Welcome

Advanced Go

We teach over 400 technologies

Prerequisites

- Comfortable with basic concepts of Go
  - Variables, loops, logic, and concurrency mechanisms
- Familiarity with the most common parts of standard library
  - fmt, strings, errors, etc.
- Comfortable navigating standard library and adopting new parts of it

Objectives

- Reinforce your understanding of Go
- Explore best practices for solving many common problems
- Understand common gotchas with Go and how to avoid them
- Expand your understanding of the Go toolchain
- Use collaboration and code review to confirm your understanding

Administrative Issues

- Class time
- Breaks
- Lunch
- If you need to step away
  - Phone calls
  - Work emergencies
  - Family

Agenda

- Language review
- Generics
- Applied Go – Best Practices and Gotchas
- Modules
- Advanced error management
- Testing and profiling (including Fuzz tests!)
- Code generation
- Concurrency patterns
- The template subsystem

Course structure

Section 1

Section 2

Section 3

Section …

Course structure

- ❑Your name
- ❑Current role and time as a developer
- ❑Things you are especially interested in learning this week
- ❑Hobbies

DISCUSSION Time

Hello

About me...

- 20+ years of development experience
- 15+ years of training experience
- Live in Ohio with my wife, 2 daughters, 2 mouthy cats, and one very quiet Leopard gecko
- Hobbies
  - Exercise
  - Volunteering
  - Exploring the local parks
  - Binging Critical Role

Language Review

The Basics

Language Review

log Package

flag Package

Concurrency

Contexts

Topical Demos

Language Review

In this lab, you will be finishing the implementation of an asynchronous logging service. This service is intended to receive HTTP messages and write the request body's content out to a log file. You have been provided with the API and partial implementation of many functions and methods. Your goal is to finish the logging service and discuss design decisions that you made.

LAB

Applied Go: Best Practices

Interfaces

Outline

Using interfaces

Interface design

Using Interfaces

Interfaces

Topical Demos

Using Interfaces

Interfaces

```
type Encoder interface {
   Encode(in string) ([]byte, error)
}
type JSONEnc struct {}
func (j *JSONEnc) Encode(s string)
   ([]byte, error) {
   return json.Marshal(s)
}

var enc Encoder
enc = JSONEnc{}                 // fails
enc = &JSONEnc{}                // works
```

declare interface

Implement interface

Method sets don’t match
Correct method set

Method sets

```
type Encoder interface {
        Encode(in string) ([]byte, error)
}
type JSONEnc struct {}
func (j *JSONEnc) Encode(s string) ([]byte, error) {
        return json.Marshal(s)
}
```

```
func NewJSONEncoder()
Encoder{
        return JSONEnc{}
}
```

```
func NewJSONEncoder() Encoder{
  return &JSONEnc{}
}
```

Which is correct and why?

Method sets

Value

All methods with value receivers

Pointer

All methods with value receivers

All methods with pointer receivers

Interface

All methods

Designing Interfaces

Interfaces

Designing Interfaces

- Small interfaces are generally better
- Single method interfaces should be named with name plus “er”
  - String() -> Stringer interface
- Compose interfaces when needed
- Design packages to receive interfaces and return concrete types
  - More on this in a bit!

Topical Demos

Interfaces

```
type Encoder interface {
  Encode(reader io.Reader)
  []byte
}
type Decoder interface {
  Decode(
       target interface{},
       writer io.Writer
  )
}
type EncoderDecoder interface {
  Encoder
  Decoder
}
```

declare simple interface

declare second interface

Compose interfaces to model complex behavior

Generics

Topical Demos

Generics

Generics

```
func foo[T any]() { ... }


func bar[T any, S any]() {...}



func baz[T any](in T) T {
   return in
}

fmt.Printf("%T", baz(3))    // int
fmt.Printf("%T", baz(true))
       // bool

any
comparable
```

create a function with a generic parameter 'T'

can use multiple generic types per function
- each generic type must resolve to one type per function call

generics maintain type from consumer's perspective

matches any type, like interface{}
matches types that can be compared

Generics

```
type Addable interface {
  int | float64
}

func add[T Addable](){ ... }


type MyTypeInterface interface {
  Addable | string | io.Reader
}
```

create a type interface

used like other types as generic parameter

type interfaces and regular interfaces can be composed with other types

- Write a program that contains a generic function that allows all real number types to be added together without type conversions. In order to be as safe as possible, the result should always be returned as a float64

- The real number types are:
  - int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, float32, float64

- Examples:
  - add(3.14, 3) // 6.14
  - add(1, 2) // 3.0

LAB

Applied Go: Best Practices

Packages

Overview

Package Mechanics

Package-Oriented Design

Package mechanics

Packages

Package Design Guidelines

- Use multiple files
- Keep types close
- Organize by responsibility

https://rakyll.org/style-packages/

Naming Packages

Short and clear

Lowercase

No underscores

Prefer Nouns

Abbreviate judiciously

`package utilities`	vague
`package data_layer`	too long, contains underscores
`package dl`	unclear
`package time`	clear and concise
`package json`	clear and concise

Topical Demos

Package Mechanics

Scoping rules

```
package user

type User struct {
  ID                    int
  Username              string
  password              string
}
func NewUser() *User {}
const maxUsers = 100
```

public struct
public field
public field
package-level field

public function
package-level constant

Package-level entities

```
// Package user manages users in our
   app
package user

import “strings”

var currentUsers []*User
const MaxUsers = 100

// GetByID retrieves a copy of the
// user with the provided ID, if
   present.
// The second return value indicates
// if a user was found or not
func GetByID(id int) (User, bool) {}
```

package comment
package declaration

import statement

variable
constant

API comment

function

Package-Oriented Design

Packages

Designing a Package

Provide a clear solution

- Single responsibility
- Cohesive API

Focus on the consumer

- Simple to use
- Minimize API
- Encapsulate changes

Maximize reusability

- Reduce dependencies
- Minimize scope

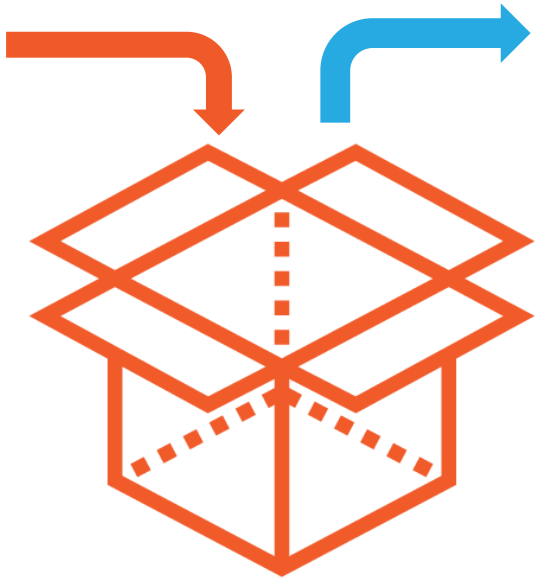
Interface Strategies

concrete types

configuration

interfaces

behavior



concrete types

configuration and behavior

errors

avoid panics

Encapsulate changes

Maximize flexibility

Avoid abstracting too early

Interface Strategies

concrete types

net/http.Request

interfaces

net/http.Handler

Interface Strategies

concrete types

net/http.Response

errors

net/http.Get

Package Design Guidelines

Purpose

Usability

Portability

https://www.ardanlabs.com/blog/2017/02/design-philosophy-on-packaging.html

Package Design Guidelines - Purpose

- Packages must be named with the intent to describe what it provides.

- Packages must not become a dumping ground of disparate concerns.

Package Design Guidelines - Usability

- Packages must be intuitive and simple to use.
- Packages must respect their impact on resources and performance.
- Packages must protect the user’s application from cascading changes.
- Packages must prevent the need for type assertions to the concrete.
- Packages must reduce, minimize, and simplify their code bases

Package Design Guidelines - Portability

- Packages must aspire for the highest level of portability.
- Packages must reduce setting policies when it’s reasonable and practical.
- Packages must not become a single point of dependency.

