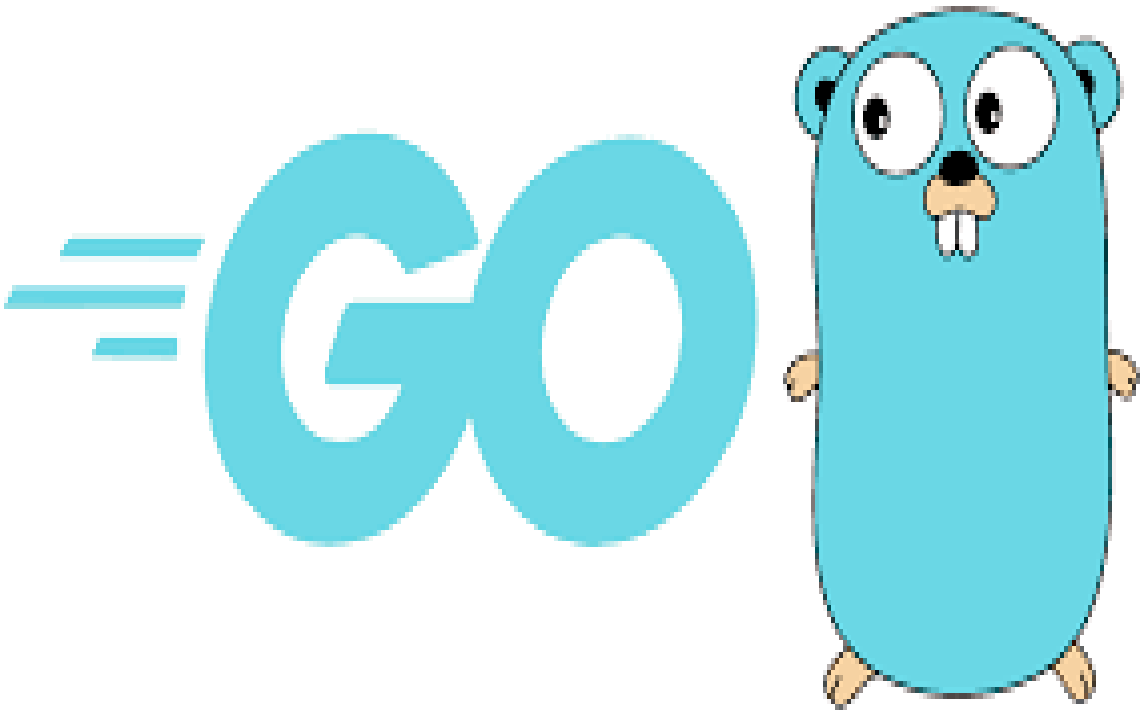
Aprendendo Go



André Luiz Lima Costa

Conteúdo

Introdução

Go é uma linguagem de programação bastante versátil. Ela oferece um ambiente de programação de código aberto que faz com que seja fácil de construir software confiáveis e eficientes, seja do mais simples aos mais complexos.

Go é uma linguagem de programação criada pela Google e lançada em código livre em novembro de 2009. É uma linguagem compilada e focada em produtividade e programação concorrente, baseada em trabalhos feitos no sistema operacional chamado Inferno. O projeto inicial da linguagem foi feito em setembro de 2007 por Robert Griesemer, Rob Pike e Ken Thompson. Atualmente, há implementações para Windows, Linux, Mac OS X e FreeBSD.

Na Parte 1 deste livro vamos cobrir todos os aspectos básicos desta linguagem de programação que servirá de base para as partes seguintes. Na parte 2 vamos falar de desenvolvimento de aplicativos para a web usando Go. Na parte 3 falaremos sobre desenvolvimento de aplicativos com interface gráfica (GUI) usando Go.

Não vou estender muito sobre como instalar Go no seu computador e existem diversos sites que explicam como instalar detalhadamente no sistema operacional de sua escolha. O Site http://www.golangbr.org/doc/instalacao é um exemplo que explica bem como instalar Go.

OSX / Linux Download o arquivo tar.gz para sua plataforma para OSX, **go#.#.#.darwin-amd64.tar.gz** para linux use **go#.##.linux-amd64-.tar,gz** , onde #.#.# são os dígitos da última versão de Go.
a) Extraia o arquivo para /usr/local usando tar -C /usr/local -xzf go#.#.#.linux-amd64.tar.gz ou tar -C /usr/local -xzf go#.#.#.darwin-amd64-osx10.8.tar.gz.
b) Prepare as duas variáveis de ambiente:

1. **GOPATH aponta para a sua pasta de trabalho, exemplo $HOME/code/go** .
2. **Adicione Go bin para o seu PATH** .

c) Use o seguinte:

```
echo 'export GOPATH=$HOME/usuario/go' >> $HOME/.profile
echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
```

É recomendável adicionar as linhas acima em $HOME/.profile .
Digite `go version` e a versão deve aparecer, algo como `go version go1.13.8 darwin/amd64` ou `go version go1.13.8 linux/amd64`

Windows Download o zip mais recente. Se seu sistema for x64 go#.#.#.windows-amd64.zip , onde #.#.# é a versão de Go.
a) Unzip o arquivo em uma pasta de sua escolha tal como: c:\Go
b) Prepare as duas variáveis de ambiente:

1. **GOPATH aponta para o seu ambiente de trabalho. Algo como c:\users\seuNome\work\go .**
2. **Adicione c:\Go\bin à sua variável de ambiente PATH**.

Abra o command prompt e digite `go version` e então `go version go1.13.8 windows/amd64` deverá aparecer.

Parte 1 – O Básico

Estrutura Básica

A estrutura básica de um programa em Go é:

```
/*
 fonte: ola.go (Comentário em bloco)
*/
package main //nome do pacote (comentário de uma linha)
import "fmt" // importando biblioteca (pacote)

func main(){//todo programa Go inicia sua execução  pela func main()
     var texto string //criando uma variável string chamada texto
     texto = "Gopher"  // assinalando o texto Gopher a variável
     fmt.Println("Olá "+texto)// usando uma função do pacote fmt
}
```

Comentários em Go podem ser feitos usando: // para comentário de uma linha ou /* */ para comentário em bloco. Comentários são ignorados pelo compilador.

Compilamos um arquivo de código em Go usando:

```
$ go build ola.go
```

E executamos o arquivo criado por `go build` usando:

```
$ ./ola
Olá Gopher
```

Existem duas formas de definir e iniciar variáveis em Go. A primeira em dois estágios ou definir e iniciar ao mesmo tempo. Quando usamos a segunda forma, Go automaticamente define o tipo da variável baseado no valor fornecido.

```
var umInteiro int                ou          umInteiro := 15
var umBoleano bool                           umBoleano := false
umInteiro = 15
umBoleano = false
```

Podemos declarar variáveis em conjunto.

```
var (
     umInteiro int
     umBoleano bool
)
```

Mas só podemos declarar e iniciar mais de uma variável se forem de um mesmo tipo, conforme abaixo:

```
unInteiro, outroInteiro := 673, 423 // variáveis do tipo int
```

Variáveis declaradas e não usadas em Go geram erro de compilação.

```
package main
func main() {
var umInteiro int //variável declarada e não utilizada
}
```

Tipos de variáveis

bool

Representa dois estados, verdadeiro (true) ou falso (false). Variáveis deste tipo são declaradas como `bool`.

int

Uma variável do tipo inteiro em Go será 32 ou 64 bits dependendo do sistema ser 32 ou 64 bits. Você pode especificar qual dimensão em bits sua variável tipo inteiro terá declarando elas como: `int8`, `int16`, `int32`, `int64`, `byte`, `uint8`, `uint16`, `uint32` e `uint64` onde byte é um apelido pra int8.

float32 e float64

Os tipos de ponto flutuante são `float32` e `float64` (em Go não existe o tipo `float`).
NOTA - Não misture tipos diferentes de int ou de float em operações pois ocorrerá um erro de compilação.

string

Em Go uma string é uma sequência de caracteres UTF-8 dentro de aspas (""). Se você usar aspas simples (‘’) significa um único caractere codificado em UTF-8, que não é uma string em Go.

```
s := "Olá Pateta"
```

Uma vez iniciada uma string em Go **ela é imutável** e não pode ser modificada

```
umaString := "Olá Pateta"
umaString[0]='B'//mudar valor de caractere gera erro de compilação
```

Para modificar a string temos de usar:

```
umaString := "Olá Pateta!"
umasRunas := []rune(umaString)
umasRunas[0] = 'B'
outraString := string(umasRunas)
fmt.Printf("%s\n",outraString)
```

rune

Runa é um apelido para int32 e é um valor pontual codificado em formato UTF-8. Ela é usada para interagir com caracteres de uma string como vimos acima. Caracteres UTF-8 não podem ser representados como inteiro de 8 bits (ASCII). Por esse motivo, caracteres em Go são do tipo `rune`.

complex64

Go suporta nativamente números complexos. Eles são declarados como `complex64` (em partes de 32 bits para real e 32 bits para o imaginário).

```
var complexado complex64 = 56+2i
```

error

Todo programa que não é simples vai, mais cedo ou mais tarde, precisar de uma forma de relatar erros. Em Go temos um tipo especial para erros. A instrução `var meuErro error` criará uma variável do tipo error com o valor `nil`. Esse tipo de variável é uma interface e falaremos mais tarde sobre ela.

Constantes

Constantes, como o próprio nome diz, são valores constantes criados durante a compilação do programa e só podem ser do tipo número, string ou lógico.

```
const (
     constanteA int = 0
     constanteB string = "0"
)
```

Operadores

Principais operadores usados em Go.

Operadores Aritméticos

+ - * / % ++ --

Operadores bitwise

& | ^ &^ << >>

Operadores Lógicos

&& || !

Operadores de comparação

== != < <= > >=

Operadores de ponteiros e channel

* & <-

Funções predefinidas

Go possui uma pequena lista de 15 funções auxiliares predefinidas que podem ser usadas sem a necessidade de utilizar funções importadas de um pacote. Elas são:

```
close     new       panic     complex
delete    make      recover   real
len       append    print     imag
cap       copy      println
```

Aos poucos veremos exemplos destas funções nas práticas 1, 2, 3 e 4.

Palavras chaves reservadas em Go

A linguagem Go tem somente 25 palavras chaves (ou instruções) reservadas:

```
break          default        func      interface      select
case           defer          go        map            struct
chan           else           goto      package        switch
const          fallthrough    if        range          type
continue       for            import    return         var
```

Veremos exemplos destas palavras chaves nas práticas 1, 2, 3 e 4.

Prática 1

Ex 1.1 – Criando e Compilando um programa Go

a) Usando um editor de texto qualquer, crie um arquivo chamado gopher.go com o texto abaixo:

```
package main
func main() {
    println("Olá Gopher")
}
```

b) Usando monitor ou cmd digite

```
go build gopher.go
```

c) Execute o seu programa usando:

```
./gopher
```

ou, se for windows

```
gopher.exe
```

d) O resultado será

```
Olá Gopher
```

Ex 1.2 – Definindo, iniciando e apresentando tipos de variáveis

a) Crie o arquivo variado.go conforme abaixo

```
package main
func main() {
        // variáveis do tipo int
        var a int
        a = 1
        b := 2
        println("a = ",a)
        println("b = ",b)
        //variáveis do tipo float64
        d := 8.87542
        var f float64
        f = 0.00000000000000000000000007
        println("d = ",d)
        println("f = ",f)
        //variáveis do tipo float32
        var d32 float32 = 8.87542
        var f32 float32
        f32 = 0.00000000000000000000000007
        println("d = ",d32)
        println("f = ",f32)
        // variáveis do tipo string
        var texto string
        texto = "Olá"
        texto2 := "Gopher"
        print(texto)
        print(" ")
        println(texto2)
        //variáveis do tipo bool
        var falso bool
```

```
        falso = false
        verdade :=true
        println("falso = ",falso)
        println("verdadeiro = ",verdade)
        //variável do tipo rune
        minhaRuna := rune('ç')
        println("Runa de ç  = ",minhaRuna)
        //variáveis do tipo complex64
        var complexado complex64 = 56+2i
        println("complexado = ",complexado)
        println("Parte real de complexado = ",real(complexado))
        println("Parte imaginária de complexado = ",imag(complexado))
        var traumatizado = complex(56,3)
        println("traumatizado = ",traumatizado)
}
```

b) Compile

```
go build variado.go
```

c) Execute

```
./variado
```

ou

```
variado.exe
```

d) O resultado será

```
a =  1
b =  2
d =  +8.875420e+000
f =  +7.000000e-025
d =  +8.875420e+000
f =  +7.000000e-025
Olá Gopher
falso =  false
verdadeiro =  true
Runa de ç  =  231
complexado =(+5.600000e+001+2.000000e+000i)
Parte real de complexado = +5.600000e+001
Parte imaginária de complexado = +2.000000e+000
traumatizado = (+5.600000e+001+3.000000e+000i)
```

Ex 1.3 – Efetuando operações aritméticas

a) Crie o arquivo ariti.go conforme abaixo

```
package main
func main() {
        intA, intB, intC := 20,8,0
        fA, fB, fC := 20.0, 8.0, 0.0
        sA, sB, sC := "vamos ", "tomar ", "uma!"
        var cpA = complex(34,1)
        var cpB =  complex(0,2)
        var cpC =  complex(3,0)
        op1 := intA + intB * intC
        op2 := intA/intB
```

```
        op3 := fA/fC
        op4 := fA/fB
        op5 := sA+sB+sC
        op6 := cpA/cpB
        op7 := cpB * cpC
        println("Operação 1 = ",op1)
        println("Operação 2 = ",op2)
        println("Operação 3 = ",op3)
        println("Operação 4 = ",op4)
        println("Operação 5 = ",op5)
        println("Operação 6 = ",op6)
        println("Operação 7 = ",op7)
}
```

b) Compile

```
go build ariti.go
```

c) Execute

```
./ariti
```

ou

```
ariti.exe
```

d) O resultado será

```
Operação 1 =  20
Operação 2 =  2
Operação 3 =  +Inf
Operação 4 =  +2.500000e+000
Operação 5 =  vamos tomar uma!
Operação 6 =  (+5.000000e-001-1.700000e+001i)
Operação 7 =  (+0.000000e+000+6.000000e+000i)
```

Ex 1.4 – Formatando a saída usando a função fmt.Printf

Go possui conjuntos de funções e estrutura de dados (interface e struct) auxiliares organizadas em pacotes. Vamos ver agora como importar um pacote (fmt) e como usar uma função dele (Printf).

a) Crie o arquivo formato.go conforme abaixo

```
package main
import "fmt" //importa o pacote padrão fmt
/*
Aqui vamos ver os principais formatadores de saída de texto
no pacote fmt usando a função Printf
*/
func main() {
//texto
fmt.Printf("Texto: %s \n", "Fui, Bar")
fmt.Printf("Texto: % x \n", "Fui, Bar")
//Inteiro
fmt.Printf("Número: %d \n", 12)
//Hexadecimal
fmt.Printf("Hex: %x \n", 12)
fmt.Printf("Hex: %X \n", 12)
fmt.Printf("Hex: %#x \n", 12)
```

```
//Binário
fmt.Printf("Binário: %b \n", 12)
//Float
fmt.Printf("Float: %e \n", 4523.896422)
fmt.Printf("Float: %f \n", 4523.896422)
fmt.Printf("Float: %.2f \n", 4523.896422)
fmt.Printf("Float: %8.1f \n", 4523.896422)
/*
Valores especiais \n como ja vimos é uma nova linha
*/
fmt.Printf("Texto: \a %s \n", "Fui, Bar")//alarme
fmt.Printf("Texto:\b %s \n", "Fui, Bar") // volta uma casa
fmt.Printf("Texto: \t %s \n", "Fui, Bar")// tab
fmt.Printf("Texto: \f %s \n", "Fui, Bar")// avanço de linha
fmt.Printf("Texto: \v %s \n", "Fui, Bar")// avanço na vertical
fmt.Printf("Texto: \\ %s \n", "Fui, Bar")// mostra uma \
fmt.Printf("\u25E2 \u03B6 \uF37A \n")    // mostra caracteres UTF-8
}
```

b) Compile

```
go build formato.go
```

c) Execute

```
./formato
```

ou

```
formato.exe
```

d) O resultado será

```
Texto: Fui, Bar
Texto: 46 75 69 2c 20 42 61 72
Número: 12
Hex: c
Hex: C
Hex: 0xc
Binário: 1100
Float: 4.523896e+03
Float: 4523.896422
Float: 4523.90
Float:   4523.9
Texto:  Fui, Bar
Texto Fui, Bar
Texto:     Fui, Bar
Texto:
        Fui, Bar
Texto:
        Fui, Bar
Texto: \ Fui, Bar
◢ ζ ⅏
```

Controles de fluxo

Existem poucos controles de fluxo em Go. A sintaxe é um pouco diferente de C e podemos omitir os parênteses na condição e iremos sempre precisar de usar os colchetes no bloco da condição.

if-else

Se x for maior que zero mostre “x é maior que zero” caso contrário mostre “x é menor ou igual a zero”

```
if x > 0 {
     println("x é maior que zero")
} else {
     println("x é menor ou igual a zero")
}
```

Você pode iniciar um valor com `if-else`, mas o escopo é limitado ao bloco if-else.

```
if num := 9; num < 0 {
    fmt.Println(num, "é negativo")
} else if num < 10 {
    fmt.Println(num, "tem 1 digito")
} else {
    fmt.Println(num, "tem múltiplos digitos")
}
```

O `else` pode ser omitido

```
f, err := os.Open(name, os.O_RDONLY, 0)
if err ! = nil {
return err
}
façaAlgo(f)
```

goto

Em Go existe uma instrução `goto`. Com ela você pode pular para uma linha de código marcada por um rótulo dentro de uma mesma função. Nesse caso, um loop disfarçado. Evite usar goto desnecessariamente.

```
func minhafuncão() {
     i := 0
Aquí:
     println(i)
     i++
     goto Aquí
}
```

for break continue

O loop `for` pode ser utilizado de três formas:

```
for inicialização ; condição ; passo { }
for condição { } //como se fosse while
for  { } //loop infinito
```

Veja o exemplo abaixo os tipos de loop `for` e de como funciona `break`, que finaliza o loop e `continue`, que pula para a próxima interação.

```
package main
func main() {
    i := 1
    for i <= 3 {
        println(i)
        i = i + 1
    }
    for j := 7; j <= 9; j++ {
        println(j)
    }
    for {
        println("loop")
        break
    }
    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        println(n)
    }
}
```

switch case default fallthrough

Instruções `switch` escolhe condições baseadas num valor em diversas ramificações possíveis.
Um exemplo simples.

```
package main
func main() {
    i := 2
    print("O número ", i, " é ")
    switch i {
    case 1:
        println("um")
    case 2:
        println("dois")
    case 3:
        println("três")
    }
}
```

A instrução `default` assinala um valor no caso de nenhuma das opções `case` forem atendidas.

```
package main
import "time"//pacote ou biblioteca time
func main() {
switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        println("FDS uhhu!!!")
    case time.Friday:
        println("SEXTOUUUUU!!!!")
    default:
        println("dia de trampo…")
    }
}
```

A instrução `fallthrough` força que o `case` seguinte a um `case` válido seja considerado como válido também .

```
package main
import "fmt"
func main() {
  valor := 2
  switch valor {
    case 3:
      fmt.Println(3)
      fallthrough
    case 2:
      fmt.Println(2)
      fallthrough
    default:
      fmt.Println("deu default")
  }
}
```

Arrays slices maps e ranges

Go possui alguns tipos de dados que são conjuntos de dados de um mesmo tipo. Esse tipos especiais são explicados abaixo

Arrays

O tipo [n]T é uma array de n valores do tipo T. A expressão `var a [10]int` declara uma variável `a` com dez números inteiros. A quantidade de elementos de uma array é parte do tipo e não pode ser redimensionado. Isso parece ser limitante mas Go provê um outro tipo mais dinâmico (slice).

```
package main
import "fmt"
func main() {
     var a [2]string
     a[0] = "Olá"
     a[1] = "Pateta"
     fmt.Println(a[0], a[1])
     fmt.Println(a)
     primos := [6]int{2, 3, 5, 7, 11, 13}
     fmt.Println(primos)
}
```

Slices

O tipo slice é um dos mais importantes em Go sendo um segmento de uma array e é redimensionável. Slices são referência a uma array tipificada pelos elementos que ela contém e não pelo número de elementos nela. A instrução abaixo cria uma slice de três strings vazias.

```
s :=make([]strings,3)
```

Vejamos como criar, inicializar, adicionar e alterar elementos de uma slice.

```
package main
import "fmt"
func main() {
    s := make([]string, 3)
    fmt.Println("criada:", s)
```

```
    s[0] = "vamos"
    s[1] = "tomar"
    s[2] = "uma?"
    fmt.Println("inicializada:", s)
    fmt.Println("elemento 2:", s[2])
    fmt.Println("comprimento:", len(s), cap(s))
    s = append(s, "Só")
    s = append(s, "se", "for","agora" )
    fmt.Println("adicionada:", s)
    fmt.Println("novo comprimento:", len(s), cap(s))
    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cópia:", c)
    s = append(s[:0], s[0],s[1],s[6])
    fmt.Println("removido:", s)
    fmt.Println("novo comprimento:", len(s), cap(s))
}
```

Maps

Maps são uma forma de se criar um tipo de dado associativo (também conhecido como dicionários) em Go.
Criamos um map usando `make(map[`*`tipo-da-chave`*`]`*`tipo-do-valor`*`)`
Vejamos na prática como um map funciona:

```
package main
import "fmt"
func main() {
    m := make(map[string]int)
    m["v1"] = 78
    m["v2"] = 873
    m["v3"] = 8
    fmt.Println("Meu mapa:", m)
    valor1 := m["v1"]
    fmt.Println("valor1 menos mapa chave_v3: ", valor1 - m["v3"])
}
```

Range

O tipo range interage com uma variedade de estruturas de dados em Go (strings, arrays, slices, maps, etc).

Interação range e string

```
package main
import "fmt"
func main() {
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

Interação range e array

```
package main
import "fmt"
func main() {
```

```
  nums := []int{2, 3, 4}
  soma := 0
  for _, num := range nums {
      soma += num
  }
  fmt.Println("A soma é:", soma)
  for i, num := range nums {
        if num == 3 {
            fmt.Println("Índice:", i)
        }
  }
}
```

Interação range e slice

```
package main
import "fmt"
func main() {
    s := make([]string, 4)
    s[0] = "foo"
    s[1] = "foi"
    s[2] = "no"
    s[3] = "bar"
    for i, tex := range s {
      fmt.Println(i,tex)
    }
}
```

Interação range e map

```
package main
import "fmt"
func main() {
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}
```

Funções

Uma função em Go é um bloco de código que leva valores, faz algo com esses valores e retorna novos valores. Elas te ajudam a dividir e organizar o seu programa em partes que podem ser usadas mais de uma vez, melhorando assim a leitura, manutenção e teste do seu programa.

Declarando e chamando funções

Declaramos uma função usando a palavra-chave `func`. A função possui um nome, uma lista de argumentos (valores) com seus respectivos nomes e tipos em parenteses, o tipo do valor a ser retornado e um corpo entre colchetes.

Seguindo a descrição acima vemos um exemplo de uma função simples que chamaremos de média que toma dois argumentos (parâmetros ou valores) do tipo float64 e retorna a média destes dois valores com o tipo float64 também.

