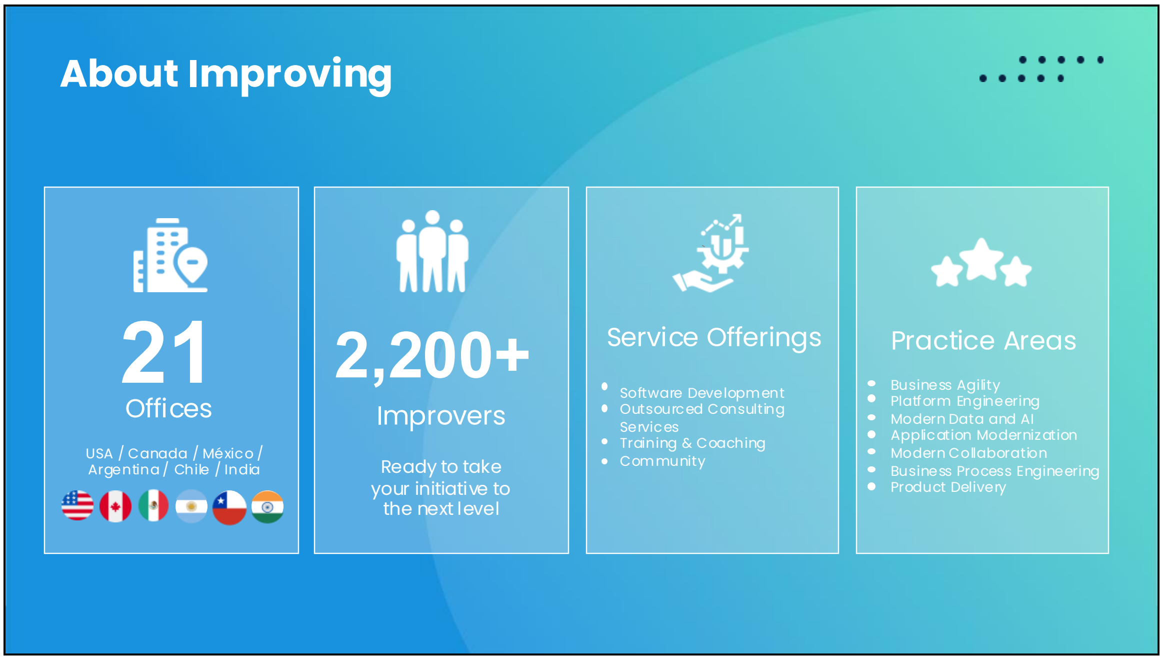
Golang for Java Developers



About Improving

21
Offices

USA / Canada / México / Argentina / Chile / India

2,200+
Improvers

Ready to take your initiative to the next level

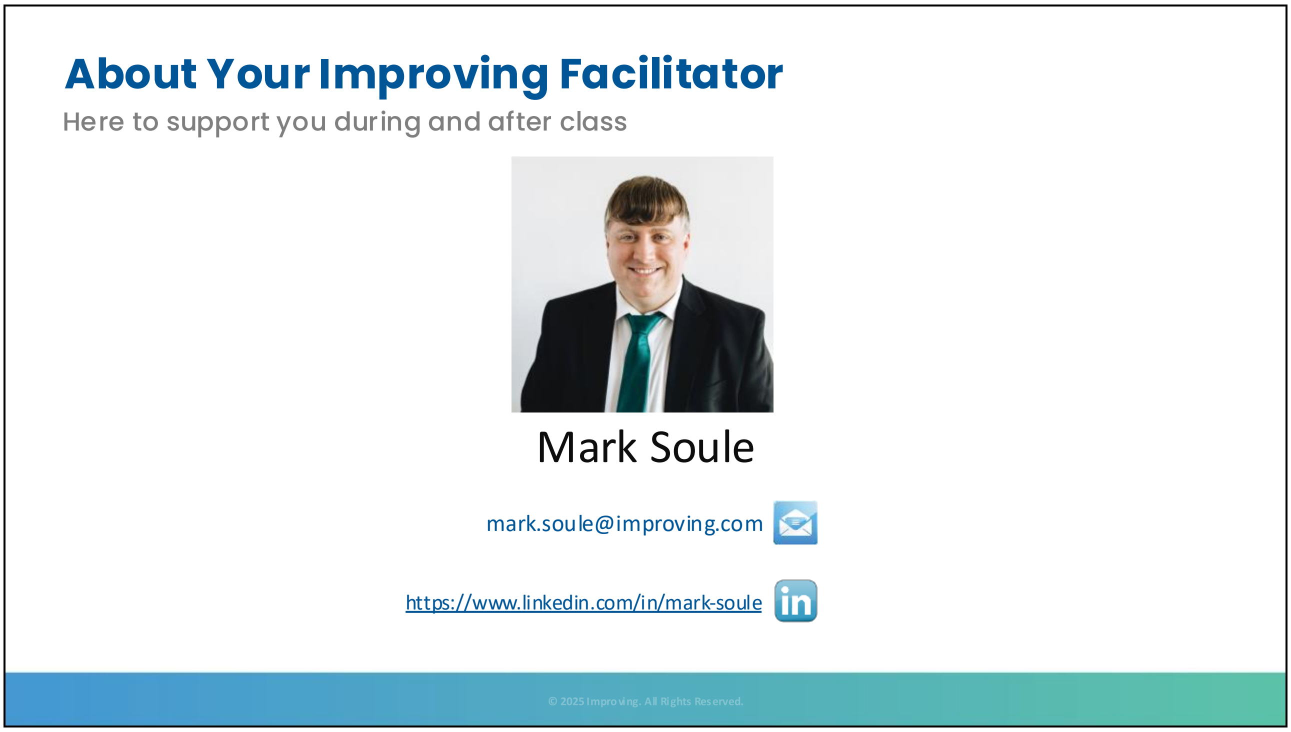
Service Offerings

- Software Development
- Outsourced Consulting Services
- Training & Coaching
- Community

Practice Areas

- Business Agility
- Platform Engineering
- Modern Data and AI
- Application Modernization
- Modern Collaboration
- Business Process Engineering
- Product Delivery

Introduction



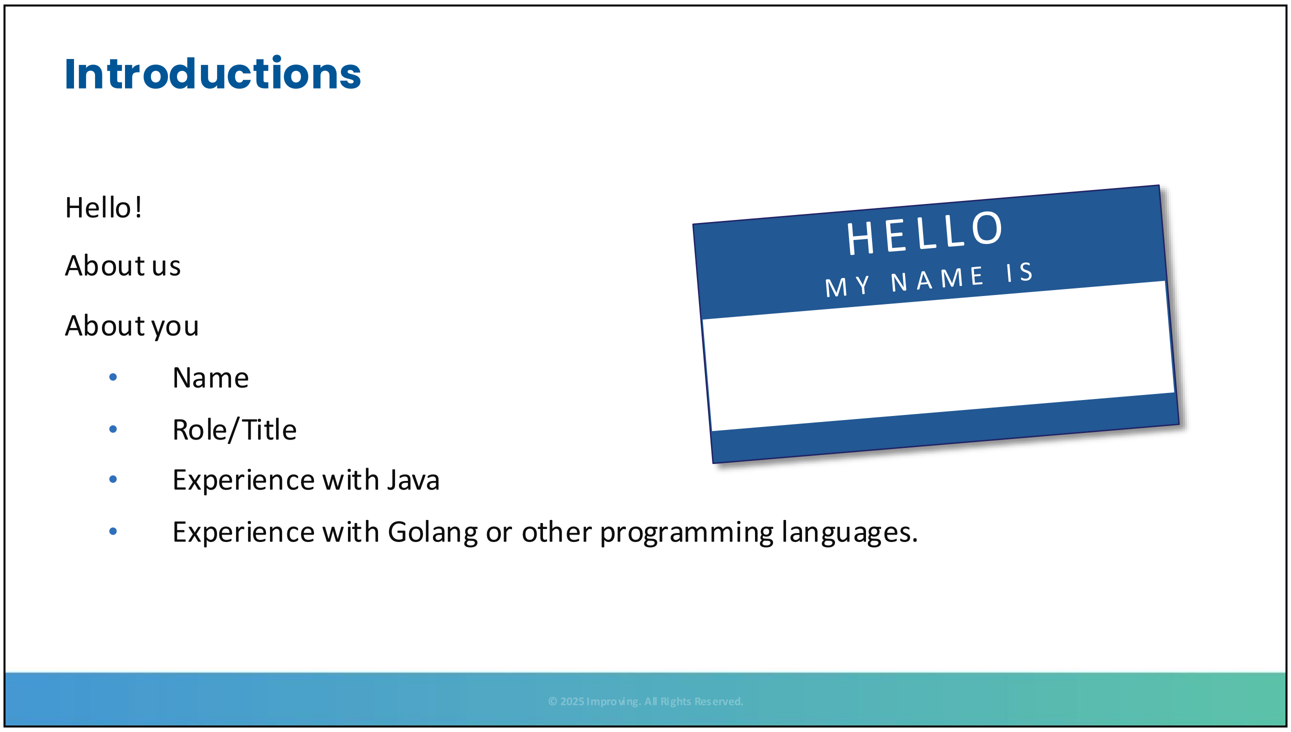
About Your Improving Facilitator

Here to support you during and after class

Mark Soule

mark.soule@improving.com

https://www.linkedin.com/in/mark-soule



Introductions

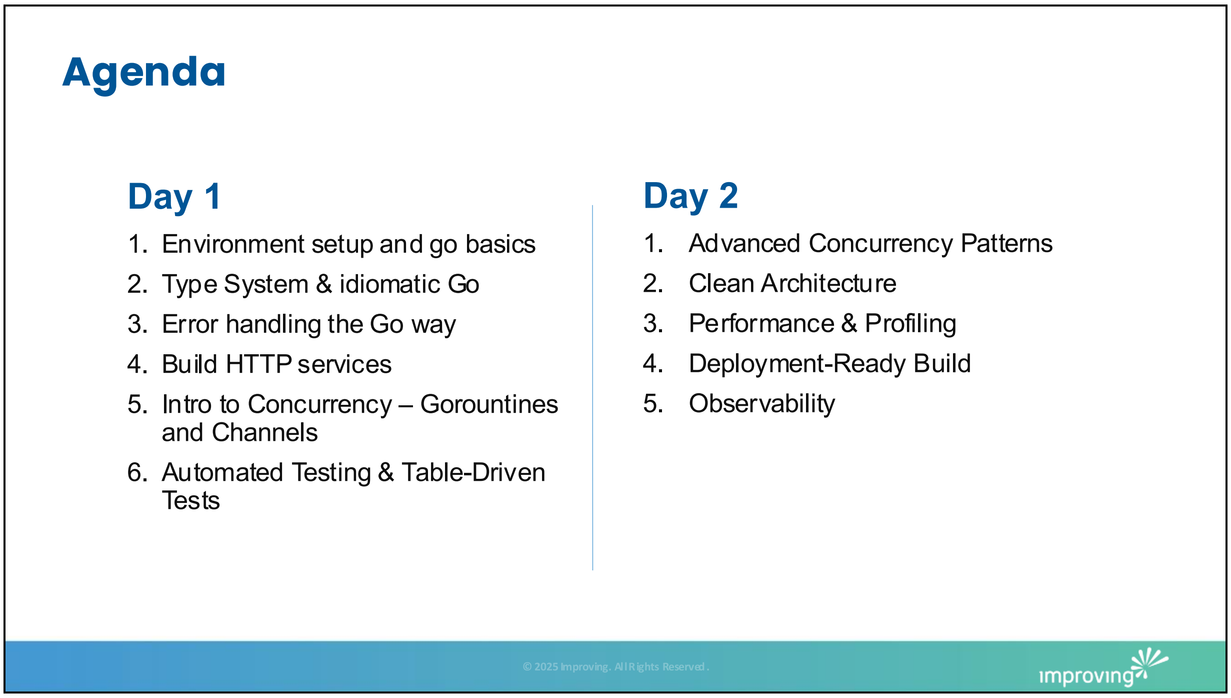
Hello!

About us

About you

- Name
- Role/Title
- Experience with Java
- Experience with Golang or other programming languages.

HELLO
MY NAME IS



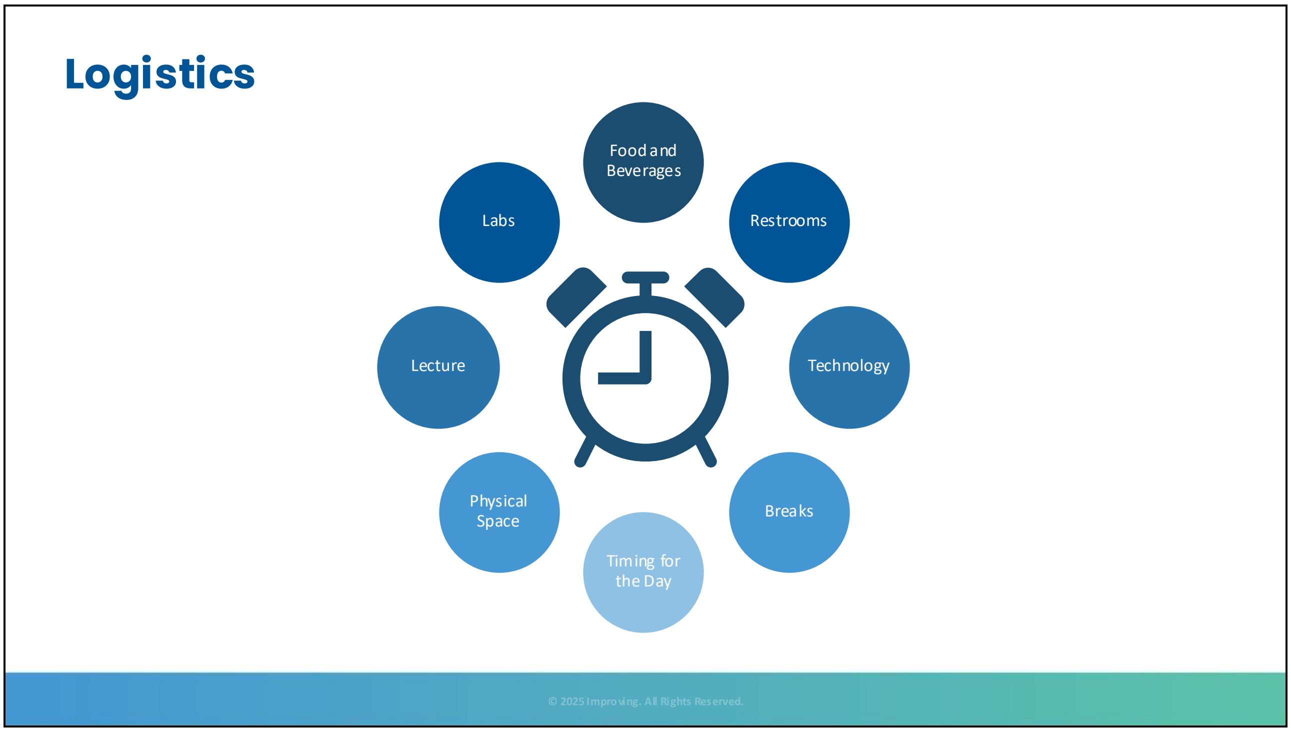
Agenda

Day 1

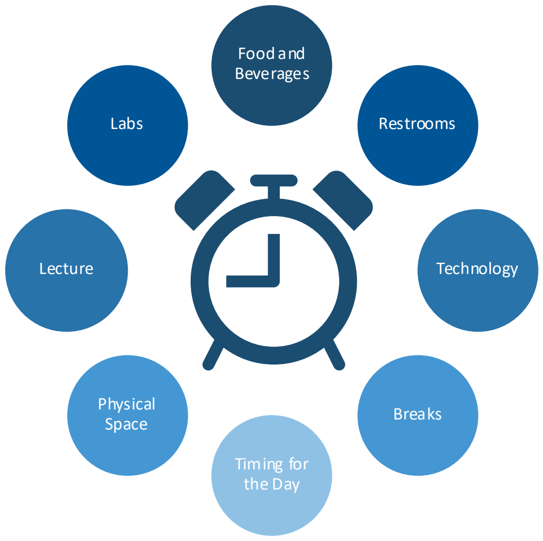
1. Environment setup and go basics
2. Type System & idiomatic Go
3. Error handling the Go way
4. Build HTTP services
5. Intro to Concurrency – Gorountines and Channels
6. Automated Testing & Table-Driven Tests

Day 2

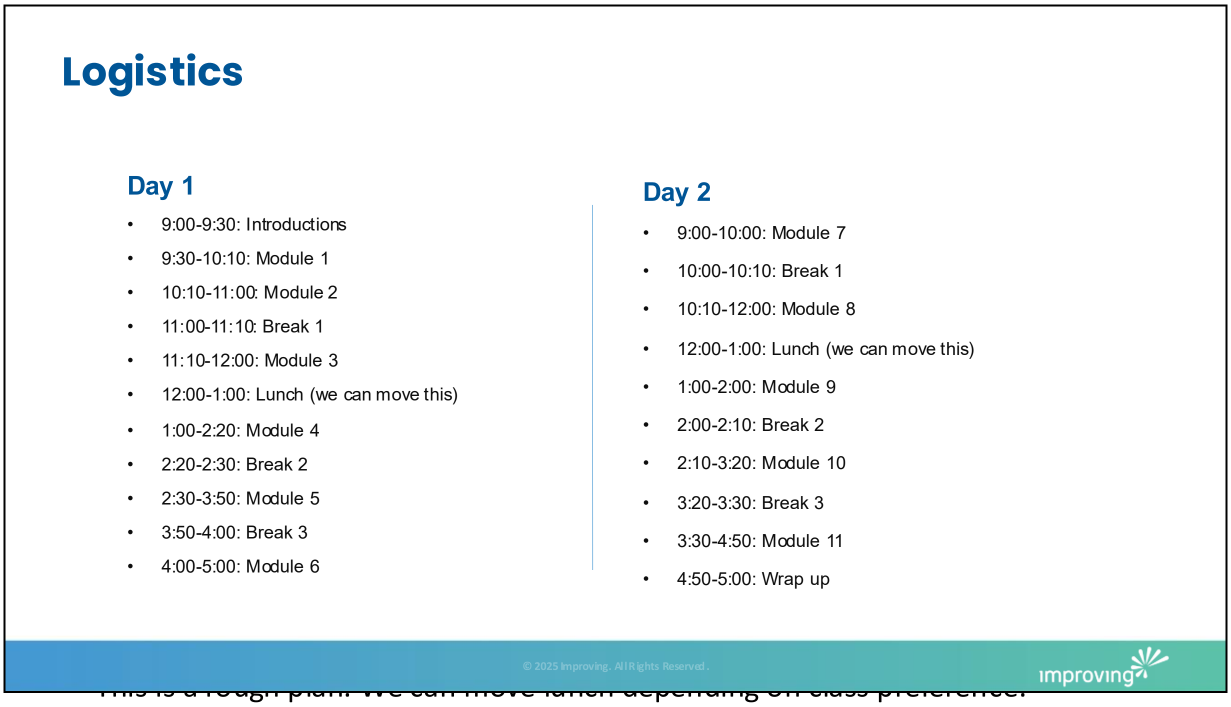
1. Advanced Concurrency Patterns
2. Clean Architecture
3. Performance & Profiling
4. Deployment-Ready Build
5. Observability



Logistics



What time is best for lunch?
Where are good places to eat?
When to start and end each day?
Where are the restrooms?
Are your phones/pagers on non-audible mode?
Materials?
Computers?
Breaks?



Logistics

Day 1

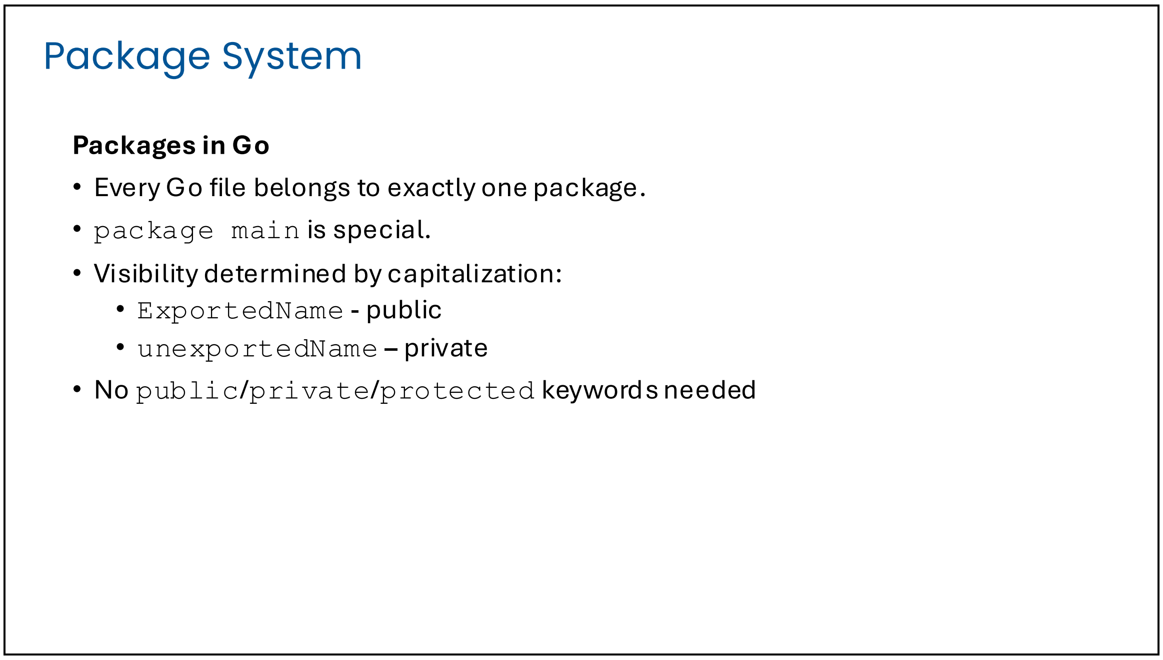
- 9:00-9:30: Introductions
- 9:30-10:10: Module 1
- 10:10-11:00: Module 2
- 11:00-11:10: Break 1
- 11:10-12:00: Module 3
- 12:00-1:00: Lunch (we can move this)
- 1:00-2:20: Module 4
- 2:20-2:30: Break 2
- 2:30-3:50: Module 5
- 3:50-4:00: Break 3
- 4:00-5:00: Module 6

Day 2

- 9:00-10:00: Module 7
- 10:00-10:10: Break 1
- 10:10-12:00: Module 8
- 12:00-1:00: Lunch (we can move this)
- 1:00-2:00: Module 9
- 2:00-2:10: Break 2
- 2:10-3:20: Module 10
- 3:20-3:30: Break 3
- 3:30-4:50: Module 11
- 4:50-5:00: Wrap up

- This is a rough plan. We can move lunch depending on class preference.
- Every module had the same structure: Intro, lab, debrief
- 2 breaks in afternoon because later labs are more difficult.
- If you finish early or otherwise leave during a lab, please return for debrief.
- I will let you know when we will debrief just before we kick off a lab.

Golang for Java Developers



Package System

Packages in Go

- Every Go file belongs to exactly one package.
- `package main` is special.
- Visibility determined by capitalization:
  - `ExportedName` - public
  - `unexportedName` – private
- No `public`/`private`/`protected` keywords needed

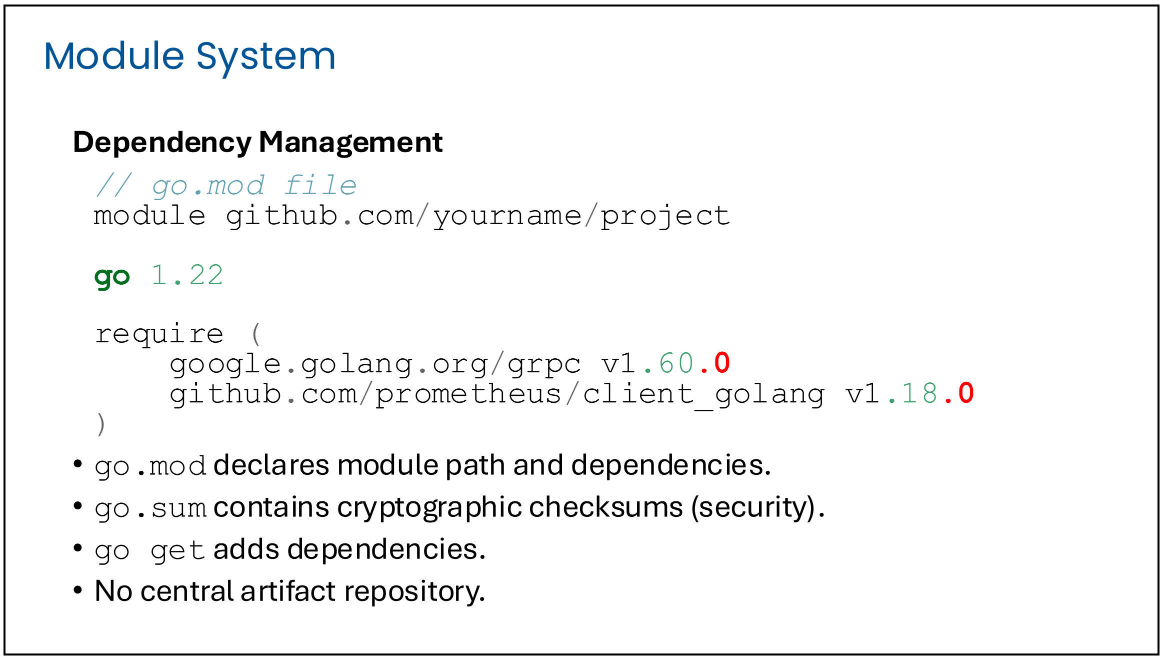
- Unlike Java's package system with access modifiers, Go uses a simple capitalization rule.
- This is enforced by the compiler - you cannot access unexported names from other packages.
- `package main` with `func main()` is the entry point, similar to Java's `public static void main`.
- Packages are directories - all .go files in same directory must have same package name.
- Import paths are relative to module path or GOPATH.
- Lab 1 introduces creating first module with package main.

