Recompiling Your Mind

A PHP Developer's Journey to Go

Achraf SOLTANI

2026

Contents

Preface

My Story

In 2005, I wrote my first line of PHP. I was hooked immediately. Over the next seventeen years, PHP became more than a programming language—it became the lens through which I saw software development. I lived through PHP 4's procedural chaos, PHP 5's object-oriented renaissance, and PHP 7's performance revolution. I built applications with Symfony from version 1.0 onwards, watching it mature into one of the most elegant frameworks in any language.

By 2022, PHP and I had developed a kind of telepathy. I could feel when code was right. I knew, without thinking, exactly how to structure a service, wire a dependency, or craft a clean controller. The language had become an extension of my thoughts.

Then I started writing Go.

The Uncomfortable Truth

Learning Go's syntax took a few weeks. Learning to think in Go has taken years—and I'm still not there.

This isn't about intelligence or experience. It's about rewiring seventeen years of deeply ingrained mental models. Every time I reach for inheritance, Go reminds me it doesn't exist. Every time I want to throw an exception, I must write `if err != nil`. Every time I expect magic, I find explicit wiring.

The transition has been humbling. And illuminating.

Why This Book Exists

Most Go books teach you Go. This book teaches you how to stop thinking in PHP.

If you've spent years mastering PHP—especially in the Symfony ecosystem—you've developed powerful mental models. These models served you well. But they're now fighting against Go's philosophy at every turn.

This book is not a beginner's guide. It assumes you can already write Go code that compiles and runs. What you might not be able to do is write *idiomatic* Go—code that feels natural to Go developers, code that leverages Go's strengths instead of fighting them.

We'll examine every mental model you've built in PHP and show you its Go equivalent (or lack thereof). We'll explore why certain patterns feel wrong in Go, and how to develop new instincts that feel right.

Who This Book Is For

You should read this book if:

- **You've mastered PHP**, especially with frameworks like Symfony
- **You've started learning Go**, but it doesn't feel natural yet
- **You keep reaching for PHP patterns** that don't exist in Go
- **You want to understand Go's philosophy**, not just its syntax
- **You're frustrated** that years of experience seem to slow you down

You should probably look elsewhere if:

- You're new to programming entirely
- You've never worked with PHP seriously
- You're already comfortable writing idiomatic Go

What You'll Learn

Part I: The Mental Shift examines the philosophical differences between PHP and Go. We'll explore why your PHP brain fights Go and how to make peace with the transition.

Part II: Structural Rewiring covers the fundamental building blocks—structs instead of classes, composition instead of inheritance, interfaces that work implicitly.

Part III: Practical Patterns takes you through real-world concerns: web development, databases, APIs, testing, and configuration—all from a PHP developer's perspective.

Part IV: Concurrency introduces Go's killer feature—something PHP simply doesn't have. We'll build new mental models from scratch.

Part V: Advanced Topics covers reflection, performance optimisation, and system programming.

Part VI: Deployment and Migration provides practical strategies for building, deploying, and migrating from PHP to Go.

How to Read This Book

Each chapter compares PHP and Go approaches side by side. We'll show Symfony patterns you know intimately, then demonstrate their Go equivalents (or explain why no equivalent exists).

Code examples assume familiarity with modern PHP (8.x) and Symfony (5.x/6.x). Go examples target Go 1.21+.

The exercises at the end of each chapter aren't optional. They're designed to break your PHP habits and build Go instincts. Do them.

A Note on Difficulty

This transition is hard. Not because Go is complex—it's famously simple. But because you're not learning something new; you're unlearning something old while learning its replacement.

Be patient with yourself. The discomfort you feel is the learning happening.

Acknowledgements

To the PHP community that shaped my thinking for seventeen years. To the Go community that's reshaping it now. And to everyone who's ever felt like an expert beginner—starting over in a new language, humbled by how much they have to relearn.

Let's begin.

"In the beginner's mind there are many possibilities, but in the expert's there are few." — Shunryu Suzuki

Table of Contents

- Explicit imports
- `go.mod` vs `composer.json`
- Internal packages
- No circular imports

9. **The Standard Library Is Your Framework**
   - `net/http` vs HttpFoundation
   - `encoding/json` vs Serializer
   - `database/sql` vs Doctrine DBAL
   - `html/template` vs Twig

Part III: Practical Patterns

10. **Web Development Without a Framework**
    - HTTP handlers
    - Middleware patterns
    - Routing
    - Request validation
11. **Database Access**
    - `database/sql` fundamentals
    - Query builders and ORMs
    - Migrations
    - Connection pooling
12. **API Development**
    - JSON encoding/decoding
    - OpenAPI integration
    - Authentication middleware
    - Validation patterns
13. **Testing—A Different Philosophy**
    - Table-driven tests
    - No assertions library
    - Mocking with interfaces
    - Benchmarking
14. **Configuration and Environment**
    - No `.env` magic
    - Viper patterns
    - 12-factor principles
    - Secret management

Part IV: Concurrency—The New Frontier

15. **Introducing Concurrency**
    - What PHP doesn't have
    - Goroutines vs processes
    - The Go scheduler
    - Mental model shift
16. **Channels—Message Passing**
    - Typed channels

- Buffered vs unbuffered
- Channel directions
- Range over channels

17. **Select and Coordination**
    - Select statements
    - Timeouts and deadlines
    - Context package
    - WaitGroups
18. **Concurrency Patterns**
    - Worker pools
    - Fan-out/fan-in
    - Pipeline processing
    - Graceful shutdown
19. **When Concurrency Goes Wrong**
    - Race conditions
    - The race detector
    - Deadlocks
    - Channel leaks

Part V: Advanced Topics

20. **Reflection and Code Generation**
    - reflect package
    - When to use reflection
    - `go generate`
    - Build-time vs runtime
21. **Performance Optimisation**
    - Profiling with pprof
    - Memory allocation patterns
    - Escape analysis
    - Pool patterns
22. **Calling C and System Programming**
    - CGO basics
    - Syscalls
    - CLI tools
    - Signal handling

Part VI: Deployment and Migration

23. **Building and Deploying**
    - Single binary deployment
    - Cross-compilation
    - Docker multi-stage builds
    - Systemd services
24. **Observability**
    - Structured logging
    - Prometheus metrics

- OpenTelemetry tracing
- Health checks

25. **Migration Strategies**
    - Strangler fig pattern
    - Side-by-side execution
    - API gateway approaches
    - Case study

Appendices

A. **PHP-to-Go Phrasebook** B. **Standard Library Essentials** C. **Common Pitfalls** D. **Symfony-to-Go Service Mapping** E. **Recommended Reading**

Chapter 1: Why Your PHP Brain Fights Go

You've spent years—perhaps decades—mastering PHP. You know its quirks, its strengths, its idioms. You can look at a codebase and immediately sense what's wrong. You've internalised patterns so deeply that they feel like instinct.

Now you're learning Go, and something strange is happening: your expertise is working against you.

The Curse of Expertise

When you're a beginner, everything is new. You have no expectations, no ingrained habits. You absorb information without resistance.

But when you're an expert learning a new language, you bring seventeen years of baggage. Every concept in Go gets filtered through your PHP lens. You see structs and think "classes without inheritance." You see error returns and think "exceptions that forgot how to throw." You see explicit imports and think "why isn't there autoloading?"

This filtering isn't conscious. It happens before you can stop it. And it's exactly what makes the transition so difficult.

The Expertise Trap

In PHP, you've developed what cognitive scientists call "chunking"—the ability to see complex patterns as single units. When you look at a Symfony controller, you don't see individual lines of code; you see a coherent whole.

```
#[Route('/users/{id}', methods: ['GET'])]
public function show(int $id, UserRepository $repo): Response
{
    $user = $repo->find($id);
    if (!$user) {
        throw new NotFoundHttpException();
    }
    return $this->json($user);
}
```

You don't consciously process the autowiring, the parameter conversion, the exception handling, the JSON serialisation. It's all one mental unit: "fetch user, return JSON."

In Go, that same operation looks like this:

```go
func (h *UserHandler) Show(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil {
        http.Error(w, "invalid id", http.StatusBadRequest)
        return
    }

    user, err := h.repo.Find(r.Context(), id)
    if err != nil {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }

    if err := json.NewEncoder(w).Encode(user); err != nil {
        http.Error(w, "encoding error", http.StatusInternalServerError)
        return
    }
}
```

Your PHP brain sees this and screams: "Why is this so verbose? Where's the magic? Why do I have to handle every error manually?"

But a Go developer sees something different: explicit, testable, and obvious code where nothing is hidden.

Interpreted vs Compiled: More Than Just Speed

PHP and Go differ fundamentally in how they execute. PHP interprets your code at runtime; Go compiles it to machine code. This isn't just a performance detail—it shapes everything about how the languages work.

PHP's Runtime Flexibility

In PHP, code is evaluated at runtime. This enables powerful features:

```php
// Dynamic method calls
$method = 'processOrder';
$service->$method($order);

// Runtime class discovery
$handlers = glob(__DIR__ . '/Handlers/*.php');
foreach ($handlers as $file) {
    require_once $file;
}

// Magic methods
public function __call($name, $args) {
    // Handle any method dynamically
}
```

This flexibility is incredibly powerful. It's what makes frameworks like Symfony possible—autowiring, event dispatching, and annotation processing all rely on runtime introspection.

Go's Compile-Time Rigidity

Go resolves everything at compile time. There's no runtime class loading, no dynamic method discovery, no magic methods:

```
// This won't compile - method must exist
method := "ProcessOrder"
service.method(order)  // Error: method is not a field

// No glob-and-load pattern
// All imports must be explicit and known at compile time

// No magic methods
// If a method doesn't exist, it doesn't exist
```

This seems limiting. But it means that if your Go code compiles, entire categories of errors are impossible:

- No "method not found" at runtime
- No typos in method names that only fail in production
- No missing dependencies discovered during a critical deployment

The Safety Trade-off

PHP trusts you to get things right at runtime. Go forces you to prove correctness at compile time. Neither is wrong, but they require different mental approaches.

In PHP, you might write:

```
$user = $repo->findOrFail($id);  // Might throw, might not
$user->activate();  // Hope $user has this method
```

In Go, you must be explicit:

```
user, err := repo.Find(ctx, id)
if err != nil {
    return nil, err  // Handle the error now
}
user.Activate()  // Compiler guarantees this exists
```

Dynamic vs Static: The Freedom You're Losing (and Gaining)

PHP's dynamic typing is one of its most defining features:

```
function process($data) {
    if (is_array($data)) {
```

```php
        return array_map(fn($x) => $x * 2, $data);
    }
    return $data * 2;
}

process(5);       // 10
process([1, 2]);  // [2, 4]
```

This flexibility is why PHP is so productive for rapid prototyping. You don't waste time declaring types—you just write code that works.

What You're Losing

In Go, every value has a single type, known at compile time:

```go
// This isn't possible in Go
func process(data any) any {
    // You'd need type assertions and it would be ugly
}

// Instead, you write separate functions or use generics
func processInt(data int) int {
    return data * 2
}

func processSlice(data []int) []int {
    result := make([]int, len(data))
    for i, v := range data {
        result[i] = v * 2
    }
    return result
}
```

You're losing the ability to write "it works on anything" functions easily. You're losing the convenience of not thinking about types until you need to.

What You're Gaining

But you're gaining something valuable: certainty.

In PHP, this code compiles and runs:

```php
function calculateTotal(array $items): float
{
    return array_sum(array_column($items, 'price'));
}

// Called with wrong data
calculateTotal(['not', 'items']);  // Returns 0, silently wrong
```

In Go, type mismatches are caught at compile time:

```go
type Item struct {
    Price float64
}

func calculateTotal(items []Item) float64 {
    var total float64
    for _, item := range items {
        total += item.Price
    }
    return total
}

// Called with wrong data
calculateTotal([]string{"not", "items"})  // Won't compile
```

The Go compiler acts as a proofreader that catches entire categories of errors before your code ever runs.

"It Just Works" vs "Prove It Works"

PHP culture values pragmatism. Get it working, ship it, iterate. This approach built the modern web.

```php
// Symfony's magic - it just works
#[Required]
public function setLogger(LoggerInterface $logger): void
{
    $this->logger = $logger;
}
```

How does `#[Required]` work? How does Symfony know to call this method? How does it find the LoggerInterface implementation? You don't need to know. It just works.

Go culture values explicitness. Show your work. Make everything visible.

```go
// Go's explicitness - prove it works
func NewService(logger *slog.Logger) *Service {
    return &Service{logger: logger}
}

// Caller
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
service := NewService(logger)
```

Nothing is hidden. Every dependency is explicitly passed. There's no container, no autowiring, no magic.

The Debugging Difference

When Symfony's autowiring breaks, you're debugging framework internals:

```
Could not autowire service "App\Service\OrderService":
argument "$repository" of method "__construct()" references
interface "App\Repository\OrderRepositoryInterface" but no
such service exists.
```

When Go code fails, you're debugging your code:

```
./main.go:15:23: cannot use repo (variable of type *OrderRepository)
as OrderRepositoryInterface value in argument to NewOrderService:
*OrderRepository does not implement OrderRepositoryInterface
(missing method FindByUser)
```

Both errors tell you what's wrong. But Go's error points directly at your code and the specific missing method.

The Discomfort Is the Learning

If Go feels awkward, that's not a sign that something is wrong with Go or with you. It's a sign that learning is happening.

Your PHP mental models are deeply ingrained. They took years to build. Replacing them with Go mental models takes time and deliberate practice.

Every time you feel the urge to: - Create a base class (Go has no inheritance) - Throw an exception (Go uses error returns) - Use a magic method (Go has no magic) - Let the framework handle it (Go uses explicit wiring)

…you're feeling the boundary between your old mental model and the new one. That friction is productive.

Embracing the Beginner's Mind

The fastest path through this transition is to temporarily let go of your expertise. Approach Go as if PHP didn't exist. Accept that things will feel verbose, explicit, and perhaps even primitive.

Then watch as the patterns start making sense. As the verbosity reveals clarity. As the explicitness enables confidence.

The goal isn't to forget PHP. It's to add Go's mental models alongside your existing ones, and to know when to apply which.

Summary

- **Expertise is a double-edged sword**: Your PHP knowledge filters how you see Go, often unhelpfully
- **Interpreted vs compiled** changes everything about how you think about code correctness
- **Dynamic vs static typing** trades flexibility for certainty
- **Explicitness vs magic** trades convenience for clarity
- **The discomfort is productive**: It means your mental models are being rewired

---

Exercises

1. **Error Archaeology**: Take a PHP project you know well. Find three places where errors could occur at runtime but wouldn't be caught by static analysis. How would Go's type system prevent each?
2. **Magic Inventory**: List all the "magic" features your favourite Symfony application uses (autowiring, annotations, event listeners, etc.). For each, describe what would need to be explicit in Go.
3. **Expertise Audit**: Write down five PHP patterns that feel "obvious" to you. For each, explain what assumptions underlie the pattern. Which assumptions don't hold in Go?
4. **Compile-Time Proof**: Take a simple PHP function and rewrite it in Go. Identify all the checks that move from runtime to compile time.
5. **Verbosity Analysis**: Compare equivalent operations in PHP and Go (e.g., HTTP handler, JSON processing). Count the lines of code. Then count the explicit decisions made in each version. What's the ratio?
6. **Dynamic Challenge**: Write PHP code that uses dynamic typing heavily (e.g., a function that accepts mixed input types). Consider how you would restructure this for Go's type system.
7. **Framework Dependency Map**: Draw a diagram showing everything Symfony does implicitly when handling a single HTTP request. How many of these steps would be explicit in Go?
8. **Beginner's Mind Exercise**: Explain a Go concept (channels, goroutines, or interfaces) to yourself as if you'd never programmed before. Notice where PHP concepts intrude on your explanation.

Chapter 2: Philosophy Differences

PHP and Go emerged from different eras, different problems, and different worldviews. Understanding these philosophical differences is key to making the mental transition.

PHP: "Get It Done, Fix It Later"

PHP was created in 1994 to make web development accessible. Rasmus Lerdorf famously didn't intend to create a programming language—he just wanted to track visits to his online resume.

This origin story matters. PHP was always about pragmatism, accessibility, and getting things working. The language evolved to solve immediate problems, often at the expense of long-term consistency.

The Pragmatist's Toolkit

PHP's philosophy can be summarised as: "Make the common case easy."

```php
// Reading a file? One line.
$content = file_get_contents('data.json');

// JSON decode? One line.
$data = json_decode($content, true);

// Database query? A few lines.
$users = $pdo->query("SELECT * FROM users")->fetchAll();
```

No setup, no boilerplate, no ceremony. Just results.

This philosophy made PHP the language of the web. When you needed a website quickly, PHP delivered. When something broke, you fixed it in production. When the code got messy, you refactored later (or didn't).

Symfony's Mature Pragmatism

Symfony brought discipline to PHP without abandoning pragmatism. It introduced:

- Conventions that reduce decisions
- Dependency injection for testability
- A component ecosystem for flexibility

But Symfony still embraces PHP's core philosophy. Magic methods, annotations, and autowiring all prioritise developer convenience over explicitness:

```php
#[Route('/api/users')]
class UserController extends AbstractController
```

```
{
    public function __construct(
        private UserRepository $users,  // Autowired
        private LoggerInterface $logger  // Autowired
    ) {}

    #[Route('/{id}', methods: ['GET'])]
    public function show(User $user): Response  // ParamConverter magic
    {
        return $this->json($user);  // Serializer magic
    }
}
```

How many implicit operations happen in this code? The route is parsed from annotations. The constructor parameters are autowired. The `$user` parameter is hydrated from the database via ParamConverter. The response is serialised by the Symfony Serializer.

None of this is visible. It just works.

Go: "Do It Right, Do It Once"

Go was created in 2007 at Google to solve Google's problems: massive codebases, thousands of engineers, slow compile times, and dependency hell.

The creators—Rob Pike, Ken Thompson, and Robert Griesemer—had decades of experience with large-scale systems. They'd seen what happens when languages accumulate features: complexity compounds, codebases become unmaintainable, and build times grow without bound.

Go's philosophy is ruthlessly minimalist: include only what's essential, and make everything explicit.

The Minimalist's Manifesto

Go's design principles:

- **One way to do things**: Less choice means less cognitive load
- **Explicit over implicit**: No hidden behaviour
- **Simplicity over expressiveness**: Readability trumps writability
- **Composition over inheritance**: Flat hierarchies
- **Fast compilation**: Measured in seconds, not minutes

This philosophy produces code that looks different from PHP:

```
// Reading a file
content, err := os.ReadFile("data.json")
if err != nil {
    return nil, fmt.Errorf("reading data: %w", err)
}

// JSON decode
var data map[string]interface{}
```

```go
if err := json.Unmarshal(content, &data); err != nil {
    return nil, fmt.Errorf("parsing JSON: %w", err)
}

// Database query
rows, err := db.QueryContext(ctx, "SELECT * FROM users")
if err != nil {
    return nil, fmt.Errorf("querying users: %w", err)
}
defer rows.Close()

var users []User
for rows.Next() {
    var u User
    if err := rows.Scan(&u.ID, &u.Name, &u.Email); err != nil {
        return nil, fmt.Errorf("scanning row: %w", err)
    }
    users = append(users, u)
}
```

More lines of code? Absolutely. But also: - Every error is handled explicitly - Every resource cleanup is visible (`defer rows.Close()`) - No magic—you can trace exactly what happens

Explicit Over Implicit (No Magic)

PHP culture embraces "magic"—behaviour that happens without explicit code. Symfony takes this further with:

- **Autowiring**: Dependencies appear without configuration
- **ParamConverters**: Request parameters become objects
- **Event listeners**: Code runs without being called
- **Annotations**: Metadata drives behaviour

```php
// How does this get called? Magic.
#[AsEventListener]
class OrderCreatedListener
{
    public function __invoke(OrderCreatedEvent $event): void
    {
        // This runs when OrderCreatedEvent is dispatched
        // But you can't tell from looking at the code
    }
}
```

Go rejects magic entirely. If something happens, the code shows it happening:

```go
// No magic - explicit subscription
type OrderService struct {
    listeners []func(Order)
}
```

```go
func (s *OrderService) OnOrderCreated(fn func(Order)) {
    s.listeners = append(s.listeners, fn)
}

func (s *OrderService) CreateOrder(o Order) error {
    // ... create order ...

    // Explicit notification
    for _, listener := range s.listeners {
        listener(o)
    }
    return nil
}

// Wiring is explicit
service := &OrderService{}
service.OnOrderCreated(func(o Order) {
    log.Printf("Order created: %s", o.ID)
})
```

The PHP version is more concise. The Go version is more traceable. Neither is objectively better—they reflect different values.

Simplicity Over Expressiveness

PHP provides many ways to express the same idea:

```php
// All valid ways to iterate
foreach ($items as $item) { ... }
array_map(fn($item) => ..., $items);
array_walk($items, function($item) { ... });
for ($i = 0; $i < count($items); $i++) { ... }
```

Go provides one way:

```go
// The only way to iterate a slice
for i, item := range items {
    // ...
}
```

PHP provides many ways to declare functions:

```php
function named($x) { return $x * 2; }
$lambda = function($x) { return $x * 2; };
$arrow = fn($x) => $x * 2;
$method = [$object, 'method'];
```

Go provides two, and they're clearly distinct:

```go
// Function declaration
func double(x int) int { return x * 2 }

// Function literal (closure)
double := func(x int) int { return x * 2 }
```

This limitation is intentional. When there's only one way to do something, code becomes consistent across teams, projects, and companies. Any Go code you read uses the same patterns.

"A Little Copying Is Better Than a Little Dependency"

This Go proverb captures a fundamental difference from PHP culture.

In PHP/Composer land, you reach for packages freely:

```json
{
    "require": {
        "symfony/string": "^6.0",
        "nesbot/carbon": "^2.0",
        "ramsey/uuid": "^4.0",
        "league/csv": "^9.0"
    }
}
```

Each package brings transitive dependencies, potential conflicts, and maintenance burden. But the PHP community considers this normal—packages are how you avoid reinventing wheels.

Go culture is more conservative:

- The standard library is comprehensive and preferred
- Third-party packages require justification
- Copying small utility functions is acceptable

```go
// Go developer's typical response to "which UUID library?"
import "github.com/google/uuid"  // Just this one, it's from Google

// But for simpler utilities, just write it
func formatBytes(b int64) string {
    const unit = 1024
    if b < unit {
        return fmt.Sprintf("%d B", b)
    }
    // ... simple formatting code ...
}
```

The overhead of importing a package for `formatBytes` isn't worth the dependency. In PHP, you might import `league/bytes` without thinking twice.

Why Go Feels Boring (And Why That's Good)

Coming from PHP's expressiveness, Go can feel painfully boring:

- No generics until recently (1.18)
- No exceptions
- No inheritance
- No magic methods
- No annotations
- No operator overloading
- No function overloading

This is by design. Go optimises for reading code, not writing it. When every codebase uses the same limited feature set, you can read any Go code fluently.

Compare two hypothetical codebases:

PHP Project A might use: - Traits extensively - Magic methods for ORM - Annotations for routing - Custom Collection classes with operator overloading

PHP Project B might use: - Interfaces exclusively (no traits) - Explicit repository patterns - YAML routing - Plain arrays with array functions

Both are valid PHP, but reading one after the other requires mental context-switching.

Go Project A and **Go Project B** will look almost identical. They'll use the same patterns because Go's feature set is small enough that everyone converges on similar solutions.

This consistency has profound benefits for large organisations and open source. Any Go developer can contribute to any Go project with minimal ramp-up time.

Symfony's "Magic" vs Go's Transparency

Let's examine a concrete example: dependency injection.

Symfony's Approach

```yaml
# services.yaml (usually autoconfigured)
services:
    _defaults:
        autowire: true
        autoconfigure: true

    App\:
        resource: '../src/'
```

```php
class OrderService
{
    public function __construct(
        private OrderRepository $repository,
        private MailerInterface $mailer,
        private LoggerInterface $logger,
    ) {}
}
```

How does Symfony know which implementations to inject? It scans your codebase, reads interfaces, matches types, and wires everything together. The process involves:

1. Compiler passes
2. Service definitions
3. Autowiring logic
4. Proxy generation (for lazy services)
5. Container compilation

This is powerful but opaque. When it works, it's magical. When it breaks, you're debugging XML service definitions and compiler pass execution order.

Go's Approach

```go
type OrderService struct {
    repository OrderRepository
    mailer     Mailer
    logger     *slog.Logger
}

func NewOrderService(repo OrderRepository, mailer Mailer, logger *slog.Logger) *OrderService {
    return &OrderService{
        repository: repo,
        mailer:     mailer,
        logger:     logger,
    }
}

// In main.go
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    db := connectToDatabase()
    repo := NewOrderRepository(db)
    mailer := NewSMTPMailer(smtpConfig)

    orderService := NewOrderService(repo, mailer, logger)
    // Use orderService
}
```

Every dependency is explicitly constructed and passed. There's no scanning, no matching, no magic. The wiring code might be tedious to write, but it's trivial to understand and debug.

If you prefer tooling assistance, code generators like Wire can generate the wiring code—but they do so at compile time, producing explicit code you can read.

Summary

- **PHP's pragmatism** prioritises getting things done quickly; Go's minimalism prioritises long-term maintainability
- **Magic vs explicitness** is a trade-off between convenience and traceability
- **Feature richness vs simplicity** affects code consistency across projects

- **Dependency culture** differs significantly between the ecosystems
- **Go's "boring" design** enables universal readability

Exercises

1. **Philosophy Archaeology**: Read the original PHP RFC for a feature (e.g., attributes, arrow functions). Then read a Go proposal that was rejected. Compare the reasoning. What values drive each decision?
2. **Magic Removal**: Take a Symfony controller with autowiring, ParamConverters, and serialisation groups. Rewrite it with everything explicit—no framework magic. How many hidden steps become visible?
3. **Consistency Check**: Find three open-source Go projects in different domains (web, CLI, library). Note the structural similarities. Then do the same for three PHP projects. Which ecosystem shows more consistency?
4. **Dependency Audit**: Run `composer show` on a PHP project. Count the total number of packages (direct + transitive). Then run `go mod graph` on a Go project. Compare the dependency counts and discuss why they differ.
5. **Simplicity Exercise**: Implement a simple in-memory cache in PHP three different ways (array, class with magic methods, class with explicit methods). Then implement it in Go. Which PHP version is closest to the Go version?
6. **One-Way Principle**: List five things PHP allows multiple ways to do. For each, explain Go's singular approach. Do you lose expressiveness or gain consistency?
7. **Boring Code Review**: Write the most "clever" PHP code you can—using all available language features expressively. Then write equivalent Go code. Which would you prefer to maintain in five years?
8. **Values Reflection**: Write a short essay explaining which philosophy (PHP's or Go's) matches your personal values as a developer. Has it changed since you started learning Go?

Chapter 3: The Type System Transition

PHP's relationship with types has evolved dramatically. From PHP 4's complete absence of type hints, through PHP 7's scalar types, to PHP 8's union types and intersection types—the language has gradually embraced static typing while preserving dynamic flexibility.

Go, by contrast, was statically typed from day one. Every value has exactly one type, known at compile time, no exceptions.

This chapter explores how to transition your mental model from PHP's flexible typing to Go's strict typing.

From $anything to Strict Types

In PHP, variables are vessels that can hold anything:

```php
$value = 42;
$value = "forty-two";
$value = ['forty', 'two'];
$value = new FortyTwo();

function process($input) {
    // $input could be anything
    // Your code must handle all possibilities
}
```

Even with modern PHP's type declarations, dynamic typing remains the default:

```php
declare(strict_types=1);

function processString(string $input): string
{
    return strtoupper($input);
}

// Without strict_types, this might work via coercion
// With strict_types, it fails at runtime
processString(42);
```

Note the key word: **runtime**. PHP discovers type errors when the code executes.

Go's Compile-Time Certainty

In Go, every variable has exactly one type, forever:

```go
var value int = 42
value = "forty-two"  // Compile error: cannot use string as int

func process(input string) string {
    // input is always a string, guaranteed
    return strings.ToUpper(input)
}

process(42)  // Compile error: cannot use int as string
```

The Go compiler rejects invalid code before it ever runs. There's no `strict_types` to enable—strictness is the only mode.

What You're Giving Up

PHP's dynamic typing enables powerful patterns:

```php
// Generic containers
$cache = [
    'user:1' => $userObject,
    'config' => ['debug' => true],
    'counter' => 42,
];

// Flexible function parameters
function dump(...$values): void
{
    foreach ($values as $value) {
        var_dump($value);  // Works with anything
    }
}

// Duck typing
function getLength($item): int
{
    return count($item);  // Works with arrays, Countable, etc.
}
```

Go requires explicit type definitions for each case:

```go
// Separate caches for different types
userCache := make(map[string]User)
configCache := make(map[string]map[string]bool)
counterCache := make(map[string]int)

// Or use interface{}/any (loses type safety)
cache := make(map[string]any)
cache["user:1"] = userObject
cache["config"] = map[string]bool{"debug": true}
cache["counter"] = 42
// But now you need type assertions to use values
```

This is the fundamental trade-off: flexibility versus safety.

Type Inference: Go's Compromise

Go's designers understood that explicit typing everywhere is tedious. Their solution: **type inference** with the short declaration operator `:=`.

```
// Explicit type
var name string = "Alice"
var age int = 30

// Inferred type (same result)
name := "Alice"  // inferred as string
age := 30        // inferred as int

// Works with complex types
users := []User{{Name: "Alice"}, {Name: "Bob"}}  // inferred as []User
config := map[string]int{"port": 8080}           // inferred as map[string]int
```

The type is still static and known at compile time—the compiler infers it from the right-hand side. This gives you PHP-like brevity with Go's compile-time safety.

Where Inference Stops

Type inference has limits:

```
// The compiler can't infer the type of an empty literal
var users []User        // Must specify type
users := []User{}       // Or use typed literal

// Function signatures are never inferred
func add(a int, b int) int {  // Must specify all types
    return a + b
}

// Interface variables need explicit types when empty
var reader io.Reader  // Must declare interface type
```

The pattern: Go infers types from values but requires explicit types for declarations without values.

When You Miss `mixed` and When You Don't

PHP 8 introduced the `mixed` type to explicitly indicate "any type":

```
function log(mixed $message): void
{
    file_put_contents('log.txt', print_r($message, true), FILE_APPEND);
}
```

Go's equivalent is `any` (alias for `interface{}`):

```
func log(message any) {
    file, _ := os.OpenFile("log.txt", os.O_APPEND|os.O_WRONLY, 0644)
    defer file.Close()
    fmt.Fprintln(file, message)
}
```

Both work, but there's a crucial difference in how you use the value:

```
// PHP: Use it directly
function processValue(mixed $value): string
{
    if (is_array($value)) {
        return implode(', ', $value);
    }
    return (string) $value;
}
```

```
// Go: Must type-assert first
func processValue(value any) string {
    switch v := value.(type) {
    case []string:
        return strings.Join(v, ", ")
    case string:
        return v
    case fmt.Stringer:
        return v.String()
    default:
        return fmt.Sprint(value)
    }
}
```

In Go, `any` values are opaque until you assert their type. This is intentionally awkward—it discourages overuse of `any`.

When You Actually Miss `mixed`

Legitimate uses of `any` in Go are rare:

1. **Serialisation**: `json.Unmarshal` into `map[string]any`
2. **Logging**: Print statements that accept anything
3. **Generic containers** (before Go 1.18 generics)

Most other uses signal design problems. If you reach for `any` often, you're probably fighting Go's type system instead of working with it.

Generics: Go's Late Arrival vs PHP 8's Union Types

PHP 8's union and intersection types provide flexibility:

```php
function processId(int|string $id): User
{
    return $this->repo->find($id);
}

function setLogger(LoggerInterface&Countable $logger): void
{
    // $logger implements both interfaces
}
```

Go 1.18 introduced generics, which solve a different problem:

```go
// Generic function: works with any ordered type
func Min[T constraints.Ordered](a, b T) T {
    if a < b {
        return a
    }
    return b
}

// Usage
minInt := Min(3, 5)       // T inferred as int
minStr := Min("a", "b")   // T inferred as string
```

Key Differences

PHP union types let a parameter accept multiple unrelated types. The function handles each type differently:

```php
function format(int|float|string $value): string
{
    if (is_string($value)) return $value;
    return number_format($value, 2);
}
```

Go generics constrain a type parameter to satisfy requirements, then treat all valid types uniformly:

```go
// T must be ordered (comparable with <)
func Sort[T constraints.Ordered](slice []T) {
    // Sorting logic that works identically for all ordered types
}
```

Go doesn't have union types. If you need `int | string`, you use:

1. **Separate functions**: `ProcessInt`, `ProcessString`
2. **Interface**: Define a common interface both types satisfy
3. **`any` with type switch**: As a last resort

```go
// Approach 1: Separate functions (clearest)
func ProcessInt(id int) User { ... }
func ProcessString(id string) User { ... }

// Approach 2: Interface (when behaviour is shared)
type Identifier interface {
    String() string
}

func Process(id Identifier) User { ... }
```

Type Assertions vs PHP's `instanceof`

PHP's type checking is intuitive:

```php
if ($value instanceof User) {
    echo $value->getName();
}

if (is_string($value)) {
    echo strtoupper($value);
}
```

Go uses type assertions:

```go
// Simple assertion (panics if wrong type)
user := value.(User)
fmt.Println(user.Name)

// Safe assertion (checks first)
if user, ok := value.(User); ok {
    fmt.Println(user.Name)
}

// Type switch (for multiple possibilities)
switch v := value.(type) {
case User:
    fmt.Println(v.Name)
case string:
    fmt.Println(strings.ToUpper(v))
case int:
    fmt.Println(v * 2)
default:
    fmt.Println("unknown type")
}
```

The two-value form (`value, ok := x.(T)`) is idiomatic Go—it never panics and lets you handle the "wrong type" case gracefully.