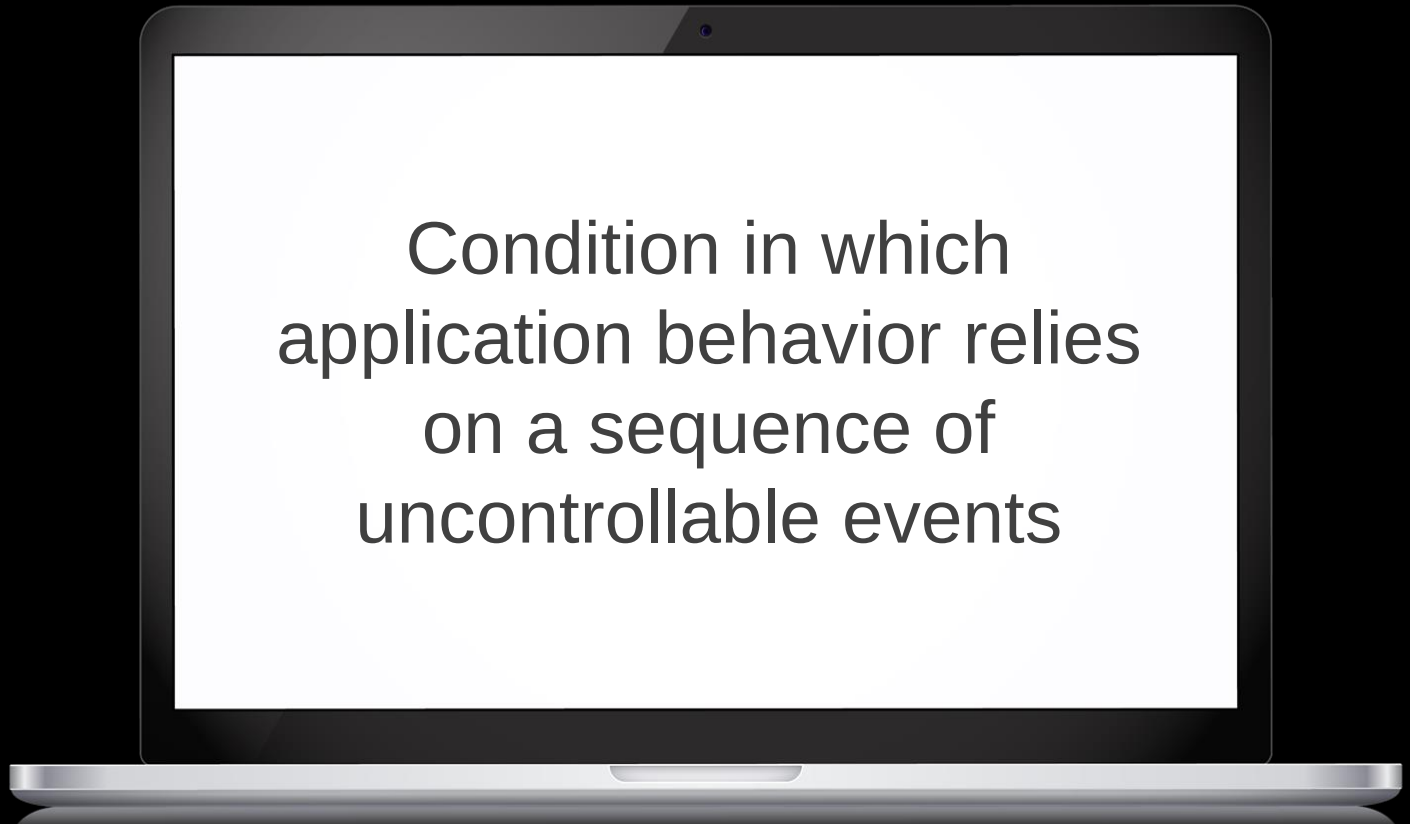
The `main.go` file contains a simple RESTful API for a blogging service. Improve the organization of the program using packages. Make sure that packages are created following package-oriented design principles. Please note that the source code will need to be updated to ensure that the application continues to function.

LAB

Applied Go: Best Practices

Race Conditions

Race Condition



Race Condition

```
func main() {
  var msg string
  go func() {
      fmt.Println(msg)
  }()
  go func() {
      msg = “foo”
  }()
  go func() {
      msg = “bar”
  }()
  // wait logic
}
```

What happens?

Topical Demos

Race Conditions

Common Causes for Race Conditions

Improperly synchronized access to shared resource

That’s pretty much it!

Resolving Race Conditions

Eliminate memory sharing

Protect shared memory with mutexes

Topical Demos

Fixing Race Conditions

Applied Go: Best Practices

Channels

Outline

Buffered channels

Balancing producers and consumers

When to avoid channels

Buffered Channels

Buffered Channels

```
ch := make(chan int)                    // an unbuffered channel

ch = make(chan int, 10)              // a buffered channel
```

What is the difference between a buffered and unbuffered channel?

What are some advantages and disadvantages of buffered channels?

What are the alternatives to buffered channels?

Topical Demos

Buffered Channels

Buffered channels

- Improve perf of production bursts
- Free producers from synchronization overhead
- Protect producers when messages might not have an active receiver
- Cannot easily resize buffer
- Extra memory required for buffer, whether used to not

Balancing Producers and Consumers

Channels

Topical Demos

Balancing Producers and Consumers

Balancing Producers and Consumers

- The ideal number of producers and consumers is one
- Increase number of consumers to handle slow consumption
- Increase number of producers to avoid blocking signal sources
  - e.g., Web services that are sending data
- Avoid buffered channels to balance work loads

Beware of synchronization issues!

When to Avoid Channels

Channels

When to Avoid Channels

- Channels are the best concurrency technique to use in Go
- Until they’re not!

Topical Demos

When to Avoid Channels

Channels

...are good at

- Synchronizing tasks
- Decoupling producers and consumers
- Transferring data ownership
- Distributing workloads
- Communicating async results

..are not so good at

- Synchronizing memory
- Controlling access to shared resource
- Code with extremely high performance requirements

https://tinyurl.com/y25u4les

Applied Go: Gotchas

Memory management

Leaking memory

Sharing memory

Leaking Memory

Topical Demos

Leaking Memory

Freeing Memory from Maps

```
m := make(map[int]string)

... add a lot of data ...

... delete a lot of data ...

func() {
  mNew := make(map[int]string, len(m))
  for k, v := range m {
      mNew[k] = v
  }
  m = mNew
}()
```

Buffered Channels

```
var ch chan int

func BenchmarkUnbuffered(b *testing.B) {
  for i := 0; i < b.N; i++ {
      ch = make(chan int)
  }
}
func BenchmarkBuffered(b *testing.B) {
  for i := 0; i < b.N; i++ {
      ch = make(chan int, 10)
  }
}
```

```
BenchmarkUnbuffered     1143 ns/op     96 B/op
BenchmarkBuffered       1146 ns/op     176 B/op
```

Sharing Memory

Memory Management

Topical Demos

Sharing Memory

Using Mutexes to Protect Memory

```
var m sync.Mutex = sync.Mutex{}

var cache map[string]Object = { ... }

func updater() {
  m.Lock()                                  // must lock mutex
  before all updates
  defer m.Unlock()
  // update cache
}
func reader() {
  m.Lock()                                  // important to
  lock for read ops too!
  defer m.Unlock()
  // query cache
}
```

Sharing Memory with Closures

```
users := []string{“Fred”, “Wilma”, “Pebbles”}

for _, u := range users {

  go func() {
       fmt.Println(u)                          // behavior is undefined
  }()
}
```

Sharing Memory with Closures

```
users := []string{“Fred”, “Wilma”, “Pebbles”}

for _, u := range users {
  u := u                                          // shadow closure
  variable
  go func() {
       fmt.Println(u)
  }()
}
```

Sharing Memory with Closures

```
users := []string{“Fred”, “Wilma”, “Pebbles”}

for _, u := range users {

   go func(u string) {              // pass closure variable as parameter
        fmt.Println(u)
   }(u)
}
```

Sharing Memory with Closures

```
users := []string{“Fred”, “Wilma”, “Pebbles”}

for _, u := range users {

   go myFunc(u)


}
myFunc := func(u string) {                    // remove closure
   fmt.Pritnln(u)
}
```

Applied Go: Gotchas

Pointers and Values

Outline

Pointers in Go

Methods

Method Sets

Pointers in Go

Pointers and Values

Pointers in Go

Isolation

Memory efficiency

Memory Isolation

- Go code isn’t generally immutable
- Focus is on isolating effects of mutations

Value types isolate mutations to their current scope

Pointers allow mutations from anywhere

Topical Demo

Pointers in Go

Pointers in Go

Isolation

Memory efficiency

Ensure that efficiency gain justifies loss of isolation!

