하면 새 배열을 만들어 복사함
	slice = append(slice, 3)
	slice = append(slice, 4, 5)
	fmt.Println(slice)
	// [1 2 3 4 5]
}
```

17. Make와 메모리 용량

슬라이스의 내부 구조

슬라이스는 내부적으로 실제 데이터를 담는 숨겨진 '배열'을 가리키는 구조체입니다.

- 요소가 추가될 때 내부 배열의 용량(Capacity)이 초과되면, 더 큰 새 배열을 할당하고 복사하는 비용이 발생합니다.
- make 함수로 미리 용량을 넉넉히 확보하면 성능을 최적화할 수 있습니다.

```
func main() {
// 타입, 길이(Length), 용량(Capacity) 명시
s := make([]int, 0, 5)
// 출력: len: 0, cap: 5
fmt.Printf("len: %d, cap: %d\n", len(s), cap(s))
s = append(s, 1, 2, 3)
// 메모리 재할당 없이 길이가 3으로 늘어남
fmt.Printf("len: %d, cap: %d\n", len(s), cap(s))
}
```

18. 맵 (Map)

해시 테이블 구조

키(Key)와 값(Value) 쌍으로 데이터를 초고속으로 저장하고 조회합니다. (Python의 Dictionary)

- 반드시 make나 중괄호 {} 리터럴로 **초기화**해야 합니다. nil 맵에 값을 넣으면 서버가 죽습니다.
- 조회 시 ok 플래그를 통해 키 존재 여부를 안전하게 확인하는 패턴이 필수적입니다.

```
func main() {
	// 맵 초기화
	userAges := make(map[string]int)
	userAges["Alice"] = 25
	// 키 존재 여부 확인 패턴 (보안/안전성 핵심)
	if age, ok := userAges["Bob"]; ok {
		fmt.Println("Bob의 나이:", age)
	} else {
		fmt.Println("Bob 데이터가 없습니다.")
		// 출력
	}
}
```

19. 컬렉션 순회 (For Range)

가장 많이 쓰이는 반복 패턴

배열, 슬라이스, 맵을 순회할 때 인덱스(또는 키)와 값을 동시에 쾌적하게 반환합니다.

- 필요 없는 반환값은 반드시 _ (언더스코어)로 무시해야 컴파일 에러를 막을 수 있습니다.
- **주의:** 맵을 Range로 순회할 때 출력 순서는 보장되지 않고 무작위(Random)입니다.

```
func main() {
	ips := []string{"192.168.0.1", "10.0.0.1"}
	// 인덱스가 필요 없으면 _ 로 무시!
	for _, ip := range ips {
		fmt.Println("접속 IP:", ip)
	}
	// 맵 순회
	//
	for key, val := range userMap {
		...
	}
}
```

20. 실습 랩 (Lab 3)

Lab 3

슬라이스와 맵 다루기

미션: 블랙리스트 IP 검사기

1. "1.1.1.1", "2.2.2.2" 를 가진 슬라이스(요청 IP 목록)를 만드세요.
2. "2.2.2.2" -> true 를 가진 맵(블랙리스트)을 만드세요.
3. 슬라이스를 For Range로 순회하면서, 해당 IP가 맵에 존재하면 "차단!", 아니면 "통과"를 출력하세요.

21. 함수와 다중 반환값

동시 반환의 강력함

Go 함수의 가장 큰 특징은 여러 개의 값을 동시에 리턴할 수 있다는 것입니다.

- 파라미터 변수명이 먼저 오고 타입이 뒤에 옵니다.
- 주로 정상적인 계산 '결과값'과 '에러(error)' 객체를 묶어서 동시에 반환하는 설계에 널리 쓰입니다.

