GO (Golang): Develop Modern, Fast & Secure Web Applications

A comprehensive introduction to developing fast & secure web applications in GO (beginners & fairly advanced developers)

Jens Schendel

Version 1.0.177, October 2023

Table of Contents

Section 1 – Introduction

I don't know where I am going, but I am on my way..
Voltaire

Lecture 1 – Allow Me To Introduce Myself - The "whoami" For Udemy Courses

Welcome to the course "GO (Golang): Develop Modern, Fast & Secure Web Applications". My name is Jens Schendel, but as you know, names are just smoke and mirrors and are rather the equivalent of typing the command `whoami` on the bash or any other Linux shell. But also a shell returns only the username the system recognizes you as. So let me drop a few words about myself here.

I am a trained media designer with specialization in media design and media operating, I work as a front-end web developer as well as a Linux administrator and system operator. In the latter field I have acquired a few additional certificates in the last few years. However, you have probably already had a look at my instructor's profile on Udemy or visited me on LinkedIn.

I'm an enthusiastic programmer by passion, running an insignificant English-language YouTube channel for C programming-related topics called myCTalkthroughs. When I discovered Google's still relatively young programming language GO for myself, I immediately found great fun in the simplicity and ease with which one can work on even more complex programming topics. And from this I developed an equally great fun in sharing this knowledge in the form of courses on Udemy some in German, but mainly in English language.

My courses (also as free tutorials) on Udemy have so far been target more at beginners, but my small select group of students tended to ask for topics that teach advanced skills recently. I'm happy to fulfill that request with this course!

And we'll leave it at that with the self-introduction, let's better get started! Or in the words of the great contemporary American philosopher Rick Sanchez:

"Wubba Lubba Dup Dup!" - Get your portal gun ready – here we go!

Lecture 2 – Why GO Of All Things? Why Not Node.js Or Another Programming Language?

Long story short: GO is awesome.

Programming with GO feels like you're dealing with C, but someone has removed everything that feels like you have to work around the pain, and instead added a few features that you would have otherwise had to program yourself anyway.

Who invented it?

- Google (GO on Wikipedia), in person of

- Rob Pike, Unix, UTF-8
- Robert Griesemer studied with the inventor of Pascal, worked on Google's V8 JavaScript engine
- Ken Thompson led the implementation of U**x and invented the B programming language and thus the predecessor of C and participated in bringing C to life.

Why now exactly GO again?

- Highly efficient compiling
- GO creates compiled programs
- There is a "garbage collector" (GC)
- There is no virtual machine in which the code is executed, no execution layer or environment, no emulator and no interpreter
- Fast execution times
- Ease of use, "Ease of programming"

In summary, three main features that make GO so successful:

1. compiles easily and very quickly – even large projects compile in seconds and minutes, not hours
2. efficient execution with very high execution speed
3. Ease of Programming – programming should be done with ease, not pain in the brain

Brad Fitzpatrick put it together in some slides in 2014 and these slides are available on go.dev: https://go.dev/talks/2014/gocon-tokyo.slide#31

What GO can be used for

- Basically, everything "that Google does" and all internet services that need to meet the highest standards and be highly scalable.
- Networking, Server-Client programming
- http/https, tcp, udp
- concurrency / parallel programming
- conditional system programming
- automation, command-line tools
- cryptography
- image processing

Creation principles

- meaningful, understandable, sophisticated
- clean, clear, easy to read

Companies using GO also in the website go.dev

Google, YouTube, Netflix, Paypal, Meta, Microsoft, Twitter, Docker, Kubernetes among others, InfluxDB, Twitter, Apple, Cloudflare, DropBox, more examples in detail

Trivia:

The inventor of Node.js has abandoned Node in favor of GO instead

GO programmers are currently the highest paid programmers in the US – 5th in the world (2017)

Lecture 3 – Installation Of GO And Setup Of A Free Integrated Development Environment (IDE)

Installing and setting up GO is trivial these days, and I'll only cover it in rudimentary detail.

In a nutshell: download GO for your operating system and suitable processor architecture and then follow the installation instructions.

GO comes with its own compiler already, but a good IDE has many advantages. Very well known are LiteIDE, GOSublime and GOLand. Unfortunately, as far as I know, they are not free. Of course, with a good text editor like Atom, VIM or Notepad++ and a few extensions, you can certainly build your own development environment that supports features like syntax highlighting and automatic compilation and much more. But for a course like this, I recommend Visual Studio Code. It's free, meets all the requirements of a good IDE for me, and is available for Linux, macOS, and Windows.

Installation again boils down to download and install. Done.

Please note that your installation will definitely look a bit better, I've set quite large fonts here so you can still easily read the code in the videos even on a mobile device.

Now let's install one more extension for GO that we might need to trust. For this we open the preferences and there extensions and search for GO, install it. You may need to restart Visual Studio GO. If you want, you can still install the Github support for Visual Studio Code. After that we are ready to go. That was easy, wasn't it?

If something didn't work with the installation of GO, or Visual Studio Code, I refer you to the websites go.dev and code.visualstudio.com. A common problem is the missing definition of a so-called environment variable, so that your user, respectively, your operation system can find GO at all. The internet should help you.

Lecture 4 – Learning Notes On This Course

To get the most out of this course, I recommend that you download the PDF with the course outline and have it at your fingertips throughout the course. Please be mindful of our environment and refrain from printing the PDF. The PDF has a table of contents with an internal link that allows you to quickly jump to the current lesson in the PDF. Make use of that. Here and there are external sources linked in the PDF.

This course is offered in English, but as you will hear, English is not my native language. I strive for clear pronunciation. Perhaps the automatically generated subtitles will help you. They are very good on Udemy and rarely have problems with abbreviations or anything like that.

However, if you do not understand a topic at all, you will get the tip in other courses to watch the lesson again directly. Personally, I don't think this is helpful and recommend the following instead: Try to research the topic yourself. Google, websites like gobyexample or stackoverflow, or even the GO website itself offer enough starting points. If the topic is still not clear, continue with the course. Often concepts become clearer in hindsight when you see them applied in a clear use case.

I put some thought into the structure of the course and put the basic topics and concepts at the beginning. Please do not skip any lessons unless you are really familiar with the topic covered. I place comments and further explanations if at the end of each lesson if there are some.

Then very important: Join in. You don't learn programming by watching others typing code. Follow the lessons and make sure you understood the content before you move on to the next lesson. And for this recommendation, it's also important to remember that typing, not copy&paste, is the way to learn. Typos are widely underrated learning guides that help form the proper memory structures in your brain. If you copy a finished result, you miss out on this progress.

Lecture 5 – Accompanying Course Outline As PDF (Also Available On Github)

The download of this PDF only.

Lecture 6 – Brief Section And Content Overview

An brief overview over the course for you to get a first impression of what awaits you.

Lecture 7 – Basic Sources Of Information On The Web About GO And Used Software

A brief list of useful sources to learn about Google's programming language GO.