Methods

Pointers and Values

Methods

Value receivers

Pointer receivers

Topical Demos

Methods

Method Receivers

Value Receiver

- Copies variable with each call
- Isolates variable from method's mutations
- Performance impact (normally small)
- Memory impact (normally small)

Pointer Receiver

- Copies pointer to variable with each call
- Allows variable to by mutated by method
- Very performant
- Very memory efficient

Method Sets

Pointers and Values

Topical Demos

Method Sets

Method sets

Values

- All methods with value receiver types

Pointers

- All methods, regardless of receiver type

https://www.ardanlabs.com/blog/2017/07/interface-semantics.html

Applied Go: Gotchas

Goroutines

Outline

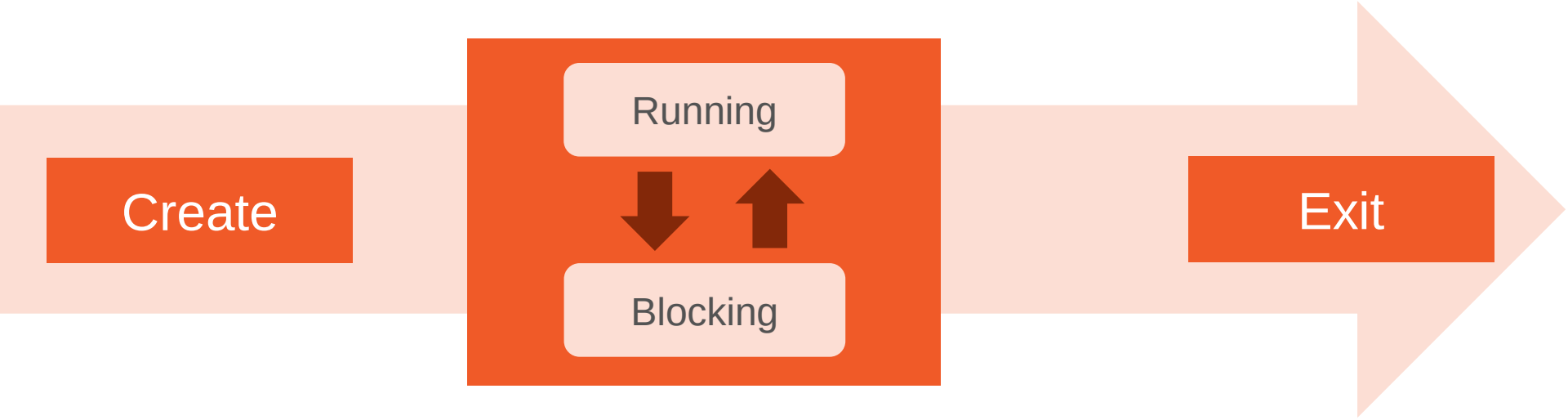
Lifecycle of Goroutine

Guidelines

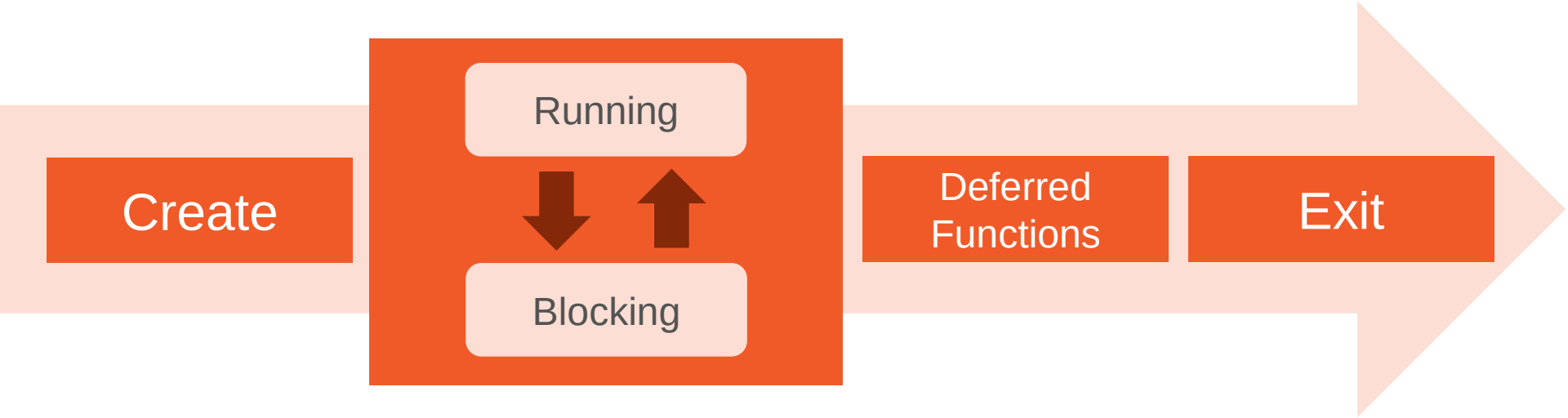
Lifecycle of a Goroutine

Lifecycle of a Goroutine

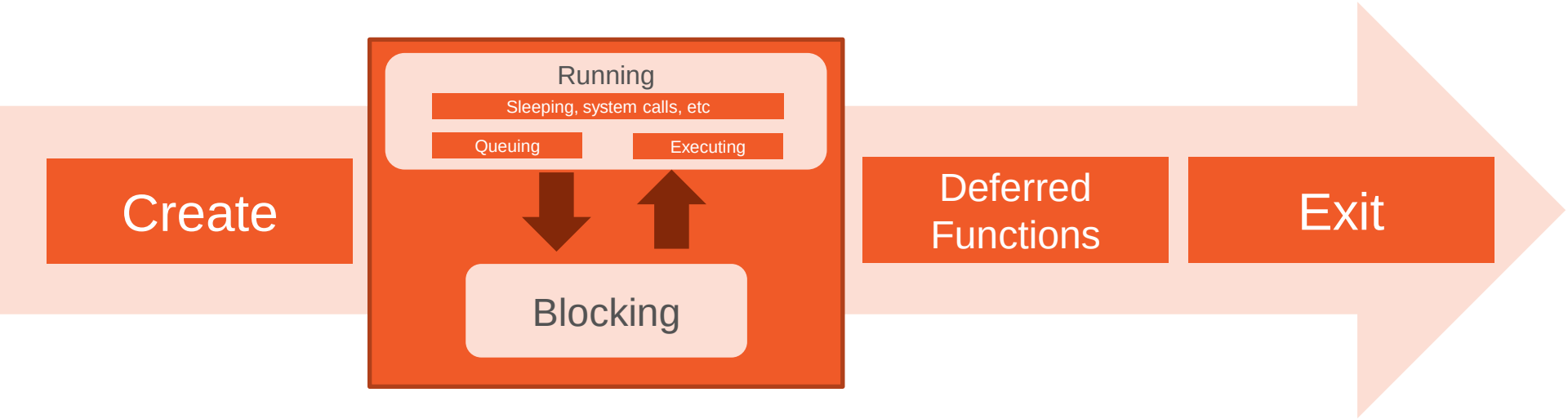
Lifecycle of a Goroutine



Lifecycle of a Goroutine



Lifecycle of a Goroutine



Guidelines

Guidelines

Goroutines are cheap – use them!