```
// 몫과 나머지를 동시 반환
func divide(a, b int) (int, int) {
	return a / b, a % b
}
func main() {
	quo, rem := divide(10, 3)
	fmt.Printf("몫: %d, 나머지: %d", quo, rem)
}
```

22. 메모리 직접 제어 (Pointers)

- 포인터는 변수의 **메모리 주소**를 보관합니다. C/C++과 달리 포인터끼리의 연산은 불가능하여 상대적으로 안전합니다.

- & (주소 연산자): 변수 앞에 붙여 변수가 저장된 RAM의 물리적 주소를 가져옵니다.
- * (역참조 연산자): 주소가 가리키는 실제 데이터 값을 읽어오거나 덮어씌웁니다.
- 대용량 구조체를 복사 없이 전달하여 성능을 높일 때 유용합니다.

23. 함수 전달 방식 (Value vs Reference)

원본 데이터 수정하기

Go는 기본적으로 데이터를 복사해서 넘깁니다(Call by Value). 원본을 직접 수정하려면 반드시 포인터를 넘겨야 합니다.

- failReset 함수는 복사본을 수정하므로 원본에 영향이 없습니다.
- successReset 함수는 메모리 주소를 받아 원본 공간에 값을 직접 덮어씁니다.

```
func failReset(pwd string) {
	pwd = "000" // 값 복사 (원본 불변)
}
func successReset(pwd *string) {
	*pwd = "000" // 포인터 전달 (원본 변경 가능)
}
func main() {
	myPwd := "123"
	failReset(myPwd)
	fmt.Println(myPwd) // 여전히 123
	successReset(&myPwd) // 주소(&)를 넘김
	fmt.Println(myPwd)   // 000 변경 성공!
}
```

24. 지연 실행 보장 (defer)

자원 누수 방어

함수가 종료되기 직전에 예약된 로직이 **반드시 실행됨**을 보장하는 우아한 문법입니다.

- 파일 열기, DB 연결, 뮤텍스 락 등을 획득한 직후 바로 밑에 defer 닫기()를 선언하는 것이 관례입니다.
- 중간에 return이나 에러로 튕겨 나가도 안전하게 자원이 해제됩니다.

```
import "fmt"

func processFile(){
      fmt.Println("1. 파일 열기")
      defer fmt.Println("3. 파일 닫기 (메모리 해제)")
      fmt.Println("2. 파일 읽는 중...")
}
```

25. 서버 크래시 방어 (Panic & Recover)

가용성을 지키는 마지막 방어막

배열 인덱스 초과, 닐 포인터 참조 등 치명적 에러 시 프로그램이 죽는 것(panic)을 낚아채어 복구(recover)합니다.

- recover()는 반드시 defer 내부에서만 동작합니다.
- 웹 프레임워크(Gin 등)는 내부적으로 이 기능을 미들웨어로 감싸 서버가 완전히 죽는 것을 막아줍니다.

```
func safeProcess(){
	defer func(){
		if r:=recover();r≠nil{
			fmt.Println("안전하게 복구됨:",r)
		}
	}()
	fmt.Println("작업 시작")
	panic("치명적인 메모리 에러 발생!")
	fmt.Println("이 줄은 영원히 실행되지 않음")
}
```

26. 데이터 상태 저장 (Struct)

객체지향의 뼈대

Go는 클래스(Class)가 없습니다. 객체의 상태는 관련된 필드들을 묶은 구조체(Struct)로 표현합니다.

- 주로 데이터베이스의 하나의 행(Row) 모델이나 클라이언트의 JSON 요청을 담는 그릇으로 사용됩니다.
- 메모리 오버헤드를 막기 위해 인스턴스 생성 시 **포인터(&)**로 할당하는 것이 실무 표준입니다.

```
type User struct{
	ID uint
	Email string
}

func main(){
	u:=&User{
		ID:1,
		Email:"admin@test.com",
	}
	fmt.Println(u.Email)
}
```

27. 상속 대신 조합 (Embedding)

구조체 임베딩

Go는 복잡한 상속(Inheritance) 계층을 배제하고, 다른 구조체를 품는 '조합(Composition)'을 사용해 코드를 재사용합니다.

- 필드 이름 없이 타입만 선언하면 내부에 완전히 스며듭니다.

- GORM(ORM)에서 gorm.Model을 삽입하여 ID, 생성일, 수정일을 일괄 관리할 때 널리 쓰입니다.