About GO

- GO Website
- The GO Playground
- GO’s Standard Library
- GO Packages
- GO Specifications
- GoByExample

Integrated Development Environment (IDE)

- Visual Studio Code

Dependencies of this project

- github.com/go-chi/chi/v5 | Router
- github.com/alexedwards/scs/v2 | Sessions
- github.com/justinas/nosurf | CSRF-Token
- github.com/asaskevich/govalidator | Validator (server-sided)
- github.com/jackc/pgx/v5 | PostgreSQL Driver & Toolkit
- github.com/xhit/go-simple-mail | Golang package for sending e-mail
- Caddy 2 | a powerful, enterprise-ready, open source web server with automatic HTTPS written in Go

Also playing a part

- github.com/twbs/bootstrap | Bootstrap - HTML, CSS, and JavaScript framework (no jQuery)
- RoyalUI-Free-Bootstrap-Admin-Template | Free Bootstrap 4 Admin Template
- github.com/fiduswriter/Simple-DataTables | DataTables but in TypeScript transpiled to Vanilla JavaScript
- github.com/postgres/postgres | PostgreSQL Server (mirror only)
- github.com/gobuffalo/pop | Soda/Migrations - standardization of database tasks
- github.com/dbeaver/dbeaver | Dbeaver - free multi-platform database tool
- github.com/mymth/vanillajs-datepicker | Vanilla JavaScript datepicker
- github.com/jaredreich/notie | unobtrusive notifications - clean and simple JavaScript
- github.com/jackc/pgx/v5 | SweetAlert2 - so many options for JavaScript popups
- github.com/mailhog/MailHog | MailHog - Web and API based SMTP testing
- Foundation for Emails 2 | Quickly create responsive HTML e-mails that work
- Cobra | A Framework for Modern CLI Apps in Go
- GoDotEnv | A Go port of Ruby's dotenv library

Section 2 – Brief dive into GO as a crash course

> *I'm trying to free your mind, Neo. But I can only show you the door. You're the one that has to walk through it.*
>
> *Morpheus*

Lecture 8 – Take This Hint And This One, And This One Too!

And we are already in section 2, where I will give you a brief overview of programming techniques commonly used in GO.

Now, in the previous section we introduced an integrated development environment, and I've been telling you in epic length to please follow along with all, don't skip any lessons, and type the code as well, and now let's break all those rules.

First of all, our IDE should be reserved for less trivial tasks. That's why I'm switching to the GO playground in these lessons for now. This has several advantages. First, you don't have to install anything. Then, the GO playground is available practically everywhere when you're online, and it provides the ability to create links and thus share code easily. These links then also go into the course outline. So no IDE and no typing.

Also, you can skip the whole section if you already have experience with GO or have worked through my GO beginner course. Of course you can also see this few lessons as a refresher.

So that's GO on the fast track now.

Lecture 9 – Hello, World

So that's GO on the fast track now.

Let's quickly go to the playground. It can be reached at go.dev/play.

The page may look a little different on your end, I've hidden the GO logo here and I prefer dark mode in my browser -- same as the way I drink coffee and my soul deep inside.

The playground works on any modern browser and always comes up with the same „Hello World“ program on every reload:

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt
```

```
func main() {
     fmt.Println("Hello, 世界")
}
```

By the Chinese characters here, which means something like "world" you can already see a special feature of GO. GO is completely written in UTF-8 and has no problems to process and output UTF-8 compatible characters. If this fails on the shell of your computer, it is due to the lack of support of your terminal, not GO.

GO programs themselves are hust a text file whose name ends in .go. Each file must belong to a package. A package can consist of several files. The package here is called `main`, because it also contains the function `main()`. This does not have to be the case, but it often is.

One, and exactly one, function `main()` is the minimum requirement to create an executable GO program, because exactly here is the entry point to the program flow. The function `main()` is declared by the keyword `func`, the identifier, let's say the name of the function, and two parentheses followed by curly braces marking the beginning and the end of the function.

We find another function called `Println()` that takes (amount other data types“ values of type string as an argument. This function comes from another package called `fmt`. If we want to use the function, we have to import the package `fmt` before. After that we can use the function `Println()` from the package `fmt`. We make the assignment using the dot operator. We can only use functions, variables and constants from imported packages, if they were exported from the package. GO does not know keywords like export or private/public for this.

You may have noticed that the identifier of the function `Println()` starts with a capital letter. This is not a characteristic of the programmer, but the common method to make a function, variable or constant available outside the package. Programming can be that simple.

Lecture 10 – Variables - First Things First

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

func main() {
     fmt.Println("Hello, 世界")
}
```

GO is a strictly or strongly typing programming language with static types. This means that all variables must be declared with one type and this type does not change during the runtime of the program. Values can be converted, but must then be stored in a variable of another type.

The declaration of a variable is always accompanied by an initialization.

```
var x int
```

declares a variable with the identifier x of type integer, i.e. a „whole number“. The variable is implicitly initialized with a zero value if we do not explicitly assign a value to it. This value is not, as for example in C, some random value that is currently in the variable's memory, but the so-called zero value, a predefined zero value for each type with which the variable is initialized.

For integers, the zero value is zero. Let's print it by adding:

```
fmt.Println(x)
```

Okay.

If we declare x with the keyword var outside a function, it has package-wide scope. This is called package scope. We can even initialize it with a value right away - maybe 23.

```
var x = 23
```

In this case, GO sets the type for us, which it infers from the assigned value.

Let's take a look at the type.

```
fmt.Printf("x is of Type %T\n", x)
```

Side note: `Printf()` is used for formatted output of a string with placeholders introduced by % characters. Here % capital T is used to retrieve the type. The variable for which the wildcard applies is the trailing comma-separated x. Backslash-n is used here in `Printf()` simply as a means to break lines.

Within and really only within a function there is the declaration and initialization with implied type specification a shorthand notation.

```
Y := 42.0
```

With this we create a variable y and assign it the value 42.0. GO should recognize that we now want to get a floating point number, but execution fails. We get the error:

```
y declared but not used
```

Why is this? A variable that does not have packet-wide scope can be created in shorthand notation with a colon and an equal sign with scope limited to the function, but it is mandatory that such a variable be used. the minimal use I can think of is to output the value of y. We'll also take the opportunity to output the type for y.

```
fmt.Println(y)
fmt.Printf("y is of Type %T\n", y)
```

Now the execution works and also the type assignment worked is clear.

For the sake of completeness we now also create a string. A string is a type in GO and the length of the string is a part of this type.

```
z := "Doh"
fmt.Println(z)
fmt.Printf("z is of Type %T\n", z)
```

We have learned about the keyword `var`, seen use of package-wide scope, we now know about implied assignment of the zero value on type assignment, but also implied type assignment on value assignment, and have used the short declaration operator, the colon equals sign.

For variables, let's leave it at that for now and take a look at functions.

Look at code on the playgorund

Lecture 11 – All Functional?

Nearly all imperative programming languages allow procedural structures in the form of functions. GO is no different.

Functions can be created wherever you want with package-wide scope. Simplified, the declaration is called the signature of a function. It looks like somthing like this:

```
// func (r receiver) identifier(arg argumentType, ...)
(returnType(s)) { code }
```

By the way, the two slashes here are a common notation for comments in GO.

Let's go through and create a simple output function.

Keyword func, receiver r we'll put back, that comes later. We ommit it here.

Identifier `outputThat`

Three comma separated arguments a, b, c of type int, float64 and string.

We want to return something here and this is where another special feature of GO comes into play, which is probably unknown to you if you have experience in Java, C or php: GO can have multiple return values. This is a unique feature of GO that we will make extensive use of. More later.

Here we want to output a status message `done` of type `string` and maybe a boolean value.

```
func outputThat(a int, b float64, c string) (string, bool) { }
```

Now we also want to execute the function and pass it x, y and z. We call `outputThat()` inside `main()`.

```
outputThat(x, y, z)
```

But we will get two return values. We need to have them taken by something and we do that using the `:=` operator.

```
done, trueOrNot :=
```

Here together one line:

```
done, trueOrNot := outputThat(x, y, z)
```

However, we also need to output these values. You remember. Every variable declared inside a function must also be used. Assignment is not enough for that.

```
fmt.Println(done, trueOrNot)
```

Before we execute that, however, our function must also do something and return something. Let's implement this:

```
fmt.Printf("Integer = %d, FloatingPoint = %f, String = %s\n", a, b, c)
return "Okay.", true
```

Now a few important notes.