Go Tooling

Essential Go Commands

- `go mod init` - Initialize new module.
- `go run` - Compile and run.
- `go build` - Compile binary.
- `go test` - Run tests.
- `go fmt` - Format code.
- `go vet` - Static analysis for common mistakes.
- `go mod tidy` - Clean up dependencies.

`go mod init` creates go.mod file, similar to Maven's pom.xml or Gradle build file.
`go run` is like `javac` + `java` in one command - useful for development.
`go build` produces platform-specific binary with no JVM required.
`go fmt` is non-negotiable - all Go code follows same format (no style debates!)
`go vet` catches issues like Printf format mismatches, unreachable code.
Unlike Java's verbose tooling, Go's tools are fast and built-in.



Module System

Dependency Management

```
// go.mod file
module github.com/yourname/project

go 1.22

require (
    google.golang.org/grpc v1.60.0
    github.com/prometheus/client_golang v1.18.0
)
```

- `go.mod` declares module path and dependencies.
- `go.sum` contains cryptographic checksums (security).
- `go get` adds dependencies.
- No central artifact repository.

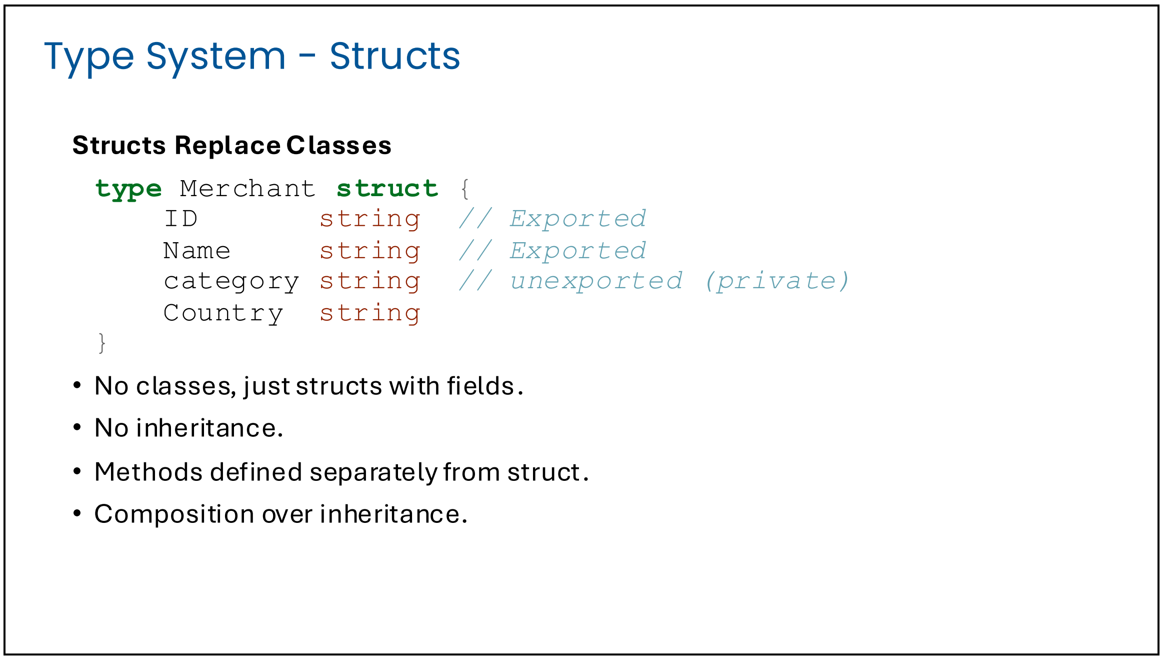
- Module path is typically your repository URL - makes code importable.
- Similar to Maven coordinates but simpler - just the import path.
- `go.sum` is like Maven's checksum verification but automatic.
- No Maven Central equivalent - can fetch from any Git repository.
- Versioning uses semantic import versioning (v2+ in import path)
- `go mod tidy` removes unused deps and adds missing ones automatically.
- Much faster than Maven/Gradle - downloads are cached in GOPATH/pkg/mod.

Lab 1

Give students the time to return for debrief.

Scalars

- Numeric types
  - int, int8, int16, int32, int64, uint, float32, float64, complex64, complex128
- Boolean
  - bool
- String
  - string
- Byte and rune
  - byte (alias for uint8), rune (alias for int32)



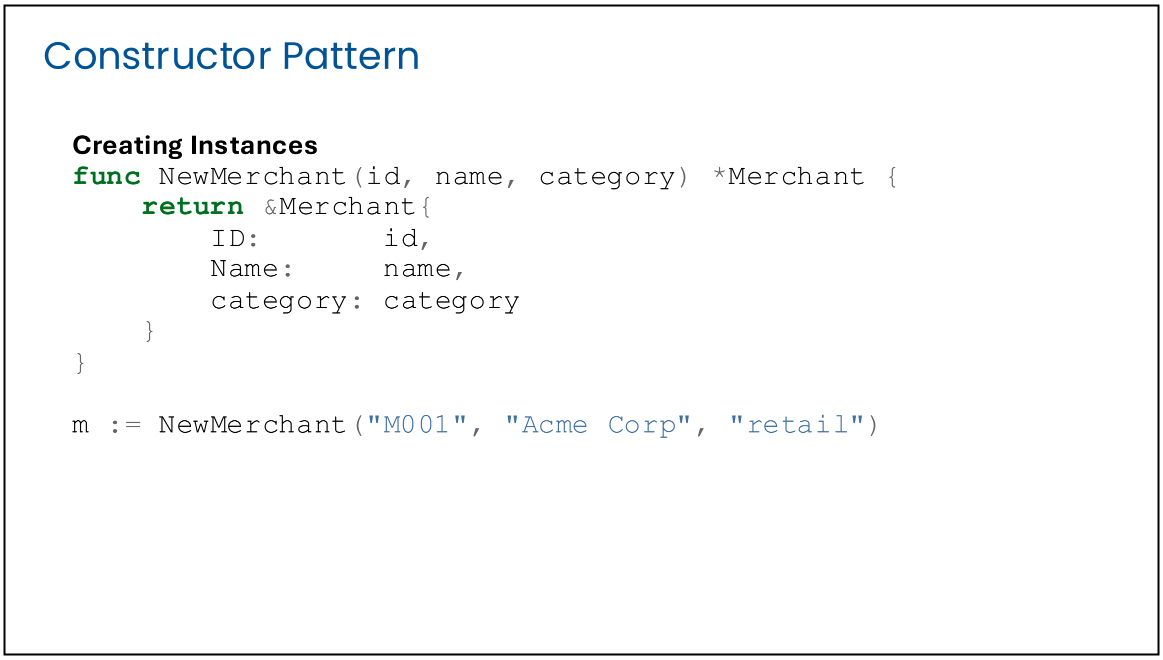
Type System - Structs

Structs Replace Classes

```go
type Merchant struct {
    ID       string  // Exported
    Name     string  // Exported
    category string  // unexported (private)
    Country  string
}
```

- No classes, just structs with fields.
- No inheritance.
- Methods defined separately from struct.
- Composition over inheritance.

- Structs are value types, similar to Java records but more fundamental.
- No `class` keyword in Go - structs + methods achieve same goal.
- Exported fields (capitalized) are like public fields in Java.
- Unexported fields are package-private.
- No getters/setters by convention - just export the field if needed.
- Emphasize the paradigm shift from OOP thinking.

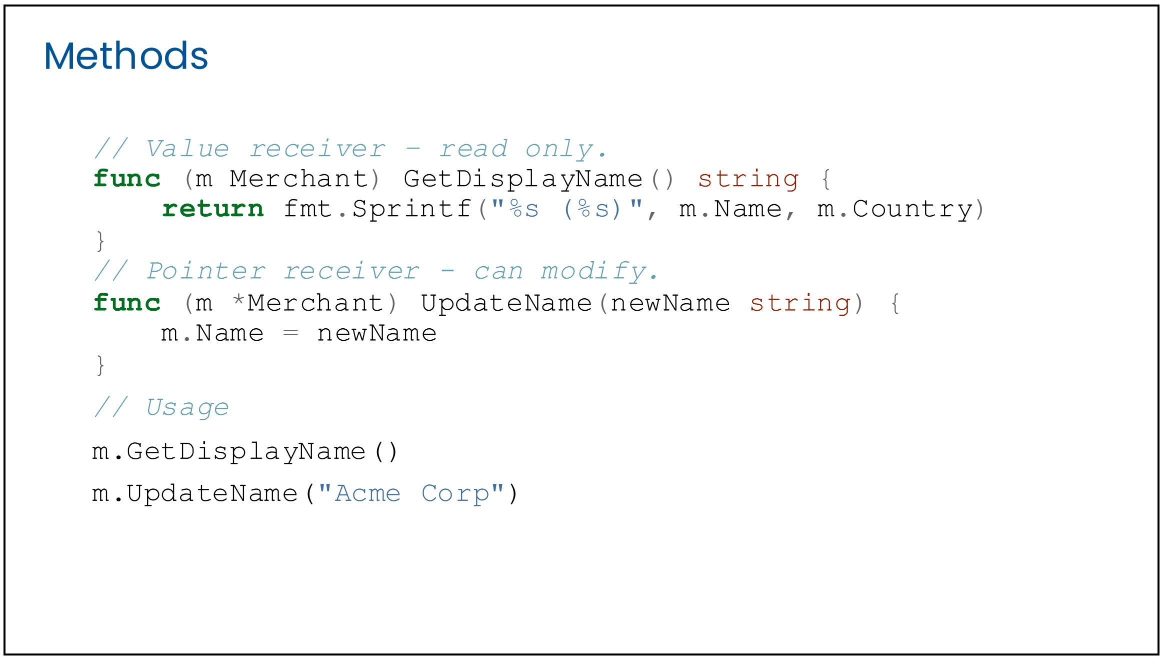


Constructor Pattern

Creating Instances

```go
func NewMerchant(id, name, category) *Merchant {
    return &Merchant{
        ID:       id,
        Name:     name,
        category: category
    }
}

m := NewMerchant("M001", "Acme Corp", "retail")
```

- Go has no constructors - use factory functions by convention.
- `New` prefix is idiomatic (NewMerchant, NewServer, NewClient)
- Returns pointer (`*Merchant`) so caller gets reference to same instance.
- The `&` operator gets the address of a value.
- Struct literal syntax with field names is preferred (more readable)
- Can validate in constructor function before returning.
- Compare to Java's `new Merchant(...)` constructor syntax.



Methods

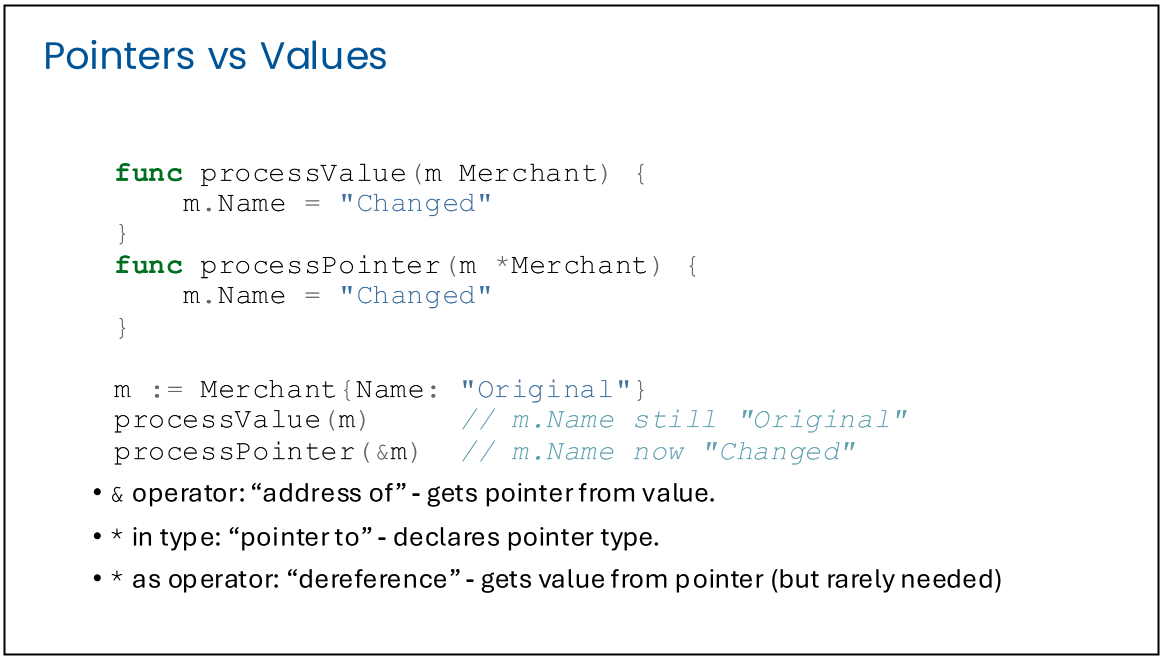
```
// Value receiver - read only.
func (m Merchant) GetDisplayName() string {
    return fmt.Sprintf("%s (%s)", m.Name, m.Country)
}
// Pointer receiver - can modify.
func (m *Merchant) UpdateName(newName string) {
    m.Name = newName
}
// Usage
m.GetDisplayName()
m.UpdateName("Acme Corp")
```

- Methods are functions with a receiver parameter.
- Value receiver: method receives a copy, cannot modify original.
- Pointer receiver: method receives pointer, can modify fields.
- Rule of thumb: use pointer receivers if method modifies state or struct is large.
- Use value receivers for small read-only operations.
- Go will auto-convert between `m.Method()` and `(&m).Method()` when addressable.

Composition Over Inheritance

```go
type Address struct {
    Street  string
    City    string
    Country string
}

type Merchant struct {
    ID   string
    Name string
    Address  // Embedded
}

// Usage
m := Merchant{}
m.Street = "123 Main St"   // Address field promoted
m.Address.Street = "123 Main St"   // Also works
```

- No inheritance in Go - use composition instead.
- Embedding promotes fields and methods to outer type.
- Can embed multiple types (unlike single inheritance)
- More flexible than inheritance - can embed interfaces too.
- Avoids deep inheritance hierarchies that plague Java codebases.
- “Composition over inheritance” is design principle, Go enforces it.

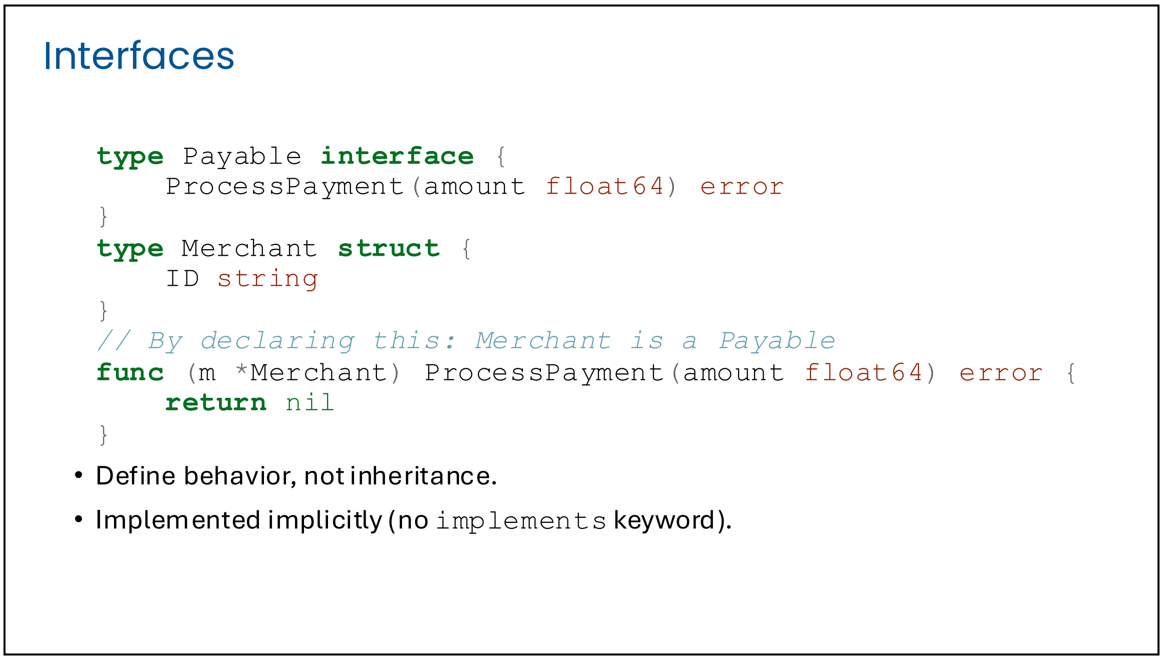


Pointers vs Values

```
func processValue(m Merchant) {
    m.Name = "Changed"
}
func processPointer(m *Merchant) {
    m.Name = "Changed"
}

m := Merchant{Name: "Original"}
processValue(m)      // m.Name still "Original"
processPointer(&m)   // m.Name now "Changed"
```

- `&` operator: “address of” - gets pointer from value.
- `*` in type: “pointer to” - declares pointer type.
- `*` as operator: “dereference” - gets value from pointer (but rarely needed)

- Unlike Java where objects are always references, Go has both.
- Pointers are explicit in type system: `*Merchant` vs `Merchant`.
- `&` operator: “address of” - gets pointer from value.
- `*` in type: “pointer to” - declares pointer type.
- `*` as operator: “dereference” - gets value from pointer (but rarely needed)
- Go automatically dereferences for field access: `ptr.Field` not `(*ptr).Field`.
- Use pointers when: modifying, avoiding copies of large structs, or for semantics.



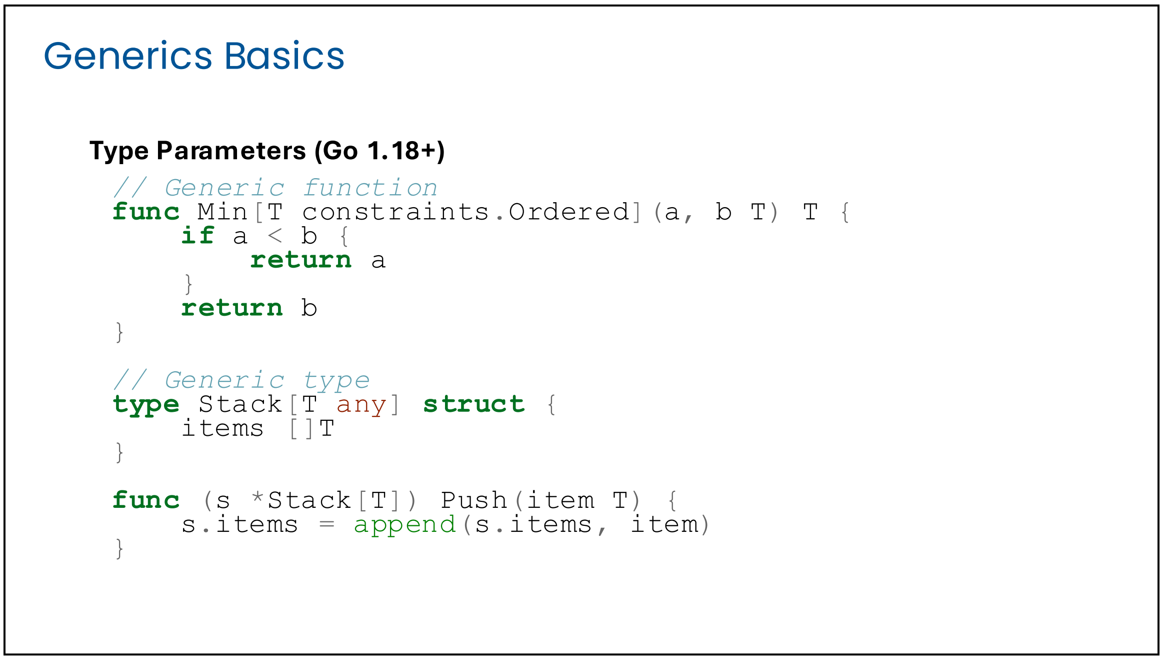
Interfaces

```go
type Payable interface {
    ProcessPayment(amount float64) error
}
type Merchant struct {
    ID string
}
// By declaring this: Merchant is a Payable
func (m *Merchant) ProcessPayment(amount float64) error {
    return nil
}
```

- Define behavior, not inheritance.
- Implemented implicitly (no `implements` keyword).

- This is radical departure from Java’s explicit `implements`.
- If a type has all the methods, it satisfies the interface automatically.
- “Duck typing” at compile time - if it walks like a duck…
- Can retrofit existing types to new interfaces without modifying them.
- No need to declare intent upfront like in Java.

- All types satisfy the empty interface.
- Compare to using Object in Java.



Generics Basics

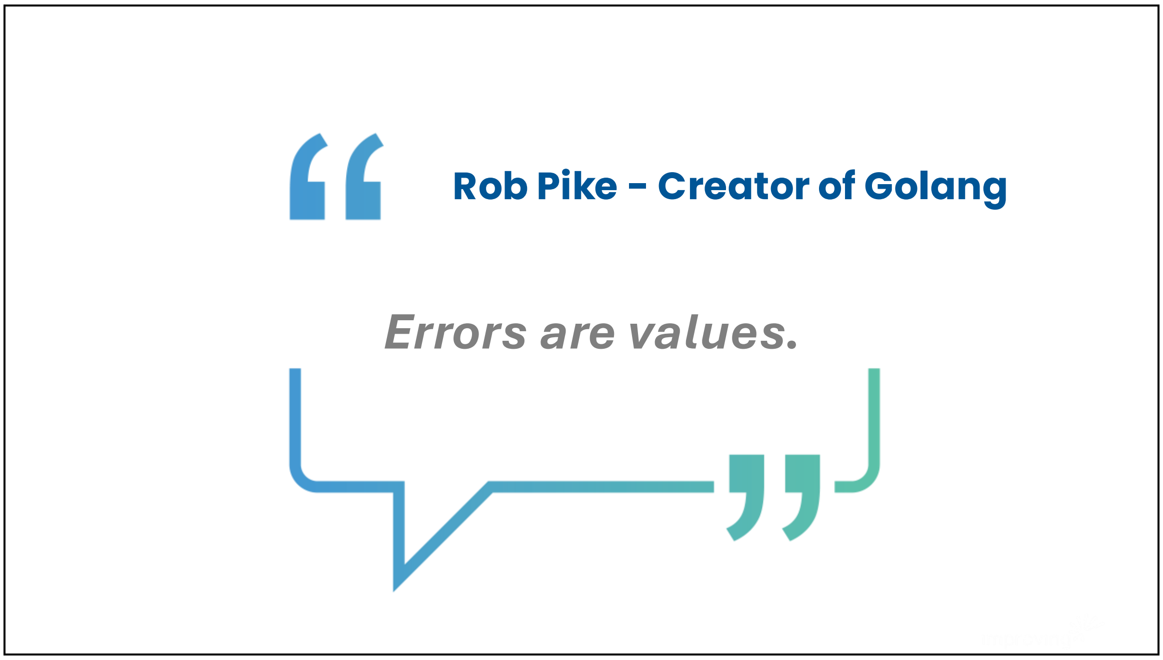
Type Parameters (Go 1.18+)

```go
// Generic function
func Min[T constraints.Ordered](a, b T) T {
    if a < b {
        return a
    }
    return b
}

// Generic type
type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(item T) {
    s.items = append(s.items, item)
}
```

- Generics added in Go 1.18.
- Square brackets `[]` denote type parameters.
- `any` constraint means any type (like Object in Java).
- `constraints.Ordered` requires types that support <, >, etc.
- Less powerful than Java generics but simpler.
- Used sparingly in Go - prefer interfaces when possible.
- Common for collections, algorithms that work on multiple types.
- No type erasure like Java - fully reified generics.