```
func média(v1 float64, v2 float64) float64 {
     return (v1 + v2) / 2
}
```

Chamamos esta função usando o nome dela com os dois valores do mesmo tipo já definido como abaixo:

```
média(51.07, 35.12)
```

Exemplo:

```
package main
import "fmt"
func média(v1 float64, v2 float64) float64 {
     return (v1 + v2) / 2
}
func main() {
    x := 51.07
    y := 36.12
    res := média(x, y)
    fmt.Printf("A média de %.2f e %.2f é %.2f\n", x, y, res)
}
```

Após execução o resultado será:

```
A média de 51.07 e 36.12 é 43.59
```

Os parâmetros de uma função e o tipo a retornar são opcionais. Como por exemplo a função main

```
func main() {

}
```

Função que retorna mais de um valor

Uma novidade em Go é que uma função pode retornar mais de um valor e até mesmo mensagem de erro se algo não estiver adequado à função. Os tipo dos valores de retorno, nesse caso, tem que estar dentro de parenteses e separados por vírgula.

Considere uma função que calcula o índice de massa corporal (imc). Nela passamos os parâmetros de sua altura, peso anterior e peso atual, O retorno da função será o seu imc atual e quanto ele variou entre o anterior e o atual em porcentagem.

```
func imc(altura, pesoAnterior, pesoAtual float64) (float64,float64){
     imc= pesoAtual/altura*altura
     imcAnt=pesoAnterior/altura*altura
     imcTaxa=(1-imc/imcAnt)*100
     return imc,imcTaxa
}
```

Vamos mostrar na prática como isso funciona no exemplo abaixo:

```
package main
import "fmt"
func imc(altura, pesoAnterior, pesoAtual float64) (float64,float64){
   imcAct := pesoAtual/(altura*altura)
   imcAnt :=pesoAnterior/(altura*altura)
   imcTaxa :=(1-imcAnt/imcAct)*100
   return imcAct,imcTaxa
}
func main() {
   alt :=1.75
   pAnterior := 98.5
   pAtual :=95.3
   imcA,taxa := imc(alt,pAnterior,pAtual)
   fmt.Printf("O seu imc é %.2f e ele variou em %.2f %%\n",imcA,taxa)
}
```

O resultado é:

```
O seu imc é 31.12 e ele variou em -3.36 %
```

Vamos agora adaptar o código anterior para podermos adicionar uma mensagem de erro.

```
package main
import (
        "fmt"
        "errors"
)
func imc(altura, pesoAnterior, pesoAtual float64) (float64, float64,
error){
   if pesoAnterior <= 0 || pesoAtual<=0 || altura<=0 {
      err := errors.New("Nenhum parâmetro pode ser zero ou negativo")
      return 0, 0, err
   }
   imcAct := pesoAtual/(altura*altura)
   imcAnt :=pesoAnterior/(altura*altura)
   imcTaxa :=(1-imcAnt/imcAct)*100
   return imcAct,imcTaxa,nil
}
func main() {
   alt :=1.75
   pAnterior := 0.0
   pAtual :=95.3
   imcA,taxa,erro := imc(alt,pAnterior,pAtual)
   if erro!=nil {
       fmt.Println("Desculpe! Ocorreu um erro: ", erro)
   }else{
   fmt.Printf("O seu imc é %.2f e ele variou em %.2f %%\n",imcA,taxa)
  }
}
```

O resultado é:

```
Desculpe! Ocorreu um erro: Nenhum parâmetro pode ser zero ou negativo
```

Retorno implícito e Blanket

Para finalizar esta parte vamos introduzir dois conceitos relacionados a funções. Primeiro vamos definir os nomes das variáveis de retorno de uma função e também vamos introduzir o conceito de valor blanket (_) que simplesmente é descartado num programa Go.

Vamos supor que só precisamos da taxa de variação do imc e não queremos saber qual é o imc atual e também vamos dar nomes para as variáveis na chamada da função para usar o retorno implícito.
Assim:

```
package main
import "fmt"
func imc(altura, pesoAnterior, pesoAtual float64) (imcAct, imcTaxa
float64){
 imcAct = pesoAtual/(altura*altura) //note ausência do : já declarado
 imcAnt := pesoAnterior/(altura*altura)
 imcTaxa = (1-imcAnt/imcAct)*100 //note ausência do : já declaramos
 return //retorno das variáveis já está implícito
}
func main() {
 alt :=1.75
 pAnterior := 98.5
 pAtual :=95.3
 _,taxa := imc(alt,pAnterior,pAtual) // aqui usamos o blanket _
 fmt.Printf("O seu imc variou em %.2f %%\n",taxa)
}
```

O resultado é:

```
O seu imc variou em -3.36 %
```

Prática 2

Ex 2.1 – Exercitando Loops e arrays

a) Criando e mostrando uma array bidimensional usando loop `for`.

```
package main
import "fmt"
func main() {
   var mtx[3][3] int
   for i:=0;i<3;i++ {// inserindo os dados na array
      for j:=0;j<3;j++ {
        mtx[i][j] = i*j
      }
   }
   for i:=0;i<3;i++ {// mostra dados da array
      for j:=0;j<3;j++ {
        fmt.Printf("%d",mtx[i][j])
        if j!=2 {
           fmt.Printf("\t")
        }
      }
     fmt.Printf("\n")
   }
}
```

b) Resultado

```
0    0    0
0    1    2
0    2    4
```

Ex 2.2 – Mais sobre slices

a) Vamos mostrar aqui que uma slice é uma referência de uma array.

```
package main
import "fmt"
func main() {
  var lang = [5]string{"go", "perl", "php", "R", "sql"}//array lang
  var s []string = lang[1:4] //slice dos elementos 1,2,e 3 da array
  fmt.Println("Array lang = ", lang)
  fmt.Println("Slice s = ", s)
  lang[1]="camile"
  lang[2]="ruby"
  lang[3]="julia"
  fmt.Println("Slice s = ", s)
}
```

b) Resultado

```
Array lang =  [go perl php R sql]
Slice s =  [perl php R]
Slice s =  [camile ruby julia]
```

Ex 2.3 – Usando Maps

a) Um exemplo de como obter dados de um map usando a chave (key).

```
package main
import "fmt"
func main() {
     var possantesPlacas = map[string]string{
          "ferrari":  "RUN3F66",
          "porshe": "FUN3U02",
          "austin": "LUS7Y76",
     }

     var placa = possantesPlacas["austin"]
     fmt.Println("A placa do Austin é : ", placa)
     // se o carro não existir (sem chave)
     placa = possantesPlacas["bugatti"]
     fmt.Println("A placa Bugatti do é : ", placa)
}
```

b) Resultado

```
A placa do Austin é :  LUS7Y76
A placa do Bugatti é :
```

Ex 2.4 – Usando Ranges

a) Veja como usar range para alterar os valores de uma array

```
package main
import "fmt"
func main() {
  galera := [5]string{"Bart","Tigre","Cisalhado","Pica-pau","Mosh"}
  fmt.Printf("Antes[%s] : ", galera[1])
```

```
  for i := range galera {
   galera[1] = "Piauí"
     if i == 1 {
       fmt.Printf("Depois[%s]\n", galera[1])
     }
  }
}
```

b) Resultado

```
Antes[Tigre] : Depois[Piauí]
```

Ex 2.5 – Mais sobre Funções

a)Podemos usar o variadic (…) para passar parâmetros (argumentos) variáveis em tamanho e usar range para navegar os valores.

```
package main
import "fmt"
func soma(valores ...int) (total int) {
  for _, v := range valores {
    total += v
  }
  return
}
func main() {
  tot := soma(1,2,3,4,5)
  fmt.Println(tot)
  tot = soma(100, 232)
  fmt.Println(tot)
  tot = soma(9,8,7,6,5,6,7,8,9,8,5,4,3,2,1,100,232)
  fmt.Println(tot)
}
```

b) Resultado

```
15
332
420
```

Ponteiros

Vamos ver o que são ponteiros (pointers) em Go e como podemos utilizar este tipo nos programas.
Pointer nada mais é do que uma variável que guarda o endereço de memória de uma outra variável ou estrutura de dados.

O endereço de uma variável

Toda variável é guardada em um endereço de memória na forma Hexadecimal. Podemos acessar seu endereço usando o operador & que informa qual é este endereço. O exemplo abaixo demonstra qual o endereço de memória de uma variável do tipo int.

```
package main
import "fmt"
func main() {
  a := 8
  fmt.Printf("O valor de a é %d e seu endereço é %#x \n",a,&a)
}
```

O resultado é:
```
O valor de a é 8 e seu endereço é 0xc0000140f0
```

Criando um pointer

Um pointer é iniciado usando `var p *T` onde T é o tipo do ponteiro. Todo ponteiro criado possui o valor `nil` inicialmente até ser inicializado.

```
package main
import "fmt"
func main() {
  var p *int
  fmt.Println("O valor do pointer p a é ",p)
}
```

O resultado é:
```
O valor do pointer p a é  <nil>
```

Inicializando um pointer

Agora vamos inicializar o ponteiro com o endereço de uma variável. O ponteiro e a variável precisam ter o mesmo tipo. Ao digitarmos var p int = &a o compilador já entende que p é do tipo pointer e o * pode ser omitido.

```
package main
import "fmt"
func main() {
  a := 8
  var p = &a
  fmt.Println("O valor de a é ",a)
  fmt.Println("O valor do endereço de a é ",&a)
  fmt.Println("O valor de p é ",p)
}
```

O resultado é:
```
O valor de a é  8
O valor do endereço de a é  0xc00008a010
O valor de p é  0xc00008a010
```

Agora sabemos que o ponteiro p tem como valor o endereço da variável `a`.

Acessando o valor do pointer

De nada valeria existir o pointer se não tivermos como acessar o valor guardado no endereço que ele aponta. Fazemos isso usando o operador *.

```
package main
import "fmt"
func main() {
  a := 8
  var p = &a
  fmt.Println("O valor de a é ",a)
  fmt.Println("O valor de p é ",p)
  fmt.Println("O apontado por p (*p) é ",*p)
}
```

O resutado é:
```
O valor de a é  8
O valor de p é  0xc00008a010
O apontado por p (*p) é  8
```

Mudando o valor do pointer

Além de acessar o valor usando o pointer podemos também modificar ele e isso é que dá a vantagem de mudarmos valores armazenados em memória de forma eficiente, não faz muito sentido para tipo básicos mas é de fundamental importância para estruturas de dados mais complexas. Vejamos como mudar o valor apontado pelo pointer usando *.

```
package main
import "fmt"
func main() {
  a := 8
  var p = &a
  fmt.Println("O valor de a é ",a)
  *p = 1
  fmt.Println("Agora o valor de a é ",a)
}
```

O resultado é:

```
O valor de a é  8
Agora o valor de a é  1
```

Criando um pointer com a função new()

Com a função `new()` podemos criar um ponteiro de determinado tipo e já alocarmos a memória para o tal tipo sem ter de termos uma variável ou estrutura de dados já iniciada. Fazemos isso assim:

```
package main
import "fmt"
func main() {
        p := new(int) // Pointer do tipo int
        *p = 17
        fmt.Printf("p é %#x e o valor de p é %d\n", p, *p)
}
```

O resultado é:

```
p é 0xc0000140f0 e o valor de p é 17
```

NOTA - Go não permite a aritmética de ponteiros.

Structs

Uma struct é um tipo definido pelo usuário que contém uma coleção de campos ou propriedades. Ela é usada para reunir um grupo de propriedades relacionáveis em uma única unidade. Struct criada com nome iniciado por letra maiúscula pode ser acessada externamente via outro pacote usando import, se o nome da struct começa com letra minúscula, ela só poderá ser acessada de dentro do próprio pacote. O mesmo se aplica a propriedades desta struct.

Criando uma struct

Vamos criar uma struct para descrever propriedades de cidades e criarmos duas variáveis do tipo Cidade, uma com valores inicializados e outra vazia (string será vazia e números serão 0).

```
package main
import "fmt"
type Cidade struct{
     Nome string
     Estado string
     População int
```

```
     IDH  float64
}
func main() {
     var ci1 = Cidade{Nome:"Tefé", População: 45000,Estado: "AM",
IDH: 0.67}
     fmt.Println("Cidade",ci1)
     ci2 := Cidade{}//Criando uma struct Cidade vazia
     fmt.Println("Cidade 2",ci2)
}
```

O resultado é:

```
Cidade {Tefé AM 45000 0.67}
Cidade 2 {  0 0}
```

Acessando propriedades de struct diretamente

Podemos acessar qualquer propriedade de uma struct usando o operador '.' após a variável do tipo struct. Veja o exemplo abaixo:

```
package main
import "fmt"
type Cidade struct{
        Nome string
        Estado string
        População int
        IDH     float64
}
func main() {
         var ci1 = Cidade{Nome:"Tefé", População: 45000,Estado: "AM",
IDH: 0.67}
        fmt.Println("Nome da Cidade :",ci1.Nome)
        fmt.Println("População :",ci1.População)
        fmt.Println("Índice de Desenvolvimento Humano: ",ci1.IDH)
}
```

O resultado é:

```
Nome da Cidade : Tefé
População : 45000
Índice de Desenvolvimento Humano:  0.67
```

Pointers de struct

Go permite o acesso direto aos campos de uma struct através de um pointer sem a necessidade de derreferenciarmos explicitamente com *. Vejamos como no exemplo abaixo.

```
package main
import "fmt"
type Ponto3D struct{
        X, Y, Z float64
        M string
}
func main() {
        var pt1 = Ponto3D{X:345678, Y: 7891011,Z: 345, M: "Alvo 1"}
        pt1ptr:= &pt1// pointer do ponto pt1
        fmt.Println(pt1ptr)
        fmt.Println(pt1ptr.X)
}
```

O resultado é:

```
&{345678 7.891011e+06 345 Alvo 1}
345678
```

Usando new() com struct

Uma forma mais efetiva de criarmos um pointer de uma struct é usando a função new().

```
package main
import "fmt"
type Ponto3D struct{
        X, Y, Z float64
        M string
}
func main() {
        var pt1ptr = new(Ponto3D)// pointer para uma Struct Ponto3D
        fmt.Println(pt1ptr)
        pt1ptr.X=234567
        pt1ptr.Y=9876543
        pt1ptr.Z=210
        pt1ptr.M="Pico do Jabuti"
        fmt.Println(pt1ptr)
}
```

O resultado é:

```
&{0 0 0 }
&{234567 9.876543e+06 210 Pico do Jabuti}
```

Methods

Um method não é nada mais do que uma função mas que pertence a um tipo, ele é declarado da mesma forma que uma função mas com uma informação a mais antes do nome do método que é o seu recebedor ou receiver. Geralmente o recebedor é uma struct mas pode ser qualquer tipo.

Criando um method

Vamos criar um method para a struct criada na seção anterior.

```
package main
import "fmt"
type Ponto3D struct{
  X, Y, Z float64
  M string
}
func (p3 Ponto3D) éMaisAltoQue(z float64) bool{
  return p3.Z >  z
}
func main() {
  var pt1ptr = new(Ponto3D)// pointer para uma Struct Ponto3D
  fmt.Println(pt1ptr)
  pt1ptr.X=234567
  pt1ptr.Y=9876543
  pt1ptr.Z=210
  pt1ptr.M="Pico do Jabuti"
  fmt.Println(" Acima de 300 metros? ",pt1ptr.éMaisAltoQue(300))
  fmt.Println(pt1ptr)
}
```

O resultado é:

```
&{0 0 0 }
Acima de 300 metros?  false
&{234567 9.876543e+06 210 Pico do Jabuti}
```

Methods usando pointers como recebedores

Para que um method altere o valor de uma struct esse deve usar um pointer dessa struct como recebedor. Fazemos isso usando:

```
package main
import "fmt"
type Ponto3D struct{
  X, Y, Z float64
  M string
}
func (p3 Ponto3D) éMaisAltoQue(z float64) bool{
  return p3.Z >  z
}
func (p3 *Ponto3D) translatarPonto(x ,y, z float64){
  p3.X = p3.X + x
  p3.Y = p3.Y + y
  p3.Z = p3.Z + z
}
func main() {
  var pt1ptr = new(Ponto3D)// pointer para uma Struct Ponto3D
  fmt.Println(pt1ptr)
  pt1ptr.X=234567
  pt1ptr.Y=9876543
  pt1ptr.Z=210
  pt1ptr.M="Pico do Jabuti"
  fmt.Println(pt1ptr)
  fmt.Println("Acima de 300 metros? ",pt1ptr.éMaisAltoQue(300))
  pt1ptr.translatarPonto(102321,500000,100)
  fmt.Println(pt1ptr)
  fmt.Println("Acima de 300 metros? ",pt1ptr.éMaisAltoQue(300))
}
```

O resultado é:

```
&{0 0 0 }
&{234567 9.876543e+06 210 Pico do Jabuti}
Acima de 300 metros?  false
&{336888 1.0376543e+07 310 Pico do Jabuti}
Acima de 300 metros?  true
```

Interfaces

Vamos explicar o que é uma interface em etapas. Primeiramente ela um tipo que pode ser qualquer coisa e nos permite assim, passar qualquer tipo para uma função que tenha uma interface como argumento. Por exemplo:

```
package main
import (
    "fmt"
)
func umaFunção(argumento interface{}) {
```

```
    fmt.Print(argumento)
    fmt.Printf(" tipo do argumento %T \n",argumento)
}
func main() {
    var idade int = 51
    var sexo string = "Masculino"
    var saldo float64 = 1.25
    var bonito bool = true
    umaFunção(idade)
    umaFunção(sexo)
    umaFunção(saldo)
    umaFunção(bonito)
}
```

O resultado é:

```
51 tipo do argumento int
Masculino tipo do argumento string
1.25 tipo do argumento float64
true tipo do argumento bool
```

Passamos diversos tipos como argumento para a função `minhaFunção` e o tipo interface soube como agir com cada um destes tipos sem dar um erro. Isso mostra a versatilidade quando usamos interface.

Criando uma interface

Uma interface vai funcionar como um contrato dizendo que todo tipo a ser criado baseado nessa interface vai ter de ter os métodos definidos na interface (Implementação). Exemplo:

```
package main
import (
    "fmt"
    "math"
)
type Forma interface {
  Área() float64
  Perímetro() float64
}
type Retângulo struct {
 Comprimento, Largura float64
}
func (r Retângulo) Área() float64 {
 return r.Comprimento * r.Largura
}
func (r Retângulo) Perímetro() float64 {
 return 2 * (r.Comprimento + r.Largura)
}
type Círculo struct {
 Raio float64
}
func (c Círculo) Área() float64 {
 return math.Pi * c.Raio * c.Raio
}
func (c Círculo) Perímetro() float64 {
 return 2 * math.Pi * c.Raio
}
```

```
func (c Círculo) Diâmetro() float64 {
 return 2 * c.Raio
}
func main() {
 var f1 Forma = Círculo{1.5}
 fmt.Printf("Tipo da Forma = %T, Valores = %v\n", f1, f1)
 fmt.Printf("Área = %f, Perímetro = %f\n", f1.Área(), f1.Perímetro())
 var f2 Forma = Retângulo{2.0, 3.0}
 fmt.Printf("Tipo da Forma = %T, Valores = %v\n", f2, f2)
 fmt.Printf("Área = %f, Perímetro = %f\n", f2.Área(), f2.Perímetro())
}
```

O resultado é:

```
Tipo da Forma = main.Círculo, Valores = {1.5}
Área = 7.068583, Perímetro = 9.424778
Tipo da Forma = main.Retângulo, Valores = {2 3}
Área = 6.000000, Perímetro = 10.000000
```

As duas struct satisfazem o contrato da interface Forma, com os métodos Área() e Perímetro() implementados.

Agrupando struct por interface

Caso queira, podemos agrupar diferente tipos (struct) baseado na interface em comum. Vejamos como:

```
package main
import (
    "fmt"
    "math"
)
type Forma interface {
     Área() float64
     Perímetro() float64
}
type Retângulo struct {
     Comprimento, Largura float64
}
func (r Retângulo) Área() float64 {
     return r.Comprimento * r.Largura
}
func (r Retângulo) Perímetro() float64 {
     return 2 * (r.Comprimento + r.Largura)
}
type Círculo struct {
     Raio float64
}
func (c Círculo) Área() float64 {
     return math.Pi * c.Raio * c.Raio
}
func (c Círculo) Perímetro() float64 {
     return 2 * math.Pi * c.Raio
}
func (c Círculo) Diâmetro() float64 {
     return 2 * c.Raio
}
func main() {
```

```
    var f1 Forma = Círculo{1.5}
    var f2 Forma = Retângulo{2.0, 3.0}
    var formas []Forma
    formas = append(formas,f1)
    formas = append(formas,f2)
    fmt.Println("Área de f1 é ",formas[0].Área())
    fmt.Println("Área de f2 é ",formas[1].Área())
    fmt.Println("f1 ",formas[0])
    fmt.Println("f2 ",formas[1])
}
```

O resultado é:

```
Área de f1 é  7.0685834705770345
Área de f2 é  6
f1  {1.5}
f2  {2 3}
```

Erros

Em Go um erro é somente um valor que uma função retorna se algo inesperado acontecer e der problema. O tipo error é construído em Go originalmente e seu valor inicial é `nil` pois o mesmo se trata de uma interface globalmente disponível.

```
type error interface {
Error() string
}
```

Como funciona

Qualquer tipo que implementa a interface error é como um erro. Então vamos criar o nosso primeiro erro implementando a interface error.