You can also create functions without identifiers. This is called anonymous functions, which can then be executed immediately by appending simple parentheses.

Also, there is no distinction between "call-by-value" and "call-by-reference" for the value passing of the arguments, which means that the mode of passing arguments is always call-by-value. Always with a function call, with which values are passed, copies of these values are created, which exist exclusively within and at the execution time of a function.

This means here, for example, that while we can assign new values to our a, b, and c, the variables x, y, and z are completely unaffected. This is also true if we would use the identifiers x, y and z within our function `outputThat()`. Even though they look the same and are of the same type, they are completely separate entities.

If we want to change values outside the function, we have to pass pointers. We will learn more about pointers in the next lesson, but this much can already be said here: When passing pointers, a copy is created again and the method of calling a function with passing arguments remains the same: call-by-value. Only that a pointer value is passed. In other programming languages this is called call-by-reference. Better get used to cal-by-value only! We will leave that here for now.

And last but not least, the rule with a variable or constant is exported from the package by capitalizing the first character of the identifier also applies to functions! This means that if someone would import our package main, he could not call `outputThat()` with

```
main.outputThat(23, 42.0, "Opps.")
```

However, for a function `OutputThat()`, a call to

```
main.OutputThat(23, 42.0, "Yuppie.")
```

would be possible.

Look at code on the playground

Lecture 12 – Pointer – Pointing The Finger At Others

If you come from a programming language like php, pointers are probably a completely new concept to you. However, pointers are incredibly useful.

You could make a video of an hour's duration to explain pointers in detail, and I've already done just that on my YouTube Channel for C programming. But that wouldn't be purposeful here, because - and this as a hint though for all of you who already know something about pointers: Pointers in GO do not need the sometimes difficult to understand and therefore dreaded pointer arithmetic. Therefore, we deal here exclusively with the fundamental concept of a pointer as it is used in GO.

To explain pointers, I use strings here. Then we have already mentioned the topic. Strings in GO are a data type of their own! Basically, a string of 32-bit wide integers: `int32`. However, each of these 32-bit numbers is interpreted as UTF-8 characters. And to distinguish numbers from UTF-8 characters, one uses a separate datatype for UTF-8 compatible 32-bit integers: runes. The datatype `rune` in turn is nothing else than an alias to `int32`. A string is a separate datatype, which is a string of such runes. Now that's not quite right either, because the string datatype doesn't contain the whole data itself, but a starting point in memory where the first rune can be found, a length within memory, and a capacity up to which runes - UTF-8 compliant characters - can be added to a string without much loss of time, without any back-and-forth copying going on under the hood to create contiguous space for our string in memory.

Whew - a lot of information at once about strings. Very briefly again: `string` is a data type. The values are strings of UTF-8 characters of data type `rune`. Strings have a length and a capacity.

And now let's go to the playground.

What do strings have to do with pointers?

Obviously nothing at first, at least nothing that would be important here now. I use a string here to explain pointers.

```
aCostume := "Clip Clop"
fmt.Printf("Roger is wearing a costume as %s.\n", acostume)
```

If I now try to pass this string to a function, it works up to a certain point for now.

```
func changeTheCostume(c string) {
     fmt.Printf("Roger is wearing a costume as %s.\n", c)
     newCostume := "Ricky Spanish"
     c = newCostume
     fmt.Printf("Roger is wearing a costume as %s.\n", c)
}
```

but if I change the content here, and print it after calling the function here in `main()`

```
changeTheCostume(aCostume)
fmt.Printf("Roger is wearing a costume as %s.\n", aCostume)
```

you will see that nothing has changed. Even if I use the same identifier inside the function now, it doesn't work. To proof that claim you can try to exchange `c` for `aCostume`.

Note: At this point in the video, `c` is temporary replaced with `aCostume`.

As I explained to you in the lesson on functions, when we pass a value into a function, a copy of that value is created. We can change the value of that copy, but our change only affects the copy then as the scope of that copy is within the function only. The copy literally exists only at runtime of this function.

Of course, one could come up with the glorious idea to make all variables with `var` available package-wide outside of all functions, but I can only warn against that. Every variable takes up memory. This may not matter at all with one or two int, but with self created data types of maybe several megabyte size you will fill up the memory very fast. You should always call only those variables into existence that you can actually handle.

This is where pointers come in.

```
func changeTheCostume(c *string)
```

With the asterisk operator we tell our function that we do not expect a value of type string, but a type pointer to a string. So we don't want to read the value, we just want to know where the value is. The pointer points to data of a type. Remember: asterisk with type makes a pointer type.

But now we must not pass the value of the string, but only the address in memory where the string value can be found. For this we use the preceding ampersand (or Roman „et“, which means „and“ by the way).

```
changeTheCostume(&aCostume)
```

And in a third step we now have to assign a new value to Roger's costume.

But we only have a place where this value is stored in memory. Here the asterisk comes into play again, but this time not with the type, but with the identifier. What was just a reference is now dereferenced, or in simple words: We want to access the value that is stored in that location. And to this we assign a new content. This time, of course, not a pointer value, but a real string value.

```
*c = newCostume
```

Inside our function `c` is no longer a string, but of type pointer to a string. If we want to print the value stored at that spot in memory with `Printf()`, we have to dereference it as well.

```
fmt.Printf("Roger is wearing a costume as %s.\n", *c)
```

Ta-da! You can see that Roger can now change his costume without any problems.

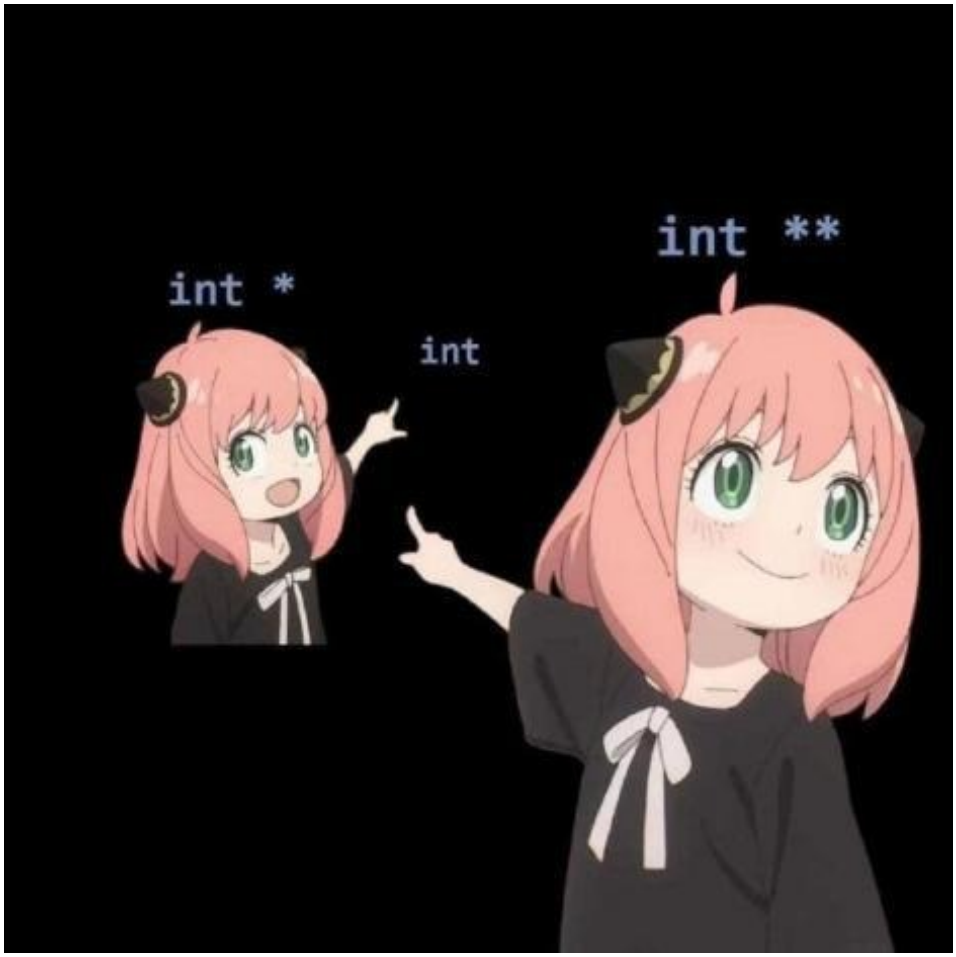


Figure 1: (Source: https://i.redd.it/r01sq96mnz691.jpg)