Lab 2

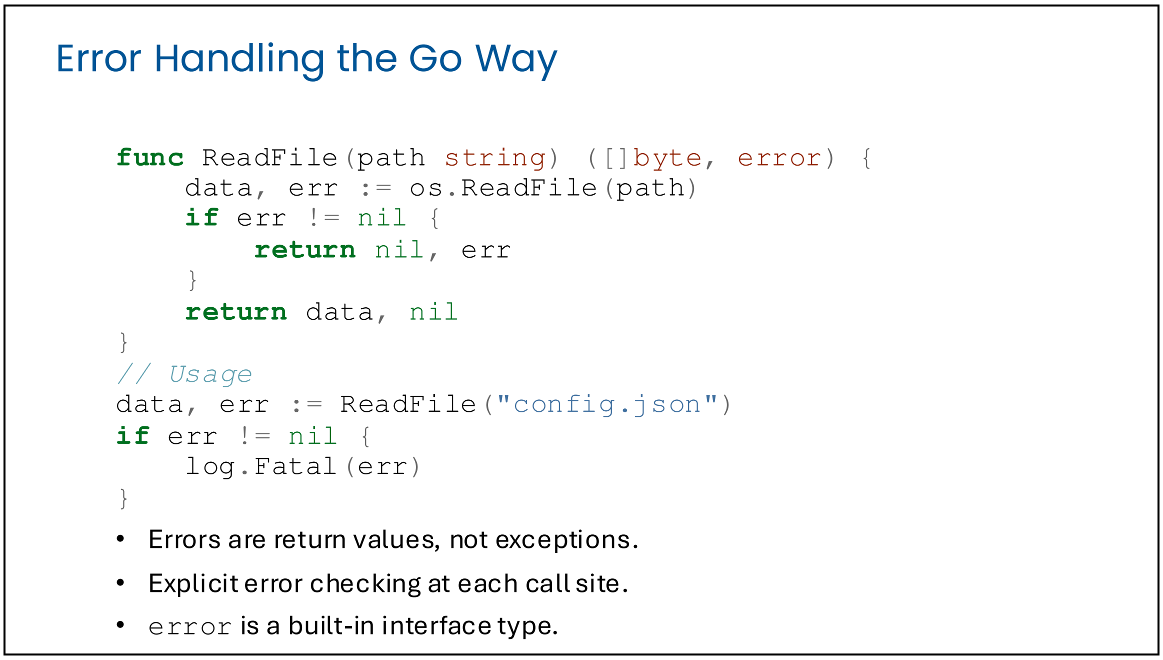
Give students the time to return for debrief.



Rob Pike – Creator of Golang

Errors are values.

- Go forces explicit handling - can’t forget to catch.
- Java exceptions can skip many stack frames invisibly.
- Go errors are part of normal control flow, not exceptional.
- Stack traces in Go require manual context but give better domain info.
- Java checked exceptions became burden, Go avoids that with return values.
- Performance: Go errors are faster (no stack unwinding)
- Readability: Go is more verbose but control flow is explicit.

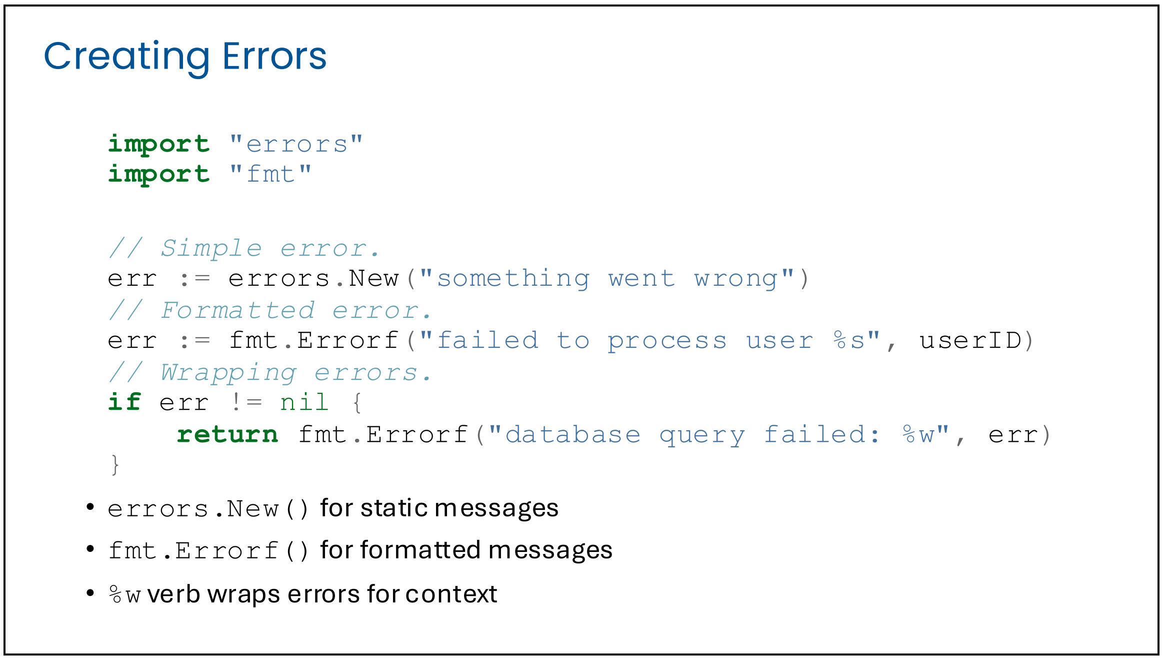


Error Handling the Go Way

```go
func ReadFile(path string) ([]byte, error) {
    data, err := os.ReadFile(path)
    if err != nil {
        return nil, err
    }
    return data, nil
}
// Usage
data, err := ReadFile("config.json")
if err != nil {
    log.Fatal(err)
}
```

- Errors are return values, not exceptions.
- Explicit error checking at each call site.
- `error` is a built-in interface type.

- This is most controversial Go feature for Java developers.
- No try-catch blocks - errors are values you check explicitly.
- Forces you to think about error handling at every step.
- `if err != nil` pattern becomes second nature.
- error is an interface with single method: `Error() string`.
- nil error means success.
- By convention, error is last return value.
- Can’t ignore errors accidentally (compiler warns on unused values)
- More verbose but clearer control flow - no hidden exception paths.



Creating Errors

```go
import "errors"
import "fmt"

// Simple error.
err := errors.New("something went wrong")
// Formatted error.
err := fmt.Errorf("failed to process user %s", userID)
// Wrapping errors.
if err != nil {
    return fmt.Errorf("database query failed: %w", err)
}
```

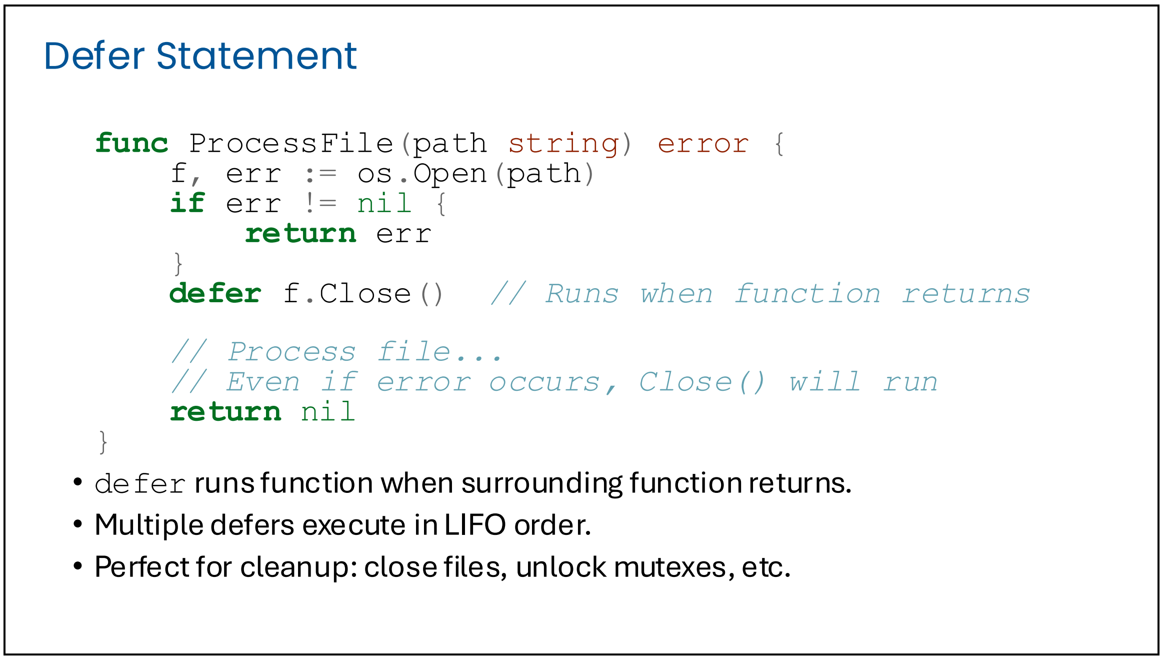
- `errors.New()` for static messages
- `fmt.Errorf()` for formatted messages
- `%w` verb wraps errors for context

- errors.New creates simple error with string message.
- fmt.Errorf like sprintf but returns error type.
- `%w` wrapping preserves original error for inspection later.
- Wrapping builds error chain showing context at each layer.
- Can unwrap with errors.Unwrap() or use errors.Is() / errors.As()
- Unlike Java stack traces, Go errors need manual context addition.
- Good practice: add context at each layer (“failed to X: %w”).

Custom Error Types

```go
type ValidationError struct {
    Field   string
    Message string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("%s: %s", e.Field, e.Message)
}

// Usage
if user.Age < 18 {
    return &ValidationError{
        Field: "Age",
        Message: "must be 18 or older",
    }
}
```

- Any type with `Error() string` method implements error interface.
- Custom types allow structured error information.
- Can include fields for error codes, HTTP status, etc.
- Type assertions let you check for specific error types.
- Similar to Java custom exception classes but without inheritance.
- Use pointer receivers for Error() method (idiomatic)
- Can have multiple custom error types for different failure modes.

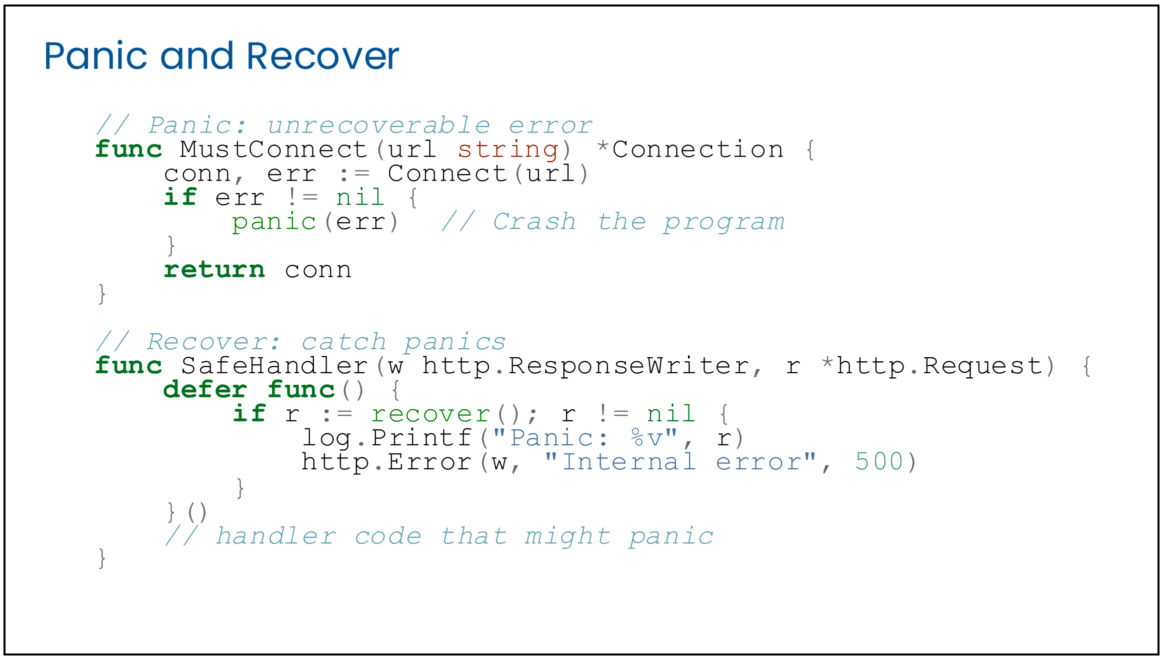


Defer Statement

```
func ProcessFile(path string) error {
    f, err := os.Open(path)
    if err != nil {
        return err
    }
    defer f.Close()  // Runs when function returns

    // Process file...
    // Even if error occurs, Close() will run
    return nil
}
```

- `defer` runs function when surrounding function returns.
- Multiple defers execute in LIFO order.
- Perfect for cleanup: close files, unlock mutexes, etc.

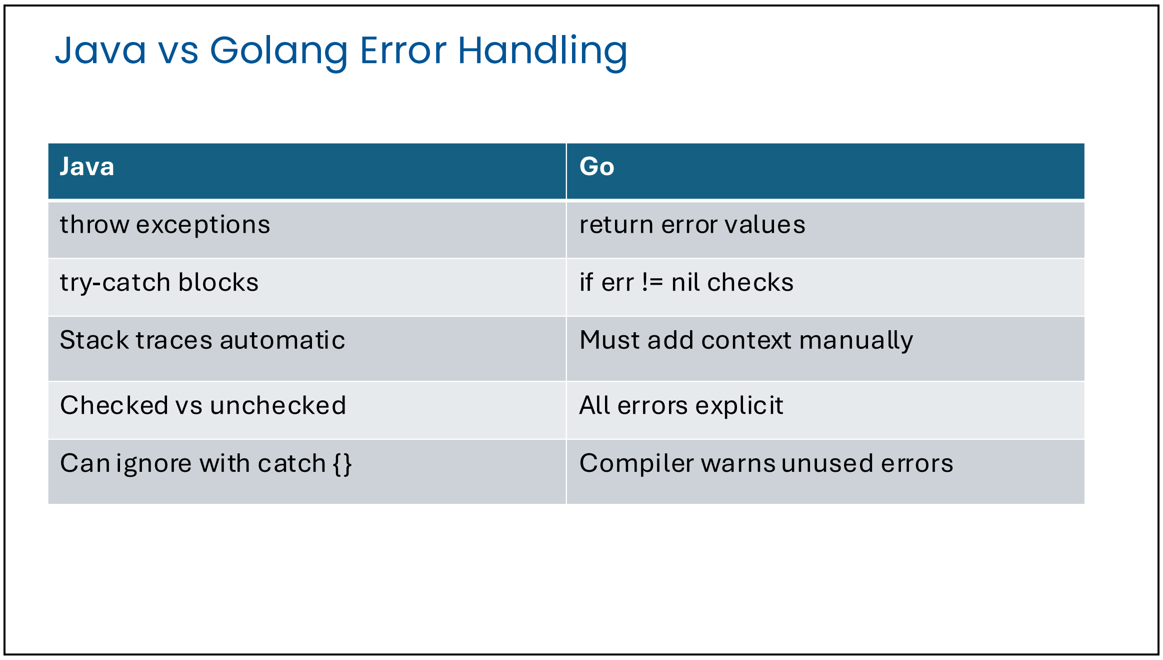
- defer is Go’s answer to Java’s try-with-resources.
- Deferred functions run after return statement evaluates.
- LIFO order: last defer runs first (like stack)
- Arguments evaluated immediately but function runs later.
- Common pattern: acquire resource, immediately defer release.
- Makes it hard to forget cleanup since it’s next to acquisition.
- Can defer anonymous functions for complex cleanup.



Panic and Recover

```go
// Panic: unrecoverable error
func MustConnect(url string) *Connection {
    conn, err := Connect(url)
    if err != nil {
        panic(err)   // Crash the program
    }
    return conn
}

// Recover: catch panics
func SafeHandler(w http.ResponseWriter, r *http.Request) {
    defer func() {
        if r := recover(); r != nil {
            log.Printf("Panic: %v", r)
            http.Error(w, "Internal error", 500)
        }
    }()
    // handler code that might panic
}
```

- panic is like throwing unchecked exception - crashes program.
- recover is like catch - but only works in deferred functions.
- Use panic for programmer errors or truly unrecoverable situations.
- Don’t use panic for normal error handling!
- recover must be called in defer to catch panics.
- Similar to Java’s RuntimeException but even more rare.
- Most Go code never uses panic/recover.
- Library code should return errors, not panic.
- `Alternatively use log.Fatal(err)`



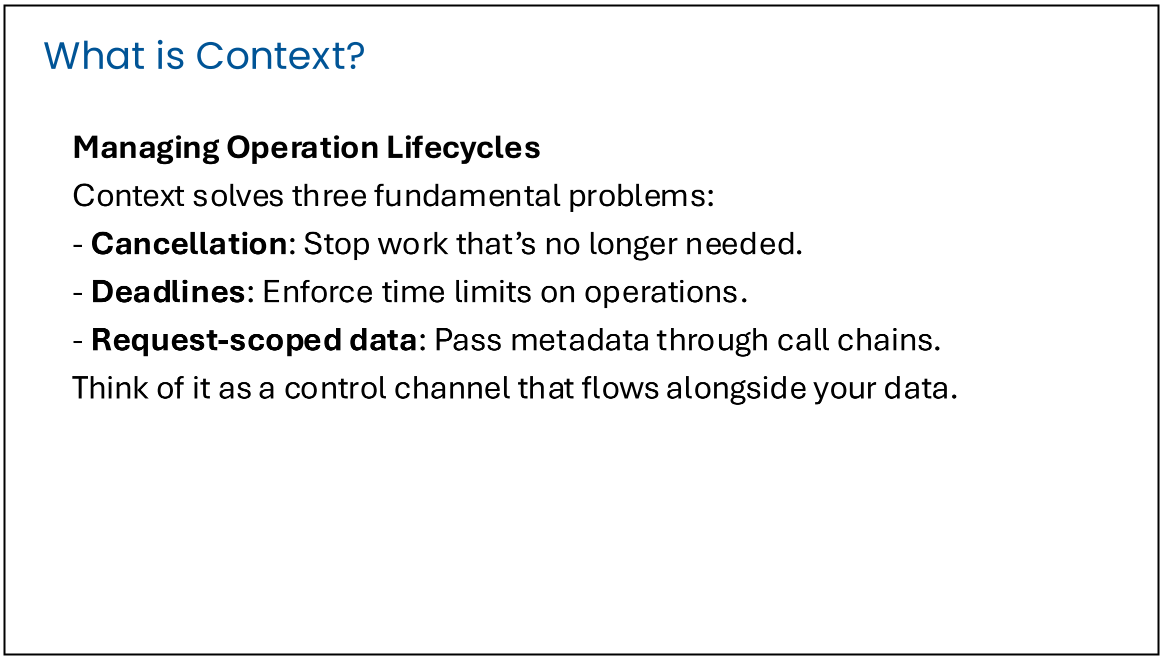
Java vs Golang Error Handling

Java	Go
throw exceptions	return error values
try-catch blocks	if err != nil checks
Stack traces automatic	Must add context manually
Checked vs unchecked	All errors explicit
Can ignore with catch {}	Compiler warns unused errors

Panic/recover is an alternative to throwing an exceptions and try-catch, but it not considered the norm.

Lab 3

Give students the time to return for debrief.



What is Context?

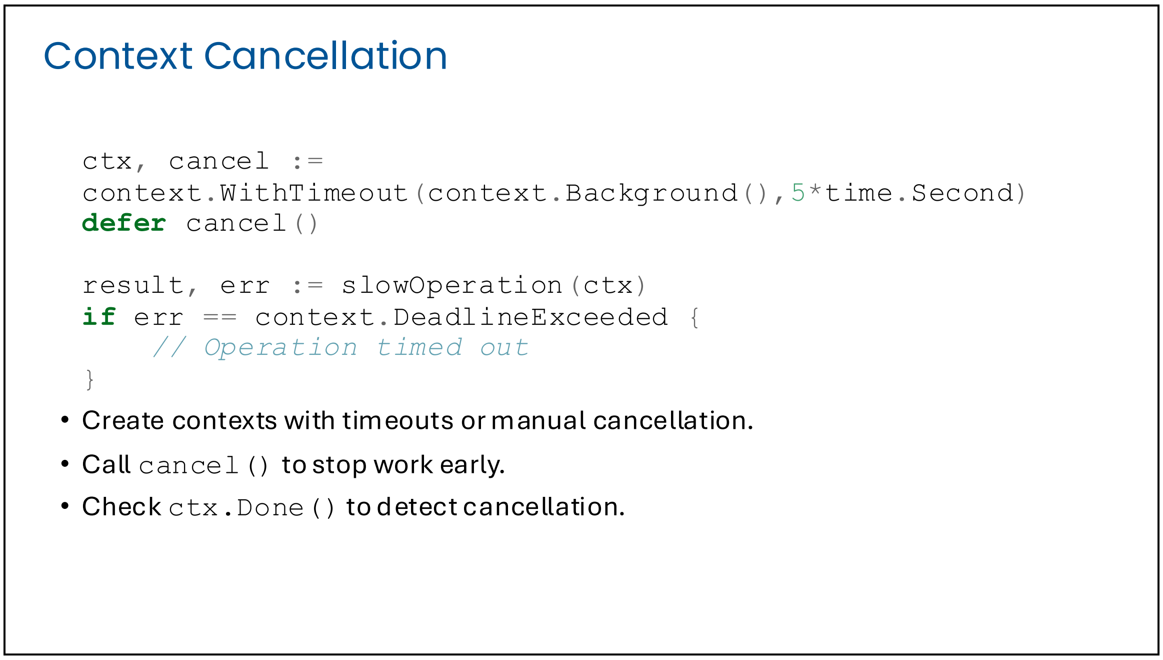
Managing Operation Lifecycles

Context solves three fundamental problems:

- **Cancellation**: Stop work that's no longer needed.
- **Deadlines**: Enforce time limits on operations.
- **Request-scoped data**: Pass metadata through call chains.

Think of it as a control channel that flows alongside your data.

- Context is one of Go's most important standard library packages.
- Not just for concurrency - used in any long-running or blocking operation.
- Similar to Java's ThreadLocal but more powerful and explicit.
- Every HTTP handler, database query, gRPC call should accept context.
- The explicit passing makes control flow clear and testable.

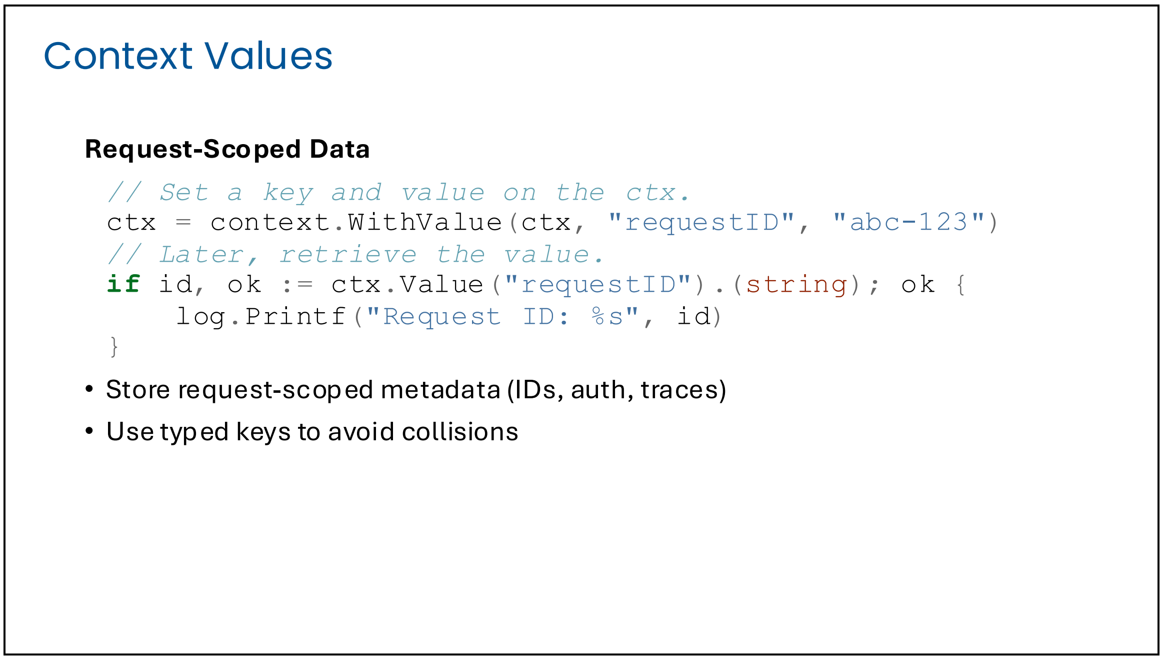


Context Cancellation

```
ctx, cancel :=
context.WithTimeout(context.Background(),5*time.Second)
defer cancel()

result, err := slowOperation(ctx)
if err == context.DeadlineExceeded {
    // Operation timed out
}
```

- Create contexts with timeouts or manual cancellation.
- Call `cancel()` to stop work early.
- Check `ctx.Done()` to detect cancellation.