```
type BaseModel struct{
	CreatedAt string
	UpdatedAt string
}

type Article struct{
	BaseModel
	Title string
}

func main(){
	a:=&Article{Title:"Go 튜토리얼"}
	a.CreatedAt="2024-01-01"
}
```

28. 객체의 행위 (Methods)

리시버(Receiver) 달기

함수 이름 앞에 괄호와 변수를 명시하여, 특정 구조체 전용 메서드로 만듭니다.

- **포인터 리시버 (*User):** 원본 구조체의 상태(필드값)를 직접 변경해야 할 때 반드시 사용합니다. 실무의 90% 이상을 차지합니다.
- 호출 시에는 객체지향 언어와 동일하게 객체.메서드() 형태로 직관적입니다.

```
type User struct{
	Status string
}

func(u *User)Ban(){
	u.Status="banned"
}

func main(){
	user:=&User{Status:"active"}
	user.Ban()
	fmt.Println(user.Status)
}
```

29. 캡슐화 (대소문자의 구분)

접근 제어의 단순화

Go는 public, private 지시어가 존재하지 않습니다. 오직 **이름의 첫 글자 대소문자**로 모든 권한을 통제합니다.

- **대문자 시작 (Exported):** 패키지 외부 접근 가능. JSON 파싱이나 DB ORM 매핑을 위해 반드시 지켜야 합니다.
- **소문자 시작 (Unexported):** 패키지 내부에서만 접근 가능. 보안에 민감한 키나 로직을 숨길 때 완벽합니다.

```
package config

type SecureConfig struct{
      APIKey string
      secretKey string
}

func(c *SecureConfig)GetSecret()string{
      return c.secretKey
}
```

30. 구조체 태그 (Struct Tags)

보안 I/O의 메타데이터

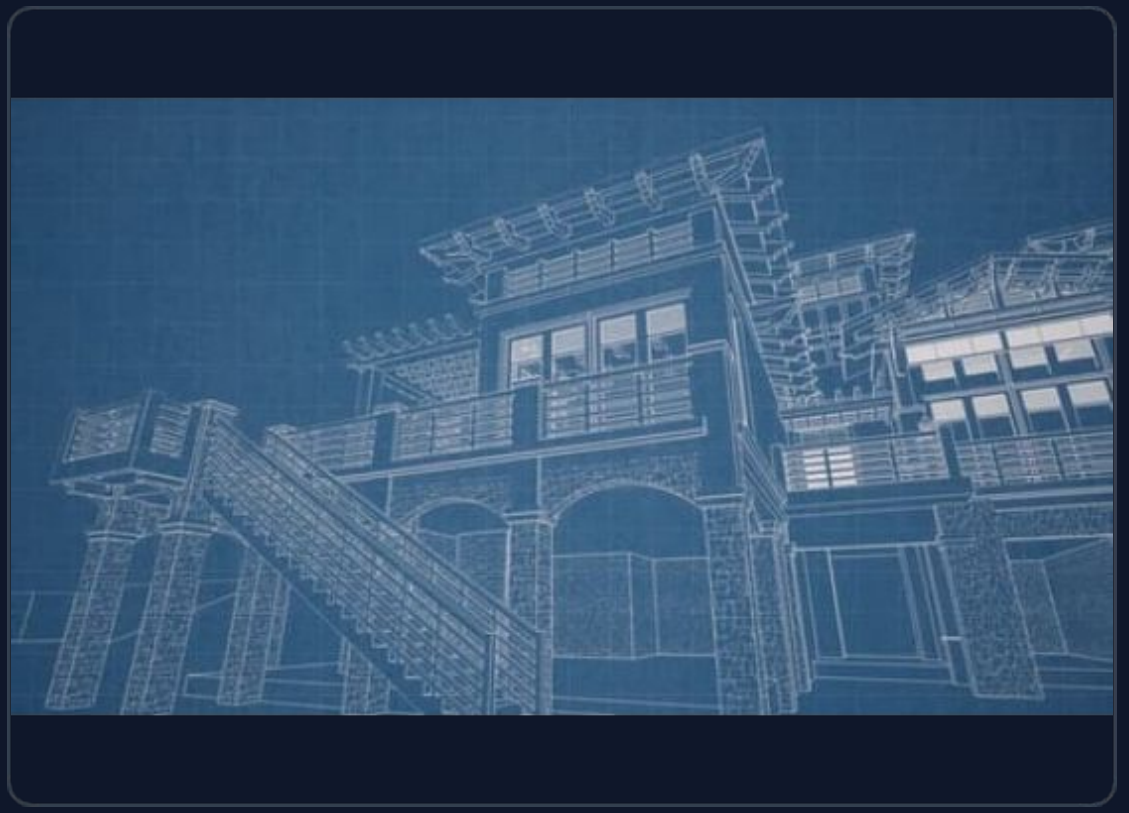
필드 선언 뒤 백틱(` `) 안에 속성을 작성하여, JSON, ORM 라이브러리가 구조체를 어떻게 읽고 쓸지 강제합니다.

- json:"이름" 형식으로 클라이언트와의 JSON 통신 키를 맞춥니다.
- **보안의 핵심 json:"-":** 비밀번호 해시와 같은 민감 정보가 응답(Output)으로 나가는 것을 프레임워크 레벨에서 원천 차단합니다.

