
Backend Development
with Go

Agenda

Today

- Introduction to Backend Development and Go
- Introduction to PostgreSQL
- Connecting Go with PostgreSQL
- Introduction to Gin

Agenda

Tomorrow

- Creating a Simple Blog Platform with Gin
- Deploying the Blog Platform
- Conclusion and Next Steps

Prerequisites

- Go
- Docker
- DBeaver
- Git
- Postman
- Clone repository to your local machine: https://github.com/mzwallow/short-course-backend-with-go.git

Introduction to Backend Development and Go

What is Backend Development?

- Development of the server-side components of a software system or application.
- Designing and implementing APIs, database management, and business logic.
- Provide reliable, scalable, and efficient services while ensuring data privacy and security

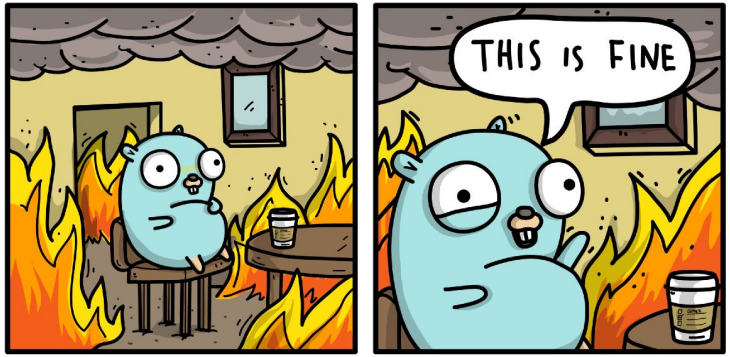
Go, What and What?

What?

- Go, aka Golang, is an open-source programming language developed by Google in 2007.

Why?

- Static typing
- Concurrency
- Fast performance
- Simple syntax?



Basic Go: Variables

Scope

```go
package main

func main() {
  // with type
  var x string = "Hello, World"

  // without type
  var y = "Hi, Mom!"

  // short variable declaration
  z := "NANI!!"
}
```

Const

```go
package main

import "fmt"

func main() {
    const x string = "Hello, World"
    x = "See ya!"
    fmt.Println(x)
}
```

Basic Go: Control Flow - Conditional Statements

If

```go
if i % 2 == 0 {
    // even
} else {
    // odd
}
```

Switch

```go
switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2:
    fmt.Println("Two")
    fallthrough
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
default: fmt.Println("Unknown")
}
```

Basic Go: Control Flow - Loops

For

```go
package main

import "fmt"

func main() {
  for i := 0; i < 5; i++ {
    fmt.Println(i)
  }
}
```

For

```go
package main

import "fmt"

func main() {
  i := 1
  for i < 5 {
    fmt.Println(i)
    i = i + 1
  }
}
```

do..while

```go
package main

import "fmt"

func main() {
    var i int

    for {
        i++
        fmt.Println(i)

        if i >= 5 {
            break
        }
    }
}
```

Basic Go: Arrays

Arrays

```
var 2 [5]int // [0 0 0 0 0]

// An array literal
b := [2]string{"Penn", "Teller"}
// Or, you can have the compiler count the array elements for you:
b := [...]string{"Penn", "Teller"}
```

Basic Go: Slices

Slices

```go
// A slice literal
letters := []string{"a", "b", "c", "d"}

// Or, yuo can use built-in "make" function
x := make([]int, 5) // len(x) == 5, cap(x) == 5
y := make([]int, 0, 5) // len(x) == 0, cap(x) == 5

// Slicing the slices/array use [low:high] expression
a := [5]int{0, 1, 2, 3, 4} // [0, 1, 2, 3, 4]
xs := a[1:4] // [1, 2, 3]

// Append
newSlice := []int{4, 1}
newSlice = append(newSlice, 99) // [4, 1, 99]

// Copy
slice1 := []int{1, 2, 3}
slice2 := make([]int, 2)
copy(slice2, slice1)
fmt.Println(slice1, slice2) // [1, 2, 3] [1, 2]
```

Basic Go: Maps

Maps

```go
// var 'identifier' map['key']'value'
var x map[string]int

x := make(map[string]int)
x["id"] = 10
fmt.Println(x["id"]) // 10

// A map literal
y := map[string]string{
  "A": "Ant",
  "B": "Bird",
}
fmt.Println(y) // map[A:Ant B:Bird]
```