```
package main
import "fmt"
type MeuErro struct{}
//agora implementando o erro na Função Error() da interface error
func (err *MeuErro) Error() string{
   return "Algo errado aconteceu!"
}
func main() {
   meuErr :=&MeuErro{}//criamos uma struct
   fmt.Println(meuErr)//mostrando a mensagem de error
}
```

O resultado é:

```
Algo errado aconteceu!
```

No exemplo acima criamos uma `struct` do tipo `MeuErro` que implementa um method `Error` que retorna uma string, Desta maneira a struct `MeuErro` implementa a interface `error`. Resumindo, `meuErr` passa a ser um erro agora. A função `fmt.Println` é foi feita de tal forma que chama o função `Error` automaticamente quando o valor é um erro, e assim, a mensagem `Algo deu errado!` é mostrada.

Mas para criar um simples erro precisamos criar uma struct e implementar nela o method `Error()`. Para evitar essa mão de obra toda Go tem o pacote errors que exporta a função `errors.New`. Essa função espera uma Mensagem de erro e retorna um erro. Veja abaixo:

```
package main
import "fmt"
import "errors"
func main() {
   meuErr :=errors.New("Algo deu errado!")//criamos uma struct
   fmt.Println(meuErr)//mostrando a mensagem de error
}
```
O resultado é:
```
Algo errado aconteceu!
```
Um exemplo na prática

Vamos ver na prática como isso funciona. Veremos mais sobre como usar error a nosso favor na parte 2 deste livro.
```
package main
import "fmt"
import "errors"
func esquerda(direção int) (bool,error){
   if direção ==0{
        return true,nil
   }else if direção == 1{
     return true,nil
   }else {
     return false,errors.New("Proibido virar a esquerda!")
   }
}
func main() {
   permitido,erro := esquerda(0)
   fmt.Print("Pode seguir em frente? ",permitido)
   fmt.Println(" Erro é :", erro)
   permitido,erro = esquerda(1)
   fmt.Print("Pode virar a direita? ",permitido)
   fmt.Println(" Erro é :", erro)
   permitido,erro = esquerda(2)
   fmt.Print("Pode virar a esquerda? ",permitido)
   fmt.Println(" Erro é :", erro)
}
```
O resultado é:
```
Pode seguir em frente? true Erro é : <nil>
Pode virar a direita? true Erro é : <nil>
Pode virar a esquerda? false Erro é : Proibido virar a esquerda!
```
Prática 3

3.1 - Pedra, Papel ou tesoura?

Vamos solidificar os conceitos que aprendemos programando este jogo simples:
```
package main
import (
 "fmt"
 "errors"
 "time"//para semente de números aleatórios
 "math/rand" //para gerar números aleatórios
```

```
)
type Jogadores interface{ //criando a interface Jogadores
  joga() (int,error)
  vêScore() (int,error)
}
type Computador struct{ //jogador computador
  jogada, score int
}
func (c *Computador) joga() (int,error){
  semente := rand.NewSource(time.Now().UnixNano())
  rnd := rand.New(semente)
  jogada := rnd.Intn(3)
  return jogada,nil
}
func (c *Computador) vêScore()(int){
  return c.score
}
type Jogador struct{ //jogador você
  jogada, score int
}
func (j *Jogador) joga() (int,error){
   var ap int
   fmt.Print("Papel 0, Tesoura 1 ou Pedra 2?")
   _, err := fmt.Scanf("%d", &ap)
   if  err ==nil && ap >=0 && ap<3 {
     return ap,nil
   }else{
     return -1,errors.New("Jogada invalida!")
  }
}
func (j *Jogador) vêScore()(int){
  return j.score
}
func checa(f int,c int) string{ //função que checa a rodada
 switch f{
  case 0:
   fmt.Print("Você Jogou Papel e o ")
  case 1:
   fmt.Print("Você Jogou Tesoura e o ")
  case 2:
   fmt.Print("Você Jogou Pedra e o ")
 }
 switch c{
  case 0:
   fmt.Println("Computador Jogou Papel")
  case 1:
   fmt.Println("Computador Jogou Tesoura")
  case 2:
   fmt.Println("Computador Jogou Pedra")
 }
 switch c{
  case 0:
   if f==0 {return "Empate"}
   if f==1 {return "Ganhou"}
```

```
   if f==2 {return "Perdeu"}
  case 1:
   if f==0 {return "Perdeu"}
   if f==1 {return "Empate"}
   if f==2 {return "Ganhou"}
  case 2:
   if f==0 {return "Ganhou"}
   if f==1 {return "Perdeu"}
   if f==2 {return "Empate"}
 }
 return "nada"
}
func main() {
  var erro error
  fu:=new(Jogador)
  ral:=new(Computador)
jogo:
  ral.jogada,_=ral.joga()
  fu.jogada,erro =fu.joga()
  if erro == nil{
    res :=checa(fu.jogada,ral.jogada)
    if res=="Empate"{
       fmt.Println("EMPATE!!!")
    }
    if res=="Ganhou"{
       fmt.Println("GANHOU!!!")
       fu.score++
    }
    if res=="Perdeu"{
       fmt.Println("PERDEU!!!")
       ral.score++
    }
    fmt.Printf("PLACAR  Você %d computador %d \n",fu.score,ral.score)
  }else{fmt.Println(erro)}
 goto jogo
}
```

Goroutines

Em Go, cada atividade executada ao mesmo tempo em um programa e chamada de goroutine. Imagine um programa com duas funções e cada uma executa uma atividade independentemente da outra. Um programa sequencial chamará uma função e depois a outra, mas num programa concorrente (com duas ou mais goroutines) chamadas para as duas podem estar em execução ao mesmo tempo.

Quando um programa se inicia a única goroutine em execução é a função main(). Novas goroutines são criadas usando a instrução `go` fazendo com que esta função seja uma nova goroutine.

Se chamamos uma função `minhaFunção()` o programa aguarda a execução desta.
Se chamamos uma função com `go minhaFunção()` o programa seguirá sem aguardar pelo resultado desta função.

Criando uma goroutine

Considere o seguinte programa onde criamos uma goroutine que é executada enquanto o programa está calculando um valor (que demora a ser concluído).

```
package main
import (
    "fmt"
    "time")
func gira(atrasa time.Duration) {
  for {
    for _, barrinha := range `-\|/` {
      fmt.Printf("\r%c", barrinha)
      time.Sleep(atrasa)
    }
  }
}
func fibonacci(x int) int {
  if x < 2 {
    return x
  }
  return fibonacci(x-1) + fibonacci(x-2)
}
func main() {
  go gira(100 * time.Millisecond)
  fN := fibonacci(45)//algo meio demorado de retornar
  fmt.Printf("\rFibonacci(45) = %d\n", fN)
}
```

O resultado é (além do indicador de giro):

```
Fibonacci(45) = 1134903170
```

Quando o função `fibonacci` é concluída e retornamos para `main()` o programa finaliza e finaliza também a função `gira()` que entrou em execução de forma concorrente usando `go` antes de chamarmos `fibonacci`.

Channel

Para se comunicar entre duas goroutines Go usa um mecanismo de comunicação bastante eficiente chamado channels . Channel funciona como um encanamento entre duas goroutines.

Criando um channel

Criamos um channel usando o tipo `chan` e usamo make para criar o channel.

```
cInteger  :=  make(chan int)
cInterface := make(chan interface{})
```

Channel usa o operador <- para receber e passar dados

```
ch  <-  v //envia v para o channel ch
v  := <- ch //recebe o dado do channel ch e atribui a v
```

Vejamos um exemplo de como usar channel:

```
package main
import "fmt"
func sum(a []int, c chan int) {
```

```
  total := 0
  for _, v := range a {
    total += v
  }
  c <- total // envia o total para c na pilha
}
func main() {
  a := []int{7, 2, 8, -9, 4, 2}
  c := make(chan int)
  go sum(a[:3], c) //soma dos 3 últimos valores de a
  go sum(a[3:], c) //soma dos 3 primeiros valores de a
  go sum(a[0:6],c) // soma de todos o valores de a
  x, y, z := <-c, <-c, <-c // recebe o valor de c da pilha
  fmt.Println(x, y, z)
}
```

O resultado é:

```
14 17 -3
```

Como podemos ver, o channel funciona como uma pilha, o primeiro a entrar no channel é o último sair dele.

Channel com buffer

Podemos delimitar a quantidade de itens que podemos alimentar um channel, isso se chama colocar um buffer no channel ou bloqueio.

```
c := make(chan int,3)
```

Vejamos abaixo como criar um channel com buffer limitado em 2.

```
package main
import "fmt"
func main() {
 c := make(chan int, 2) //channellimitado a dois elementos
 c <- 1
 c <- 2
 fmt.Println(<-c)
 fmt.Println(<-c)
}
```

O resultado é:

```
1
2
```

Select

Select funciona como um switch para channel. Vejamos o exemplo abaixo:

```
package main
import "fmt"
import "time"
func olá(ch1 chan string) {
  for {
    ch1 <- "Olá"
    time.Sleep(time.Second * 2)
  }
}
func pateta(ch2 chan string) {
    for {
```

```
      ch2 <- "Pateta"
      time.Sleep(time.Second * 3)
    }
}
func seleciona(ch1, ch2 chan string) {
    for {
      select {
        case mensagem1 := <- ch1:
        fmt.Println(mensagem1)
        case mensagem2 := <- ch2:
        fmt.Println(mensagem2)
      }
    }
}
func main() {
  ch1 := make(chan string)
  ch2 := make(chan string)
  go olá(ch1)
  go pateta(ch2)
  go seleciona(ch1,ch2)
  var aguarda string
  fmt.Scanln(&aguarda)
}
```

O resultado é:

```
Pateta
Olá
Olá
Pateta
Olá
...
```

O programa imprime "Olá" a cada 2 segundos e "pateta" a cada 3 segundos. select pega o primeiro channel que está pronto e recebe dele (ou envia para ele). Se mais de um dos channels estão prontos ele aleatoriamente escolhe um deles. Se nenhum está pronto ele bloqueia até um ficar pronto.

Packages

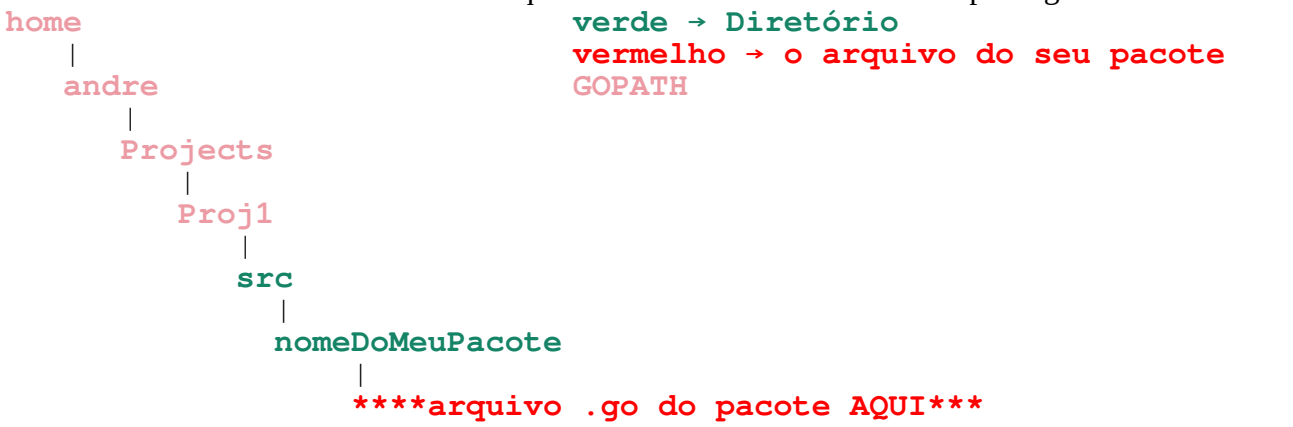
Um package é constituído por arquivos dentro de um mesmo diretório e iniciam com o mesmo nome de pacote na primeira linha do código `package nomePacote`. Alguns pacotes estão disponíveis na biblioteca padrão de linguagem Go (fmt, time, os, encoding, etc...). Outros podem ser instalados com o comando `go get`. E podemos construir nossos próprios packages personalizados. Veremos agora como fazer isso.

Estrutura de um package

Para criarmos um package (pacote) personalizado precisamos primeiramente identificar qual é o diretório GOPATH em seu computador. É dentro deste diretório que devemos instalar nosso package. Na linha de comando digite `echo $GOPATH` e algo do tipo deverá aparecer:

```
/home/andre/Projects/Proj1
```

É dentro do diretório src deste diretório que criaremos o diretório do nosso package. No meu caso é:



O package personalizado terá o nome `nomeDoMeuPacote` e todo arquivo go dentro vai iniciar com `package nomeMeuDoPacote` (nesse caso).

Criando um package

Primeiro, no diretório GOPATH/src crie um diretório com o nome de `pateta`. Dentro deste diretório crie o arquivo mensagem.go abaixo:

```
package pateta
import "fmt"
func Olá() {
    fmt.Println("Olá Pateta!")
}
```

Agora digite, de dentro do diretório `pateta`:

```
go install
```

Pronto! O pacote personalizado está pronto, agora vamos utilizar ele. Em qualquer lugar crie o arquivo `testePacote.go` conforme abaixo:

```
package main
import "pateta"

func main() {
    pateta.Olá()
}
```

O resultado é:

```
Olá Pateta!
```

NOTA - Lembre-se que para uma função ou tipo de um pacote ser acessível externamente a função ou o tipo TEM QUE INICIAR COM LETRA MAIÚSCULA.

Vamos agora concluir a Parte 1 deste livro com a última prática. Neste ponto você já tem toda a base para escrever programas em Go.

Prática 4

4.1 – Um package que lê um arquivo CSV e grava em formato JSON

Vamos reforçar com essa prática o uso de packages, seja interno ou personalizado. Neste exercício vamos ler um arquivo do tipo texto separado por vírgulas (CSV) e vamos gravar o seu conteúdo no formato JSON que é um formato padrão para intercâmbio de dados, principalmente na web.

Será um pacote bem simples e rudimentar mas que serve para mostrar como podemos facilmente interagir com dados usando Go.

Crie o arquivo CSVcom o nome `teste.csv` na mesma pasta que o programa arquivo `prat4.go` mostrado no final do exercício:

```
aluno,nota1,nota2,nota3,nota4
João,4.5,8.9,7.4,5.8
Maria,6.7,9.8,7.8,6.0
```

Vamos criar nosso package conforme explicado anteriormente. Crie na pasta `$GOPATH/src/` uma pasta `reformata` e dentro dela crie estes dois arquivos abaixo:

csv2mat.go

```
package reformata
import (
  "os" //pacote interno que lê e escreve arquivos
  "encoding/csv")//pacote interno que manipula dados csv
func CSVpraMatrix2D(arq string)[][]string {
  arquivo, erro := os.Open(arq) //lê o arquivo csv
  if erro != nil {
    println("Falha ao abrir arquivo teste.csv:", erro)
  }
  defer arquivo.Close() //fecha o arquivo no final da função
  r := csv.NewReader(arquivo)//transforma o arquivo csv em [][]string
  rows, erro2 := r.ReadAll()
  if erro2 != nil {
          println("Cannot read CSV data:", erro2.Error())
     }
  return rows
}
```

mat2json.go

```
package reformata
import "os" //pacote que lê escreve arquivos
func Matrix2DparaJSON(dados [][]string,arq string){
  nomeDados := "dados"
  colunas := len(dados[0])
  linhas := len(dados)
  a := "{\""+nomeDados+"\":[" //formatando em JSON básico
  for i := 1;i < linhas;i++{
    if i == 1 {a=a+"{"
    }else{a = a+",\n{"}
    for j:= 0; j < colunas; j++{
       if j !=(colunas -1){
        a = a + "\"" + dados[0][j] + "\""+":\"" + dados[i][j] + "\","
       }else{
```

```
            a = a + "\"" + dados[0][j] + "\""+":\"" + dados[i][j] + "\""
          }
        }
       a = a+"}"
      }
      a = a+"]}\n" //final da formatação do JSON na string a
      arquivo, erro := os.Create(arq) //cria arquivo teste.json
      defer arquivo.Close()//fecha arquivo no final da função
      if erro != nil {
          println(erro)
          return
      }
      l,erro := arquivo.WriteString(a)//escreve string a no arquivo
      if erro != nil {
          println(erro,l)
          return
      }
     println("Arquivo JSON criado!")//PRONTO
}
```

Após criar estes dois arquivos na pasta `$GOPATH/src/reformata` executamos de dentro dela:

```
go install
```

Agora vamos criar um programa simples que lê o nosso arquivo CSV e converte o mesmo para um arquivo JSON.

prat4.go

```
package main
import "reformata"
func main() {
  dados := reformata.CSVpraMatrix2D("teste.csv")
  reformata.Matrix2DparaJSON(dados,"teste.json")
}
```

Finalmemte, se abrirmos o arquivo `teste.json` veremos:

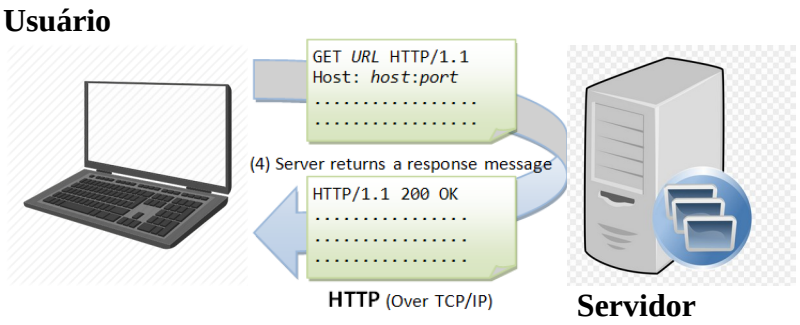
```
{"dados":[{"aluno":"João","nota1":"4.5","nota2":"8.9","nota3":"7.4","nota4":"5.8"},
{"aluno":"Maria","nota1":"6.7","nota2":"9.8","nota3":"7.8","nota4":"6.0"}]}
```

Parte 2 - Desenvolvimento para a Web Usando Go

Go é ideal para criar aplicativos web e agora vamos mostrar como fazer isso. Inicialmente vamos criar nosso primeiro aplicativo web que pouco faz (com poucas linhas de código) e a partir daí criarmos os aplicativos web um pouco mais robustos.

Como funciona a web

De forma bem simples, a web funciona com um pedido de página feito pelo computador do usuário para um servidor da web e este envia uma resposta com um conteúdo de volta para o usuário.



Então vamos precisar de um servidor de páginas web.

Criando um Servidor de aplicativos web com Go.

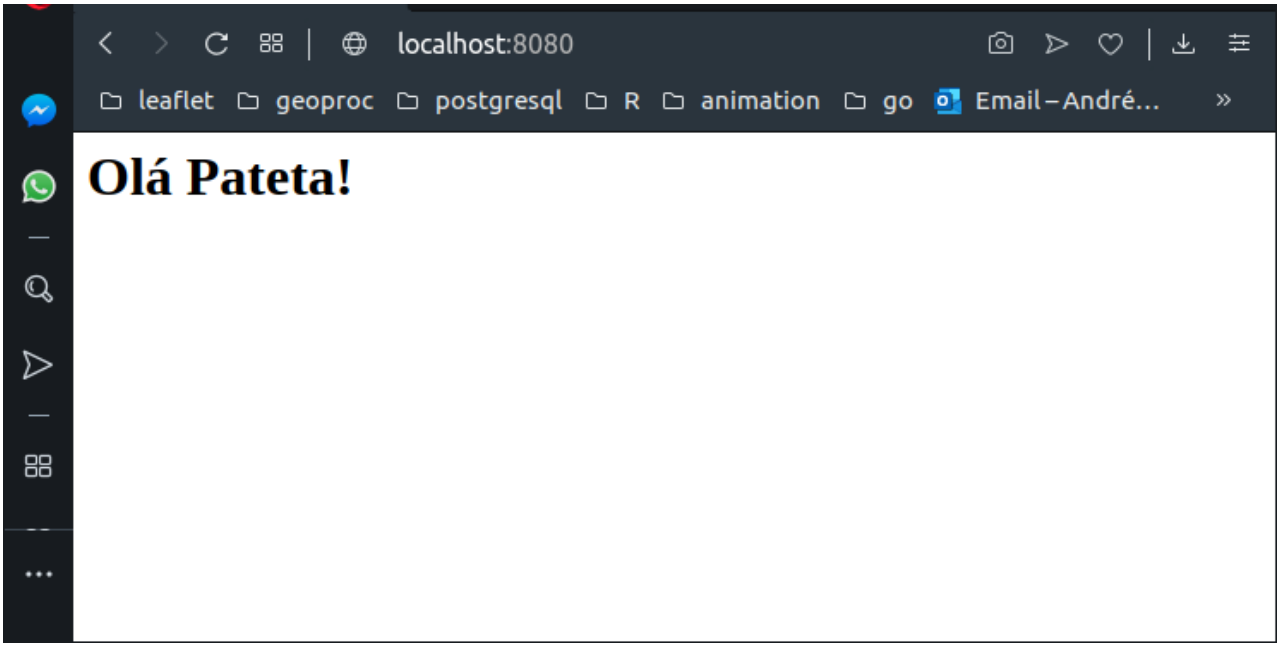
Primeiro crie o diretório `webapp1` dentro do diretório `$GOPATH/src`. Dentro desse diretório crie o arquivo `servidor.go` conforme abaixo:

```
package main
import (
"fmt"
"net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
fmt.Fprintf(w, "<h1>Olá Pateta! %s </h1>", r.URL.Path[1:])
}
func main() {
http.HandleFunc("/", handler)
http.ListenAndServe(":8080", nil)
}
```

Caso seu GOPATH esteja correto, construa o servidor digitando em qualquer local:
`go install webapp1`

Agora vamos acionar nosso servidor web com:
`$GOPATH/bin/webapp1`

Abra seu navegador de preferência e digite localhost:8080 nele. A seguinte página com o nosso primeiro aplicativo web em Go deverá aparecer:



Se digitarmos localhost:8080/vamos/programar/em/Go veremos:

Não faz muita coisa esse aplicativo web mas mostra como é simples criar um. Vamos criar algo mais interessante nos próximos exemplos .

Criando um formulário web

Vamos mostrar como criar um formulário da web e processar o resultado do envio usando exclusivamente Go.

Crie a pasta webapp2 no `$GOPATH/src` da mesma forma que fizemos anteriormente e crie o seguinte arquivo servidor.go dentro desta pasta:

```
package main
import (
 "fmt"
 "net/http"
)
func digaOlá(w http.ResponseWriter, r *http.Request) {
fmt.Fprintf(w,"<h1>Olá</h1>Experimente  o  <a  href='login'>Formulário
</a>")
}
func login(w http.ResponseWriter, r *http.Request) {
  pagina1 := "<html><head><title>Login do site</title></head>"+
   " <body><form action='/login' method='post'>"+
   "Nome:<input type='text' name='nome'><br><br>"+
   "Sobrenome:<input type='text' name='sobrenome'><br><br>"+
   "<input type='submit' value='Entrar'></form></body></html>"
  if r.Method == "GET" {
   fmt.Fprintf(w, pagina1)
  }else {
    r.ParseForm()
    fmt.Fprintf(w, "<h1>Olá %s </h1> <h1> Seu sobrenome é %s </h1>",
               (r.Form["nome"])[0],(r.Form["sobrenome"])[0])
  }
}
func main() {
  http.HandleFunc("/", digaOlá)
  http.HandleFunc("/login", login)
  http.ListenAndServe(":8080", nil)
}
```

Instale o servidor digitando :

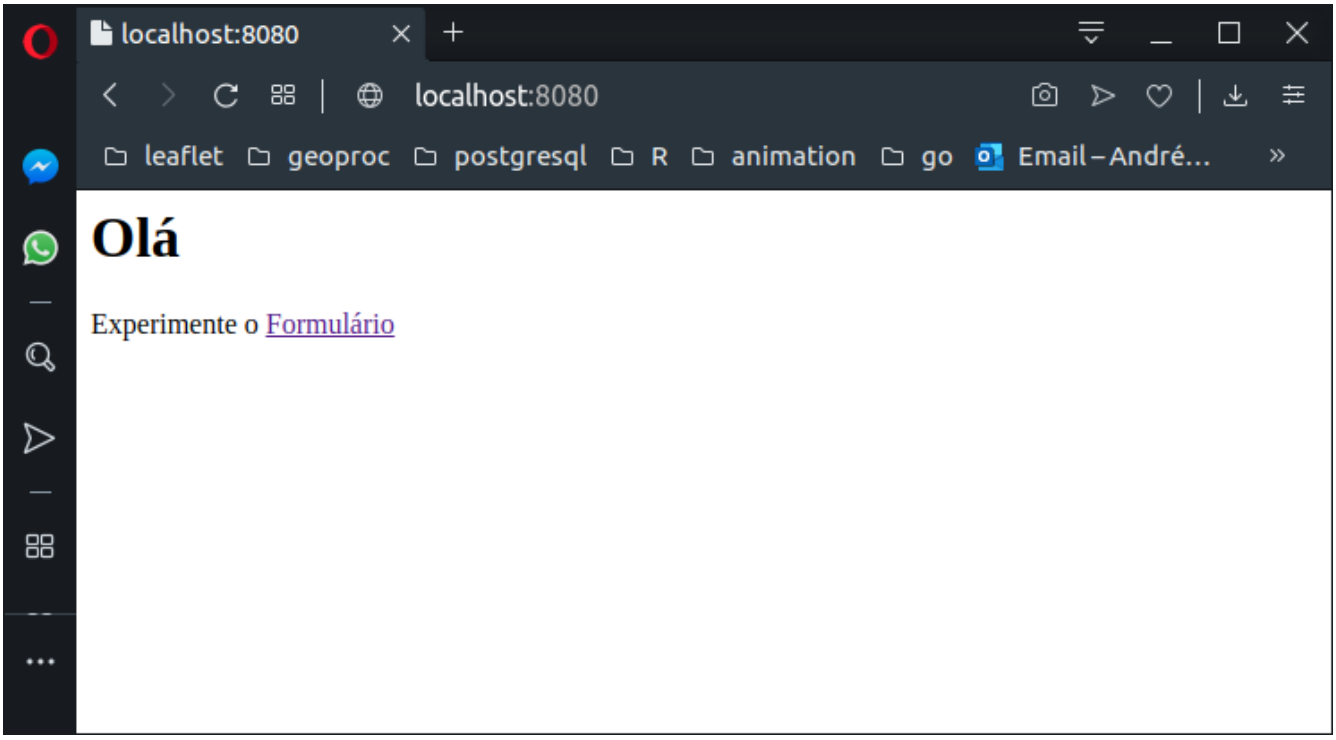
```
go install webapp2
```

Inicie o servidor com:

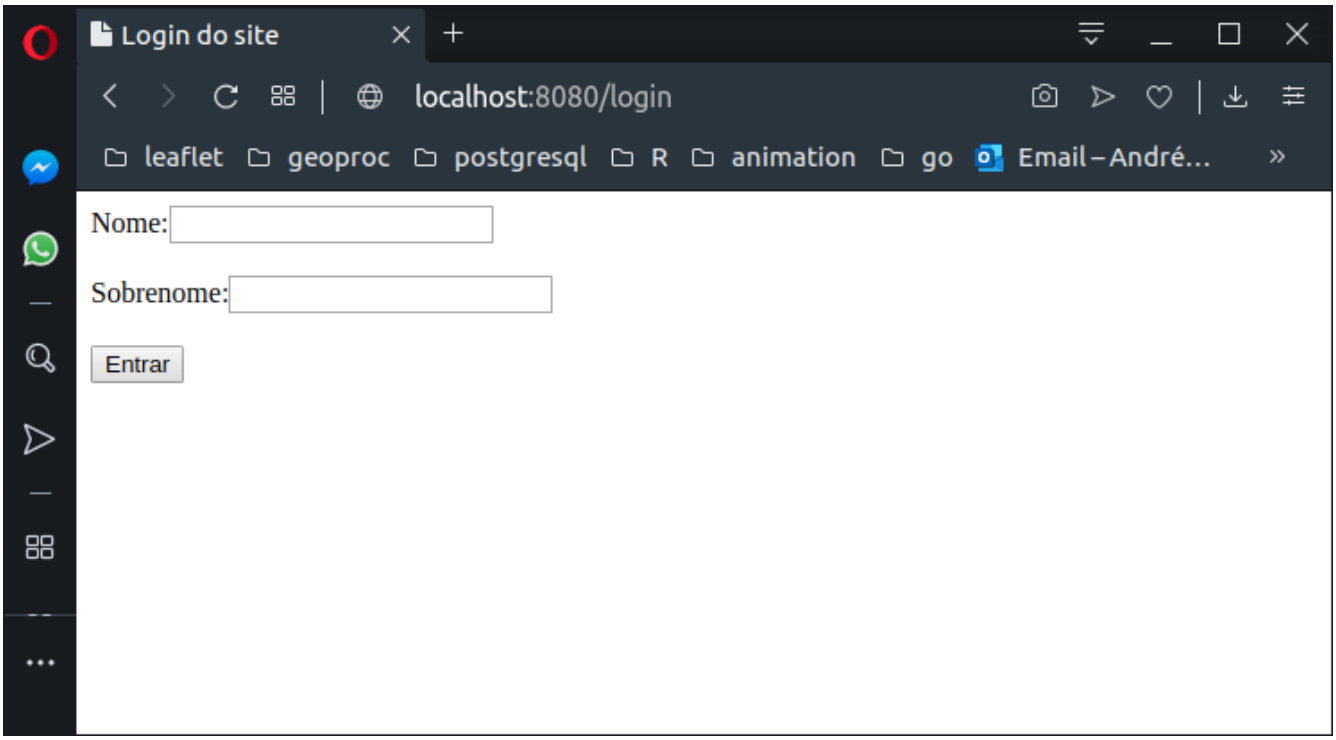
```
$GOPATH/bin/webapp2
```

Este programa incrementa o anterior adicionando mais uma função handle para **`/login`** chamada `login` além da função handle para **`/`** chamada `digaOlá`.
`digaOlá` apresenta uma mensagem e o link de acesso para `/login`
Quando **`/login`** é acessado a função `login` assume e analisa se o tipo do método é GET ou POST.
Se for GET o formulário é apresentado, se for POST (chamado do próprio botão “Entrar” do formulário) o resultado é apresentado.
Nenhuma forma de checagem, além de GET ou POST, foi implementada nesse exemplo.

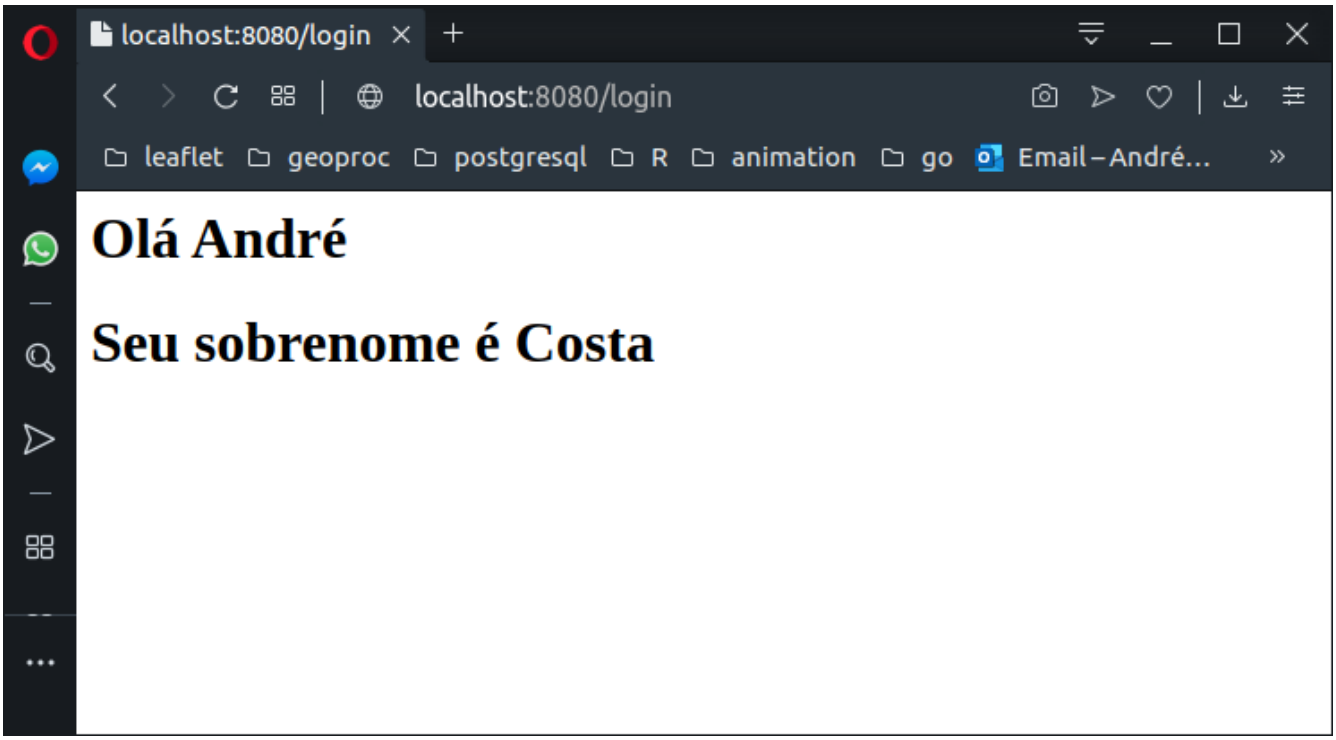
A tela do nosso webapp quando acessamos usando localhost:8080 no navegador é:



Ao clicarmos no link Formulário temos (método GET):



Ao enviarmos o formulário com os campos preenchidos (ou não) usando o botão “Entrar” teremos: (método POST):



Formulários e Banco de Dados

Hoje em dia não tem como falar em aplicativo web sem falar sobre banco de dados. Vamos somente pincelar sobre banco de dados PostgreSQL uma vez que não é o objetivo deste livro.

Para instalar PostgreSQL siga as inúmeras instruções que existem na web como por exemplo:
- https://www.devmedia.com.br/instalando-postgresql/23364 para windows.
- https://www.devmedia.com.br/instalacao-e-configuracao-do-servidor-postgresql-no-linux/26184 para Linux.
- http://groselhas.maurogeorge.com.br/instalando-o-postgresql-no-mac-os-com-o-homebrew.html#sthash.bIRXnfZs.dpbs no mac OSX

Uma vez instalado e configurado o PostgreSQL crie o banco de dados usando:

```
createdb webapp --encoding=utf8
```

E crie a nossa tabela usando:

```
psql webapp -c "CREATE TABLE membros (id serial primary key, nome varchar(60), sobrenome varchar(60));"
```

Para usarmos o banco de dados PostgresqSQL vamos instalar o driver pq usando

```
go get -u github.com/lib/pq
```

Pronto, isso é tudo que vamos precisar para rodar esse próximo exemplo de aplicação web.

Usando PostgreSQL com aplicações web

Crie o diretório webapp3 no `$GOPATH/src` e dentro dele escreva o arquivo `servidor.go` abaixo:

```
package main
import (
  "net/http"
  "database/sql"
  "fmt"
  _ "github.com/lib/pq"//driver para postgresql
)
const (//credenciais de acesso ao banco de dados e páginas
  sqlInfo = "host=localhost port=5432 user=andre "+
              "password=segredo dbname=webapp sslmode=disable"
  pagina1 = "<html><head><title>Membros</title></head>"+
   " <body><form action='/insere' method='post'>"+
   "Nome:<input type='text' name='nome'><br><br>"+
   "Sobrenome:<input type='text' name='sobrenome'><br><br>"+
   "<input type='submit' value='Inserir'></form>"+
   "<br><a href='.'> <--VOLTAR</a></body></html>"
  pagOlá ="<html><head><title>Membros</title></head><body>"+
     "<h3>Olá</h3>Insira membro no <a href='insere'>Formulário</a>"+
     "<br>Ver membros <a href='ver'> aquí</a></body></html>"
  cabeça = "<html><head><title>Membros</title></head><body><table>"
  pé = "</table><br><a href='.'> <--VOLTAR</a></body></html>"
)
//------------Função que insere registro
func inserindo(nome, sobrenome string){
  db, erro := sql.Open("postgres", sqlInfo)
  if erro != nil {
    panic(erro)
  }
  defer db.Close()
  sqlStatement := `INSERT INTO membros (nome, sobrenome)
  VALUES ($1, $2)RETURNING id`
  id := 0
  erro = db.QueryRow(sqlStatement, nome, sobrenome).Scan(&id)
  if erro != nil {
     panic(erro)
  }
  fmt.Println("ID do registro novo é: ", id)
}
//--------------Função da pagina inicial
func digaOlá(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w,pagOlá)
}
//--------Função da página com formulário de inserção
func insere(w http.ResponseWriter, r *http.Request) {
  if r.Method == "GET" {
    fmt.Fprintf(w, pagina1)
  }else {
    r.ParseForm()
    inserindo((r.Form["nome"])[0],(r.Form["sobrenome"])[0])
    fmt.Fprintf(w,cabeça+ "Membro %s %s inserido!"+pé,
```

```
                (r.Form["nome"])[0],(r.Form["sobrenome"])[0])
  }
}
// --função de página de visualização da tabela do banco de dados
func ver(w http.ResponseWriter, r *http.Request) {
  var vt string
  vt = ""
  db, erro := sql.Open("postgres", sqlInfo)
  if erro != nil {
    panic(erro)
  }
  defer db.Close()
  rows, erro := db.Query("SELECT id, nome, sobrenome FROM membros")
  if erro != nil {
    panic(erro)
  }
  defer rows.Close()
  for rows.Next() {
    var id, nome, sobrenome string
    erro = rows.Scan(&id,&nome,&sobrenome)
    if erro != nil {
       panic(erro)
    }
    vt = vt + "<tr><th>" + id +"</th><th>" + nome +
        "</th><th>" + sobrenome + "</th></tr>"
  }
  fmt.Fprintf(w, cabeça + vt + pé)
}
//-------------------------------Função Main
func main() {
  http.HandleFunc("/", digaOlá)
  http.HandleFunc("/insere", insere)
  http.HandleFunc("/ver", ver)
  http.ListenAndServe(":8080", nil)
}
```

A função main agora tem três handlers apontando para três funções:
- Uma de início onde temos a página inicial (digaOlá);
- Uma função para mostrar a página com formulário (insere). No caso GET o formulário aparece e no caso POST ela chama a função inserindo que adiciona o membro no banco de dados Postgresql e apresenta a página de resultado
- Uma função que mostra todos os membros na tabela do banco de dados (ver).

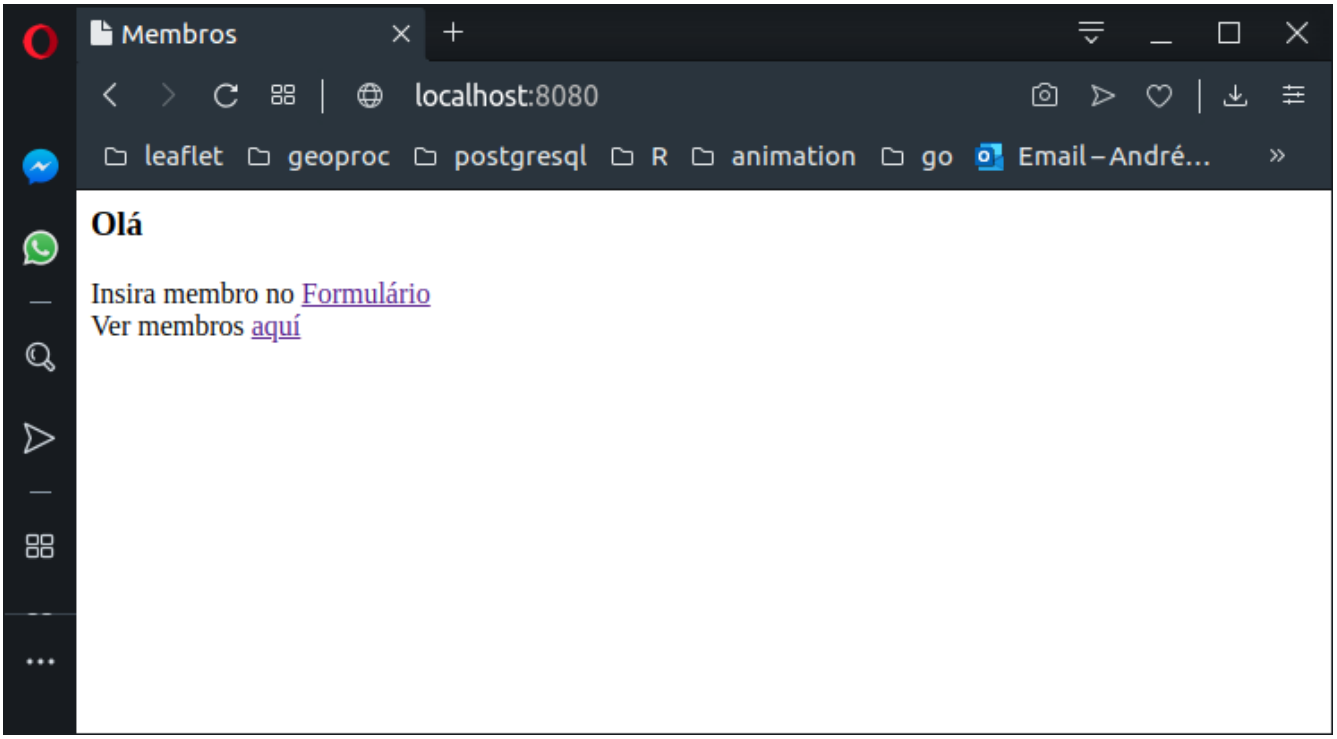
As páginas pre-formatadas foram adicionadas a constantes bem como a string de conexão ao banco de dados, mudar os valores de usuário (user) e senha (password) apropriadamente com o seu sistema.

```
 sqlInfo = "host=localhost port=5432 user=andre "+
              "password=segredo dbname=webapp sslmode=disable"
```

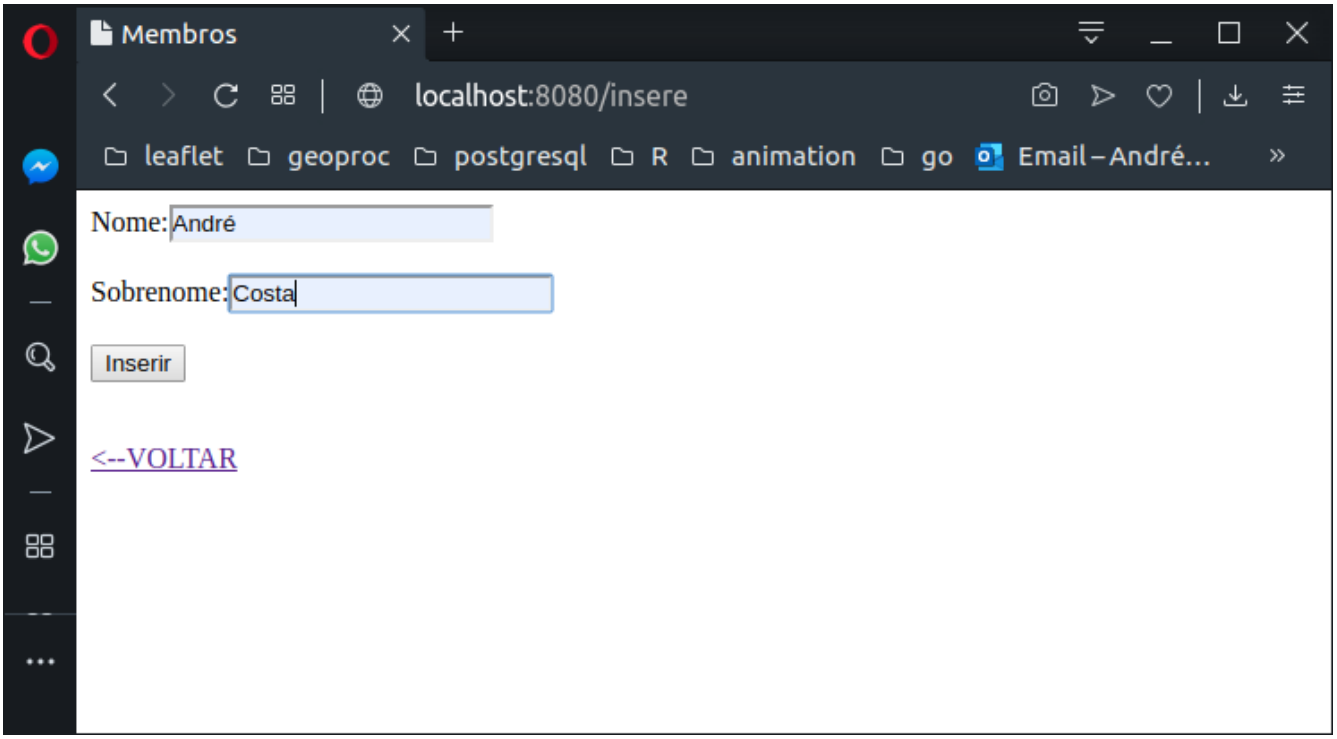
Vamos agora montar e rodar nosso aplicativo web com:

```
go install webapp3
$GOPATH/bin/webapp3
```

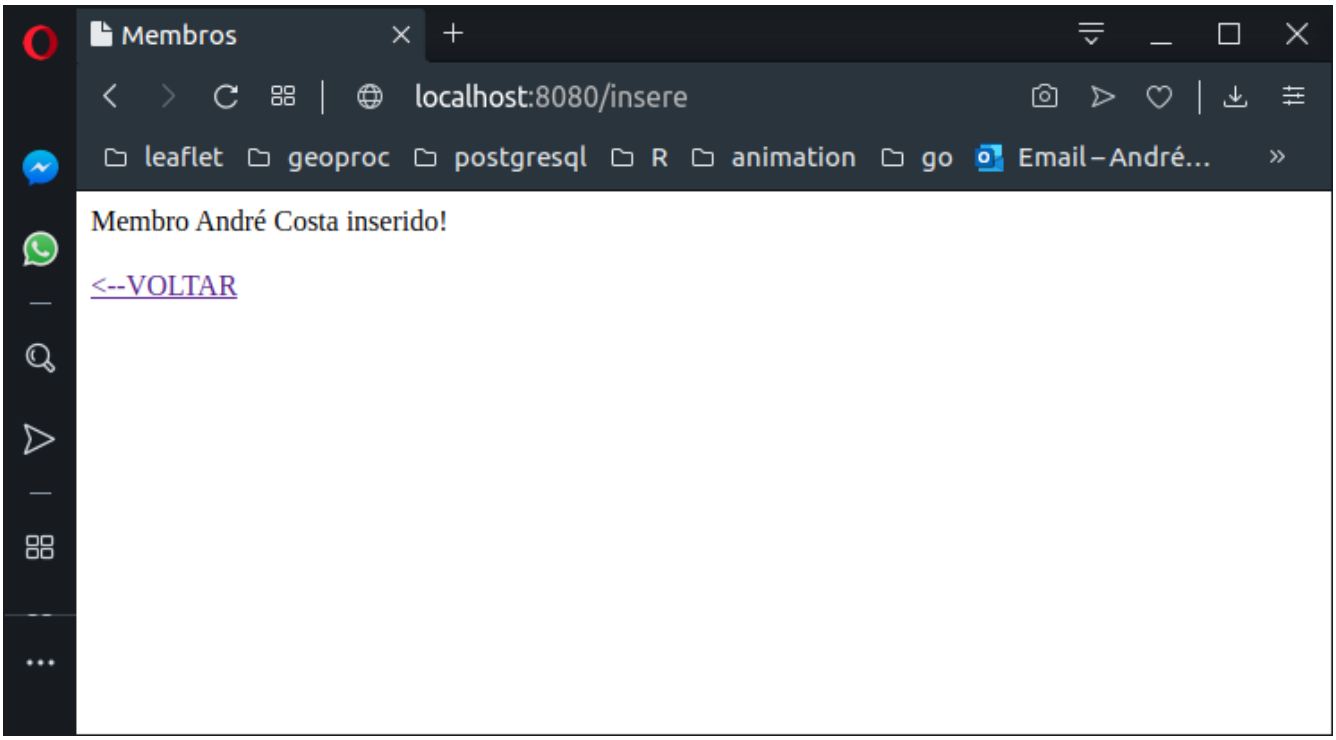
A página inicial com localhost:8080 é:



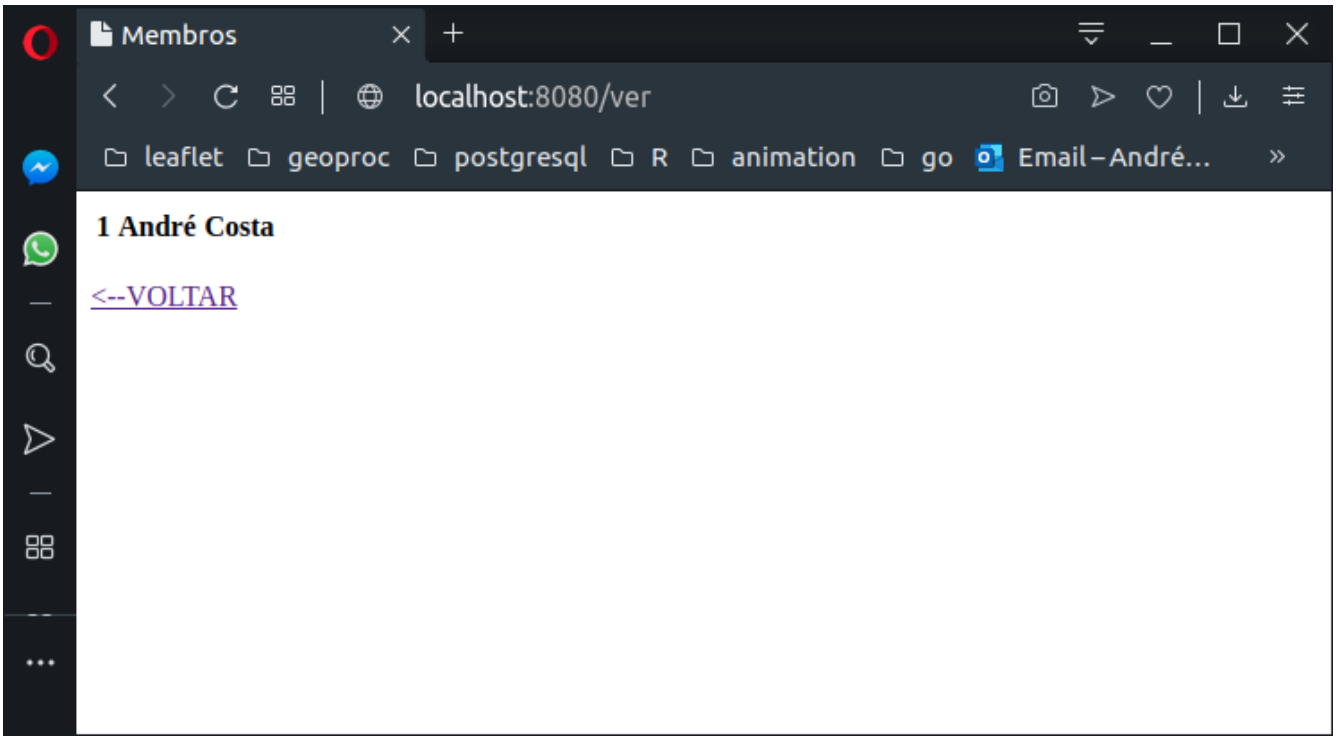
Clicando em formulários e preenchendo os campos:



Apos clicar em Inserir:



E finalmente, voltando a página inicial, e clicando em “Ver membros aqui”:



Inclua mais nomes de membros e teste o aplicativo da web.

Organizando o aplicativo web

Nos exemplos anteriores usamos um único arquivo para todo o site, isso fica mais complicado com o aumento das atribuições do aplicativo web. Vamos ver agora novas maneiras de tirar proveito de capacidades de frontend (do tipo css e javaScript) e backend (do tipo templates).

Arquivos Estáticos

Arquivos estáticos são aqueles que são utilizados pelo aplicativo web e não mudam de conteúdo. Eles podem ser arquivos estilizadores do tipo css, imagens ou scripts tipo javaScript.

Vamos criar uma pasta para o nosso projeto chamada webapp4 e dentro dela vamos criar uma pasta chamada static. Dentro da página static organizamos nossos arquivos estáticos. Vamos usar um arquivo css chamado estilo.css que colocaremos na pasta css que criaremos. A estrutura desse aplicativo web fica assim por enquanto.