```
type UserDTO struct{
      ID uint `json:"id"`
      Name string `json:"name"`
      Hash string `json:"-"`
      Email string `json:"email"
binding:"required,email"`
}
```

31. 인터페이스 (Interface)

- 구현 코드가 전혀 없는 메서드들의 규격(시그니처) 모음입니다.
- Layered Architecture(계층 분리)에서 Service와 Repository 사이의 결합도를 끊어주는 완충재 역할을 합니다.
- 자바의 implements 키워드가 필요 없습니다. 구조체가 인터페이스에 명시된 메서드를 모두 가지고 있으면 **자동으로 만족(Duck Typing)**합니다.



32. 실습 랩 (Lab 4)

Lab 4

객체 지향 훈련

미션: 은행 계좌 입출금기

1. Balance (int) 필드를 가진 Account 구조체를 만드세요.
2. 금액을 받아 잔액을 증가시키는 Deposit(amount int) 메서드를 **포인터 리시버**를 사용하여 정의하세요.
3. 초기 잔액 1000을 가진 인스턴스를 만들고 500을 입금한 뒤 최종 잔액을 출력하세요.

33. 초경량 스레드 (Goroutine)

- 무거운 OS 스레드를 쓰지 않고 Go 런타임이 관리하는 논리적 스레드입니다.
- 함수 호출 앞에 go 키워드 하나만 붙이면 즉시 비동기로 백그라운드 분기됩니다.
- 생성 시 단 2KB의 메모리만 차지하여, 단일 서버에서 수만 개의 요청을 거뜬히 병렬로 처리하며 압도적인 가용성(Availability)을 보장합니다.

34. 안전한 동기화 (WaitGroup)

흐름 기다리기

비동기 작업이 끝날 때까지 time.Sleep으로 기다리는 것은 불안정합니다. 작업 카운터를 통해 안전하게 동기화합니다.

- Add(n): 기다릴 고루틴의 갯수 증가.
- Done(): 작업 완료 시 카운트 1 감소 (주로 defer와 결합).
- Wait(): 카운터가 0이 될 때까지 메인 함수 블로킹 대기.

```
import "sync"

func main(){
	var wg sync.WaitGroup
	wg.Add(1)
	go func(){
		defer wg.Done()
		fmt.Println("무거운 로그 분석 완료")
	}()
	wg.Wait()
	fmt.Println("모든 작업 종료, 서버 중지")
}
```

35. 실무 웹 프레임워크 (Gin)

빠른 라우팅

표준 라이브러리 대비 40배 이상 빠른 트리 기반 라우팅을 지원하는 실무 표준 프레임워크입니다.

편리한 바인딩

클라이언트의 복잡한 JSON 입력을 Go 언어의 구조체(Struct)로 즉시 파싱하여 매핑해줍니다.

미들웨어 방어벽

라우터 앞단에 인증(JWT), 로깅 등의 공통 보안 필터를 깔끔하게 부착하여 전역 통제가 가능합니다.

36. 첫 API 서버 띄우기

서버 구동의 뼈대

단 몇 줄의 코드로 HTTP 200 JSON 응답을 내뱉는 완벽한 서버를 구동합니다.

- gin.Default()는 로거와 치명적 에러 시 서버가 죽지 않게 살려주는 Recovery 미들웨어를 자동 포함합니다.
- 메서드(GET)와 경로("/ping")를 지정하고, 익명 핸들러 함수를 콜백으로 연결합니다.