Basic Go: Functions

functions

```go
package main

import "fmt"

func main() {
    xs := []int{1, 2, 3}
    sum, fire := sum(xs)
    fmt.Println(sum, fire) // 6 🔥
}

func sum(xs []int) (int, string) {
    sum := 0
    for _, v := range xs {
        sum += v
    }
    return sum, "🔥"
}
```

Basic Go: Functions

functions

```go
package main

import "fmt"

func main() {
    xs := []int{1, 2, 3}
    sum, fire := sum(xs)
    fmt.Println(sum, fire) // 6 🔥
}

func sum(xs []int) (int, string) {
    sum := 0
    for _, v := range xs {
        sum += v
    }
    return sum, "🔥"
}
```

Basic Go: Functions - Variadic Functions

Variadic Functions

```go
package main

import "fmt"

func main() {
    xs := []int{1, 2, 3}
    result, fire := sum(xs...) // use ellipsis to pass a slice
    fmt.Println(result, fire) // 6 🔥

    result, fire = sum(9, 8, 7)
    fmt.Println(result, fire) // 24 🔥
}

func sum(xs ...int) (int, string) {
    sum := 0
    for _, v := range xs {
        sum += v
    }
    return sum, "🔥"
}
```

Basic Go: Functions - defer

```go
func main() {
    defer fmt.Println("fourth")
    defer fmt.Println("third")
    defer fmt.Println("second")

    fmt.Println("first")
}
```

Deferred function calls are pushed onto a stack

Basic Go: Structs

- Struct is a user-defined type
- Allows you to group items of different types into a single type
- Structs are the only way to create user-defined types in Go

Structs

```
type Rectangle struct {
  width   int
  height  int
}

type Person struct {
  firstName, lastName string
}
```

Basic Go: Structs

Structs

```go
package main

import (
  "fmt"
  "math"
)

type Circle struct {
  radius float64
}

func area(c Circle) float64 {
  return math.Pi * c.radius*c.radius
}

func main() {
  c := Circle{2.7}
  fmt.Println(area(c)) // 22.902210444669592
}
```

Basic Go: Methods

Structs

```go
type Circle struct {
  r float64
}

func (c *Circle) area() float64 {
  return math.Pi * c.r * c.r
}

type Rectangle struct {
  w, h float64
}

func (r *Rectangle) area() float64 {
  return r.w * r.h
}

func main() {
  c := Circle{5}
  r := Rectangle{4, 5}
  fmt.Println("Cicle's area:", c.area())     // Cicle's area: 78.53981633974483
  fmt.Println("Rectangle's area:", r.area()) // Rectangle's area: 20
}
```

In between the keyword **func** and the name of the function, we called it a **receiver.**

Basic Go: Interfaces

Interfaces provide a way to specify the behaviour of an object

if something can do this, then it can be used here

Basic Go: Interfaces

```go
type Person interface {
    Walk()
    Run()
}

type Superman struct {
    Name string
}

func (s Superman) Walk() {
    fmt.Println(s.Name + " (superman) is walking")
}

func (s Superman) Run() {
    fmt.Println(s.Name + " (superman) is running")
}

type TheFlash struct {
    Name  string
    Speed float32
}

func (f TheFlash) Walk() {
    fmt.Println(f.Name + " (the flash) is walking")
}

func (f TheFlash) Run() {
    fmt.Printf("%s (the flash) is running at speed %.2f km/h\n", f.Name,
f.Speed)
}
```

```go
func DoSomething(p Person) {
    p.Walk()
    p.Run()
}

func main() {
    theFlash := TheFlash{
        Name:  "Barry Allen",
        Speed: 192168.3046,
    }
    superman := Superman{
        Name: "Clark Kent",
    }

    DoSomething(theFlash)
    DoSomething(superman)
}
```

Basic Go: Pointers

Pointers

```go
func main() {
  x := 5
  zero(x)
  fmt.Println(x) // x is still 5
}

func zero(x int) {
  x = 0
}
```

Pointers

```go
func main() {
  x := 5
  zero(&x)
  fmt.Println(x) // x is still 0
}

func zero(xPtr *int) {
  *xPtr = 0
}
```

https://www.golang-book.com/books/intro/8#section1

Basic Go: Pointers - The * and & operators

- A pointer is represented using an **asterisk (*)** followed by the type of the stored value. In the **zero** function in previous slide, **xPtr** is a pointer to an **int**.
- An asterisk is also used to **dereference** pointer variables. Dereferencing a pointer gives us access to the value the pointer points to.
- We use `&` operator to find the address of a variable.