```
webapp4
    |
   static
    |  |
    |  css
    |    |
    |     -estilo.css
    |
```

O arquivo estilo.css é mostrado abaixo:

```
body{font-family:Verdana,sans-serif;font-size:0.9em;
  background-color:#cfcfc1;margin:0 auto;max-width:60em;
  background-image: linear-gradient(to bottom right,#cfcfc1,efefaa);}
header,footer{padding:1em;color:white;font-weight:bolder;
  background-color:#8b4513;}
section{margin:0.5em;padding:1em;background-color:#fef9c9;}
article{margin:0.5em;padding:1em;background-color:white;}
nav{padding:0;background-color:#cfcfc81;}
nav ul li {display:inline;margin:0.5em;}
a { color:lightblue; font-weight:bolder;font-size: 1em; }
a:visited { color: lightblue; }
a:hover { color: orange; }
h1 {font-size: 2.2em;font-weight: bolder;line-height: 0.8em;
  padding: 0.6em 0 0.1em 2%;margin: 0;display:inline;}
h2,h3,h4,h5,h6{color: #453229;font-weight:bolder;
font-family: Georgia,"Times New Roman",Times, serif;}|
```

Templates

Vamos agora criar uma pasta chamada templates onde colocaremos os arquivos de nossas páginas que seguirão um layout predefinido.

Nele colocaremos o arquivo layout.html com os posicionadores de contexto dinâmico delimitado por {{ }} e os arquivos pindex.hmtl, insere.html e ver.html que terá o conteúdo dinâmico da página somente demarcado por {{ }}.

A extrutura será:

```
webapp4
    |
   static
    |  |
    |  css
    |    |
    |     -estilo.css
    |
  templates
    |    |
    |     - layout.html
    |     - pindex.html
    |     - insere.html
    |     - ver.html
```

layout.html:

```
{{define "layout"}}
<!DOCTYPE html>
<html lan-="pt-br">
 <head>
 <title>{{template "title"}}</title>
 <meta charset="utf-8" />
 <meta content='width=device-width,initial-scale=1' name='viewport'/>
 <link href="static//css/estilo.css" rel="stylesheet" type="text/css"
       media="screen, projection" />
 </head>
 <body>
 <header>
 <nav><h1>Os Membros</h1>
  <ul>
  <li><a href="ver.html">Ver Membros</a></li>
  <li><a href="insere.html">Inserir Membros</a></li>
  <li><a href="pindex.html">Home</a> </li>
  </ul>
 </nav>
 </header>
  <section>
   <h2>Os Membros</h2>
    <article>
       {{template "body"}}
    </article>
  </section>
  <footer><small><center>Aprendendo Go</center></small></footer>
 </body>
</html>
{{end}}
```

ver.html

```
{{define "title"}}Ver Membros{{end}}
{{define "body"}} {{end}}
```

insere.html

```
{{define "title"}}Insere{{end}}
{{define "body"}}
<h2>Entre com o nome e sobrenome nos campos abaixo</h2>
     <form action='/insere' method='post'><table>
     <tr><td>Nome:</td><td colspan=2>
        <input type='text' name='nome'></td></tr>
     <tr><td>Sobrenome:</td><td colspan=2>
        <input type='text' name='sobrenome'></td></tr>
     <tr><td></td><td><input type='submit' value='Inserir'>
         </td><td></td></tr></table></form>
{{end}}
```

pindex.html

```
{{define "title"}}WEBAPP{{end}}
{{define "body"}}
<h3>Site de aplicativo web</h3>
{{end}}
```

É bastante óbvio aqui que os campos `define title` e `body` dos arquivos pindex.html, ver.html e insere.html irão substituir os campos `template title` e `body` do arquivo layout.html no funcionamento do nosso aplicativo web.

O arquivo Go

Vamos agora finalizar acrescentando na estrutura o nosso arquivo servidor.go:

```
webapp4
    |
   static
    |  |
    |  css
    |    |
    |     -estilo.css
    |
   templates
    |    |
    |     - layout.html
    |     - pindex.html
    |     - insere.html
    |     - ver.html
    |
   servidor.go
```

servidor.go:

```
package main
import (
  "net/http"
  "database/sql"
  "html/template"
  "path/filepath"
  _ "github.com/lib/pq"
)
```

```
const sqlInfo = "host=localhost port=5432 user=andre "+
            "password=devcor dbname=webapp sslmode=disable"

func inserindo(nome, sobrenome string){
  db, erro := sql.Open("postgres", sqlInfo)
  if erro != nil {
    panic(erro)
  }
  defer db.Close()
  sqlStatement := `INSERT INTO membros (nome, sobrenome)
  VALUES ($1, $2)RETURNING id`
  id := 0
  erro = db.QueryRow(sqlStatement, nome, sobrenome).Scan(&id)
  if erro != nil {
     panic(erro)
  }
  println("ID do registro novo é: ", id)
}

func insere(w http.ResponseWriter, r *http.Request) {
  r.ParseForm()
  inserindo((r.Form["nome"])[0],(r.Form["sobrenome"])[0])
  l := filepath.Join("templates", "layout.html")
  f := filepath.Join("templates", "insere.html")
  tpl, _ := template.ParseFiles(l, f)
  tpl.Parse("{{define \"body\"}}"+"Membro Inserido"+"{{end}}")
  tpl.ExecuteTemplate(w, "layout",nil)
}

func ver(w http.ResponseWriter, r *http.Request) {
  var vt string
  vt = "<table>"
  db, erro := sql.Open("postgres", sqlInfo)
  if erro != nil {
    panic(erro)
  }
  defer db.Close()
  rows, erro := db.Query("SELECT id, nome, sobrenome FROM membros")
  if erro != nil {
    panic(erro)
  }
  defer rows.Close()
  for rows.Next() {
    var id, nome, sobrenome string
    erro = rows.Scan(&id,&nome,&sobrenome)
    if erro != nil {
       panic(erro)
    }
    vt = vt + "<tr><th>" + id +"</th><th>" + nome +
        "</th><th>" + sobrenome + "</th></tr>"
  }
  vt = vt + "</table>"
  l := filepath.Join("templates", "layout.html")
  f := filepath.Join("templates", filepath.Clean(r.URL.Path))
```

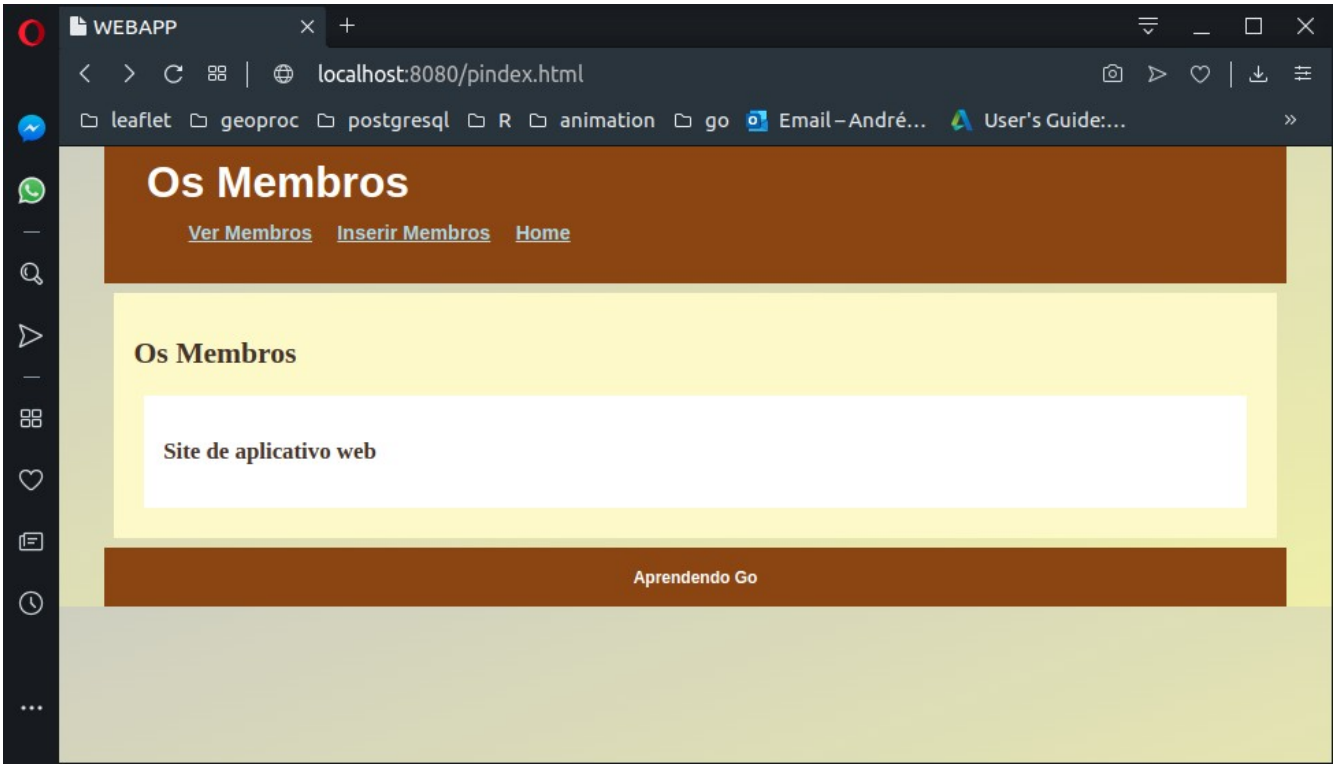
```
  tpl, _ := template.ParseFiles(l, f)
  tpl.Parse("{{define \"body\"}}"+vt+"{{end}}")
  tpl.ExecuteTemplate(w, "layout",nil)
}

func abre(w http.ResponseWriter, r *http.Request) {
  l := filepath.Join("templates", "layout.html")
  f := filepath.Join("templates", filepath.Clean(r.URL.Path))
  tpl, _ := template.ParseFiles(l, f)
  tpl.ExecuteTemplate(w, "layout", nil)
}

func main() {
  fs := http.FileServer(http.Dir("static"))
  http.Handle("/static/", http.StripPrefix("/static/", fs))
  http.HandleFunc("/pindex.html", abre)
  http.HandleFunc("/insere.html", abre)
  http.HandleFunc("/insere", insere)
  http.HandleFunc("/ver.html", ver)
  println("Listening...")
  http.ListenAndServe(":8080", nil)
}
```

Tudo junto Agora

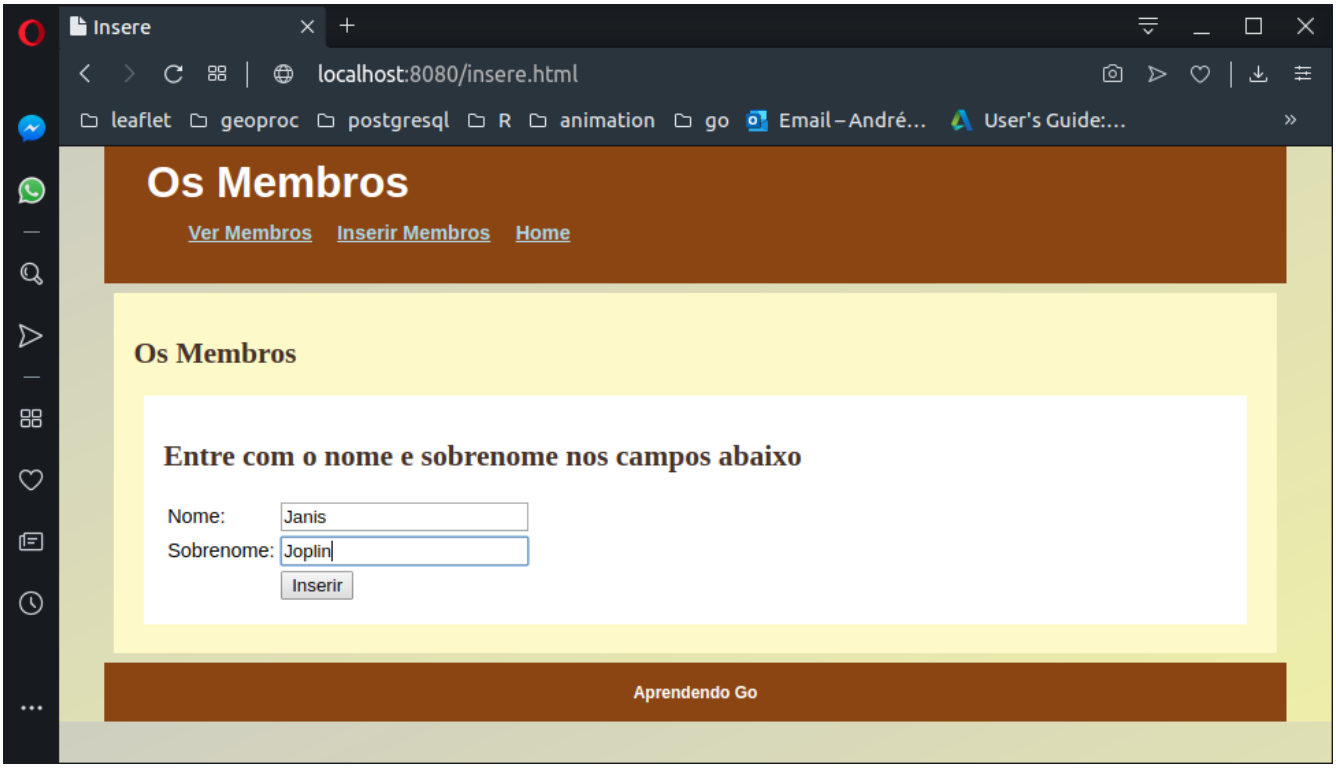
Coloque seu aplicativo para funcionar com:

```
go install webapp4
$GOPATH/bin/webapp4
```

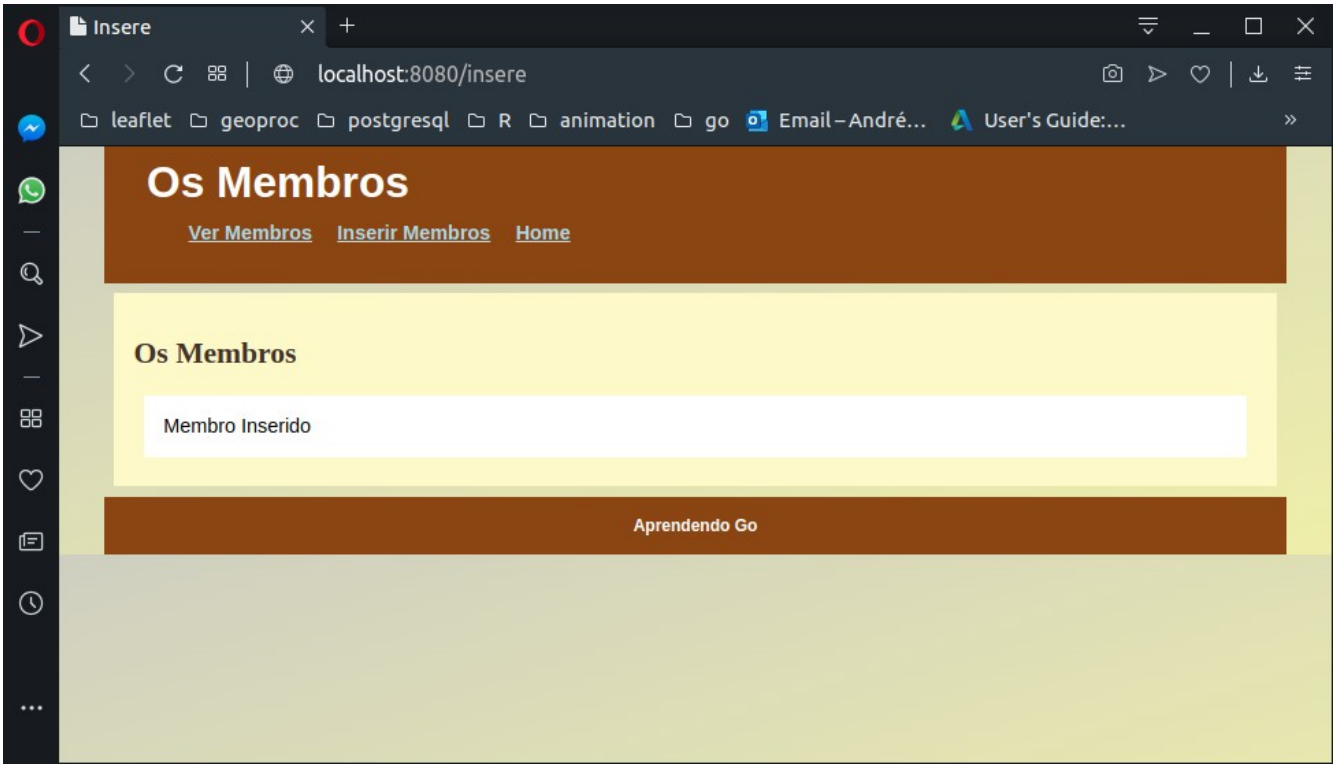
No navegador digite **`localhost:8080/pindex.html`** e a página abaixo deve aparecer:



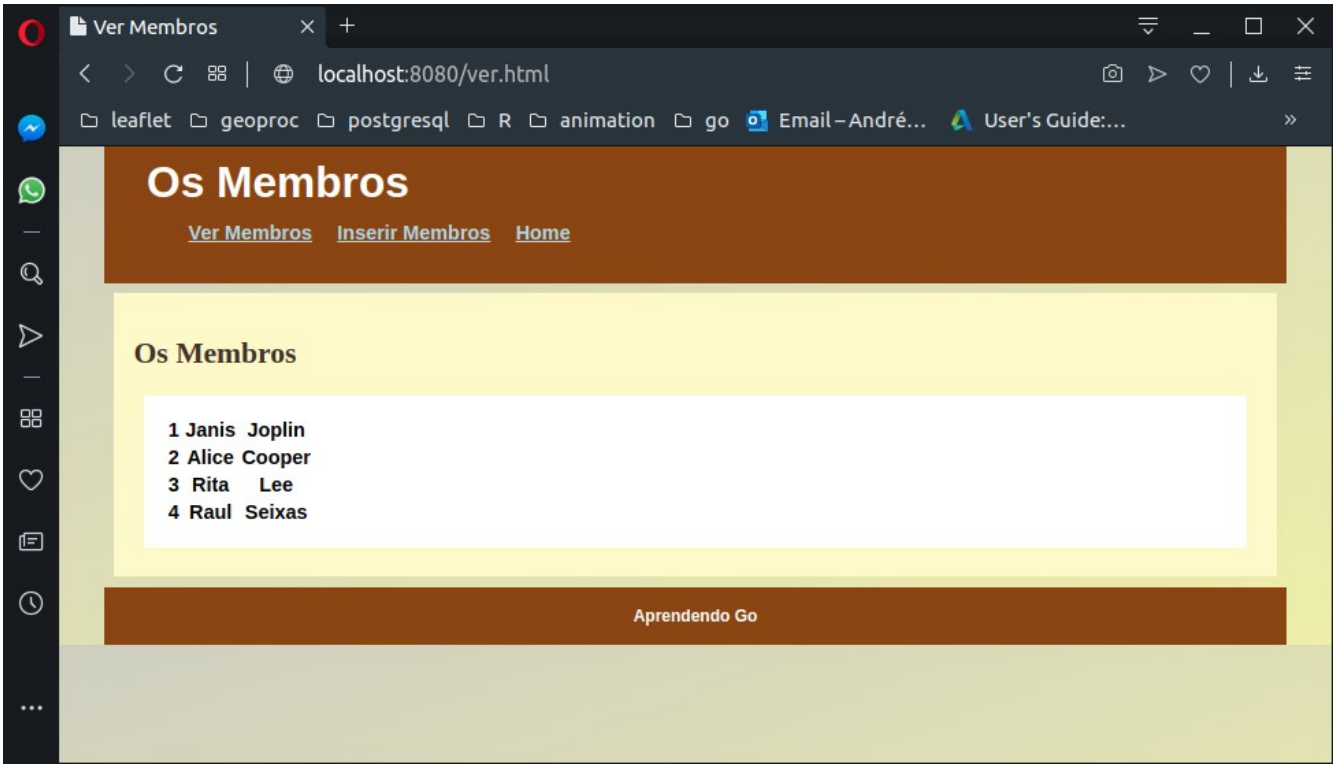
Ao clicar em ‘Inserir Membros’ a página abaixo aparecerá:



Vamos inserir um membro e ao clicar inserir a página abaixo aparecerá:



Clicando no menu ‘Ver Membros’, após inserir alguns membros a página abaixo será mostrada:



Como funciona

Na função `main` criamos os handlers de cada página e iniciamos o servidor. Quando usamos o handlers para as páginas pindex ou insere chamamos a mesma função (`abre`) e unimos o layout com pindex ou com insere de acordo. Pindex é a página inicial e insere a página de inserção.

Na página insere preenchemos os campos e ao clicar em inserir a função `insere` é acionada . A função `insere` chama a função `inserindo` que carrega os valores no banco de dados e em seguida carrega a página com a mensagem que o registro foi inserido.

Quando clicamos no link ver membros temos a função `ver` acionada, ela extrai os valores do banco de dados e apresenta o resultado na página.

Conclusão

Foram mostrados alguns conceitos básicos de como fazer um aplicativo web usando Go. Existem muitas outras técnicas e processos associados a aplicativos web com Go tais como sessões, cookies, sockets e etc. Seja curioso e pesquise sobre o assunto.

Parte 3 - Desenvolvimento de Aplicativos Go com Interface Gráfica

Nessa parte vamos ver como criar aplicativos com interface gráfica usando Go. Existem algumas plataformas para desenvolvimento de interfaces gráficas tais como Fyne, Qt, GTK, webview e outras. Neste livro vamos falar sobre fyne por sua portabilidade e simplicidade, também pelo fato de ter sido desenvolvida nativamente para uso com Go.

O objetivo é te dar uma base de como desenvolver aplicativos de interface gráfica com Go, Vamos lá!

Instalando Fyne

Para instalar fyne use o seguinte comando:

```
go get fyne.io/fyne
```

Primeiros passos

Vamos criar um aplicativo básico para testar fyne e introduzir os primeiros conceitos.

Crie o arquivo fyne1.go abaixo:

```
package main
import (
  "fyne.io/fyne"//carrega pacote fyne base
  "fyne.io/fyne/app"//carrega pacote fyne app
)
func main() {
  a := app.New()//cria novo aplicativo a
  w := a.NewWindow("GUI - Olá Pateta!")//adiciona janela w
  w.Resize(fyne.NewSize(360,250))// modifica o tamanho da janela w
  w.ShowAndRun()//mostra janela w e executa a
}
```

Construa e execute com:

```
go build fyne1.go
./fyne1
```

O seguinte ‘programa ‘ aparecerá:

Vamos detalhar o que acabamos de fazer.
a) Carregamos os pacotes fyne e app
`"fyne.io/fyne"`
`"fyne.io/fyne/app"`
b) Criamos um aplicativo fyne chamado “a” com
`a := app.New()`
c) Criamos uma janela “w” para o nosso aplicativo e passamos o título da janela como argumento
`w := a.NewWindow("GUI – Olá Pateta!")`
d) Redefinimos o tamanho da nossa janela com um tipo Size
`w.Resize(fyne.NewSize(360,250))`
e) Finalmente mostramos a janela “w” e executamos o aplicativo “a”
`w.ShowAndRun()`

O tipo App

A interface App é o nosso aplicativo propriamente dito e seus principais métodos são:
`NewWindow(title string) Window` – Cria uma nova janela para o seu aplicativo. Sem uma janela o aplicativo não é visível.
`Run()` - Executa o seu aplicativo. Geralmente Um aplicativo é executado no fim da função main() do seu programa pela função window `ShowAndRun()`.
`Quit()` - Fecha o aplicativo e todas as janelas existentes.
`SetIcon(Resource)` - Define o Icon do tipo Resource do seu aplicativo.

Associado a este tipo temos a função `CurrentApp()` que retorna o corrente aplicativo em execução pelo processo.

O tipo Window

Window define uma janela de interface gráfica de usuário (GUI). Dependo da plataforma um aplicativo pode ter um ou mais tipos window. Os principais métodos do tipo Window são:
`Title() string` - Retorna o título da janela
`SetTitle(string)` – Define o título da janela
`FullScreen() bool` - Retorna se a janela está em modo tela cheia ou não
`SetFullScreen(bool)` – Diz para a janela entrar ou sair do modo tela cheia
`Resize(Size)` – define o tamanho da janela com o tipo fyne.Size
`RequestFocus()` - Coloca o foco na janela
`FixedSize() bool` – Retorna se a janela tem tamanho fixo ou não
`SetFixedSize(bool)` – Diz para a janela fixar seu tamanho ou não
`CenterOnScreen()` - Centraliza a janela no monitor
`Padded() bool` - Retorna se a janela possui preenchimento ou não
`SetPadded(bool)` – Diz para usar preenchimento ou não na janela
`MainMenu() *MainMenu` – Retorna o menu principal da janela
`SetMainMenu(*MainMenu)` – Define o menu principal da janela
`SetOnClosed(func())` - Executa a função ao fechar a janela.
`Show()` - Mostra a janela
`Hide()` - Esconde a janela
`Close()` - Fecha a janela, se for a master fecha também o aplicativo
`ShowAndRun()` - Executa o aplicativo e mostra a janela

`Content() CanvasObject` – retorna o conteúdo da janela
`SetContent(CanvasObject)` – define o conteúdo da janela
`Canvas()` - Retorna o contexto do Canvas a ser renderizado na janela
`Clipboard() Clipboard` – Retorna o Clipboard do sistema

Testando as funções de App e Window

Para testar as funções vistas acima. Vamos antecipar o conceito de widget e fazer uma rápida apresentação ao widget botão.

```
widget.NewButton("Rótulo", func() {
                //ação
        })
```

Ao introduzirmos um widget botão com window.SetContent, conforme acima, criamos um botão com um rótulo e com uma função que é chamada quando clicamos no botão. Por enquanto é só isso que você precisa saber sobre esse widget.

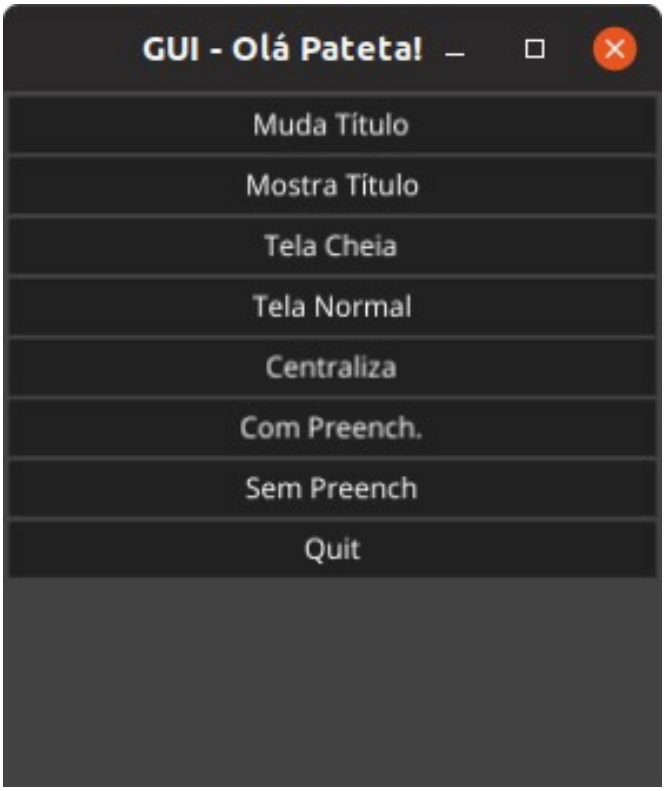
Vamos criar um programa que muda o comportamento/propriedade do aplicativo e da janela ao clicarmos determinados botões.

Crie o codigo fyne2.go abaixo:

```
package main
import (
     "fyne.io/fyne"
     "fyne.io/fyne/app"
     "fyne.io/fyne/widget"
)
func main() {
  a := app.New()
  w := a.NewWindow("GUI - Olá Pateta!")
  w.Resize(fyne.Size{360,380})
  w.SetContent(widget.NewVBox(
      widget.NewButton("Muda Título", func() {
               w.SetTitle("Estamos aí!")
          }),
      widget.NewButton("Mostra Título", func() {
                        println(w.Title())
                }),
      widget.NewButton("Tela Cheia", func() {
                        w.SetFullScreen(true)
                }),
      widget.NewButton("Tela Normal", func() {
                        w.SetFullScreen(false)
                }),
      widget.NewButton("Centraliza", func() {
                        w.CenterOnScreen()
                }),
      widget.NewButton("Com Preench.", func() {
                        w.SetPadded(true)
                }),
      widget.NewButton("Sem Preench", func() {
                        w.SetPadded(false)
                }),
```

```
      widget.NewButton("Quit", func() {
                        a.Quit()
                }),
  ))
  w. SetOnClosed(func(){
     println("Buuuuaaaaaaaa o aplicativo fechou!!!!")
  })
  w.ShowAndRun()
}
```

Compile e execute o programa e o GUI abaixo deverá aparecer, experimente com os botões e entenda o que cada função altera no tipo Window.



Use Estrutura

Use sempre uma estrutura para seu aplicativo, desta forma poderemos aliviar a função main() do seu programa. Pense sempre nesse conceito, pois ele que facilitará a criação e leitura de um programa mais robusto adiante.

Vamos transformar o programa acima em uma estrutura, crie o arquivo fyne3.go abaixo:
package main

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
```

```
  "fyne.io/fyne/widget"
)
type meusBotões struct{
  botões map[string]*widget.Button
  janela  fyne.Window
}
func (mb *meusBotões) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  mb.botões[texto] = botn
  return botn
}
func (mb *meusBotões) carregaUI(a fyne.App) {
  mb.janela = a.NewWindow("GUI - Olá Pateta!")
  mb.janela.Resize(fyne.Size{360,380})
  mb.janela.SetContent(widget.NewVBox(
      mb.adicionaBotão("Muda Título", func() {
          mb.janela.SetTitle("Estamos aí!")
     }),
      mb.adicionaBotão("Mostra Título", func() {
                println(mb.janela.Title())
        }),
      mb.adicionaBotão("Tela Cheia", func() {
                mb.janela.SetFullScreen(true)
        }),
      mb.adicionaBotão("Tela Normal", func() {
                mb.janela.SetFullScreen(false)
        }),
      mb.adicionaBotão("Centraliza", func() {
                mb.janela.CenterOnScreen()
        }),
      mb.adicionaBotão("Com Preench.", func() {
                mb.janela.SetPadded(true)
        }),
      mb.adicionaBotão("Sem Preench", func() {
                mb.janela.SetPadded(false)
        }),
      mb.adicionaBotão("Quit", func() {
                 a.Quit()
        }),
  ))
  mb.janela. SetOnClosed(func(){
   println("O aplicativo diz: Buuuuaaaaaaaa o aplicativo fechou!!!!")
  })
  mb.janela.ShowAndRun()
}
///////inicia struct e app--------------------------------
func iniciaMeusBotões() *meusBotões {
  mbotões := &meusBotões{}
  mbotões.botões = make(map[string]*widget.Button)
  return mbotões
}
func main() {
  mb := iniciaMeusBotões() //cria a struct do aplicativo
```

```
  mb.carregaUI(app.New()) //carrega/inicializa o aplicativo na struct
  println("Main() diz: Tchau Pateta!")
}
```

O programa faz exatamente o que o anterior fazia mas agora ele foi criado como uma struct. Seguiremos este formato durante toda essa parte.

Widgets

Agora sim vamos falar dos widgets que são os componentes (como os botões) que fazem parte do GUI que vão interagir com o usuário do programa.

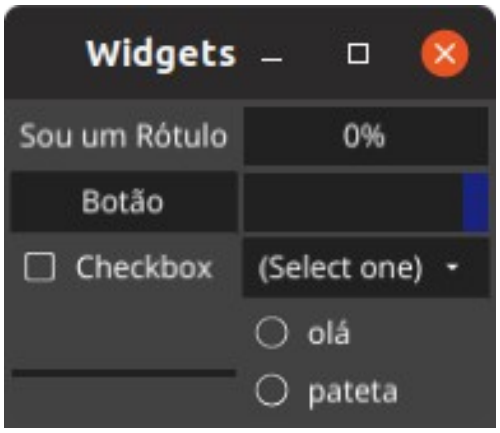
Primeiramente vamos apresentar os principais widgets que abordaremos neste livro em um único programa. Em seguida vamos falar mais detalhadamente sobre cada um deles.

Crie o programa abaixo com o nome de fyne4.go:

```
package main
import (
     "fyne.io/fyne"
     "fyne.io/fyne/app"
     "fyne.io/fyne/widget"
)
type meusWidgets struct{
  janela       fyne.Window
  horizontal   *widget.Box
  vertical1    *widget.Box
  vertical2    *widget.Box
  rótulo       *widget.Label
  botão        *widget.Button
  checa        *widget.Check
  texto        *widget.Entry
  progresso    *widget.ProgressBar
  infinito     *widget.ProgressBarInfinite
  seleciona    *widget.Select
  radio        *widget.Radio
}
func (mw *meusWidgets) carregaUI(a fyne.App) {
  mw.janela = a.NewWindow("Widgets")
  mw.rótulo = widget.NewLabel("Sou um Rótulo")
  mw.botão = widget.NewButton("Botão", func() {})
  mw.checa = widget.NewCheck("Checkbox",func(bool){})
  mw.texto = widget.NewEntry()
  mw.progresso = widget.NewProgressBar()
  mw.infinito = widget.NewProgressBarInfinite()
mw.seleciona=widget.NewSelect([]string{"olá","pateta"},func(string)
{})
  mw.radio = widget.NewRadio([]string{"olá","pateta"},func(string){})
  mw.vertical1 = widget.NewVBox(mw.rótulo,mw.botão,mw.checa,mw.texto)
   mw.vertical2=widget.NewVBox(mw.progresso,mw.infinito,mw.seleciona,m
w.radio)
  mw.horizontal = widget.NewHBox(mw.vertical1,mw.vertical2)
  mw.janela.SetContent(mw.horizontal)
```

```
  mw.janela. SetOnClosed(func(){
     println("Aplicativo finalizado!")
  })
  mw.janela.ShowAndRun()
}
//--------inicia struct e app--------------------------------
func iniciaMeusWidgets() *meusWidgets {
     mwid := &meusWidgets{}
     return mwid
}
func main() {
  mw := iniciaMeusWidgets() //cria a struct do aplicativo
  mw.carregaUI(app.New()) //carrega/inicializa o aplicativo na struct
  println("Função main() finalizada")
}
```

Ao executar teremos:



Além dos widgets mostrados nesse exemplo, também vamos falar sobre os widgets Group, toolbar e tabContainer. No **Apêndice 1** temos a lista destes tipos e seus métodos associados como referência. Vamos agora mostrar mais detalhes de cada widget.

Button

O primeiro widget que vamos analisar é o button (botão). Ele é um dos tipos que, com certeza, é usado em qualquer aplicativo gráfico. Muitos dos widget tem métodos em comum que são usados em todos os widgets e outros que são específicos de alguns widgets. Vamos cobrir todos eles mas não vamos repetir o uso, um método comum explicado em button não será descrito novamente em Box ou Entry.
Vamos criar o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type meusBotões struct{
  botões map[string]*widget.Button
  janela  fyne.Window
```

```
}
func (b *meusBotões) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meusBotões) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Botões")
  b.janela.Resize(fyne.Size{360,600})
  b.janela.SetContent(widget.NewVBox(
    b.adicionaBotão("A Cobaia", func() {
      println("Olá Pateta!")}),
    b.adicionaBotão("Esconde", func() {
      b.botões["A Cobaia"].Hide()}),
    b.adicionaBotão("Mostra", func() {
      b.botões["A Cobaia"].Show()}),
    b.adicionaBotão("Habilita", func() {
      b.botões["A Cobaia"].Enable()}),
    b.adicionaBotão("Desabilita", func() {
      b.botões["A Cobaia"].Disable()}),
    b.adicionaBotão("Está Visível", func() {
      if b.botões["A Cobaia"].Visible() {
        println("SIM")
      }else{
        println("NÃO")
      }}),
    b.adicionaBotão("Está desabilitado", func() {
      if b.botões["A Cobaia"].Disabled() {
        println("SIM")
      }else{
        println("NÃO")
      }}),
    b.adicionaBotão("Muda Texto", func() {
      b.botões["A Cobaia"].SetText("Agora sou Gopher")}),
    b.adicionaBotão("Tamanho", func() {
      fmt.Printf("Meu Tamanho é %d\n",b.botões["A Cobaia"].Size())}),
    b.adicionaBotão("Tamanho Mínimo", func() {
              fmt.Printf("Meu  Tamanho  minino  é  %d\n",b.botões["A
Cobaia"].MinSize())}),
    b.adicionaBotão("Diminui", func() {
      b.botões["A Cobaia"].Resize(fyne.Size{150,29})}),
    b.adicionaBotão("Restaura Tamanho", func() {
      b.botões["A Cobaia"].Resize(fyne.Size{351,29})}),
    b.adicionaBotão("Minha Posição", func() {
                fmt.Printf("Minha   Posição   é   %d\n",b.botões["A
Cobaia"].Position())}),
    b.adicionaBotão("Move", func() {
      b.botões["A Cobaia"].Move(fyne.Position{0,500})}),
    b.adicionaBotão("Move de volta", func() {
      b.botões["A Cobaia"].Move(fyne.Position{0,0})}),
  ))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
```

```
  })
  b.janela.ShowAndRun()
}
func iniciaMeusBotões() *meusBotões {
  mbotões := &meusBotões{}
  mbotões.botões = make(map[string]*widget.Button)
  return mbotões
}
func main() {
  b := iniciaMeusBotões()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Ao executar o programa teremos:

Criamos uma struct que é nosso aplicativo “MeusBotões” . Ela é composta por um map de botões e pela window do aplicativo.

```
type meusBotões struct{
  botões map[string]*widget.Button
  janela  fyne.Window
}
```

O method `adicionaBotão` faz exatamente o que o nome diz, ele toma dois argumentos. O primeiro é uma string que será o texto apresentado no botão e também o nome do índice no map `botões`. O segundo é a função a ser executada quando o botão é clicado.

```
func (b *meusBotões) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
```

A função `main` inicia o programa chamando a função que por sua vez inicia os itens da nossa struct (aplicativo). Em seguida ela chama o method `carregaUI` passando um tipo novo aplicativo que montará o nosso programa.

```
func iniciaMeusBotões() *meusBotões {
  mbotões := &meusBotões{}
  mbotões.botões = make(map[string]*widget.Button)
  return mbotões
}
func main() {
  b := iniciaMeusBotões()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

No method criamos nosso window e definimos seu tamanho.

```
b.janela = a.NewWindow("GUI - Widget Botões")
b.janela.Resize(fyne.Size{360,600})
```

Em seguida criamos um widget do tipo Box usando a função `widget.NewVBox` (que veremos em seguida) e nele adicionamos vários botões que mostrarão como funcionam os methods do widget botões. O primeiro a ser criado foi o nosso botão “A Cobaia” que ao ser clicado mostra a mensagem “Olá Pateta”. Nele aplicaremos os methods através dos botões criados em seguida.

```
b.adicionaBotão("A Cobaia", func() {
println("Olá Pateta!")
})
```

Este botão aciona o método Hide() que esconde o botão que chama ele (A cobaia).

```
b.adicionaBotão("Esconde", func() {
b.botões["A Cobaia"].Hide()
})
```

Este botão aciona o método Show() que mostra o botão que chama o método.

```
b.adicionaBotão("Mostra", func() {
b.botões["A Cobaia"].Show()
})
```

Este botão chama o método que habilita o botão que o chama.

```
b.adicionaBotão("Habilita", func() {
b.botões["A Cobaia"].Enable()
})
```

Este botão chama o método que habilita o botão “A Cobaia”

```
b.adicionaBotão("Desabilita", func() {
b.botões["A Cobaia"].Disable()
})
```

Agora, com esse método temos um valor de retorno do tipo bool, então vamos apresentar a resposta conforme o resultado. Se o botão “A Cobaia” estiver visível a o método Visible() responderá com esse código abaixo SIM, caso contrário será NÃO.

```
b.adicionaBotão("Está Visível", func() {
   if b.botões["A Cobaia"].Visible() {
     println("SIM")
   }else{
     println("NÃO")
   }
})
```

Da mesma forma checamos agora se ele está habilitado ou não com o método `Disabled()`.

```
b.adicionaBotão("Está desabilitado", func() {
   if b.botões["A Cobaia"].Disabled() {
     println("SIM")
   }else{
     println("NÃO")
  }
})
```

Método SetText(string) altera o texto do botão (note que isso não altera o valor do índice do map)

```
b.adicionaBotão("Muda Texto", func() {
b.botões["A Cobaia"].SetText("Agora sou Gopher")
})
```

O método size() retorna o tamanho do widget que o chama

```
b.adicionaBotão("Tamanho", func() {
fmt.Printf("Meu Tamanho é %d\n",b.botões["A Cobaia"].Size())
})
```

O método MinSize() retorna o tamanho mínimo possível do botão, baseado no texto que ele apresenta.

```
b.adicionaBotão("Tamanho Mínimo", func() {
fmt.Printf("Meu Tamanho minino é %d\n",b.botões["A
Cobaia"].MinSize())
})
```

O método Resize(Size) modifica o tamanho do widget que o chama.

```
b.adicionaBotão("Diminui", func() {
b.botões["A Cobaia"].Resize(fyne.Size{150,29})
})
```

Aqui chamamos novamente Resize para voltar o botão ao seu tamanho original.

```
b.adicionaBotão("Restaura Tamanho", func() {
b.botões["A Cobaia"].Resize(fyne.Size{351,29})
})
```

O método Position() retorna a posição do widget que o chama.

```
b.adicionaBotão("Minha Posição", func() {
fmt.Printf("Minha Posição é %d\n",b.botões["A Cobaia"].Position())
})
```

O método Move(Position), move o widget para a posição especificada por Position

```
b.adicionaBotão("Move", func() {
b.botões["A Cobaia"].Move(fyne.Position{0,500})
})
```

Chamamos Move(Position) novamente para mover o botão para sua posição original

```
b.adicionaBotão("Move de volta", func() {
b.botões["A Cobaia"].Move(fyne.Position{0,0})
})
```

Neste exemplo mostramos como é simples usar widgets e como podemos mudar suas características usando os métodos deste widget. Muitos widgets tem métodos em comum e não voltaremos a falar deles no próximo widget. A lista de métodos de cada widget está presente no Apêndice 1.

Box e Label

Vamos agora falar sobre o widget Box e Label. Usamos Box no exemplo acima mas agora vamos falar sobre ele em mais detalhes. Do widget Label falaremos do método String() que falta ser explicado.
Crie o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type minhaBox struct{
  vertical     *widget.Box
  horizontal   *widget.Box
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func (b *minhaBox) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *minhaBox) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Box")
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Olá Pateta")
  b.vertical = widget.NewVBox()
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.adicionaBotão("Gopher",func(){
      println("Estou no VBox")}))
```

```
  b.vertical.Append(b.adicionaBotão("Gopher2",func(){
      println("Eu estou no VBox")}))
  b.horizontal = widget.NewHBox()
  b.vertical.Append(b.horizontal)
  b.horizontal.Append(b.rótulo)
  b.horizontal.Prepend(b.adicionaBotão("Pateta",func(){
       fmt.Printf("Olá, Estou no VBox, mas em um HBox\n O rótulo %s
também está comigo\n",b.rótulo.String())}))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMinhaBox() *minhaBox {
  mbox := &minhaBox{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMinhaBox()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Executando o programa teremos:

A struct do aplicativo é composta por, além da window, um map de botões, um rótulo e dois box.

```
type minhaBox struct{
  vertical     *widget.Box
  horizontal   *widget.Box
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
```

Montamos o aplicativo construindo e dimensionando a window janela, criando o widget rótulo do tipo Label, e depois criamos e adicionamos o box vertical na janela.