```
package main

import "github.com/gin-gonic/gin"

func main(){
	r:=gin.Default()
	r.GET("/ping",func(c *gin.Context){
		c.JSON(200,gin.H{"msg":"pong"})
	})
	r.Run(":8080")
}
```

37. 네트워크의 총괄자 (gin.Context)

- **요청/응답 통제:** 핸들러 파라미터 *gin.Context는 클라이언트 Input과 서버 Output의 모든 데이터를 관리하는 상자입니다.
- **입력 읽기:** URL 파라미터, 쿼리 스트링, HTTP 헤더 정보, JSON 바디 데이터 등을 추출합니다.
- **출력 통제:** HTTP 상태 코드(200, 400, 500) 결정, JSON 응답 전송, 보안 쿠키 발급 등을 제어합니다.
- **흐름 제어:** 미들웨어 단계에서 c.Abort()를 호출하면 악성 요청이 본 로직으로 진입하는 것을 즉시 차단합니다.

38. URL 파라미터 & 쿼리 스트링

조회용 데이터 추출

클라이언트가 URL을 통해 전달하는 식별자나 검색 옵션을 추출합니다.

- **Path:** :name 형태로 정의하며, 자원 식별에 쓰입니다. (추출값은 항상 문자열)
- **Query:** URL 뒤에 ?page=2 형태로 붙습니다. DefaultQuery를 쓰면 값이 누락되었을 때 기본값을 세팅하여 안전합니다.

```
func main(){
	r:=gin.Default()

	r.GET("/user/:name",func(c *gin.Context){
		name:=c.Param("name")
		c.JSON(200,gin.H{"user":name})
	})

	r.GET("/posts",func(c *gin.Context){
		page:=c.DefaultQuery("page","1")
		c.JSON(200,gin.H{"page":page})
	})
	r.Run()
}
```

39. 헤더 제어 (Headers)

보안 메타데이터 교환

인증 토큰(JWT) 교환이나 보안 정책(CORS) 적용을 위해 HTTP 헤더를 자유롭게 읽고 씁니다.

- c.GetHeader(): 클라이언트가 보낸 특정 헤더 값을 문자열로 추출합니다. 없으면 빈 문자열 리턴.
- c.Header(): 응답 시 클라이언트 브라우저에 지시할 커스텀 헤더를 세팅합니다.

```
r.GET("/auth",func(c *gin.Context){
	token:=c.GetHeader("Authorization")
	c.Header("X-Security-Level","High")
	if token==""{
		c.JSON(401,gin.H{"err":"토큰 누락"})
		return
	}
	c.JSON(200,gin.H{"msg":"인증 통과"})
})
```

40. 입력값 매핑 (JSON 바인딩)

ShouldBindJSON의 마법

클라이언트의 복잡한 JSON 바디를 Go 구조체로 즉각 파싱하는 1차 방어선입니다.

- 구조체 태그에 binding:"required" 속성을 달아두면 필드 누락 시 에러를 뱉습니다.
- 파라미터로 반드시 구조체의 **포인터(&)**를 넘겨 메모리 공간을 프레임워크에 위임해야 합니다.

```
type LoginReq struct{
	Email string `json:"email" binding:"required"`
	Pwd string `json:"password" binding:"required"`
}
r.POST("/login",func(c *gin.Context){
	var req LoginReq
	if err:=c.ShouldBindJSON(&req);err≠nil{
		c.JSON(400,gin.H{"err":"필수값 누락/타입오류"})
		return
	}
	c.JSON(200,gin.H{"email":req.Email})
})
```

41. 순수 로직 검증 (수동 방어)

프레임워크 우회 방지

바인딩(타입 검사) 통과 후, 프레임워크의 매직 기능에 의존하지 않는 본질적인 비즈니스 룰 검증입니다.

- 데이터의 길이, 특수문자 포함 여부, 음수 여부 등을 순수 if문 로직으로 걸러냅니다.
- "모든 입력은 악의적이다"라는 대전제를 실천하는 핵심 구역입니다.