Now it should be clear where the name pointer comes from. They are pointers that point to a certain place in memory where a certain value of a certain type is stored. You can take the game even further and create pointers to pointers, because a pointer itself represents a value that must be stored somewhere in memory.

Look at Code on the playground

Summary, what we should have understood by now:

An asterisk "*" together with a type defines a pointer to a type.

Ampersand "&" in front of a variable’s identifier returns the address of the variable, and represents a value that a pointer contains.

Asterisk "*" together with identifier, a variable name, however, represents a dereference of the value pointed to by the pointer.

All clear?

Lecture 13 – The Shadow World - It's Always About Types And Structs

Variable shadowing

Before we turn to types and structs, let's take a quick look at the secret shadow existence that variables can live.

```
// You can edit this code!
// Click here and start typing.
package main

import "fmt"

var role1 = "Billionaire"

func main() {
     var role2 = "Inventor"
     fmt.Println("Bruce Wayne is", role1)
     fmt.Println("Bruce Wayne is", role2)

     tellMeWhoYouAre("Batman")
}

func tellMeWhoYouAre(role1 string) string {
     fmt.Println("Bruce Wayne is", role1)
     return role1
}
```

In this simple example, if we swap the name of the argument of the function and for the return value from `role1` to `role3`, the value defined globally for the package becomes valid. From this we can see that the scope of variables declared within a function have a higher priority than the global declaration. You have to let this sink in, because it also means that variables can have a secret shadow existence, which you don't know about, because you are not aware that they have a secret shadow existence in some function imported somewhere under the same identifier. This is called variable shadowing. So always be careful with the naming. This is also the reason, why in functions often rather generic variable names with one letter are used, globally or in the main program however rather meaningful ones - I assume now simply so.

Variable shadowing can also happen if I declare `role1` again within the `main()` function and assign a value to it. I will do this with the short declaration operator.

```
role1 := "playboy"
```

This is totally allowed in main() and so role1 becomes "playboy".

https://go.dev/play/p/AAtDF7mxhtd

So much for variable shadowing. If you take a look at the specification of GO on https://go.dev/ref/spec, you will quickly see: It is all about types and typing. Besides the usual suspects, even strings are a type as well as even functions themselves are just types.

One type stands out that we'll look at now, and that's structs. You may be familiar with structs from a variety of the object-oriented programming languages. Whether GO should be counted among them is still a matter of debate. In any case, GO also offers structs.

Structs are used to structure data

The easiest way to explain what a struct is is to try to describe the Smith family with simple properties.

```
var familyName string
var firstName string
var age int
var birthdate time.Time
var species string
```

However, this quickly gets out of control if you want to pass such a group for a single record to a function, or perhaps store it in a database.

But you can combine such records in a structure using the keyword struct and usually you create your own datatype for it, which you give its own identifier.

```
type FamilyMember struct {
     FamilyName string
     FirstName  string
     Age         int
     Birthdate  time.Time
     Species    string
}
```

Now we have declared a type, now we declare a value. This works very fine with the short declaration operator inside main().

```
francine := FamilyMember{
```

```
          FamilyName: "Smith",
          FirstName:  "Francine",
          Species:    "Human",
     }
```

And now we can print the complete data set, or individual elements.

```
fmt.Println(francine)
fmt.Println(francine.FirstName)
```

If we add a Roger, the principle is quickly clear.

```
roger := FamilyMember{
          FamilyName: "Smith",
          FirstName:  "Roger",
          Age:        1600,
          Species:    "Alien",
     }

fmt.Println(roger)
fmt.Println("Roger is", roger.Age, "earth years old.")
```

There is little to consider when using structs in GO.

First, you can only use a self-defined type outside of a package if the identifier of the type is capitalized. Second, of course, the same applies to each element of the structs. The identifier of a struct or the elements is no different than that of a variable, constant or function.

On the other hand you have to take care that you don't have to repeat the element identifier when assigning values to the elements. But this is part of the good tone and best practice. And, that all value assignments must be separated by a comma - even the last one. Otherwise this is simply a syntax error.

So now you know about variable shadowing and have learned about the use of structs.

View code on the playground

Lecture 14 – Receiver - The Madness Gets Method

What is a method? Classically, a method is a function that is an element of a struct.

What you may be familiar with as a method from object-oriented programming has also found its way into GO and is not strictly implemented as an element of a struct.

What is meant by this? In C++, for example, functions can be integrated as elements in a struct and defined as a function with access to the other struct elements using the dot operator. This way you get something like an internal function that you can call like an element. This structure is known as a method.

In GO, to create a method, you use the receiver r, which I kept from you in the lesson introducing functions. That's exactly what we're catching up on now. I've added a few values to the example from the previous lesson to represent the members of the Smith family from the show American Dad. I've removed the Date of Birth element. And output all the values once because you can't create unused variables in GO.

Let's take a look: https://go.dev/play/p/BtgNcikw83A

I'll add a function right here below the struct:

```
func (r FamilyMember) sayYourName() string {
     fullName := r.FirstName + " " + r.FamilyName
     return fullName
}
```

The function does not have to be below the structs. In principle, you can create it anywhere in the package, because the connecting part here is the element Receiver r.

And now let's use such a method.

Let's say Roger wants to get his complete name from Klaus. We do it like this:

```
fmt.Println("Hello my name is", roger.sayYourName(), "- show me
your ID card and I tell you who you are. Klaus says: ",
klaus.sayYourName())
```

Here we call the method `sayYourName()` of two different values. This is how you create methods in GO. You simply create a function that as receiver r refers to a type defined as struct. Done.

View code on the playground

Lecture 15 – Maps And Slices

Now let's push the envelope a bit and cover two topics again in this lesson - maps and slices. After all, we want to start building a web application in the foreseeable future. And the current section only deals with the basics of the GO programming language.

As a programmer, you spend a lot of your time giving data a structure or finding the appropriate data type for a structure of data.

Maps

To make large amounts of values of unordered data pairs available, there are so-called hash tables. Explaining hash tables in detail is far beyond the scope of this lesson, but there is a very nice article on medium.com, that explains the basics of hash tables and the concrete implementation in GO in easy to understand English.

hash tables in GO are called maps. Maps are a composite data type that makes unordered pairs of data findable by a key value. Let's look at a simple example. Very often user data is used here because this data is usually unique. We can easily imagine that a service like Google has the need to identify single users out of many millions of user accounts very quickly by their - let's say - e-mail address and to keep this data unordered and quickly searchable. Let's use this as an example.