- context.Background() is the root - never cancelled, used at program start.
- WithTimeout/WithDeadline return context that auto-cancels after duration.
- WithCancel gives you manual control via cancel function.
- Always defer cancel() to prevent resource leaks.
- Cancellation cascades to child contexts automatically.
- ctx.Done() returns a channel that closes when cancelled.
- ctx.Err() tells you why: Cancelled or DeadlineExceeded.



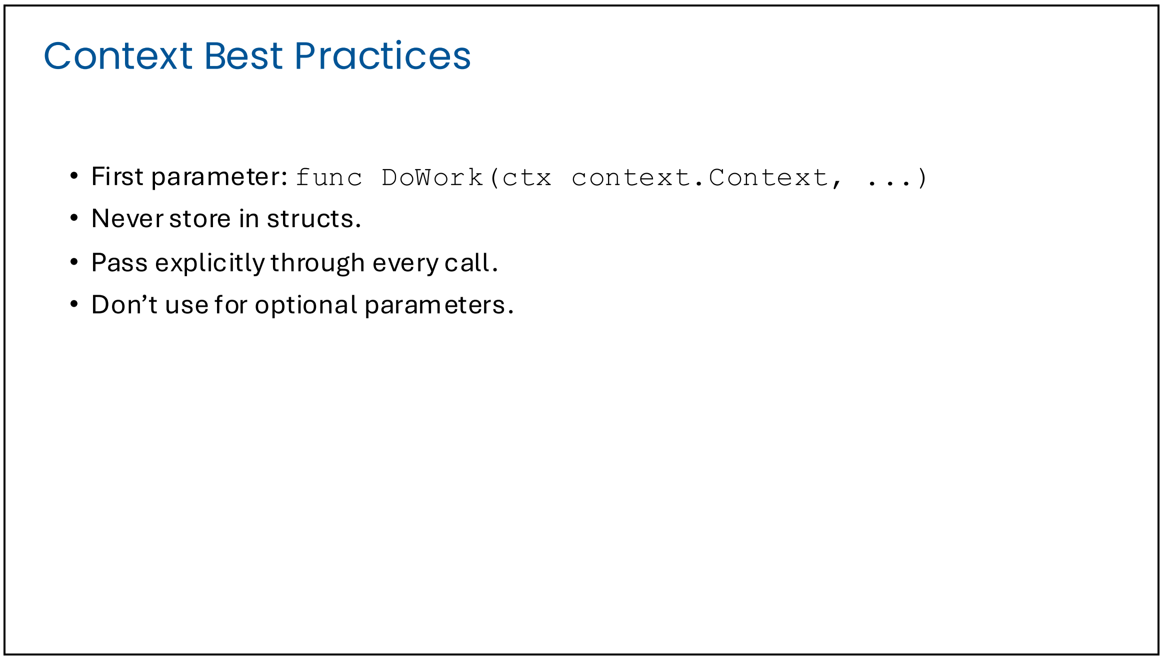
Context Values

Request-Scoped Data

```go
// Set a key and value on the ctx.
ctx = context.WithValue(ctx, "requestID", "abc-123")
// Later, retrieve the value.
if id, ok := ctx.Value("requestID").(string); ok {
    log.Printf("Request ID: %s", id)
}
```

- Store request-scoped metadata (IDs, auth, traces)
- Use typed keys to avoid collisions

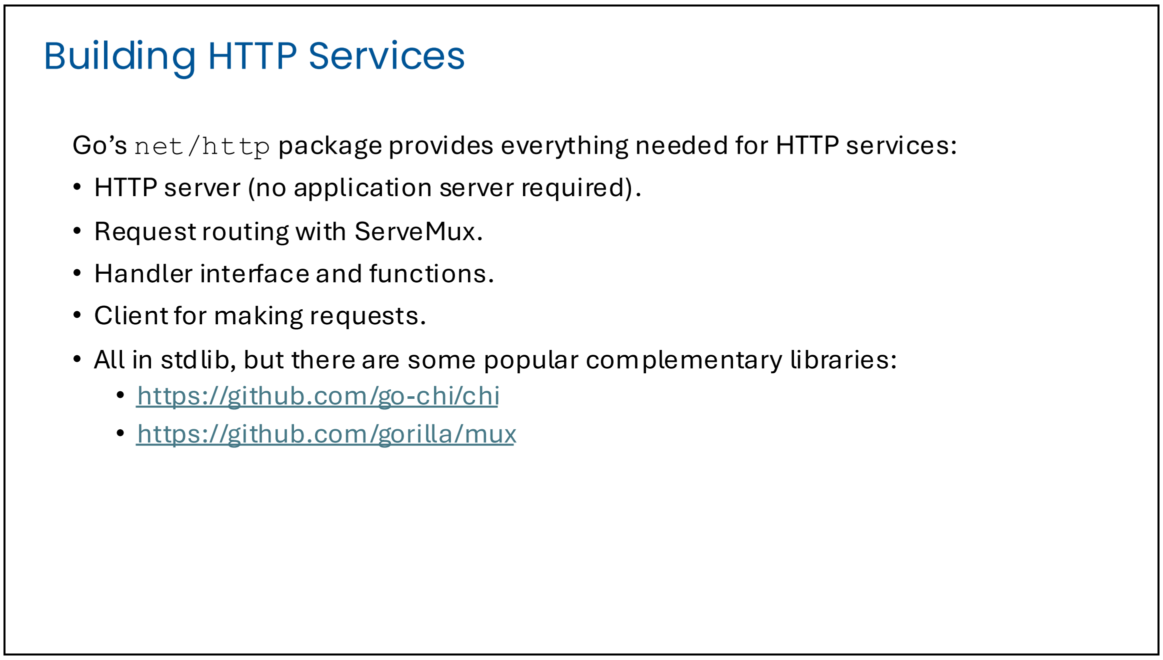
- Context values flow through the entire request lifecycle.
- Similar to Java’s ThreadLocal or SLF4J’s MDC (Mapped Diagnostic Context)
- Use sparingly - only for cross-cutting concerns like request IDs.
- Type assertion needed when retrieving (returns interface{})
- Common use cases: request ID, trace context, user auth tokens.
- OpenTelemetry propagates trace spans through context values.
- Don’t abuse it - prefer explicit function parameters for business logic.



Context Best Practices

- First parameter: `func DoWork(ctx context.Context, ...)`
- Never store in structs.
- Pass explicitly through every call.
- Don’t use for optional parameters.

- First parameter convention is universal in Go.
- Storing context in struct creates lifecycle confusion and leaks.
- Each operation creates its own context tree.
- http.Request provides Context() method - extract and pass it.
- Middleware can wrap contexts to add timeouts or values.
- Context package is stdlib but conventions evolved from community practice.

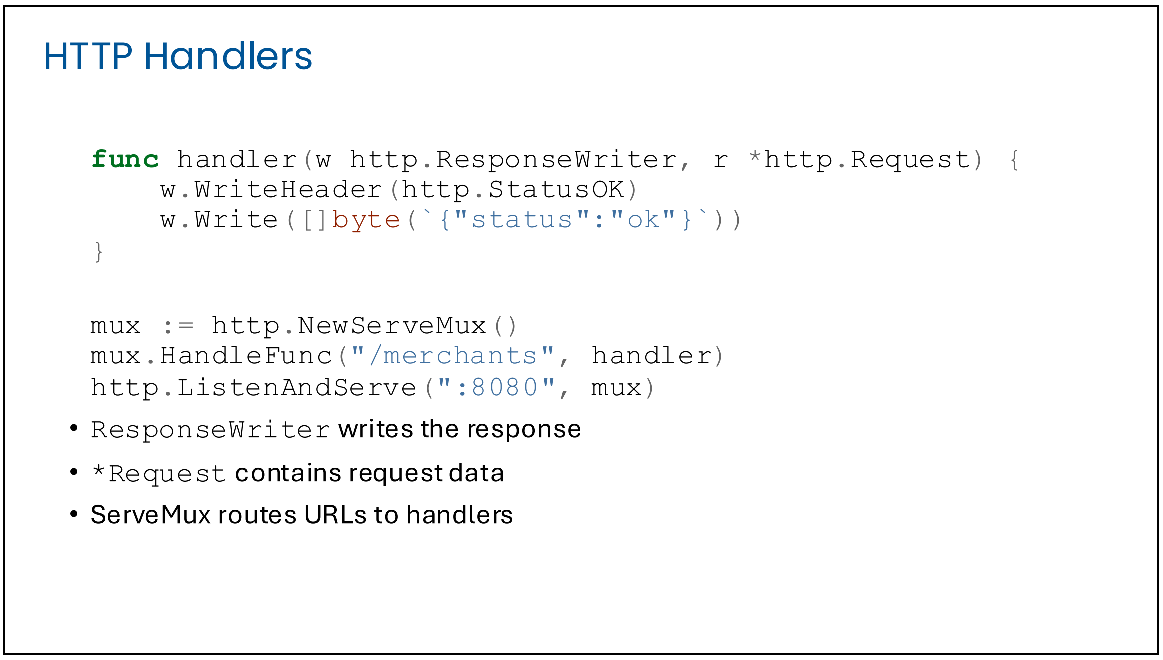


Building HTTP Services

Go's `net/http` package provides everything needed for HTTP services:

- HTTP server (no application server required).
- Request routing with ServeMux.
- Handler interface and functions.
- Client for making requests.
- All in stdlib, but there are some popular complementary libraries:
  - https://github.com/go-chi/chi
  - https://github.com/gorilla/mux

- Unlike Java which requires Tomcat/Jetty or Spring Boot, Go includes HTTP server.
- net/http is used in production at Google, Uber, Netflix, many others.
- Simple enough for beginners, powerful enough for production.
- Handler pattern is just a function signature - no complex interfaces.
- ServeMux is basic router, but third-party routers (chi, gorilla/mux) integrate easily.
- Go's approach: minimalism and clarity over framework magic.
- Students often surprised at how little code needed.

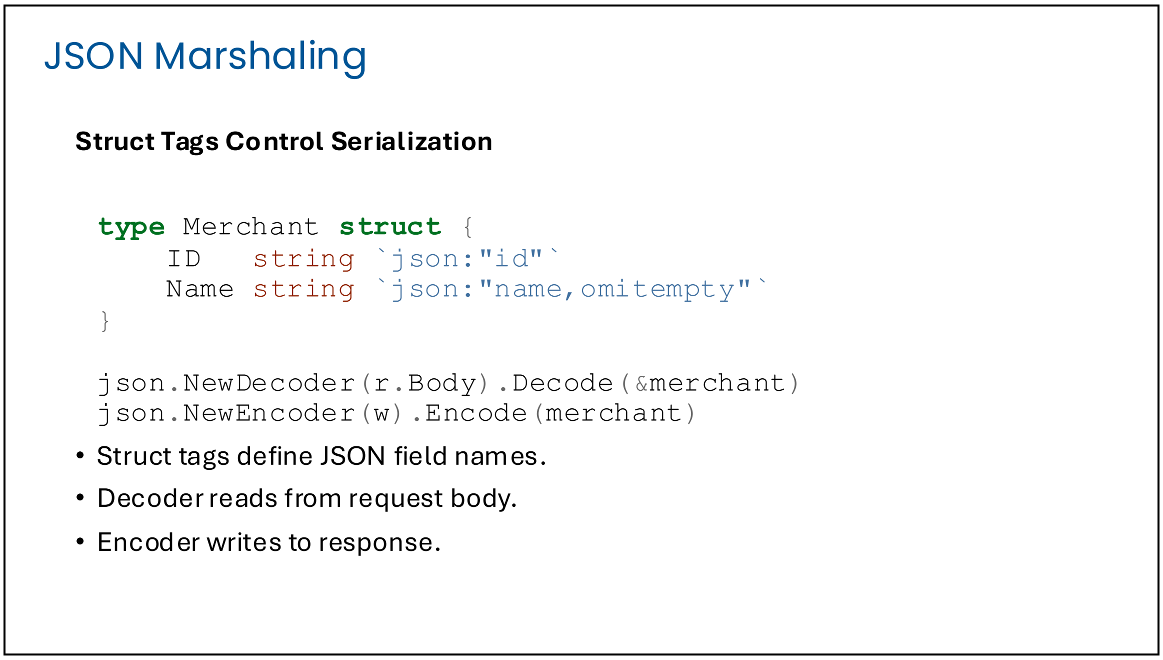


HTTP Handlers

```go
func handler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"status":"ok"}`))
}

mux := http.NewServeMux()
mux.HandleFunc("/merchants", handler)
http.ListenAndServe(":8080", mux)
```

- `ResponseWriter` writes the response
- `*Request` contains request data
- ServeMux routes URLs to handlers

- Handler signature is all you need - no interface implementation required.
- ResponseWriter is like Java’s HttpServletResponse.
- Request has method, headers, body, URL, and context.
- ServeMux maps paths to handlers (basic but sufficient).
- ListenAndServe blocks and starts the server.
- Unlike Spring’s @RestController annotations, this is explicit.
- Can see entire request flow - no hidden framework magic.



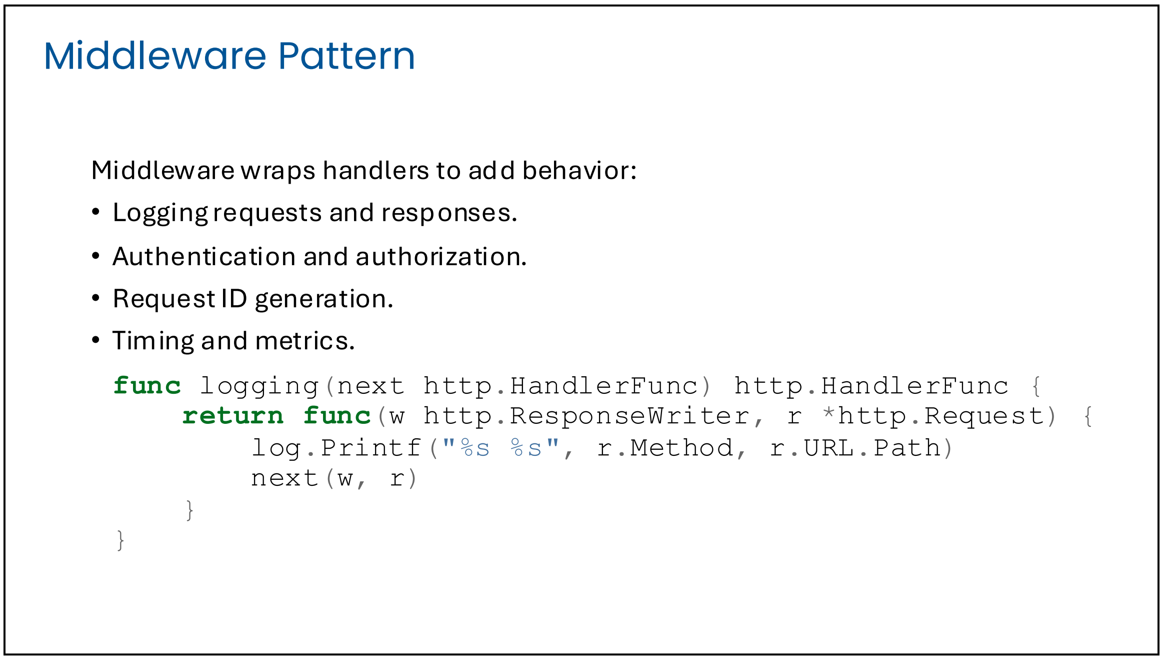
JSON Marshaling

Struct Tags Control Serialization

```
type Merchant struct {
    ID   string `json:"id"`
    Name string `json:"name,omitempty"`
}

json.NewDecoder(r.Body).Decode(&merchant)
json.NewEncoder(w).Encode(merchant)
```

- Struct tags define JSON field names.
- Decoder reads from request body.
- Encoder writes to response.

- Similar to Java’s Jackson @JsonProperty annotations.
- `json:"fieldName"` sets the JSON key.
- `omitempty` skips zero values when marshaling.
- Unexported fields (lowercase) are never marshaled.
- Encoder writes directly to ResponseWriter (no intermediate buffer)
- Decoder reads from request body stream.



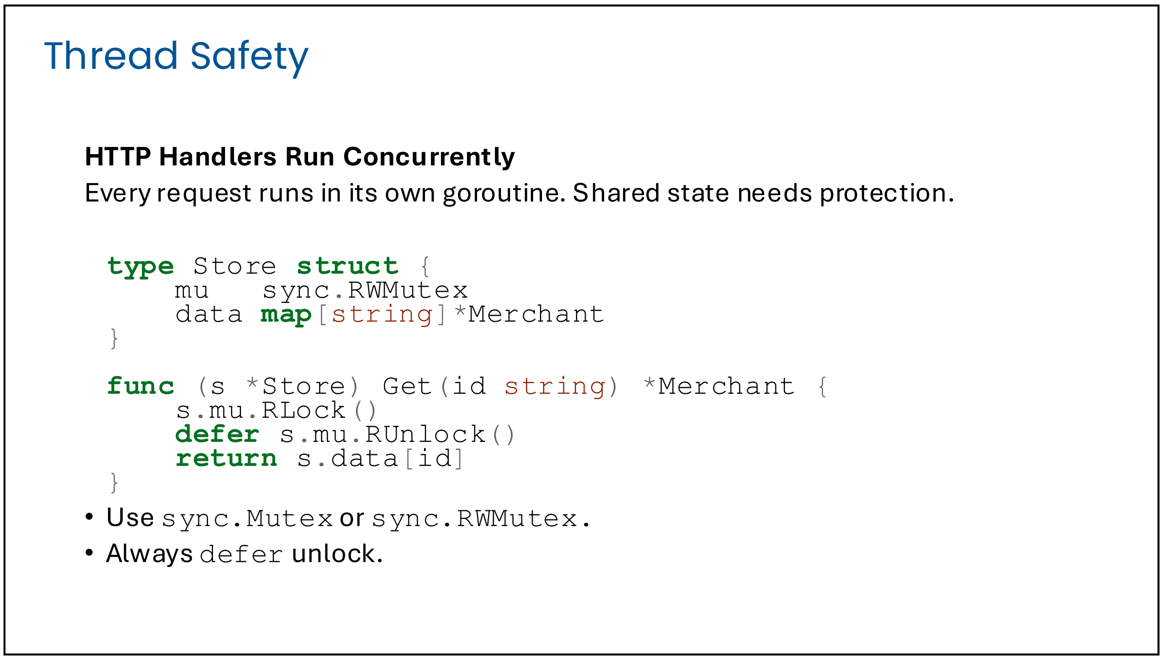
Middleware Pattern

Middleware wraps handlers to add behavior:

- Logging requests and responses.
- Authentication and authorization.
- Request ID generation.
- Timing and metrics.

```go
func logging(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        log.Printf("%s %s", r.Method, r.URL.Path)
        next(w, r)
    }
}
```

- Middleware is just a function that takes handler, returns handler.
- Pure function composition - no special interfaces.
- Can stack middleware: `auth(logging(rateLimit(handler)))`.
- Compare to Java servlet filters or Spring interceptors.
- More explicit than annotations - see the wrapping clearly.
- Context perfect for passing middleware data to handlers.



Thread Safety

HTTP Handlers Run Concurrently

Every request runs in its own goroutine. Shared state needs protection.

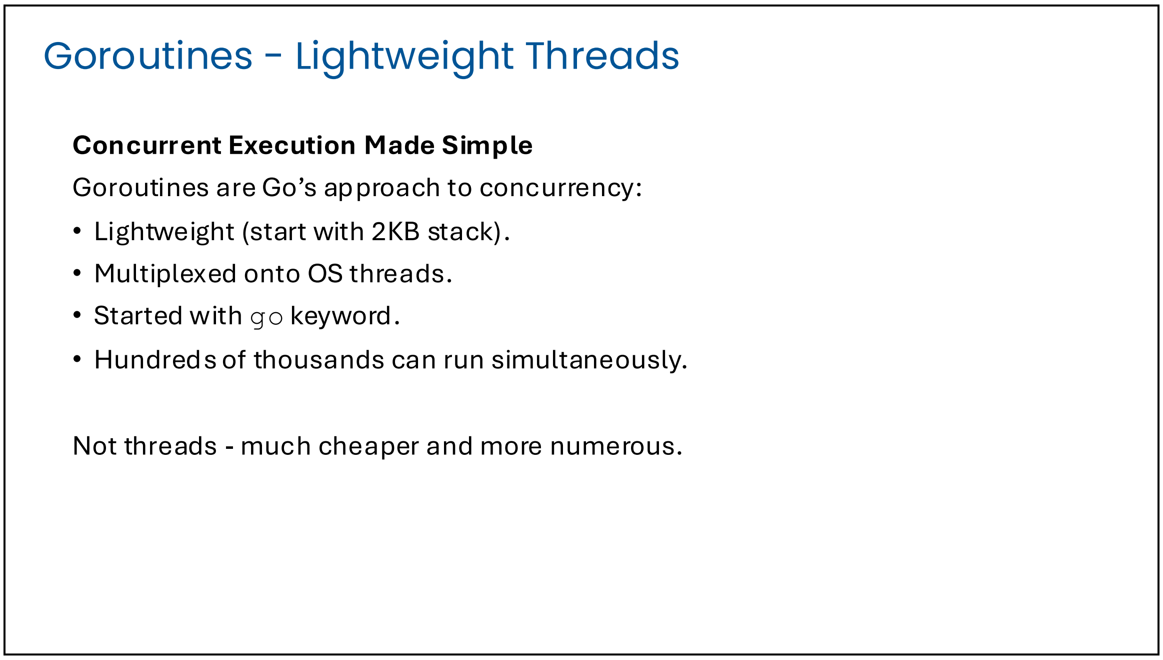
```go
type Store struct {
    mu   sync.RWMutex
    data map[string]*Merchant
}

func (s *Store) Get(id string) *Merchant {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.data[id]
}
```

- Use `sync.Mutex` or `sync.RWMutex`.
- Always `defer` unlock.

- HTTP handlers run concurrently - one goroutine per request.
- Unlike Java where you might use synchronized or concurrent collections.
- Go maps are NOT thread-safe (different from ConcurrentHashMap)
- defer ensures unlock even if function panics.
- Students learn to think about concurrent access from the start.

Lab 4

Give students the time to return for debrief.



Goroutines - Lightweight Threads

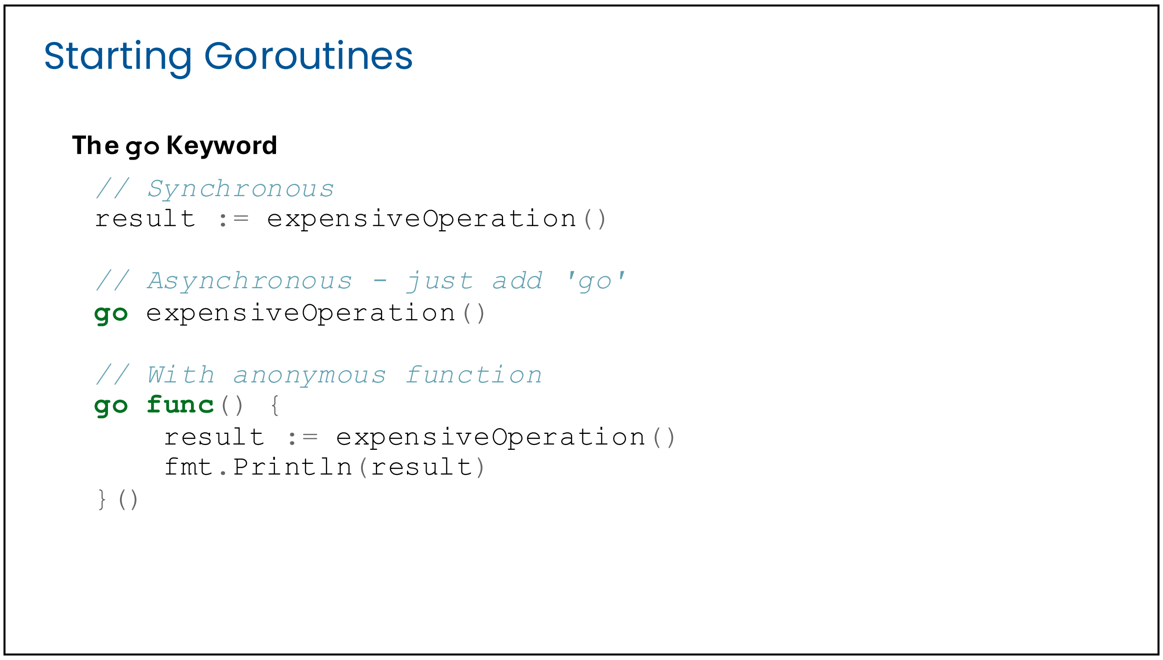
Concurrent Execution Made Simple

Goroutines are Go's approach to concurrency:

- Lightweight (start with 2KB stack).
- Multiplexed onto OS threads.
- Started with `go` keyword.
- Hundreds of thousands can run simultaneously.