```
r.POST("/signup",func(c *gin.Context){
	var req LoginReq
	if err:=c.ShouldBindJSON(&req);err≠nil{
		return
	}
	if len(req.Pwd)<8{
		c.JSON(400,gin.H{"err":"비밀번호 8자리 이상 필수"})
		return
	}
	c.JSON(200,gin.H{"msg":"안전하게 검증 통과"})
})
```

42. 실습 랩 (Lab 5)

Lab 5

API 설계 및 바인딩

미션: 안전한 송금 API 만들기

1. To (string), Amount (int) 필드를 가진 구조체를 만들고 required 태그를 부착하세요.
2. POST /transfer 라우터를 만들고 JSON 데이터를 바인딩하세요.
3. 로직 검증을 추가하여 balance 보다 더 많은 amount 입력하면 "금액 조작 에러" (400)를 반환하도록 하세요.

기본 파일 입출력 (os 패키지)

영속적 기록의 시작

Go 표준 라이브러리인 os 패키지를 사용하여 서버 로컬 디스크에 직접 기록을 남깁니다.

- os.WriteFile(): 파일에 데이터를 즉시 씁니다. 파일이 이미 존재하면 **완전히 덮어씁니다.**
- os.OpenFile(): **이어쓰기(Append)** 모드나 권한 설정 등 정밀한 제어가 필요할 때 사용합니다.
- 메모리 누수를 방지하기 위해 작업 후 defer f.Close()는 필수입니다.

```
import "os"

func main(){
    data:=[]byte("Server Start Log\n")
    os.WriteFile("app.log",data,0644)
f,_:=os.OpenFile("app.log",os.O_APPEND|os.O
_CREATE|os.O_WRONLY,0644)
    defer f.Close()

    f.WriteString("New Request Inbound\n")
}
```

구조화된 로깅 (Logrus)

JSON 기반 관제 시스템 연동

실무에서는 단순 텍스트보다 **JSON 포맷** 로그를 선호합니다. ELK 스택(Elasticsearch) 등에서 파싱하기 쉽기 때문입니다.

- log.SetFormatter(): 로그의 전역 출력 형식을 지정합니다.
- WithFields(): 특정 보안 이벤트에 메타데이터(접속 IP, User ID 등)를 구조적으로 묶어서 기록합니다.
- 장애 수준별로 Info, Warn, Error, Fatal 레벨을 분리합니다.

```
$ go get github.com/sirupsen/logrus
```

```
import log "github.com/sirupsen/logrus"

func main(){
    log.SetFormatter(&log.JSONFormatter{})

    log.WithFields(log.Fields{
        "user_id":"alice",
        "action":"login",
    }).Info("인증 성공")
}
```

로그 로테이션 (Lumberjack)

디스크 장애(Disk Full) 방지

서버가 무한히 한 파일에 로그를 쓰면 디스크가 꽉 차서 서버가 뻗게 됩니다. 가용성 유지를 위해 로테이션은 필수적입니다.

- MaxSize: 파일이 지정된 MB 크기에 도달하면 새로운 파일로 쪼갭니다.
- MaxBackups: 디스크에 보관할 백업 파일의 최대 개수입니다.
- MaxAge: 며칠이 지난 오래된 로그를 자동으로 삭제할지 결정합니다.

```
$ go get gopkg.in/natefinch/lumberjack.v2
```

```
import "gopkg.in/natefinch/lumberjack.v2"

func initLogger(){
    log.SetOutput(&lumberjack.Logger{
        Filename:"./logs/api.log",
        MaxSize:10,
        MaxBackups:5,
        MaxAge:30,
        Compress:true,
    })
}
```

전역 방어벽 (Middleware)

흐름의 가로채기와 통제

비즈니스 로직(핸들러)에 도달하기 전 요청을 가로채어 공통 작업(인증, 로깅, CORS)을 수행하는 최전방 수문장입니다.

- c.Next(): 모든 사전 검증을 통과했을 때, **다음 로직(또는 최종 핸들러)으로 제어권을 넘깁니다.**
- c.Abort(): 악성 요청이나 인증 실패 발견 시, 시스템 내부로의 진입을 **전역에서 즉시 끊어버립니다.**