First we create a map on the playground. This can be done with the Short Declaration Operator and is quite simple and understandable by using the keywords make.

```
//    mapIdentifier := make(map[keyType]dataType)
```

`make` itself is a wrapping command that ensures that memory is reserved for some data types, that this memory is extended and also released at the end of the program run. As argument we call a datatype - here map with the properties we need.

The map datatype looks like this, the datatype label map is followed by square brackets holding the type of a key value, followed by a datatype. Maps in GO are not rocket science.

```
users := make(map[string]string)
```

And now let's create some data.

```
users["peter@griffin.family"] = "Peter Griffin"
```

and then we do the test and output the value we have stored under this e-mail address:

```
fmt.Println(users["peter@griffin.family"])
```

Works on the first try. We add a few more data pairs.

```
users["lois@griffin.family"] = "Lois Griffin"
users["meg@griffin.family"] = "Meg Griffin"
users["chris@griffin.family"] = "Chris Griffin"
users["stewie@griffin.family"] = "Stewie Griffin"
users["brian@griffin.family"] = "Brian Griffin"
```

Can we also output the complete `map`?

```
fmt.Println(users)
```

You can see that this also works, but it should be used with caution, because retrieving a few million records can take some time. Please note that the individual records are not output in the same or reverse order in which they were created.

A few resourceful programmers thought they could save themselves some work and use this fact like an index, but this is not the way the GO developers envisioned using Maps. Therefore, they have taken care to ensure that the output is always in a different order. Maps are just unordered lists.

What happens if we now want to add another record with the same key value?

```
users["stewie@griffin.family"] = "Steward Griffin"
fmt.Println(users["stewie@griffin.family"])
```

Of course, this does not work because each key value is unique. Re-creating a data value with an already existing key simply overwrites the already created value.

Deleting a value is easy and works with the `delete()` function, which takes the identifier of the map and the key value of the element to be deleted.

```
delete(users, "stewie@griffin.family")
fmt.Println(users["stewie@griffin.family"])
```

And here you can also see that a key value not present in a map does not lead to an error message. So if you want to check if a value is present in a map or not, there is another way.

Last but not least there is the question what is the zero value of a map. So, what is in there if the map has no values yet? We simply add at the appropriate place:

```
fmt.Println(users)
```

But we only receive:

```
map[]
```

Instead, we must either determine the length with the `len()` function or know that the pointer of an empty map is of data type `nil`. This is because under the hood there is a pointer that points to the first element of a map. If this is not present, it has the value `nil`.

```
fmt.Println(len(users))
fmt.Println(users)
```

It seems here that users already has a value not equal to `nil`. This is because when we use the short declaration operator and make, the pointer is already directed to a memory address where GO plans to create the data of the structure. To infer an empty map using `nil` is only possible if we created the map using `var`. Then we have also seen this once:

```
var dummyMap map[string]string
fmt.Println(dummyMap == nil)
```

This zero value exists not only for empty maps but also for the data types `pointer`, `function`, `interface`, `channel` and `slice`. Basically, `nil` serves as a value for an empty pointer, so that it doesn't have to point to the number or address zero, and all these data types describe different composite structures that are located somewhere in memory. And it's the latter, `slice`, that we're going to take a look at now.

View code on the playground

Slices

A slice in GO is basically a data type built on top of an array to address the shortcomings of dealing with arrays, making the values of any data type available in sequential order. Other data types are built on top of slices. For example, the data type string is a slice of the underlying data type `byte`, which in turn is just a series of unsigned 8-bit values. In string, these are interpreted as rune on output, but then according to UTF-8 in packets of 4. But that's just as a side note.

To explain slices, let's create a typical array.

```
var meeseeks [3]string
```

```
fmt.Println(meeseeks)
```

Now quickly a few values that we can also easily access:

```
meeseeks[0] = "Meeseek#1"
meeseeks[1] = "Meeseek#2"
meeseeks[2] = "Meeseek#3"
fmt.Println(meeseeks)
fmt.Println(meeseeks[0])
```

If we want to add another meeseek, it will fail.

```
meeseeks[3] = "Meeseek#4"
invalid argument: index 3 out of bounds [0:3]
```

View code on the playground

For us, this means that when using arrays, the size of the array must usually already be fixed at compile time. But this is rarely the case. You can now either assume an array that is far too large and waste a lot of memory, or you have to dynamically reserve memory and add it to your arrays. But this brings speed problems at runtime. Each extension or reduction of an array means finding new contiguous storage, copying the existing data there, creating the additional data there as well, and finally deleting or freeing the previous storage. If you have to do this once in a while, you might be okay, but if you do it for every element that is added, dynamically reserved memory for arrays quickly becomes a test of strength for any computer. And then imagine trying to insert or delete an element at a specific location without creating a gap, and doing it with millions of elements.

Basically, dynamically allocated memory for arrays is always a trade-off between execution speed and flexibility. In addition, the programming techniques for this must first be mastered and then used with consideration for the corresponding processor architecture. Different rules apply for a workstation than for a Raspberry Pi.

The solution that GO has found to prevent these and other problems in connection with arrays from arising in the first place is the slice types.

Slices can be created in several ways.

1. with `var` just like arrays, by omitting the number in the preceding brackets. With this you leave all the basic settings and the initialization to the compiler.
2. with short declaration operator and a direct value assignment for at least one element.

3. or with `make`. This should be the preferred method, if you already know the initial size and an expected size at runtime or at least can reasonably estimate it.

Let's look at all three options on the Playground. We will learn to create slices, assign values, and we will learn about two properties that are a part of your type: Size and Capacity

https://go.dev/play/p/nAKPG3ogHth

Very important: Slices are "immutable", so to speak. What does this mean? Of course you can assign a value to a slice and then assign a new value. And apparently the value changes then. But the truth is that a new slice is created with the desired properties of the new slice. You remember, size and capacity are properties of a slice and different slices are not compatible with each other.

Therefore it is necessary to use functions for operations with slices. We have learned about `append()`, but there is no delete() function! As we said, it is necessary to rebuild a slice anyway and reassign the value and that works with `append()`.

We'll just do that very briefly here, because deleting something by adding something sounds paradoxical at first if you don't know that you can continue slicing a slice without any problems.

Let's take the slice `someFloats`:

```
[42.42 0 0 23.23 123.456 555.555 888.888] 7 12
```

We want to delete 123.456 from the slice. To do this, we take the slice and assign a new value. But we start only up to a part from the beginning including up to the part that should be removed exclusively. We use a selector separated by a colon inside square brackets following the identifier of the slice. Following the same principle, we then add the part that should follow. Remember that the index starts at zero.

```
someFloats = append(someFloats[0:4], someFloats[5:7])
```

which gives us an error:

```
cannot use someFloats[5:7] (value of type []float64) as type
float64 in argument to append
```

`append()` does not expect a slice to be appended, but any number of single values of the underlying data type. We need to "unroll" or "unspool" those somehow. And in GO, there's a very simple operator for this. We append three dots to the slice that we want to pass as single values.

```
someFloats = append(someFloats[0:4], someFloats[5:7]...)
```

This already works great, but there is an even clearer, shortened notation if we want to select "from the beginning" and the "to the end". We can then simply omit the index completely:

```
someFloats = append(someFloats[:4], someFloats[5:]...)
```

There are already some functions that make your life easier, and which are optimized for handling slices and speed and extensively tested. Example is the sorting, which I will demonstrate with the method `Ints()` from the package `sort`:

```
sort.Ints(someIntegers)
fmt.Println(someIntegers, len(someIntegers), cap(someIntegers))
```

Bam! Done.

You don't have to worry about memory allocation, claiming and freeing at all. Slices are one of the features in GO that make "Ease of Programming" possible in the first place. If arrays are programmer's bread and butter, slices are kind of like "super food" like goji berries on steroids. The GO developers strongly recommend the use of slices instead of arrays.