Know how a goroutine will stop when you start it

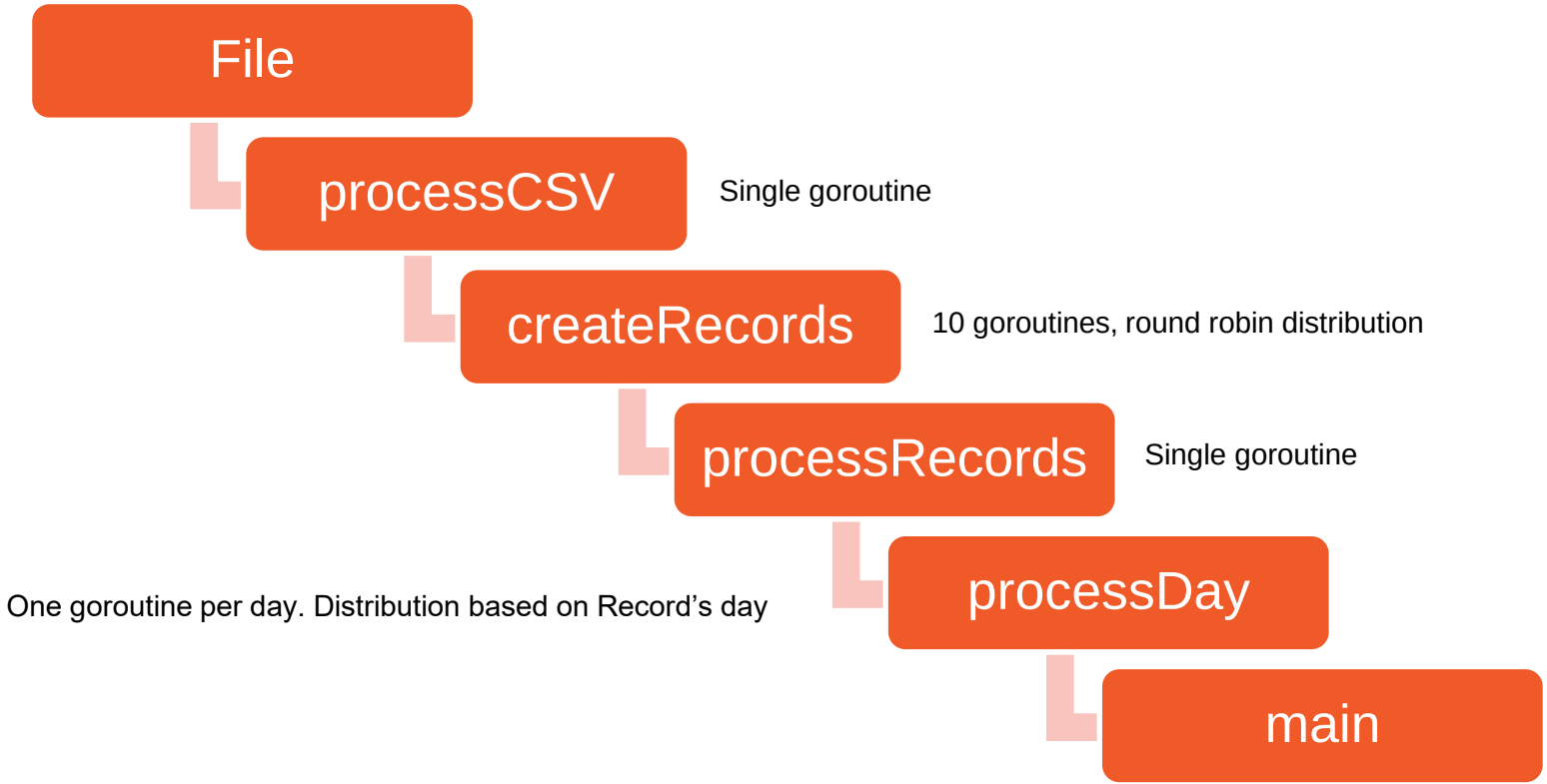
Use channels to communicate between goroutines

Use sync.WaitGroup to synchronize completion of tasks*

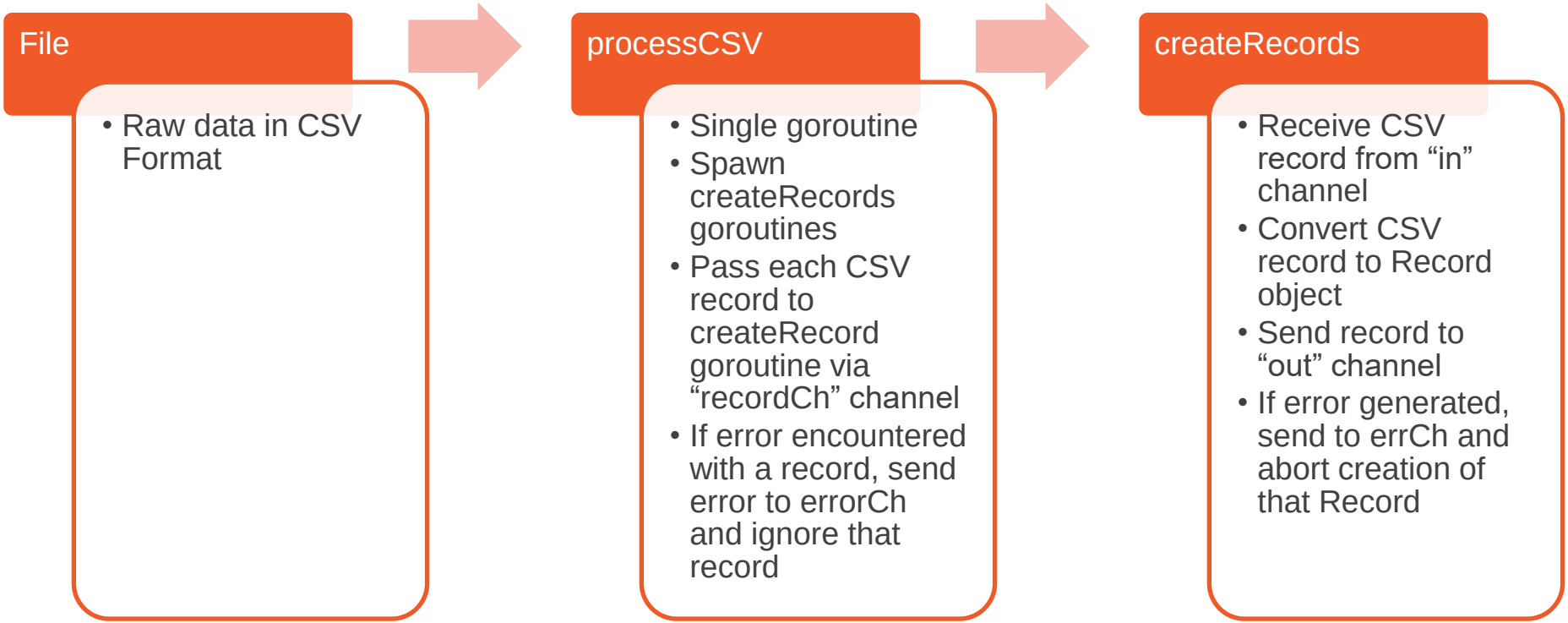
In this lab, you will complete an application that is performing an ETL (extract, transform, and load) operation. The application's goal is to process a data extract from the [NOAA Local Climatological Database (LCD)](https://www.ncdc.noaa.gov/cdo-web/datatools/lcd) and generate a report indicating the minimum, maximum, and average temperature for each day in the dataset. In order to maximize processing speed, this application will perform its calculations using multiple goroutines.