https://www.golang-book.com/books/intro/8#section1

Basic Go: Pointers - new

Pointers

```go
func main() {
  xPtr := new(int)
  one(xPtr)
  fmt.Println(*xPtr) // x is 1
}

func one(xPtr *int) {
  *xPtr = 1
}
```

https://www.golang-book.com/books/intro/8#section1

Basic Go: JSON - marshal

JSON

```go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    type ColorGroup struct {
        ID     int
        Name   string
        Colors []string
    }
    group := ColorGroup{
        ID:     1,
        Name:   "Reds",
        Colors: []string{"Crimson", "Red", "Ruby", "Maroon"},
    }
    b, err := json.Marshal(group)
    if err != nil {
        fmt.Println("error:", err)
    }
    fmt.Println(string(b))
}
```

Output

```
{"ID":1,"Name":"Reds","Colors":["Crimson","Red","Ruby","Maroon"]}
```

Basic Go: JSON - marshal with tags

JSON

```
type ColorGroup struct {
  ID     int      `json:"id"`
  Name   string   `json:"name"`
  Colors []string `json:"colors"`
}
```

Output

```
{"id":1,"name":"Reds","colors":["Crimson","Red","Ruby","Maroon"]}
```

Basic Go: JSON - unmarshal

JSON

```
func main() {
  var jsonBlob = []byte(`[
    {"Name": "Platypus", "Order": "Monotremata"},
    {"Name": "Quoll",    "Order": "Dasyuromorphia"}
  ]`)
  type Animal struct {
    Name  string
    Order string
  }
  var animals []Animal
  err := json.Unmarshal(jsonBlob, &animals)
  if err != nil {
    fmt.Println("error:", err)
  }
  fmt.Printf("%+v", animals)
}
```

Output

```
[{Name:Platypus Order:Monotremata} {Name:Quoll Order:Dasyuromorphia}]
```

Basic Go: Error Handling

Error Handling

```go
func main() {
    if res, err := division(10, 0); err != nil {
        fmt.Println("returned error:", err)
    } else {
        fmt.Println("result:", res)
    }
}

func division(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}
```

Output

```
returned error: division by zero
```

Databases

What is a Database?

A database is a set of data stored in a computer. This data is usually structured in a way that makes the data easily accessible.

Types of DBMS

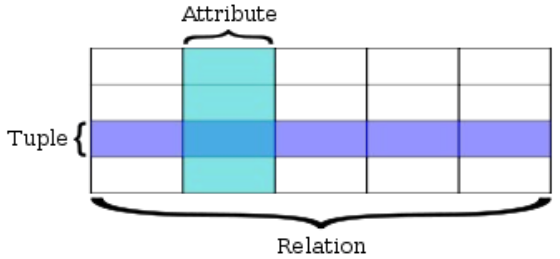
There are several types of database management systems. Here is a list of some common database management systems:

- Relational DBMS
- Object-Oriented DBMS
- Non-Relational DBMS (NoSQL)
- NewSQL

Relational DBMS

A relational database is a digital database based on the **relational model of data**, as proposed by E. F. Codd in 1970.

SQL term	Relational database term	Description
Row	*Tuple* or *record*	A data set representing a single item
Column	***Attribute*** or ***field***	A labeled element of a tuple, e.g. "Address" or "Date of birth"
Table	*Relation* or *Base relvar*	A set of tuples sharing the same attributes; a set of columns and rows
View or *result set*	***Derived relvar***	Any set of tuples; a data report from the RDBMS in response to a query



https://en.wikipedia.org/wiki/Relational_database

Data Definition Language (DDL)

Data definition or data description language (DDL) is a syntax for creating and modifying database objects such as tables, indices, and users.

Common examples of DDL statements include **CREATE**, **ALTER**, and **DROP**.