View code on the playground

Lecture 16 – Decisions & Conditionals – If, Else, Else If, Switch

Decisions & Conditionals - if, else, else if, switch

Have you ever struggled with making a decision? I think we all have at one time or another. Yet we make hundreds maybe thousands of decisions every day - good ones and not so good ones. The conditions on which our decisions are based are varied both in the nature of the condition and in the value itself and importance of that value.

Computers have it easier. They are, at the end of the day, very binary in nature. You could think of them as a very widely and sophisticated light switch that knows exactly two states: on and off, 1 and 0, true and false.

So for a computer there is no answer to the question "yes or no?" like "maybe", "approximately" or "something like that". Therefore, all states can basically be checked for "true" or "false".

Certainly known from other programming languages are keywords like `if, else, else if`.

We'll have a quick look at them and take the opportunity to get to know `Println()` from the `log` package. (More explanations in the video)

Watch the code on the playground

The only important thing to remember is that the logic of nesting `if, else if,` and `else` statements in GO is represented by indentations.

More than one `else if` query is rarely useful. If you need more complex logic, you can usually resort to a `switch` statement.

Let's look at an example for this as well. (Further explanation in the video)

View code from the Playground

Lecture 17 – In Da Loop: "For" And "Range" As A Team

No other loops, no while, no do-while-loops.

3 steps:

1. Initialization
2. Condition
3. Value change

For data types where length, number of elements, changes dynamically, you can finally take advantage of the fact that length is a part of that data types. You can query them. This is the basis of keyword range, which is used to iterate over the values of various data types in for-loops.

https://go.dev/play/p/h7VkPNB4T38

https://go.dev/play/p/QqWrZ__fsvi

Lecture 18 – Polymorphism: Interfaces - The Name Says It All

Interfaces are a special feature of GO. Interfaces serve to establish a property of many object-oriented programming languages: Polymorphism.

Basically, values can be of different types. First of all, each value has a data type assigned to it, but from certain points of view, different values can be considered as belonging to one or more other data types as well. This is called polymorphism and is achieved in GO exclusively with a data type called interface.

An interface is also in real life a link between different types, which creates a connection between them. Let's take a look at an example.

https://go.dev/play/p/HQT40d_4DCB

This means that when a type is declared to be of type interface, it means something like: If any type implements following methods with following return values of certain types, then GO also considers it to be of the type of that interface.

So you can create a parent type that you just generically require that whatever else it does, it is also of that very type if it satisfies certain conditions.

Now it becomes clear why interface{}, called "empty interface", means something like "all types". Everything, to which no condition is attached to fulfill the type "empty interface", is also of the type "empty interface". So there is no restriction to be of type "empty interface" and that applies to all data types.

Lecture 19 – GO Modules

A brief overview of using GO’s built-in package management: GO modules

Package management is crucial for the development process in any programming language. With the introduction of GO modules, GO has taken a big step forward in the management of packages and dependencies. GO modules provide a way to manage the dependencies of your project, making it easier to share your code with others and collaborate on projects. They also help ensure that the correct version of a package is being used, reducing the risk of compatibility issues. GO modules make it easy to keep track of your dependencies, making it simpler to upgrade or revert to previous versions if necessary. With GO modules, you can be confident that your code will work as expected, regardless of changes to your dependencies.

GO modules

Lecture 20 – Channels are the key to concurrent communication in GO!

In this video, we'll explore the fascinating world of channels in GO, a powerful feature that enables you to communicate and synchronize between multiple GO routines.

In this code, we created a channel c using the make function. We then used a GOroutine to send the value 42 to the channel c. Finally, in the main function, we printed the value received from the channel c using the <- operator.

As you can see, channels allow you to safely pass data between GO routines. But, that's just the tip of the iceberg! Channels also allow you to control the flow of communication, buffer the data, and even close the channel to signal the end of communication.

If you have problems to understand the concept of channels you better have a look this examples on gobyexample.com.

When using channels in GO, there are several important considerations to keep in mind:

- Channel Direction: Channels can be either bidirectional or unidirectional. It is important to choose the right direction for your use case, as it affects the way data is transmitted and received.
- Buffer Size: Channels can be buffered or unbuffered Buffered channels can store data without blocking, while unbuffered channels block until the data is received. It is important to choose the right buffer size for your use case to ensure efficient communication.
- Channel Closing: Channels can be closed to signal the end of communication. When a channel is closed, any attempts to send data to it will result in a panic. It is important to handle channel closing correctly to avoid unexpected behavior.
- Synchronization: Channels can be used to synchronize GO routines. However, care must be taken to avoid deadlocks, where two or more GO routines are waiting for each other to complete.
- Data Types: Channels can transmit any data type in GO, but it is important to ensure that the correct data type is being transmitted and received.

Overall, it is crucial to understand the properties and behavior of channels in GO to effectively use them in your projects.

View code on the playground

Lecture 21 – Import And Export Of Data In JSON Format

The exchange of data between programs written in different programming languages is a common practice in software development. One widely adopted method for this purpose is using the JSON (JavaScript Object Notation) format, which is a lightweight and text-based data interchange format. JSON provides a standard way of representing data that is easy to read and write, and can be parsed and generated by almost all modern programming languages.

In GO, data can be imported and exported in various formats such as CSV, JSON, XML, and more. I talk you through two simple examples to import and export data in JSON format.

Importing data from JSON

To import data from a JSON file, we can use the `encoding/json` package provided by GO's standard library.

Suppose we have a JSON file named `data.json` with the following content:

```
[
  {
    "name": "Stan",
    "age": 42,
    "gender": "Male"
  },
  {
    "name": "Francine",
    "age": 37,
    "gender": "Female"
  }
]
```

We can import this data into our GO program as follows:

```
package main

import (
     "encoding/json"
     "fmt"
```

```
    "os"
)

type Person struct {
    Name   string `json:"name"`
    Age    int    `json:"age"`
    Gender string `json:"gender"`
}

func main() {
    file, err := os.Open("data.json")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    decoder := json.NewDecoder(file)
    var people []Person
    if err := decoder.Decode(&people); err != nil {
        panic(err)
    }

    fmt.Println(people)
}

   fmt.Println(people)
}
```

Here, we first open the JSON file using the `os.Open` function and check for any errors. We then create a new `json.Decoder` using the opened file and declare a slice of Person structs to hold the decoded data. Finally, we decode the JSON data using the Decode method and print the result to the console. Note that we pass a reference to the people slice so that the Decode method can populate it with the decoded data.

Exporting data to JSON

To export data to a JSON file, we can use the `encoding/json` package provided by GO's standard library.