The Empty Interface Dance

When working with `any`/`interface{}`, you'll often need multiple assertions:

```
func extractName(data any) string {
    // Is it a map?
    if m, ok := data.(map[string]any); ok {
        // Is the "name" key a string?
        if name, ok := m["name"].(string); ok {
            return name
        }
    }
    // Is it a struct with Name field? (can't do this directly)
    // You'd need reflection or an interface
    return ""
}
```

This verbosity is intentional—it's showing you how much type information you've lost. In Go, you're better off designing types that don't require such assertions.

Symfony's Type-Hinted DI vs Go's Explicit Wiring

Let's compare how type systems interact with dependency injection.

Symfony: Types as Configuration

```
class OrderService
{
    public function __construct(
        private OrderRepository $repository,
        private MailerInterface $mailer,
    ) {}
}
```

Symfony's container uses type hints as configuration: - `OrderRepository` is a concrete class → inject it directly - `MailerInterface` is an interface → find a matching service

The wiring is implicit, driven by types.

Go: Types as Constraints Only

```
type OrderService struct {
    repository OrderRepository  // Interface
    mailer     Mailer          // Interface
}

func NewOrderService(repo OrderRepository, mailer Mailer) *OrderService {
    return &OrderService{
        repository: repo,
        mailer:     mailer,
```

```
    }
}

// Wiring is explicit
func main() {
    repo := NewSQLOrderRepository(db)
    mailer := NewSMTPMailer(config)
    service := NewOrderService(repo, mailer)
}
```

Go's types constrain what can be passed but don't configure how to find it. You write the wiring code explicitly.

This might seem like a step backward, but consider:

- **Clarity**: Every dependency is visible in `main.go`
- **Testability**: Swap dependencies by passing different implementations
- **No surprises**: No container magic to debug

Summary

- **Static typing** catches errors at compile time, not runtime
- **Type inference** (`:=`) provides convenience without sacrificing safety
- **Generics** solve different problems than PHP's union types
- **Type assertions** replace `instanceof` but require more explicit handling
- **Explicit wiring** replaces type-driven dependency injection

---

Exercises

1. **Type Conversion Audit**: Take PHP code that relies on type coercion (e.g., concatenating int with string). Rewrite it in Go with explicit conversions. How many hidden conversions become visible?
2. **Union Type Refactor**: Find PHP code using union types (`int|string`). Design the Go equivalent using either separate functions, interfaces, or generics. Compare the approaches.
3. **Generic Implementation**: Implement a generic `Stack[T]` in Go with `Push`, `Pop`, and `Peek` methods. Then implement the same in PHP using union types or mixed. Which is more type-safe?
4. **Type Assertion Chains**: Write Go code that parses a JSON object into `map[string]any` and extracts deeply nested values safely. Count the type assertions needed. Consider how you'd redesign with defined struct types.
5. **Interface Discovery**: Take a PHP class that implements multiple interfaces. Convert it to Go. How does implicit interface satisfaction change the design?
6. **Inference Limits**: Write Go code that uses `:=` extensively, then convert to explicit `var` declarations. Do the explicit types reveal any surprises about what types were actually inferred?

7. **Container Replacement**: Take a Symfony service with autowired dependencies. Write equivalent Go code with manual wiring. Measure lines of code versus clarity of dependency flow.

8. **Type Safety Comparison**: Create a scenario where PHP's dynamic typing would allow a bug that Go's static typing prevents. Then create the opposite—a scenario where Go's strictness creates more verbose code for an obviously safe operation.

Chapter 4: Error Handling — The Hardest Shift

If there's one aspect of Go that drives PHP developers crazy, it's error handling. The constant `if err != nil` checks feel primitive, verbose, and frankly annoying.

But error handling is where Go's philosophy shines most clearly. Once you internalise it, you'll understand why many Go developers consider it superior to exceptions.

Why `if err != nil` Feels Wrong at First

Your PHP brain has been trained to expect exceptions:

```php
public function getUser(int $id): User
{
    $user = $this->repository->find($id);
    if (!$user) {
        throw new UserNotFoundException($id);
    }
    return $user;
}

// Caller
try {
    $user = $service->getUser($id);
    // Happy path continues
} catch (UserNotFoundException $e) {
    // Handle error
}
```

The error handling is separated from the main logic. You write the happy path, and exceptions handle the unhappy path elsewhere.

Now look at Go:

```go
func (s *Service) GetUser(id int) (User, error) {
    user, err := s.repository.Find(id)
    if err != nil {
        return User{}, fmt.Errorf("getting user %d: %w", id, err)
    }
    return user, nil
}

// Caller
user, err := service.GetUser(id)
```

```
if err != nil {
    // Handle error
}
// Happy path continues
```

The error check interrupts the flow. Every function that can fail returns an error. Every call site checks it. The happy path is littered with error handling.

This feels *wrong* when you're used to exceptions. Where's the separation of concerns? Why is error handling polluting every function?

The Visibility Trade-off

Consider this PHP code:

```
public function processOrder(Order $order): void
{
    $this->validator->validate($order);
    $payment = $this->paymentGateway->charge($order);
    $this->inventory->reserve($order->getItems());
    $this->mailer->sendConfirmation($order, $payment);
    $this->analytics->track('order.completed', $order);
}
```

Clean, readable, focused. But how many ways can this fail? Each method might throw. The `validate` might throw multiple exception types. The `charge` could fail for network, fraud, or insufficient funds. The `reserve` could fail if items are out of stock.

None of these failure modes are visible. You'd need to read each method's implementation or documentation to know what might happen.

Now the Go version:

```
func (s *Service) ProcessOrder(order Order) error {
    if err := s.validator.Validate(order); err != nil {
        return fmt.Errorf("validating order: %w", err)
    }
    payment, err := s.paymentGateway.Charge(order)
    if err != nil {
        return fmt.Errorf("charging payment: %w", err)
    }
    if err := s.inventory.Reserve(order.Items); err != nil {
        return fmt.Errorf("reserving inventory: %w", err)
    }
    if err := s.mailer.SendConfirmation(order, payment); err != nil {
        return fmt.Errorf("sending confirmation: %w", err)
    }
    if err := s.analytics.Track("order.completed", order); err != nil {
        // Log but don't fail on analytics errors
        s.logger.Error("analytics tracking failed", "error", err)
    }
    return nil
```

```
}
```

More verbose? Yes. But look at what's visible: - Every operation that can fail is marked with `err` - You can see exactly how each error is handled - The analytics error is explicitly logged but not propagated - The error context ("validating order", "charging payment") creates an error trail

Exceptions vs Explicit Errors: The Philosophical Divide

Exceptions and error returns represent fundamentally different philosophies.

Exceptions: Errors as Exceptional Events

The exception model treats errors as *exceptional*—things that shouldn't happen in normal operation. When they occur, control flow jumps to a handler, potentially far up the call stack:

```php
// Deep in the call stack
public function parseConfig(string $json): array
{
    $config = json_decode($json, true);
    if (json_last_error() !== JSON_ERROR_NONE) {
        throw new ConfigParseException(json_last_error_msg());
    }
    return $config;
}

// Far up the call stack
public function bootstrap(): void
{
    try {
        $config = $this->loadConfig();
        $this->initializeServices($config);
        $this->startServer();
    } catch (ConfigParseException $e) {
        // Handle config errors
    } catch (ServiceException $e) {
        // Handle service errors
    } catch (Exception $e) {
        // Catch-all
    }
}
```

The error handling is centralised. The intervening code doesn't need to know about or handle the exceptions—they bubble up automatically.

Error Returns: Errors as Values

Go treats errors as ordinary values—data to be inspected, transformed, and passed along:

```go
// Deep in the call stack
func parseConfig(data []byte) (Config, error) {
    var config Config
    if err := json.Unmarshal(data, &config); err != nil {
        return Config{}, fmt.Errorf("parsing config: %w", err)
    }
    return config, nil
}

// Each level handles or propagates
func loadConfig() (Config, error) {
    data, err := os.ReadFile("config.json")
    if err != nil {
        return Config{}, fmt.Errorf("reading config file: %w", err)
    }
    return parseConfig(data)
}

func bootstrap() error {
    config, err := loadConfig()
    if err != nil {
        return fmt.Errorf("loading config: %w", err)
    }
    // ... continue
}
```

Every function explicitly handles or propagates errors. There's no invisible control flow—you can trace the error path by reading the code linearly.

Why Go Chose Explicit Errors

Go's designers had experience with exceptions in other languages and found them problematic:

1. **Invisible control flow**: Exceptions can jump anywhere, making code flow unpredictable
2. **Easy to forget**: It's easy to omit `catch` blocks for exceptions you didn't know could occur
3. **Cleanup complexity**: `finally` blocks and exception-safe code are error-prone
4. **Performance**: Exception handling has runtime overhead

Error values solve these issues: - Control flow is explicit and linear - The return type forces you to acknowledge errors - Cleanup uses `defer`, which is straightforward - Error handling is just function return overhead

Error Wrapping and the %w Verb

PHP exceptions carry their own context—message, code, stack trace:

```php
throw new OrderException(
    "Payment failed for order $orderId",
    code: OrderException::PAYMENT_FAILED,
    previous: $paymentException
);
```

Go errors are simpler by design, but can be wrapped to build context:

```go
// Basic error
return errors.New("payment failed")

// With context using fmt.Errorf
return fmt.Errorf("order %s: payment failed", orderID)

// Wrapping another error (preserves the original)
return fmt.Errorf("processing order %s: %w", orderID, err)
```

The `%w` verb is crucial. It wraps the error while preserving the original, allowing inspection with `errors.Is` and `errors.As`:

```go
var ErrNotFound = errors.New("not found")

func (r *Repo) Find(id int) (User, error) {
    // ...
    return User{}, ErrNotFound
}

func (s *Service) GetUser(id int) (User, error) {
    user, err := s.repo.Find(id)
    if err != nil {
        return User{}, fmt.Errorf("finding user %d: %w", id, err)
    }
    return user, nil
}

// Caller can check for specific error
user, err := service.GetUser(42)
if errors.Is(err, ErrNotFound) {
    // Handle not found specifically
}
```

The wrapped error message might be `"finding user 42: not found"` but `errors.Is` still recognises it as `ErrNotFound`.

Custom Error Types (Like Symfony's Custom Exceptions)

In PHP, you create custom exceptions by extending Exception:

```php
class ValidationException extends Exception
{
    private array $errors;

    public function __construct(array $errors)
    {
        $this->errors = $errors;
        parent::__construct('Validation failed');
    }
```

```
    public function getErrors(): array
    {
        return $this->errors;
    }
}
```

In Go, any type implementing the `error` interface is an error:

```go
// The error interface
type error interface {
    Error() string
}

// Custom error type
type ValidationError struct {
    Fields map[string]string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("validation failed: %d errors", len(e.Fields))
}

// Usage
func Validate(user User) error {
    errors := make(map[string]string)
    if user.Email == "" {
        errors["email"] = "required"
    }
    if len(errors) > 0 {
        return &ValidationError{Fields: errors}
    }
    return nil
}

// Caller extracts details
var validationErr *ValidationError
if errors.As(err, &validationErr) {
    for field, msg := range validationErr.Fields {
        fmt.Printf("%s: %s\n", field, msg)
    }
}
```

`errors.As` unwraps to find an error of a specific type, similar to `catch (ValidationException $e)` in PHP.

When to Panic (Almost Never)

Go has `panic` for truly exceptional situations:

```go
func mustParseURL(s string) *url.URL {
    u, err := url.Parse(s)
    if err != nil {
        panic(fmt.Sprintf("invalid URL: %s", s))
    }
    return u
}
```

But panic should be rare. It's for:

1. **Programmer errors**: Bugs that indicate broken invariants (like array out of bounds)
2. **Initialisation failures**: When the program can't continue (config missing at startup)
3. **Impossible states**: Conditions that "can't happen" but you want to detect

Never panic for: - User input errors - Network failures - File not found - Any error a caller might want to handle

The convention `Must*` (like `template.Must()`) indicates a function that panics on error—use only with known-good values or during initialisation.

Learning to Love Explicit Error Paths

After enough Go code, something shifts. You start to appreciate:

1. Error Paths Are Visible

Reading Go code, you can trace exactly what happens on failure. No need to check documentation or source code for exception types.

2. Errors Get Context

Each level adds information:

```
"processing order abc123: charging payment: connecting to gateway: dial tcp: connection refuse
```

This error message tells you the entire call path. You know exactly where it failed.

3. Forced Consideration

The return type `(T, error)` forces you to decide: handle it, propagate it, or explicitly ignore it. You can't accidentally forget.

4. Easy Testing

Error paths are just return values—easy to test:

```go
func TestGetUser_NotFound(t *testing.T) {
    repo := &MockRepo{err: ErrNotFound}
    service := NewService(repo)

    _, err := service.GetUser(1)
```

```
    if !errors.Is(err, ErrNotFound) {
        t.Errorf("expected ErrNotFound, got %v", err)
    }
}
```

No More Try/Catch Blocks

The absence of try/catch changes how you structure code:

PHP: Group Operations, Handle Failures Together

```php
try {
    $user = $this->userService->find($id);
    $orders = $this->orderService->findByUser($user);
    $recommendations = $this->recService->forUser($user);
    return compact('user', 'orders', 'recommendations');
} catch (UserNotFoundException $e) {
    throw new NotFoundException("User not found");
} catch (ServiceException $e) {
    $this->logger->error("Service error", ['exception' => $e]);
    throw new InternalErrorException();
}
```

Go: Handle Each Failure Inline

```go
user, err := s.userService.Find(id)
if err != nil {
    if errors.Is(err, ErrNotFound) {
        return nil, NewNotFoundError("user not found")
    }
    return nil, fmt.Errorf("finding user: %w", err)
}

orders, err := s.orderService.FindByUser(user)
if err != nil {
    s.logger.Error("failed to fetch orders", "error", err)
    // Continue without orders (graceful degradation)
    orders = nil
}

recommendations, err := s.recService.ForUser(user)
if err != nil {
    s.logger.Error("failed to fetch recommendations", "error", err)
    recommendations = nil
}

return &Response{User: user, Orders: orders, Recommendations: recommendations}, nil
```

The Go version makes decisions explicit at each step: propagate the error, transform it, log it, or ignore it.

Summary

- `if err != nil` is verbose but makes every failure path visible
- **Errors as values** enable straightforward handling, wrapping, and testing
- **Error wrapping** (`%w`) builds context while preserving the original error
- **Custom error types** carry additional data, like custom exceptions
- **Panic** is for programmer errors, not expected failures
- **Explicit error handling** forces you to consider failures at every step

---

Exercises

1. **Exception Inventory**: List all exception types thrown in a Symfony service class. Convert each to a Go error type or sentinel error. Compare the calling patterns.
2. **Error Context Chain**: Write a Go function that calls three other functions, each of which can fail. Wrap errors at each level with context. Verify the final error message contains the full trace.
3. **Graceful Degradation**: Design a Go service that calls three external APIs. If one fails, the others should still succeed. Compare to implementing the same with PHP exceptions.
4. **Custom Error Type**: Create a Go `ValidationError` type that holds multiple field errors. Implement `Error()` and write code using `errors.As` to extract the field errors.
5. **Panic vs Error**: Identify three scenarios where panic is appropriate and three where it's not. Implement examples of each.
6. **Error Handling Patterns**: Implement three different error handling strategies in Go:
   - Propagate with context
   - Transform to a different error type
   - Log and suppress (with explicit `_ =` ignore)
7. **Test Error Paths**: Write a table-driven test that verifies a function returns the correct error types for different failure scenarios.
8. **PHP to Go Migration**: Take a PHP controller action with multiple try/catch blocks. Convert it to Go with explicit error handling. Count the error checks. Does the code still read cleanly?

Chapter 5: From Classes to Structs

PHP classes are rich constructs with constructors, destructors, inheritance, traits, interfaces, visibility modifiers, magic methods, and more. Go structs are deliberately simple: they're just collections of fields.

This simplicity is jarring at first, but it leads to cleaner designs.

No Constructors: The `New*` Pattern

In PHP, the constructor is special:

```php
class User
{
    public function __construct(
        private string $name,
        private string $email,
        private DateTimeImmutable $createdAt = new DateTimeImmutable(),
    ) {}
}

$user = new User('Alice', 'alice@example.com');
```

Go has no constructors. You create structs directly or via factory functions:

```go
type User struct {
    Name      string
    Email     string
    CreatedAt time.Time
}

// Direct creation (all fields)
user := User{
    Name:      "Alice",
    Email:     "alice@example.com",
    CreatedAt: time.Now(),
}

// Factory function (conventional)
func NewUser(name, email string) *User {
    return &User{
        Name:      name,
        Email:     email,
        CreatedAt: time.Now(),
    }
}
```

```
user := NewUser("Alice", "alice@example.com")
```

The `New*` Convention

The `New*` prefix is Go's convention for factory functions:

- `NewUser(name, email)` — create a User
- `NewServer(config)` — create a Server
- `NewClient(options...)` — create a Client with options

These aren't special—they're just functions that return your type. But they provide:

1. **Validation**: Check invariants before creation
2. **Defaults**: Set fields callers shouldn't specify
3. **Privacy**: Work with unexported fields

```
func NewUser(name, email string) (*User, error) {
    if name == "" {
        return nil, errors.New("name is required")
    }
    if !strings.Contains(email, "@") {
        return nil, errors.New("invalid email")
    }
    return &User{
        Name:      name,
        Email:     email,
        CreatedAt: time.Now(),
        id:        uuid.New(),  // unexported field
    }, nil
}
```

When to Use Direct Struct Literals

Not everything needs a factory function. Use struct literals for:

- **Simple value types**: `Point{X: 10, Y: 20}`
- **Configuration structs**: `Config{Port: 8080, Debug: true}`
- **Test data**: `User{Name: "test"}`

Use `New*` functions when you need:

- **Validation**: Ensure invariants hold
- **Defaults**: Set fields automatically
- **Unexported fields**: Access private state
- **Non-trivial setup**: Connect, initialise, register

Methods as Functions with Receivers

PHP methods live inside the class:

```php
class Calculator
{
    private int $value = 0;

    public function add(int $n): self
    {
        $this->value += $n;
        return $this;
    }

    public function getValue(): int
    {
        return $this->value;
    }
}
```

Go methods are functions declared with a receiver:

```go
type Calculator struct {
    value int
}

func (c *Calculator) Add(n int) *Calculator {
    c.value += n
    return c
}

func (c *Calculator) Value() int {
    return c.value
}
```

The receiver `(c *Calculator)` is like `$this`—it's the instance the method operates on. But there's a key difference: the receiver is *explicit*.

The Explicit Receiver

In PHP, `$this` is implicit—you don't declare it:

```php
public function getName(): string
{
    return $this->name;  // $this appears magically
}
```

In Go, the receiver is part of the function signature:

```go
func (u *User) Name() string {
    return u.name  // u is explicitly declared
}
```

You can name the receiver anything, but convention is:

- Use the first letter of the type: `u` for `User`, `s` for `Server`

- Be consistent within a type
- Avoid generic names like `this` or `self`

Methods Are Just Functions

Syntactically, methods are sugar for functions with the receiver as the first parameter:

```
// Method syntax
func (u *User) Greet() string {
    return "Hello, " + u.Name
}
user.Greet()

// Equivalent function call
(*User).Greet(&user)  // Method expression
```

This isn't just trivia—it means methods can be passed as functions:

```
greet := user.Greet  // Method value
fmt.Println(greet())  // Calls user.Greet()

// Or extract method for a type
greetFunc := (*User).Greet
fmt.Println(greetFunc(&user))
```

Value Receivers vs Pointer Receivers

This is one of Go's most confusing aspects for PHP developers.

PHP: Always References (Sort Of)

In PHP, objects are always passed by reference (technically, by handle):

```
function modify(User $user): void
{
    $user->name = 'Modified';  // Affects the original
}
```

Go: Value vs Pointer Receivers

In Go, you choose:

```
// Value receiver: operates on a copy
func (u User) FullName() string {
    return u.FirstName + " " + u.LastName
}

// Pointer receiver: operates on the original
func (u *User) SetName(name string) {
```

```
    u.FirstName = name  // Modifies the original
}
```

When to Use Which

Use pointer receiver when: - The method modifies the receiver - The struct is large (avoid copying) - Consistency—if some methods need pointers, use pointers for all

Use value receiver when: - The struct is small and immutable - The method doesn't modify state - You want a defensive copy

```
// Small immutable type: value receivers
type Point struct {
    X, Y float64
}

func (p Point) Distance(other Point) float64 {
    dx := p.X - other.X
    dy := p.Y - other.Y
    return math.Sqrt(dx*dx + dy*dy)
}

// Larger mutable type: pointer receivers
type Server struct {
    config Config
    router *Router
    db     *sql.DB
    // ... more fields
}

func (s *Server) HandleRequest(w http.ResponseWriter, r *http.Request) {
    // ...
}
```

The Automatic Dereference

Go automatically takes addresses and dereferences for method calls:

```
user := User{Name: "Alice"}
user.SetName("Bob")  // Go converts to (&user).SetName("Bob")

userPtr := &User{Name: "Carol"}
userPtr.FullName()   // Go converts to (*userPtr).FullName()
```

This convenience can mask whether you're working with a pointer or value—be mindful when it matters.

Where Did `$this` Go?

In PHP, `$this` is always available in non-static methods:

```php
class Service
{
    public function process(): void
    {
        $this->validate();
        $this->save();
        $this->notify();
    }
}
```

In Go, the receiver name replaces `$this`:

```go
func (s *Service) Process() {
    s.validate()
    s.save()
    s.notify()
}
```

The explicit naming has benefits:

1. **Clarity**: You see exactly what `s` is
2. **Shadowing prevention**: No conflict with local `this` variable
3. **Consistency**: Same pattern in all methods

But it requires adjustment. Your fingers will type `$this->` and Go will complain.

Private/Public via Case (No Keywords)

PHP uses visibility keywords:

```php
class User
{
    private string $id;
    protected string $name;
    public string $email;

    private function validate(): void { }
    public function save(): void { }
}
```

Go uses capitalisation:

```go
type User struct {
    id    string  // unexported (private to package)
    Name  string  // exported (public)
    Email string  // exported
}

func (u *User) validate() { }  // unexported
func (u *User) Save() { }      // exported
```

- **Uppercase first letter**: Exported (public)
- **Lowercase first letter**: Unexported (private to the package)

There's no "protected" equivalent. Unexported means only the package can access it—not sub-packages, not embedding types.

Package-Level Privacy

Importantly, unexported fields are visible to all code in the same package:

```go
// user.go
type User struct {
    id   string
    Name string
}

// repository.go (same package)
func (r *Repo) Save(u *User) error {
    // Can access u.id because we're in the same package
    return r.db.Exec("INSERT INTO users (id, name) VALUES (?, ?)", u.id, u.Name)
}
```

This is different from PHP's private, which restricts access to the class itself.

Symfony Services vs Go Structs

Let's convert a typical Symfony service to Go.

Symfony Service

```php
#[AsService]
class OrderService
{
    public function __construct(
        private OrderRepository $repository,
        private PaymentGateway $payment,
        private MailerInterface $mailer,
        private LoggerInterface $logger,
    ) {}

    public function createOrder(Cart $cart): Order
    {
        $order = Order::fromCart($cart);

        $this->repository->save($order);

        $this->payment->charge($order);

        $this->mailer->send(
            new OrderConfirmationEmail($order)
        );
```

```
        $this->logger->info('Order created', ['id' => $order->getId()]);

        return $order;
    }
}
```

Go Equivalent

```
type OrderService struct {
    repository OrderRepository  // interface
    payment    PaymentGateway   // interface
    mailer     Mailer           // interface
    logger     *slog.Logger
}

func NewOrderService(
    repo OrderRepository,
    payment PaymentGateway,
    mailer Mailer,
    logger *slog.Logger,
) *OrderService {
    return &OrderService{
        repository: repo,
        payment:    payment,
        mailer:     mailer,
        logger:     logger,
    }
}

func (s *OrderService) CreateOrder(cart Cart) (Order, error) {
    order := OrderFromCart(cart)

    if err := s.repository.Save(order); err != nil {
        return Order{}, fmt.Errorf("saving order: %w", err)
    }

    if err := s.payment.Charge(order); err != nil {
        return Order{}, fmt.Errorf("charging payment: %w", err)
    }

    if err := s.mailer.Send(NewOrderConfirmationEmail(order)); err != nil {
        s.logger.Error("failed to send confirmation", "error", err, "order_id", order.ID)
        // Don't fail on email errors
    }

    s.logger.Info("order created", "id", order.ID)

    return order, nil
}
```

Key differences:

1. **No autowiring**: Dependencies are passed explicitly
2. **Factory function**: `NewOrderService` replaces constructor
3. **Error handling**: Each operation returns an error
4. **No attributes**: Configuration is explicit code

Summary

- **No constructors**: Use `New*` factory functions for initialisation
- **Explicit receivers**: Methods declare their receiver like a parameter
- **Value vs pointer receivers**: Choose based on mutation and size
- **Case-based visibility**: Uppercase = exported, lowercase = unexported
- **Package-level privacy**: No class-level private—only package boundaries

---

Exercises

1. **Constructor Migration**: Take three PHP classes with different constructor patterns (required parameters, optional with defaults, many dependencies). Convert each to Go using `New*` functions.
2. **Receiver Selection**: Write a Go type with 5 methods. Decide for each method whether it should use a value or pointer receiver. Justify each choice.
3. **Method Expression**: Write a Go program that extracts a method from a struct and passes it to another function. When would this be useful?
4. **Visibility Audit**: Take a PHP class with mixed visibility (private, protected, public). Convert to Go and note which fields would need restructuring due to package-level privacy.
5. **Zero Value Safety**: Design a Go struct where the zero value is invalid. Then redesign it so the zero value is usable. Which design is better?
6. **Builder Pattern**: Implement the builder pattern in Go for a complex struct. Compare to PHP's fluent setters.
7. **Service Conversion**: Convert a Symfony service with 5+ dependencies to Go. Include the wiring code in `main.go`. Count the lines of explicit wiring versus Symfony's implicit wiring.
8. **Immutable Types**: Design an immutable `Money` type in Go with value receiver methods that return new values. Compare to a mutable PHP Money class.

Chapter 6: Inheritance Is Dead — Long Live Composition

If you've been using PHP for years, inheritance is deeply wired into your thinking. Base classes, abstract methods, parent::calls—these are fundamental tools in your mental toolkit.

Go has no inheritance. None. This isn't a limitation to work around; it's a deliberate design choice that leads to better code.

Why Go Has No Inheritance

Inheritance creates several problems at scale:

1. The Fragile Base Class Problem

When you modify a base class, you might break subclasses in unexpected ways:

```
abstract class PaymentProcessor
{
    public function process(Payment $payment): void
    {
        $this->validate($payment);  // Added in version 2
        $this->doProcess($payment);
    }

    protected function validate(Payment $payment): void
    {
        if ($payment->getAmount() <= 0) {
            throw new InvalidPaymentException();
        }
    }

    abstract protected function doProcess(Payment $payment): void;
}
```

Now every subclass's `doProcess` must handle pre-validated payments. If a subclass was doing its own validation, there's duplicate validation. If the base validation is too strict for some subclass, tough luck.

2. The Diamond Problem

PHP solves this with traits and explicit conflict resolution, but it's inherently complex:

```php
trait Loggable {
    public function log(string $msg): void { /* ... */ }
}

trait Auditable {
    public function log(string $msg): void { /* ... */ }  // Conflict!
}

class Service {
    use Loggable, Auditable {
        Loggable::log insteadof Auditable;
        Auditable::log as auditLog;
    }
}
```

3. Deep Hierarchies

Inheritance encourages deep hierarchies:

```
Entity
    TimestampedEntity
        SoftDeletableEntity
            User
                AdminUser
```

Understanding `AdminUser` requires understanding four parent classes. Debugging means jumping between files. Changes ripple unpredictably.

Go's Solution: Don't Provide It

Rather than solving inheritance problems, Go simply doesn't offer inheritance. Instead, it provides:

- **Composition**: Structs containing other structs
- **Embedding**: Composition with syntactic sugar
- **Interfaces**: Behavioural contracts without hierarchies

Embedding: "Inheritance" Without Hierarchy

Go's embedding provides some inheritance-like behaviour:

```go
type Animal struct {
    Name string
}

func (a *Animal) Speak() string {
    return "..."
}

type Dog struct {
    Animal  // Embedded
    Breed   string
```

```
}

func (d *Dog) Speak() string {
    return "Woof!"
}

// Usage
dog := Dog{Animal: Animal{Name: "Rex"}, Breed: "German Shepherd"}
fmt.Println(dog.Name)      // Promoted from Animal
fmt.Println(dog.Speak())   // Dog's method, not Animal's
```

The embedded `Animal` fields and methods are "promoted" to `Dog`. You can access `dog.Name` instead of `dog.Animal.Name`.

What Embedding Is Not

Embedding looks like inheritance but isn't:

```
var animal *Animal = &dog  // Error! Dog is not an Animal
```

A `Dog` doesn't substitute for an `Animal`. There's no subtype relationship. Embedding is purely syntactic convenience for composition.

Embedding vs Inheritance

Inheritance	Embedding
`Dog IS-A Animal`	`Dog HAS-A Animal`
Subtype relationship	No type relationship
Virtual dispatch	Method promotion
Parent can reference child	Embedded doesn't know embedder
`protected` access	No special access

Interface Composition

Instead of inheriting behaviour, Go composes interfaces:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

`ReadWriter` embeds `Reader` and `Writer`. Any type implementing both interfaces automatically implements `ReadWriter`.

This approach is more flexible than class inheritance:

```
// PHP: Must explicitly declare interfaces
class FileHandler implements Reader, Writer { }
```

```
// Go: Implements ReadWriter automatically if it has both methods
type FileHandler struct { /* ... */ }
func (f *FileHandler) Read(p []byte) (int, error) { /* ... */ }
func (f *FileHandler) Write(p []byte) (int, error) { /* ... */ }
// FileHandler implements Reader, Writer, AND ReadWriter
```

The PHP Developer's Temptation to Fake Inheritance

Coming from PHP, you might try to recreate inheritance:

Don't: Recreating Abstract Classes

```
// Tempting but wrong
type BaseService struct {
    logger *slog.Logger
}

func (b *BaseService) Log(msg string) {
    b.logger.Info(msg)
}

type UserService struct {
    BaseService
    repo UserRepository
}

type OrderService struct {
    BaseService
    repo OrderRepository
}
```

This looks like inheritance but has problems:

1. `UserService` and `OrderService` don't share a type
2. Changes to `BaseService` affect both opaquely
3. It's fighting Go's design

Do: Use Composition Explicitly

```
type Logger interface {
    Info(msg string)
}

type UserService struct {
    logger Logger
    repo   UserRepository
}

func (s *UserService) CreateUser(u User) error {
    s.logger.Info("creating user")
    // ...
}

type OrderService struct {
    logger Logger
    repo   OrderRepository
}

func (s *OrderService) CreateOrder(o Order) error {
    s.logger.Info("creating order")
    // ...
}
```

Each service has a logger, explicitly. The relationship is clear. Testing is straightforward—inject mock loggers.

Don't: Deep Embedding Chains

```
// Tempting but problematic
type Entity struct {
    ID        uuid.UUID
    CreatedAt time.Time
}

type SoftDeletable struct {
    Entity
    DeletedAt *time.Time
}

type User struct {
    SoftDeletable
    Name  string
    Email string
}
```

This recreates PHP's deep hierarchies. Go encourages flat structures:

```go
type User struct {
    ID        uuid.UUID
    CreatedAt time.Time
    DeletedAt *time.Time  // nullable for soft delete
    Name      string
    Email     string
}
```

Is there code duplication if `Order` has the same fields? Yes, a little. But each type is independent, understandable, and modifiable without affecting others.

Flattening Deep Hierarchies

When converting PHP code with deep inheritance, flatten aggressively.