```
b.janela = a.NewWindow("GUI - Widget Box")
b.janela.Resize(fyne.Size{360,360})
b.rótulo = widget.NewLabel("Olá Pateta")
b.vertical = widget.NewVBox()
b.janela.SetContent(b.vertical)
```

No box vertical adicionamos dois botões e um box horizontal usando os métodos Prepend() que adiciona no início do box ou append(), que adiciona ao final do box.

```
b.vertical.Prepend(b.adicionaBotão("Gopher",func(){
    println("Estou no VBox")}))
b.vertical.Append(b.adicionaBotão("Gopher2",func(){
    println("Eu estou no VBox")}))
b.horizontal = widget.NewHBox()
b.vertical.Append(b.horizontal)
```

Por último, adicionamos o rótulo e o botão na box horizontal. Na função do botão chamamos o método String() que retorna o texto do rótulo no formato string.

```
b.horizontal.Append(b.rótulo)
b.horizontal.Prepend(b.adicionaBotão("Pateta",func(){
 fmt.Printf("Olá, Estou no VBox, mas em um HBox\n O rótulo %s também
está comigo\n",b.rótulo.String())}))
```

Entry

Vamos agora ver o widget Entry, que são os campos de entrada de informações alfanuméricas de seu programa.

Crie o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type minhaEntry struct{
  vertical     *widget.Box
  entrada      *widget.Entry
  entMulti     *widget.Entry
  entSenha     *widget.Entry
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func (b *minhaEntry) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *minhaEntry) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Entry")
```

```
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Entre Algo")
  b.vertical = widget.NewVBox()
  b.entrada = widget.NewEntry()
  b.entMulti= widget.NewMultiLineEntry()
  b.entSenha = widget.NewPasswordEntry()
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.rótulo)
  b.vertical.Append(b.entrada)
  b.vertical.Append(b.entMulti)
  b.vertical.Append(b.entSenha)
  b.entrada.SetText("Olá Pateta")
  b.entrada.SetReadOnly(true)
  b.entMulti.SetPlaceHolder("Este é\n Só para ficar no lugar")
  b.entSenha.SetText("segredo")
  b.vertical.Append( b.adicionaBotão("Mostra",func(){
   fmt.Printf("Entrada: %s\nMultilinha: %s\nSenha: %s\n",
   b.entrada.Text,b.entMulti.Text, b.entSenha.Text)}))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMinhaEntry() *minhaEntry {
  mbox := &minhaEntry{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMinhaEntry()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado será:

Introduzimos no nosso aplicativo três tipos distintos de Entry, um para Entrada normal, outra para entrada com mais de uma linha e o último para entrada de senhas.

```
b.entrada = widget.NewEntry()
b.entMulti= widget.NewMultiLineEntry()
b.entSenha = widget.NewPasswordEntry()
```

O método SetText() adiciona um texto no Entry e na variável Text, SetReadyOnly() impede a edição do Entry, SetPlaceHolder(string) adiciona uma string no campo mas não faz parte do valor dele.

```
b.entrada.SetText("Olá Pateta")
b.entrada.SetReadOnly(true)
b.entMulti.SetPlaceHolder("Este é\n Só para ficar no lugar")
b.entSenha.SetText("segredo")
```

Quando chamamos Entry.Text (usando o botão no nosso caso) o conteúdo da variável Text é mostrado.

```
b.vertical.Append( b.adicionaBotão("Mostra",func(){
fmt.Printf("Entrada: %s\nMultilinha: %s\nSenha: %s\n",
b.entrada.Text,b.entMulti.Text, b.entSenha.Text)}))
```

Select

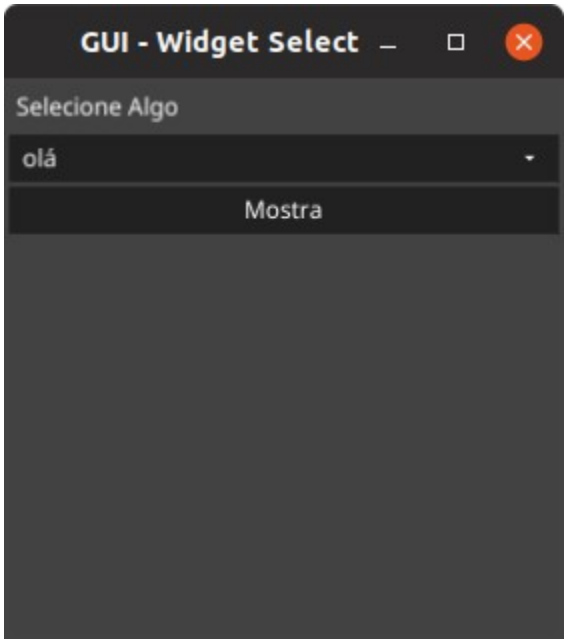
Select é um widget usado para escolha de uma opção entre várias onde o usuário do programa aciona o widget e uma lista é apresentada a ele.

Crie o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type meuSelect struct{
  vertical     *widget.Box
  seleciona    *widget.Select
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func (b *meuSelect) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuSelect) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Select")
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Selecione Algo")
  b.vertical = widget.NewVBox()
  opções := make([]string,2)
  opções[0]="olá"
  opções[1]="pateta"
  b.seleciona = widget.NewSelect(opções,func(a string){
```

```
    fmt.Println(a)
    })
  b.seleciona.SetSelected(opções[0])
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.rótulo)
  b.vertical.Append(b.seleciona)


  b.vertical.Append( b.adicionaBotão("Mostra",func(){
   fmt.Printf("Foi selecionado: %s\n",
   b.seleciona.Selected)
   }))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuSelect() *meuSelect {
  mbox := &meuSelect{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMeuSelect()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O programa abaixo será criado:



Radio

Uma outra forma de escolhermos opções de valores é o widget Radio. Onde várias opções são mostradas ao mesmo tempo para você fazer a escolha marcando a opção.
Crie o arquivo abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type meuRadio struct{
  vertical     *widget.Box
  radiobtn     *widget.Radio
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func (b *meuRadio) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuRadio) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Radio")
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Selecione Algo")
  b.vertical = widget.NewVBox()
  opções := make([]string,2)
  opções[0]="olá"
  opções[1]="pateta"
  b.radiobtn = widget.NewRadio(opções,func(a string){
    fmt.Println(a)
    })
  b.radiobtn.Append("radio")
  b.radiobtn.SetSelected(opções[0])
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.rótulo)
  b.vertical.Append(b.radiobtn)


  b.vertical.Append( b.adicionaBotão("Mostra",func(){
   fmt.Printf("Foi selecionado: %s\n",
   b.radiobtn.Selected)
   }))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuRadio() *meuRadio {
```

```
  mbox := &meuRadio{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMeuRadio()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Ao executar teremos:

Check

Usamos o widget Check quando precisamos de escolher uma ou mais opções em determinada tarefa.

Crie o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fmt"
)
type meuCheck struct{
  vertical     *widget.Box
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  checa         map[string]*widget.Check
```

```
  janela       fyne.Window
}
func (b *meuCheck) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuCheck) adicionaCheca(texto string, ação func(bool))
*widget.Check {
  chk := widget.NewCheck(texto, ação)
  b.checa[texto] = chk
  return chk
}
func (b *meuCheck) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Check")
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Escolha opções")
  b.vertical = widget.NewVBox()
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.rótulo)
  b.vertical.Append( b.adicionaCheca("Olá",func(bool){}))
  b.vertical.Append( b.adicionaCheca("Pateta",func(bool){}))
  b.vertical.Append( b.adicionaCheca("Gopher",func(bool){}))
  b.checa["Gopher"].SetChecked(true)
  b.vertical.Append( b.adicionaBotão("Selecionados",func(){
     for i := range b.checa {
      if b.checa[i].Checked {
       fmt.Printf("Checado: %s \n", b.checa[i].Text)
      }
     }
   }))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuCheck() *meuCheck {
  mbox := &meuCheck{}
  mbox.botões = make(map[string]*widget.Button)
  mbox.checa = make(map[string]*widget.Check)
  return mbox
}
func main() {
  b := iniciaMeuCheck()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Ao executar veremos o seguinte:

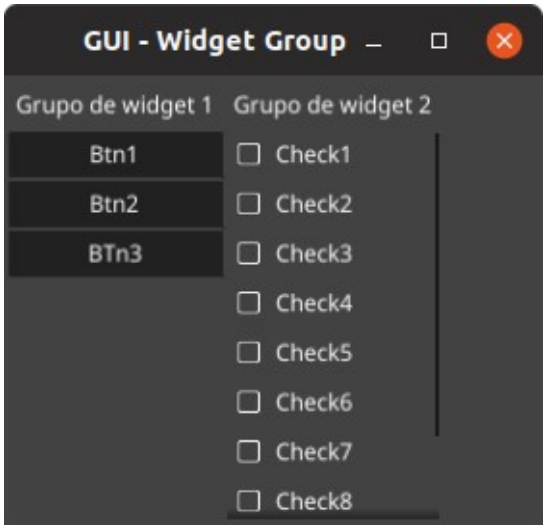
Group

Podemos agrupar um conjunto de widgets usando o widget Group, ele pode ser fixo ou deslizável. Crie o programa abaixo para ver como o widget Group opera:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
)
type meuGroup struct{
  horizontal   *widget.Box
  fixo         *widget.Group
  rolável *widget.Group
  botões  map[string]*widget.Button
  checa         map[string]*widget.Check
  janela       fyne.Window
}
func (b *meuGroup) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuGroup) adicionaCheca(texto string, ação func(bool))
*widget.Check {
  chk := widget.NewCheck(texto, ação)
  b.checa[texto] = chk
  return chk
}
func (b *meuGroup) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Group")
```

```
  b.janela.Resize(fyne.Size{360,300})
  b.horizontal = widget.NewHBox()
  b.janela.SetContent(b.horizontal)
  b.fixo = widget.NewGroup("Grupo de widget 1",
  b.adicionaBotão("Btn1",func(){}),
  b.adicionaBotão("Btn2",func(){}),
  b.adicionaBotão("BTn3",func(){}))
  b.rolável = widget.NewGroupWithScroller("Grupo de widget 2",
  b.adicionaCheca("Check1",func(bool){}),
  b.adicionaCheca("Check2",func(bool){}),
  b.adicionaCheca("Check3",func(bool){}),
  b.adicionaCheca("Check4",func(bool){}),
  b.adicionaCheca("Check5",func(bool){}),
  b.adicionaCheca("Check6",func(bool){}),
  b.adicionaCheca("Check7",func(bool){}),
  b.adicionaCheca("Check8",func(bool){}),
  b.adicionaCheca("Check9",func(bool){}),
  b.adicionaCheca("Check10",func(bool){}))
  b.horizontal.Append(b.fixo)
  b.horizontal.Append(b.rolável)
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuGroup() *meuGroup {
  mbox := &meuGroup{}
  mbox.botões = make(map[string]*widget.Button)
  mbox.checa = make(map[string]*widget.Check)
  return mbox
}
func main() {
  b := iniciaMeuGroup()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado do programa será:



ProgressBar

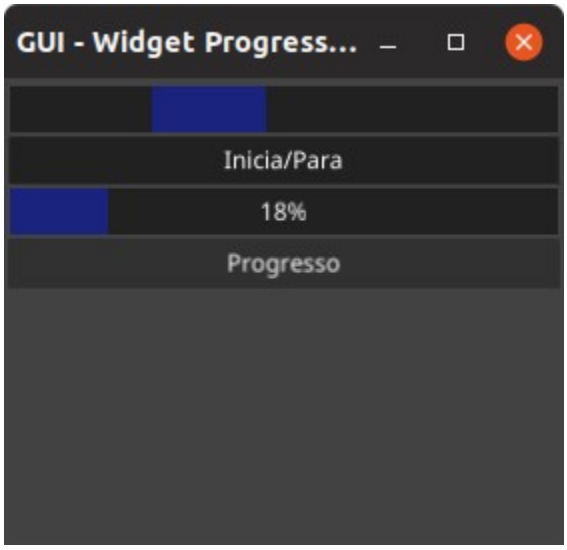
O widget ProgressBar cria uma barra de progresso que serve para informar ao usuário sobre o progresso de uma tarefa. Esta barra pode ser infinita que não fornece informação do quanto já foi feito, ou com indicação de quanto já foi executado da tarefa.

Vamos ver no exemplo abaixo como este widget funciona:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "time"
)
type meuProgresso struct{
  vertical     *widget.Box
  progresso    *widget.ProgressBar
  infinito     *widget.ProgressBarInfinite
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func (b *meuProgresso) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuProgresso) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget ProgressBar")
  b.janela.Resize(fyne.Size{360,300})
  b.vertical = widget.NewVBox()
  b.progresso = widget.NewProgressBar()
  b.progresso.Min=0
  b.progresso.Max=100
  b.infinito = widget.NewProgressBarInfinite()
  b.janela.SetContent(b.vertical)
  b.vertical.Append(b.infinito)
  b.vertical.Append( b.adicionaBotão("Inicia/Para",func(){
   if b.infinito.Running(){
     b.infinito.Stop()
   }else{
     b.infinito.Start()
   }
  }))
  b.vertical.Append(b.progresso)
  b.vertical.Append( b.adicionaBotão("Progresso",func(){
   p:=1.0
   for a:=0;a<=100;a++ {
    time.Sleep(time.Second/3)
    p++;
    b.progresso.SetValue(p)
   }
   b.progresso.SetValue(0)
```

```
  }))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuProgresso() *meuProgresso {
  mbox := &meuProgresso{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMeuProgresso()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Ao executar o programa teremos:



TabContainer

O widget TabContainer separa a área de trabalho do seu aplicativo em abas, é bastante útil para organizar alguns tipos de aplicativos.

Demonstraremos no programa abaixo o widget TabContainer

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
)
type minhaAba struct{
  aba          *widget.TabContainer
  item         map[string]*widget.TabItem
  botões  map[string]*widget.Button
  janela       fyne.Window
}
```

```
func (b *minhaAba) adicionaBotão(texto string, ação func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *minhaAba) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget TabContainer")
  b.janela.Resize(fyne.Size{560,400})
  b.item["aba1"] = widget.NewTabItem("Aba 1",widget.NewLabel("Estamos
na aba 1"))
  b.item["aba2"] = widget.NewTabItem("Aba 2",widget.NewLabel("E agora
nós \nestamos na Aba 2"))
  b.aba = widget.NewTabContainer(b.item["aba1"],b.item["aba2"])
  b.janela.SetContent(b.aba)
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMinhaAba() *minhaAba {
  mbox := &minhaAba{}
  mbox.item = make(map[string]*widget.TabItem)
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMinhaAba()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado do programa é apresentado abaixo:

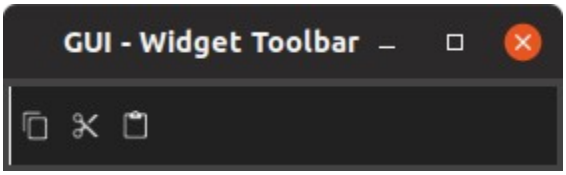
Toolbar

O widget Toolbar consiste de uma barra de ferramentas que na verdade são controles que chamam outras atividades, tarefas ou diálogos. São bastante úteis e tornam a chamada de tarefas simples e diretas assim como menus que veremos em seguida.

Vamos ver no exemplo abaixo como criar um Toolbar para um programa:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fyne.io/fyne/theme"
)
type meuToolbar struct{
  vertical     *widget.Box
  ferramenta   *widget.Toolbar
  itemTool     map[string]widget.ToolbarItem
  janela       fyne.Window
}
func (b *meuToolbar) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Widget Toolbar")
  b.janela.Resize(fyne.Size{360,60})
  b.ferramenta=widget.NewToolbar(widget.NewToolbarSeparator())
  b.itemTool["copia"]=widget.NewToolbarAction(theme.ContentCopyIcon()
,func(){println("copia")})
  b.itemTool["corta"]=widget.NewToolbarAction(theme.ContentCutIcon(),
func(){println("corta")})
  b.itemTool["cola"]=widget.NewToolbarAction(theme.ContentPasteIcon()
,func(){println("cola")})
  b.ferramenta.Append(b.itemTool["copia"])
  b.ferramenta.Append(b.itemTool["corta"])
  b.ferramenta.Append(b.itemTool["cola"])
  b.janela.SetContent(b.ferramenta)
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuToolbar() *meuToolbar {
  mbox := &meuToolbar{}
  mbox.itemTool = make(map[string]widget.ToolbarItem)
  return mbox
}
func main() {
  b := iniciaMeuToolbar()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado é:



Menus

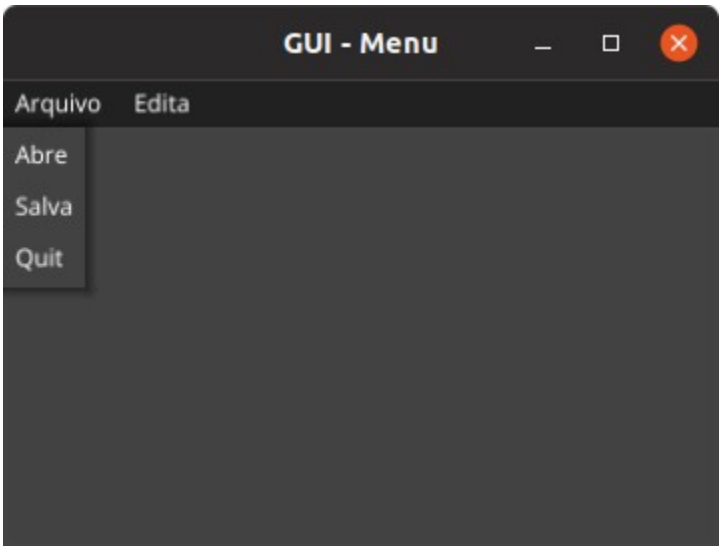
Após vermos os widgets acima, vamos falar de Menus. Menus estão presentes em quase todos aplicativos de desktop. São bastante eficientes na navegação de todas as tarefas de um programa.

Vamos ver um exemplo de uso de menus abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
)
type meuMenu struct{
  principal    *fyne.MainMenu
  oMenu        map[string]*fyne.Menu
  itemMenu     map[string]*fyne.MenuItem
  janela       fyne.Window
}
func   (b   *meuMenu)   adicionaItem(texto   string,   ação   func())
*fyne.MenuItem {
  mni := fyne.NewMenuItem(texto, ação)
  b.itemMenu[texto] = mni
  return mni
}
func (b *meuMenu) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Menu")
  b.janela.Resize(fyne.Size{460,300})
  b.adicionaItem("Abre", func() {
                        println("Menu item Abre")
                })
  b.adicionaItem("Salva", func() {
                        println("Menu item Salva")
                })
   b.oMenu["Arquivo"]=fyne.NewMenu("Arquivo",b.itemMenu["Abre"],b.item
Menu["Salva"])

  b.adicionaItem("Procura", func() {
                        println("Menu item Procura")
                })
  b.adicionaItem("Substitui", func() {
                        println("Menu item Substitui")
                })
   b.oMenu["Edita"]=fyne.NewMenu("Edita",b.itemMenu["Procura"],b.itemM
enu["Substitui"])

  b.principal = fyne.NewMainMenu(b.oMenu["Arquivo"],b.oMenu["Edita"])
```

```
  b.janela.SetMainMenu(b.principal)
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuMenu() *meuMenu {
  mbox := &meuMenu{}
  mbox.oMenu = make(map[string]*fyne.Menu)
  mbox.itemMenu = make(map[string]*fyne.MenuItem)
  return mbox
}
func main() {
  b := iniciaMeuMenu()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado é:



Dialog

Uma forma efetiva de interação entre o aplicativo e o usuário é o dialog (diálogo). Nele obtemos uma resposta do tipo Sim/Não ou Ok ou simplesmente uma forma de informar um passo feito ou um erro que ocorreu.

Vamos ver no nosso exemplo as formas de diálogo mencionadas acima. Cada botão acionará um dialog distinto com características também distintas.

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fyne.io/fyne/dialog"
)
```

```
type MeuErro struct{}
func (err *MeuErro) Error() string{
   return "Algo errado aconteceu!"
}
type meuDialogo struct{
  vertical     *widget.Box
  rótulo  *widget.Label
  botões  map[string]*widget.Button
  janela       fyne.Window
}
func  (b  *meuDialogo)  adicionaBotão(texto  string,  ação  func())
*widget.Button {
  botn := widget.NewButton(texto, ação)
  b.botões[texto] = botn
  return botn
}
func (b *meuDialogo) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Diálogos")
  b.janela.Resize(fyne.Size{360,360})
  b.rótulo = widget.NewLabel("Acione um Diálogo")
  b.vertical = widget.NewVBox()
  b.janela.SetContent(b.vertical)
  b.vertical.Prepend(b.rótulo)
  b.vertical.Append( b.adicionaBotão("Confirma",func(){
    dialog.ShowConfirm("Confirma", "Confirma que você gosta de Go?",
func(resp bool){
     if resp {
       println("Resposta Sim")
     } else {
       println("Resposta Não")
     }
    }, b.janela)
   }))
  b.vertical.Append( b.adicionaBotão("Customizado",func(){
                        dialog.ShowCustom("Customizado",       "Deixa
assim?",widget.NewEntry() , b.janela)
   }))
   b.vertical.Append( b.adicionaBotão("Informe",func(){
        dialog.ShowInformation("Informação",   "Sempre   se   mantenha
informado!", b.janela)
   }))
  meuErr :=&MeuErro{}
  b.vertical.Append( b.adicionaBotão("Erro",func(){
   dialog.ShowError(meuErr, b.janela)
   }))
  b.vertical.Append( b.adicionaBotão("Confirma",func(){
   dialog.ShowCustomConfirm("Confirma", "Confirmo","Não Confirmo",
    widget.NewEntry(),func(resp bool){
     if resp {
       println("Resposta Confirmado")
     } else {
       println("Resposta Não Confirmo")
     }
    }, b.janela)
```

```
    }))
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuDialogo() *meuDialogo {
  mbox := &meuDialogo{}
  mbox.botões = make(map[string]*widget.Button)
  return mbox
}
func main() {
  b := iniciaMeuDialogo()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

Ao executarmos veremos:

Containers e Layout

Containers e layouts são usados para organizarmos os widgets dentro de um aplicativo.
Basicamente a janela possui um container que por sua vez contém um layout. Criaremos um Container e vamos adicionar um layout do tipo grid no container.

Crie o programa abaixo:

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fyne.io/fyne/layout"
)
type meuConte struct{
  conteiner     *fyne.Container
  laylay  fyne.Layout
  janela       fyne.Window
}
func (b *meuConte) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("GUI - Container")
  b.janela.Resize(fyne.Size{720,480})
  b.laylay =  layout.NewGridLayout(3)
  b.conteiner= fyne.NewContainerWithLayout(b.laylay,
     widget.NewLabel("o"),widget.NewLabel("pato"),
     widget.NewLabel("pateta"),widget.NewLabel("pintou"),
     widget.NewLabel("o"),widget.NewLabel("caneco"),
     widget.NewLabel("surrou"),widget.NewLabel("a"),
     widget.NewLabel("galinha"),widget.NewLabel("bateu"),
     widget.NewLabel("no"),widget.NewLabel("marreco"),
     widget.NewLabel("pulou..."))
  b.janela.SetContent(b.conteiner)
  b.janela. SetOnClosed(func(){
     println("O aplicativo fechou!")
  })
  b.janela.ShowAndRun()
}
func iniciaMeuConte() *meuConte {
  mbox := &meuConte{}
  return mbox
}
func main() {
  b := iniciaMeuConte()
  b.carregaUI(app.New())
  println("Função main() diz: Tchau!")
}
```

O resultado será:

Exemplo: Gerenciador de Banco de Dados, estilo GUI

Vamos colocar em prática o que aprendemos sobre GUI usando Go criando um programa que conecta a um banco de dados e apresenta cada tabela do banco de dados em abas distintas.