Suppose we have a slice of structs representing some data:

```
type Person struct {
     Name   string `json:"name"`
     Age    int    `json:"age"`
     Gender string `json:"gender"`
}

people := []Person{
     {"Peter", 44, "Male"},
     {"Lois", 43, "Female"},
}
```

We can export this data to a JSON file named `data.json` as follows:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Person struct {
    Name   string `json:"name"`
    Age    int    `json:"age"`
    Gender string `json:"gender"`
}

func main() {
    people := []Person{
```

```
        {"John", 25, "Male"},
        {"Mary", 31, "Female"},
    }

    file, err := os.Create("data.json")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    encoder := json.NewEncoder(file)
    encoder.SetIndent("", "  ")
    if err := encoder.Encode(people); err != nil {
        panic(err)
    }

    fmt.Println("Data exported to data.json")
}
```

Here, we first create a slice of `Person` structs representing the data we want to export. We then create a new file using `os.Create`, and check for any errors. We create a new `json.Encoder` using the created file, and set the indentation for the output using the `SetIndent` method. Finally, we encode the data using the `Encode` method, and print a message to the console confirming the export.

Lecture 22 – Unit Tests

Here's an example of how to perform unit tests in GO using the built-in testing package.

Let's say we have a simple function that adds two integers:

```
func Add(a, b int) int {
    return a + b
}
```

To write a unit test for this function, we can create a new file named `add_test.go` in the same package directory, with the following contents:

```
package main

import "testing"

func TestAdd(t *testing.T) {
     // Test cases
     cases := []struct {
          a, b, want int
     }{
          {1, 2, 3},
          {0, 0, 0},
          {-1, 1, 0},
          {-1, -1, -2},
     }
     // Iterate over test cases
     for _, c := range cases {
          got := Add(c.a, c.b)
          if got != c.want {
               t.Errorf("Add(%d, %d) == %d, want %d", c.a, c.b,
got, c.want)
          }
     }
}
```

This code defines a test function named `TestAdd` that will run a series of test cases, each of which consists of two input integers and the expected output of the `Add` function. The test function uses the `t.Errorf` method to report an error if the function returns an unexpected result.

To run the test, we can use the `go test` command in the terminal from the package directory:

```
go test .
```

This will run all test functions in the package and report the results. In this case, if everything is working properly, we should see output like the following:

```
PASS
ok      _/path/to/package    0.001s
```

If any test cases fail, go test will report an error with a message that includes the details of the failure.

That's a simple example of how to perform tests in Golang.

In summary

Golang has a built-in testing package that provides a simple and consistent framework for writing and running tests.

Test files in Golang follow a naming convention of `*_test.go`, and are located in the same package directory as the code being tested.

Test functions in Golang begin with the prefix Test, and take a single argument of type `*testing.T`, which is used to report test failures and log messages.

Test functions should be well-structured and easy to read, with clear and descriptive test cases that cover all aspects of the code being tested.

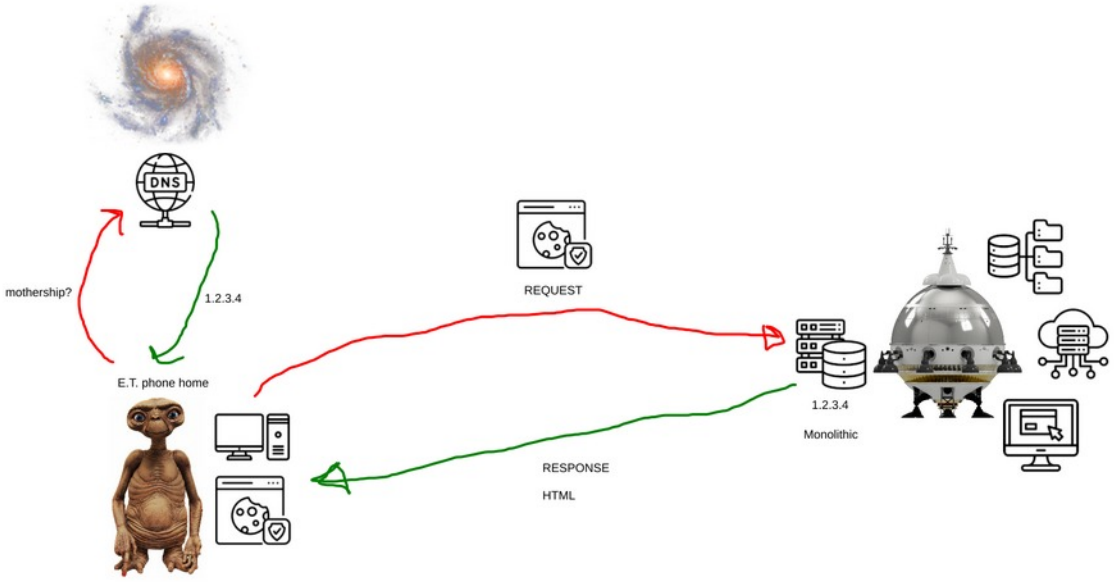
Golang supports a variety of testing methods and tools, including unit testing, integration testing, benchmarking, and code coverage analysis.

By following these guidelines, you can write effective tests for your Golang code and ensure that your software is reliable and robust.

Section 3 – Basic Web Application - The Beginning

Things work in cycles.
Joan Jett

Lecture 23 – The HTTP Request/Response Cycle



Lecture 24 – The First Web Application "It's Alive! It's Alive!"

This code in GO sets up a basic HTTP server that listens on port 8080 and responds to any HTTP requests sent to the root path ("/") with the message "Hello, 世界" (which means "Hello, world" in Chinese).

The `http.HandleFunc()` method sets up a handler function that takes two arguments: a `http.ResponseWriter` interface and a `*http.Request` pointer. When a request is made to the root path, this function is called and it writes the message "Hello, 世界" to the `http.ResponseWriter`. The `fmt.Fprintf()` function is used to write the message to the response writer and returns the number of bytes written and any error encountered during the write operation.

The last line of the `main()` function starts the HTTP server by calling the `http.ListenAndServe()` method, which listens on the specified port and serves incoming

requests using the registered handler function. The underscore character before the `http.ListenAndServe()` call is used to discard any error value returned by the function, which is a common way to handle errors that are not expected to occur.

v1.0.24

Lecture 25 – Unleash The Full Potential Of Handlers With The Magic Of Functions!

We set up a web server listening on port 8080 that responds to requests for two URLs: "/" and "/about".

The "/" URL is handled by the `Home` function, which simply writes a string to the `ResponseWriter`. The "/about" URL is handled by the `About` function, which calls the `getData` function to get a name and a saying, and the `addValues` function to add two numbers. The results of these function calls are then written to the `ResponseWriter` in a formatted string.

The `main()` function sets up the handlers for the two URLs, starts the web server, and prints a message indicating that the application has started.

v1.0.25

Lection 26 – Errors Have Value And Are A Value

A beneficial feature in the GO programming language is the feature of functions to provide two or more return values. This gave its designers the option to handle errors where they potentially occur, during the actual operation, rather than having every possible error escalate to a potential program crash and then be handled in bulk. And to achieve that, there is even a distinct `error` data type that you can check for. We'll look at that in the following example.

v1.0.26

Lection 27 – HTML Templates: Because Ain't Nobody Got Time To Code That From Scratch!

So far we have only delivered simple text, but GO was developed by Google and is equipped with an extensive standard library to, - yes - to do everything on the net, what Google just does. And this also includes the delivery of HTML, and there again there is a package that provides templates for this. And how to use it, we will now have a look at.

A note: The inclusion of the package `text/template` served here now rather educational purposes, in order to clarify the procedure in handling HTML templates in GO. Later we will use a more sophisticated package for this purpose.

v1.0.27

Lection 28 – Organize And Conquer: Let's Tidy Up And Optimize Our Space!

We cleaned up the code, split it into various files, and finally integrated Bootstrap into our HTML templates.

Bootstrap

v1.0.28

Lection 29 – Restructuring – A Structure Like From The Textbook

We create a folder structure that is supported by `go modules`, consider our program parts that we have distributed to different files as packages - like the third-party ones. We move our application and our new packages to the appropriate folders and make the necessary changes in the code so that `go modules` can find and import them accordingly.