Not threads - much cheaper and more numerous.

- Unlike Java threads which map 1:1 to OS threads.
- Goroutines managed by Go runtime - more efficient.
- Starting a goroutine: just add `go` before function call.
- No thread pools to manage - runtime handles scheduling.
- Can create millions of goroutines (vs thousands of threads)
- Stack grows and shrinks automatically.
- Java's Project Loom (virtual threads) inspired by goroutines.

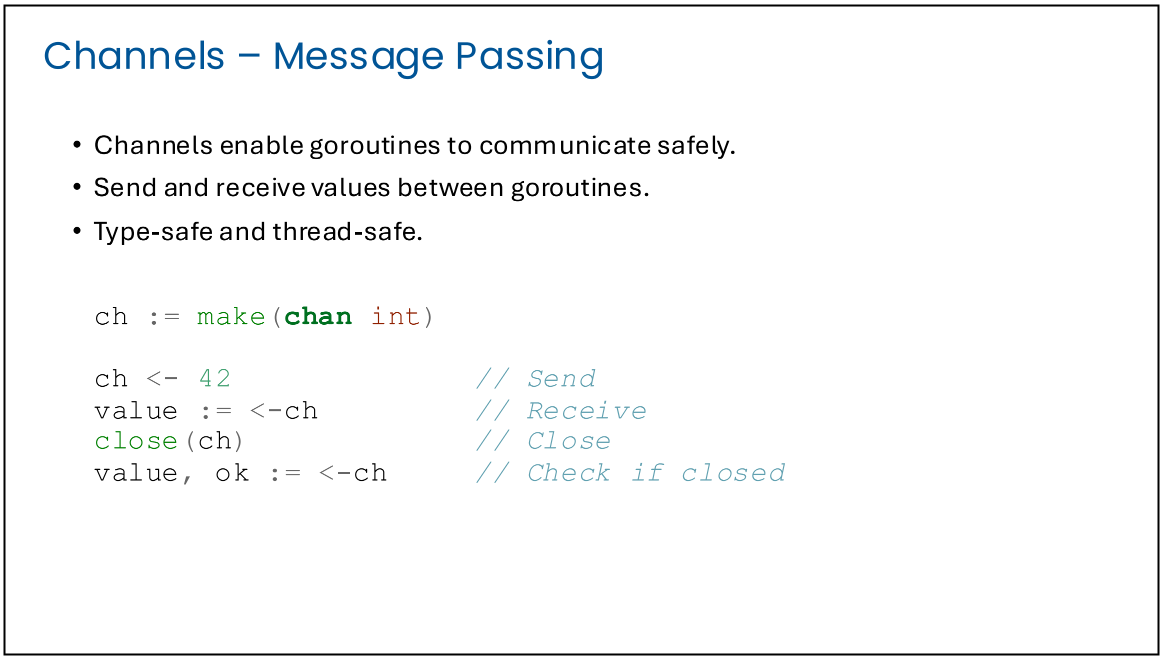


Starting Goroutines

The go Keyword

```go
// Synchronous
result := expensiveOperation()

// Asynchronous - just add 'go'
go expensiveOperation()

// With anonymous function
go func() {
    result := expensiveOperation()
    fmt.Println(result)
}()
```

- go keyword starts new goroutine - that simple.
- Function executes concurrently with caller.
- Caller doesn’t wait - continues immediately.
- Often used with anonymous functions for inline concurrency.
- No thread pool creation, no executor service needed.
- Be careful: goroutines must coordinate to avoid races.
- Need synchronization primitive (channels, WaitGroup) for coordination.



Channels – Message Passing

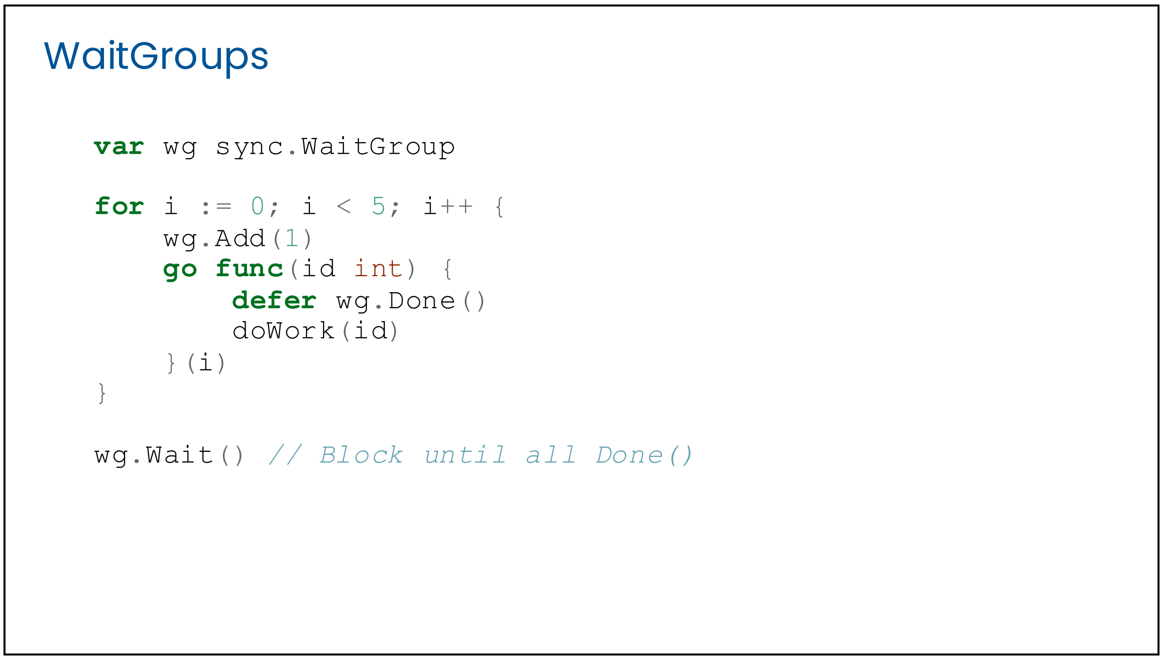
- Channels enable goroutines to communicate safely.
- Send and receive values between goroutines.
- Type-safe and thread-safe.

```go
ch := make(chan int)

ch <- 42                    // Send
value := <-ch               // Receive
close(ch)                   // Close
value, ok := <-ch           // Check if closed
```

- “Don’t communicate by sharing memory; share memory by communicating.”
- Channels are Go’s primary synchronization mechanism.
- Unlike Java’s shared memory with locks.
- Unbuffered channels block until both send and receive ready.
- Buffered channels: `make(chan int, 10)` hold values in queue.
- Closing channel signals no more values coming.
- <- operator for send and receive (direction matters!)

- Unbuffered channels are synchronization points.
- Sending blocks until someone receives.
- Receiving blocks until someone sends.
- close() signals no more values (sender’s responsibility)
- Receiving from closed channel returns zero value and false.

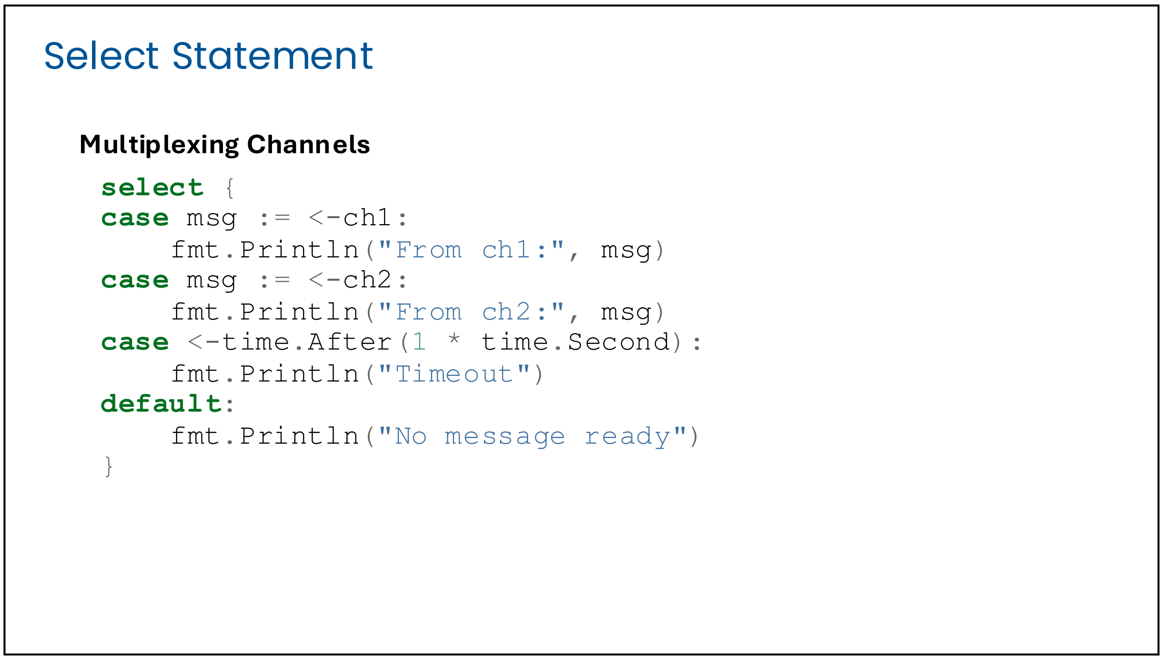
- Can range over channel: for value := range ch.
- Common pattern: close channel to signal completion.



WaitGroups

```go
var wg sync.WaitGroup

for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done()
        doWork(id)
    }(i)
}

wg.Wait() // Block until all Done()
```

- WaitGroup is like Java’s CountDownLatch.
- Use when you need to wait for multiple goroutines to complete.
- Add() before starting goroutine.
- defer Done() ensures it’s called even on panic.
- Wait() blocks until all goroutines call Done()
- Common mistake: forgetting to pass loop variables to goroutine.
- Must pass as parameter: go func(i int) { }(i)
- Alternative: use channels to signal completion.



Select Statement

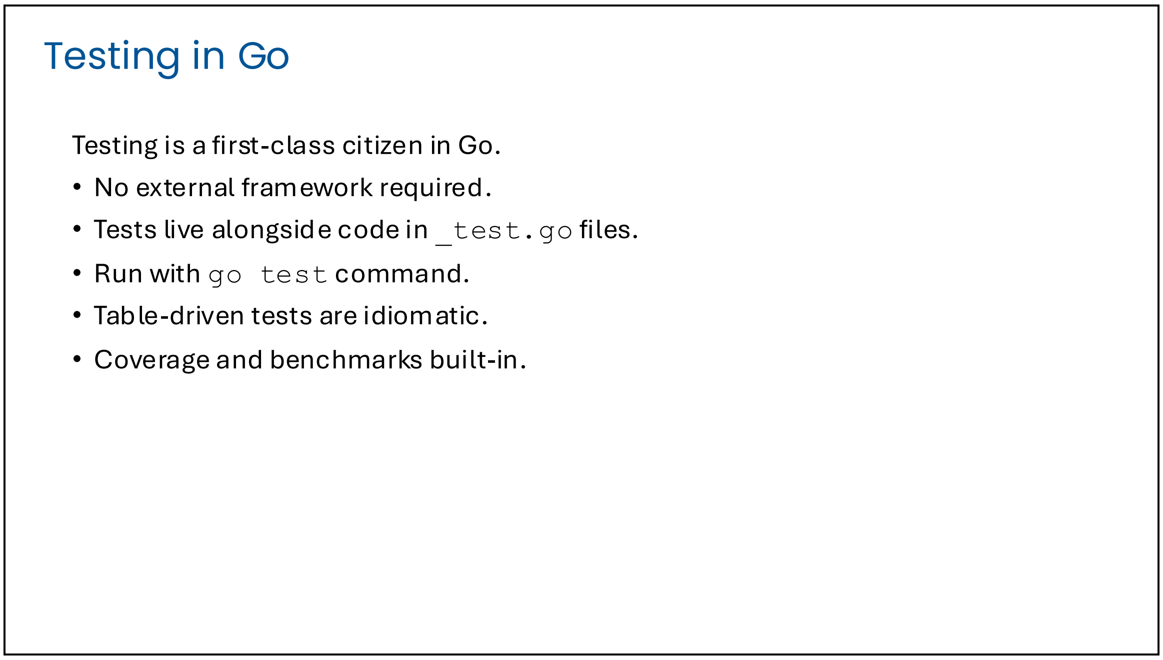
Multiplexing Channels

```go
select {
case msg := <-ch1:
    fmt.Println("From ch1:", msg)
case msg := <-ch2:
    fmt.Println("From ch2:", msg)
case <-time.After(1 * time.Second):
    fmt.Println("Timeout")
default:
    fmt.Println("No message ready")
}
```

- select is like switch but for channels.
- Waits until one case can proceed.
- If multiple ready, chooses randomly (fairness)
- time.After() for timeouts (common pattern)
- default case makes select non-blocking.
- Without default, select blocks until a case ready.
- Very powerful for complex coordination.

Lab 5

Give students the time to return for debrief.

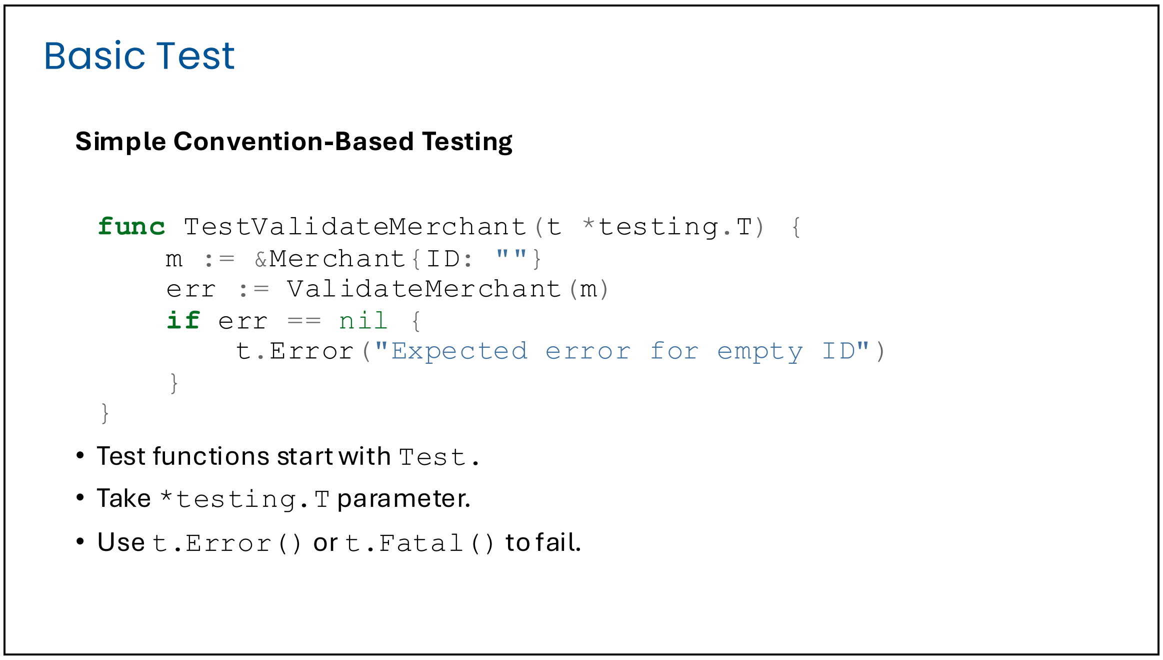


Testing in Go

Testing is a first-class citizen in Go.

- No external framework required.
- Tests live alongside code in `_test.go` files.
- Run with `go test` command.
- Table-driven tests are idiomatic.
- Coverage and benchmarks built-in.

- Unlike Java requiring JUnit, Go’s testing is in stdlib.
- No annotations, no complex setup - just functions.
- Tests in same package can test unexported functions.
- go test discovers and runs all tests automatically.
- Fast compilation means fast test feedback loops.
- Coverage built-in: go test -cover.
- Benchmarks built-in: go test -bench.



Basic Test

Simple Convention-Based Testing

```go
func TestValidateMerchant(t *testing.T) {
    m := &Merchant{ID: ""}
    err := ValidateMerchant(m)
    if err == nil {
        t.Error("Expected error for empty ID")
    }
}
```

- Test functions start with `Test`.
- Take `*testing.T` parameter.
- Use `t.Error()` or `t.Fatal()` to fail.

- Test files end in _test.go and live next to tested code.
- Function name must start with Test followed by exported name.
- t.Error reports failure but continues test.
- t.Fatal reports failure and stops test immediately.
- t.Errorf and t.Fatalf provide formatted messages.
- Tests in same package can access unexported identifiers

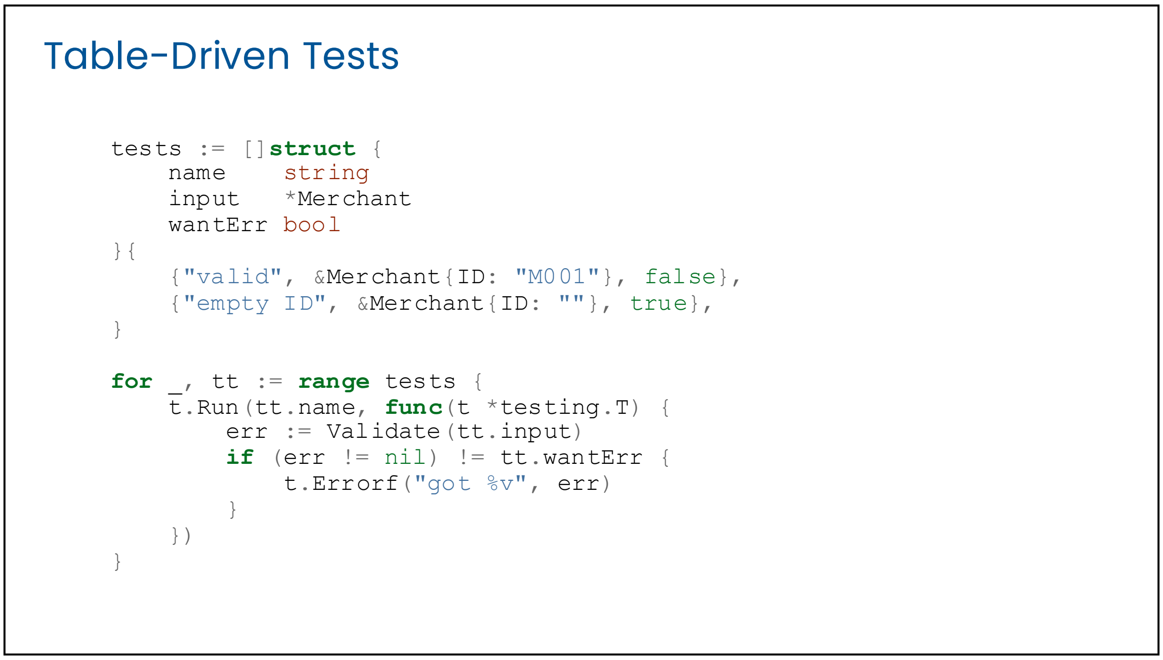
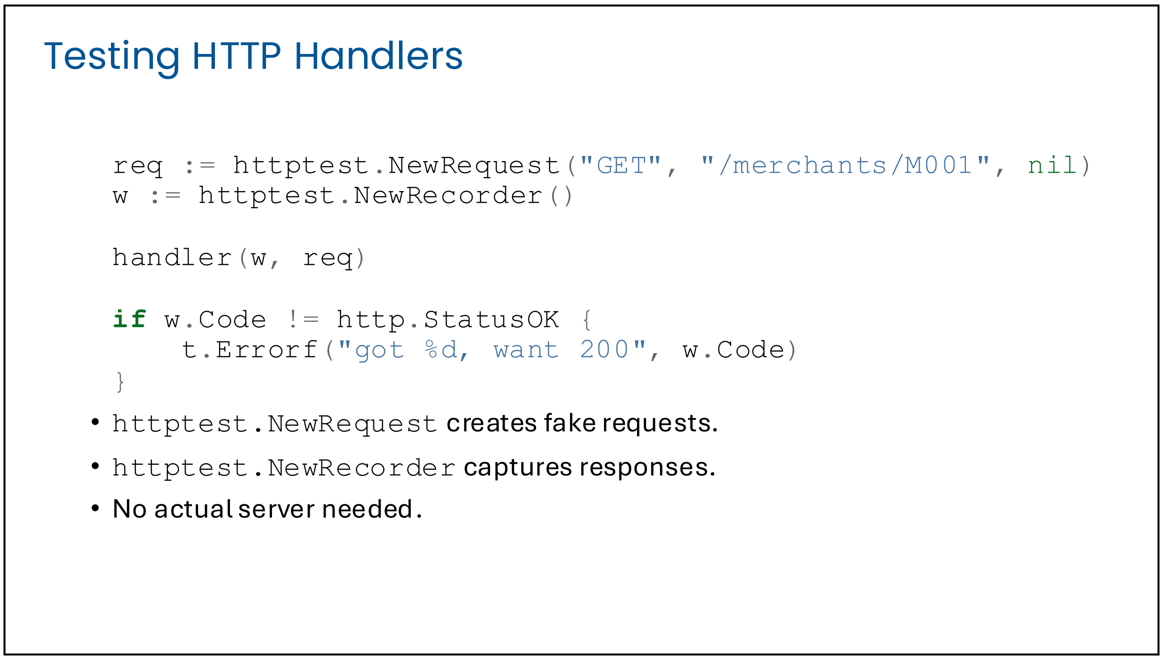


Table-Driven Tests

```go
tests := []struct {
    name    string
    input   *Merchant
    wantErr bool
}{
    {"valid", &Merchant{ID: "M001"}, false},
    {"empty ID", &Merchant{ID: ""}, true},
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        err := Validate(tt.input)
        if (err != nil) != tt.wantErr {
            t.Errorf("got %v", err)
        }
    })
}
```

- Table-driven tests are idiomatic in Go - use them everywhere.
- Slice of anonymous structs defines test cases.
- Each case has name, input, and expected output.
- Loop uses t.Run() to create subtests.
- Failure messages show which case failed by name.
- Easy to add new cases without changing logic.
- Can vary inputs, outputs, setup per case.