```
func MyMiddleware()gin.HandlerFunc{
    return func(c *gin.Context){
        fmt.Println("요청 검사 시작")
        c.Next()
        fmt.Println("요청 처리 완료")
    }
}
```

파일 로깅 미들웨어 구현

로거와 미들웨어의 결합

Gin 프레임워크의 Context 정보를 추출하여 앞서 배운 **Logrus + Lumberjack** 조합으로 파일에 자동 기록합니다.

- 클라이언트 IP, 요청 경로(Path), HTTP 메서드를 구조화하여 일관성 있게 저장합니다.
- 비즈니스 로직이 실행되기 전 무조건 로그를 남기게 되어 누락 없는 완벽한 관제(Audit)를 보장합니다.

```
func JSONLogger()gin.HandlerFunc{
    return func(c *gin.Context){
        log.WithFields(log.Fields{
            "ip":c.ClientIP(),
            "method":c.Request.Method,
            "path":c.Request.URL.Path,
        }).Info("Incoming Request")
        c.Next()
    }
}
```

신뢰 경계와 라우터 그룹핑

구조적 분리 및 일괄 방어

모든 API에 똑같은 보안 검사를 걸지 않고, 보안 요구사항 수준별로 그룹을 찢어 관리합니다.

- public: 로그인, 회원가입 (비인증 허용 구역)
- private: 게시판, 개인정보 (JWT 인증 필수 구역)
- 각 라우터 그룹 선언 직후 group.Use()로 미들웨어를 **단 한 번만 선언**하여 개발자의 실수(누락)를 원천 차단합니다.

```
r:=gin.Default()

r.Use(JSONLogger())

v1:=r.Group("/api/v1")
{
    v1.Use(MyAuthMiddleware())
    v1.GET("/secret",secretHandler)
}
```

46. 가용성 보장 (Graceful Shutdown)

안전한 하차

서버 배포나 강제 종료 시, 기존에 유저가 처리 중이던 결제/조회 트랜잭션이 끊기지 않게 끝까지 기다린 후 우아하게 종료하는 필수 기법입니다.

- OS의 종료 시그널(Ctrl+C 등)을 채널로 낚아챕니다.
- srv.Shutdown()을 호출하여 유입을 막고 남은 I/O 작업 마무리를 보장합니다.

```
srv:=&http.Server{
      Addr:":8080",
      Handler:r,
}
go func(){
      srv.ListenAndServe()
}()
quit:=make(chan os.Signal,1)
signal.Notify(quit,syscall.SIGINT,syscall.SIGTERM)
←quit
fmt.Println("서버 종료 준비 중...")
srv.Shutdown(context.Background())
fmt.Println("서버 안전 종료 완료")
```

실습 랩 (Lab 7)

Lab6

통합 로깅 시스템

미션: API 접속 파일 로거 구축

1. logrus와 lumberjack 패키지를 설치하고 1MB 단위 크기로 로테이션 되도록 SetOutput을 설정하세요.
2. 모든 접속 요청의 Method, Path를 JSON 포맷으로 기록하는 미들웨어를 제작하고, 라우터 그룹 /api/v1 에 부착하세요.

왜 **ORM** 대신 순수 **SQL**인가?

- 실무에서는 GORM 같은 ORM을 널리 사용하지만, 내부에서 쿼리가 어떻게 가공되는지 **원리를 감추는 단점**이 있습니다.
- database/sql 표준 패키지와 Raw 쿼리를 직접 다뤄봄으로써 **SQL Injection이 발생하는 근본 원리**와 방어 메커니즘을 명확히 이해합니다.
- 별도 인프라 셋업이 필요 없는 초경량 파일 기반 관계형 DB인 **SQLite**를 사용하여 실습에 집중합니다.

SQLite 설치 방법

- **공식 패키지 다운로드:** sqlite.org/download.html 에 접속하여 운영체제에 맞는 tools 를 다운로드 합니다
- **설치 확인:** 다운로드한 폴더 경로를 path 에 넣고 sqlite 명령어를 실행해봅니다
- **DB 생성:** .open test.db 명령을 이용해 파일을 생성하고, 데이터베이스를 사용합니다

SQLite 설치 방법

$ go get github.com/mattn/go-sqlite3

DB 연결 및 스키마 생성

Go 표준 database/sql 패키지와 sqlite3 드라이버를 연결하여 통신합니다.

- _ "github.com/..." 형태의 언더스코어 임포트는 패키지의 init() 함수만 실행하여 드라이버를 등록하기 위함입니다.
- db.Exec() 함수를 사용하여 DDL(CREATE TABLE 등) 쿼리를 명시적으로 실행합니다.