LAB

Goroutine Lab – ETL Processing



Goroutine Lab – ETL Processing



Goroutine Lab – ETL Processing

createRecords

- Receive CSV record from “in” channel
- Convert CSV record to Record object
- Send record to “out” channel
- If error generated, send to errCh and abort creation of that Record

processRecords

- Receive Record
- For each day, spawn a processDay goroutine and matching channel to analyze each day as separate task
- Send Records to correct processDay goroutine, based on date of measurement

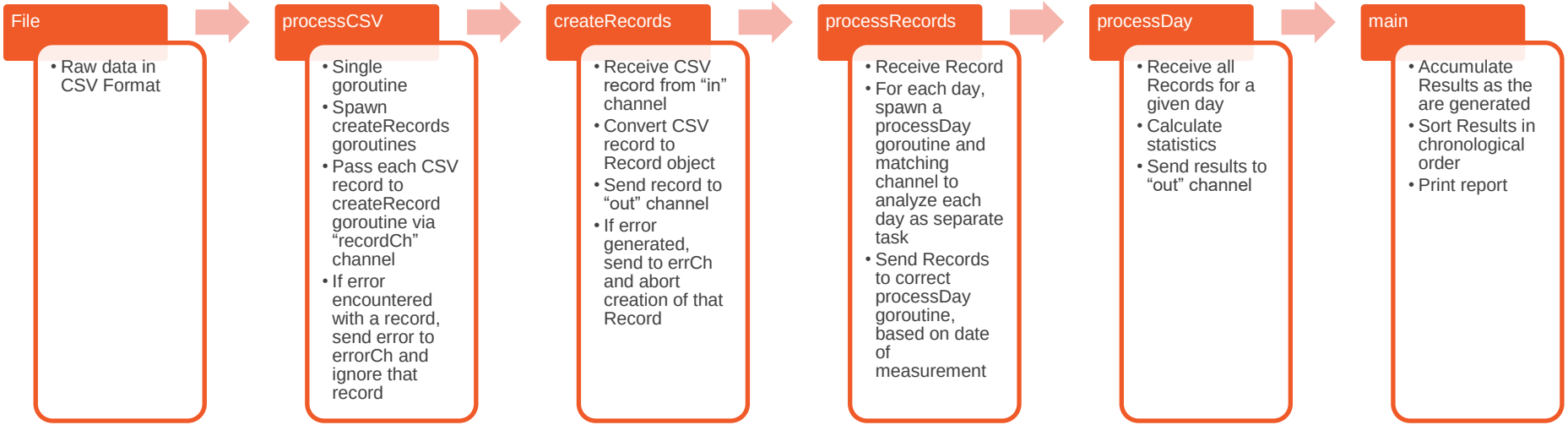
processDay

- Receive all Records for a given day
- Calculate statistics
- Send results to “out” channel

main

- Accumulate Results as the are generated
- Sort Results in chronological order
- Print report