1) Crie o banco de dados nos postgreSQL com:

```
createdb progma --encoding=utf-8
```

2)Crie duas tabelas no banco de dados com:

```
psql
progma=# create table programador (id serial primary key, nome varchar(50), time varchar(20), salario real);
progma=# create table produtos (produto int, tarefa varchar(50), id_programador int, concluido boolean);
```

3)Insira os dados nas tabelas com:

```
progma=# insert into programador Values(1,'Ritchie','OS',200);
progma=# insert into programador Values(2,'Torvalds','OS',130);
progma=# insert into programador Values(3,'Knuth','Algorítimos',100);
progma=# insert into programador Values(4,'Thompson','C',140);
progma=# insert into programador Values(5,'Stroustrup','C++',130);
progma=# insert into programador Values(6,'Berners-Lee','HTML',70);
progma=# insert into programador Values(7,'Kernighan','C',170);
progma=# insert into programador Values(8,'Gosling','Java',120);
progma=# insert into programador Values(9,'Carmack','Games',220);
progma=# insert into programador Values(10,'Eich','javaScript',120);
```

```
progma=# insert into programador Values(11,'Matsumoto','Ruby',130);
progma=# insert into programador Values(12,'Lerdorf','PHP',230);
progma=# insert into programador Values(13,'Stonebraker',
'PostgreSQL' ,250);
progma=# insert into programador Values(14,'Pike','Go',220);
progma=# insert into produtos Values(1,'algoritimo de escolha',3,
true);
progma=# insert into produtos Values(1,'revisar licença',2,false);
progma=# insert into produtos Values(2,'revisar licença',2,false);
progma=# insert into produtos Values(3,'revisar licença',2,false);
progma=# insert into produtos Values(2,'algoritimo de
produção',3,true);
progma=#  insert  into  produtos  Values(3,'algoritimo  estatístico',3,
false);
```

4) Crie o programa abaixo

```
package main
import (
  "fyne.io/fyne"
  "fyne.io/fyne/app"
  "fyne.io/fyne/widget"
  "fyne.io/fyne/layout"
  "database/sql"
  _ "github.com/lib/pq"//driver para postgresql
  "strconv"
)
const sqlInfo = "host=localhost port=5432 user=andre "+
              "password=segredo dbname=progma sslmode=disable"
type Progma struct{
  aba          *widget.TabContainer
  item         map[string]*widget.TabItem
  janela       fyne.Window
  db      *sql.DB  //createdb progma --encoding=utf-8
  principal    *fyne.MainMenu
  oMenu        map[string]*fyne.Menu
  itemMenu     map[string]*fyne.MenuItem
  conteiner     *fyne.Container
  laylay  fyne.Layout
  conteiner2     *fyne.Container
  laylay2 fyne.Layout
}
func   (b   *Progma)   adicionaItem(texto   string,   ação   func())
*fyne.MenuItem {
  mni := fyne.NewMenuItem(texto, ação)
  b.itemMenu[texto] = mni
  return mni
}
func (b *Progma) conecta(){
  b.db,_ = sql.Open("postgres", sqlInfo)
  b.carrega()
}
func (b *Progma) carrega(){
  rows, erro := b.db.Query("SELECT * FROM programador")
  if erro != nil {
    panic(erro)
```

```
  }
  defer rows.Close()
  for rows.Next() {
    var nome, time string
    var id, salario int
    erro = rows.Scan(&id,&nome,&time,&salario)
    if erro != nil {
       panic(erro)
    }
    b.conteiner.AddObject(widget.NewLabel(strconv.Itoa(id)))
    b.conteiner.AddObject(widget.NewLabel(nome))
    b.conteiner.AddObject(widget.NewLabel(time))
    b.conteiner.AddObject(widget.NewLabel(strconv.Itoa(salario)))
  }
  rows, erro = b.db.Query("SELECT * FROM produtos")
  if erro != nil {
    panic(erro)
  }
  defer rows.Close()
  for rows.Next() {
    var tarefa, conc string
    var produto,idprog int
    erro = rows.Scan(&produto,&tarefa,&idprog,&conc)
    if erro != nil {
       panic(erro)
    }
    b.conteiner2.AddObject(widget.NewLabel(strconv.Itoa(produto)))
    b.conteiner2.AddObject(widget.NewLabel(tarefa))
    b.conteiner2.AddObject(widget.NewLabel(strconv.Itoa(idprog)))
    b.conteiner2.AddObject(widget.NewLabel(conc))
  }
}
func (b *Progma) desconecta(){
  b.db.Close()
  println("Desconectado")
}
func (b *Progma) carregaUI(a fyne.App) {
  b.janela = a.NewWindow("PROGMA")
  b.janela.Resize(fyne.Size{960,540})
  b.adicionaItem("Conecta", func(){b.conecta()})
  b.adicionaItem("Desconecta",func(){b.desconecta()})
                b.oMenu["BancoDeDados"]=fyne.NewMenu("Banco        de
Dados",b.itemMenu["Conecta"],b.itemMenu["Desconecta"])
  b.principal = fyne.NewMainMenu(b.oMenu["BancoDeDados"])
  b.janela.SetMainMenu(b.principal)
  b.laylay =  layout.NewGridLayout(4)
  b.conteiner= fyne.NewContainerWithLayout(b.laylay,
     widget.NewLabel("ID"),widget.NewLabel("NOME"),
     widget.NewLabel("PROGRAMADOR"),widget.NewLabel("SALARIO/HORA"))
                              b.item["Programadores"]                =
widget.NewTabItem("Programadores",b.conteiner)
  b.laylay2 =  layout.NewGridLayout(4)
  b.conteiner2= fyne.NewContainerWithLayout(b.laylay2,
     widget.NewLabel("PRODUTO"),widget.NewLabel("TAREFA"),
```

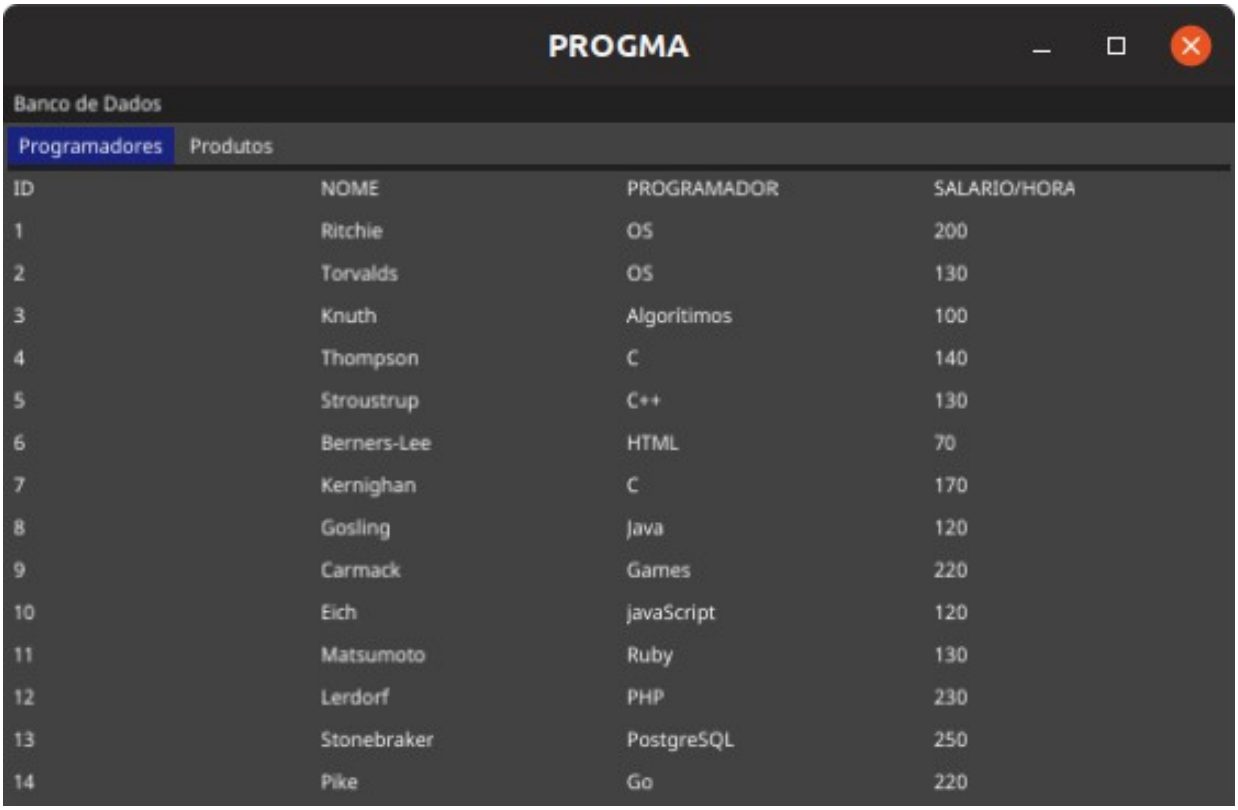
```
     widget.NewLabel("ID PROGRAMADOR"),widget.NewLabel("CONCLUÍDO"))
  b.item["Produtos"] = widget.NewTabItem("Produtos",b.conteiner2)
                                                b.aba                    =
widget.NewTabContainer(b.item["Programadores"],b.item["Produtos"])
  b.janela.SetContent(b.aba)
  b.janela.ShowAndRun()
}
func iniciaProgma() *Progma {
  pgm := &Progma{}
  pgm.item = make(map[string]*widget.TabItem)
  pgm.oMenu = make(map[string]*fyne.Menu)
  pgm.itemMenu = make(map[string]*fyne.MenuItem)
  return pgm
}
func main() {
  b := iniciaProgma()
  b.carregaUI(app.New())
}
```

5) Preste atenção na string de conexão para os dados de user e password, ajuste conforme os valores no seu sistema:

```
const sqlInfo = "host=localhost port=5432 user=andre "+
              "password=segredo dbname=progma sslmode=disable"
```

6) Ao executar teremos:

7) Ao clicar no menu em Banco de Dados → conecta teremos na primeira aba:



ID	NOME	PROGRAMADOR	SALARIO/HORA
1	Ritchie	OS	200
2	Torvalds	OS	130
3	Knuth	Algoritimos	100
4	Thompson	C	140
5	Stroustrup	C++	130
6	Berners-Lee	HTML	70
7	Kernighan	C	170
8	Gosling	Java	120
9	Carmack	Games	220
10	Eich	javaScript	120
11	Matsumoto	Ruby	130
12	Lerdorf	PHP	230
13	Stonebraker	PostgreSQL	250
14	Pike	Go	220

e na outra aba:

PRODUTO	TAREFA	ID PROGRAMADOR	CONCLUÍDO
1	algoritimo de escolha	3	true
1	revisar licença	2	false
2	revisar licença	2	false
3	revisar licença	2	false
2	algoritimo de produção	3	true
3	algoritimo estatístico	3	false

Este é um exemplo bem simples de como podemos usar de forma eficiente fyne para criar um aplicativo para visualizar duas tabelas de um banco de dados.

Muitas outras possibilidades existem e continue praticando com todos os widgets e layouts para você conseguir fazer bons aplicativos. A linha de aprendizado é um pouco longa mas é bastante satisfatória e produtiva.

Este livro teve a intenção de te mostrar os aspectos básicos da linguagem Go. O aprimoramento vem com a prática e mais leitura sobre a linguagem.

Apêndice 1 Widgets principais – Tipos e funções

```
type Box struct {
    Horizontal bool
    Children   []fyne.CanvasObject
    // contains filtered or unexported fields
}
func NewHBox(children ...fyne.CanvasObject) *Box
func NewVBox(children ...fyne.CanvasObject) *Box
func (b *Box) Append(object fyne.CanvasObject)
func (b *Box) ApplyTheme()
func (b *Box) Hide()
func (b *Box) MinSize() fyne.Size
func (b *Box) Move(pos fyne.Position)
func (w *Box) Position() fyne.Position
func (b *Box) Prepend(object fyne.CanvasObject)
func (b *Box) Resize(size fyne.Size)
func (b *Box) Show()
func (w *Box) Size() fyne.Size
func (w *Box) Visible() bool

type Button struct {
    Text  string
    Style ButtonStyle
    Icon  fyne.Resource
    OnTapped func() `json:"-"`
    HideShadow bool
}
func NewButton(label string, tapped func()) *Button
func NewButtonWithIcon(label string, icon fyne.Resource, tapped func()) *Button
func (b *Button) Disable()
func (b *Button) Disabled() bool
func (b *Button) Enable()
func (b *Button) Hide()
func (b *Button) MinSize() fyne.Size
func (b *Button) MouseIn(*desktop.MouseEvent)
func (b *Button) MouseMoved(*desktop.MouseEvent)
func (b *Button) MouseOut()
func (b *Button) Move(pos fyne.Position)
func (w *Button) Position() fyne.Position
func (b *Button) Resize(size fyne.Size)
func (b *Button) SetIcon(icon fyne.Resource)
func (b *Button) SetText(text string)
func (b *Button) Show()
func (w *Button) Size() fyne.Size
func (b *Button) Tapped(*fyne.PointEvent)
```

```
func (b *Button) TappedSecondary(*fyne.PointEvent)
func (w *Button) Visible() bool
type ButtonStyle int
const (
    // DefaultButton is the standard button style
    DefaultButton ButtonStyle = iota
    // PrimaryButton that should be more prominent to the user
    PrimaryButton
)

type Check struct {
    Text    string
    Checked bool
    OnChanged func(bool) `json:"-"`
    // contains filtered or unexported fields
}
func NewCheck(label string, changed func(bool)) *Check
func (c *Check) Disable()
func (c *Check) Disabled() bool
func (c *Check) Enable()
func (c *Check) FocusGained()
func (c *Check) FocusLost()
func (c *Check) Focused() bool
func (c *Check) Hide()
func (c *Check) MinSize() fyne.Size
func (c *Check) MouseIn(*desktop.MouseEvent)
func (c *Check) MouseMoved(*desktop.MouseEvent)
func (c *Check) MouseOut()
func (c *Check) Move(pos fyne.Position)
func (w *Check) Position() fyne.Position
func (c *Check) Resize(size fyne.Size)
func (c *Check) SetChecked(checked bool)
func (c *Check) Show()
func (w *Check) Size() fyne.Size
func (c *Check) Tapped(*fyne.PointEvent)
func (c *Check) TappedSecondary(*fyne.PointEvent)
func (c *Check) TypedKey(key *fyne.KeyEvent)
func (c *Check) TypedRune(r rune)
func (w *Check) Visible() bool

type Entry struct {
    sync.RWMutex
    Text        string
    PlaceHolder string
    OnChanged   func(string) `json:"-"`
    Password    bool
    ReadOnly    bool
```

```
    MultiLine   bool
    CursorRow, CursorColumn int
    OnCursorChanged         func() `json:"-"`
    // contains filtered or unexported fields
}
func NewEntry() *Entry
func NewMultiLineEntry() *Entry
func NewPasswordEntry() *Entry
func (e *Entry) DoubleTapped(ev *fyne.PointEvent)
func (e *Entry) DragEnd()
func (e *Entry) Dragged(d *fyne.DragEvent)
func (e *Entry) FocusGained()
func (e *Entry) FocusLost()
func (e *Entry) Focused() bool
func (e *Entry) Hide()
func (e *Entry) KeyDown(key *fyne.KeyEvent)
func (e *Entry) KeyUp(key *fyne.KeyEvent)
func (e *Entry) MinSize() fyne.Size
func (e *Entry) MouseDown(m *desktop.MouseEvent)
func (e *Entry) MouseUp(m *desktop.MouseEvent)
func (e *Entry) Move(pos fyne.Position)
func (w *Entry) Position() fyne.Position
func (e *Entry) Resize(size fyne.Size)
func (e *Entry) SetPlaceHolder(text string)
func (e *Entry) SetReadOnly(ro bool)
func (e *Entry) SetText(text string)
func (e *Entry) Show()
func (w *Entry) Size() fyne.Size
func (e *Entry) Tapped(ev *fyne.PointEvent)
func (e *Entry) TappedSecondary(_ *fyne.PointEvent)
func (e *Entry) TypedKey(key *fyne.KeyEvent)
func (e *Entry) TypedRune(r rune)
func (e *Entry) TypedShortcut(shortcut fyne.Shortcut) bool
func (w *Entry) Visible() bool

type Group struct {
    Text string
    // contains filtered or unexported fields
}
func NewGroup(title string, children ...fyne.CanvasObject) *Group
func NewGroupWithScroller(title string, children ...fyne.CanvasObject) *Group
func (g *Group) Append(object fyne.CanvasObject)
func (g *Group) Hide()
func (g *Group) MinSize() fyne.Size
func (g *Group) Move(pos fyne.Position)
func (w *Group) Position() fyne.Position
func (g *Group) Prepend(object fyne.CanvasObject)
```

```
func (g *Group) Resize(size fyne.Size)
func (g *Group) Show()
func (w *Group) Size() fyne.Size
func (w *Group) Visible() bool

type Label struct {
    Text      string
    Alignment fyne.TextAlign // The alignment of the Text
    TextStyle fyne.TextStyle // The style of the label text
    // contains filtered or unexported fields
}
func NewLabel(text string) *Label
func NewLabelWithStyle(text string, alignment fyne.TextAlign, style fyne.TextStyle) *Label
func (l *Label) Hide()
func (l *Label) MinSize() fyne.Size
func (l *Label) Move(pos fyne.Position)
func (l *Label) Resize(size fyne.Size)
func (l *Label) SetText(text string)
func (l *Label) Show()
func (t *Label) String() string

type ProgressBar struct {
    Min, Max, Value float64
    // contains filtered or unexported fields
}
func NewProgressBar() *ProgressBar
func (p *ProgressBar) Hide()
func (p *ProgressBar) MinSize() fyne.Size
func (p *ProgressBar) Move(pos fyne.Position)
func (w *ProgressBar) Position() fyne.Position
func (p *ProgressBar) Resize(size fyne.Size)
func (p *ProgressBar) SetValue(v float64)
func (p *ProgressBar) Show()
func (w *ProgressBar) Size() fyne.Size
func (w *ProgressBar) Visible() bool

type ProgressBarInfinite struct {
    // contains filtered or unexported fields
}
func NewProgressBarInfinite() *ProgressBarInfinite
func (p *ProgressBarInfinite) Hide()
func (p *ProgressBarInfinite) MinSize() fyne.Size
func (p *ProgressBarInfinite) Move(pos fyne.Position)
func (w *ProgressBarInfinite) Position() fyne.Position
func (p *ProgressBarInfinite) Resize(size fyne.Size)
func (p *ProgressBarInfinite) Running() bool
func (p *ProgressBarInfinite) Show()
```

```
func (w *ProgressBarInfinite) Size() fyne.Size
func (p *ProgressBarInfinite) Start()
func (p *ProgressBarInfinite) Stop()
func (w *ProgressBarInfinite) Visible() bool

type Radio struct {
    Options  []string
    Selected string
    OnChanged  func(string) `json:"-"`
    Horizontal bool
    // contains filtered or unexported fields
}
func NewRadio(options []string, changed func(string)) *Radio
func (r *Radio) Append(option string)
func (r *Radio) Disable()
func (r *Radio) Disabled() bool
func (r *Radio) Enable()
func (r *Radio) Hide()
func (r *Radio) MinSize() fyne.Size
func (r *Radio) MouseIn(event *desktop.MouseEvent)
func (r *Radio) MouseMoved(event *desktop.MouseEvent)
func (r *Radio) MouseOut()
func (r *Radio) Move(pos fyne.Position)
func (w *Radio) Position() fyne.Position
func (r *Radio) Resize(size fyne.Size)
func (r *Radio) SetSelected(option string)
func (r *Radio) Show()
func (w *Radio) Size() fyne.Size
func (r *Radio) Tapped(event *fyne.PointEvent)
func (r *Radio) TappedSecondary(*fyne.PointEvent)
func (w *Radio) Visible() bool

type Select struct {
    Selected string
    Options  []string
    OnChanged func(string) `json:"-"`
    // contains filtered or unexported fields
}
func NewSelect(options []string, changed func(string)) *Select
func (s *Select) Hide()
func (s *Select) MinSize() fyne.Size
func (s *Select) MouseIn(*desktop.MouseEvent)
func (s *Select) MouseMoved(*desktop.MouseEvent)
func (s *Select) MouseOut()
func (s *Select) Move(pos fyne.Position)
func (w *Select) Position() fyne.Position
func (s *Select) Resize(size fyne.Size)
```

```
func (s *Select) SetSelected(text string)
func (s *Select) Show()
func (w *Select) Size() fyne.Size
func (s *Select) Tapped(*fyne.PointEvent)
func (s *Select) TappedSecondary(*fyne.PointEvent)
func (w *Select) Visible() bool

type TabContainer struct {
    Items []*TabItem
    // contains filtered or unexported fields
}
func NewTabContainer(items ...*TabItem) *TabContainer
func (t *TabContainer) CurrentTab() *TabItem
func (t *TabContainer) CurrentTabIndex() int
func (t *TabContainer) Hide()
func (t *TabContainer) MinSize() fyne.Size
func (t *TabContainer) Move(pos fyne.Position)
func (w *TabContainer) Position() fyne.Position
func (t *TabContainer) Resize(size fyne.Size)
func (t *TabContainer) SelectTab(item *TabItem)
func (t *TabContainer) SelectTabIndex(index int)
func (t *TabContainer) SetTabLocation(l TabLocation)
func (t *TabContainer) Show()
func (w *TabContainer) Size() fyne.Size
func (w *TabContainer) Visible() bool

type TabItem struct {
    Text    string
    Icon    fyne.Resource
    Content fyne.CanvasObject
}
func NewTabItem(text string, content fyne.CanvasObject) *TabItem
func NewTabItemWithIcon(text string, icon fyne.Resource, content fyne.CanvasObject) *TabItem

type TabLocation int
const (
    TabLocationTop TabLocation = iota
    TabLocationLeading
    TabLocationBottom
    TabLocationTrailing
)
TabLocation values
```

```
type Toolbar struct {
    Items []ToolbarItem
    // contains filtered or unexported fields
}
func NewToolbar(items ...ToolbarItem) *Toolbar
func (t *Toolbar) Append(item ToolbarItem)
func (t *Toolbar) ApplyTheme()
func (t *Toolbar) Hide()
func (t *Toolbar) MinSize() fyne.Size
func (t *Toolbar) Move(pos fyne.Position)
func (w *Toolbar) Position() fyne.Position
func (t *Toolbar) Prepend(item ToolbarItem)
func (t *Toolbar) Resize(size fyne.Size)
func (t *Toolbar) Show()
func (w *Toolbar) Size() fyne.Size
func (w *Toolbar) Visible() bool

type ToolbarAction struct {
    Icon        fyne.Resource
    OnActivated func()
}
func (t *ToolbarAction) ToolbarObject() fyne.CanvasObject

type ToolbarItem interface {
    ToolbarObject() fyne.CanvasObject
}
func NewToolbarAction(icon fyne.Resource, onActivated func()) ToolbarItem
func NewToolbarSeparator() ToolbarItem
func NewToolbarSpacer() ToolbarItem

type ToolbarSeparator struct {
}
func (t *ToolbarSeparator) ToolbarObject() fyne.CanvasObject

type ToolbarSpacer struct {
}
func (t *ToolbarSpacer) ToolbarObject() fyne.CanvasObject
```