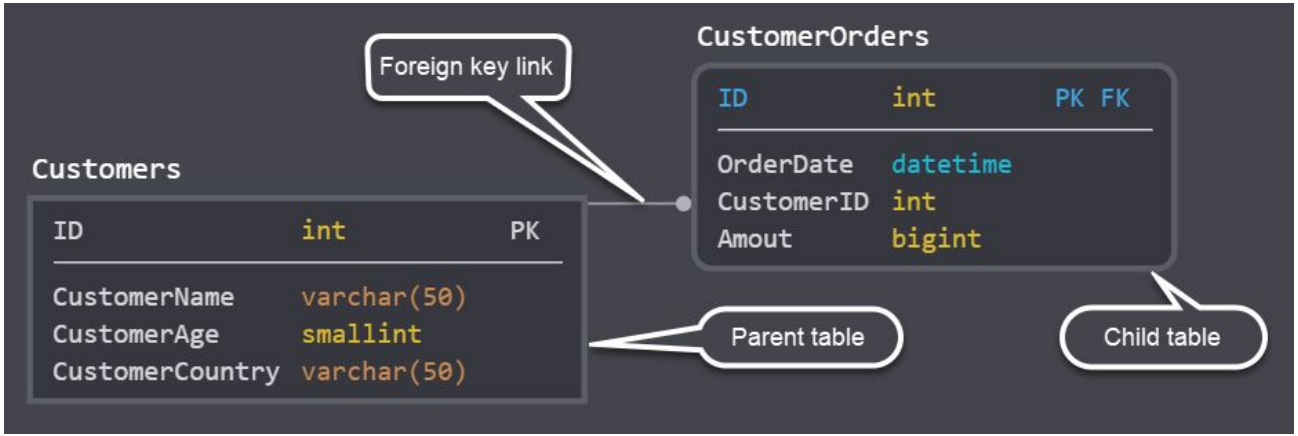
Data Manipulation Language (DML)

A data manipulation language (DML) is a computer programming language used for adding (inserting), deleting, and modifying (updating) data in a database.

SQL

SQL (Structured Query Language) is a programming language used to communicate with data stored in a relational database management system.

Key Constraints: Primary/Secondary Key



https://www.sqlshack.com/what-is-a-foreign-key-in-sql-server/

Introduction to PostgreSQL

PostgreSQL, What and Why?

What?

- PostgreSQL is an open-source relational database management system (RDBMS) that was first released in 1996.

Why?

- Features
- Open-source
- Extensibility

Workshop 1 - Create your first database

- Create a table named **blogs** with the following attributes
  - **id** as SERIAL PRIMARY KEY
  - **title** as VARCHAR(100) NOT NULL
  - **content** as TEXT NOT NULL
  - **created_at** as TIMESTAMP NOT NULL DEFAULT NOW()
  - **updated_at** as TIMESTAMP NOT NULL DEFAULT NOW()

- Make **id** a primary key

Workshop 1 - Create your first database

- Create a table named **comments** with the following attributes
  - **id** as SERIAL PRIMARY KEY
  - **blog_id** as INT NOT NULL
  - **content** as TEXT NOT NULL
  - **created_at** as TIMESTAMP NOT NULL DEFAULT NOW()
  - **updated_at** as TIMESTAMP NOT NULL DEFAULT NOW()

- Make **blog_id** a foreign key to **blogs(id)**

Connecting Go with PostgreSQL

Find and import a database driver

- The **database/sql** package includes functions specifically designed for the kind of database operation you're executing.
- Visit the SQLDrivers wiki page to identify a driver you can use.
- In this course we will use **PostgreSQL** driver
    - https://github.com/jackc/pgx

API

What’s an API?

An API or Application Programming Interface is a set of guidelines or rules that state how applications interact. An API lets one application request data from another system.