PHP: Deep Hierarchy

```php
abstract class Controller
{
    protected function render(string $template, array $data): Response;
    protected function json($data): JsonResponse;
    protected function redirect(string $url): RedirectResponse;
}

abstract class ApiController extends Controller
{
    protected function validate(Request $request, array $rules): array;
    protected function paginate(QueryBuilder $query): Paginator;
}

class UserApiController extends ApiController
{
    public function index(Request $request): JsonResponse
    {
        $validated = $this->validate($request, ['page' => 'integer']);
        $users = $this->paginate($this->userQuery);
        return $this->json($users);
    }
}
```

Go: Flat Composition

```go
type UserHandler struct {
    repo       UserRepository
    validator  Validator
    paginator  Paginator
}

func (h *UserHandler) List(w http.ResponseWriter, r *http.Request) {
```

```
    params, err := h.validator.Validate(r, ListUsersRequest{})
    if err != nil {
        writeError(w, err)
        return
    }

    users, err := h.paginator.Paginate(h.repo.Query(), params.Page)
    if err != nil {
        writeError(w, err)
        return
    }

    writeJSON(w, users)
}

// Helper functions, not inherited methods
func writeJSON(w http.ResponseWriter, data any) {
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(data)
}

func writeError(w http.ResponseWriter, err error) {
    // Error handling
}
```

The `UserHandler` doesn't inherit from anything. It composes the pieces it needs. Helper functions live in the package, not a base class.

Doctrine Entities Without Inheritance

Doctrine encourages inheritance for common entity behaviour:

```
/** @MappedSuperclass */
abstract class BaseEntity
{
    /** @Column(type="uuid") */
    protected UuidInterface $id;

    /** @Column(type="datetime_immutable") */
    protected DateTimeImmutable $createdAt;

    /** @Column(type="datetime_immutable", nullable=true) */
    protected ?DateTimeImmutable $updatedAt = null;
}

/** @Entity */
class User extends BaseEntity
{
    /** @Column */
    private string $name;
}
```

In Go, you typically define each type fully:

```
type User struct {
    ID        uuid.UUID
    CreatedAt time.Time
    UpdatedAt *time.Time
    Name      string
}

type Order struct {
    ID        uuid.UUID
    CreatedAt time.Time
    UpdatedAt *time.Time
    UserID    uuid.UUID
    Total     decimal.Decimal
}
```

If you truly need shared fields, you can embed—but carefully:

```
type Timestamps struct {
    CreatedAt time.Time
    UpdatedAt *time.Time
}

type User struct {
    ID uuid.UUID
    Timestamps
    Name string
}

// Accessing fields
user.CreatedAt     // Works via promotion
user.Timestamps.CreatedAt  // Also works
```

But consider: is the embedding adding value, or just reducing a few lines of duplication while adding abstraction?

Summary

- **Go has no inheritance**—this is intentional, not a limitation
- **Embedding** provides method promotion but not subtyping
- **Interface composition** creates flexible behavioural contracts
- **Flat structures** are preferred over deep embedding chains
- **Fight the temptation** to recreate inheritance patterns

Exercises

1. **Hierarchy Flattening**: Take a PHP class hierarchy with 3+ levels. Convert to Go with flat structs and composition. Compare the dependency graph before and after.

2. **Interface Extraction**: Find a PHP class that extends a base class. Identify what behaviour it inherits. Define Go interfaces for that behaviour instead.
3. **Embedding Evaluation**: Write a Go struct that embeds another struct. Then rewrite it without embedding, using explicit fields. Which is clearer?
4. **Trait Replacement**: Take PHP code using traits. Convert to Go using either embedding or composition. Which approach better matches Go idioms?
5. **Template Method Refactor**: Find a PHP class using the Template Method pattern (abstract base class with hook methods). Refactor to Go using interfaces and composition.
6. **Entity Duplication Analysis**: Write Go structs for 5 related entities (User, Order, Product, Review, Category). Note the duplicated fields. Decide whether embedding helps or hurts.
7. **Decorator Pattern**: Implement the decorator pattern in PHP using inheritance, then in Go using interface wrapping. Compare the flexibility and testability.
8. **Inheritance Smell Detection**: Review PHP code for these inheritance smells:
   - Deep hierarchies (>2 levels)
   - Base classes with many abstract methods
   - Subclasses that override most methods

   For each smell, design a Go alternative using composition.

Chapter 7: Interfaces — Go's Hidden Superpower

PHP interfaces are explicit contracts. You declare `implements`, and the class must provide all methods. Go interfaces work differently—and this difference is profound.

Implicit Satisfaction (No `implements`)

In PHP, the relationship is declared:

```php
interface Logger
{
    public function info(string $message): void;
    public function error(string $message): void;
}

class FileLogger implements Logger
{
    public function info(string $message): void { /* ... */ }
    public function error(string $message): void { /* ... */ }
}
```

In Go, satisfaction is implicit:

```go
type Logger interface {
    Info(msg string)
    Error(msg string)
}

type FileLogger struct {
    file *os.File
}

func (l *FileLogger) Info(msg string) {
    fmt.Fprintln(l.file, "INFO:", msg)
}

func (l *FileLogger) Error(msg string) {
    fmt.Fprintln(l.file, "ERROR:", msg)
}

// FileLogger implements Logger automatically
// No declaration needed
```

`FileLogger` implements `Logger` because it has the required methods. No `implements` keyword. No explicit declaration. The compiler figures it out.

Why Implicit Is Powerful

This enables decoupled design:

1. **You can define interfaces where they're used**, not where implementations live
2. **Third-party types can satisfy your interfaces** without modification
3. **Small interfaces are trivial to create** after the fact

```go
// In your package
type Storer interface {
    Store(key string, value []byte) error
}

// A third-party type might already satisfy this
// without knowing about your interface
type RedisClient struct { /* ... */ }
func (c *RedisClient) Store(key string, value []byte) error { /* ... */ }

// Your code works with RedisClient automatically
func SaveData(s Storer, key string, data []byte) error {
    return s.Store(key, data)
}
```

In PHP, you'd need `RedisClient` to declare `implements Storer`—which requires modifying third-party code or wrapping it.

Small Interfaces: The `io.Reader` Philosophy

PHP interfaces often have many methods:

```php
interface RepositoryInterface
{
    public function find(int $id): ?Entity;
    public function findAll(): array;
    public function findBy(array $criteria): array;
    public function save(Entity $entity): void;
    public function delete(Entity $entity): void;
    public function count(): int;
}
```

Go favours tiny interfaces:

```go
// The famous io.Reader - just one method
type Reader interface {
    Read(p []byte) (n int, err error)
}

// io.Writer - just one method
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

```go
// io.Closer - just one method
type Closer interface {
    Close() error
}

// Compose them as needed
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

Why Small Interfaces Win

More types satisfy them: A one-method interface is easy to implement. Many existing types accidentally implement `io.Reader`.

Better composition: Combine small interfaces into exactly what you need:

```go
// A function that only needs reading
func Process(r io.Reader) error { /* ... */ }

// A function that needs reading and closing
func ProcessAndClose(r io.ReadCloser) error {
    defer r.Close()
    return Process(r)
}
```

Easier testing: Mock one method, not ten:

```go
type mockReader struct {
    data []byte
}

func (m *mockReader) Read(p []byte) (int, error) {
    copy(p, m.data)
    return len(m.data), io.EOF
}
```

The Interface Segregation Principle by Default

PHP's SOLID principles include Interface Segregation: "Clients should not be forced to depend on methods they do not use."

Go makes this the natural state. When you create a one-method interface, clients only depend on that one method.

Accept Interfaces, Return Structs

This Go proverb captures a key design pattern.

Accept Interfaces

Functions should accept the minimal interface they need:

```
// Good: accepts io.Reader
func ParseJSON(r io.Reader, v any) error {
    return json.NewDecoder(r).Decode(v)
}

// Can be called with:
ParseJSON(os.Stdin, &config)           // *os.File
ParseJSON(resp.Body, &data)            // http.Response.Body
ParseJSON(bytes.NewReader(b), &msg)    // *bytes.Reader
ParseJSON(strings.NewReader(s), &doc)  // *strings.Reader
```

Return Structs

Functions should return concrete types:

```
// Good: returns concrete *Server
func NewServer(config Config) *Server {
    return &Server{config: config}
}

// Not: returns interface
func NewServer(config Config) ServerInterface {
    return &Server{config: config}  // Unnecessary abstraction
}
```

Why? Returning interfaces: - Hides what the caller actually gets - Prevents access to type-specific methods - Adds a layer of indirection without benefit

The caller can always store the result in an interface variable if they want:

```
var s ServerInterface = NewServer(cfg)  // Caller's choice
```

PHP Comparison

PHP often does the opposite:

```
// Common PHP pattern
interface UserRepositoryInterface { /* ... */ }

class DoctrineUserRepository implements UserRepositoryInterface { /* ... */ }

// Returns interface
public function getUserRepository(): UserRepositoryInterface
{
    return $this->userRepository;
}
```

This is idiomatic PHP for dependency injection. In Go, you'd return the concrete type and accept interfaces where needed.

The Empty Interface and When to Avoid It

Go's `any` (alias for `interface{}`) is like PHP's `mixed`:

```
func PrintAnything(v any) {
    fmt.Println(v)
}

PrintAnything(42)
PrintAnything("hello")
PrintAnything(User{Name: "Alice"})
```

Legitimate Uses

1. **Serialisation**: `json.Marshal(v any)`
2. **Logging**: `log.Printf("%v", value)`
3. **Generic containers** (pre-generics code)

When to Avoid

Most uses of `any` indicate a design problem:

```
// Bad: loses type safety
func Process(data any) any {
    // Now you need type switches everywhere
}

// Better: use generics
func Process[T any](data T) T {
    // T is still a type parameter, but preserved through the function
}

// Or: use specific interfaces
func Process(data Processable) Result {
    return data.Process()
}
```

The `any` Smell Test

Ask yourself: "Why don't I know the type here?"

- **I'm writing a library for many types** → Consider generics
- **The type varies at runtime** → Define an interface for the common behaviour
- **I really don't care about the type** → `any` might be appropriate (logging, debugging)
- **I'm being lazy** → Define proper types

Comparing to Symfony's Interface-Driven Design

Symfony uses interfaces extensively:

```php
interface EventDispatcherInterface
{
    public function dispatch(object $event): object;
    public function addListener(string $eventName, callable $listener): void;
    public function addSubscriber(EventSubscriberInterface $subscriber): void;
    public function removeListener(string $eventName, callable $listener): void;
    public function removeSubscriber(EventSubscriberInterface $subscriber): void;
    public function getListeners(?string $eventName = null): array;
    public function hasListeners(?string $eventName = null): bool;
}
```

This interface has seven methods. Any implementation must provide all of them, even if a consumer only needs `dispatch`.

Go's Approach

In Go, you'd define interfaces at the point of use:

```go
// In a package that just dispatches events
type Dispatcher interface {
    Dispatch(ctx context.Context, event Event) error
}

// In a package that manages listeners
type ListenerManager interface {
    AddListener(eventName string, listener Listener)
    RemoveListener(eventName string, listener Listener)
}

// The full implementation satisfies both
type EventDispatcher struct { /* ... */ }
func (d *EventDispatcher) Dispatch(ctx context.Context, event Event) error { /* ... */ }
func (d *EventDispatcher) AddListener(eventName string, listener Listener) { /* ... */ }
func (d *EventDispatcher) RemoveListener(eventName string, listener Listener) { /* ... */ }
```

Each consumer depends only on what it needs. Testing is simpler. Dependencies are minimal.

Interface Location

In PHP, interfaces live with (or near) their implementations.

In Go, interfaces live with their consumers:

```
// PHP structure
src/EventDispatcher/EventDispatcherInterface.php
src/EventDispatcher/EventDispatcher.php

// Go structure
```

```
eventdispatcher/dispatcher.go  // Implementation only

// Consumer defines its own interface
orderservice/service.go
    type Dispatcher interface {
        Dispatch(ctx context.Context, event Event) error
    }
```

This inverts the dependency—implementations don't know about consumers' interfaces.

Summary

- **Implicit satisfaction** enables powerful decoupling
- **Small interfaces** (1-3 methods) are idiomatic Go
- **Accept interfaces, return structs** for flexible APIs
- **Avoid `any`** unless you genuinely don't care about the type
- **Define interfaces where used**, not where implemented

Exercises

1. **Interface Extraction**: Take a PHP class with 5+ methods. Define the smallest Go interface(s) that different consumers would actually need.
2. **Third-Party Satisfaction**: Find a third-party Go package type. Define an interface in your code that it satisfies without modification. Use it in a function.
3. **Interface Composition**: Create three single-method interfaces. Compose them into two different combined interfaces. Write functions accepting each combination.
4. **Refactor Away `any`**: Find Go code using `any` or `interface{}`. Refactor to use specific types or generics. What type information was lost with `any`?
5. **Consumer-Defined Interfaces**: Take a Go package that returns concrete types. In a separate package, define interfaces for only the methods you need. Verify the types satisfy them.
6. **Mock Creation**: Write a one-method interface. Create a mock implementation for testing. Compare this to mocking a PHP interface with 10 methods.
7. **Interface Location Analysis**: In a PHP Symfony project, note where interfaces are defined (with implementations or separately). Redesign for Go's consumer-defined pattern.
8. **The `any` Audit**: Search a Go codebase for uses of `any` or `interface{}`. Categorise each use as legitimate or avoidable. Propose refactors for the avoidable ones.

Chapter 8: Packages and Modules

PHP's Composer revolutionised dependency management. Go's module system takes a different approach—simpler in some ways, stricter in others. Understanding these differences is key to structuring Go projects effectively.

No Autoloading: Explicit Imports

PHP's autoloading is magical:

```php
// No require statements needed
use App\Service\UserService;
use App\Repository\UserRepository;
use Symfony\Component\Mailer\MailerInterface;

class SomeController
{
    public function __construct(
        private UserService $userService,  // Autoloaded
    ) {}
}
```

Composer's autoloader finds classes based on namespace conventions. You never write `require` statements.

Go requires explicit imports:

```go
package main

import (
    "context"
    "fmt"
    "log/slog"

    "github.com/yourorg/yourproject/internal/service"
    "github.com/yourorg/yourproject/internal/repository"
)

func main() {
    // Use imported packages
    repo := repository.NewUserRepository(db)
    svc := service.NewUserService(repo)
}
```

Every package you use must be imported. The compiler enforces this—unused imports are errors.

Import Paths

Import paths are URLs (without the protocol):

```
import (
    // Standard library
    "encoding/json"
    "net/http"

    // Third-party
    "github.com/gin-gonic/gin"
    "github.com/jmoiron/sqlx"

    // Your project
    "github.com/yourorg/yourproject/internal/config"
)
```

This explicitness has benefits: - You can see all dependencies at a glance - No magic resolution rules to remember - Tooling can analyse imports statically

go.mod vs composer.json

Both files declare project dependencies, but they work differently.

Composer's Approach

```
{
    "name": "myorg/myproject",
    "require": {
        "php": "^8.2",
        "symfony/framework-bundle": "^6.3",
        "doctrine/orm": "^2.15"
    },
    "require-dev": {
        "phpunit/phpunit": "^10.0"
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Composer: - Supports semantic version constraints (`^6.3`, `~2.15`) - Separates dev dependencies - Configures autoloading - Uses `composer.lock` for exact versions

Go's Approach

```
// go.mod
module github.com/yourorg/yourproject

go 1.21

require (
    github.com/gin-gonic/gin v1.9.1
    github.com/jmoiron/sqlx v1.3.5
)

require (
    // indirect dependencies listed here
    github.com/go-playground/validator/v10 v10.14.0 // indirect
)
```

Go modules: - Use exact versions (no ^ or ~) - Don't separate dev dependencies (use build tags instead) - No autoload configuration (Go has fixed conventions) - Use `go.sum` for checksums

Version Selection

Composer picks the highest version satisfying all constraints.

Go uses **Minimal Version Selection (MVS)**: it picks the minimum version that satisfies all requirements. This is more predictable—adding a dependency can't unexpectedly upgrade another.

Internal Packages: Visibility Control

Go has a special `internal` directory that restricts imports:

```
myproject/
  cmd/
      server/
          main.go
  internal/           # Can only be imported by myproject
      config/
      repository/
      service/
  pkg/                # Can be imported by anyone (convention)
      utils/
  go.mod
```

Code in `internal/` can only be imported by packages within the same module (or parent directory). External projects cannot import your internal packages.

This enforces public API boundaries:

```
// External project
import "github.com/yourorg/yourproject/internal/service"  // Error!
```

```
import "github.com/yourorg/yourproject/pkg/utils"         // OK
```

PHP Comparison

PHP has no equivalent. Any class in `vendor/` can be used:

```
// Using Symfony's internal classes (bad practice but possible)
use Symfony\Component\HttpKernel\Internal\SomeInternalClass;
```

The `@internal` annotation is advisory only—it doesn't prevent imports.

No Circular Imports: Designing for DAGs

Go forbids circular imports:

```
// package a
import "myproject/b"  // a imports b

// package b
import "myproject/a"  // b imports a - ERROR!
```

This seems restrictive but enforces good design. Dependencies must form a Directed Acyclic Graph (DAG).

PHP's Circular Dependencies

PHP allows circular dependencies:

```
// UserService.php
use App\Service\OrderService;

class UserService
{
    public function __construct(private OrderService $orders) {}
}

// OrderService.php
use App\Service\UserService;

class OrderService
{
    public function __construct(private UserService $users) {}
}
```

Symfony's container resolves this at runtime via lazy loading. But circular dependencies indicate design problems—the classes are too tightly coupled.

Breaking Cycles in Go

When you hit a circular import, you must restructure:

Option 1: Extract shared code to a third package

```
Before:
a → b → a (cycle)

After:
a → common
b → common
```

Option 2: Use interfaces to invert dependencies

```go
// package user
type OrderFinder interface {
    FindByUser(userID string) ([]Order, error)
}

type Service struct {
    orders OrderFinder  // Interface, not concrete type
}

// package order
type Service struct { /* ... */ }

func (s *Service) FindByUser(userID string) ([]Order, error) { /* ... */ }
```

Now `user` depends on an interface it defines, not on the `order` package.

Vendor vs Module Proxy

Composer downloads packages to `vendor/` and commits or ignores it.

Go has two modes:

Module Proxy (Default)

Go downloads modules from a proxy (default: `proxy.golang.org`):

```
# Downloads to module cache
go mod download

# Module cache location
~/go/pkg/mod/
```

The proxy: - Caches modules for availability - Provides checksums for verification - Speeds up downloads globally

Vendoring (Optional)

You can vendor dependencies:

```
go mod vendor
```

This creates a `vendor/` directory like Composer. Use when: - You need reproducible builds without network - You want to audit or patch dependencies - Your CI doesn't have proxy access

Most Go projects use the proxy, not vendoring.

Migrating a Composer Mindset

Several mental shifts are needed:

1. One Package Per Directory

PHP: Multiple classes per file, namespaces independent of directories.

Go: One package per directory. All `.go` files in a directory are the same package:

```
repository/
  user.go       // package repository
  order.go      // package repository
  product.go    // package repository
```

2. Package Names Are Short

PHP: `App\Repository\User\UserRepository`

Go: `repository.NewUserRepository()`

Package names are typically one word. The full path provides context:

```go
import "myproject/internal/repository"

repo := repository.NewUser(db)  // "repository.NewUser" is clear enough
```

3. No Private Packages

PHP: You can have `private` Composer packages.

Go: All modules are public by default. Private modules require: - GOPRIVATE environment variable - Git authentication configuration - Private proxy (Athens, Artifactory)

4. No Package Versions in Import Paths (Usually)

PHP: Namespace is independent of version.

Go: Major versions 2+ must be in the import path:

```go
import "github.com/example/lib/v2"  // Major version 2
import "github.com/example/lib/v3"  // Major version 3
```

This allows using multiple major versions simultaneously.

Flex Recipes vs Go's Simplicity

Symfony Flex provides recipes that configure packages automatically:

```
composer require symfony/mailer
# Flex: Creates config/packages/mailer.yaml
# Flex: Adds MAILER_DSN to .env
# Flex: Registers bundles
```

Go has no equivalent. Adding a package means:

1. `go get github.com/example/package`
2. Write code that uses it
3. That's it

No automatic configuration, no generated files, no magic wiring.

This simplicity means: - Less to learn - Fewer surprises - More explicit code - More manual setup

Summary

- **Explicit imports** replace autoloading
- **`go.mod`** uses exact versions and minimal version selection
- **`internal/`** enforces API boundaries
- **No circular imports** forces DAG dependency structures
- **Module proxy** replaces `vendor/` by default
- **Simpler packaging** means more explicit code

Exercises

1. **Import Analysis**: Run `go mod graph` on a Go project and `composer show -t` on a PHP project. Compare the dependency tree structures.
2. **Circular Dependency Break**: Create a Go project with two packages that should be circular. Use interfaces to break the cycle without merging them.
3. **Internal Package Design**: Design a Go project structure with clear public API (`pkg/`) and private implementation (`internal/`). What goes where?
4. **Version Selection Comparison**: In a PHP project, add a dependency that requires an older version of an existing dependency. How does Composer resolve it? Then try similar in Go.
5. **Package Naming**: Take a PHP project's namespace structure. Propose a Go package structure with idiomatic short names.
6. **Vendor vs Proxy**: Set up a Go project with vendoring. Then remove `vendor/` and use the proxy. Compare workflow and reproducibility.

7. **Recipe Replacement**: Take a Symfony Flex recipe (e.g., for mailer). List all automatic changes it makes. Write equivalent manual Go setup.
8. **DAG Verification**: Draw the package dependency graph for a Go project. Verify it's a DAG. Then draw one for a PHP project—is it also a DAG?

Chapter 9: The Standard Library Is Your Framework

PHP developers reach for frameworks instinctively. Symfony, Laravel, Slim—these provide the foundation for most PHP applications. Go developers often don't use frameworks at all. The standard library is comprehensive enough for most needs.

Why Go Doesn't Need Symfony

Symfony provides: - HTTP handling (HttpFoundation, HttpKernel) - Routing (Routing component) - Dependency injection (DependencyInjection component) - Configuration (Config, Yaml, Dotenv) - Serialisation (Serializer) - Validation (Validator) - Database abstraction (Doctrine DBAL) - Templating (Twig) - Caching (Cache) - Logging (Monolog integration)

Go's standard library provides equivalents for most of these:

Symfony Component	Go Standard Library
HttpFoundation	`net/http`
Routing	`net/http` (1.22+)
Serializer	`encoding/json`, `encoding/xml`
Validator	(none—use patterns or packages)
Doctrine DBAL	`database/sql`
Twig	`html/template`, `text/template`
Cache	(none—use packages)
Monolog	`log/slog`

The gaps are intentional. Go philosophy says: if it can't be done well generically, don't include it.

`net/http` vs Symfony HttpFoundation

Symfony wraps PHP's superglobals in objects:

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$name = $request->query->get('name', 'World');

$response = new Response(
    "Hello, $name!",
    Response::HTTP_OK,
    ['Content-Type' => 'text/plain']
```

```
);
$response->send();
```

Go's `net/http` provides similar abstractions:

```
func handler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("name")
    if name == "" {
        name = "World"
    }

    w.Header().Set("Content-Type", "text/plain")
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, "Hello, %s!", name)
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Request Object Comparison

```
// Symfony Request
$request->getMethod();           // GET, POST, etc.
$request->getPathInfo();         // /users/123
$request->query->get('page');    // Query params
$request->request->get('name');  // POST body
$request->headers->get('Accept'); // Headers
$request->getContent();          // Raw body
```

```
// Go http.Request
r.Method              // GET, POST, etc.
r.URL.Path            // /users/123
r.URL.Query().Get("page")  // Query params
r.FormValue("name")   // POST body (form-encoded)
r.Header.Get("Accept") // Headers
io.ReadAll(r.Body)    // Raw body
```

Response Writing

Symfony builds a Response object, then sends it.

Go writes directly to the ResponseWriter:

```
func handler(w http.ResponseWriter, r *http.Request) {
    // Set headers before writing body
    w.Header().Set("Content-Type", "application/json")
```

```
    // WriteHeader sets the status (optional-defaults to 200)
    w.WriteHeader(http.StatusCreated)

    // Write body (implements io.Writer)
    json.NewEncoder(w).Encode(map[string]string{"status": "ok"})
}
```

The streaming model is different—you can't modify headers after writing body bytes.

encoding/json vs Symfony Serializer

Symfony Serializer is powerful and complex:

```
use Symfony\Component\Serializer\SerializerInterface;

class UserController
{
    public function show(User $user, SerializerInterface $serializer): Response
    {
        return new Response(
            $serializer->serialize($user, 'json', ['groups' => ['public']]),
            200,
            ['Content-Type' => 'application/json']
        );
    }
}
```

Features include: - Serialisation groups - Custom normalisers - Multiple formats (JSON, XML, CSV) - Object denormalisation - Circular reference handling

Go's `encoding/json` is simpler:

```
type User struct {
    ID       int    `json:"id"`
    Name     string `json:"name"`
    Email    string `json:"email"`
    Password string `json:"-"`  // Excluded
}

func showUser(w http.ResponseWriter, r *http.Request) {
    user := getUserFromDB()

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(user)
}
```

Struct tags control JSON encoding: - `json:"name"` — field name in JSON - `json:"-"` — exclude field - `json:"name,omitempty"` — omit if zero value - `json:",string"` — encode number as string

What Go Lacks

- **Serialisation groups**: Use different structs or custom marshalling
- **Custom normalisers**: Implement `json.Marshaler` interface
- **Circular references**: Handle manually (or redesign)

```
// Custom JSON marshalling
type User struct {
    BirthDate time.Time
}

func (u User) MarshalJSON() ([]byte, error) {
    type Alias User
    return json.Marshal(&struct {
        Alias
        BirthDate string `json:"birth_date"`
    }{
        Alias:     Alias(u),
        BirthDate: u.BirthDate.Format("2006-01-02"),
    })
}
```

database/sql vs Doctrine DBAL

Doctrine DBAL provides: - Query builders - Schema abstraction - Multiple database support - Connection pooling - Type mapping

Go's `database/sql` is lower-level:

```
import (
    "database/sql"
    _ "github.com/lib/pq"  // PostgreSQL driver
)

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/dbname")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // Query
    rows, err := db.QueryContext(ctx, "SELECT id, name FROM users WHERE active = $1", true)
    if err != nil {
        return err
    }
    defer rows.Close()

    var users []User
    for rows.Next() {
        var u User
```

```
        if err := rows.Scan(&u.ID, &u.Name); err != nil {
            return err
        }
        users = append(users, u)
    }
}
```

Key Differences

Connection Pooling: Built into `database/sql`—it manages a pool automatically.

Type Safety: You scan into Go types directly. No result arrays.

No Query Builder: Write SQL strings. Use libraries like `sqlx` or `squirrel` if needed.

Prepared Statements: Built-in via `db.Prepare()` or automatic with `db.Query()`.

```
// Single row
var name string
err := db.QueryRowContext(ctx, "SELECT name FROM users WHERE id = $1", id).Scan(&name)
if err == sql.ErrNoRows {
    // Not found
}

// Execute (INSERT, UPDATE, DELETE)
result, err := db.ExecContext(ctx, "UPDATE users SET active = $1 WHERE id = $2", true, id)
rowsAffected, _ := result.RowsAffected()
```

`html/template` vs Twig

Twig provides:

```
{% extends 'base.html.twig' %}

{% block content %}
    <h1>{{ user.name }}</h1>
    {% for post in posts %}
        <article>{{ post.title | upper }}</article>
    {% endfor %}
{% endblock %}
```

Go's `html/template` is simpler:

```
const tmpl = `
<!DOCTYPE html>
<html>
<head><title>{{.Title}}</title></head>
<body>
    <h1>{{.User.Name}}</h1>
    {{range .Posts}}
```

```
        <article>{{.Title}}</article>
    {{end}}
</body>
</html>
`

func handler(w http.ResponseWriter, r *http.Request) {
    t := template.Must(template.New("page").Parse(tmpl))
    t.Execute(w, map[string]any{
        "Title": "My Page",
        "User":  user,
        "Posts": posts,
    })
}
```

Key Differences

Auto-escaping: `html/template` automatically escapes HTML. Twig does too.

Inheritance: No built-in template inheritance. Use `template.ParseFiles()` with `define`/`template` blocks:

```
// base.html
{{define "base"}}
<!DOCTYPE html>
<html>
<body>{{template "content" .}}</body>
</html>
{{end}}

// page.html
{{define "content"}}
<h1>{{.Title}}</h1>
{{end}}
```

Filters: Define functions:

```
funcs := template.FuncMap{
    "upper": strings.ToUpper,
}
t := template.New("page").Funcs(funcs).Parse(`{{.Name | upper}}`)
```

When to Reach for Third-Party Packages

The standard library covers 80% of needs. Reach for packages when:

Routing Complexity

Go 1.22 added method-based routing to `http.ServeMux`, but for complex routing, consider: - `chi` — lightweight, idiomatic - `gorilla/mux` — feature-rich (deprecated but stable) - `gin` — fast, full-

featured

Validation

No standard validation library. Consider: - `go-playground/validator` — struct tag validation

```
type User struct {
    Email string `validate:"required,email"`
    Age   int    `validate:"gte=0,lte=130"`
}
```

Caching

No standard caching. Consider: - `patrickmn/go-cache` — in-memory - Redis clients for distributed caching

Configuration

No standard config loading. Consider: - `spf13/viper` — full-featured - `kelseyhightower/envconfig` — environment variables

Database

For more than raw SQL: - `sqlx` — extensions to `database/sql` - `GORM` — full ORM - `sqlc` — generates Go from SQL

Summary

- **Go's standard library** is comprehensive for HTTP, JSON, SQL, and templating
- **`net/http`** provides complete HTTP server/client functionality
- **`encoding/json`** handles serialisation with struct tags
- **`database/sql`** provides connection pooling and query execution
- **`html/template`** offers auto-escaped templating
- **Third-party packages** fill gaps (validation, caching, config)

Exercises

1. **HTTP Server**: Build a simple REST API using only `net/http`. Implement GET, POST, PUT, DELETE for a resource. No external packages.
2. **JSON Customisation**: Create a struct with a `time.Time` field and custom JSON format. Implement `MarshalJSON` and `UnmarshalJSON`.
3. **Database CRUD**: Implement full CRUD operations using `database/sql`. Handle `sql.ErrNoRows` appropriately.
4. **Template Composition**: Build a multi-page site using `html/template` with a shared layout. Implement a custom template function.

5. **Middleware Stack**: Create logging and authentication middleware using only `net/http`. Chain them together.
6. **Symfony Replacement**: Take a simple Symfony controller. Rewrite it using only Go's standard library. List what you miss.
7. **Package Evaluation**: For validation, routing, and caching, evaluate two packages each. Recommend one per category with justification.
8. **Standard Library Limits**: Identify three things you commonly do in Symfony that require third-party Go packages. Evaluate whether the packages are worth it.

Chapter 10: Web Development Without a Framework

Symfony gives you everything: routing, controllers, request handling, response building, middleware, sessions, security. In Go, you build these yourself—but it's easier than you think.

Building HTTP Handlers

Symfony controllers are classes with action methods:

```
class UserController extends AbstractController
{
    #[Route('/users/{id}', methods: ['GET'])]
    public function show(int $id): Response
    {
        $user = $this->userRepository->find($id);
        return $this->json($user);
    }
}
```

Go handlers are functions with a specific signature:

```
func (h *UserHandler) Show(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")  // Go 1.22+
    user, err := h.repo.Find(r.Context(), id)
    if err != nil {
        http.Error(w, "user not found", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(user)
}
```

The Handler Interface

Go's `http.Handler` interface is simple:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Any type implementing this method is a handler. The `http.HandlerFunc` adapter turns functions into handlers:

```go
// Function
func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

// Convert to Handler
var h http.Handler = http.HandlerFunc(hello)
```

Handler Structs

For handlers with dependencies, use structs:

```go
type UserHandler struct {
    repo   UserRepository
    logger *slog.Logger
}

func NewUserHandler(repo UserRepository, logger *slog.Logger) *UserHandler {
    return &UserHandler{repo: repo, logger: logger}
}

func (h *UserHandler) List(w http.ResponseWriter, r *http.Request) {
    users, err := h.repo.FindAll(r.Context())
    if err != nil {
        h.logger.Error("failed to list users", "error", err)
        http.Error(w, "internal error", http.StatusInternalServerError)
        return
    }
    writeJSON(w, http.StatusOK, users)
}

func (h *UserHandler) Create(w http.ResponseWriter, r *http.Request) {
    var input CreateUserInput
    if err := json.NewDecoder(r.Body).Decode(&input); err != nil {
        http.Error(w, "invalid JSON", http.StatusBadRequest)
        return
    }
    user, err := h.repo.Create(r.Context(), input)
    if err != nil {
        http.Error(w, "failed to create user", http.StatusInternalServerError)
        return
    }
    writeJSON(w, http.StatusCreated, user)
}
```

Middleware Patterns (Like Symfony Middlewares)

Symfony uses event listeners and kernel events for cross-cutting concerns:

```php
class AuthenticationListener
{
    public function onKernelRequest(RequestEvent $event): void
    {
        $request = $event->getRequest();
        // Check authentication
    }
}
```

Go uses middleware—functions that wrap handlers:

```go
func loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        slog.Info("request",
            "method", r.Method,
            "path", r.URL.Path,
            "duration", time.Since(start),
        )
    })
}

func authMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        token := r.Header.Get("Authorization")
        if !isValidToken(token) {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
        next.ServeHTTP(w, r)
    })
}
```

Chaining Middleware

Stack middleware by nesting:

```go
handler := authMiddleware(loggingMiddleware(actualHandler))
```

Or create a helper:

```go
func chain(h http.Handler, middlewares ...func(http.Handler) http.Handler) http.Handler {
    for i := len(middlewares) - 1; i >= 0; i-- {
        h = middlewares[i](h)
    }
    return h
}

handler := chain(actualHandler, loggingMiddleware, authMiddleware)
```

Passing Data Through Context

Symfony stores data in request attributes. Go uses context:

```
type contextKey string

const userKey contextKey = "user"

func authMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        user := authenticateUser(r)
        if user == nil {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
        ctx := context.WithValue(r.Context(), userKey, user)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
}

func getUser(ctx context.Context) *User {
    user, _ := ctx.Value(userKey).(*User)
    return user
}

// In handler
func (h *Handler) Profile(w http.ResponseWriter, r *http.Request) {
    user := getUser(r.Context())
    // ...
}
```

Routing: `http.ServeMux` vs Symfony Routing

Symfony Routing is powerful:

```
#[Route('/users/{id}', name: 'user_show', requirements: ['id' => '\d+'])]
public function show(int $id): Response { }

#[Route('/posts/{slug}', name: 'post_show')]
public function post(string $slug): Response { }
```

Go 1.22 improved `http.ServeMux` significantly:

```
mux := http.NewServeMux()

// Method + path patterns
mux.HandleFunc("GET /users/{id}", userHandler.Show)
mux.HandleFunc("POST /users", userHandler.Create)
mux.HandleFunc("PUT /users/{id}", userHandler.Update)
mux.HandleFunc("DELETE /users/{id}", userHandler.Delete)
```

```go
// Wildcards
mux.HandleFunc("GET /files/{path...}", fileHandler.Serve)

// Access path values
func (h *Handler) Show(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")  // Extract {id}
}
```

When You Need More

For complex routing (regex constraints, named routes, reverse routing), use a router package:

```go
// Using chi
r := chi.NewRouter()
r.Get("/users/{id:[0-9]+}", userHandler.Show)
r.Get("/posts/{slug:[a-z-]+}", postHandler.Show)
```

Request Validation Without Annotations

Symfony Validator uses annotations:

```php
class CreateUserInput
{
    #[Assert\NotBlank]
    #[Assert\Email]
    public string $email;

    #[Assert\NotBlank]
    #[Assert\Length(min: 8)]
    public string $password;
}
```

Go doesn't have annotations. Use struct tags with a validation library:

```go
import "github.com/go-playground/validator/v10"

type CreateUserInput struct {
    Email    string `json:"email" validate:"required,email"`
    Password string `json:"password" validate:"required,min=8"`
}

var validate = validator.New()

func (h *Handler) Create(w http.ResponseWriter, r *http.Request) {
    var input CreateUserInput
    if err := json.NewDecoder(r.Body).Decode(&input); err != nil {
        http.Error(w, "invalid JSON", http.StatusBadRequest)
        return
    }
```

```
    if err := validate.Struct(input); err != nil {
        errors := formatValidationErrors(err)
        writeJSON(w, http.StatusUnprocessableEntity, errors)
        return
    }

    // Input is valid
}
```

Manual Validation

For simple cases, validate manually:

```
func (input CreateUserInput) Validate() error {
    if input.Email == "" {
        return errors.New("email is required")
    }
    if !strings.Contains(input.Email, "@") {
        return errors.New("invalid email format")
    }
    if len(input.Password) < 8 {
        return errors.New("password must be at least 8 characters")
    }
    return nil
}
```

Response Patterns

Create helper functions for common responses:

```
func writeJSON(w http.ResponseWriter, status int, data any) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    json.NewEncoder(w).Encode(data)
}

func writeError(w http.ResponseWriter, status int, message string) {
    writeJSON(w, status, map[string]string{"error": message})
}

// Usage
func (h *Handler) Show(w http.ResponseWriter, r *http.Request) {
    user, err := h.repo.Find(r.Context(), id)
    if errors.Is(err, ErrNotFound) {
        writeError(w, http.StatusNotFound, "user not found")
        return
    }
    if err != nil {
        writeError(w, http.StatusInternalServerError, "internal error")
        return
```

```
    }
    writeJSON(w, http.StatusOK, user)
}
```

Session Management Without Symfony Session

Symfony provides session management out of the box:

```php
$session = $request->getSession();
$session->set('user_id', $userId);
$userId = $session->get('user_id');
```

Go needs a session library. `gorilla/sessions` is popular:

```
import "github.com/gorilla/sessions"

var store = sessions.NewCookieStore([]byte("secret-key"))

func (h *Handler) Login(w http.ResponseWriter, r *http.Request) {
    session, _ := store.Get(r, "session-name")
    session.Values["user_id"] = user.ID
    session.Save(r, w)
}

func (h *Handler) Profile(w http.ResponseWriter, r *http.Request) {
    session, _ := store.Get(r, "session-name")
    userID, ok := session.Values["user_id"].(int)
    if !ok {
        http.Error(w, "unauthorized", http.StatusUnauthorized)
        return
    }
    // ...
}
```

Stateless APIs

Many Go APIs are stateless, using JWTs instead of sessions:

```
func authMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        tokenString := extractToken(r)
        claims, err := validateJWT(tokenString)
        if err != nil {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
        ctx := context.WithValue(r.Context(), userKey, claims.UserID)
        next.ServeHTTP(w, r.WithContext(ctx))
    })
```

```
}
```

Putting It Together: Complete Server

```go
func main() {
    // Dependencies
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    db := connectDatabase()
    userRepo := repository.NewUserRepository(db)
    userHandler := handler.NewUserHandler(userRepo, logger)

    // Router
    mux := http.NewServeMux()
    mux.HandleFunc("GET /users", userHandler.List)
    mux.HandleFunc("POST /users", userHandler.Create)
    mux.HandleFunc("GET /users/{id}", userHandler.Show)
    mux.HandleFunc("PUT /users/{id}", userHandler.Update)
    mux.HandleFunc("DELETE /users/{id}", userHandler.Delete)

    // Middleware stack
    handler := chain(mux,
        recoveryMiddleware,
        loggingMiddleware,
        corsMiddleware,
    )

    // Server
    server := &http.Server{
        Addr:         ":8080",
        Handler:      handler,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    logger.Info("server starting", "addr", server.Addr)
    if err := server.ListenAndServe(); err != nil {
        logger.Error("server error", "error", err)
    }
}
```

Summary

- **Handlers** are functions or structs implementing `http.Handler`
- **Middleware** wraps handlers for cross-cutting concerns
- **Routing** uses `http.ServeMux` (Go 1.22+) or router libraries
- **Validation** uses struct tags or manual validation
- **Response helpers** provide consistent JSON responses
- **Sessions** use libraries like `gorilla/sessions` or JWT

Exercises

1. **Full CRUD API**: Build a complete REST API for a resource using only `net/http`. Include all HTTP methods.
2. **Middleware Chain**: Implement logging, recovery (panic handling), and request ID middleware. Chain them correctly.
3. **Authentication Flow**: Implement login/logout with JWT tokens. Store user info in context.
4. **Validation Layer**: Create a validation system for request bodies. Handle validation errors with proper HTTP responses.
5. **Response Writer Wrapper**: Create a `ResponseWriter` wrapper that captures the status code for logging middleware.
6. **Route Groups**: Implement route grouping with shared middleware (e.g., `/api/v1/users` with auth middleware).
7. **Error Handling**: Design an error type that carries HTTP status codes. Use it throughout handlers.
8. **Graceful Shutdown**: Implement graceful shutdown that waits for active requests to complete.

Chapter 11: Database Access

Doctrine is central to Symfony development—entities, repositories, the EntityManager, DQL, migrations. Go's database story is simpler but requires more manual work.

`database/sql` Fundamentals

Go's `database/sql` is a thin abstraction over database drivers:

```
import (
    "database/sql"
    _ "github.com/lib/pq"  // PostgreSQL driver
)

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/dbname?sslmode=disable")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // Verify connection
    if err := db.Ping(); err != nil {
        log.Fatal(err)
    }

    // db is now ready for queries
}
```

The `sql.Open` doesn't connect—it prepares the connection pool. `Ping` verifies connectivity.

Connection Pool Configuration

```
db.SetMaxOpenConns(25)                 // Maximum open connections
db.SetMaxIdleConns(25)                 // Maximum idle connections
db.SetConnMaxLifetime(5 * time.Minute) // Maximum connection lifetime
```

Doctrine manages pooling via DBAL configuration. Go's pooling is built into `database/sql`.

Basic Queries

```
// Query multiple rows
rows, err := db.QueryContext(ctx, "SELECT id, name, email FROM users WHERE active = $1", true)
if err != nil {
    return nil, err
}
defer rows.Close()

var users []User
for rows.Next() {
    var u User
    if err := rows.Scan(&u.ID, &u.Name, &u.Email); err != nil {
        return nil, err
    }
    users = append(users, u)
}
if err := rows.Err(); err != nil {
    return nil, err
}

// Query single row
var name string
err := db.QueryRowContext(ctx, "SELECT name FROM users WHERE id = $1", id).Scan(&name)
if err == sql.ErrNoRows {
    return "", ErrNotFound
}
if err != nil {
    return "", err
}

// Execute (INSERT, UPDATE, DELETE)
result, err := db.ExecContext(ctx, "INSERT INTO users (name, email) VALUES ($1, $2)", name, email)
if err != nil {
    return 0, err
}
id, err := result.LastInsertId()  // Or result.RowsAffected()
```

Always Use Context

```
// Good: Use context for cancellation and timeouts
rows, err := db.QueryContext(ctx, "SELECT ...")

// Avoid: No context means no cancellation
rows, err := db.Query("SELECT ...")  // Don't use this
```

Query Builders: SQLC vs Doctrine QueryBuilder

Doctrine QueryBuilder lets you build queries programmatically:

```php
$qb = $this->createQueryBuilder('u')
    ->where('u.active = :active')
    ->andWhere('u.createdAt > :since')
    ->orderBy('u.createdAt', 'DESC')
    ->setParameter('active', true)
    ->setParameter('since', $since);
```

Go has several approaches:

1. Raw SQL (Most Common)

```go
query := `
    SELECT id, name, email
    FROM users
    WHERE active = $1
      AND created_at > $2
    ORDER BY created_at DESC
`
rows, err := db.QueryContext(ctx, query, true, since)
```

Many Go developers prefer raw SQL—it's explicit, performant, and your DBA can read it.

2. squirrel (Query Builder)

```go
import sq "github.com/Masterminds/squirrel"

query, args, err := sq.Select("id", "name", "email").
    From("users").
    Where(sq.Eq{"active": true}).
    Where(sq.Gt{"created_at": since}).
    OrderBy("created_at DESC").
    PlaceholderFormat(sq.Dollar).
    ToSql()

rows, err := db.QueryContext(ctx, query, args...)
```

3. SQLC (Code Generation)

SQLC generates Go code from SQL:

```sql
-- queries.sql
-- name: GetUser :one
SELECT id, name, email FROM users WHERE id = $1;

-- name: ListActiveUsers :many
SELECT id, name, email FROM users WHERE active = true ORDER BY created_at DESC;

-- name: CreateUser :one
INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id, name, email;
```

Run `sqlc generate` to create type-safe Go code:

```
// Generated code
func (q *Queries) GetUser(ctx context.Context, id int64) (User, error)
func (q *Queries) ListActiveUsers(ctx context.Context) ([]User, error)
func (q *Queries) CreateUser(ctx context.Context, arg CreateUserParams) (User, error)

// Usage
user, err := queries.GetUser(ctx, 42)
```

SQLC provides type safety without runtime overhead—the SQL is still explicit.

ORMs: GORM vs Doctrine ORM (And Why Many Skip Them)

Doctrine ORM is central to Symfony:

```
#[Entity]
class User
{
    #[Id]
    #[GeneratedValue]
    #[Column]
    private int $id;

    #[Column]
    private string $name;

    #[OneToMany(targetEntity: Post::class, mappedBy: 'author')]
    private Collection $posts;
}

// Usage
$user = $em->find(User::class, $id);
$em->persist($newUser);
$em->flush();
```

GORM is Go's most popular ORM:

```
type User struct {
    ID        uint   `gorm:"primaryKey"`
    Name      string
    Posts     []Post `gorm:"foreignKey:AuthorID"`
    CreatedAt time.Time
}

// Usage
var user User
db.First(&user, id)

db.Create(&User{Name: "Alice"})
```

```
db.Model(&user).Update("Name", "Bob")
```

Why Many Go Developers Skip ORMs

Go culture is skeptical of ORMs:

1. **Hidden queries**: ORMs generate SQL you don't see
2. **N+1 problems**: Easy to create accidentally
3. **Learning curve**: Another abstraction to learn
4. **Performance**: Raw SQL is faster
5. **Go's philosophy**: Explicit over magic

The popular alternatives: - **SQLC**: Type-safe, generated from SQL - **sqlx**: Extensions to `database/sql` (named parameters, struct scanning) - **Raw SQL**: Just write queries

sqlx: A Happy Medium

```
import "github.com/jmoiron/sqlx"

type User struct {
    ID    int    `db:"id"`
    Name  string `db:"name"`
    Email string `db:"email"`
}

// Scan into struct
var user User
err := db.GetContext(ctx, &user, "SELECT * FROM users WHERE id = $1", id)

// Scan into slice
var users []User
err := db.SelectContext(ctx, &users, "SELECT * FROM users WHERE active = $1", true)

// Named parameters
query := "INSERT INTO users (name, email) VALUES (:name, :email)"
result, err := db.NamedExecContext(ctx, query, user)
```

Migrations: Goose vs Doctrine Migrations

Doctrine Migrations generates PHP files:

```
public function up(Schema $schema): void
{
    $this->addSql('CREATE TABLE users (...)');
}

public function down(Schema $schema): void
{
    $this->addSql('DROP TABLE users');
```

```
}
```

Go has several migration tools. Goose is popular:

```sql
-- migrations/001_create_users.sql

-- +goose Up
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- +goose Down
DROP TABLE users;
```

```
goose postgres "postgres://user:pass@localhost/db" up
goose postgres "postgres://user:pass@localhost/db" down
```

Other options: - **golang-migrate**: Another popular choice - **atlas**: Schema-based migrations - **sqlc**: Can manage schemas

Connection Pooling (Built-In)

Doctrine DBAL configures pooling via environment:

```yaml
doctrine:
    dbal:
        connections:
            default:
                pooled: true
```

Go's `database/sql` pools automatically:

```
db.SetMaxOpenConns(25)
db.SetMaxIdleConns(10)
db.SetConnMaxLifetime(5 * time.Minute)
db.SetConnMaxIdleTime(1 * time.Minute)
```

The pool: - Opens connections on demand - Reuses idle connections - Closes connections past lifetime - Blocks when pool is exhausted

Transactions Without Doctrine's `flush()`

Doctrine batches changes and flushes:

```php
$em->persist($user);
$em->persist($order);
$em->flush();  // Single transaction with all changes
```

Go transactions are explicit:

```go
tx, err := db.BeginTx(ctx, nil)
if err != nil {
    return err
}
defer tx.Rollback()  // Rollback if not committed

_, err = tx.ExecContext(ctx, "INSERT INTO users (name) VALUES ($1)", name)
if err != nil {
    return err
}

_, err = tx.ExecContext(ctx, "INSERT INTO orders (user_id) VALUES ($1)", userID)
if err != nil {
    return err
}

return tx.Commit()  // Commit only if all succeeded
```

Transaction Helper

```go
func withTx(ctx context.Context, db *sql.DB, fn func(*sql.Tx) error) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback()

    if err := fn(tx); err != nil {
        return err
    }

    return tx.Commit()
}

// Usage
err := withTx(ctx, db, func(tx *sql.Tx) error {
    // All operations use tx
    return nil
})
```

Summary

- **`database/sql`** provides connection pooling and basic queries
- **SQLC** generates type-safe code from SQL

- **sqlx** adds struct scanning to `database/sql`
- **GORM** exists but many prefer explicit SQL
- **Migrations** use tools like Goose or golang-migrate
- **Transactions** are explicit with `BeginTx`, `Commit`, `Rollback`

Exercises

1. **Repository Pattern**: Implement a User repository with `database/sql`. Include Find, FindAll, Create, Update, Delete.
2. **SQLC Setup**: Set up SQLC for a simple schema. Write queries and generate code. Compare to hand-written code.
3. **Transaction Handling**: Implement a function that creates a user and their initial preferences in a single transaction.
4. **Connection Pool Tuning**: Write a load test that stresses the connection pool. Experiment with pool settings.
5. **Migration Workflow**: Set up Goose for a project. Create up/down migrations. Practice rolling back.
6. **Query Builder Comparison**: Implement the same complex query using raw SQL, squirrel, and GORM. Compare readability and safety.
7. **N+1 Detection**: Write code that accidentally creates N+1 queries. Then fix it with a JOIN or batch query.
8. **Nullable Handling**: Handle nullable columns using `sql.NullString`, `sql.NullInt64`, etc. Then try with pointer types. Compare approaches.

Chapter 12: API Development

Symfony's API Platform or FOSRestBundle provide complete API solutions. Go developers typically build APIs from smaller pieces. This chapter covers the patterns.

JSON APIs: Encoding/Decoding Patterns

Symfony Serializer handles complex cases:

```php
$user = $serializer->deserialize($json, User::class, 'json');
$json = $serializer->serialize($user, 'json', ['groups' => ['public']]);
```

Go uses `encoding/json`:

```go
// Decode
var user User
if err := json.NewDecoder(r.Body).Decode(&user); err != nil {
    http.Error(w, "invalid JSON", http.StatusBadRequest)
    return
}

// Encode
w.Header().Set("Content-Type", "application/json")
json.NewEncoder(w).Encode(user)
```

Struct Tags Control Serialisation

```go
type User struct {
    ID        int       `json:"id"`
    Name      string    `json:"name"`
    Email     string    `json:"email"`
    Password  string    `json:"-"`              // Never serialised
    CreatedAt time.Time `json:"created_at"`
    DeletedAt *time.Time `json:"deleted_at,omitempty"` // Omit if nil
}
```

Different Input/Output Structs

Unlike PHP where you might use serialisation groups, Go often uses separate structs:

```go
// Input (what clients send)
type CreateUserInput struct {
    Name     string `json:"name" validate:"required"`
    Email    string `json:"email" validate:"required,email"`
```

```
    Password string `json:"password" validate:"required,min=8"`
}

// Output (what API returns)
type UserResponse struct {
    ID        int       `json:"id"`
    Name      string    `json:"name"`
    Email     string    `json:"email"`
    CreatedAt time.Time `json:"created_at"`
}

// Domain model (internal)
type User struct {
    ID           int
    Name         string
    Email        string
    PasswordHash string
    CreatedAt    time.Time
}

// Conversion
func (u User) ToResponse() UserResponse {
    return UserResponse{
        ID:        u.ID,
        Name:      u.Name,
        Email:     u.Email,
        CreatedAt: u.CreatedAt,
    }
}
```

Custom Marshalling

For complex serialisation, implement `json.Marshaler`:

```
type Money struct {
    Amount   int64  // Stored in cents
    Currency string
}

func (m Money) MarshalJSON() ([]byte, error) {
    return json.Marshal(map[string]interface{}{
        "amount":   float64(m.Amount) / 100,
        "currency": m.Currency,
    })
}

func (m *Money) UnmarshalJSON(data []byte) error {
    var raw struct {
        Amount   float64 `json:"amount"`
        Currency string  `json:"currency"`
    }
    if err := json.Unmarshal(data, &raw); err != nil {
```

```
        return err
    }
    m.Amount = int64(raw.Amount * 100)
    m.Currency = raw.Currency
    return nil
}
```

OpenAPI/Swagger Integration

Symfony has NelmioApiDocBundle for OpenAPI generation. Go has several options:

swag (Generate from Comments)

```
// @Summary Create a new user
// @Description Create a user with the input payload
// @Tags users
// @Accept json
// @Produce json
// @Param user body CreateUserInput true "User input"
// @Success 201 {object} UserResponse
// @Failure 400 {object} ErrorResponse
// @Router /users [post]
func (h *Handler) CreateUser(w http.ResponseWriter, r *http.Request) {
    // Implementation
}
```

Run `swag init` to generate OpenAPI spec.

oapi-codegen (Generate from Spec)

Write OpenAPI spec first, generate Go code:

```
# openapi.yaml
paths:
  /users:
    post:
      operationId: createUser
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateUserInput'
      responses:
        '201':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/UserResponse'
```

```
oapi-codegen -generate types,server openapi.yaml > api.gen.go
```

This generates types and server interfaces you implement.

Authentication Middleware (vs Symfony Security)

Symfony Security provides: - Firewalls - Voters - Guards - User providers

Go uses middleware:

```
func authMiddleware(tokenService TokenService) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            token := extractBearerToken(r)
            if token == "" {
                writeError(w, http.StatusUnauthorized, "missing token")
                return
            }

            claims, err := tokenService.Validate(token)
            if err != nil {
                writeError(w, http.StatusUnauthorized, "invalid token")
                return
            }

            ctx := context.WithValue(r.Context(), userClaimsKey, claims)
            next.ServeHTTP(w, r.WithContext(ctx))
        })
    }
}

func extractBearerToken(r *http.Request) string {
    auth := r.Header.Get("Authorization")
    if strings.HasPrefix(auth, "Bearer ") {
        return strings.TrimPrefix(auth, "Bearer ")
    }
    return ""
}
```

Role-Based Access

```
func requireRole(role string) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            claims := getUserClaims(r.Context())
            if claims == nil {
                writeError(w, http.StatusUnauthorized, "not authenticated")
                return
            }
```

```
            if !claims.HasRole(role) {
                writeError(w, http.StatusForbidden, "insufficient permissions")
                return
            }

            next.ServeHTTP(w, r)
        })
    }
}

// Usage
mux.Handle("DELETE /users/{id}", chain(handler,
    authMiddleware(tokenSvc),
    requireRole("admin"),
))
```

Validation Patterns (vs Symfony Validator)

Symfony Validator uses annotations:

```
class CreateUserInput
{
    #[Assert\NotBlank(message: "Name is required")]
    #[Assert\Length(min: 2, max: 100)]
    public string $name;

    #[Assert\Email]
    public string $email;
}
```

Go uses `go-playground/validator`:

```
import "github.com/go-playground/validator/v10"

type CreateUserInput struct {
    Name  string `json:"name" validate:"required,min=2,max=100"`
    Email string `json:"email" validate:"required,email"`
}

var validate = validator.New()

func validateInput(input any) map[string]string {
    err := validate.Struct(input)
    if err == nil {
        return nil
    }

    errors := make(map[string]string)
    for _, err := range err.(validator.ValidationErrors) {
        field := strings.ToLower(err.Field())
```

```
        errors[field] = formatValidationMessage(err)
    }
    return errors
}

func formatValidationMessage(err validator.FieldError) string {
    switch err.Tag() {
    case "required":
        return "This field is required"
    case "email":
        return "Must be a valid email address"
    case "min":
        return fmt.Sprintf("Must be at least %s characters", err.Param())
    default:
        return "Invalid value"
    }
}
```

Custom Validation

```
func init() {
    validate.RegisterValidation("username", func(fl validator.FieldLevel) bool {
        username := fl.Field().String()
        return regexp.MustCompile(`^[a-z0-9_]+$`).MatchString(username)
    })
}

type User struct {
    Username string `validate:"required,username,min=3,max=20"`
}
```

Error Response Standards

Symfony normalises errors via the Serializer. Build a consistent error format:

```
type ErrorResponse struct {
    Error   string            `json:"error"`
    Code    string            `json:"code,omitempty"`
    Details map[string]string `json:"details,omitempty"`
}

func writeError(w http.ResponseWriter, status int, message string) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    json.NewEncoder(w).Encode(ErrorResponse{Error: message})
}

func writeValidationError(w http.ResponseWriter, errors map[string]string) {
    w.Header().Set("Content-Type", "application/json")
```

```
    w.WriteHeader(http.StatusUnprocessableEntity)
    json.NewEncoder(w).Encode(ErrorResponse{
        Error:   "Validation failed",
        Code:    "VALIDATION_ERROR",
        Details: errors,
    })
}
```

Error Types for HTTP

```
type HTTPError struct {
    Status  int
    Message string
    Code    string
}

func (e HTTPError) Error() string {
    return e.Message
}

var (
    ErrNotFound     = HTTPError{Status: 404, Message: "Resource not found", Code: "NOT_FOUND"}
    ErrUnauthorized = HTTPError{Status: 401, Message: "Unauthorized", Code: "UNAUTHORIZED"}
)

// In handler
func (h *Handler) Show(w http.ResponseWriter, r *http.Request) {
    user, err := h.repo.Find(ctx, id)
    if err != nil {
        handleError(w, err)
        return
    }
    writeJSON(w, http.StatusOK, user)
}

func handleError(w http.ResponseWriter, err error) {
    var httpErr HTTPError
    if errors.As(err, &httpErr) {
        writeError(w, httpErr.Status, httpErr.Message)
        return
    }
    // Log unexpected errors
    slog.Error("unexpected error", "error", err)
    writeError(w, http.StatusInternalServerError, "Internal server error")
}
```

Versioning Strategies

Symfony supports URL, header, and query parameter versioning. Go doesn't have built-in support—implement your preferred strategy:

URL Versioning

```
mux := http.NewServeMux()
mux.Handle("/api/v1/", http.StripPrefix("/api/v1", v1Router))
mux.Handle("/api/v2/", http.StripPrefix("/api/v2", v2Router))
```

Header Versioning

```
func versionMiddleware(v1, v2 http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        version := r.Header.Get("API-Version")
        switch version {
        case "2", "2.0":
            v2.ServeHTTP(w, r)
        default:
            v1.ServeHTTP(w, r)
        }
    })
}
```

Summary

- **JSON encoding** uses struct tags for field mapping
- **Separate structs** for input, output, and domain models
- **OpenAPI** via swag (generate from code) or oapi-codegen (generate from spec)
- **Authentication** is middleware that populates context
- **Validation** uses `go-playground/validator` with struct tags
- **Error responses** follow consistent structure
- **Versioning** is implemented manually (URL or header)

Exercises

1. **Complete API Resource**: Build a full REST API for a resource with create, read, update, delete, and list operations.
2. **Custom Validation**: Add three custom validation rules (e.g., strong password, valid slug, future date).
3. **OpenAPI Generation**: Set up swag for a small API. Generate documentation and verify it matches your handlers.
4. **Error Handling System**: Create an error handling system with different error types (validation, not found, unauthorized, internal).
5. **Pagination**: Implement cursor-based pagination for a list endpoint. Include pagination metadata in response.

6. **Rate Limiting**: Add rate limiting middleware using a token bucket or sliding window algorithm.
7. **Request ID Tracing**: Add request ID middleware. Include the ID in logs and error responses.
8. **API Versioning**: Implement URL-based versioning with two API versions that differ in response format.

Chapter 13: Testing — A Different Philosophy

PHPUnit is the standard for PHP testing—assertions, mocks, data providers, coverage. Go's testing approach is deliberately simpler, built into the language and standard library.

Table-Driven Tests

PHPUnit uses data providers:

```php
#[DataProvider('additionProvider')]
public function testAdd(int $a, int $b, int $expected): void
{
    $this->assertEquals($expected, $this->calculator->add($a, $b));
}

public static function additionProvider(): array
{
    return [
        [1, 2, 3],
        [0, 0, 0],
        [-1, 1, 0],
    ];
}
```

Go uses table-driven tests:

```go
func TestAdd(t *testing.T) {
    tests := []struct {
        name     string
        a, b     int
        expected int
    }{
        {"positive numbers", 1, 2, 3},
        {"zeros", 0, 0, 0},
        {"negative and positive", -1, 1, 0},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            result := Add(tt.a, tt.b)
            if result != tt.expected {
                t.Errorf("Add(%d, %d) = %d; want %d", tt.a, tt.b, result, tt.expected)
            }
        })
```

```
    }
}
```

Why Table-Driven?

1. **Easy to add cases**: Just add a row to the table
2. **Clear structure**: Input → expected output
3. **Named subtests**: Each case runs as `TestAdd/positive_numbers`
4. **Parallel execution**: Add `t.Parallel()` for concurrent tests

```go
func TestProcess(t *testing.T) {
    tests := []struct {
        name    string
        input   string
        want    string
        wantErr bool
    }{
        {"valid input", "hello", "HELLO", false},
        {"empty input", "", "", true},
        {"special chars", "hello!", "HELLO!", false},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()  // Run subtests in parallel

            got, err := Process(tt.input)

            if (err != nil) != tt.wantErr {
                t.Errorf("Process() error = %v, wantErr %v", err, tt.wantErr)
                return
            }
            if got != tt.want {
                t.Errorf("Process() = %v, want %v", got, tt.want)
            }
        })
    }
}
```

No Assertions Library (By Design)

PHPUnit has rich assertions:

```php
$this->assertEquals($expected, $actual);
$this->assertContains($item, $array);
$this->assertInstanceOf(User::class, $result);
$this->assertGreaterThan(0, $count);
```

Go's `testing` package has only `t.Error`, `t.Errorf`, `t.Fatal`, `t.Fatalf`:

```go
if got != want {
    t.Errorf("got %v, want %v", got, want)
}

if user == nil {
    t.Fatal("user is nil")  // Stops the test
}
```

Why No Assertions?

Go's philosophy: assertions are just `if` statements with better errors. The testing package doesn't need to provide them—you write clear comparison code.

Third-Party Options

If you want assertions, use `testify`:

```go
import "github.com/stretchr/testify/assert"

func TestUser(t *testing.T) {
    user := NewUser("Alice")

    assert.Equal(t, "Alice", user.Name)
    assert.NotNil(t, user.ID)
    assert.True(t, user.IsActive())
}
```

But many Go developers prefer vanilla testing for consistency.

Mocking with Interfaces (vs Prophecy/Mockery)

PHP uses mocking libraries:

```php
$repository = $this->createMock(UserRepository::class);
$repository
    ->expects($this->once())
    ->method('find')
    ->with(42)
    ->willReturn($user);

$service = new UserService($repository);
```

Go mocks via interfaces:

```go
// Interface to mock
type UserRepository interface {
    Find(ctx context.Context, id int) (*User, error)
}
```

```go
// Test mock implementation
type mockUserRepo struct {
    user *User
    err  error
}

func (m *mockUserRepo) Find(ctx context.Context, id int) (*User, error) {
    return m.user, m.err
}

// Test
func TestGetUser(t *testing.T) {
    expectedUser := &User{ID: 42, Name: "Alice"}
    repo := &mockUserRepo{user: expectedUser}
    service := NewUserService(repo)

    user, err := service.GetUser(context.Background(), 42)

    if err != nil {
        t.Fatalf("unexpected error: %v", err)
    }
    if user.Name != "Alice" {
        t.Errorf("got name %s, want Alice", user.Name)
    }
}
```

Why Manual Mocks?

1. **Type-safe**: The compiler ensures mock implements interface
2. **Explicit**: You see exactly what the mock does
3. **Flexible**: Add any behaviour you need
4. **No runtime reflection**: Pure Go code

Mock Generation Tools

For large interfaces, generate mocks:

```go
//go:generate mockgen -source=repository.go -destination=mock_repository.go

type UserRepository interface {
    Find(ctx context.Context, id int) (*User, error)
    Create(ctx context.Context, user *User) error
    Update(ctx context.Context, user *User) error
    Delete(ctx context.Context, id int) error
}
```

`mockgen` creates a mock with expectation setting and verification.

Integration Tests

PHPUnit integration tests often use Symfony's WebTestCase:

```php
class UserControllerTest extends WebTestCase
{
    public function testCreateUser(): void
    {
        $client = static::createClient();
        $client->request('POST', '/api/users', [], [],
            ['CONTENT_TYPE' => 'application/json'],
            json_encode(['name' => 'Alice'])
        );

        $this->assertResponseStatusCodeSame(201);
    }
}
```

Go uses `httptest`:

```go
func TestCreateUser(t *testing.T) {
    // Setup
    repo := NewInMemoryUserRepo()
    handler := NewUserHandler(repo)

    // Create request
    body := strings.NewReader(`{"name":"Alice"}`)
    req := httptest.NewRequest("POST", "/users", body)
    req.Header.Set("Content-Type", "application/json")
    rec := httptest.NewRecorder()

    // Execute
    handler.Create(rec, req)