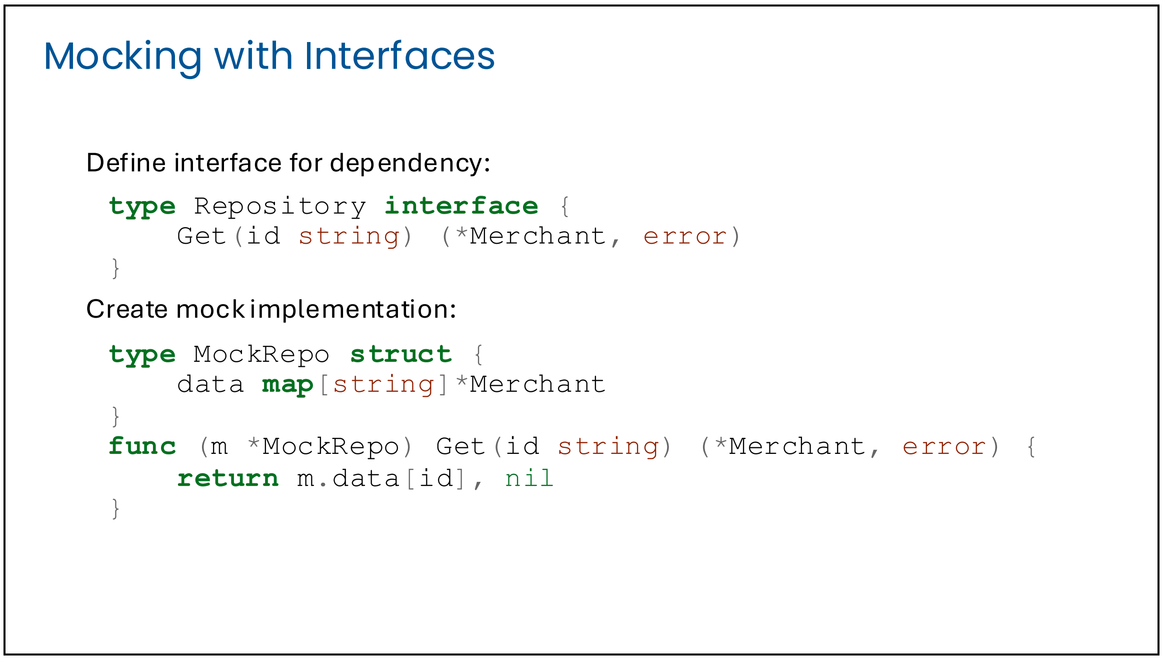


Testing HTTP Handlers

```go
req := httptest.NewRequest("GET", "/merchants/M001", nil)
w := httptest.NewRecorder()

handler(w, req)

if w.Code != http.StatusOK {
    t.Errorf("got %d, want 200", w.Code)
}
```

- `httptest.NewRequest` creates fake requests.
- `httptest.NewRecorder` captures responses.
- No actual server needed.

- httptest package makes HTTP testing trivial.
- NewRequest creates *http.Request without network.
- NewRecorder implements ResponseWriter and records output.
- Test status codes, headers, body content easily.
- No actual HTTP server - just call handler function directly.
- Similar to Spring’s MockMvc but simpler.



Mocking with Interfaces

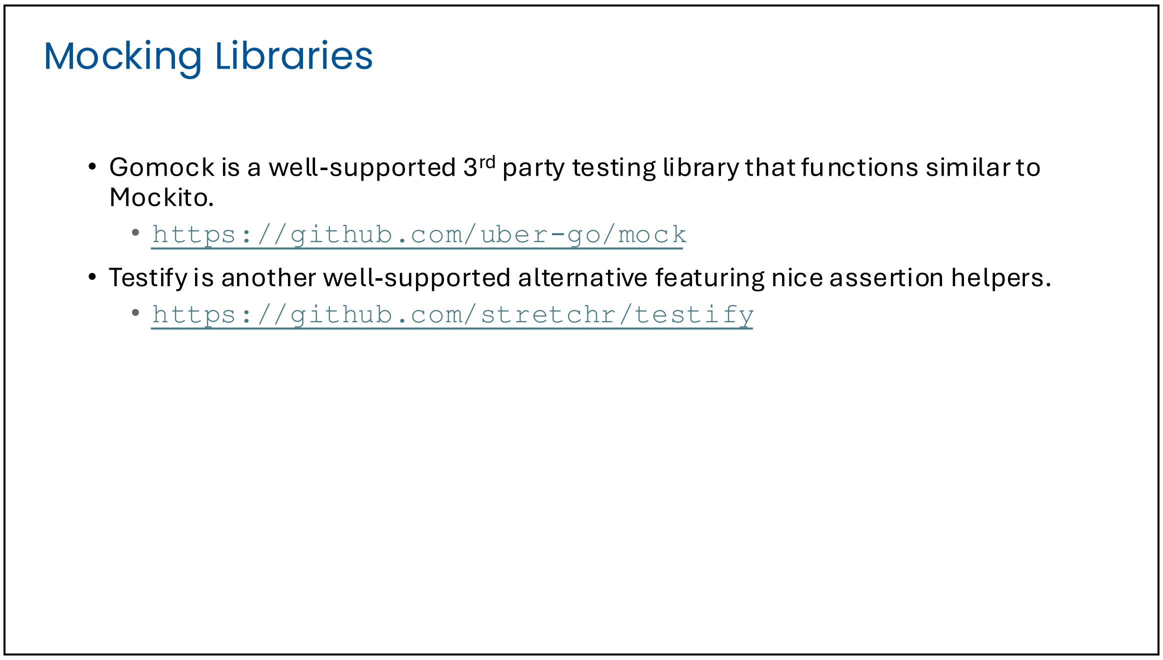
Define interface for dependency:

```go
type Repository interface {
    Get(id string) (*Merchant, error)
}
```

Create mock implementation:

```go
type MockRepo struct {
    data map[string]*Merchant
}
func (m *MockRepo) Get(id string) (*Merchant, error) {
    return m.data[id], nil
}
```

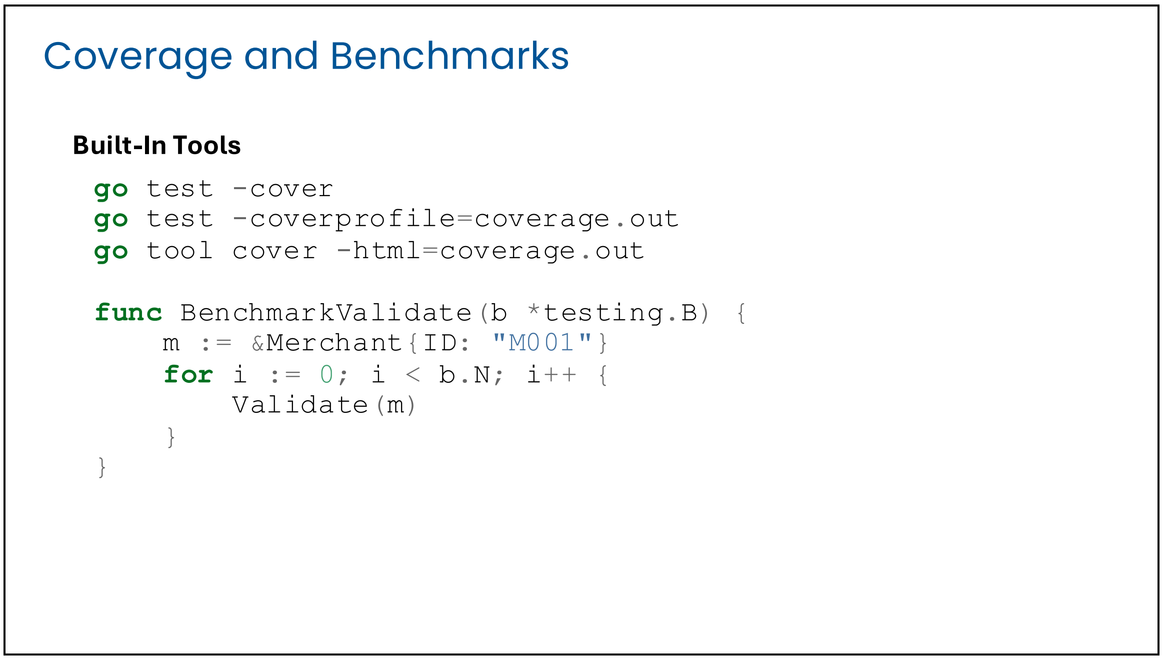
No mocking framework required - just implement the interface.
Mock is just another implementation you control.
Can return specific errors, test edge cases.
Compare to Mockito which uses reflection and proxies.
Small interfaces make mocking easy.
This is why interface-based design matters.



Mocking Libraries

- Gomock is a well-supported 3rd party testing library that functions similar to Mockito.
  - https://github.com/uber-go/mock
- Testify is another well-supported alternative featuring nice assertion helpers.
  - https://github.com/stretchr/testify

Third-party tools exist (gomock, testify/mock) but often unnecessary.



Coverage and Benchmarks

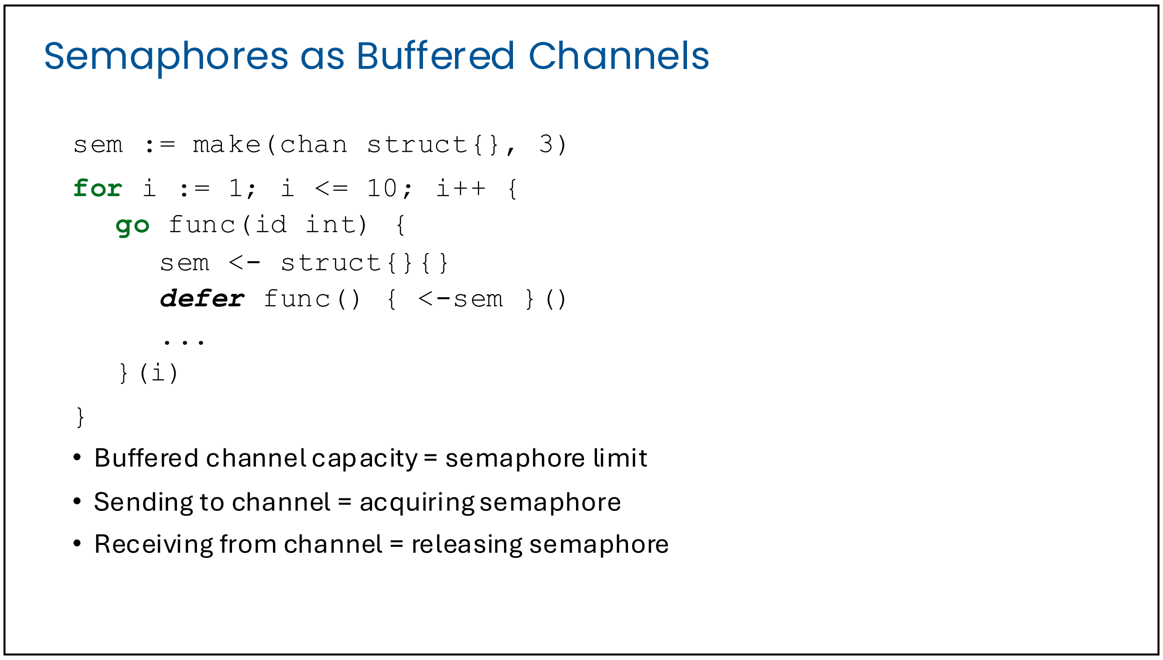
Built-In Tools

```
go test -cover
go test -coverprofile=coverage.out
go tool cover -html=coverage.out

func BenchmarkValidate(b *testing.B) {
    m := &Merchant{ID: "M001"}
    for i := 0; i < b.N; i++ {
        Validate(m)
    }
}
```

- Coverage built into go test - no plugins needed.
- HTML coverage report shows line-by-line coverage.
- Benchmarks use testing.B similar to testing.T.
- b.N automatically adjusted for stable timing.
- go test -bench runs benchmarks.
- go test -benchmem shows memory allocations.

Lab 6

Give students the time to return for debrief.



Semaphores as Buffered Channels

```go
sem := make(chan struct{}, 3)
for i := 1; i <= 10; i++ {
   go func(id int) {
      sem <- struct{}{}
      defer func() { <-sem }()
      ...
   }(i)
}
```

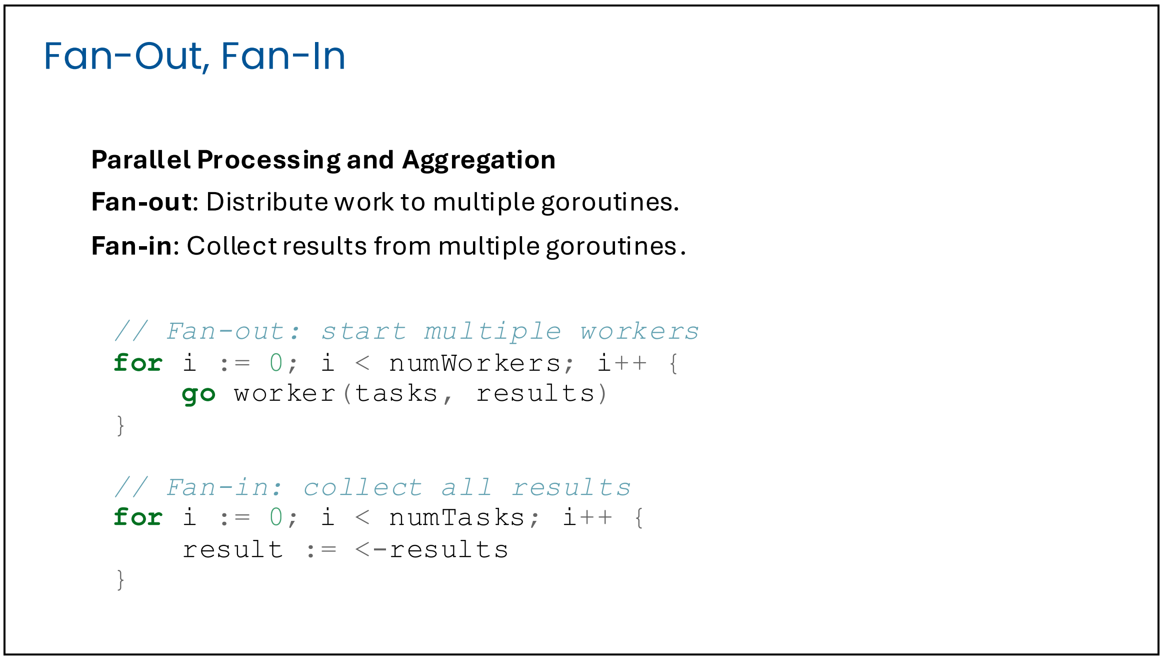
- Buffered channel capacity = semaphore limit
- Sending to channel = acquiring semaphore
- Receiving from channel = releasing semaphore

- **How It Works**
- Channel blocks when full → limits concurrency
- Empty struct{} uses zero memory
- Defer ensures release even on panic
- **Common Use Cases**
- Rate limiting API calls
- Limiting concurrent database connections
- Controlling worker pool size
- Preventing resource exhaustion

Worker Pool Pattern

```
// Java
ExecutorService executor = Executors.newFixedThreadPool(numWorkers);
executor.submit(() -> process(tx));
executor.shutdown();
executor.awaitTermination(timeout, TimeUnit.SECONDS);

// Golang
for w := 0; w < numWorkers; w++ {
        go func() {
             for tx := range jobs {
                    process(tx)
             }
        }
}
```

- Common pattern for processing many tasks:
- Worker pool limits concurrent operations.
- Create N workers, send M tasks through channel.
- Workers pull tasks from channel, process, repeat.
- Similar to Java’s ExecutorService with fixed thread pool.
- But simpler - just goroutines reading from channel.
- Prevents resource exhaustion from too many goroutines.
- Good for rate-limiting external API calls.
- Pattern used in production for job processing systems.



Fan-Out, Fan-In

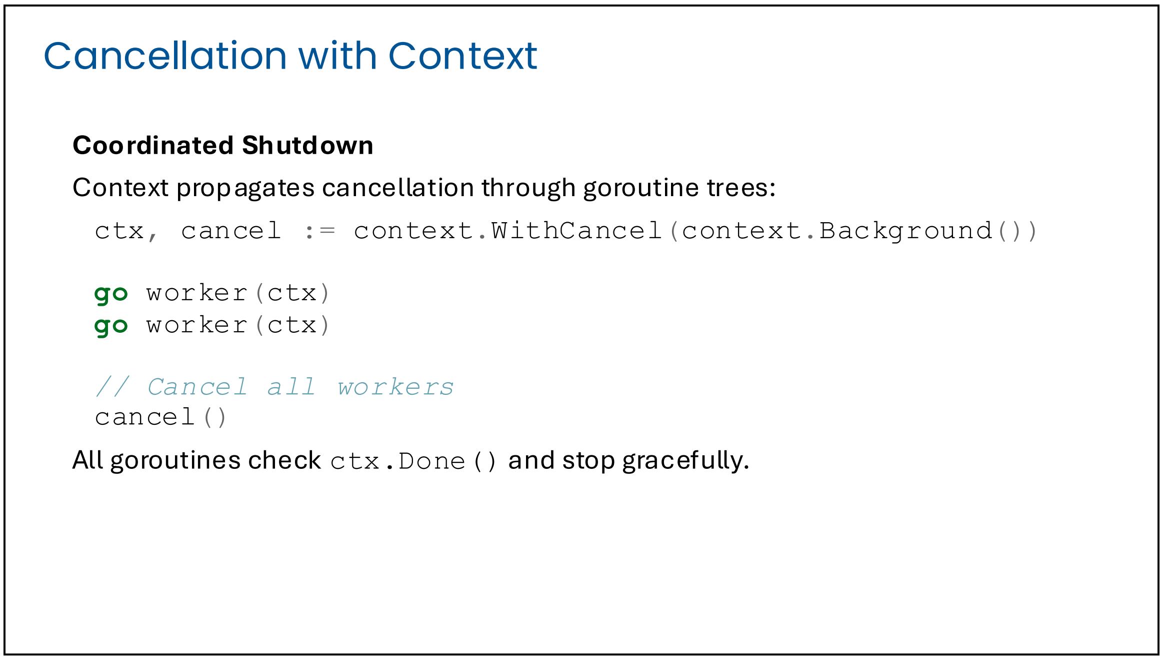
Parallel Processing and Aggregation

Fan-out: Distribute work to multiple goroutines.

Fan-in: Collect results from multiple goroutines.

```go
// Fan-out: start multiple workers
for i := 0; i < numWorkers; i++ {
    go worker(tasks, results)
}

// Fan-in: collect all results
for i := 0; i < numTasks; i++ {
    result := <-results
}
```

Fan-out: split work across many goroutines for parallel execution.
Fan-in: merge results from many channels into one.
Common for parallel computation (map-reduce style)
Each worker processes independently.
Results aggregated at the end.
Important pattern for maximizing CPU utilization.



Cancellation with Context

Coordinated Shutdown

Context propagates cancellation through goroutine trees:

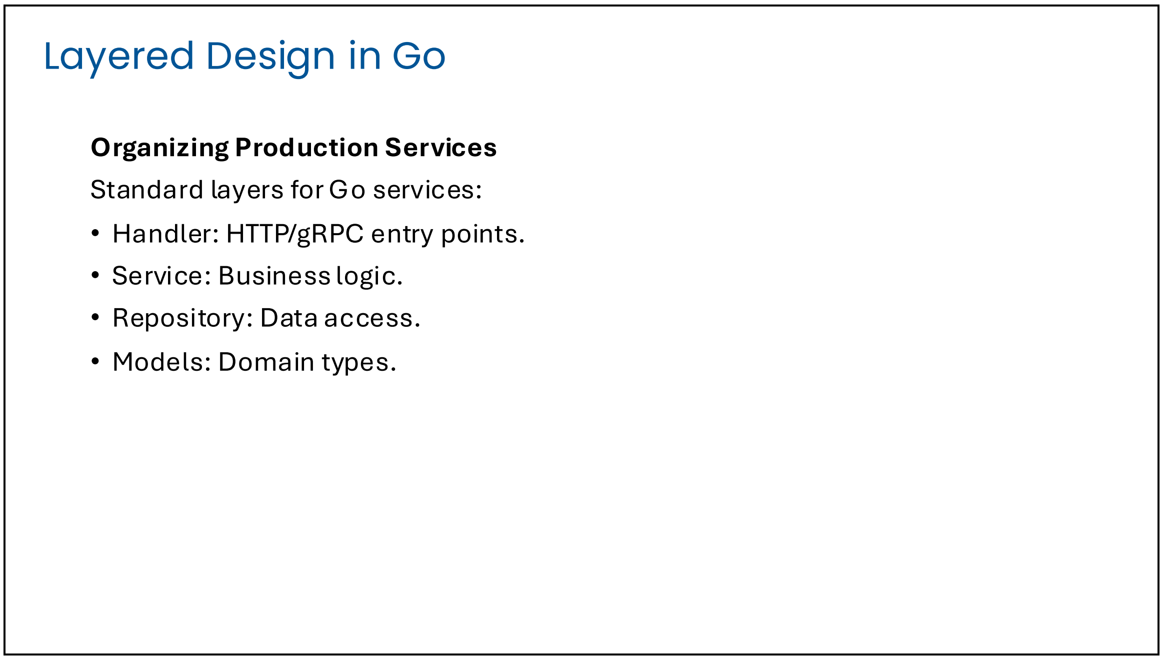
```
ctx, cancel := context.WithCancel(context.Background())

go worker(ctx)
go worker(ctx)

// Cancel all workers
cancel()
```

All goroutines check `ctx.Done()` and stop gracefully.

- When you cancel parent context, all children contexts cancelled.
- Each goroutine checks ctx.Done() in their loop.
- Enables coordinated shutdown of complex systems.
- Prevents goroutine leaks when operations no longer needed.
- Compare to Java’s interrupt mechanism but more composable.

Lab 7

Give students the time to return for debrief.



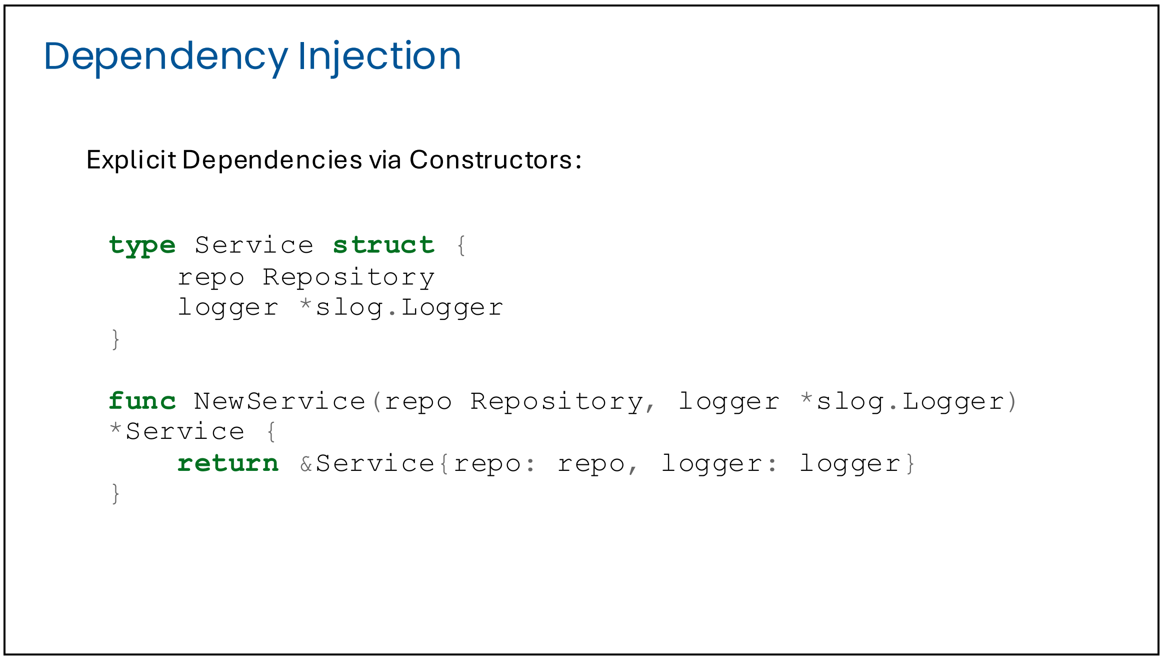
Layered Design in Go

Organizing Production Services

Standard layers for Go services:

- Handler: HTTP/gRPC entry points.
- Service: Business logic.
- Repository: Data access.
- Models: Domain types.

- Similar to hexagonal architecture / clean architecture principles.
- Handler layer deals with HTTP/gRPC specifics.
- Service layer contains business rules, no HTTP knowledge.
- Repository abstracts data storage (interfaces for testability)
- Models/domain types shared across layers.
- Dependencies point inward: handler -> service -> repository.
- Makes testing easy: mock repository, test service layer.