Goroutine Lab – ETL Processing



Modules

Outline

Goals and Overview

Standard workflows

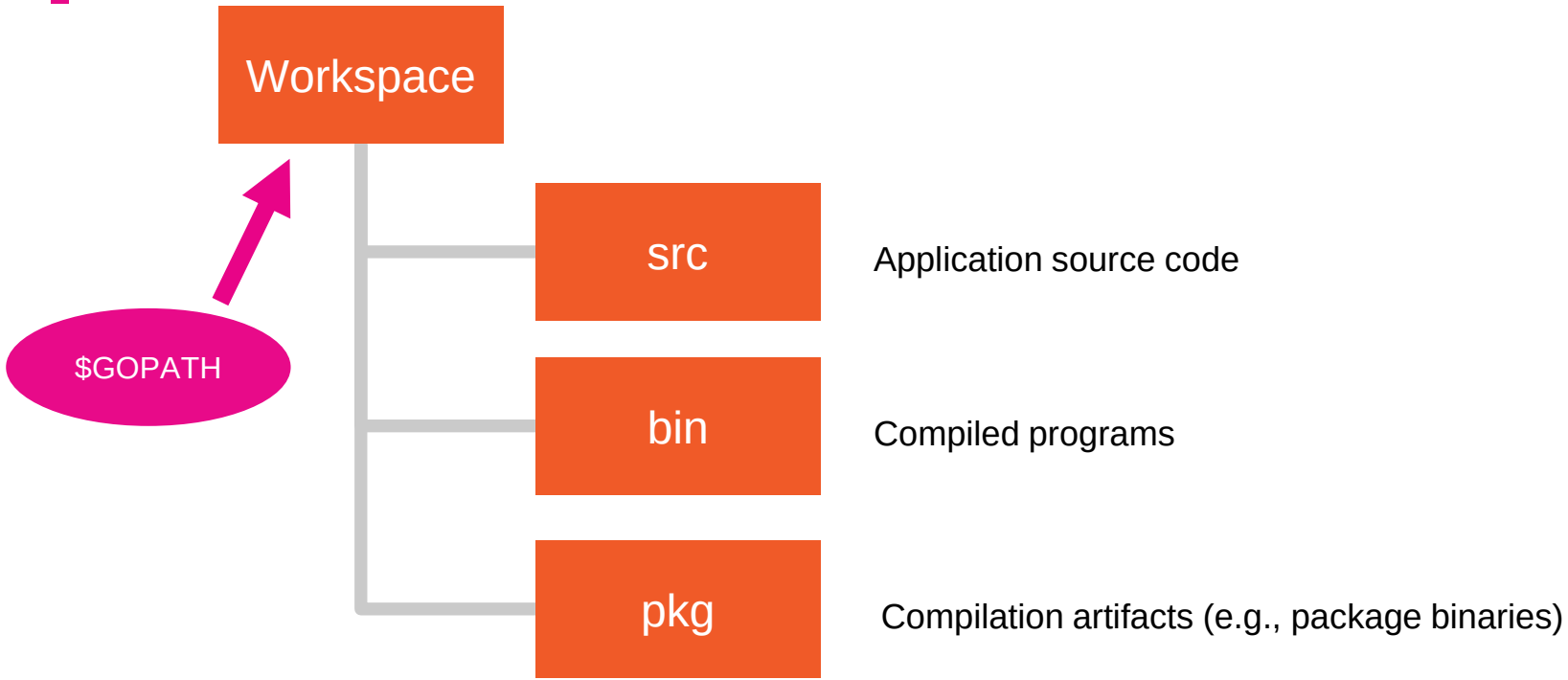
Versioning

Identifying Conflicts

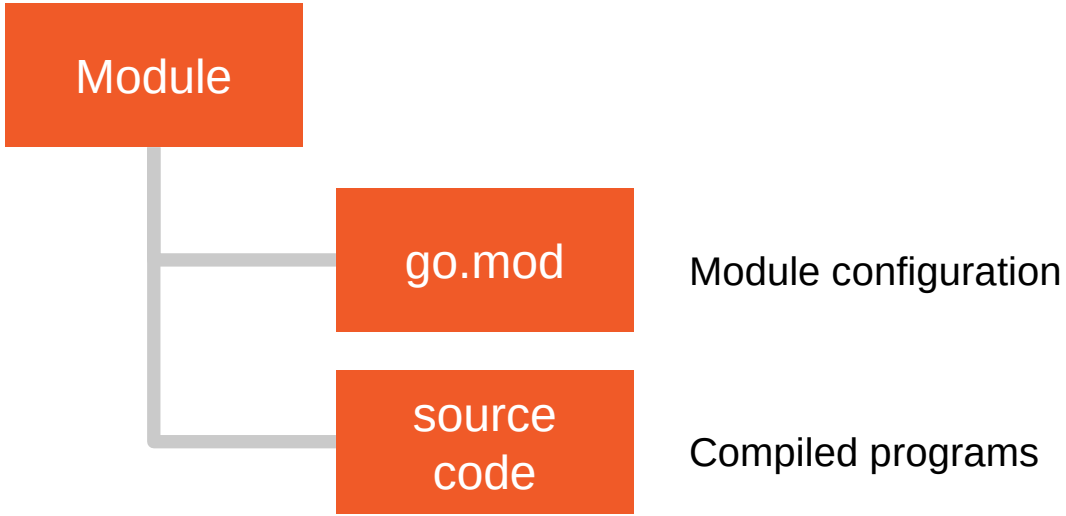
Goals and Overview

Modules

Workspaces: Pre-cursor to Modules



Modules



Module configuration

Compiled programs

Goals of Module System

Keep what works well with workspaces

Address weaknesses

Goal: Keep What Works Well

Simplify build process

Retrieve dependencies

Share projects*

Goal: Address Weaknesses

Versioning and API stability

Vendoring and reproducible builds

Overview of Modules

One or more related packages

Configured via go.mod file

Version controlled

Strict semantic versioning

Dependent libraries kept in cache

Integrity checks via checksums

Standard Workflows

Modules

Topical Demos

Standard Workflows

Common commands

`go mod init`	Initialize a new module
`go get`	Retrieve a module as a dependency
`go list -m`	List module dependencies
`go mod verify`	Verify module integrity
`go mod tidy`	Remove unused depedencies

Versioning

Modules

Versioning

Semantic versioning

Changing major versions

Module queries

Semantic Versioning

Semantic Versioning

v1.5.3-pre1

Semantic Versioning

v1.5.3-pre1

v	Version prefix (required)
1	Major revision (likely to break backward compatibility)
5	Minor revision (new features, doesn’t break BC)
3	Patch (bug fixes, no new features, and doesn’t break BC)
pre1	Pre-release of new version, if applicable (text is arbitrary)

https://semver.org

Changing Major Versions

Topical Demos

Versioning

Module Queries

Modules - Versioning

Module Queries

```
go get github.com/gorilla/mux

go get github.com/gorilla/mux@latest

go get github.com/gorilla/mux@v1.6.2

go get github.com/gorilla/mux/v2@v2

go get github.com/gorilla/mux@main

go get
   github.com/gorilla/mux@<v1.6.2

go get
   github.com/gorilla/mux@>v1.6.2
```

Retrieve latest published version

Retrieve latest published version

Retrieve specific version

Retrieve specific version beyond v1

Retrieve main branch (might be master!)

Retrieve most recent version before 1.6.2

Retrieve first version after 1.6.2

Closest match wins!

Identifying Conflicts

Modules

Topical Demos

Identifying Conflicts

Identifying Conflicts

```
go mod why -m

go mod graph

go mod edit

go mod edit -replace=old=new

go mod edit -
  exclude=path@version

go mod edit -
  require=path@version
```

Why a module is included

Print module tree in module / requirement pairs

Edit the go.mod file programmatically

Replace module with another one

Exclude module from build

Include module in build (similar to go get ...)

Error Management

Outline

Wrapping and unwrapping errors

Error groups

Wrapping and Unwrapping Errors

Topical Demos

Wrapping and Unwrapping Errors

Errors

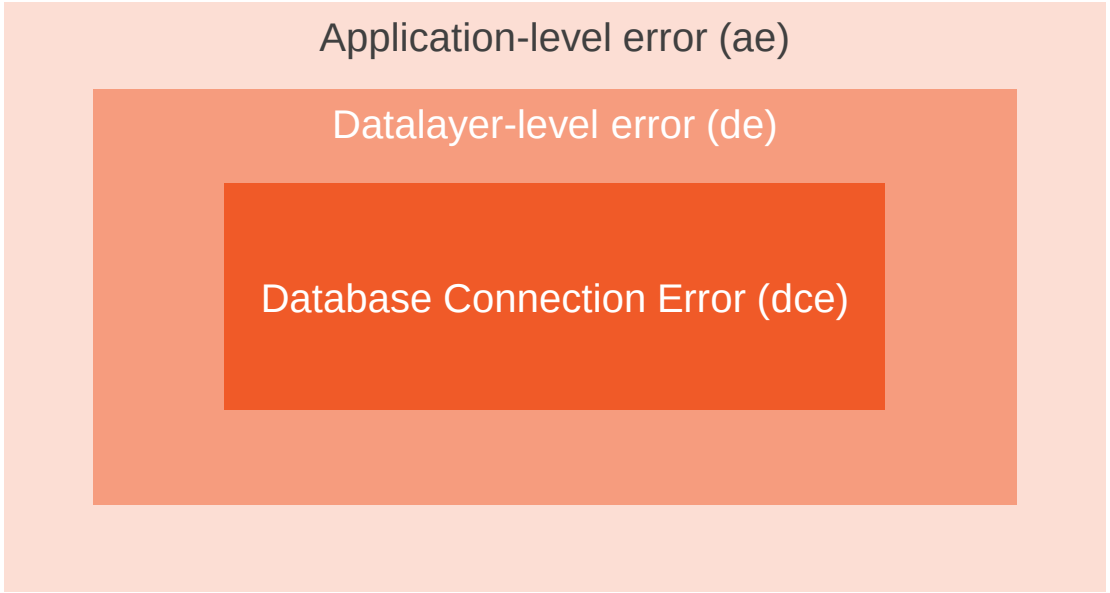
```
firstErr := errors.New(“error
  value”)

secondErr := fmt.Errorf(“foo
  %w bar”,    firstErr)
```