    // Assert
    if rec.Code != http.StatusCreated {
        t.Errorf("status = %d; want %d", rec.Code, http.StatusCreated)
    }

    var response User
    json.NewDecoder(rec.Body).Decode(&response)
    if response.Name != "Alice" {
        t.Errorf("name = %s; want Alice", response.Name)
    }
}
```

Testing the Full Stack

```go
func TestAPI(t *testing.T) {
    // Setup real server
    db := setupTestDB(t)
    server := NewServer(db)

    ts := httptest.NewServer(server)
    defer ts.Close()
```

```
    // Make HTTP request
    resp, err := http.Post(ts.URL+"/users", "application/json",
        strings.NewReader(`{"name":"Alice"}`))
    if err != nil {
        t.Fatal(err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusCreated {
        t.Errorf("status = %d; want %d", resp.StatusCode, http.StatusCreated)
    }
}
```

Benchmarking Built-In

PHPUnit benchmarking requires additional tooling. Go has built-in benchmarks:

```
func BenchmarkProcess(b *testing.B) {
    input := "test data"

    for i := 0; i < b.N; i++ {
        Process(input)
    }
}
```

Run with:

```
go test -bench=. -benchmem

# Output:
# BenchmarkProcess-8   1000000   1234 ns/op   256 B/op   2 allocs/op
```

Benchmark Best Practices

```
func BenchmarkProcess(b *testing.B) {
    input := generateLargeInput()  // Setup outside loop

    b.ResetTimer()  // Don't count setup time

    for i := 0; i < b.N; i++ {
        Process(input)
    }
}

// Compare implementations
func BenchmarkProcessV1(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ProcessV1(input)
    }
```

```
}

func BenchmarkProcessV2(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ProcessV2(input)
    }
}
```

Coverage Tooling

PHPUnit coverage with Xdebug:

```
XDEBUG_MODE=coverage phpunit --coverage-html coverage/
```

Go coverage:

```
go test -coverprofile=coverage.out ./...
go tool cover -html=coverage.out  # Open in browser
go tool cover -func=coverage.out  # Print coverage by function
```

Coverage in CI

```
go test -coverprofile=coverage.out ./...
go tool cover -func=coverage.out | grep total:
# total:  (statements)  85.2%
```

Test Containers for Integration Tests

PHPUnit might use Docker through manual setup. Go has `testcontainers-go`:

```
import "github.com/testcontainers/testcontainers-go"

func TestWithPostgres(t *testing.T) {
    ctx := context.Background()

    container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
        ContainerRequest: testcontainers.ContainerRequest{
            Image:        "postgres:15",
            ExposedPorts: []string{"5432/tcp"},
            Env: map[string]string{
                "POSTGRES_PASSWORD": "test",
                "POSTGRES_DB":       "test",
            },
            WaitingFor: wait.ForListeningPort("5432/tcp"),
        },
        Started: true,
    })
```

```
    if err != nil {
        t.Fatal(err)
    }
    defer container.Terminate(ctx)

    host, _ := container.Host(ctx)
    port, _ := container.MappedPort(ctx, "5432")

    // Connect to container's postgres
    dsn := fmt.Sprintf("postgres://postgres:test@%s:%s/test", host, port.Port())
    db, err := sql.Open("postgres", dsn)
    // ... run tests
}
```

Summary

- **Table-driven tests** are idiomatic for parameterised testing
- **No assertions library** by design; use `if` statements
- **Interface mocking** is manual but type-safe
- **httptest** provides test servers and recorders
- **Benchmarking** is built into `go test`
- **Coverage** via `go test -cover`
- **Test containers** for integration tests with real dependencies

---

Exercises

1. **Table-Driven Conversion**: Convert a PHPUnit test with data provider to Go table-driven style.
2. **Mock Implementation**: Define an interface with 3 methods. Write a manual mock. Write a test using it.
3. **HTTP Handler Test**: Write tests for a handler covering success, validation error, and not found cases.
4. **Benchmark Comparison**: Write two implementations of the same function. Benchmark both. Identify the faster one.
5. **Coverage Analysis**: Run coverage on a package. Identify untested code paths. Add tests to increase coverage.
6. **Test Containers**: Set up a test with testcontainers for a real database. Run migrations. Execute queries.
7. **Parallel Tests**: Convert sequential tests to parallel using `t.Parallel()`. Verify they don't interfere.
8. **Test Helper Functions**: Create reusable test helpers for common setup (creating users, authenticated requests, etc.).

Chapter 14: Configuration and Environment

Symfony's configuration system is comprehensive: YAML files, environment variables, parameters, service bindings. Go's approach is simpler but requires more explicit code.

No `.env` Magic: Explicit Configuration

Symfony Dotenv loads `.env` files automatically:

```
# .env
DATABASE_URL=mysql://user:pass@localhost/db
MAILER_DSN=smtp://localhost
APP_SECRET=abc123
```

```
// Automatically available
$_ENV['DATABASE_URL'];
$this->getParameter('database_url');
```

Go reads environment variables directly:

```
import "os"

func main() {
    dbURL := os.Getenv("DATABASE_URL")
    if dbURL == "" {
        log.Fatal("DATABASE_URL is required")
    }

    // Or with default
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }
}
```

Loading `.env` Files

Use `godotenv` if you want `.env` file loading:

```
import "github.com/joho/godotenv"

func main() {
```

```
    // Load .env file (optional in production)
    godotenv.Load()

    dbURL := os.Getenv("DATABASE_URL")
}
```

But many Go developers skip `.env` files entirely, preferring: - Environment variables set by the deployment platform - Configuration files (YAML, JSON, TOML) - Command-line flags

Viper vs symfony/dotenv

Viper is Go's most comprehensive configuration library:

```
import "github.com/spf13/viper"

func loadConfig() (*Config, error) {
    viper.SetConfigName("config")
    viper.SetConfigType("yaml")
    viper.AddConfigPath(".")
    viper.AddConfigPath("/etc/myapp/")

    // Environment variables override file values
    viper.AutomaticEnv()
    viper.SetEnvPrefix("MYAPP")

    // Defaults
    viper.SetDefault("server.port", 8080)
    viper.SetDefault("server.timeout", "30s")

    if err := viper.ReadInConfig(); err != nil {
        if _, ok := err.(viper.ConfigFileNotFoundError); !ok {
            return nil, err
        }
        // Config file not found; use defaults and env vars
    }

    var config Config
    if err := viper.Unmarshal(&config); err != nil {
        return nil, err
    }

    return &config, nil
}
```

Configuration Struct

```
type Config struct {
    Server   ServerConfig   `mapstructure:"server"`
    Database DatabaseConfig `mapstructure:"database"`
    Redis    RedisConfig    `mapstructure:"redis"`
}

type ServerConfig struct {
    Port    int           `mapstructure:"port"`
    Timeout time.Duration `mapstructure:"timeout"`
}

type DatabaseConfig struct {
    URL             string `mapstructure:"url"`
    MaxConnections  int    `mapstructure:"max_connections"`
}
```

Config File

```
# config.yaml
server:
  port: 8080
  timeout: 30s

database:
  url: postgres://localhost/myapp
  max_connections: 25
```

Environment Variable Override

```
MYAPP_SERVER_PORT=9000 ./myapp
# Uses 9000 instead of 8080
```

Feature Flags Patterns

Symfony might use a feature flag bundle. Go uses simple configuration:

```
type FeatureFlags struct {
    NewCheckout    bool `mapstructure:"new_checkout"`
    BetaDashboard  bool `mapstructure:"beta_dashboard"`
    ExperimentalAPI bool `mapstructure:"experimental_api"`
}

type Config struct {
    Features FeatureFlags `mapstructure:"features"`
}
```

```
// Usage
if config.Features.NewCheckout {
    return newCheckoutHandler(w, r)
}
return legacyCheckoutHandler(w, r)
```

More Sophisticated Feature Flags

For percentage rollouts or user targeting:

```
type FeatureFlag struct {
    Enabled    bool     `mapstructure:"enabled"`
    Percentage int      `mapstructure:"percentage"`
    Users      []string `mapstructure:"users"`
}

func (f FeatureFlag) IsEnabledFor(userID string) bool {
    if !f.Enabled {
        return false
    }

    // Specific users
    for _, u := range f.Users {
        if u == userID {
            return true
        }
    }

    // Percentage rollout
    if f.Percentage > 0 {
        hash := hashUserID(userID)
        return hash%100 < f.Percentage
    }

    return f.Enabled && len(f.Users) == 0
}
```

12-Factor App Principles in Go

The 12-factor methodology is natural in Go:

III. Config: Store config in environment

```
type Config struct {
    DatabaseURL string
    RedisURL    string
    Port        int
}
```

```go
func LoadFromEnv() Config {
    return Config{
        DatabaseURL: mustEnv("DATABASE_URL"),
        RedisURL:    getEnv("REDIS_URL", "redis://localhost:6379"),
        Port:        getEnvInt("PORT", 8080),
    }
}

func mustEnv(key string) string {
    val := os.Getenv(key)
    if val == "" {
        log.Fatalf("%s is required", key)
    }
    return val
}

func getEnv(key, defaultVal string) string {
    if val := os.Getenv(key); val != "" {
        return val
    }
    return defaultVal
}
```

VI. Processes: Execute as stateless processes

Go applications are naturally stateless—no session state in memory:

```go
// Bad: State in memory
var sessionStore = make(map[string]Session)

// Good: External state store
type SessionStore interface {
    Get(id string) (*Session, error)
    Set(id string, session *Session) error
}

func NewRedisSessionStore(client *redis.Client) SessionStore {
    // ...
}
```

XI. Logs: Treat logs as event streams

```go
import "log/slog"

func main() {
    // Log to stdout as JSON
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    slog.SetDefault(logger)

    slog.Info("server starting", "port", port)
```

```
}
```

No log files—let the platform capture stdout.

Secret Management

Symfony might use secrets with `symfony/secrets`:

```
php bin/console secrets:set DATABASE_PASSWORD
```

Go approaches vary:

Environment Variables

Simple but limited:

```
DATABASE_PASSWORD=secret ./myapp
```

Secret Files

Mount secrets as files:

```go
func loadSecret(path string) (string, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return "", err
    }
    return strings.TrimSpace(string(data)), nil
}

// Usage
dbPassword, err := loadSecret("/run/secrets/db_password")
```

Secret Managers

For AWS Secrets Manager, HashiCorp Vault, etc.:

```go
import "github.com/aws/aws-sdk-go-v2/service/secretsmanager"

func loadFromSecretsManager(ctx context.Context, name string) (string, error) {
    client := secretsmanager.NewFromConfig(cfg)
    result, err := client.GetSecretValue(ctx, &secretsmanager.GetSecretValueInput{
        SecretId: &name,
    })
    if err != nil {
        return "", err
    }
    return *result.SecretString, nil
}
```

No Symfony parameters.yaml

Symfony's parameters:

```yaml
# config/services.yaml
parameters:
    mailer.sender: 'noreply@example.com'

services:
    App\Mailer:
        arguments:
            $sender: '%mailer.sender%'
```

Go uses explicit wiring:

```go
type Config struct {
    Mailer MailerConfig `mapstructure:"mailer"`
}

type MailerConfig struct {
    Sender string `mapstructure:"sender"`
}

// Wiring
func main() {
    config := loadConfig()
    mailer := NewMailer(config.Mailer.Sender)
}
```

Configuration Validation

Validate at startup:

```go
func (c *Config) Validate() error {
    if c.Database.URL == "" {
        return errors.New("database.url is required")
    }
    if c.Server.Port < 1 || c.Server.Port > 65535 {
        return errors.New("server.port must be between 1 and 65535")
    }
    if c.Server.Timeout <= 0 {
        return errors.New("server.timeout must be positive")
    }
    return nil
}

func main() {
    config, err := loadConfig()
    if err != nil {
        log.Fatal(err)
    }
```

```
    if err := config.Validate(); err != nil {
        log.Fatal("invalid configuration: ", err)
    }
}
```

Summary

- **Environment variables** are read directly with `os.Getenv`
- **Viper** provides file-based config with environment override
- **Feature flags** are configuration values, not framework features
- **12-factor principles** align naturally with Go
- **Secret management** via environment, files, or secret managers
- **Explicit wiring** replaces Symfony's parameter injection

Exercises

1. **Config Struct Design**: Design a configuration struct for a web application with database, cache, and HTTP server settings.
2. **Environment Loading**: Write a config loader that reads from environment variables with required vs optional handling.
3. **Viper Setup**: Set up Viper with a YAML config file, environment variable override, and defaults.
4. **Feature Flag System**: Implement a feature flag system with percentage rollout and user targeting.
5. **Secret Rotation**: Design a system that can reload secrets without restarting the application.
6. **Configuration Validation**: Add comprehensive validation to a config struct. Test with invalid configurations.
7. **Multi-Environment Config**: Support different configurations for development, staging, and production.
8. **Command-Line Flags**: Add command-line flag support using `flag` package or `cobra`. Override config file values.

Chapter 15: Introducing Concurrency

This is where your PHP mental model needs the biggest rewrite. PHP is fundamentally single-threaded. Each request gets one thread of execution, start to finish. Go can run thousands of concurrent operations in a single process.

What PHP Doesn't Have (And Why)

PHP's execution model:

```
Request arrives → PHP-FPM worker handles it → Response sent → Worker freed
```

Each worker processes one request at a time. Concurrency in PHP happens at the process level—multiple workers handling multiple requests simultaneously.

Within a single request, everything is sequential:

```php
// Sequential execution - no concurrency
$user = $this->userService->find($id);      // Wait
$orders = $this->orderService->findByUser($user);  // Wait
$recommendations = $this->recService->forUser($user);  // Wait
// Total time: sum of all three operations
```

Why PHP Avoided Concurrency

1. **Simplicity**: Sequential code is easier to reason about
2. **Shared-nothing**: Each request is isolated
3. **Stateless design**: No shared state means no race conditions
4. **Historical context**: PHP was designed for simple web scripts

This model works—most of the web runs on it. But you can't: - Fetch multiple API endpoints simultaneously within a request - Process background jobs without separate workers - Handle WebSocket connections efficiently - Build high-performance real-time systems

PHP's Concurrency Workarounds

PHP has evolved: - **ReactPHP/Swoole**: Event-loop extensions - **Symfony Messenger**: Async message processing - **Parallel extension**: True threading - **Fibers (PHP 8.1)**: Cooperative multitasking

But these are add-ons, not the core model. Go was designed for concurrency from the start.

Goroutines vs Threads vs Processes

PHP concurrency uses processes:

```
php-fpm master
   worker 1 (handles request A)
   worker 2 (handles request B)
   worker 3 (handles request C)
   ... (50-200 workers typical)
```

Each worker is an OS process with its own memory. Safe but heavy—each process might use 20-50MB.

Traditional threading (Java, C++):

```
Process
   Thread 1 (shared memory)
   Thread 2 (shared memory)
   Thread 3 (shared memory)
```

Threads share memory. Lighter than processes (~1MB stack each) but require careful synchronisation.

Go uses goroutines:

```
Go process
   goroutine 1 (~2KB initially)
   goroutine 2 (~2KB initially)
   goroutine 1000 (~2KB initially)
   ... (millions possible)
```

Goroutines are extremely lightweight—thousands or millions in a single process.

Creating Goroutines

```go
// Sequential
result1 := fetchUserData(id)
result2 := fetchOrderHistory(id)

// Concurrent
go fetchUserData(id)     // Starts immediately, doesn't block
go fetchOrderHistory(id)  // Starts immediately, doesn't block
// Both run concurrently
```

The `go` keyword starts a goroutine. The function runs concurrently with the caller.

Waiting for Goroutines

```go
var wg sync.WaitGroup

wg.Add(2)  // We'll wait for 2 goroutines

go func() {
    defer wg.Done()
    fetchUserData(id)
```

```
}()

go func() {
    defer wg.Done()
    fetchOrderHistory(id)
}()

wg.Wait()  // Block until both complete
```

The Go Scheduler Overview

Go has its own scheduler that maps goroutines to OS threads.

The G-M-P Model

- **G (Goroutine)**: The unit of work
- **M (Machine)**: OS thread
- **P (Processor)**: Scheduler context (number = GOMAXPROCS)

```
P1          P2          P3          P4

  G1          G4          G7          G10
  G2          G5          G8          G11
  G3          G6          G9

Running on:
M1          M2          M3          M4
```

The scheduler: - Multiplexes millions of goroutines onto few OS threads - Handles blocking operations by parking goroutines - Distributes work across available CPU cores - Enables preemption to prevent starvation

Why This Matters

You don't manage threads. You just write:

```
for i := 0; i < 100000; i++ {
    go processItem(items[i])
}
```

The scheduler handles: - Which goroutines run when - How many OS threads to use - Load balancing across cores - Context switching

Why PHP-FPM's Model Is Fundamentally Different

PHP-FPM scales by adding workers:

```
Load increases → Add more workers → Each handles one request
```

Go scales by adding goroutines:

```
Load increases → Add more goroutines → All share the same process
```

Memory Efficiency

PHP (200 workers at 30MB each):

```
200 workers × 30MB = 6GB memory for 200 concurrent requests
```

Go (200,000 goroutines at 8KB each):

```
200,000 goroutines × 8KB = 1.6GB memory for 200,000 concurrent operations
```

Connection Handling

PHP-FPM: One worker = one connection.

Go: One goroutine per connection, thousands of connections per process:

```go
func main() {
    listener, _ := net.Listen("tcp", ":8080")

    for {
        conn, _ := listener.Accept()
        go handleConnection(conn)  // New goroutine per connection
    }
}

func handleConnection(conn net.Conn) {
    defer conn.Close()
    // Handle this connection
    // Thousands can run concurrently
}
```

This is why Go excels at WebSockets, real-time APIs, and connection-heavy workloads.

Mental Model: Thousands of Lightweight Threads

Think of goroutines as: - **Cheap**: Start millions without concern - **Independent**: Each has its own stack - **Cooperative**: They yield at certain points - **Scheduled**: Go manages their execution

PHP Mental Model

```
Request → Execute code sequentially → Response
(One path through the code)
```

Go Mental Model

```
Request → Spawn goroutines → Coordinate results → Response
(Multiple paths executing simultaneously)
```

Example: Parallel API Calls

PHP (sequential):

```php
$user = $this->httpClient->get('/api/user/1');         // 100ms
$orders = $this->httpClient->get('/api/orders/1');     // 150ms
$reviews = $this->httpClient->get('/api/reviews/1');   // 80ms
// Total: 330ms
```

Go (concurrent):

```go
var (
    user    User
    orders  []Order
    reviews []Review
    wg      sync.WaitGroup
)

wg.Add(3)

go func() {
    defer wg.Done()
    user, _ = fetchUser(ctx, 1)  // 100ms
}()

go func() {
    defer wg.Done()
    orders, _ = fetchOrders(ctx, 1)  // 150ms
}()

go func() {
    defer wg.Done()
    reviews, _ = fetchReviews(ctx, 1)  // 80ms
}()

wg.Wait()
// Total: ~150ms (slowest operation)
```

The concurrent version is limited by the slowest operation, not the sum.

Summary

- **PHP is single-threaded** per request; concurrency is at the process level
- **Goroutines are extremely lightweight**—start thousands without concern
- **The Go scheduler** manages goroutine-to-thread mapping automatically
- **PHP-FPM's model** scales with processes; Go scales with goroutines
- **Concurrent code** can dramatically reduce response times

Exercises

1. **Goroutine Creation**: Write a program that starts 1,000 goroutines, each printing its index. Observe they don't print in order.
2. **WaitGroup Usage**: Fetch data from 5 URLs concurrently using goroutines and `sync.WaitGroup`. Time it versus sequential fetches.
3. **Memory Comparison**: Write a program that creates 100,000 goroutines and check memory usage. Compare to theoretical PHP worker memory.
4. **Scheduler Observation**: Use `runtime.GOMAXPROCS()` to limit to 1 CPU. Run concurrent work and observe scheduling behaviour.
5. **Request Aggregation**: Implement an HTTP handler that aggregates data from 3 internal APIs concurrently before responding.
6. **Goroutine Leak Detection**: Create a program with an intentional goroutine leak. Use `runtime.NumGoroutine()` to detect it.
7. **PHP Comparison**: Take a PHP script that makes sequential API calls. Estimate the speedup if rewritten with Go concurrency.
8. **Mental Model Documentation**: Write a document explaining Go concurrency to a PHP developer. Include diagrams of goroutines vs PHP workers.

Chapter 16: Channels — Message Passing

Goroutines run concurrently, but how do they communicate? Go's answer is channels—typed conduits for passing data between goroutines.

Channels: Typed Message Passing

The philosophy: "Don't communicate by sharing memory; share memory by communicating."

PHP (if it had concurrency) might share state:

```php
// Hypothetical PHP threading (don't do this)
$results = [];
$mutex = new Mutex();

parallel([
    fn() => { $mutex->lock(); $results[] = fetch1(); $mutex->unlock(); },
    fn() => { $mutex->lock(); $results[] = fetch2(); $mutex->unlock(); },
]);
```

Go uses channels:

```go
results := make(chan string, 2)  // Channel of strings

go func() {
    results <- fetch1()  // Send to channel
}()

go func() {
    results <- fetch2()  // Send to channel
}()

r1 := <-results  // Receive from channel
r2 := <-results  // Receive from channel
```

Creating Channels

```go
// Unbuffered channel (blocks until receiver ready)
ch := make(chan int)

// Buffered channel (can hold 10 items before blocking)
ch := make(chan int, 10)
```

```go
// Channel of structs
ch := make(chan User)

// Channel of channels (yes, this is valid)
ch := make(chan chan int)
```

Basic Operations

```go
ch := make(chan string)

// Send (blocks until someone receives)
ch <- "hello"

// Receive (blocks until something sent)
msg := <-ch

// Receive and discard
<-ch
```

Buffered vs Unbuffered

Unbuffered Channels

```go
ch := make(chan int)  // Unbuffered

go func() {
    ch <- 42  // Blocks until someone receives
}()

val := <-ch  // Receives 42
```

Unbuffered channels synchronise goroutines—the sender blocks until the receiver is ready.

Buffered Channels

```go
ch := make(chan int, 3)  // Buffer size 3

ch <- 1  // Doesn't block (buffer has space)
ch <- 2  // Doesn't block
ch <- 3  // Doesn't block
ch <- 4  // BLOCKS until someone receives

val := <-ch  // Receives 1, makes room for next send
```

Buffered channels decouple sender and receiver: - Sender blocks only when buffer is full - Receiver blocks only when buffer is empty

When to Use Which

Unbuffered (default): - Synchronisation points - Guaranteed handoff - Simple request-response

Buffered: - Asynchronous communication - Rate smoothing - Known batch sizes

Channel Directions (Send-Only, Receive-Only)

Function signatures can restrict channel direction:

```
// Send-only channel
func producer(out chan<- int) {
    out <- 42
    // <-out  // Error: cannot receive from send-only channel
}

// Receive-only channel
func consumer(in <-chan int) {
    val := <-in
    // in <- 1  // Error: cannot send to receive-only channel
}

// Bidirectional in main, restricted in functions
func main() {
    ch := make(chan int)

    go producer(ch)  // ch becomes send-only inside producer
    go consumer(ch)  // ch becomes receive-only inside consumer
}
```

This provides compile-time safety—functions can't misuse channels.

Closing Channels

The sender signals completion by closing:

```
ch := make(chan int)

go func() {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch)  // Signal no more values
}()

// Receivers detect closure
for val := range ch {
    fmt.Println(val)  // Prints 0, 1, 2, 3, 4
}
// Loop exits when channel closed
```

Detecting Closure

```
val, ok := <-ch
if !ok {
    // Channel is closed
}
```

Closing Rules

- Only senders should close channels
- Closing a closed channel panics
- Sending on a closed channel panics
- Receiving from a closed channel returns zero value and `false`

Range Over Channels

The `range` keyword iterates until channel closes:

```
func producer(ch chan<- int) {
    for i := 0; i < 10; i++ {
        ch <- i
    }
    close(ch)
}

func main() {
    ch := make(chan int)
    go producer(ch)

    for val := range ch {
        fmt.Println(val)
    }
    // Exits after channel closes
}
```

This is idiomatic for producer-consumer patterns.

Practical Example: Parallel Processing

```
func processItems(items []Item) []Result {
    results := make(chan Result, len(items))

    // Start workers
    for _, item := range items {
        item := item  // Capture for goroutine
        go func() {
            results <- process(item)
        }()
    }
```

```
    // Collect results
    output := make([]Result, 0, len(items))
    for range items {
        output = append(output, <-results)
    }

    return output
}
```

With Error Handling

```
type result struct {
    value Result
    err   error
}

func processItems(ctx context.Context, items []Item) ([]Result, error) {
    results := make(chan result, len(items))

    for _, item := range items {
        item := item
        go func() {
            val, err := process(ctx, item)
            results <- result{val, err}
        }()
    }

    output := make([]Result, 0, len(items))
    for range items {
        r := <-results
        if r.err != nil {
            return nil, r.err  // Fail fast
        }
        output = append(output, r.value)
    }

    return output, nil
}
```

Common Patterns

Generator Pattern

```
func generateNumbers(max int) <-chan int {
    ch := make(chan int)
    go func() {
        defer close(ch)
        for i := 0; i < max; i++ {
            ch <- i
```

```
        }
    }()
    return ch
}

// Usage
for num := range generateNumbers(100) {
    fmt.Println(num)
}
```

Request-Response

```
type Request struct {
    Query    string
    Response chan<- Result  // Channel to send response
}

func server(requests <-chan Request) {
    for req := range requests {
        result := processQuery(req.Query)
        req.Response <- result
    }
}

// Client
func query(requests chan<- Request, q string) Result {
    response := make(chan Result)
    requests <- Request{Query: q, Response: response}
    return <-response
}
```

Done Channel

```
func worker(done <-chan struct{}, work <-chan Job) {
    for {
        select {
        case <-done:
            return  // Stop when done is closed
        case job := <-work:
            process(job)
        }
    }
}

// Usage
done := make(chan struct{})
// ... start workers ...
close(done)  // Signal all workers to stop
```

Summary

- **Channels** are typed conduits for goroutine communication
- **Unbuffered channels** synchronise sender and receiver
- **Buffered channels** allow asynchronous communication
- **Channel directions** provide compile-time safety
- **Closing channels** signals no more values
- **Range over channels** iterates until closure

Exercises

1. **Basic Channel**: Create a goroutine that sends 10 numbers on a channel. Receive and print them in main.
2. **Buffered Channel**: Implement a simple job queue with a buffered channel of size 5. Observe blocking behaviour.
3. **Direction Restrictions**: Write producer and consumer functions with appropriate channel direction restrictions.
4. **Generator**: Implement a Fibonacci generator that returns a receive-only channel.
5. **Request-Response**: Implement a calculator server that receives operations via channel and returns results.
6. **Parallel Map**: Write a function that applies a transformation to slice elements in parallel using channels.
7. **Fan-Out**: Send work to multiple worker goroutines via a shared channel.
8. **Done Pattern**: Implement graceful shutdown using a done channel to signal workers to stop.

Chapter 17: Select and Coordination

Real concurrent programs coordinate multiple channels, handle timeouts, and propagate cancellation. This chapter covers the tools for sophisticated coordination.

Select Statements

The `select` statement waits on multiple channel operations:

```
select {
case msg := <-ch1:
    fmt.Println("Received from ch1:", msg)
case msg := <-ch2:
    fmt.Println("Received from ch2:", msg)
case ch3 <- value:
    fmt.Println("Sent to ch3")
}
```

`select` blocks until one case can proceed, then executes that case. If multiple cases are ready, one is chosen at random.

Non-Blocking Operations

Use `default` for non-blocking:

```
select {
case msg := <-ch:
    process(msg)
default:
    // Channel wasn't ready, do something else
}
```

Infinite Select Loop

```
for {
    select {
    case msg := <-input:
        process(msg)
    case <-done:
        return
    }
}
```

This is the foundation for event loops in Go.

Timeouts and Deadlines

PHP might use cURL timeouts:

```
$client = new HttpClient(['timeout' => 5.0]);
```

Go uses `time.After` or context:

Timeout with Select

```
select {
case result := <-doWork():
    return result
case <-time.After(5 * time.Second):
    return nil, errors.New("operation timed out")
}
```

Ticker for Periodic Work

```
ticker := time.NewTicker(1 * time.Second)
defer ticker.Stop()

for {
    select {
    case <-ticker.C:
        doPeriodicWork()
    case <-done:
        return
    }
}
```

Context Package Deep Dive

The `context` package is Go's standard for cancellation, deadlines, and request-scoped values.

Why Context?

Imagine an HTTP handler that: 1. Queries the database 2. Calls an external API 3. Processes results

If the client disconnects, all this work should stop. Context propagates cancellation signals through the call tree.

Creating Contexts

```
// Background context (root, never cancelled)
ctx := context.Background()

// With timeout
ctx, cancel := context.WithTimeout(parent, 5*time.Second)
defer cancel()  // Always call cancel to release resources

// With deadline
deadline := time.Now().Add(30 * time.Second)
ctx, cancel := context.WithDeadline(parent, deadline)
defer cancel()

// Manually cancellable
ctx, cancel := context.WithCancel(parent)
// Call cancel() when you want to cancel
```

Using Context

```
func fetchData(ctx context.Context) (*Data, error) {
    // Check if already cancelled
    if ctx.Err() != nil {
        return nil, ctx.Err()
    }

    // Pass context to operations
    resp, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    // ...

    rows, err := db.QueryContext(ctx, "SELECT ...")
    // ...
}
```

Context in Select

```
func worker(ctx context.Context, jobs <-chan Job) {
    for {
        select {
        case <-ctx.Done():
            log.Println("Worker cancelled:", ctx.Err())
            return
        case job := <-jobs:
            processJob(ctx, job)
        }
    }
}
```

HTTP Handler Context

```
func handler(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()  // Cancelled if client disconnects

    result, err := fetchData(ctx)
    if err == context.Canceled {
        // Client disconnected, stop work
        return
    }
    // ...
}
```

Cancellation Propagation

Cancellation flows down the context tree:

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())

    go worker1(ctx)
    go worker2(ctx)
    go worker3(ctx)

    time.Sleep(5 * time.Second)
    cancel()  // All workers receive cancellation
}

func worker1(ctx context.Context) {
    <-ctx.Done()  // Unblocks when cancel() called
    fmt.Println("worker1 stopping")
}
```

Nested Contexts

```
func handler(w http.ResponseWriter, r *http.Request) {
    // Request context (cancelled on disconnect)
    ctx := r.Context()

    // Add timeout for this specific operation
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

    // Cancelled if: client disconnects OR timeout
    result, err := fetchData(ctx)
}
```

WaitGroups

`sync.WaitGroup` waits for a collection of goroutines to finish:

```
var wg sync.WaitGroup

for i := 0; i < 10; i++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done()
        doWork(id)
    }(i)
}

wg.Wait()  // Block until all Done() called
fmt.Println("All workers finished")
```

WaitGroup Rules

- Call `Add` before starting the goroutine
- Call `Done` when the goroutine completes (usually `defer`)
- `Wait` blocks until counter reaches zero

Combining WaitGroup with Context

```
func processAll(ctx context.Context, items []Item) error {
    var wg sync.WaitGroup
    errs := make(chan error, len(items))

    for _, item := range items {
        wg.Add(1)
        item := item
        go func() {
            defer wg.Done()
            if err := process(ctx, item); err != nil {
                errs <- err
            }
        }()
    }

    // Wait in separate goroutine
    go func() {
        wg.Wait()
        close(errs)
    }()

    // Check for first error
    for err := range errs {
        if err != nil {
            return err
        }
```

```
    }
    return nil
}
```

errgroup for Error Handling

The `golang.org/x/sync/errgroup` package combines WaitGroup with error handling:

```
import "golang.org/x/sync/errgroup"

func fetchAll(ctx context.Context, urls []string) ([]Result, error) {
    g, ctx := errgroup.WithContext(ctx)
    results := make([]Result, len(urls))

    for i, url := range urls {
        i, url := i, url
        g.Go(func() error {
            result, err := fetch(ctx, url)
            if err != nil {
                return err  // Cancels context, stops other goroutines
            }
            results[i] = result
            return nil
        })
    }

    if err := g.Wait(); err != nil {
        return nil, err
    }
    return results, nil
}
```

Key features: - First error cancels the context - `Wait` returns the first error - All goroutines share the cancellable context

Summary

- **Select** waits on multiple channels simultaneously
- **Timeouts** use `time.After` or context deadlines
- **Context** propagates cancellation and deadlines
- **WaitGroups** wait for goroutine completion
- **errgroup** combines waiting with error handling

Exercises

1. **Multi-Channel Select**: Create two goroutines sending on different channels. Use select to receive from whichever is ready first.