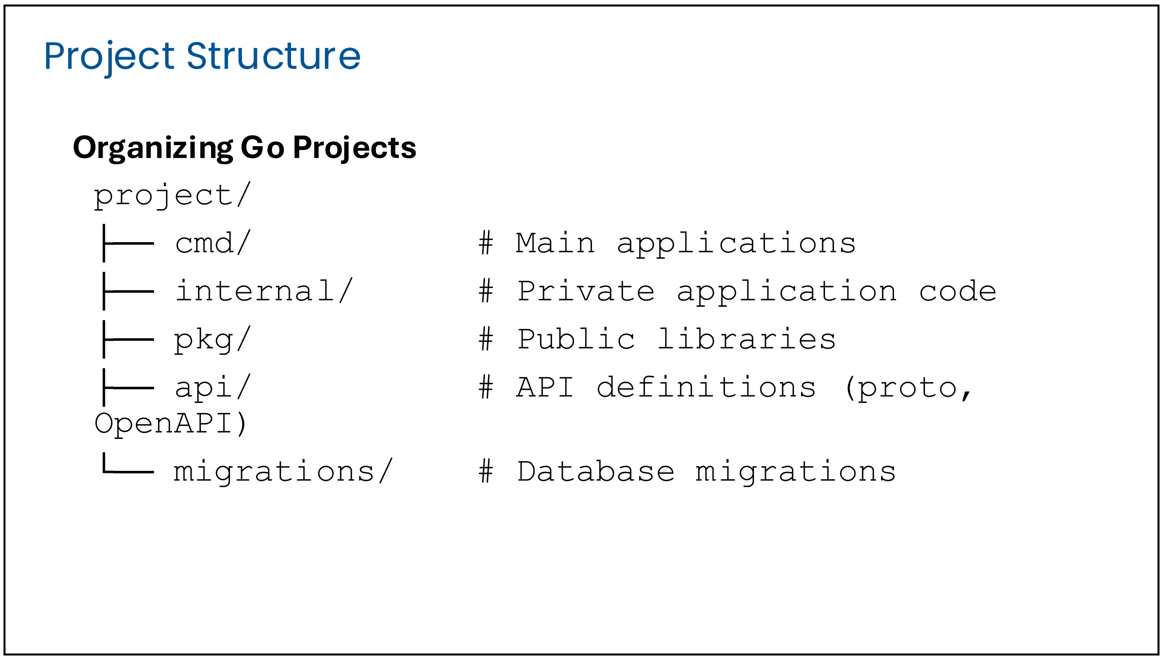


Dependency Injection

Explicit Dependencies via Constructors:

```
type Service struct {
    repo Repository
    logger *slog.Logger
}

func NewService(repo Repository, logger *slog.Logger)
*Service {
    return &Service{repo: repo, logger: logger}
}
```

- No Spring-style DI framework - just constructor injection.
- Dependencies passed explicitly as parameters.
- Store as struct fields.
- Makes dependencies visible and testable.
- Can easily swap implementations (real vs mock)
- Constructor functions ensure valid initialization.
- Students learn DI doesn’t require magic frameworks.

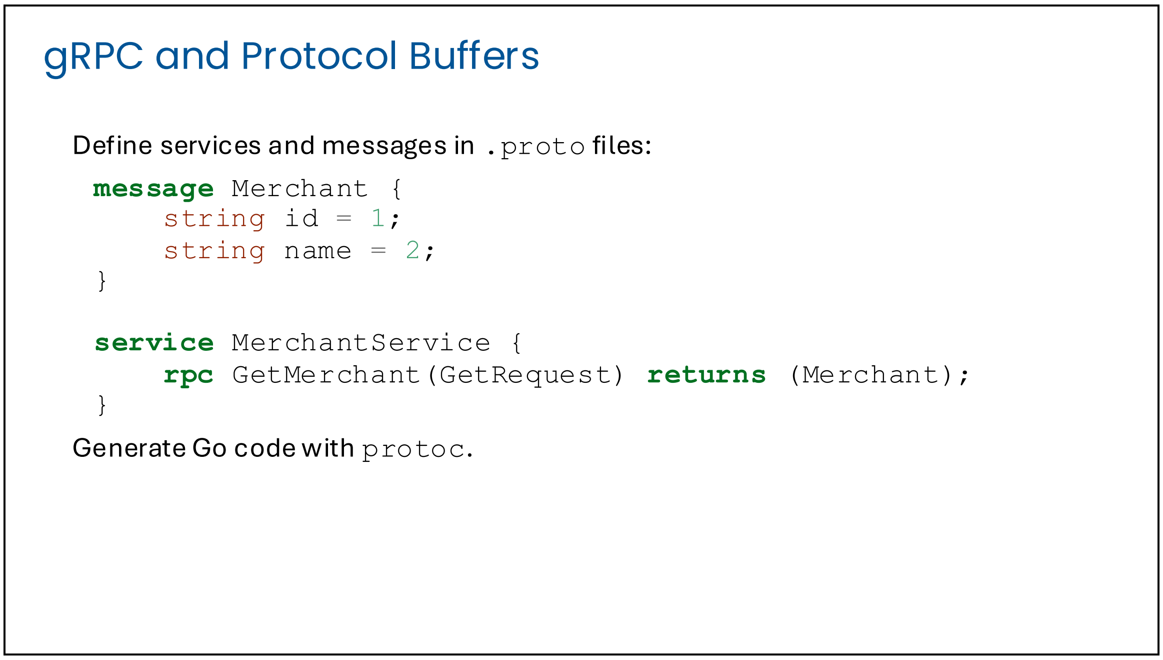


Project Structure

Organizing Go Projects

```
project/
├── cmd/            # Main applications
├── internal/       # Private application code
├── pkg/            # Public libraries
├── api/            # API definitions (proto,
OpenAPI)
└── migrations/     # Database migrations
```

- cmd/ for main packages - each subdirectory is a binary.
- internal/ prevents imports by other projects.
- pkg/ for code you’d be okay with others importing.
- Flat structure preferred over deep nesting.
- Compare to Java’s src/main/java hierarchy - simpler.
- Go prefers packages by feature, not by layer.
- No strict rules - teams adapt based on size.



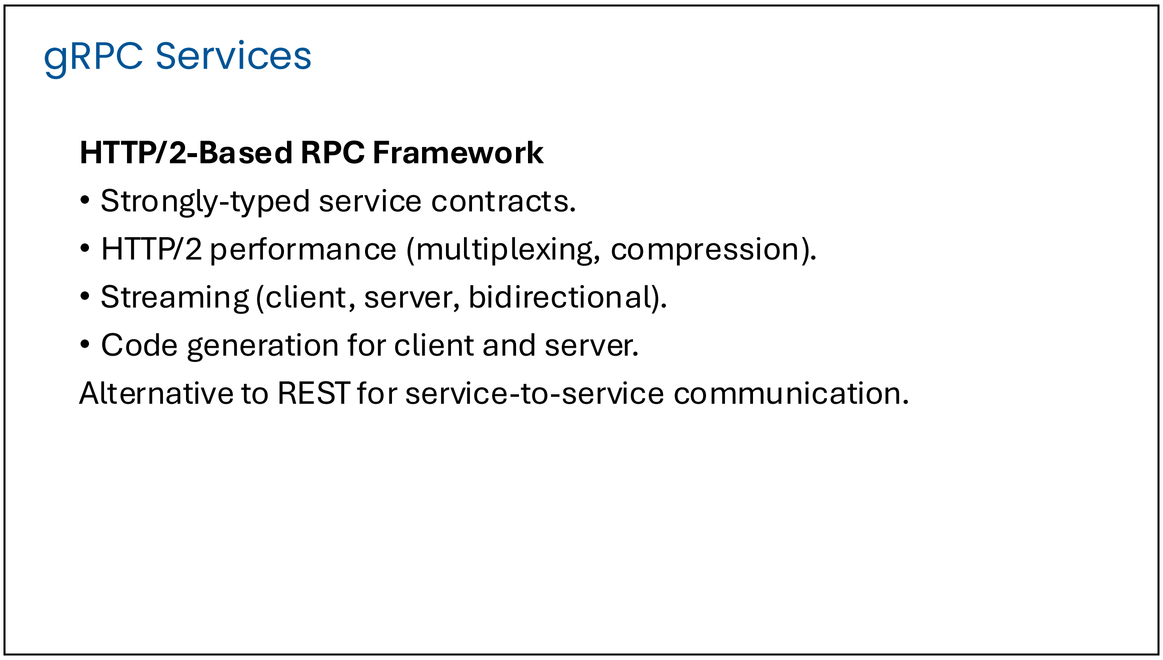
gRPC and Protocol Buffers

Define services and messages in `.proto` files:

```
message Merchant {
    string id = 1;
    string name = 2;
}

service MerchantService {
    rpc GetMerchant(GetRequest) returns (Merchant);
}
```

Generate Go code with `protoc`.

- Protocol Buffers: Google’s serialization format.
- More efficient than JSON (binary, smaller, faster)
- Define once, generate for multiple languages.
- Backwards compatible through field numbers.
- protoc compiler generates Go structs and gRPC code.
- Strongly typed - catches errors at compile time.
- Used heavily in microservices architectures.



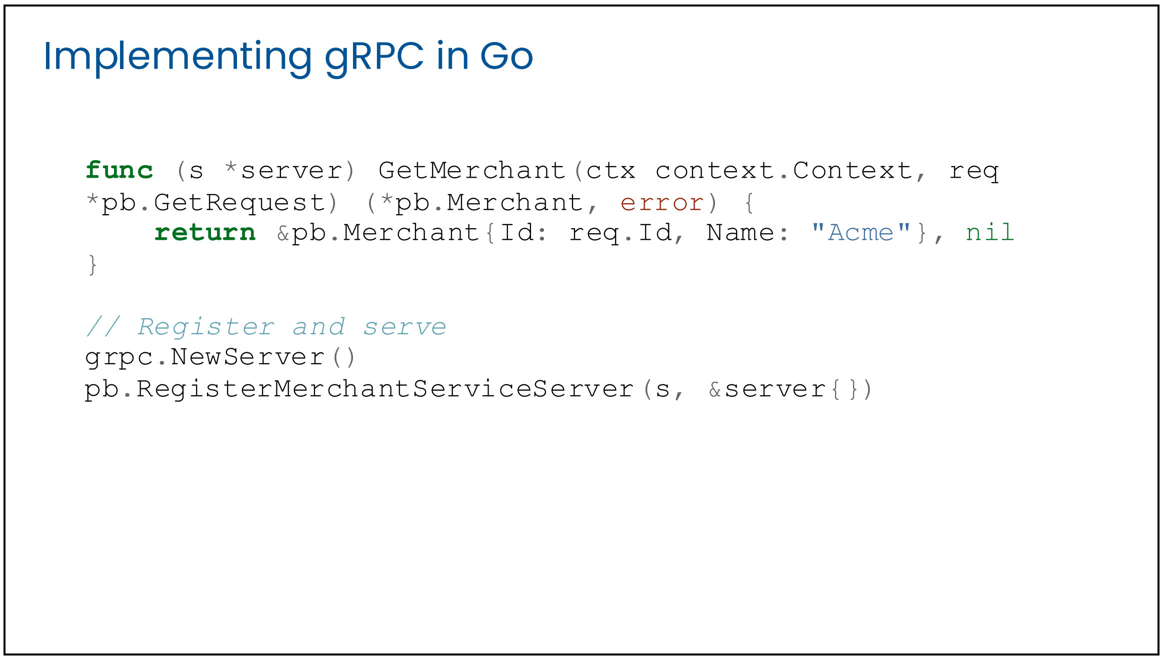
gRPC Services

HTTP/2-Based RPC Framework

- Strongly-typed service contracts.
- HTTP/2 performance (multiplexing, compression).
- Streaming (client, server, bidirectional).
- Code generation for client and server.

Alternative to REST for service-to-service communication.

gRPC: Google’s RPC framework, now CNCF project.
Built on HTTP/2 for efficiency.
Four modes: unary, client streaming, server streaming, bidirectional.
Automatic code generation from proto files.
Better performance than REST/JSON for high-throughput services.
Works across languages - Java can call Go service easily.
Common in cloud-native architectures (Kubernetes, service meshes).



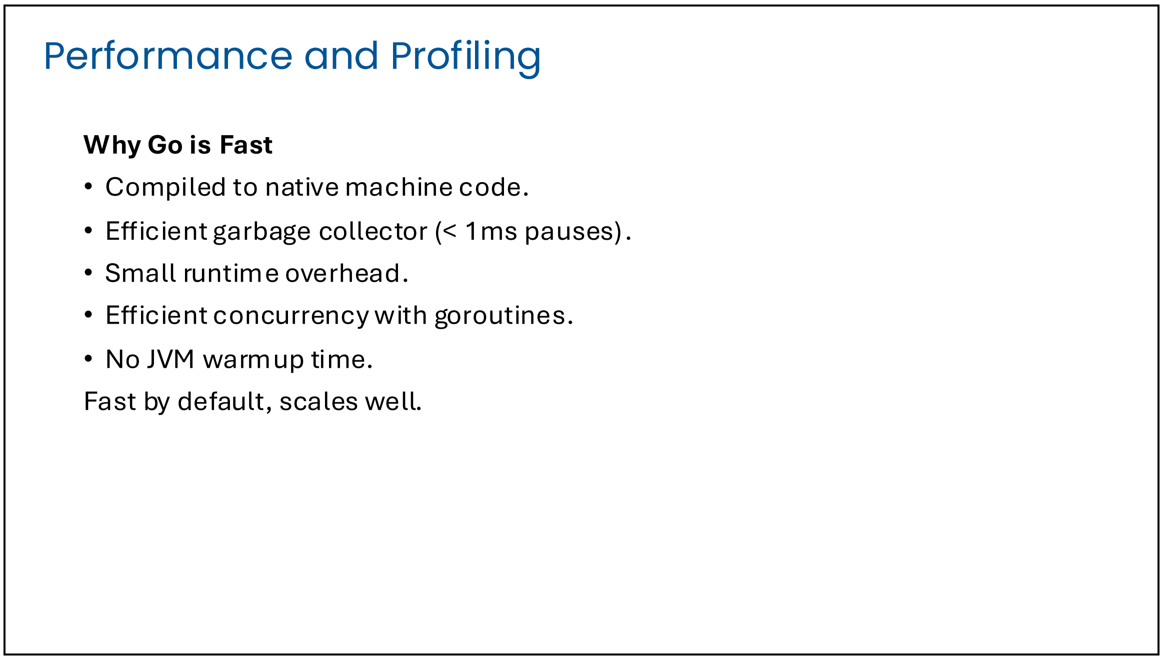
Implementing gRPC in Go

```go
func (s *server) GetMerchant(ctx context.Context, req
*pb.GetRequest) (*pb.Merchant, error) {
    return &pb.Merchant{Id: req.Id, Name: "Acme"}, nil
}

// Register and serve
grpc.NewServer()
pb.RegisterMerchantServiceServer(s, &server{})
```

- Implement interface generated from proto file.
- Context is first parameter (standard Go pattern)
- Return protobuf message types.
- grpc.NewServer() creates server.
- Register your service implementation.

Lab 8

Give students the time to return for debrief.



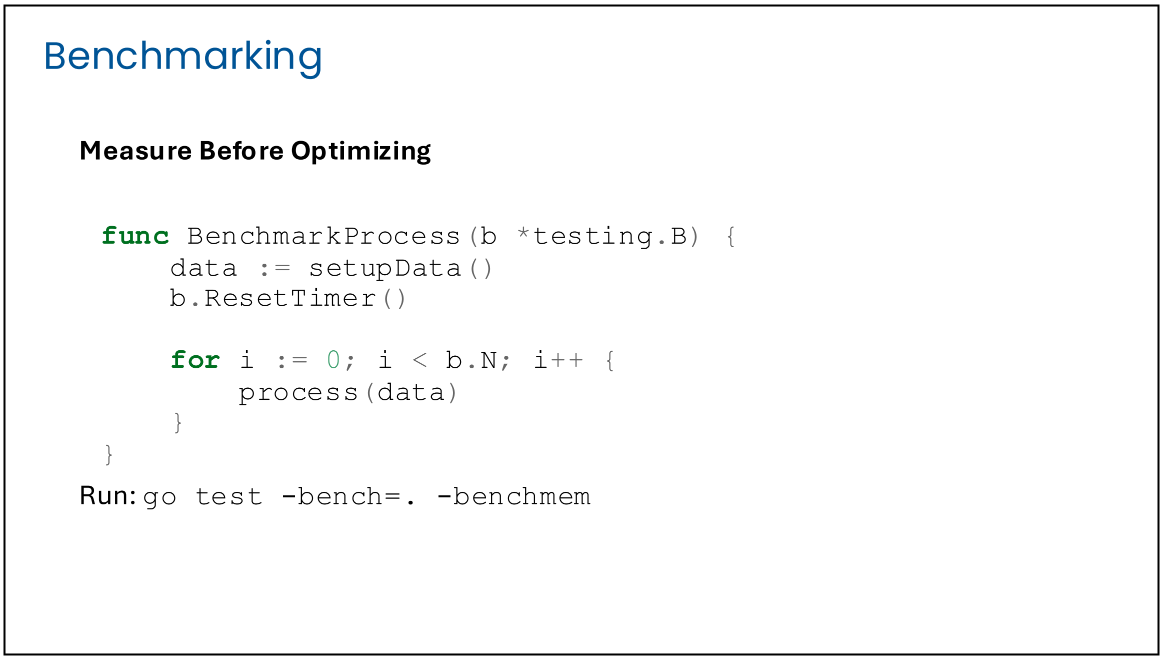
Performance and Profiling

Why Go is Fast

- Compiled to native machine code.
- Efficient garbage collector (< 1ms pauses).
- Small runtime overhead.
- Efficient concurrency with goroutines.
- No JVM warmup time.

Fast by default, scales well.

Compiles directly to machine code - no JVM interpretation.
GC optimized for low latency (sub-millisecond pauses common)
Small binaries include only what’s used.
Goroutines more efficient than OS threads.
Starts fast - no warmup period like JVM.
Compare to Java which optimizes over time but starts slower.
Go’s performance more predictable - less variance.
Good enough for most use cases without optimization.
When you need more: profiling tools built-in.



Benchmarking

Measure Before Optimizing

```
func BenchmarkProcess(b *testing.B) {
    data := setupData()
    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        process(data)
    }
}
```

Run: `go test -bench=. -benchmem`

Benchmarks use testing.B from stdlib.
b.N automatically adjusted for stable results.
b.ResetTimer() excludes setup time.
-benchmem shows memory allocations.
Compare iterations per second and bytes allocated.
Can benchmark against different implementations.
No need for JMH or external tools.
Results show ns/op and allocs/op.

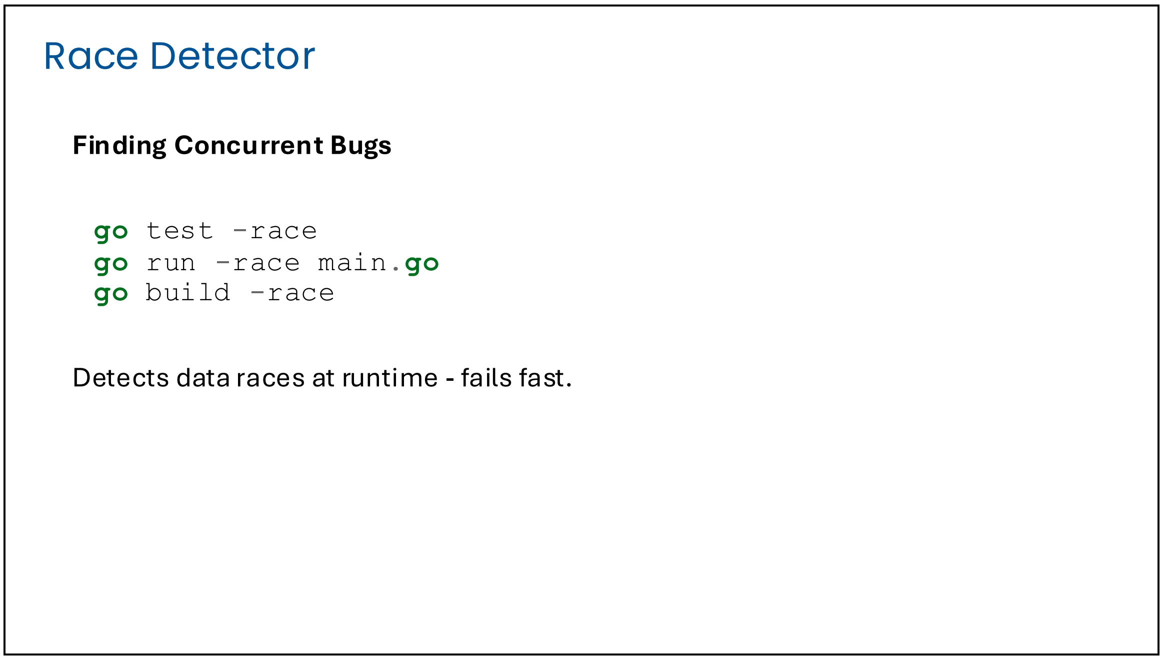
pprof - Built-In Profiler

```go
import _ "net/http/pprof"

go func() {
    http.ListenAndServe("localhost:6060", nil)
}()
```

Run:

```
go tool pprof http://localhost:6060/debug/pprof/profile
```

- pprof built into Go - no separate profiler needed.
- Import net/http/pprof registers handlers automatically.
- Can profile: CPU, memory, goroutines, blocking.
- Visualize with flame graphs, call graphs.
- Similar to Java's VisualVM or JProfiler but built-in.
- Can profile production systems (minimal overhead)
- Heap profiling finds memory leaks.
- Goroutine profiling finds leaks or blocked goroutines.



Race Detector

Finding Concurrent Bugs

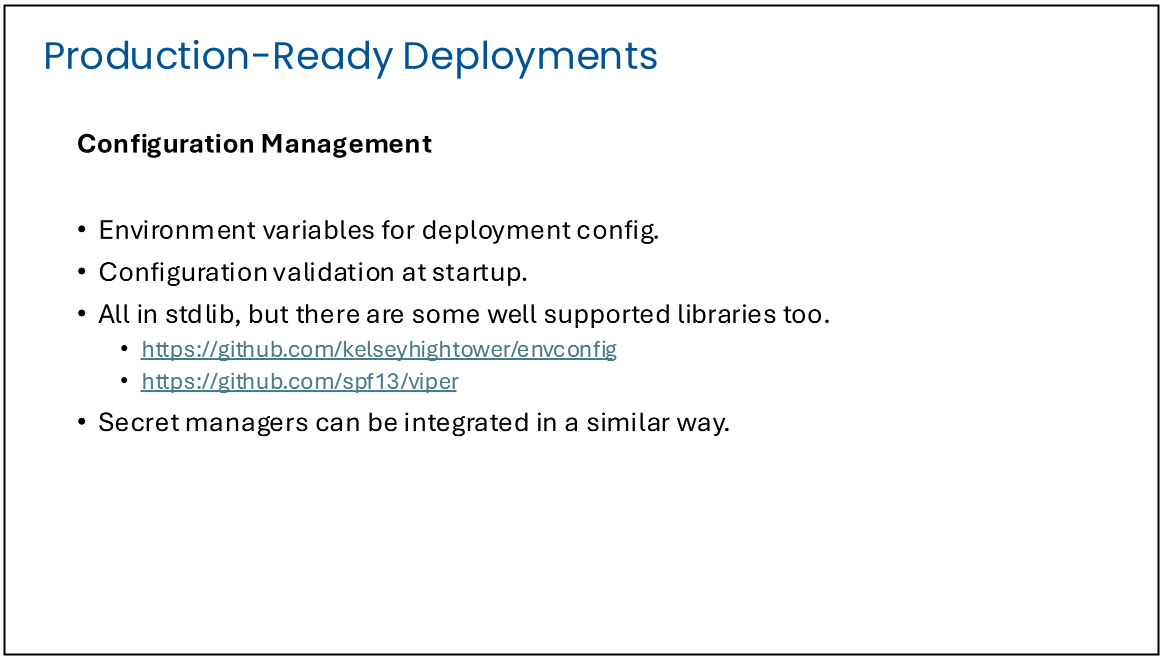
```
go test -race
go run -race main.go
go build -race
```

Detects data races at runtime - fails fast.

- Race detector finds concurrent access to shared memory.
- Works by instrumenting code during compilation.
- Detects actual races during test execution.
- Reports exact file and line numbers of conflicting accesses.
- No equivalent in Java - requires tools like ThreadSanitizer.
- Overhead: 10x slower, 10x more memory (only use in testing)
- Should run all tests with -race in CI.
- Catches bugs that are hard to find otherwise.

Lab 9

Give students the time to return for debrief.