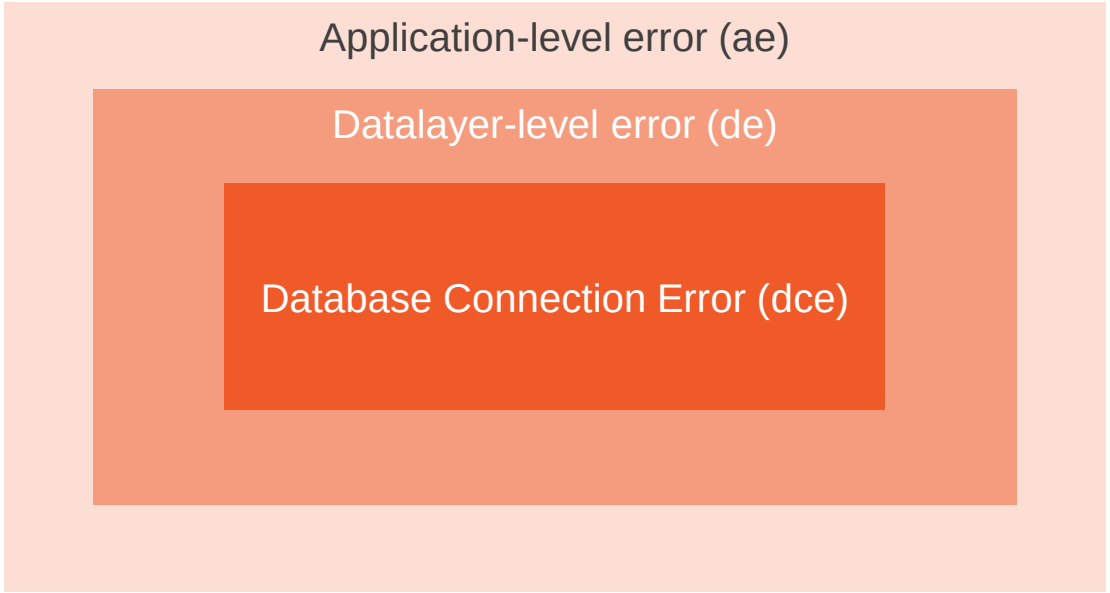
Create new error

Wrap existing error

Unwrapping Errors

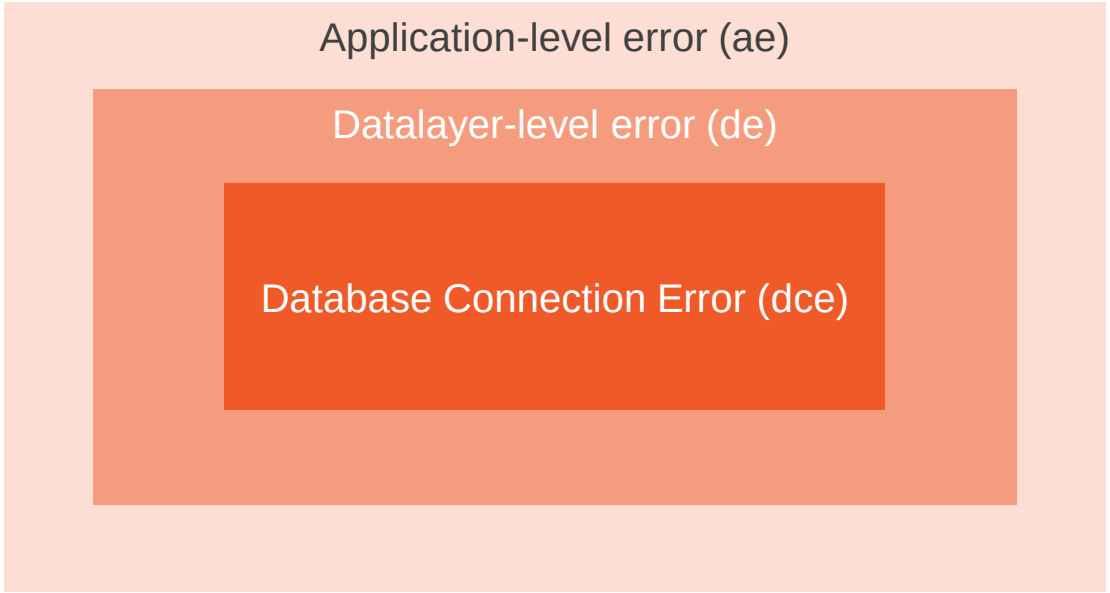


Unwrapping Errors



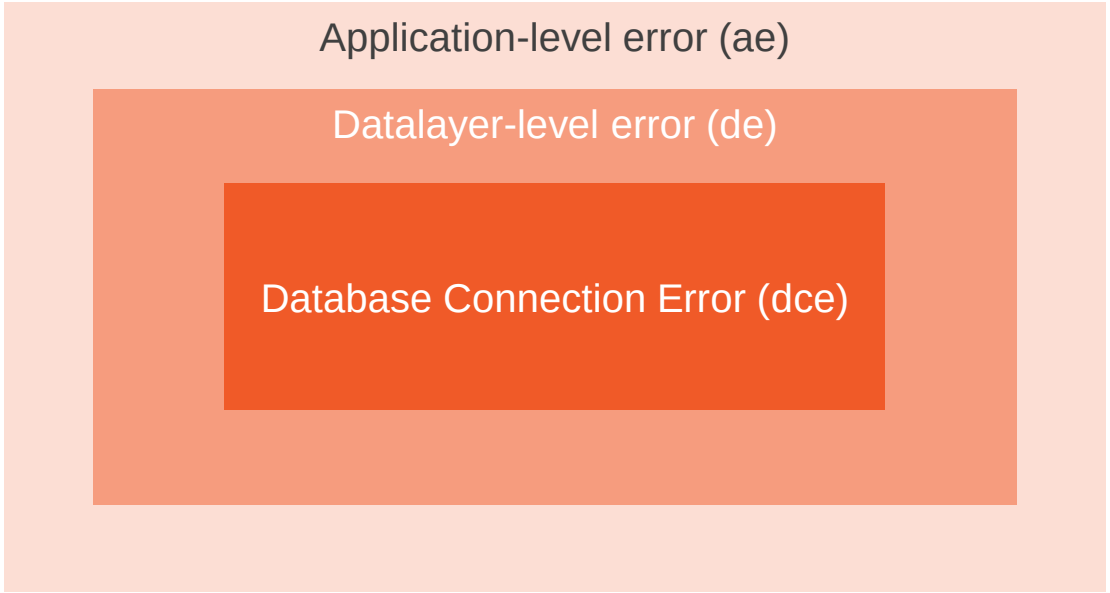

```
errors.Unwrap(ae)                    // de
```

Unwrapping Errors




```
errors.Is(ae, dce)      // true
```

Unwrapping Errors




```
var newDce DatabaseConnectionError
errors.As(ae, &newDce)                    // true
                                                //
```

Wrapping and Unwrapping Errors

Do

- Wrap errors when passing package boundaries
- Return predefined errors and test for them with `errors.Is`
- Create custom error types to add additional information

Don’t

- Pass errors along without adding context
- Ignore errors
- Panic in a library package

Error Groups

Topical Demos

Error Groups

Error Groups

```
go get
  golang.org/x/sync/errgroup

g := new(errgroup.Group)

g.Go(f func() error)

err := g.Wait()

g, newCtx :=
  errgroup.WithContext(ctx)
```

Retrieve the errgroup package

Create a new error group

Add a function to the group

Wait for all goroutines to finish

Create group with derived context

- Context canceled with non-nil error returned by a goroutine or Wait() returns

Testing

Outline

Review types

Including Fuzz Tests

Table-driven tests

Testing web services

Review Test Types

Topical Demos

Test Types

Testing

```
foo_test.go

package foo

package foo_test

func TestFoo(t *testing.T) {}

go test

go test {pkg1} {pkg2}

go test ./...
```

Test file name

Package declaration for white-box testing

Package declaration for black-box testing

Test function name

Run tests in current package

Run tests for specified packages

Run tests for current package and descendants

Reporting test failures

Non-immediate failures

- `t.Fail()`
- `t.Error(...interface{})`
- `t.Errorf(string, ...interface{})`

Immediate failures

- `t.FailNow()`
- `t.Fatal(...interface{})`
- `t.Fatalf(string, ...interface{})`

Benchmark tests

```
func BenchmarkFoo(b *testing.B)
  {
  // setup code
  b.StartTimer()
  for i := 0; i < b.N; i++ {
      ...
  }
  b.StopTimer()
  // tear down code
}

go test –bench .

go test –bench . –benchtime 1m
```

Benchmark test signature

Start benchmark timer
Run benchmarked iterations

Stop benchmark timer

Include benchmark tests in test run

Tune b.N to run tests for approx. 1 minute

Example tests

```
func ExampleFoo() {
   fmt.Println(“Hello,“)
   fmt.Println(“World”)
   // Output:
   // Hello,
   // World
}

func Example{FunctionName}
func Example{TypeName}
func Example{Type + Method}
func Example{*}_suffix

godoc
```

Example test signature

Start describing expected output to stdout

Example for function
Example for type
Example for type’s method
Description of example test

Tool to view documentation, including examples
`golang.org/x/tools/cmd/godoc`

Fuzz tests

```
func FuzzFoo(f *testing.F) {

   f.Add(...args)

   f.Fuzz(func(t *testing.T, ...args) {
        // arrange
        // act
        // assert
   }
}
go test –fuzz=regexp
```

Prefix test with “Fuzz”

Add arguments in order they should be passed to fuzz test

- string, []byte
- int, int8, int16, int32, int64
- uint, uint8, uint16, uint32, uint64
- float32, float64
- bool

One and only one f.Fuzz per test

- Tests run in parallel – don’t test shared memory!
- Arguments controlled by fuzzing engine
- Assertions typically made against arguments

Run fuzz tests matching regular expression
Failed tests stored in ./testdata/fuzz/{FuzzTestName}

https://go.dev/doc/fuzz/

Table-driven Tests

Testing

Topical Demos

Table-driven Tests

Testing Web Services

Topical Demos

Testing Web Services

Profiling

Outline

Code coverage reports

Profiling programs

Profiling web services

Code Coverage Reports

Topical Demos

Code Coverage

Code Coverage Reports

```
go test -cover

go test -coverprofile
  cover.out

go tool cover

go test -coverprofile
  cover.out
  -covermode count
```