2. **Timeout Implementation**: Write a function that returns an error if an operation takes longer than a specified duration.
3. **Context Cancellation**: Create a worker that respects context cancellation. Verify it stops when context is cancelled.
4. **Deadline Propagation**: Implement a chain of three functions, each adding context. Verify deadline propagates through all.
5. **WaitGroup Coordination**: Start N workers, wait for all to complete, then aggregate their results.
6. **errgroup Usage**: Fetch data from 5 URLs using errgroup. Handle the first error appropriately.
7. **Graceful HTTP Server**: Implement an HTTP server that gracefully shuts down on SIGINT, waiting for active requests.
8. **Heartbeat Pattern**: Implement a worker that sends periodic heartbeats on a channel while processing work.

Chapter 18: Concurrency Patterns

Armed with goroutines, channels, select, and context, you can implement powerful concurrency patterns. These patterns solve common problems elegantly.

Worker Pools

Process items in parallel with a fixed number of workers:

```
func workerPool(ctx context.Context, jobs <-chan Job, results chan<- Result, numWorkers int) {
    var wg sync.WaitGroup

    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go func(workerID int) {
            defer wg.Done()
            for {
                select {
                case <-ctx.Done():
                    return
                case job, ok := <-jobs:
                    if !ok {
                        return  // Jobs channel closed
                    }
                    results <- processJob(job)
                }
            }
        }(i)
    }

    wg.Wait()
    close(results)
}

// Usage
func processAll(items []Item) []Result {
    jobs := make(chan Job, len(items))
    results := make(chan Result, len(items))

    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // Start worker pool
    go workerPool(ctx, jobs, results, 10)

    // Send jobs
    for _, item := range items {
```

```
        jobs <- Job{Item: item}
    }
    close(jobs)

    // Collect results
    var output []Result
    for result := range results {
        output = append(output, result)
    }
    return output
}
```

Bounded Worker Pool

Limit concurrent work to prevent resource exhaustion:

```
type Pool struct {
    sem     chan struct{}
    wg      sync.WaitGroup
}

func NewPool(maxWorkers int) *Pool {
    return &Pool{
        sem: make(chan struct{}, maxWorkers),
    }
}

func (p *Pool) Submit(task func()) {
    p.wg.Add(1)
    go func() {
        p.sem <- struct{}{}  // Acquire semaphore
        defer func() {
            <-p.sem  // Release semaphore
            p.wg.Done()
        }()
        task()
    }()
}

func (p *Pool) Wait() {
    p.wg.Wait()
}
```

Fan-Out/Fan-In

Fan-out: Distribute work across multiple goroutines. **Fan-in**: Combine results from multiple goroutines.

```go
func fanOut(input <-chan int, numWorkers int) []<-chan int {
    outputs := make([]<-chan int, numWorkers)

    for i := 0; i < numWorkers; i++ {
        outputs[i] = worker(input)
    }

    return outputs
}

func worker(input <-chan int) <-chan int {
    output := make(chan int)
    go func() {
        defer close(output)
        for n := range input {
            output <- process(n)
        }
    }()
    return output
}

func fanIn(inputs ...<-chan int) <-chan int {
    output := make(chan int)
    var wg sync.WaitGroup

    for _, in := range inputs {
        wg.Add(1)
        in := in
        go func() {
            defer wg.Done()
            for n := range in {
                output <- n
            }
        }()
    }

    go func() {
        wg.Wait()
        close(output)
    }()

    return output
}

// Usage
func processNumbers(numbers []int) []int {
    input := make(chan int)
    go func() {
        defer close(input)
        for _, n := range numbers {
            input <- n
        }
```

```
    }()

    // Fan out to 5 workers
    workers := fanOut(input, 5)

    // Fan in results
    results := fanIn(workers...)

    var output []int
    for result := range results {
        output = append(output, result)
    }
    return output
}
```

Pipeline Processing

Connect stages of processing:

```
func generator(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            out <- n
        }
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            out <- n * n
        }
    }()
    return out
}

func filter(in <-chan int, predicate func(int) bool) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            if predicate(n) {
                out <- n
            }
        }
    }()
```

```
    return out
}

// Pipeline: generate → square → filter even
func main() {
    numbers := generator(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    squared := square(numbers)
    even := filter(squared, func(n int) bool { return n%2 == 0 })

    for n := range even {
        fmt.Println(n)  // 4, 16, 36, 64, 100
    }
}
```

Pipeline with Context

```
func pipelineWithCancel(ctx context.Context, nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for _, n := range nums {
            select {
            case <-ctx.Done():
                return
            case out <- n:
            }
        }
    }()
    return out
}
```

Semaphores

Limit concurrent access to a resource:

```
type Semaphore chan struct{}

func NewSemaphore(n int) Semaphore {
    return make(chan struct{}, n)
}

func (s Semaphore) Acquire() {
    s <- struct{}{}
}

func (s Semaphore) Release() {
    <-s
}

// Usage: limit concurrent HTTP requests
```

```go
func fetchAll(urls []string) {
    sem := NewSemaphore(10)  // Max 10 concurrent
    var wg sync.WaitGroup

    for _, url := range urls {
        wg.Add(1)
        url := url
        go func() {
            defer wg.Done()
            sem.Acquire()
            defer sem.Release()
            fetch(url)
        }()
    }

    wg.Wait()
}
```

Rate Limiting

Control the rate of operations:

```go
// Simple rate limiter
func rateLimiter(rate time.Duration) <-chan time.Time {
    return time.Tick(rate)
}

func processWithRateLimit(items []Item) {
    limiter := rateLimiter(100 * time.Millisecond)  // 10 per second

    for _, item := range items {
        <-limiter  // Wait for tick
        go process(item)
    }
}
```

Token Bucket Rate Limiter

```go
import "golang.org/x/time/rate"

func main() {
    // 10 requests per second, burst of 5
    limiter := rate.NewLimiter(10, 5)

    for i := 0; i < 100; i++ {
        if err := limiter.Wait(context.Background()); err != nil {
            log.Fatal(err)
        }
        go handleRequest(i)
    }
```

```
}
```

Graceful Shutdown

Stop accepting new work, complete in-flight work, then exit:

```
func main() {
    // Create server
    server := &http.Server{Addr: ":8080"}

    // Start server in goroutine
    go func() {
        if err := server.ListenAndServe(); err != http.ErrServerClosed {
            log.Fatal(err)
        }
    }()

    // Wait for interrupt signal
    quit := make(chan os.Signal, 1)
    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
    <-quit

    log.Println("Shutting down...")

    // Give outstanding requests 30 seconds to complete
    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()

    if err := server.Shutdown(ctx); err != nil {
        log.Fatal("Forced shutdown:", err)
    }

    log.Println("Server stopped")
}
```

Worker Pool Graceful Shutdown

```
type WorkerPool struct {
    jobs     chan Job
    quit     chan struct{}
    wg       sync.WaitGroup
}

func (p *WorkerPool) Start(numWorkers int) {
    for i := 0; i < numWorkers; i++ {
        p.wg.Add(1)
        go p.worker()
    }
}
```

```go
func (p *WorkerPool) worker() {
    defer p.wg.Done()
    for {
        select {
        case <-p.quit:
            return
        case job, ok := <-p.jobs:
            if !ok {
                return
            }
            process(job)
        }
    }
}

func (p *WorkerPool) Stop() {
    close(p.quit)  // Signal workers to stop
    p.wg.Wait()    // Wait for workers to finish
}

func (p *WorkerPool) StopAcceptingAndDrain() {
    close(p.jobs)  // Stop accepting new jobs
    p.wg.Wait()    // Wait for in-flight jobs
}
```

Summary

- **Worker pools** limit concurrent workers to N
- **Fan-out/fan-in** distributes and collects work
- **Pipelines** chain processing stages
- **Semaphores** limit concurrent resource access
- **Rate limiting** controls operation frequency
- **Graceful shutdown** completes work before exit

Exercises

1. **Worker Pool Implementation**: Build a worker pool that processes 1000 items with 10 workers. Measure speedup versus sequential.
2. **Fan-Out/Fan-In**: Implement image processing that fans out to resize images concurrently, then fans in results.
3. **Pipeline Design**: Create a data processing pipeline with 4 stages. Add context cancellation to all stages.
4. **Semaphore for API**: Use a semaphore to limit concurrent API requests to 5. Verify rate limiting works.
5. **Token Bucket**: Implement a token bucket rate limiter from scratch. Test with burst traffic.

6. **Graceful HTTP Server**: Build an HTTP server with graceful shutdown. Verify long requests complete during shutdown.
7. **Combined Patterns**: Build a system that uses worker pool + rate limiting + graceful shutdown together.
8. **Pipeline Metrics**: Add metrics to a pipeline (items processed, time per stage, errors). Make them accessible via HTTP.

Chapter 19: When Concurrency Goes Wrong

Concurrency introduces failure modes that don't exist in sequential PHP code. Race conditions, deadlocks, and goroutine leaks are new territory for PHP developers.

Race Conditions (New Territory for PHP Developers)

A race condition occurs when multiple goroutines access shared data, and at least one modifies it:

```
// RACE CONDITION!
var counter int

func increment() {
    counter++  // Read-modify-write is not atomic
}

func main() {
    for i := 0; i < 1000; i++ {
        go increment()
    }
    time.Sleep(time.Second)
    fmt.Println(counter)  // Not 1000!
}
```

`counter++` is three operations: 1. Read current value 2. Add one 3. Write new value

Two goroutines can: 1. Both read 5 2. Both write 6 3. One increment is lost

Why PHP Developers Don't See This

PHP's shared-nothing model means each request has its own memory:

```
// Each request has its own $counter
$counter = 0;
$counter++;  // No race possible
```

Multiple PHP requests might race on database state, but not in-memory state.

Fixing Race Conditions

Option 1: Mutex

```go
var (
    counter int
    mu      sync.Mutex
)

func increment() {
    mu.Lock()
    counter++
    mu.Unlock()
}
```

Option 2: Atomic operations

```go
var counter int64

func increment() {
    atomic.AddInt64(&counter, 1)
}
```

Option 3: Channels (share by communicating)

```go
func counter(increments <-chan struct{}) <-chan int {
    result := make(chan int)
    go func() {
        count := 0
        for range increments {
            count++
        }
        result <- count
        close(result)
    }()
    return result
}
```

The Race Detector

Go has a built-in race detector:

```
go run -race main.go
go test -race ./...
```

Output for the counter example:

```
WARNING: DATA RACE
Read at 0x00c00001c0b8 by goroutine 7:
  main.increment()
      main.go:10 +0x3a

Previous write at 0x00c00001c0b8 by goroutine 6:
  main.increment()
```

```
main.go:10 +0x50
```

Using the Race Detector

- Run tests with `-race` in CI
- Test under realistic concurrency (race detector needs actual concurrent access)
- Races are non-deterministic—run tests multiple times
- The race detector slows execution 2-10x; don't use in production

Common Race Patterns

Map access:

```go
// RACE!
var cache = make(map[string]string)

go func() { cache["a"] = "1" }()
go func() { _ = cache["b"] }()
```

Fix with `sync.Map` or mutex:

```go
var cache sync.Map
cache.Store("a", "1")
val, _ := cache.Load("a")
```

Slice append:

```go
// RACE!
var results []int

go func() { results = append(results, 1) }()
go func() { results = append(results, 2) }()
```

Struct field access:

```go
// RACE if fields accessed concurrently
type Stats struct {
    Count int
    Total int
}
```

Deadlocks

A deadlock occurs when goroutines wait for each other forever:

```go
// DEADLOCK!
func main() {
    ch := make(chan int)  // Unbuffered
    ch <- 1  // Blocks forever-no receiver!
```

```
    fmt.Println(<-ch)
}
```

Classic Deadlock: Two Mutexes

```
var mu1, mu2 sync.Mutex

// Goroutine 1
go func() {
    mu1.Lock()
    time.Sleep(time.Millisecond)
    mu2.Lock()  // Waits for goroutine 2
    // ...
}()

// Goroutine 2
go func() {
    mu2.Lock()
    time.Sleep(time.Millisecond)
    mu1.Lock()  // Waits for goroutine 1
    // ...
}()

// DEADLOCK! Both wait forever
```

Prevention Strategies

Consistent lock ordering:

```
// Always lock mu1 before mu2
func operation() {
    mu1.Lock()
    mu2.Lock()
    // ...
    mu2.Unlock()
    mu1.Unlock()
}
```

Use timeouts:

```
select {
case ch <- value:
    // Sent successfully
case <-time.After(5 * time.Second):
    // Timed out, handle gracefully
}
```

Use context with deadline:

```
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

select {
case result := <-doWork(ctx):
    return result, nil
case <-ctx.Done():
    return nil, ctx.Err()
}
```

Channel Leaks

A channel leak occurs when a goroutine is blocked on a channel forever:

```
// LEAK!
func process(items []int) <-chan int {
    results := make(chan int)

    go func() {
        for _, item := range items {
            if item < 0 {
                return  // Exits without closing channel
            }
            results <- item * 2
        }
        close(results)
    }()

    return results
}

// Caller waits forever if early return
for result := range process([]int{1, 2, -1, 4}) {
    fmt.Println(result)
}
```

Preventing Leaks

Always close channels:

```
go func() {
    defer close(results)  // Always closes
    for _, item := range items {
        if item < 0 {
            return
        }
        results <- item * 2
    }
}()
```

Use context for cancellation:

```go
func process(ctx context.Context, items []int) <-chan int {
    results := make(chan int)

    go func() {
        defer close(results)
        for _, item := range items {
            select {
            case <-ctx.Done():
                return
            case results <- item * 2:
            }
        }
    }()

    return results
}
```

Debugging Concurrent Code

Goroutine Dumps

```go
import "runtime/pprof"

// Dump all goroutine stacks
pprof.Lookup("goroutine").WriteTo(os.Stdout, 1)
```

Or via HTTP:

```go
import _ "net/http/pprof"

go func() {
    http.ListenAndServe(":6060", nil)
}()
```

Then: `curl http://localhost:6060/debug/pprof/goroutine?debug=1`

Counting Goroutines

```go
fmt.Println("Goroutines:", runtime.NumGoroutine())
```

Growing goroutine count suggests leaks.

Logging with Goroutine ID

```go
func getGoroutineID() uint64 {
    b := make([]byte, 64)
    runtime.Stack(b, false)
    var id uint64
```

```
    fmt.Sscanf(string(b), "goroutine %d", &id)
    return id
}

log.Printf("[goroutine %d] processing item", getGoroutineID())
```

Common Mistakes from PHP Developers

1. Forgetting Goroutines Outlive Function Calls

```
func handler(w http.ResponseWriter, r *http.Request) {
    go sendEmail(email)  // Goroutine continues after handler returns
    w.Write([]byte("OK"))
}

// If server shuts down, email might not send
```

2. Closing Channels from Wrong Side

```
// WRONG: Receiver closing sender's channel
go func() {
    for val := range ch {
        if val < 0 {
            close(ch)  // Sender will panic!
        }
    }
}()
```

3. Assuming Channel Order

```
ch := make(chan int, 10)
for i := 0; i < 10; i++ {
    go func(n int) {
        ch <- n
    }(i)
}

// Order is NOT guaranteed!
for i := 0; i < 10; i++ {
    fmt.Println(<-ch)  // Could be any order
}
```

4. Not Waiting for Goroutines

```
func main() {
    go doWork()
    // Program exits before goroutine completes!
}
```

Summary

- **Race conditions** occur when goroutines share mutable data
- **The race detector** (`-race`) finds races automatically
- **Deadlocks** happen when goroutines wait for each other
- **Channel leaks** leave goroutines blocked forever
- **Common mistakes** include assuming order and not waiting

Exercises

1. **Create a Race**: Write code with an intentional race condition. Verify the race detector finds it.
2. **Fix the Race**: Fix the race using mutex, atomic, and channel approaches. Benchmark each.
3. **Deadlock Scenario**: Create a deadlock with two mutexes. Then fix it with consistent ordering.
4. **Channel Leak**: Create a goroutine leak. Use `runtime.NumGoroutine()` to detect it.
5. **Race Detector CI**: Add race detection to a test suite. Simulate running in CI.
6. **Debug with pprof**: Set up pprof HTTP endpoint. Analyse goroutine dumps.
7. **Timeout Prevention**: Take blocking code and add timeouts to prevent hangs.
8. **Leak Prevention Pattern**: Implement a worker pattern that guarantees no goroutine leaks even on errors.

Chapter 20: Reflection and Code Generation

PHP uses reflection extensively—dependency injection, ORMs, serialisation. Go has reflection too, but the culture prefers code generation. Understanding both is key to writing idiomatic Go.

reflect Package Basics

PHP reflection:

```php
$reflectionClass = new ReflectionClass(User::class);
$properties = $reflectionClass->getProperties();
$methods = $reflectionClass->getMethods();
```

Go reflection:

```go
import "reflect"

type User struct {
    Name  string `json:"name"`
    Email string `json:"email"`
}

func inspectType(v any) {
    t := reflect.TypeOf(v)
    fmt.Println("Type:", t.Name())
    fmt.Println("Kind:", t.Kind())

    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        fmt.Printf("Field: %s, Type: %s, Tag: %s\n",
            field.Name, field.Type, field.Tag.Get("json"))
    }
}

inspectType(User{})
// Type: User
// Kind: struct
// Field: Name, Type: string, Tag: name
// Field: Email, Type: string, Tag: email
```

Type vs Value

```
t := reflect.TypeOf(user)   // Type information
v := reflect.ValueOf(user)  // Actual value

// Get field value
nameValue := v.FieldByName("Name")
fmt.Println(nameValue.String())

// Set field value (must be addressable)
v := reflect.ValueOf(&user).Elem()
v.FieldByName("Name").SetString("New Name")
```

Calling Methods via Reflection

```
method := reflect.ValueOf(user).MethodByName("Greet")
args := []reflect.Value{reflect.ValueOf("World")}
results := method.Call(args)
fmt.Println(results[0].String())
```

When to Use Reflection (Rarely)

Legitimate uses:

1. Serialisation/Deserialisation

```
// encoding/json uses reflection internally
func marshalStruct(v any) ([]byte, error) {
    val := reflect.ValueOf(v)
    if val.Kind() != reflect.Struct {
        return nil, errors.New("expected struct")
    }

    result := make(map[string]any)
    typ := val.Type()

    for i := 0; i < val.NumField(); i++ {
        field := typ.Field(i)
        jsonTag := field.Tag.Get("json")
        if jsonTag == "-" {
            continue
        }
        name := jsonTag
        if name == "" {
            name = field.Name
        }
        result[name] = val.Field(i).Interface()
    }
```

```
    return json.Marshal(result)
}
```

2. Generic Utilities

```
func isNil(v any) bool {
    if v == nil {
        return true
    }
    val := reflect.ValueOf(v)
    switch val.Kind() {
    case reflect.Chan, reflect.Func, reflect.Map,
         reflect.Pointer, reflect.Interface, reflect.Slice:
        return val.IsNil()
    }
    return false
}
```

3. Testing Utilities

```
func assertEqual(t *testing.T, expected, actual any) {
    if !reflect.DeepEqual(expected, actual) {
        t.Errorf("expected %v, got %v", expected, actual)
    }
}
```

When NOT to Use Reflection

- **Regular type switching**: Use type assertions or generics
- **Performance-critical code**: Reflection is slow
- **When types are known**: Just use the type directly

Code Generation: `go generate`

Go culture prefers generating code at build time over reflection at runtime.

The `go generate` Command

```
//go:generate stringer -type=Status

type Status int

const (
    StatusPending Status = iota
    StatusActive
    StatusCompleted
)
```

Running `go generate ./...` invokes the `stringer` tool, which generates:

```
// status_string.go (generated)
func (s Status) String() string {
    switch s {
    case StatusPending:
        return "Pending"
    case StatusActive:
        return "Active"
    case StatusCompleted:
        return "Completed"
    }
    return fmt.Sprintf("Status(%d)", s)
}
```

Writing a Simple Generator

```
// gen/main.go
package main

import (
    "os"
    "text/template"
)

const tmpl = `
// Code generated by gen/main.go. DO NOT EDIT.
package {{.Package}}

var Endpoints = []string{
{{range .Endpoints}}    "{{.}}",
{{end}}}
`

func main() {
    data := struct {
        Package   string
        Endpoints []string
    }{
        Package:   "api",
        Endpoints: []string{"/users", "/orders", "/products"},
    }

    t := template.Must(template.New("endpoints").Parse(tmpl))
    f, _ := os.Create("endpoints_gen.go")
    defer f.Close()
    t.Execute(f, data)
}
```

Usage:

```
//go:generate go run ./gen
```

Build-Time vs Runtime (Unlike PHP's Runtime Reflection)

PHP resolves types at runtime:

```php
// Symfony Container: Resolved at runtime (cached, but still runtime)
$service = $container->get(UserService::class);

// Doctrine: Metadata parsed at runtime
$user = $em->find(User::class, $id);
```

Go prefers build-time resolution:

```go
// Wire: Generates wiring code at build time
//go:generate wire

func InitializeApp() *App {
    wire.Build(NewApp, NewUserService, NewUserRepository, NewDatabase)
    return nil  // Wire replaces this
}
```

Generated `wire_gen.go`:

```go
func InitializeApp() *App {
    database := NewDatabase()
    userRepository := NewUserRepository(database)
    userService := NewUserService(userRepository)
    app := NewApp(userService)
    return app
}
```

Benefits of Build-Time

1. **No runtime overhead**: Code is just function calls
2. **Compile-time safety**: Errors caught during build
3. **Better debugging**: Generated code is readable
4. **No surprises**: What you see is what runs

SQLC, Wire, and Other Generators

SQLC: SQL to Go

```sql
-- queries.sql
-- name: GetUser :one
SELECT * FROM users WHERE id = $1;
```

Generates:

```go
func (q *Queries) GetUser(ctx context.Context, id int64) (User, error) {
    row := q.db.QueryRowContext(ctx, getUserQuery, id)
    var i User
    err := row.Scan(&i.ID, &i.Name, &i.Email)
    return i, err
}
```

Wire: Dependency Injection

```go
// wire.go
//go:build wireinject

func InitializeServer() (*Server, error) {
    wire.Build(
        NewServer,
        NewUserHandler,
        NewUserService,
        NewUserRepository,
        NewDatabase,
    )
    return nil, nil
}
```

mockgen: Interface Mocks

```go
//go:generate mockgen -source=repository.go -destination=mock_repository.go

type UserRepository interface {
    Find(ctx context.Context, id int) (*User, error)
}
```

Other Popular Generators

- **stringer**: String methods for enums
- **enumer**: Extended enum utilities
- **go-bindata**: Embed binary files
- **protoc-gen-go**: Protocol Buffers
- **oapi-codegen**: OpenAPI to Go

Summary

- **Reflection** inspects types and values at runtime
- **Use reflection sparingly**—it's slow and bypasses type safety
- **Code generation** creates type-safe code at build time
- **`go generate`** runs generators defined in source comments
- **Popular generators** include SQLC, Wire, mockgen, stringer

---

Exercises

1. **Struct Inspector**: Write a function that uses reflection to print all fields and tags of any struct.
2. **Generic Validator**: Use reflection to validate struct fields based on tags (e.g., `validate:"required"`).
3. **Simple Generator**: Write a generator that creates getter methods for struct fields.
4. **Wire Setup**: Set up Wire for a simple application with 3-4 dependencies.
5. **SQLC Workflow**: Set up SQLC for a schema. Write queries. Generate code. Use in tests.
6. **Reflection Performance**: Benchmark reflection-based field access versus direct access.
7. **Custom Stringer**: Write a generator that creates `String()` methods with custom formatting.
8. **Generator Testing**: Write tests for a code generator to ensure correct output.

Chapter 21: Performance Optimisation

PHP performance tuning involves OPcache, database queries, and caching layers. Go performance tuning is more granular—memory allocations, escape analysis, and CPU profiling become important.

Profiling: pprof (CPU, Memory, Goroutine)

Go has built-in profiling via `pprof`:

```
import (
    "net/http"
    _ "net/http/pprof"
)

func main() {
    go func() {
        http.ListenAndServe(":6060", nil)
    }()
    // Application code
}
```

CPU Profiling

```
# 30-second CPU profile
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

# Interactive commands
(pprof) top10        # Top functions by CPU
(pprof) list funcName  # Source with CPU annotations
(pprof) web          # Visualise in browser
```

Memory Profiling

```
go tool pprof http://localhost:6060/debug/pprof/heap

(pprof) top          # Top memory allocators
(pprof) list funcName  # Source with allocation annotations
```

Goroutine Profiling

```
go tool pprof http://localhost:6060/debug/pprof/goroutine

(pprof) top          # Where goroutines are stuck
```

Command-Line Profiling

```
import "runtime/pprof"

func main() {
    f, _ := os.Create("cpu.prof")
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()

    // Run your code

    // Memory profile
    f2, _ := os.Create("mem.prof")
    pprof.WriteHeapProfile(f2)
}
```

Then analyse:

```
go tool pprof cpu.prof
```

Benchmarking Methodology

Go benchmarks are built-in:

```
func BenchmarkProcess(b *testing.B) {
    input := generateInput()
    b.ResetTimer()  // Don't count setup

    for i := 0; i < b.N; i++ {
        Process(input)
    }
}
```

Run:

```
go test -bench=. -benchmem

# Output:
# BenchmarkProcess-8   1000000   1234 ns/op   256 B/op   2 allocs/op
```

Comparing Benchmarks

```
# Save baseline
go test -bench=. -count=5 > old.txt

# Make changes, re-run
go test -bench=. -count=5 > new.txt

# Compare
benchstat old.txt new.txt
```

Avoiding Benchmark Pitfalls

```
// BAD: Compiler might optimise away
func BenchmarkBad(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = compute()  // Result unused, might be eliminated
    }
}

// GOOD: Use result
var result int

func BenchmarkGood(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        r = compute()
    }
    result = r  // Prevent elimination
}
```

Memory Allocation Patterns

Allocation Costs

Each allocation has overhead: - Memory allocation - Garbage collection tracking - Potential GC pause contribution

Reducing Allocations

Pre-allocate slices:

```
// BAD: Multiple allocations as slice grows
var items []Item
for _, v := range data {
    items = append(items, transform(v))
}

// GOOD: Single allocation
```

```
items := make([]Item, 0, len(data))
for _, v := range data {
    items = append(items, transform(v))
}
```

Reuse buffers:

```
// BAD: New buffer each call
func process(data []byte) []byte {
    buf := new(bytes.Buffer)
    buf.Write(data)
    return buf.Bytes()
}

// GOOD: Reuse buffer
func (p *Processor) process(data []byte) []byte {
    p.buf.Reset()
    p.buf.Write(data)
    return p.buf.Bytes()
}
```

Use `strings.Builder`:

```
// BAD: String concatenation allocates
s := ""
for _, part := range parts {
    s += part
}

// GOOD: Builder
var b strings.Builder
for _, part := range parts {
    b.WriteString(part)
}
s := b.String()
```

Escape Analysis Awareness

Go's compiler decides whether variables live on stack (fast) or heap (slower, GC-tracked).

Viewing Escape Analysis

```
go build -gcflags="-m" .

# Output:
# ./main.go:10:6: moved to heap: x
# ./main.go:15:9: &User{} escapes to heap
```

Common Escape Causes

Returning pointers:

```
// Escapes to heap
func newUser() *User {
    u := User{Name: "Alice"}
    return &u  // Address taken, escapes
}

// Stays on stack
func newUser() User {
    u := User{Name: "Alice"}
    return u  // Value copy, no escape
}
```

Interface conversions:

```
func process(v any) { }

func main() {
    x := 42
    process(x)  // x escapes (boxed in interface)
}
```

Closures capturing variables:

```
func createCounter() func() int {
    count := 0
    return func() int {
        count++  // count escapes (captured by closure)
        return count
    }
}
```

When to Care

For hot paths (millions of calls), reducing allocations matters. For most code, clarity beats micro-optimisation.

Pool Patterns for Allocation Reduction

`sync.Pool` provides object reuse:

```
var bufferPool = sync.Pool{
    New: func() any {
        return new(bytes.Buffer)
    },
}
```

```go
func process(data []byte) {
    buf := bufferPool.Get().(*bytes.Buffer)
    defer func() {
        buf.Reset()
        bufferPool.Put(buf)
    }()

    buf.Write(data)
    // Use buffer
}
```

Pool Caveats

- Pool may be cleared between GC cycles
- Not for long-lived objects
- Best for frequently allocated short-lived objects
- Thread-safe

Common Pool Use Cases

- Byte buffers
- Temporary slices
- JSON encoder/decoder buffers
- HTTP request/response objects

Comparing to Blackfire/Xdebug Profiling

PHP profiling:

```
// Xdebug profiler output: cachegrind files
// Blackfire: Timeline visualisation
```

Go profiling is similar in concept but: - Built into the language (no extensions) - Works in production (low overhead) - Includes memory and goroutine profiling - `pprof` output is analysable offline

Go Profiling Workflow

1. **Identify**: Which endpoint or function is slow?
2. **Profile**: Run pprof on that code path
3. **Analyse**: Find hot spots (CPU) or allocation sources (memory)
4. **Optimise**: Fix the bottleneck
5. **Benchmark**: Verify improvement
6. **Repeat**: Profile again

Summary

- **pprof** profiles CPU, memory, and goroutines
- **Benchmarks** measure and compare performance

- **Allocation patterns** affect GC and speed
- **Escape analysis** determines stack vs heap
- **sync.Pool** reuses frequently allocated objects
- **Profile before optimising**—intuition is often wrong

Exercises

1. **CPU Profile**: Profile an application. Find the hottest function. Optimise it.
2. **Memory Profile**: Profile memory. Find the biggest allocator. Reduce allocations.
3. **Benchmark Comparison**: Write a function two ways. Benchmark both. Use benchstat to compare.
4. **Escape Analysis**: Write code that escapes to heap. Modify it to stay on stack. Verify with `-gcflags="-m"`.
5. **sync.Pool**: Add pooling to a frequently allocated object. Benchmark before and after.
6. **Allocation Audit**: Use `-benchmem` to find allocations in a function. Reduce to zero allocations.
7. **Goroutine Leak Detection**: Profile goroutines. Find where they're blocked. Fix the leak.
8. **Production Profiling**: Set up pprof in an HTTP server. Profile under load.

Chapter 22: Calling C and System Programming

PHP extensions are written in C, but most PHP developers never touch them. Go makes C interop and system programming more accessible—though it's still advanced territory.

CGO Basics

CGO allows Go to call C code:

```
package main

/*
#include <stdio.h>
#include <stdlib.h>

void sayHello(const char* name) {
    printf("Hello, %s!\n", name);
}
*/
import "C"

import "unsafe"

func main() {
    name := C.CString("World")
    defer C.free(unsafe.Pointer(name))
    C.sayHello(name)
}
```

The comment before `import "C"` contains C code. CGO compiles it and generates bindings.

C Types in Go

```
// C types
var i C.int = 42
var f C.float = 3.14
var s *C.char = C.CString("hello")

// Converting to Go types
goInt := int(i)
goFloat := float32(f)
goString := C.GoString(s)
```

Calling C Libraries

```go
/*
#cgo LDFLAGS: -lm
#include <math.h>
*/
import "C"

func main() {
    result := C.sqrt(16.0)
    fmt.Println(float64(result))  // 4.0
}
```

The `#cgo` directive sets compiler/linker flags.