JSON

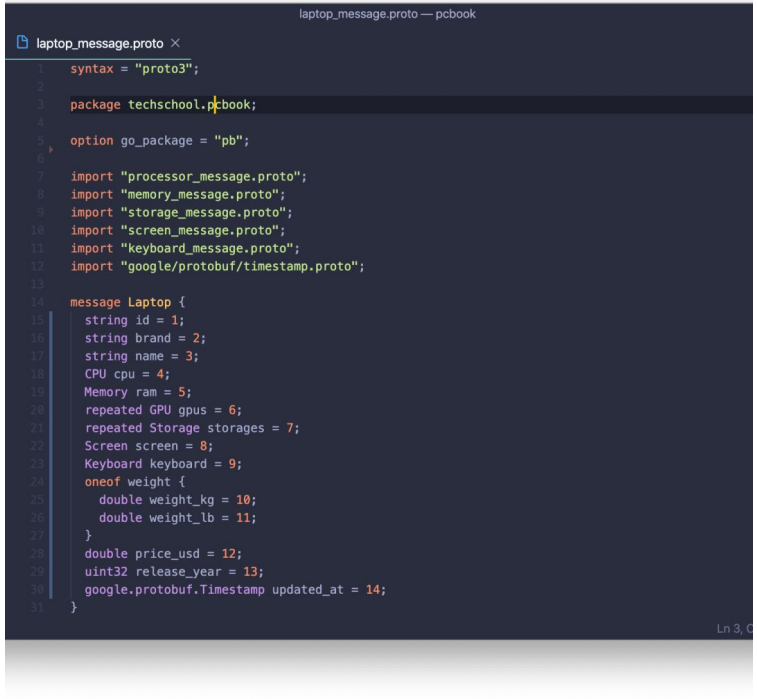
```
{
    "d": {
        "results": [
            {
                "__metadata": {
                    "type": "EmployeeDetails.Employee"
                },
                "UserID": "E12012",
                "RoleCode": "35"
            }
        ]
    }
}
```

XML

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<NewDataSet>
  ▼<Table1>
     <Id>1203</Id>
     <Title>Test Discussion, Dont Reply</Title>
     <CreatedDate>4/25/2014 10:24:56 AM</CreatedDate>
     <Status>Not Sent</Status>
   </Table1>
  ▼<Table1>
     <Id>1182</Id>
     <Title>Negotiation Skills discussion</Title>
     <CreatedDate>4/25/2014 7:47:51 AM</CreatedDate>
     <Status>Not Sent</Status>
   </Table1>
 </NewDataSet>
```

Protobuf


laptop_message.proto — pcbook
laptop_message.proto
syntax = "proto3";
package techschool.pcbook;
option go_package = "pb";
import "processor_message.proto";
import "memory_message.proto";
import "storage_message.proto";
import "screen_message.proto";
import "keyboard_message.proto";
import "google/protobuf/timestamp.proto";
message Laptop {
string id = 1;
string brand = 2;
string name = 3;
CPU cpu = 4;
Memory ram = 5;
repeated GPU gpus = 6;
repeated Storage storages = 7;
Screen screen = 8;
Keyboard keyboard = 9;
oneof weight {
double weight_kg = 10;
double weight_lb = 11;
}
double price_usd = 12;
uint32 release_year = 13;
google.protobuf.Timestamp updated_at = 14;
}

Simple Object Access Protocol (SOAP)

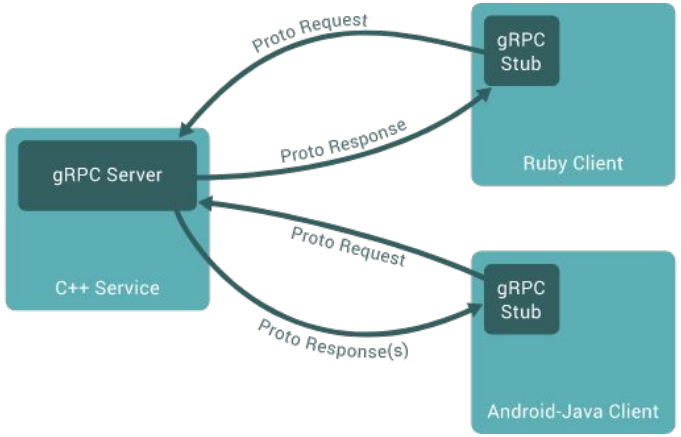
Simple Object Access Protocol (SOAP) is a messaging protocol for transferring information via the internet. SOAP uses XML for exchanging messages.

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is one of the simplest API paradigms, in which a client executes a block of code on another server. Whereas REST is about resources, RPC is about actions. Clients typically pass a method name and arguments to a server and receive back JSON or XML.

gRPC

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment.



Representational State Transfer (REST)

A REST API is an API that follows the design principles of the REST (or REpresentational State Transfer) architecture.

REST APIs communicate via HTTP requests to perform CRUD (or Create, Read, Update, and Delete) operations.

REST Constraints

- Client-Server Architecture
- Stateless
- Cacheable
- Layered System
- Uniform Interface
  - Identification of resources - URI
  - Manipulation of resources through representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)

API Development with Go

Thank You!