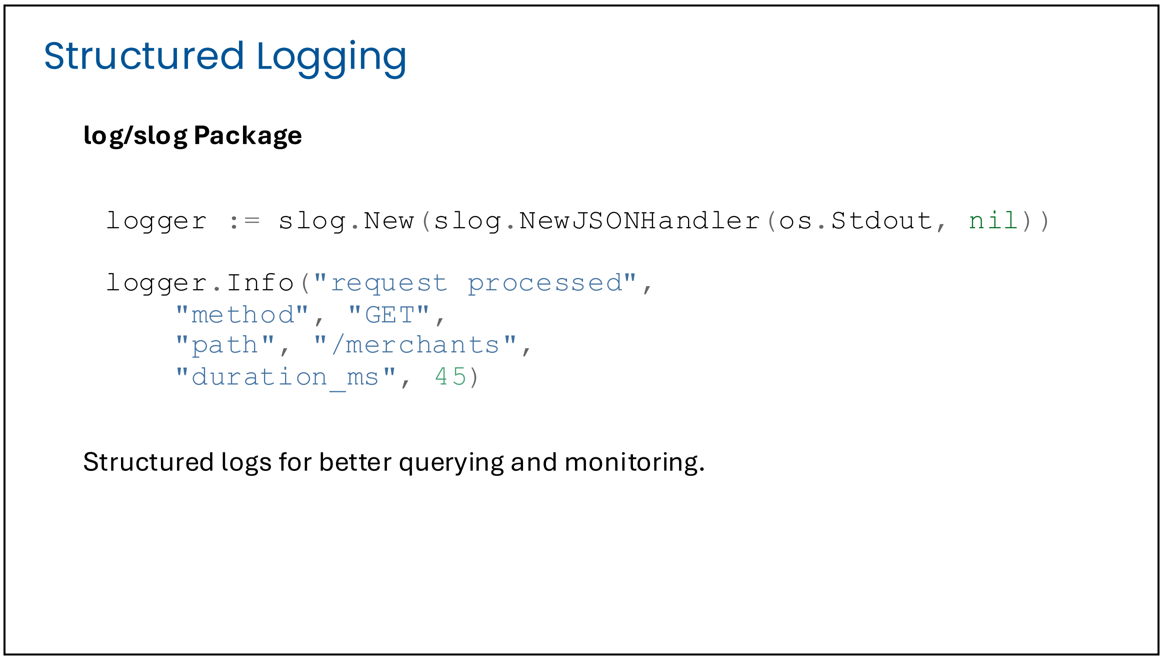


Production-Ready Deployments

Configuration Management

- Environment variables for deployment config.
- Configuration validation at startup.
- All in stdlib, but there are some well supported libraries too.
  - https://github.com/kelseyhightower/envconfig
  - https://github.com/spf13/viper
- Secret managers can be integrated in a similar way.

- 12-factor app principle: config in environment.
- Use libraries like envconfig or viper for parsing.
- Validate all required config at startup - fail if missing.
- Use secret managers (AWS Secrets Manager, Vault) in a similar way. Using the proper SDK.
- Compare to Java's application.properties or YAML configs.



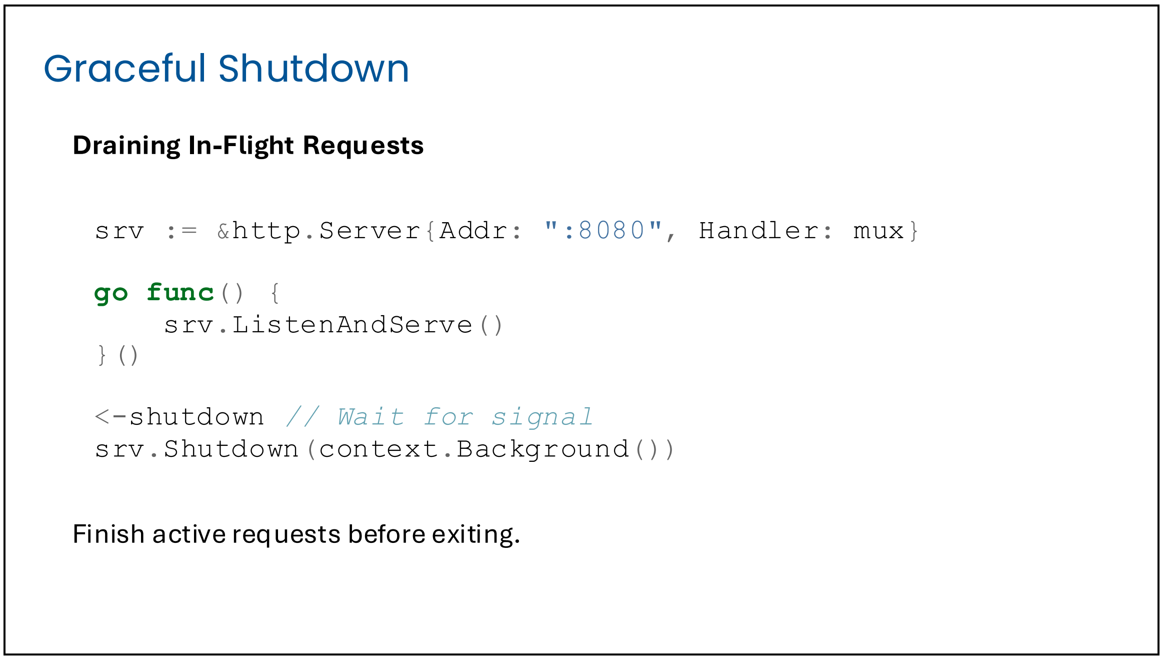
Structured Logging

log/slog Package

```go
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))

logger.Info("request processed",
    "method", "GET",
    "path", "/merchants",
    "duration_ms", 45)
```

Structured logs for better querying and monitoring.

- log/slog added in Go 1.21 - modern structured logging.
- Key-value pairs instead of printf-style strings.
- JSON output for parsing by log aggregators.
- Levels: Debug, Info, Warn, Error.
- Context integration: pass logger through context.
- Compare to Java's SLF4J + Logback.
- Structured logs essential for production debugging.
- Can query by fields in log aggregation systems.



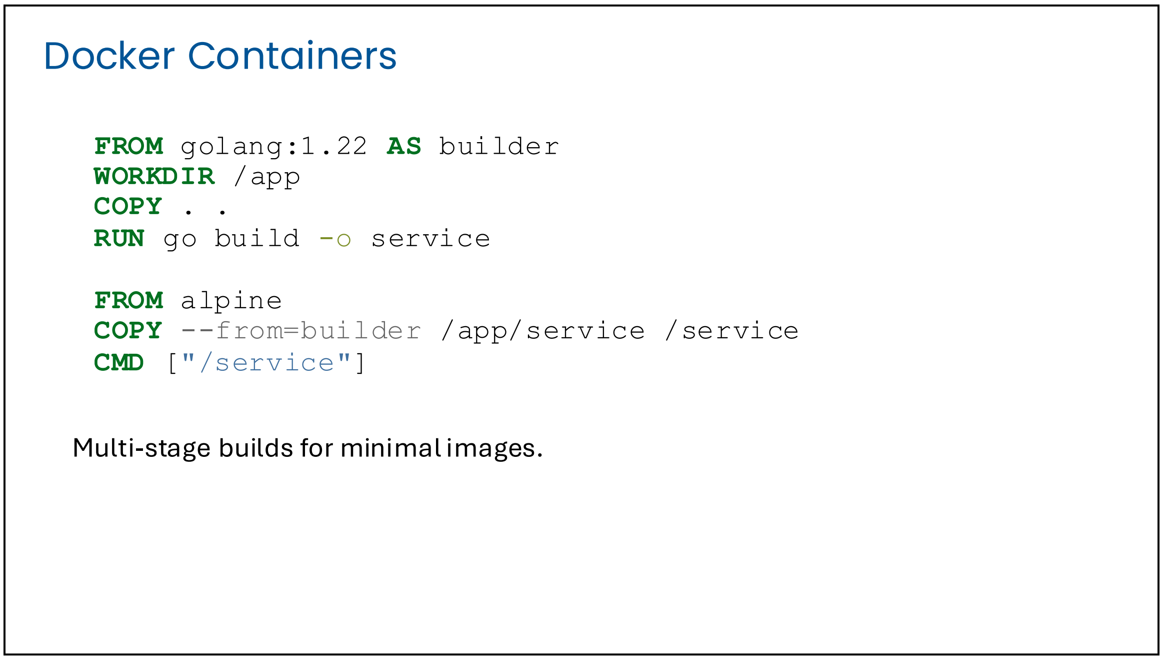
Graceful Shutdown

Draining In-Flight Requests

```go
srv := &http.Server{Addr: ":8080", Handler: mux}

go func() {
    srv.ListenAndServe()
}()

<-shutdown // Wait for signal
srv.Shutdown(context.Background())
```

Finish active requests before exiting.

- Graceful shutdown: stop accepting new requests, finish existing ones.
- http.Server has Shutdown() method for this.
- Pass context to control shutdown timeout.
- Important for zero-downtime deployments.
- In Kubernetes: respects SIGTERM signal.

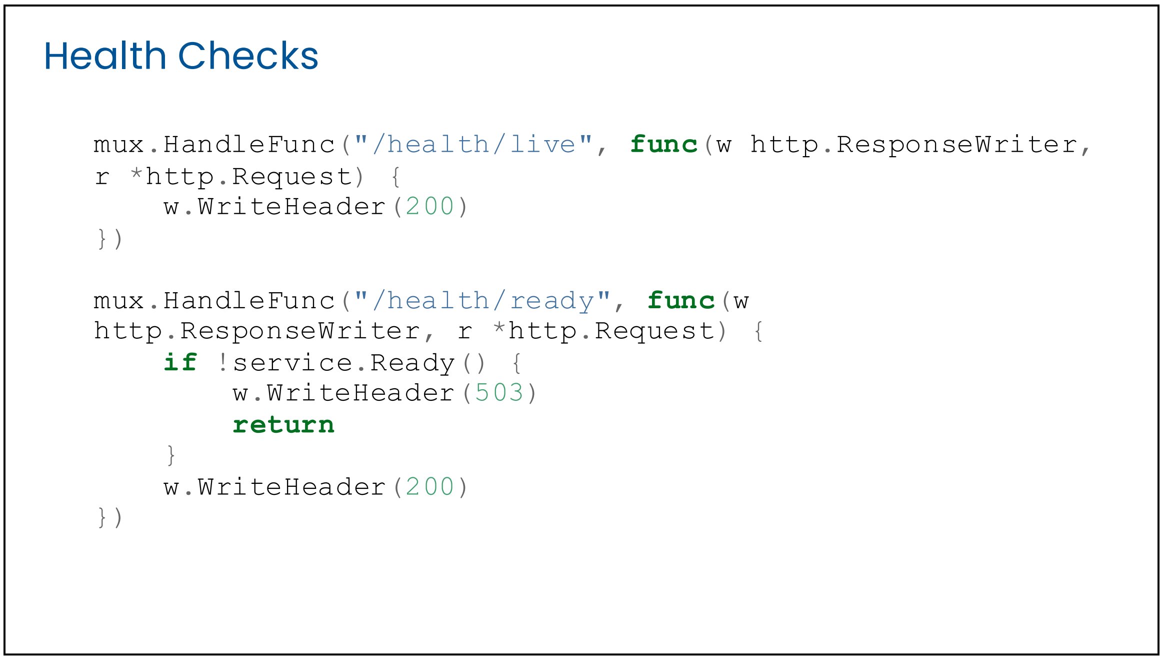


Docker Containers

```
FROM golang:1.22 AS builder
WORKDIR /app
COPY . .
RUN go build -o service

FROM alpine
COPY --from=builder /app/service /service
CMD ["/service"]
```

Multi-stage builds for minimal images.

- Multi-stage: build in golang image, run in minimal alpine.
- Results in tiny images.
- No JVM needed - binary is self-contained.
- Can use distroless or scratch for even smaller images.
- Fast startup - no JVM initialization.
- Compare to Java: need full JRE in container.
- Static linking: binary includes everything.



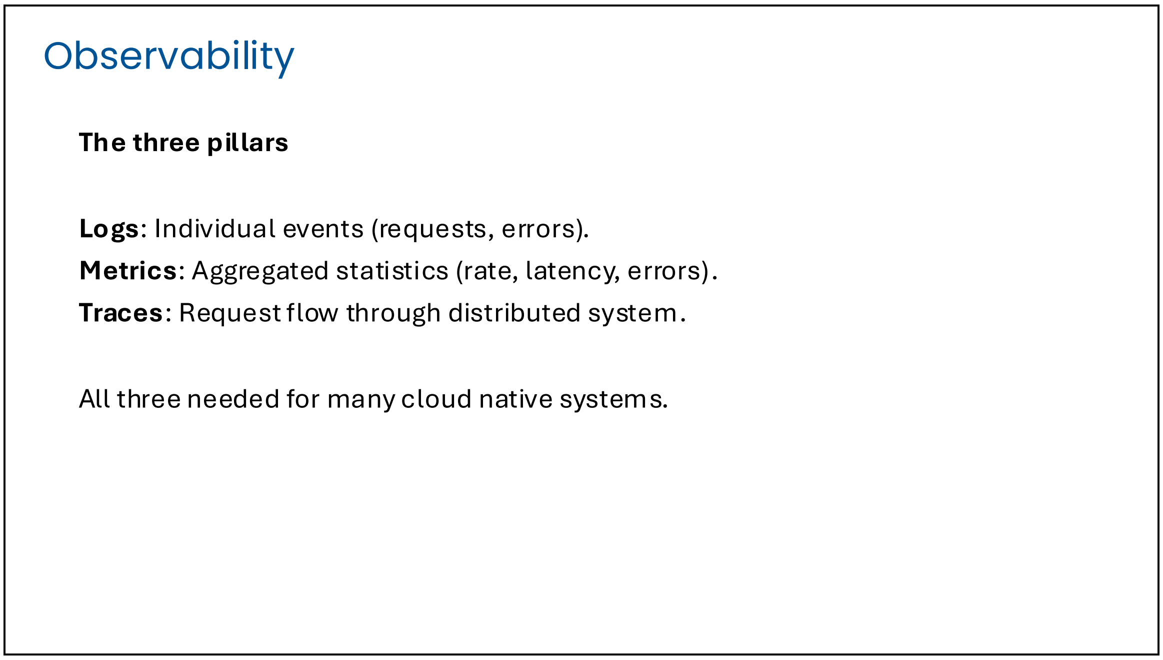
Health Checks

```
mux.HandleFunc("/health/live", func(w http.ResponseWriter,
r *http.Request) {
    w.WriteHeader(200)
})

mux.HandleFunc("/health/ready", func(w
http.ResponseWriter, r *http.Request) {
    if !service.Ready() {
        w.WriteHeader(503)
        return
    }
    w.WriteHeader(200)
})
```

- Liveness: is process alive? (200 = yes, kill if no)
- Readiness: can it handle traffic? (200 = ready, 503 = not yet)
- Kubernetes uses these for orchestration decisions.
- Liveness probe: restart if failing.
- Readiness probe: remove from load balancer if failing.
- Check dependencies in readiness (database, cache)
- Don’t check dependencies in liveness (or infinite restart loop)

Lab 10

Give students the time to return for debrief.



Observability

The three pillars

Logs: Individual events (requests, errors).
Metrics: Aggregated statistics (rate, latency, errors).
Traces: Request flow through distributed system.

All three needed for many cloud native systems.

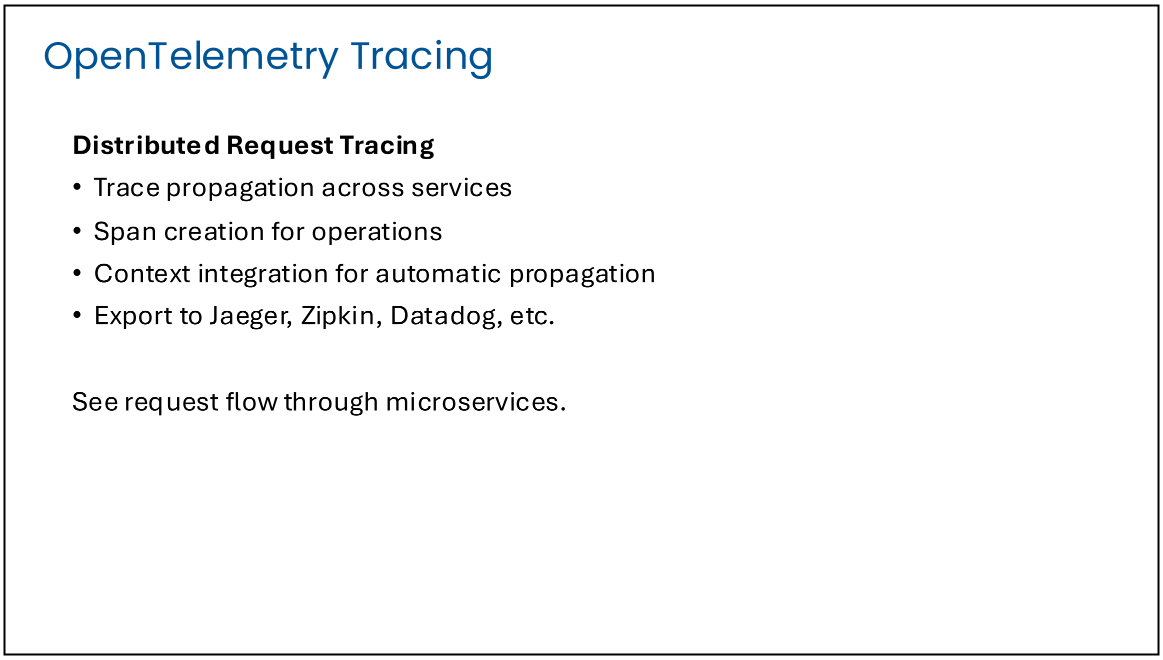
- Logs: what happened (structured logs with slog).
- Metrics: how many, how fast (Prometheus).
- Traces: where did request go (OpenTelemetry).
- Each pillar answers different questions.
- Logs for debugging specific requests.
- Metrics for dashboards and alerting.
- Traces for understanding distributed system behavior.
- Go has excellent support for all three.

Prometheus Metrics

```go
import "github.com/prometheus/client_golang/prometheus"

var requestCounter = prometheus.NewCounterVec(
    prometheus.CounterOpts{Name: "requests_total"},
    []string{"method", "path"},
)

requestCounter.WithLabelValues("GET", "/merchants").Inc()
```

Prometheus scrapes `/metrics` endpoint.

- Prometheus: industry-standard metrics system.
- Four metric types: Counter, Gauge, Histogram, Summary.
- Labels for dimensionality (method, status, path)
- Metrics exported via HTTP endpoint.
- Prometheus server scrapes periodically.
- Use for: request rates, error rates, latency, resource usage.
- Compare to Java's Micrometer or Dropwizard Metrics.
- Essential for production monitoring and alerting.



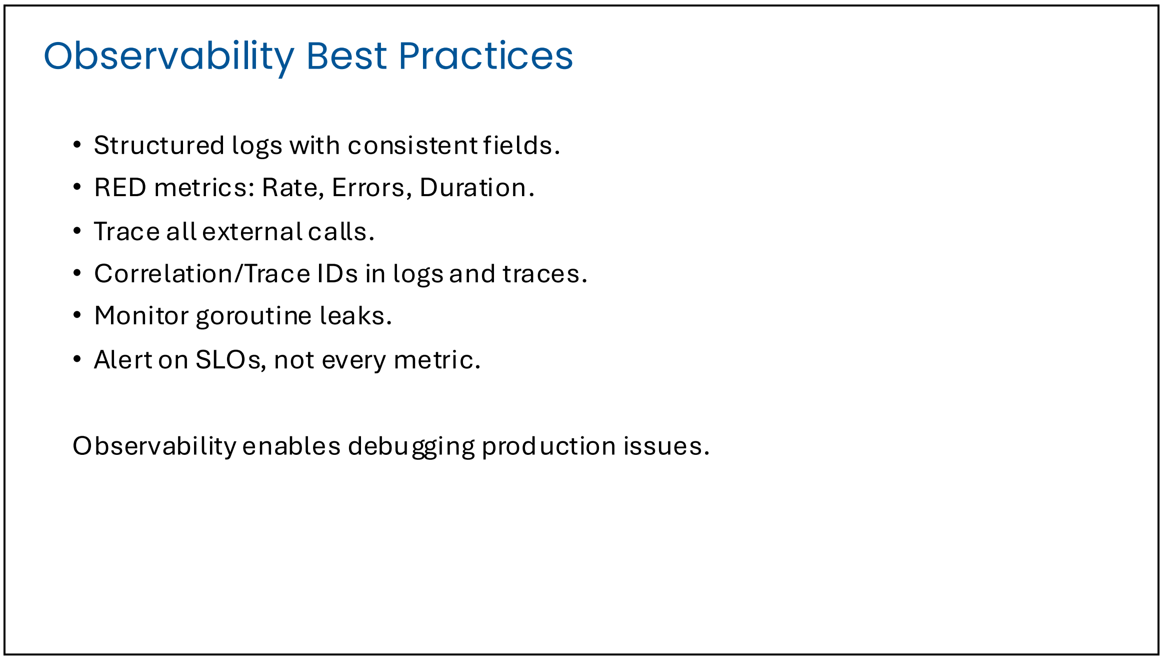
OpenTelemetry Tracing

Distributed Request Tracing

- Trace propagation across services
- Span creation for operations
- Context integration for automatic propagation
- Export to Jaeger, Zipkin, Datadog, etc.

See request flow through microservices.

- OpenTelemetry: CNCF standard for observability.
- Traces show request path through distributed system.
- Each operation is a span within a trace.
- Context carries trace ID automatically.
- Propagates across HTTP/gRPC boundaries.
- Visualize in Jaeger or other trace UIs.
- Essential for debugging microservices.
- Compare to Java's Sleuth or OpenTracing.



Observability Best Practices

- Structured logs with consistent fields.
- RED metrics: Rate, Errors, Duration.
- Trace all external calls.
- Correlation/Trace IDs in logs and traces.
- Monitor goroutine leaks.
- Alert on SLOs, not every metric.

Observability enables debugging production issues.

- RED metrics: requests/sec, error rate, duration (latency)
- Correlation ID: track single request through logs.
- Include correlation ID in all log statements.
- Use context to propagate trace and request IDs.
- Monitor goroutine count - leaks show as growth.
- Memory profiling for memory leaks.

Lab 11

Give students the time to return for debrief.

References and Resources

- https://go.dev/doc/effective_go
- https://go.dev/doc/tutorial/getting-started
- https://go.dev/blog/errors-are-values
- https://go.dev/play/

Recommended:

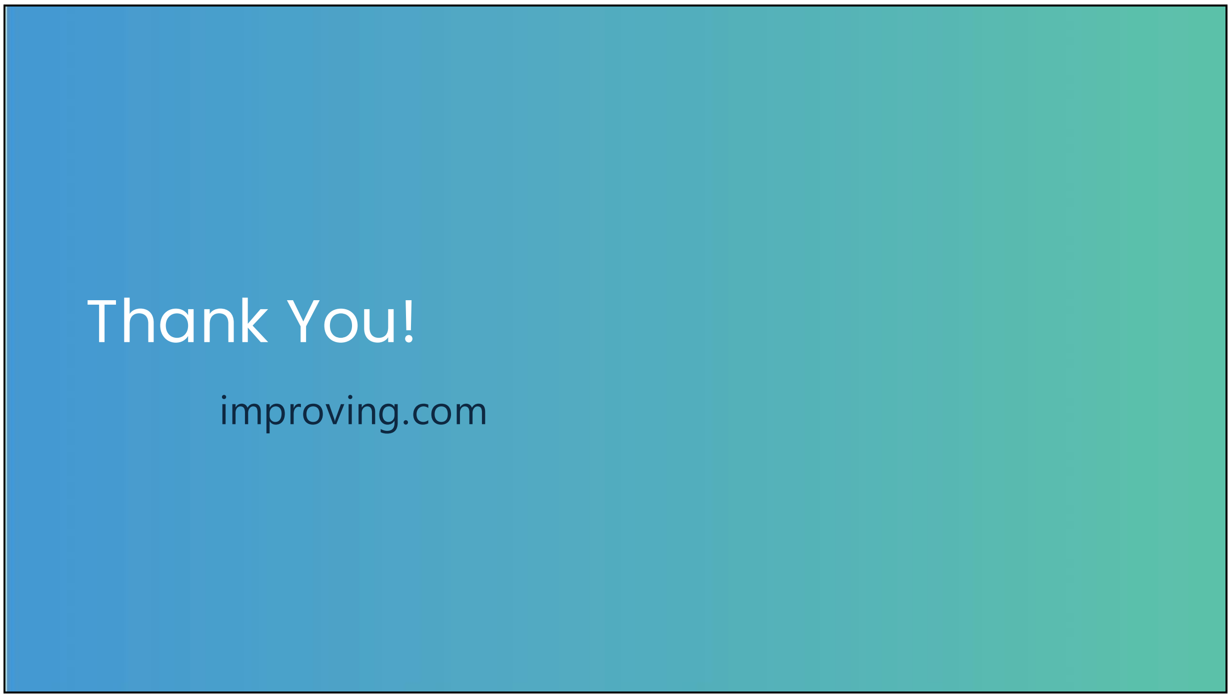
“Cloud Native Go” by Matthew A. Titmus

Thank You!

Feedback is important and we take it seriously. Your feedback helps us continually inspect and adapt our courses.

Course Name: **Golang Workshop**

Please share your feedback: https://www.surveymonkey.com/r/improving-training



Thank You!

improving.com