Memory Management

C memory must be managed manually:

```go
// Allocate
ptr := C.malloc(100)
defer C.free(ptr)

// String to C (allocates)
cstr := C.CString("hello")
defer C.free(unsafe.Pointer(cstr))

// C string to Go (copies)
gostr := C.GoString(cstr)
```

When CGO Makes Sense

Good Use Cases

1. **Wrapping existing C libraries**: SQLite, OpenSSL, image codecs
2. **Performance-critical code**: When pure Go isn't fast enough
3. **System interfaces**: Hardware access, kernel interfaces
4. **Legacy integration**: Existing C codebases

When to Avoid CGO

1. **Cross-compilation**: CGO complicates builds
2. **Simple tasks**: Pure Go is often fast enough
3. **Concurrency**: CGO calls block OS threads
4. **Deployment**: Static binaries become harder

CGO Trade-offs

Pure Go	With CGO
Simple cross-compilation	Complex cross-compilation
Single static binary	May need shared libraries
Goroutine-friendly	Blocks OS threads
Memory-safe	Manual memory management

Syscalls and unsafe Package

For system programming without CGO:

```
import (
    "syscall"
    "unsafe"
)

func getpid() int {
    pid, _, _ := syscall.Syscall(syscall.SYS_GETPID, 0, 0, 0)
    return int(pid)
}
```

The unsafe Package

`unsafe` bypasses Go's type safety:

```
import "unsafe"

// Get size
size := unsafe.Sizeof(int64(0))  // 8

// Pointer arithmetic
arr := [3]int{1, 2, 3}
ptr := unsafe.Pointer(&arr[0])
ptr2 := unsafe.Add(ptr, unsafe.Sizeof(int(0)))
val := *(*int)(ptr2)  // 2

// Type punning
var f float64 = 3.14
bits := *(*uint64)(unsafe.Pointer(&f))
```

When to Use unsafe

- **Performance-critical code**: Avoiding copies
- **System interfaces**: Memory-mapped I/O
- **FFI**: Foreign function interface
- **Never in normal code**: There's usually a better way

Building CLI Tools

Go excels at CLI tools—single binary, fast startup, cross-platform:

```
package main

import (
    "flag"
    "fmt"
    "os"
)

func main() {
    verbose := flag.Bool("v", false, "verbose output")
    output := flag.String("o", "", "output file")
    flag.Parse()

    args := flag.Args()
    if len(args) == 0 {
        fmt.Fprintln(os.Stderr, "usage: mytool [-v] [-o file] input")
        os.Exit(1)
    }

    if *verbose {
        fmt.Println("Processing:", args[0])
    }

    // Process input
}
```

Using Cobra for Complex CLIs

```
import "github.com/spf13/cobra"

var rootCmd = &cobra.Command{
    Use:   "myapp",
    Short: "My application does amazing things",
}

var serveCmd = &cobra.Command{
    Use:   "serve",
    Short: "Start the server",
    Run: func(cmd *cobra.Command, args []string) {
        port, _ := cmd.Flags().GetInt("port")
        startServer(port)
    },
}

func init() {
    serveCmd.Flags().IntP("port", "p", 8080, "port to listen on")
    rootCmd.AddCommand(serveCmd)
```

```
}

func main() {
    rootCmd.Execute()
}
```

Signal Handling

Handle OS signals for graceful shutdown:

```
import (
    "os"
    "os/signal"
    "syscall"
)

func main() {
    // Setup
    server := startServer()

    // Handle signals
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

    <-sigChan  // Block until signal received

    fmt.Println("Shutting down...")
    server.Shutdown(context.Background())
}
```

Signal Handling Patterns

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())

    // Start workers with context
    go worker(ctx)

    // Handle signals
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

    select {
    case sig := <-sigChan:
        log.Printf("Received signal: %v", sig)
        cancel()  // Cancel context, workers should stop
    }

    // Wait for workers (with timeout)
    time.Sleep(5 * time.Second)
```

```
}
```

Handling SIGHUP for Config Reload

```
func main() {
    config := loadConfig()

    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGHUP)

    go func() {
        for range sigChan {
            log.Println("Reloading configuration...")
            config = loadConfig()
        }
    }()

    // Application runs with reloadable config
}
```

Summary

- **CGO** enables calling C code but adds complexity
- **Use CGO** for existing C libraries or performance-critical code
- **unsafe** bypasses type safety—use sparingly
- **CLI tools** are a Go strength—single binary deployment
- **Signal handling** enables graceful shutdown and config reload

---

Exercises

1. **CGO Hello World**: Write a Go program that calls a C function to print a message.
2. **Wrap a C Library**: Wrap a simple C library (e.g., math functions) with Go bindings.
3. **Memory Safety**: Demonstrate a memory leak in CGO code. Then fix it.
4. **CLI Tool**: Build a CLI tool with Cobra that has multiple subcommands and flags.
5. **Signal Handler**: Implement graceful shutdown on SIGINT with active request draining.
6. **SIGHUP Reload**: Implement configuration reload on SIGHUP without restart.
7. **unsafe Exploration**: Use unsafe to examine the memory layout of a struct.
8. **Cross-Compilation**: Cross-compile a Go program for multiple platforms. Then try with CGO.

Chapter 23: Building and Deploying

PHP deployment means copying files and configuring PHP-FPM. Go deployment means shipping a single binary. This simplicity transforms how you think about deployment.

Single Binary Deployment (vs PHP's File Deployment)

PHP deployment:

```
/var/www/myapp/
   composer.json
   composer.lock
   vendor/             # Dependencies
   public/
       index.php
   src/
   config/
   var/                # Cache, logs
   .env
```

Requires: - PHP runtime - Required extensions - Composer dependencies - Web server (nginx + PHP-FPM) - Writeable directories

Go deployment:

```
/opt/myapp/
   myapp               # Single binary
```

Requires: - Nothing

Building the Binary

```
# Simple build
go build -o myapp .

# Optimised build
go build -ldflags="-s -w" -o myapp .
# -s: Strip symbol table
# -w: Strip debug info
# Reduces binary size ~30%
```

Embedding Version Info

```go
// main.go
var (
    version = "dev"
    commit  = "unknown"
    date    = "unknown"
)

func main() {
    if len(os.Args) > 1 && os.Args[1] == "version" {
        fmt.Printf("version: %s\ncommit: %s\nbuilt: %s\n", version, commit, date)
        return
    }
    // ...
}
```

```
go build -ldflags="-X main.version=1.0.0 -X main.commit=$(git rev-parse HEAD) -X main.date=$(date -u
```

Cross-Compilation

PHP can't cross-compile. Go can:

```
# Linux from macOS
GOOS=linux GOARCH=amd64 go build -o myapp-linux .

# Windows from macOS
GOOS=windows GOARCH=amd64 go build -o myapp.exe .

# ARM (Raspberry Pi)
GOOS=linux GOARCH=arm GOARM=7 go build -o myapp-arm .

# Apple Silicon
GOOS=darwin GOARCH=arm64 go build -o myapp-darwin-arm64 .
```

Supported Platforms

```
go tool dist list
# Shows all GOOS/GOARCH combinations
```

Common targets: - `linux/amd64` — Linux servers - `linux/arm64` — AWS Graviton, Apple Silicon Linux - `darwin/amd64` — Intel Mac - `darwin/arm64` — Apple Silicon Mac - `windows/amd64` — Windows

Build Matrix

```
# Makefile
BINARY=myapp
VERSION=$(shell git describe --tags --always)

.PHONY: build-all
build-all:
    GOOS=linux GOARCH=amd64 go build -o dist/$(BINARY)-linux-amd64 .
    GOOS=linux GOARCH=arm64 go build -o dist/$(BINARY)-linux-arm64 .
    GOOS=darwin GOARCH=amd64 go build -o dist/$(BINARY)-darwin-amd64 .
    GOOS=darwin GOARCH=arm64 go build -o dist/$(BINARY)-darwin-arm64 .
    GOOS=windows GOARCH=amd64 go build -o dist/$(BINARY)-windows-amd64.exe .
```

Docker Images: Multi-Stage Builds

PHP Dockerfile:

```
FROM php:8.2-fpm
RUN apt-get update && apt-get install -y libpq-dev
RUN docker-php-ext-install pdo pdo_pgsql
COPY composer.json composer.lock ./
RUN composer install --no-dev
COPY . .
# Image size: 500MB+
```

Go multi-stage Dockerfile:

```
# Build stage
FROM golang:1.21 AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -ldflags="-s -w" -o myapp .

# Runtime stage
FROM scratch
COPY --from=builder /app/myapp /myapp
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
EXPOSE 8080
ENTRYPOINT ["/myapp"]
# Image size: ~10MB
```

Using `scratch` vs `alpine`

scratch (empty image):

```
FROM scratch
COPY --from=builder /app/myapp /myapp
# Size: Just your binary (~10-20MB)
# No shell, no debugging tools
```

alpine (minimal Linux):

```
FROM alpine:3.18
RUN apk --no-cache add ca-certificates
COPY --from=builder /app/myapp /myapp
# Size: ~15-25MB
# Has shell for debugging
```

distroless (Google's minimal images):

```
FROM gcr.io/distroless/static-debian11
COPY --from=builder /app/myapp /myapp
# Size: ~20MB
# Minimal but debuggable
```

No Runtime Dependencies

PHP requires: - PHP runtime - Extensions (pdo, json, mbstring, etc.) - Composer autoloader - Configuration files

Go binary is self-contained: - Statically linked (with CGO_ENABLED=0) - All dependencies compiled in - No runtime needed

Verifying Static Build

```
# Check if truly static
file myapp
# myapp: ELF 64-bit LSB executable, x86-64, ... statically linked

ldd myapp
# not a dynamic executable
```

Embedding Files

Go 1.16+ can embed files in the binary:

```
import "embed"

//go:embed static/*
var staticFiles embed.FS

//go:embed templates/*.html
var templates embed.FS
```

```
func main() {
    data, _ := staticFiles.ReadFile("static/index.html")
    // ...
}
```

No need to deploy separate static files—they're in the binary.

Systemd Services vs PHP-FPM

PHP-FPM + nginx:

```
# /etc/php/8.2/fpm/pool.d/www.conf
[www]
user = www-data
pm = dynamic
pm.max_children = 50
pm.start_servers = 5
```

```
# /etc/nginx/sites-available/myapp
server {
    listen 80;
    root /var/www/myapp/public;
    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
    }
}
```

Go systemd service:

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Go Application
After=network.target

[Service]
Type=simple
User=myapp
ExecStart=/opt/myapp/myapp
Restart=always
RestartSec=5
Environment=PORT=8080
Environment=DATABASE_URL=postgres://...

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable myapp
sudo systemctl start myapp
```

```
sudo journalctl -u myapp -f  # View logs
```

Advantages

- No nginx needed (Go handles HTTP directly)
- No PHP-FPM process management
- Automatic restart on crash
- Simple log management via journald
- Single process to monitor

Summary

- **Single binary** simplifies deployment dramatically
- **Cross-compilation** builds for any platform from any platform
- **Multi-stage Docker** creates tiny production images
- **No runtime** means no dependency management in production
- **Systemd** manages Go services simply and reliably

Exercises

1. **Build Optimisation**: Build the same application with and without `-ldflags="-s -w"`. Compare sizes.
2. **Version Embedding**: Add version, commit, and build date to a binary using ldflags.
3. **Cross-Compile**: Build a binary for 3 different OS/arch combinations from your machine.
4. **Docker Multi-Stage**: Write a multi-stage Dockerfile. Compare image sizes with single-stage.
5. **Scratch Image**: Create a Docker image from scratch. Verify it runs and what's missing.
6. **File Embedding**: Embed static files and templates. Deploy as a single binary.
7. **Systemd Service**: Write a systemd unit file for a Go application. Test restart behaviour.
8. **CI/CD Pipeline**: Create a GitHub Actions workflow that builds for multiple platforms and creates releases.

Chapter 24: Observability

PHP logging typically uses Monolog with various handlers. Go's observability stack is different—structured logging, Prometheus metrics, and OpenTelemetry tracing form the modern approach.

Structured Logging (slog vs Monolog)

Monolog:

```php
$logger->info('User created', [
    'user_id' => $user->getId(),
    'email' => $user->getEmail(),
]);
// Output depends on handler (JSON, line format, etc.)
```

Go's `log/slog` (Go 1.21+):

```go
import "log/slog"

func main() {
    // JSON output
    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    slog.SetDefault(logger)

    slog.Info("User created",
        "user_id", user.ID,
        "email", user.Email,
    )
}

// Output:
// {"time":"2024-01-15T10:30:00Z","level":"INFO","msg":"User created","user_id":123,"email":"alice@ex
```

Log Levels

```go
slog.Debug("Detailed info", "key", "value")
slog.Info("Normal operation", "key", "value")
slog.Warn("Something unusual", "key", "value")
slog.Error("Something failed", "error", err, "key", "value")
```

Contextual Logging

```
// Add context that applies to all subsequent logs
logger := slog.With("request_id", requestID, "user_id", userID)
logger.Info("Processing request")
logger.Info("Request completed")
```

Handler Configuration

```
// JSON with custom options
handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
    Level: slog.LevelDebug,
    AddSource: true,  // Include file:line
})

// Text for development
handler := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{
    Level: slog.LevelDebug,
})
```

Request Logging Middleware

```
func loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        requestID := uuid.New().String()

        // Add to context for downstream use
        ctx := context.WithValue(r.Context(), "request_id", requestID)
        r = r.WithContext(ctx)

        // Wrap response writer to capture status
        lrw := &loggingResponseWriter{ResponseWriter: w, status: 200}

        next.ServeHTTP(lrw, r)

        slog.Info("HTTP request",
            "method", r.Method,
            "path", r.URL.Path,
            "status", lrw.status,
            "duration", time.Since(start),
            "request_id", requestID,
        )
    })
}
```

Metrics with Prometheus

PHP metrics might use StatsD or custom solutions. Go typically uses Prometheus.

Setup

```
import (
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
    httpRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "http_requests_total",
            Help: "Total number of HTTP requests",
        },
        []string{"method", "path", "status"},
    )

    httpRequestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name:    "http_request_duration_seconds",
            Help:    "HTTP request duration in seconds",
            Buckets: prometheus.DefBuckets,
        },
        []string{"method", "path"},
    )
)

func init() {
    prometheus.MustRegister(httpRequestsTotal)
    prometheus.MustRegister(httpRequestDuration)
}

func main() {
    http.Handle("/metrics", promhttp.Handler())
    http.ListenAndServe(":8080", nil)
}
```

Metrics Middleware

```
func metricsMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        lrw := &loggingResponseWriter{ResponseWriter: w, status: 200}

        next.ServeHTTP(lrw, r)

        duration := time.Since(start).Seconds()
        status := strconv.Itoa(lrw.status)

        httpRequestsTotal.WithLabelValues(r.Method, r.URL.Path, status).Inc()
        httpRequestDuration.WithLabelValues(r.Method, r.URL.Path).Observe(duration)
    })
```

```
}
```

Metric Types

```
// Counter: Only goes up
requestCount := prometheus.NewCounter(prometheus.CounterOpts{
    Name: "requests_total",
})
requestCount.Inc()

// Gauge: Can go up or down
activeConnections := prometheus.NewGauge(prometheus.GaugeOpts{
    Name: "active_connections",
})
activeConnections.Inc()
activeConnections.Dec()

// Histogram: Distribution of values
requestDuration := prometheus.NewHistogram(prometheus.HistogramOpts{
    Name:    "request_duration_seconds",
    Buckets: []float64{.001, .005, .01, .05, .1, .5, 1},
})
requestDuration.Observe(0.042)

// Summary: Similar to histogram with percentiles
requestLatency := prometheus.NewSummary(prometheus.SummaryOpts{
    Name:       "request_latency_seconds",
    Objectives: map[float64]float64{0.5: 0.05, 0.9: 0.01, 0.99: 0.001},
})
```

Tracing with OpenTelemetry

Distributed tracing tracks requests across services.

Setup

```
import (
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracehttp"
    "go.opentelemetry.io/otel/sdk/trace"
)

func initTracer() (*trace.TracerProvider, error) {
    exporter, err := otlptracehttp.New(context.Background())
    if err != nil {
        return nil, err
    }

    tp := trace.NewTracerProvider(
```

```
        trace.WithBatcher(exporter),
        trace.WithResource(resource.NewWithAttributes(
            semconv.SchemaURL,
            semconv.ServiceNameKey.String("myapp"),
        )),
    )
    otel.SetTracerProvider(tp)
    return tp, nil
}
```

Creating Spans

```
var tracer = otel.Tracer("myapp")

func handleRequest(ctx context.Context) error {
    ctx, span := tracer.Start(ctx, "handleRequest")
    defer span.End()

    // Add attributes
    span.SetAttributes(
        attribute.String("user_id", userID),
        attribute.Int("item_count", len(items)),
    )

    // Call other services (context propagates trace)
    if err := callDatabase(ctx); err != nil {
        span.RecordError(err)
        span.SetStatus(codes.Error, err.Error())
        return err
    }

    return nil
}
```

HTTP Instrumentation

```
import "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"

handler := otelhttp.NewHandler(mux, "server")
http.ListenAndServe(":8080", handler)
```

Health Checks

Kubernetes and load balancers need health endpoints:

```
func healthHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
```

```
}

func readinessHandler(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ctx, cancel := context.WithTimeout(r.Context(), 5*time.Second)
        defer cancel()

        if err := db.PingContext(ctx); err != nil {
            http.Error(w, "Database unavailable", http.StatusServiceUnavailable)
            return
        }

        w.WriteHeader(http.StatusOK)
        w.Write([]byte("Ready"))
    }
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/health", healthHandler)       // Liveness
    mux.HandleFunc("/ready", readinessHandler(db)) // Readiness
}
```

Health Check Best Practices

- **Liveness** (`/health`): Is the process running? Keep simple.
- **Readiness** (`/ready`): Can the process handle traffic? Check dependencies.
- **Startup** (`/startup`): Has the process finished initialising?

Error Tracking (Sentry Integration)

```
import "github.com/getsentry/sentry-go"

func init() {
    sentry.Init(sentry.ClientOptions{
        Dsn:         os.Getenv("SENTRY_DSN"),
        Environment: os.Getenv("ENV"),
        Release:     version,
    })
}

func handleError(err error) {
    sentry.CaptureException(err)
}

// HTTP middleware
func sentryMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
```

```
                sentry.CurrentHub().Recover(err)
                http.Error(w, "Internal error", http.StatusInternalServerError)
            }
        }()
        next.ServeHTTP(w, r)
    })
}
```

Summary

- **Structured logging** with `slog` outputs JSON for log aggregation
- **Prometheus metrics** expose counters, gauges, and histograms
- **OpenTelemetry** provides distributed tracing
- **Health checks** enable container orchestration
- **Error tracking** with Sentry captures exceptions

Exercises

1. **Structured Logging**: Replace `fmt.Println` with `slog` throughout an application.
2. **Request ID Propagation**: Add request ID to all logs within a request lifecycle.
3. **Prometheus Metrics**: Add request count and duration metrics. View in Prometheus.
4. **Custom Metrics**: Create business metrics (orders placed, users registered, etc.).
5. **OpenTelemetry Setup**: Add tracing to a multi-service application. View traces in Jaeger.
6. **Health Checks**: Implement health, readiness, and startup probes with dependency checks.
7. **Sentry Integration**: Set up Sentry. Trigger errors and verify they appear in Sentry.
8. **Observability Dashboard**: Create a Grafana dashboard showing logs, metrics, and traces together.

Chapter 25: Migration Strategies

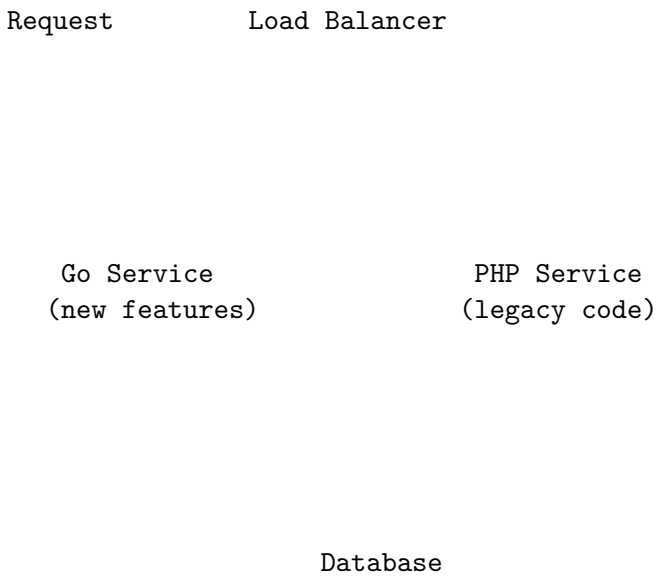
Migrating from PHP to Go isn't typically a big-bang rewrite. This chapter covers practical strategies for gradual migration, drawing on patterns used successfully in production.

Strangler Fig Pattern

The strangler fig tree grows around its host, eventually replacing it. Apply this to your PHP application:

1. **New features in Go**: Build new functionality in Go
2. **Route requests**: Proxy to PHP or Go based on path
3. **Migrate incrementally**: Move existing features one by one
4. **Remove PHP**: When all features migrated, retire PHP

Implementation



Routing at the Load Balancer

nginx:

```
upstream go_service {
    server go-app:8080;
}
```

```
upstream php_service {
    server php-fpm:9000;
}

server {
    # New API endpoints → Go
    location /api/v2/ {
        proxy_pass http://go_service;
    }

    # Legacy endpoints → PHP
    location / {
        fastcgi_pass php_service;
    }
}
```

Running PHP and Go Side-by-Side

Shared Authentication

Both services need to validate the same sessions/tokens:

```go
// Go: Validate PHP session
func validatePHPSession(sessionID string) (*User, error) {
    // Option 1: Shared Redis session store
    data, err := redis.Get("PHPREDIS_SESSION:" + sessionID).Result()

    // Option 2: Call PHP validation endpoint
    resp, err := http.Get(phpURL + "/api/validate-session?id=" + sessionID)

    // Option 3: Shared JWT with same secret
    claims, err := jwt.Parse(token, func(t *jwt.Token) (interface{}, error) {
        return []byte(sharedSecret), nil
    })
}
```

Session Sharing via Redis

PHP:

```
// php.ini
session.save_handler = redis
session.save_path = "tcp://redis:6379"
```

Go:

```go
import "github.com/go-redis/redis/v8"

func getSessionUser(sessionID string) (*User, error) {
```

```
    data, err := redisClient.Get(ctx, "PHPREDIS_SESSION:"+sessionID).Bytes()
    if err != nil {
        return nil, err
    }

    // PHP serialises sessions-use a PHP deserialiser or standardise on JSON
    var session map[string]interface{}
    // Decode PHP serialisation or JSON
    return extractUser(session)
}
```

JWT Sharing

PHP:

```
use Firebase\JWT\JWT;

$token = JWT::encode([
    'user_id' => $user->getId(),
    'exp' => time() + 3600,
], getenv('JWT_SECRET'), 'HS256');
```

Go:

```
import "github.com/golang-jwt/jwt/v5"

func validateToken(tokenString string) (*Claims, error) {
    token, err := jwt.ParseWithClaims(tokenString, &Claims{}, func(t *jwt.Token) (interface{}, error)
        return []byte(os.Getenv("JWT_SECRET")), nil
    })
    if err != nil {
        return nil, err
    }
    return token.Claims.(*Claims), nil
}
```

API Gateway Approaches

A dedicated API gateway can route between PHP and Go:

```
Client → API Gateway → PHP Service
                     → Go Service
                     → Other Services
```

Using Kong or Similar

```yaml
# Kong declarative config
services:
  - name: go-users
    url: http://go-service:8080
    routes:
      - paths: ["/api/v2/users"]

  - name: php-legacy
    url: http://php-service
    routes:
      - paths: ["/api/v1/", "/legacy/"]
```

Go as the Gateway

Go can serve as the gateway itself:

```go
func main() {
    goHandler := newGoHandler()
    phpProxy := newReverseProxy("http://php-service")

    mux := http.NewServeMux()

    // Go handles new API
    mux.Handle("/api/v2/", goHandler)

    // Proxy everything else to PHP
    mux.Handle("/", phpProxy)

    http.ListenAndServe(":8080", mux)
}

func newReverseProxy(target string) *httputil.ReverseProxy {
    url, _ := url.Parse(target)
    return httputil.NewSingleHostReverseProxy(url)
}
```

Database Sharing Strategies

Both services typically share a database during migration.

Shared Read, Separate Write

- PHP writes to its tables
- Go writes to its tables
- Both read from all tables
- Clear ownership prevents conflicts

Event-Driven Sync

```
PHP writes → Database → Triggers → Events → Go consumes
Go writes → Database → Triggers → Events → PHP consumes
```

Using Debezium or similar for change data capture:

```go
// Go: Consume database changes
func consumeChanges(ch <-chan ChangeEvent) {
    for event := range ch {
        switch event.Table {
        case "users":
            syncUser(event)
        case "orders":
            syncOrder(event)
        }
    }
}
```

Eventual Consistency

Accept that data might be briefly inconsistent:

```go
// Go service caches PHP data
func getUser(id int) (*User, error) {
    // Try cache first
    if user, ok := cache.Get(id); ok {
        return user, nil
    }

    // Call PHP API for authoritative data
    user, err := phpClient.GetUser(id)
    if err != nil {
        return nil, err
    }

    // Cache for next time
    cache.Set(id, user, 5*time.Minute)
    return user, nil
}
```

Gradual Team Transition

Technical migration is only half the challenge. Team transition matters equally.

Training Path

1. **Go basics**: Syntax, types, control flow (1 week)
2. **Go idioms**: Error handling, interfaces, packages (2 weeks)
3. **Concurrency**: Goroutines, channels, patterns (2 weeks)
4. **Production code**: Review and contribute to Go services

5. **Lead features**: Own a feature from design to deployment

Pairing and Review

- Pair PHP developers with Go-experienced developers
- Review all Go code from PHP developers carefully
- Discuss idiomatic alternatives, not just correctness

Start Small

First Go services should be: - Low-risk (non-critical path) - Well-defined scope - Good learning opportunities - Not time-sensitive

Case Study: Migrating a Symfony Application

Consider a typical Symfony app: - REST API for mobile apps - Admin dashboard (Twig templates) - Background workers (Messenger) - Doctrine ORM entities

Migration Plan

Phase 1: New API Endpoints (Month 1-2) - Build `/api/v2/` in Go - Share authentication via JWT - Both services use same database - Load balancer routes by path

Phase 2: Migrate High-Traffic Endpoints (Month 3-4) - Identify top 20% of endpoints by traffic - Rewrite in Go - Run shadow traffic to verify - Cut over one at a time

Phase 3: Background Workers (Month 5) - Build Go workers consuming same queues - Run PHP and Go workers in parallel - Gradually increase Go worker count - Retire PHP workers

Phase 4: Admin Dashboard (Month 6-8) - Build new admin in Go (or modern frontend) - Or: Keep PHP for admin (acceptable for low-traffic)

Phase 5: Retire PHP (Month 9+) - All traffic to Go - Remove PHP infrastructure - Archive PHP code

Success Metrics

- **Latency**: P99 latency of Go vs PHP endpoints
- **Throughput**: Requests per second per container
- **Resource usage**: Memory and CPU per request
- **Error rate**: Compare error rates during transition
- **Developer productivity**: Time to ship features

Summary

- **Strangler fig** enables gradual migration without big-bang rewrites
- **Side-by-side operation** requires shared auth and careful routing
- **API gateways** simplify routing between services
- **Database sharing** is common during transition
- **Team transition** requires training, pairing, and patience

- **Start small** with low-risk, well-scoped services

Exercises

1. **Strangler Design**: Draw a migration architecture for a Symfony app with 10 controllers.
2. **Reverse Proxy**: Implement a Go reverse proxy that routes to PHP for legacy paths.
3. **JWT Sharing**: Create matching JWT generation in PHP and validation in Go.
4. **Session Sharing**: Set up Redis session sharing between PHP and Go.
5. **Database Migration**: Design a strategy for migrating a Doctrine entity to Go without downtime.
6. **Traffic Shadowing**: Implement shadow traffic to compare PHP and Go responses.
7. **Migration Checklist**: Create a checklist for migrating a single endpoint from PHP to Go.
8. **Team Training Plan**: Design a 3-month training plan for transitioning PHP developers to Go.

Appendix A: PHP-to-Go Phrasebook

Quick reference for common PHP/Symfony patterns and their Go equivalents.

Language Basics

PHP	Go
`$variable = 42;`	`variable := 42`
`$arr = [];`	`arr := []T{}` or `arr := make([]T, 0)`
`$map = [];`	`m := make(map[K]V)`
`function name($a) {}`	`func name(a T) {}`
`public function`	`func (r *Receiver) Method()` (uppercase)
`private function`	`func (r *Receiver) method()` (lowercase)
`class Foo {}`	`type Foo struct {}`
`new Foo()`	`&Foo{}` or `NewFoo()`
`$this->property`	`r.property`
`$this->method()`	`r.Method()`
`null`	`nil`
`true/false`	`true/false`
`echo "text";`	`fmt.Println("text")`

Control Flow

PHP	Go
`if ($x) { }`	`if x { }`
`if ($x) { } else { }`	`if x { } else { }`
`elseif`	`else if`
`switch ($x) { case 1: break; }`	`switch x { case 1: }` (no break needed)
`for ($i = 0; $i < 10; $i++)`	`for i := 0; i < 10; i++`
`foreach ($arr as $v)`	`for _, v := range arr`
`foreach ($arr as $k => $v)`	`for k, v := range arr`
`while ($cond) { }`	`for cond { }`
`try { } catch { }`	`if err != nil { }`
`throw new Exception()`	`return errors.New()`

Types

PHP	Go
`int`	`int`, `int64`, `int32`
`float`	`float64`, `float32`
`string`	`string`
`bool`	`bool`
`array` (sequential)	`[]T` (slice)
`array` (associative)	`map[K]V`
`?string` (nullable)	`*string` or custom null type
`mixed`	`any` or `interface{}`
`object`	`struct`
`callable`	`func(args) returns`

String Operations

PHP	Go
`strlen($s)`	`len(s)` (bytes) or `utf8.RuneCountInString(s)`
`$s1 . $s2`	`s1 + s2` or `fmt.Sprintf("%s%s", s1, s2)`
`strpos($s, $sub)`	`strings.Index(s, sub)`
`substr($s, $start, $len)`	`s[start:start+len]`
`str_replace($old, $new, $s)`	`strings.Replace(s, old, new, -1)`
`explode(",", $s)`	`strings.Split(s, ",")`
`implode(",", $arr)`	`strings.Join(arr, ",")`
`trim($s)`	`strings.TrimSpace(s)`
`strtolower($s)`	`strings.ToLower(s)`
`sprintf("%s", $v)`	`fmt.Sprintf("%s", v)`

Array/Slice Operations

PHP	Go
`count($arr)`	`len(arr)`
`$arr[] = $v`	`arr = append(arr, v)`
`array_push($arr, $v)`	`arr = append(arr, v)`
`array_pop($arr)`	`arr = arr[:len(arr)-1]`
`array_slice($arr, $start, $len)`	`arr[start:start+len]`
`in_array($v, $arr)`	Loop or `slices.Contains(arr, v)`
`array_keys($arr)`	`maps.Keys(m)` (Go 1.21+)
`array_values($arr)`	`maps.Values(m)` (Go 1.21+)
`array_merge($a, $b)`	`slices.Concat(a, b)` (Go 1.22+)
`array_filter($arr, $fn)`	Loop with condition
`array_map($fn, $arr)`	Loop with transformation
`usort($arr, $fn)`	`slices.SortFunc(arr, fn)`

Error Handling

PHP	Go
`throw new Exception($msg)`	`return fmt.Errorf("msg: %w", err)`
`try { } catch (E $e) { }`	`if err != nil { }`
`$e->getMessage()`	`err.Error()`
`$e instanceof MyException`	`errors.Is(err, ErrMy)` or `errors.As(err, &myErr)`
Custom exception class	Custom error type implementing `error`
`finally { }`	`defer func() { }()`

Doctrine ORM → database/sql

Doctrine	Go
`$em->find(User::class, $id)`	`db.QueryRowContext(ctx, "SELECT...", id).Scan(&u)`
`$em->persist($entity)`	`db.ExecContext(ctx, "INSERT...", fields...)`
`$em->flush()`	Transactions: `tx.Commit()`
`$repo->findBy(['status' => $s])`	`db.QueryContext(ctx, "SELECT...WHERE status=$1", s)`
`$qb->select()...->getQuery()`	Raw SQL or squirrel query builder
`@Entity`	`type Entity struct {}`
`@Column`	Struct fields
`@OneToMany`	Separate queries or JOINs

Symfony HttpFoundation → net/http

Symfony	Go
`$request->query->get('key')`	`r.URL.Query().Get("key")`
`$request->request->get('key')`	`r.FormValue("key")`
`$request->getContent()`	`io.ReadAll(r.Body)`
`$request->headers->get('X-Foo')`	`r.Header.Get("X-Foo")`
`$request->getMethod()`	`r.Method`
`$request->getPathInfo()`	`r.URL.Path`
`new Response($body, 200)`	`w.WriteHeader(200); w.Write([]byte(body))`
`new JsonResponse($data)`	`json.NewEncoder(w).Encode(data)`
`$response->headers->set(...)`	`w.Header().Set(...)`

Symfony Services

Symfony	Go
`#[AsService]`	No equivalent—just a struct
Constructor injection	Pass dependencies to `New*` function
`#[Required]`	Constructor parameter
`services.yaml`	Explicit wiring in `main.go`
`$container->get(Foo::class)`	Direct instantiation
Interface binding	Accept interface in `New*` function

Testing

PHPUnit	Go testing
`class FooTest extends TestCase`	`func TestFoo(t *testing.T)`
`$this->assertEquals($a, $b)`	`if a != b { t.Errorf(...) }`
`$this->assertTrue($x)`	`if !x { t.Error(...) }`
`$this->expectException(E::class)`	Check returned error
`@dataProvider`	Table-driven tests
`$this->createMock(Foo::class)`	Manual mock or mockgen
`setUp()`	Code before test or `t.Cleanup()`
`tearDown()`	`t.Cleanup()` or `defer`

Common Patterns

Singleton (PHP) → Package Variable (Go)

```php
class Database {
    private static ?self $instance = null;
    public static function getInstance(): self {
        return self::$instance ??= new self();
    }
}
```

```go
var db *sql.DB
var once sync.Once

func GetDB() *sql.DB {
    once.Do(func() {
        db, _ = sql.Open("postgres", dsn)
    })
    return db
}
```

Factory (PHP) → New* Function (Go)

```php
class UserFactory {
    public function create(string $name): User {
        return new User($name, new DateTime());
    }
}
```

```go
func NewUser(name string) *User {
    return &User{Name: name, CreatedAt: time.Now()}
}
```