```
import(
	"database/sql"
	_ "github.com/mattn/go-sqlite3"
)

func main(){
	db,err:=sql.Open("sqlite3","./test.db")
	if err≠nil{
		panic(err)
	}
	defer db.Close()

	query:=`CREATE TABLE IF NOT EXISTS users(
		id INTEGER PRIMARY KEY AUTOINCREMENT,
		email TEXT NOT NULL UNIQUE
	);`

	_,err=db.Exec(query)
```

안전한 데이터 삽입 (Prepared Statement)

SQL Injection 원천 차단

동적 데이터를 쿼리에 직접 문자열로 결합(+)하면 해킹(SQLi)에 무조건 노출됩니다.

- 반드시 ? (Placeholder) 문법을 사용하여 쿼리의 명령어 구조와 데이터를 완전히 격리해야 합니다.
- db.Exec()의 파라미터로 데이터를 넘겨주면, 드라이버가 안전하게 값을 **바인딩(Escaping)** 처리해 줍니다.

```
inputEmail:="user@test.com' OR '1'='1"

insertQuery:="INSERT INTO users (email) VALUES (?)"
result,err:=db.Exec(insertQuery,inputEmail)
if err≠nil{
      fmt.Println("DB 삽입 실패:",err)
}
```

트랜잭션 제어 (Transaction)

All or Nothing (데이터 원자성)

송금 작업처럼 여러 개의 쿼리가 **모두 성공하거나, 하나라도 실패하면 모두 취소(Rollback)**되어야 할 때 사용합니다.

- db.Begin()으로 트랜잭션을 시작하고 tx 객체를 얻습니다. 모든 쿼리는 tx를 통해 실행해야 같은 스코프에 묶입니다.
- 작업 도중 에러 시 defer tx.Rollback()이 자동 발동하며, 모두 성공 시에만 tx.Commit()을 명시적으로 호출합니다.

```
CREATE TABLE IF NOT EXISTS accounts(
id INTEGER PRIMARY KEY AUTOINCREMENT,
balance integer
)
insert into accounts (balance) values(0);
insert into accounts (balance) values(0);
```

트랜잭션 제어 (Transaction)

All or Nothing (데이터 원자성)

송금 작업처럼 여러 개의 쿼리가 **모두 성공하거나, 하나라도 실패하면 모두 취소(Rollback)**되어야 할 때 사용합니다.

- db.Begin()으로 트랜잭션을 시작하고 tx 객체를 얻습니다. 모든 쿼리는 tx를 통해 실행해야 같은 스코프에 묶입니다.
- 작업 도중 에러 시 defer tx.Rollback()이 자동 발동하며, 모두 성공 시에만 tx.Commit()을 명시적으로 호출합니다.

```
tx,err:=db.Begin()
if err≠nil{
      return err
}
defer tx.Rollback()

_,err=tx.Exec("UPDATE accounts SET bal = bal - 100
WHERE id = 1")
if err≠nil{
      return err
}

_,err=tx.Exec("UPDATE accounts SET bal = bal + 100
WHERE id = 2")
if err≠nil{
      return err
}

err=tx.Commit()
return err
```

실습 랩 (Lab 8)

Lab 7

DB 연동 & 트랜잭션

미션: 안전한 송금 트랜잭션

1. database/sql과 go-sqlite3 드라이버를 연결하여 test.db를 생성하세요.
2. accounts (id INTEGER PRIMARY KEY, balance INTEGER) 테이블을 생성하고, 초기 유저 2명을 1000원씩 INSERT 해두세요.
3. POST /transfer API를 호출하면, **트랜잭션(tx)**을 열어 A유저의 돈을 500원 깎고 B유저의 돈을 500원 올린 뒤 완벽하게 Commit() 하는 로직을 작성하세요.