Run tests with basic coverage stats

Generate coverage report to cover.out

Analyze coverage report

Set cover mode

- set – is statement executed
- count – execution count
- atomic – execution count (threadsafe)

Profiling Programs

Topical Demos

Profiling Programs

Profiling

```
go test -{profiletype} {dest}

go test ... -
  {profiletype}Rate {num}

go tool pprof myprofile.out

go tool pprof -http
  localhost:3000 prf.out
```

http://graphviz.org/

Run test with profile type

Set profiling rate

Analyze profile with pprof

Explore profile with local web server

Profiling Options

blockprofile

cpuprofile

memprofile

mutexprofile

trace

Profiling Web Services

Profiling

Topical Demos

Profiling Programs

Profiling

```
import _ “net/http/pprof”

go tool pprof http://localhost:8000/debug/pprof/heap                // memory
go tool pprof http://localhost:8000/debug/pprof/profile             // CPU
go tool pprof http://localhost:8000/debug/pprof/block               // goroutines
go tool pprof http://localhost:8000/debug/pprof/trace?seconds=5     // trace

http://localhost:8000/debug/pprof                                     // website
```

Code Generation

Topical Demos

Code Generation

Code Generation

```
go generate

//go:generate {command}
  {argument...}

$GOARCH // execution architecture
$GOOS             // execution
  operating system
$GOFILE // base name of file
$GOLINE // line containing directive
$GOPACKAGE        // package
  containing $GOFILE
$DOLLAR // literal dollar sign ($)
        // used for var expansion
```

Command to initiate code generation

Code generation directive

Preset environment variables

Templates

Outline

Creating and compiling

Data pipelines

Control flow

Functions

Creating and Compiling

Templates

Topical Demos

Creating and Compiling

Creating and Compiling Templates

```
t := template.New({name})

template.Must

t, err :=
  {template}.Parse({text})

err :=
  {template}.Execute({w},
  {data})
```

Create a new template named ‘name’

Compile template, panic on error

Compile ‘text’ as template’s body

Generate template’s output and send to writer, ‘w’. Execution done with data context ‘data’

Data Pipelines

Templates

Topical Demos

Data Pipelines

Data Pipelines

```
type Context struct {
  Title    string
  ImageURL string
}
// t defined previously
err := t.Execute(w, Context{
  Title: “The Title”,
  ImageURL: “https://...”,
  })
```

Pass data into template

Data Pipelines

```
<!DOCTYPE html>
<html>
<head>
  <title>{{.Title}}</title>
</head>
<body>
  <img src="{{.ImageURL}}"/>
</body>
</html>
```

```
type Context struct {
        Title    string
        ImageURL string
}
```

Retrieve property from context

Control Flow

Templates

Topical Demos

Control Flow

If Blocks

```
{{if pipeline}}
      T1
{{end}}
```

```
{{if pipeline}}
      T1
{{else}}
      T2
{{end}}
```

```
{{if pipeline}}
      T1
{{else if pipeline}}
      T2
{{end}}
```

T1 prints if pipeline results in non-empty value

Empty values

- false
- zero
- nil
- empty collection

T1 prints if pipeline is non-empty, otherwise T2 prints

T1 prints if first pipeline is non-empty, otherwise T2 prints if second pipeline is non-empty

Logical Operators

eq / ne	lt / gt	le / ge
and	or	not

All arguments are evaluated!

Range Blocks

```
{{range pipeline}}
  T1
{{end}}

{{range pipeline}}
      T1
{{else}}
      T2
{{end}}
```

Pipeline must be array, slice, map or channel

Data context of T1 is the current collection item

T1 executed if pipeline is non-empty, otherwise T2 is executed

Functions

Templates

Topical Demos

Functions

Functions

```
{{.Title}}

{{template “content”}}

type Data struct {}
func (d Data) SayMsg(m string)
  string {
  return m
}

{{.SayMsg “Hello World!”}}
```

Data command

Function with one argument

Method with one argument

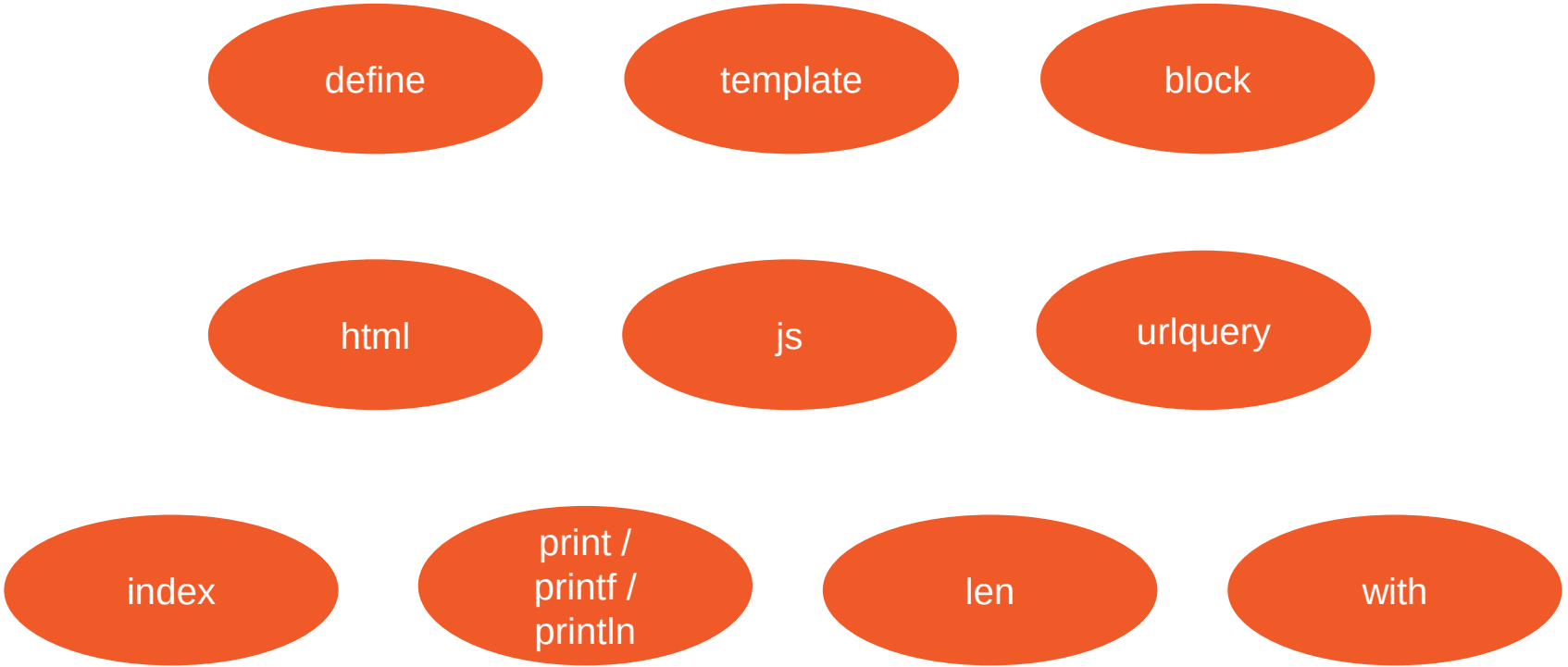
Pipelines

```
{{ command1 command2 | command3 }}
```

Pipe operator

Pass result of previous command as last argument of next command

Built-in Functions



Topical Demos

Custom Functions

Custom Functions

```
template.Funcs(funcMap FuncMap) *Template
type FuncMap map[string]interface{}
template.New(“”).Funcs(funcMap).Parse(...)
```

Acceptable values

- Function that returns a single value
- Function that returns a single value and an error type

Concurrency Patterns

Topical Demos

Concurrency Patterns

Reflection

Topical Demos

Reflection

THANK YOU