Builder (PHP) → Functional Options (Go)

```php
$user = (new UserBuilder())
    ->setName("Alice")
    ->setEmail("alice@example.com")
    ->build();
```

```go
type Option func(*User)

func WithEmail(e string) Option { return func(u *User) { u.Email = e } }

func NewUser(name string, opts ...Option) *User {
    u := &User{Name: name}
    for _, opt := range opts {
        opt(u)
    }
    return u
}

user := NewUser("Alice", WithEmail("alice@example.com"))
```

Repository (PHP) → Interface + Struct (Go)

```php
interface UserRepositoryInterface {
    public function find(int $id): ?User;
}

class DoctrineUserRepository implements UserRepositoryInterface {
    public function find(int $id): ?User { }
}
```

```go
type UserRepository interface {
    Find(ctx context.Context, id int) (*User, error)
}
```

```go
type PostgresUserRepository struct {
    db *sql.DB
}

func (r *PostgresUserRepository) Find(ctx context.Context, id int) (*User, error) {
    // Implementation
}
```

Appendix B: Standard Library Essentials

Key Go standard library packages with Symfony component comparisons.

net/http (HttpFoundation + HttpKernel)

```go
import "net/http"

// Server
http.HandleFunc("/", handler)
http.ListenAndServe(":8080", nil)

// Custom handler
type MyHandler struct{}
func (h *MyHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {}

// Request
r.Method           // GET, POST, etc.
r.URL.Path         // /users/123
r.URL.Query()      // Query parameters
r.Header           // Headers
r.Body             // Request body (io.ReadCloser)
r.Context()        // Request context
r.FormValue("key") // Form value

// Response
w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusOK)
w.Write([]byte("body"))

// Client
client := &http.Client{Timeout: 10 * time.Second}
resp, err := client.Get("https://example.com")
req, _ := http.NewRequestWithContext(ctx, "POST", url, body)
```

encoding/json (Serializer)

```go
import "encoding/json"

// Struct tags
type User struct {
    ID        int       `json:"id"`
```

```
    Name      string    `json:"name"`
    Email     string    `json:"email,omitempty"`
    Password  string    `json:"-"`
    CreatedAt time.Time `json:"created_at"`
}

// Marshal (encode)
data, err := json.Marshal(user)
json.NewEncoder(w).Encode(user)

// Unmarshal (decode)
var user User
err := json.Unmarshal(data, &user)
err := json.NewDecoder(r.Body).Decode(&user)

// Raw JSON
var raw json.RawMessage
var generic map[string]interface{}
```

database/sql (Doctrine DBAL)

```
import (
    "database/sql"
    _ "github.com/lib/pq"
)

// Connect
db, err := sql.Open("postgres", dsn)
db.SetMaxOpenConns(25)
db.SetMaxIdleConns(25)

// Query multiple rows
rows, err := db.QueryContext(ctx, "SELECT id, name FROM users WHERE active = $1", true)
defer rows.Close()
for rows.Next() {
    var id int
    var name string
    rows.Scan(&id, &name)
}

// Query single row
var name string
err := db.QueryRowContext(ctx, "SELECT name FROM users WHERE id = $1", id).Scan(&name)
if err == sql.ErrNoRows { /* not found */ }

// Execute
result, err := db.ExecContext(ctx, "INSERT INTO users (name) VALUES ($1)", name)
id, _ := result.LastInsertId()
affected, _ := result.RowsAffected()
```

```
// Transactions
tx, err := db.BeginTx(ctx, nil)
defer tx.Rollback()
tx.ExecContext(ctx, "...")
tx.Commit()

// Prepared statements
stmt, err := db.PrepareContext(ctx, "SELECT * FROM users WHERE id = $1")
defer stmt.Close()
stmt.QueryRowContext(ctx, id)
```

html/template (Twig)

```
import "html/template"

// Parse template
t := template.Must(template.New("page").Parse(`
<!DOCTYPE html>
<html>
<body>
    <h1>{{.Title}}</h1>
    {{range .Items}}
        <p>{{.}}</p>
    {{end}}
    {{if .ShowFooter}}
        <footer>Footer</footer>
    {{end}}
</body>
</html>
`))

// Execute
t.Execute(w, map[string]interface{}{
    "Title":      "My Page",
    "Items":      []string{"a", "b", "c"},
    "ShowFooter": true,
})

// Custom functions
funcs := template.FuncMap{
    "upper": strings.ToUpper,
    "formatDate": func(t time.Time) string {
        return t.Format("2006-01-02")
    },
}
t := template.New("page").Funcs(funcs).Parse("{{.Name | upper}}")

// Parse files
t := template.Must(template.ParseFiles("base.html", "page.html"))
```

log/slog (Monolog)

```go
import "log/slog"

// Setup
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
slog.SetDefault(logger)

// Logging
slog.Debug("Debug message", "key", "value")
slog.Info("Info message", "user_id", 123)
slog.Warn("Warning", "err", err)
slog.Error("Error occurred", "error", err)

// With context
logger := slog.With("request_id", requestID)
logger.Info("Processing")

// Groups
slog.Info("Request",
    slog.Group("request",
        slog.String("method", r.Method),
        slog.String("path", r.URL.Path),
    ),
)

// Levels
handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
    Level: slog.LevelDebug,
})
```

context (Request-scoped data)

```go
import "context"

// Create contexts
ctx := context.Background()
ctx := context.TODO()

// With timeout
ctx, cancel := context.WithTimeout(parent, 5*time.Second)
defer cancel()

// With deadline
ctx, cancel := context.WithDeadline(parent, time.Now().Add(30*time.Second))

// With cancellation
ctx, cancel := context.WithCancel(parent)
cancel() // Cancel when done
```

```
// With value
ctx := context.WithValue(parent, "user_id", 123)
userID := ctx.Value("user_id").(int)

// Check cancellation
select {
case <-ctx.Done():
    return ctx.Err()
default:
    // Continue
}

// In functions
func doWork(ctx context.Context) error {
    if ctx.Err() != nil {
        return ctx.Err()
    }
    // Work...
}
```

time (DateTime)

```
import "time"

// Current time
now := time.Now()
now.UTC()

// Create time
t := time.Date(2024, time.January, 15, 10, 30, 0, 0, time.UTC)

// Parse
t, err := time.Parse("2006-01-02", "2024-01-15")
t, err := time.Parse(time.RFC3339, "2024-01-15T10:30:00Z")

// Format
s := t.Format("2006-01-02 15:04:05")
s := t.Format(time.RFC3339)

// Duration
d := 5 * time.Second
d := time.Hour
d := time.Since(start)
d := time.Until(deadline)

// Arithmetic
tomorrow := now.Add(24 * time.Hour)
yesterday := now.Add(-24 * time.Hour)
diff := t2.Sub(t1)
```

```
// Sleep
time.Sleep(time.Second)

// Ticker
ticker := time.NewTicker(time.Second)
for t := range ticker.C {
    // Every second
}
ticker.Stop()

// Timer
timer := time.NewTimer(5 * time.Second)
<-timer.C // After 5 seconds
```

sync (Concurrency primitives)

```
import "sync"

// Mutex
var mu sync.Mutex
mu.Lock()
defer mu.Unlock()

// RWMutex (read-write)
var rw sync.RWMutex
rw.RLock()  // Multiple readers OK
rw.RUnlock()
rw.Lock()   // Exclusive write
rw.Unlock()

// WaitGroup
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    // Work
}()
wg.Wait()

// Once (singleton)
var once sync.Once
once.Do(func() {
    // Runs exactly once
})

// Pool (object reuse)
var pool = sync.Pool{
    New: func() interface{} {
        return new(bytes.Buffer)
    },
```

```
}
buf := pool.Get().(*bytes.Buffer)
defer pool.Put(buf)

// Map (concurrent-safe)
var m sync.Map
m.Store("key", "value")
v, ok := m.Load("key")
m.Delete("key")
m.Range(func(k, v interface{}) bool {
    return true // Continue iteration
})
```

os (Environment, Files)

```
import "os"

// Environment
val := os.Getenv("KEY")
os.Setenv("KEY", "value")
os.Unsetenv("KEY")
os.Environ() // All env vars

// Files
f, err := os.Open("file.txt")       // Read
f, err := os.Create("file.txt")     // Write (create/truncate)
f, err := os.OpenFile("file.txt", os.O_APPEND|os.O_WRONLY, 0644)
defer f.Close()

data, err := os.ReadFile("file.txt")
err := os.WriteFile("file.txt", data, 0644)

// Directories
err := os.Mkdir("dir", 0755)
err := os.MkdirAll("path/to/dir", 0755)
err := os.Remove("file.txt")
err := os.RemoveAll("dir")
entries, err := os.ReadDir("dir")

// Process
os.Exit(1)
os.Getpid()
os.Args // Command line arguments
```

io (Readers/Writers)

```
import "io"

// Read
data, err := io.ReadAll(r)
n, err := io.Copy(dst, src)
n, err := io.CopyN(dst, src, 100)

// Limited read
lr := io.LimitReader(r, 1024)

// Multi-reader
r := io.MultiReader(r1, r2, r3)

// Multi-writer
w := io.MultiWriter(w1, w2)

// Pipe
pr, pw := io.Pipe()
go func() {
    pw.Write(data)
    pw.Close()
}()
io.ReadAll(pr)

// NopCloser (add Close() to Reader)
rc := io.NopCloser(r)
```

fmt (Formatting)

```
import "fmt"

// Print
fmt.Print("no newline")
fmt.Println("with newline")
fmt.Printf("formatted: %s %d\n", s, n)

// Sprint (return string)
s := fmt.Sprint(value)
s := fmt.Sprintf("format: %v", value)

// Fprint (write to io.Writer)
fmt.Fprint(w, "to writer")
fmt.Fprintf(w, "format: %v", value)

// Scan (read input)
var s string
var n int
fmt.Scan(&s, &n)
```

```
fmt.Scanf("%s %d", &s, &n)

// Format verbs
%v   // Default format
%+v  // With field names (structs)
%#v  // Go syntax
%T   // Type
%s   // String
%d   // Integer
%f   // Float
%t   // Boolean
%p   // Pointer
%w   // Error wrapping (Errorf only)
```

Appendix C: Common Pitfalls

Mistakes PHP developers commonly make when learning Go.

1. Forgetting to Handle Errors

Wrong:

```
result, _ := doSomething()  // Ignoring error!
```

Right:

```
result, err := doSomething()
if err != nil {
    return nil, fmt.Errorf("doing something: %w", err)
}
```

Why: Go doesn't have exceptions. Ignored errors cause silent failures.

---

2. Nil Pointer Dereference

Wrong:

```
func getName(u *User) string {
    return u.Name  // Panics if u is nil!
}
```

Right:

```
func getName(u *User) string {
    if u == nil {
        return ""
    }
    return u.Name
}
```

Why: Unlike PHP's null-safe operator, Go panics on nil pointer access.

---

3. Modifying Slice While Iterating

Wrong:

```
for i, v := range items {
    if shouldRemove(v) {
        items = append(items[:i], items[i+1:]...)  // Dangerous!
    }
}
```

Right:

```
result := items[:0]
for _, v := range items {
    if !shouldRemove(v) {
        result = append(result, v)
    }
}
items = result
```

Why: Range iterates over a copy of the slice header; modifying during iteration causes skips or panics.

---

4. Goroutine Loop Variable Capture

Wrong:

```
for _, item := range items {
    go func() {
        process(item)  // All goroutines see the same (last) item!
    }()
}
```

Right (Go $<$ 1.22):

```
for _, item := range items {
    item := item  // Shadow the variable
    go func() {
        process(item)
    }()
}
```

Right (Go 1.22+):

```
for _, item := range items {
    go func() {
        process(item)  // Fixed in Go 1.22
    }()
}
```

Why: Before Go 1.22, the loop variable was reused; goroutines captured its address.

---

5. Using Defer in a Loop

Wrong:

```
for _, file := range files {
    f, _ := os.Open(file)
    defer f.Close()  // All files stay open until function returns!
}
```

Right:

```
for _, file := range files {
    func() {
        f, _ := os.Open(file)
        defer f.Close()
        // Process file
    }()
}
```

Why: Defer runs when the function returns, not when the loop iteration ends.

---

6. Expecting Maps to Be Ordered

Wrong:

```
m := map[string]int{"a": 1, "b": 2, "c": 3}
for k, v := range m {
    fmt.Println(k, v)  // Order is random!
}
```

Right:

```
keys := make([]string, 0, len(m))
for k := range m {
    keys = append(keys, k)
}
sort.Strings(keys)
for _, k := range keys {
    fmt.Println(k, m[k])
}
```

Why: Go maps are unordered by design. PHP arrays maintain insertion order.

---

7. Returning Interface When Concrete Would Work

Wrong:

```go
func NewService() ServiceInterface {
    return &service{}  // Loses concrete type info
}
```

Right:

```go
func NewService() *Service {
    return &Service{}  // Return concrete type
}
```

Why: Return concrete types; accept interfaces. Callers can store in interface variables if needed.

---

8. Forgetting that Strings Are Immutable

Wrong:

```go
s := "hello"
s[0] = 'H'  // Compile error!
```

Right:

```go
s := "hello"
b := []byte(s)
b[0] = 'H'
s = string(b)
```

Why: Go strings are immutable byte sequences. Use `[]byte` or `strings.Builder` for modification.

---

9. Not Understanding Zero Values

Surprise:

```go
var s string   // "" not nil
var n int      // 0
var b bool     // false
var slice []int // nil (but usable with append!)
var m map[string]int // nil (NOT usable-must make())
```

Right:

```go
m := make(map[string]int)  // Initialize before use
```

Why: Zero values are useful but nil maps panic on write. Nil slices are safe to append.

---

10. Comparing Slices Directly

Wrong:

```
if a == b {  // Compile error for slices!
}
```

Right:

```
if slices.Equal(a, b) {  // Go 1.21+
}
// Or manual comparison
```

Why: Slices are reference types; use `slices.Equal` or loop comparison.

---

11. Modifying a Map While Reading

Wrong (concurrent):

```
var m = make(map[string]int)

go func() {
    for k := range m {
        fmt.Println(k)
    }
}()

go func() {
    m["key"] = 1  // Race condition!
}()
```

Right:

```
var m sync.Map
// Or: protect with mutex
```

Why: Go maps are not concurrency-safe. Use `sync.Map` or mutex.

---

12. Assuming Printf Arguments Are Evaluated Lazily

Wrong:

```
slog.Debug("expensive", "data", computeExpensiveData())  // Always computed!
```

Right:

```
if slog.Default().Enabled(ctx, slog.LevelDebug) {
    slog.Debug("expensive", "data", computeExpensiveData())
}
```

Why: Go evaluates all arguments before the function call. Unlike PHP's short-circuit evaluation.

---

13. Forgetting Context Cancellation

Wrong:

```
ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
// Forgot cancel()! Resources leak.
```

Right:

```
ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
defer cancel()  // Always call cancel
```

Why: Cancel releases resources associated with the context.

---

14. Shadowing Variables Accidentally

Surprise:

```
err := doFirst()
if err != nil {
    return err
}

result, err := doSecond()  // This is the same err
if err != nil {
    return err
}

result, err := doThird()  // Compile error if err not used!
```

Watch for:

```
x := 1
if true {
    x := 2  // New x! Shadows outer x
}
fmt.Println(x)  // Still 1
```

Why: `:=` creates new variables; watch for unintentional shadowing.

---

15. Expecting Short-Circuit Evaluation in Custom Types

Wrong:

```
type MyBool bool

func (b MyBool) And(other MyBool) MyBool {
    return b && other  // Both sides always evaluated
}

a.And(expensiveOperation())  // Always runs!
```

Right:

```
if a && expensiveOperation() {  // Built-in && short-circuits
}
```

Why: Only built-in `&&` and `||` short-circuit. Method calls always evaluate arguments first.

---

16. Using Append Without Assigning

Wrong:

```
items := []int{1, 2, 3}
append(items, 4)  // Result discarded!
```

Right:

```
items = append(items, 4)  // Must assign
```

Why: `append` may return a new slice; always assign the result.

---

17. Passing Structs by Value When You Want Mutation

Wrong:

```
func updateUser(u User) {
    u.Name = "Updated"  // Modifies copy!
}
```

Right:

```go
func updateUser(u *User) {
    u.Name = "Updated"  // Modifies original
}
```

Why: Go passes by value. Structs are copied unless you use pointers.

---

18. Assuming HTTP Client Reuse

Wrong:

```go
func fetch(url string) {
    client := &http.Client{}  // New client each call!
    client.Get(url)
}
```

Right:

```go
var client = &http.Client{
    Timeout: 10 * time.Second,
}

func fetch(url string) {
    client.Get(url)  // Reuse client
}
```

Why: Creating clients is expensive; reuse them for connection pooling.

---

19. Not Closing HTTP Response Bodies

Wrong:

```go
resp, _ := http.Get(url)
body, _ := io.ReadAll(resp.Body)
// Body never closed! Connection leak.
```

Right:

```go
resp, err := http.Get(url)
if err != nil {
    return err
}
defer resp.Body.Close()
body, _ := io.ReadAll(resp.Body)
```

Why: Unclosed bodies prevent connection reuse and cause resource leaks.

---

20. Expecting JSON Numbers to Be int

Surprise:

```
var data map[string]interface{}
json.Unmarshal([]byte(`{"count": 42}`), &data)
count := data["count"].(int)  // Panic! It's float64
```

Right:

```
count := data["count"].(float64)
// Or use a typed struct
```

Why: JSON numbers unmarshal to `float64` by default in Go.

Appendix D: Symfony-to-Go Service Mapping

Detailed mappings from Symfony components to Go patterns.

HttpFoundation → net/http

Request Object

Symfony	Go
`$request->getMethod()`	`r.Method`
`$request->getPathInfo()`	`r.URL.Path`
`$request->getUri()`	`r.URL.String()`
`$request->getScheme()`	`r.URL.Scheme`
`$request->getHost()`	`r.Host`
`$request->query->get('key')`	`r.URL.Query().Get("key")`
`$request->query->all()`	`r.URL.Query()`
`$request->request->get('key')`	`r.FormValue("key")`
`$request->request->all()`	`r.PostForm` (after `r.ParseForm()`)
`$request->headers->get('X-Key')`	`r.Header.Get("X-Key")`
`$request->headers->all()`	`r.Header`
`$request->cookies->get('name')`	Loop `r.Cookies()` or `r.Cookie("name")`
`$request->getContent()`	`io.ReadAll(r.Body)`
`$request->toArray()`	`json.NewDecoder(r.Body).Decode(&v)`
`$request->getSession()`	Use session library (gorilla/sessions)
`$request->attributes->get()`	`r.Context().Value(key)`
`$request->getClientIp()`	Parse `r.Header.Get("X-Forwarded-For")` or `r.RemoteAddr`

Response Object

Symfony	Go
`new Response($body)`	`w.Write([]byte(body))`
`new Response($body, 201)`	`w.WriteHeader(201); w.Write(...)`
`$response->headers->set()`	`w.Header().Set(key, val)`

Symfony	Go
`new JsonResponse($data)`	`json.NewEncoder(w).Encode(data)`
`new RedirectResponse($url)`	`http.Redirect(w, r, url, http.StatusFound)`
`new BinaryFileResponse($path)`	`http.ServeFile(w, r, path)`
`$response->setStatusCode()`	`w.WriteHeader(code)`

Example: Full Handler

```php
// Symfony
#[Route('/users/{id}', methods: ['GET'])]
public function show(int $id, Request $request): Response
{
    $format = $request->query->get('format', 'json');
    $user = $this->userRepository->find($id);

    if (!$user) {
        throw $this->createNotFoundException();
    }

    return $this->json($user);
}
```

```go
// Go
func (h *UserHandler) Show(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil {
        http.Error(w, "invalid id", http.StatusBadRequest)
        return
    }

    format := r.URL.Query().Get("format")
    if format == "" {
        format = "json"
    }

    user, err := h.repo.Find(r.Context(), id)
    if errors.Is(err, ErrNotFound) {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    if err != nil {
        http.Error(w, "internal error", http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(user)
}
```

Serializer → encoding/json

Basic Serialisation

```php
// Symfony
$json = $serializer->serialize($user, 'json');
$user = $serializer->deserialize($json, User::class, 'json');
```

```go
// Go
data, err := json.Marshal(user)
err := json.Unmarshal(data, &user)
```

Serialisation Groups

```php
// Symfony
#[Groups(['public'])]
private string $email;

$json = $serializer->serialize($user, 'json', ['groups' => ['public']]);
```

```go
// Go: Use separate structs
type UserPublic struct {
    ID    int    `json:"id"`
    Name  string `json:"name"`
}

type UserPrivate struct {
    UserPublic
    Email string `json:"email"`
}

func (u User) ToPublic() UserPublic {
    return UserPublic{ID: u.ID, Name: u.Name}
}
```

Custom Normalisers

```php
// Symfony
class MoneyNormalizer implements NormalizerInterface
{
    public function normalize($object, $format = null, array $context = [])
    {
        return ['amount' => $object->getAmount() / 100, 'currency' => $object->getCurrency()];
    }
}
```

```go
// Go: Implement json.Marshaler
func (m Money) MarshalJSON() ([]byte, error) {
    return json.Marshal(map[string]interface{}{
        "amount":   float64(m.Amount) / 100,
        "currency": m.Currency,
    })
}
```

Validator → go-playground/validator

Constraints Mapping

Symfony	go-playground/validator
`#[NotBlank]`	`validate:"required"`
`#[NotNull]`	`validate:"required"`
`#[Email]`	`validate:"email"`
`#[Length(min: 2, max: 50)]`	`validate:"min=2,max=50"`
`#[Range(min: 1, max: 100)]`	`validate:"min=1,max=100"`
`#[Positive]`	`validate:"gt=0"`
`#[Regex(pattern: '/^\d+$/')]`	Custom validator
`#[Url]`	`validate:"url"`
`#[Uuid]`	`validate:"uuid"`
`#[Valid]`	`validate:"dive"` (for nested)

Example

```php
// Symfony
class CreateUserInput
{
    #[NotBlank]
    #[Length(min: 2, max: 100)]
    public string $name;

    #[NotBlank]
    #[Email]
    public string $email;

    #[NotBlank]
    #[Length(min: 8)]
    public string $password;
}
```

```go
// Go
type CreateUserInput struct {
    Name     string `json:"name" validate:"required,min=2,max=100"`
    Email    string `json:"email" validate:"required,email"`
    Password string `json:"password" validate:"required,min=8"`
```

```
}

var validate = validator.New()

func (input CreateUserInput) Validate() error {
    return validate.Struct(input)
}
```

Security → Middleware Patterns

Authentication

```
// Symfony Security
#[IsGranted('ROLE_USER')]
public function profile(): Response
{
    $user = $this->getUser();
}
```

```
// Go: Middleware
func authRequired(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        user := getUserFromContext(r.Context())
        if user == nil {
            http.Error(w, "unauthorized", http.StatusUnauthorized)
            return
        }
        next.ServeHTTP(w, r)
    })
}

// Apply to routes
mux.Handle("/profile", authRequired(http.HandlerFunc(profileHandler)))
```

Voters

```
// Symfony Voter
class PostVoter extends Voter
{
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token): bool
    {
        $user = $token->getUser();
        return $subject->getAuthor() === $user;
    }
}
```

```go
// Go: Check in handler or middleware
func canEditPost(user *User, post *Post) bool {
    return post.AuthorID == user.ID
}

func editPost(w http.ResponseWriter, r *http.Request) {
    user := getUserFromContext(r.Context())
    post := getPost(r)

    if !canEditPost(user, post) {
        http.Error(w, "forbidden", http.StatusForbidden)
        return
    }
    // Edit post
}
```

Messenger → Channels and Workers

Message Dispatching

```php
// Symfony Messenger
$this->messageBus->dispatch(new OrderCreatedEvent($order));
```

```go
// Go: Channel
type OrderCreatedEvent struct {
    OrderID int
}

var events = make(chan OrderCreatedEvent, 100)

// Dispatch
events <- OrderCreatedEvent{OrderID: order.ID}

// Worker
go func() {
    for event := range events {
        handleOrderCreated(event)
    }
}()
```

Message Handlers

```php
// Symfony
#[AsMessageHandler]
class OrderCreatedHandler
{
    public function __invoke(OrderCreatedEvent $event): void
    {
```

```
        // Handle
    }
}
```

```
// Go: Worker function
func orderCreatedWorker(ctx context.Context, events <-chan OrderCreatedEvent) {
    for {
        select {
        case <-ctx.Done():
            return
        case event := <-events:
            handleOrderCreated(event)
        }
    }
}
```

Cache → go-cache or Redis

Basic Caching

```
// Symfony Cache
$value = $cache->get('key', function (ItemInterface $item) {
    $item->expiresAfter(3600);
    return computeExpensiveValue();
});
```

```
// Go: go-cache
import "github.com/patrickmn/go-cache"

var c = cache.New(5*time.Minute, 10*time.Minute)

func getValue(key string) (interface{}, error) {
    if val, found := c.Get(key); found {
        return val, nil
    }

    val := computeExpensiveValue()
    c.Set(key, val, time.Hour)
    return val, nil
}
```

Redis Cache

```
// Go: Redis
import "github.com/go-redis/redis/v8"

var rdb = redis.NewClient(&redis.Options{Addr: "localhost:6379"})
```

```go
func getValue(ctx context.Context, key string) (string, error) {
    val, err := rdb.Get(ctx, key).Result()
    if err == redis.Nil {
        val = computeExpensiveValue()
        rdb.Set(ctx, key, val, time.Hour)
        return val, nil
    }
    return val, err
}
```

EventDispatcher → Callbacks or Channels

Event Dispatching

```php
// Symfony
$this->eventDispatcher->dispatch(new UserCreatedEvent($user));
```

```go
// Go: Callback pattern
type EventDispatcher struct {
    listeners map[string][]func(any)
}

func (d *EventDispatcher) Dispatch(name string, event any) {
    for _, listener := range d.listeners[name] {
        listener(event)
    }
}

func (d *EventDispatcher) AddListener(name string, fn func(any)) {
    d.listeners[name] = append(d.listeners[name], fn)
}

// Or: Channel-based
type UserCreatedEvent struct {
    User User
}

var userCreatedChan = make(chan UserCreatedEvent, 100)

// Dispatch
userCreatedChan <- UserCreatedEvent{User: user}

// Listen
go func() {
    for event := range userCreatedChan {
        sendWelcomeEmail(event.User)
    }
}()
```

Console → cobra or flag

Command Definition

```php
// Symfony Console
class ImportUsersCommand extends Command
{
    protected function configure(): void
    {
        $this->setName('app:import-users')
             ->addArgument('file', InputArgument::REQUIRED)
             ->addOption('dry-run', null, InputOption::VALUE_NONE);
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $file = $input->getArgument('file');
        $dryRun = $input->getOption('dry-run');
        // Import users
        return Command::SUCCESS;
    }
}
```

```go
// Go: cobra
var importCmd = &cobra.Command{
    Use:   "import-users [file]",
    Short: "Import users from file",
    Args:  cobra.ExactArgs(1),
    RunE: func(cmd *cobra.Command, args []string) error {
        file := args[0]
        dryRun, _ := cmd.Flags().GetBool("dry-run")

        // Import users
        return nil
    },
}

func init() {
    importCmd.Flags().Bool("dry-run", false, "Dry run mode")
    rootCmd.AddCommand(importCmd)
}
```

Appendix E: Recommended Reading

Resources for continued learning after this book.

Official Documentation

Go

- **The Go Programming Language Specification** https://go.dev/ref/spec The definitive language reference. Read when you need precise semantics.
- **Effective Go** https://go.dev/doc/effective_go Essential reading for writing idiomatic Go. Covers conventions, patterns, and style.
- **Go Code Review Comments** https://go.dev/wiki/CodeReviewComments Common code review feedback. Great for learning what experienced Go developers look for.
- **Go Blog** https://go.dev/blog/ Authoritative posts on new features, patterns, and internals.

Books

Essential

- **The Go Programming Language** by Alan A. A. Donovan and Brian W. Kernighan The comprehensive Go book. Co-authored by a Go team member and a Unix/C legend. Covers fundamentals through advanced topics.
- **Concurrency in Go** by Katherine Cox-Buday Deep dive into Go's concurrency model. Essential for understanding goroutines, channels, and patterns.
- **Learning Go** by Jon Bodner Modern introduction covering Go 1.18+ features including generics. Good for solidifying fundamentals.

Advanced

- **100 Go Mistakes and How to Avoid Them** by Teiva Harsanyi Comprehensive guide to common pitfalls. Excellent for intermediate developers.
- **Let's Go** and **Let's Go Further** by Alex Edwards Practical web development in Go. Build a complete application from scratch.
- **Writing an Interpreter in Go** and **Writing a Compiler in Go** by Thorsten Ball Learn language implementation through building. Great for deepening understanding.

Online Resources

Tutorials and Guides

- **Go by Example** https://gobyexample.com/ Hands-on examples for every Go concept. Excellent quick reference.
- **Go Web Examples** https://gowebexamples.com/ Web development patterns with complete examples.
- **Gophercises** https://gophercises.com/ Practical exercises for building Go skills.

Blogs

- **Dave Cheney's Blog** https://dave.cheney.net/ Deep technical posts on Go internals and best practices.
- **Eli Bendersky's Website** https://eli.thegreenplace.net/tag/go Detailed explanations of Go concepts and implementations.
- **Ardan Labs Blog** https://www.ardanlabs.com/blog/ Enterprise Go patterns and practices.
- **Applied Go** https://appliedgo.net/ Practical articles on applying Go to real problems.

Newsletters

- **Golang Weekly** https://golangweekly.com/ Weekly roundup of Go news, articles, and packages.

Video Resources

- **justforfunc** https://www.youtube.com/c/JustForFunc Video series by Francesc Campoy (former Go team). Excellent for seeing Go in action.
- **GopherCon Talks** https://www.youtube.com/c/GopherAcademy Conference talks from Go experts. Great for advanced topics.

Go Internals

- **Go Internals** (this author's companion book) Deep dive into compiler, runtime, and built-in types.
- **Go 101** https://go101.org/ Free online book covering Go details often missed.
- **research!rsc (Russ Cox's Blog)** https://research.swtch.com/ Posts from Go's tech lead on language design decisions.

Standard Library Deep Dives

- **net/http source code** https://github.com/golang/go/tree/master/src/net/http The best way to understand http is reading the source.

- **encoding/json source code** https://github.com/golang/go/tree/master/src/encoding/json Understanding json helps with custom marshalling.

Community

Forums and Discussion

- **r/golang** https://reddit.com/r/golang Active community for questions and discussion.
- **Gophers Slack** https://gophers.slack.com/ Real-time help from the community. Get an invite at https://invite.slack.golangbridge.org/
- **Go Forum** https://forum.golangbridge.org/ Official community forum.

Conferences

- **GopherCon** The main Go conference. Talks available online.
- **GopherCon EU** European Go conference.
- **GoLab** Italian Go conference with excellent talks.

PHP-to-Go Specific

Migration Case Studies

- Uber's Go adoption
- Cloudflare's Go at scale
- Various company engineering blogs documenting transitions

Comparison Articles

- "From PHP to Go" series on various blogs
- Stack Overflow discussions on PHP vs Go

Tools and Ecosystem

Must-Know Tools

- **golangci-lint**: Comprehensive linter
- **go-critic**: Additional checks
- **gopls**: Language server (IDE support)
- **dlv (Delve)**: Debugger

Useful Packages

- **github.com/spf13/cobra**: CLI applications
- **github.com/spf13/viper**: Configuration
- **github.com/go-chi/chi**: HTTP router
- **github.com/jmoiron/sqlx**: Database extensions

- **github.com/stretchr/testify**: Testing assertions
- **go.uber.org/zap**: High-performance logging

Keeping Up-to-Date

- **Go Release Notes** https://go.dev/doc/devel/release What's new in each release.
- **Go Proposals** https://github.com/golang/go/issues?q=is%3Aopen+is%3Aissue+label%3AProposal Future language changes under discussion.
- **golang/go Issues** https://github.com/golang/go/issues The source for understanding Go's evolution.

Practice Platforms

- **Exercism Go Track** https://exercism.org/tracks/go Mentored exercises with feedback.
- **LeetCode** https://leetcode.com/ Algorithm practice in Go.
- **Advent of Code** https://adventofcode.com/ Annual programming challenges. Great for Go practice.

Reading Path Recommendation

Week 1-2: Foundations

1. Go Tour (tour.golang.org)
2. Effective Go
3. Go by Example

Month 1: Deepening

1. The Go Programming Language (book)
2. Go Code Review Comments
3. This book's exercises

Month 2-3: Specialisation

1. Concurrency in Go (book)
2. Let's Go / Let's Go Further (web development)
3. Package documentation for your domain

Ongoing

1. Golang Weekly newsletter
2. GopherCon talks
3. Go Blog
4. Community participation (Slack, Reddit)