```
myGOWebApplication
├── cmd
│   └── web
│       └── main.go
├── go.mod
├── LICENSE
├── pkg
│   ├── handlers
│   │   └── handlers.go
│   └── render
│       └── render.go
├── README.md
└── templates
    ├── about-page.tpml
    └── home-page.tpml
```

v1.0.29

Lection 30 – Layouts Like A Boss

We extend our templates and use layouts to generically generate web pages for our responses.

v1.0.30

Lection 31 – A Dynamic Cache For Effective Template Processing

Learn how to build a simple dynamically growing cache.

v1.0.31

Lection 32 – Creating A Static Cache #1: Efficient Template Processing

A static cache for templates offers advantages for our web application.

v1.0.32

Lection 33 – Creating A Static Cache #2: Introducing Configuration File For Global Variables

We use a global variable that we make available in a configuration file.

v1.0.33

Lection 34 – Creating A Static Cache #3: Final Step Of Implementation Using Global Variables

We use the global variable defined in the configuration file to create a static cache that can be turned on and off.

v1.0.34

Lection 35 – What Else You Can Do With A Configuration File

Notes on the use of a configuration file.

Lection 36 – Sharing Is Caring: Sharing Data With Templates

Learn how to share information with templates to process data.

v1.0.36

Section 4 – GO With The Flow: An Introduction To Middlewares In GO!

> *A horse is dangerous at both ends and uncomfortable in the middle.*
> *Ian Fleming*

Lecture 37 – Introduction Of Middleware And Routing In GO

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

https://dev.to/karankumarshreds/middlewares-in-go-41j

https://www.alexedwards.net/blog/which-go-router-should-i-use

Lecture 38 – Implementation Of A Simple Routing Package (bmizerany/pat)

We import a first external packages and prepare some routes with it.

bmizerany/pat

v1.0.38

Lecture 39 – Developer's Favorite: go-hhi/chi As New External Routing Package

Exchange of pat with chi as routing and middleware package.

go-chi/chi

v1.0.39

Lecture 40 – Middleware: DIY Your Own Middleware Today And Be The Coolest Coder In Town

Learn how to create middleware in GO with a quick practice and become a middleware master in no time!

justinas/nosurf

v1.0.40

Lecture 41 – State Management With Session(s)

Implementation of a middleware in the form of a package for the creation and management of session data.

alexedwards/scs

v1.0.41

Lecture 42 – Brief Function Test For Session Data

A quick test to see if passing data embedded in sessions works.

v1.0.42

Section 5 – Project Picking And Working With Forms: A Paperless Dream!

May your choices reflect your hopes, not your fears.
Nelson Mandela

Lecture 43 – Thoughts On Project Selection

Learn to roughly outline a project and estimate scope and workload.

PLANING

I. What do we want our web application to be?

II. What is the scope?

III. What are the basic functionalities?

I. WEB APPLICATION

- Booking/Reservation
- Two objects/services

II. SCOPE OF THE WEB APPLICATION

- Landing page or showcase page
- Single pages for the rental objects
- Availability query
- Booking/reservation execution
- Notification system

III. FUNCTIONALITIES

- Back-end & Login
- Overview of current bookings
- Calendar of upcoming and/or past bookings
- Changes/cancellation of a booking/reservation

What do we need to implement?

- Secure authentication (login)

- Data storage, i.e. database

- Messaging system (SMS or e-mail)

Lecture 44 – Brief Note on Github

A brief note of why to use github.com may be a good idea!

Lecture 45 – Static Files: Hold still and get integrated!

To serve static files from a specific folder is easy:

Pixabay

v1.0.45

Lecture 46 – HTML - A Trip To The 1990th

A simple example how HTML works.

v1.0.46

Lecture 47 – Spot Landing! We Create a Landing Page

We set up a serverless development environment to create HTML pages that we will later convert into GO templates. We start with the landing page.

v1.0.47

Lecture 48 – Preparing The HTML Of The Bungalow Pages

With a simple landing page, bungalow pages are 1-2-3-Finished!

v1.0.48

Lecture 49 – Create And Pimp An Availability Check HTML Page

Creation of an HTML page with a form to get dates from the user.

v1.0.49

Lecture 50 – make-reservation.html Is Our Answer To: "Do You Have A Reservation?"

An HTML form to make a reservation.

v1.0.50

Section 6 – Code-Kaboom! JavaScript And CSS Come Into Play

> *Design is not just what it looks like and feels like. Design is how it works.*
>
> *Steve Jobs*

Lecture 51 – JavaScript: Friend Or Foe?

v1.0.51

Lecture 52 – Effortlessly Pick Dates: Grabbing A Vanilla Js Datepicker Package Now!

We take control of date selection by importing a third-party (Vanilla) JavaScript package.

Vanilla JavaScript Datepicker

v1.0.52

Lecture 53 – Notie By Nature: Show Simple Messages

Implementation of JavaScript package „notie“.

notie

v1.0.53

Lecture 54 – Sweetalert: Candy Time!

Implementation of JavaScript package „sweetalert2“.

sweetalert2

v1.0.54

Lecture 55 – Sweetalert Is A Candy Store - Our Own JavaScript Module

We create our own JavaScript module.

JavaScript modules:
developer.mozilla.org
www.freecodecamp.org

v1.0.55

Lecture 56 – From Boring Button To Superstar: A New Functionality In Our JavaScript Module

We implement a new function in our JavaScript module and try to deploy it to a button.

v1.0.56

Lecture 57 – CSS: Making Websites Less Ugly Since 1996

Swapping your own CSS to a local file, sorting through CSS imports, and a quick look at how CSS works.

v1.0.57

Section 7 – Turn HTML Into GO Templates, Server-Side Validation And Even More Handlers

> *Coming together is a beginning, staying together is progress, and working together is success.*
>
> *Henry Ford*

Lecture 58 – Brief Overview What's Going On In This Section

Overview of this section.

v1.0.58 (You can start with this code base which is already cleaned up some)

Lecture 59 – From HTML To "Happily Ever After": Conversion Into GO Templates

Learn how to turn plain HTML into GO templates.

v1.0.59

Lecture 60 – CSRF-Token – Implementation

We provide security against Cross-Site Request Forgery (CSRF) by generating a token for each potential POST-Request using the middleware NoSurf.

v1.0.60

Lecture 61 – Unlocking The Power Of JSON In Golang: A Handler That Returns Data In JSON

Explore how to use Golang to create a handler that returns data in JSON format.

v1.0.61

Lecture 62 – Preparations For Submitting And Processing AJAX Requests

In this lesson, you will learn how a request via JavaScript to our web application can trigger a response from our JSON data providing handler, interpret its response, and use the data in JavaScript!

V1.0.62

Lecture 63 – From GET To POST: Let'S Teach The AJAX Requests Some Manners!

In addition, you will learn to customize the `custom()` function in JavaScript to make it more generic.

v1.0.63

Lecture 64 – Pimp Your Code: Refactoring Made Easy!

Keep your code’s structure clean – do refactoring.

v1.0.64

Lecture 65 – Server-Sided Validation - The What, The How And The Why Of It All!

A small overview of what server-side validation is and why it makes more sense than client-side validation (alone) in many cases.

Client-side vs. Server-side Validation (StackOverflow)
Client-side vs. Server-side Validation (Medium)

Lecture 66 – Implementation Server-Side Form Validation I - Form Field Data & Errors

Evaluate form data and identify errors.

v1.0.66

Lecture 67 – Implementation Server-Side Form Validation II - Forms Model & Error Displaying

Create a packages model that holds data models and create new validators.

v1.0.67

Lecture 68 – Implementation Server-Side Form Validation III - More Fields & A Required Func

Create more fields and make several of them "required" fields at once.

v1.0.68

Lecture 69 – Implementation Server-Side Form Validation IV - More Validators & Govalidator

Learn how to use an external package that provides additional validators.

v1.0.69

Lecture 70 – Display Of An Overview Of The Reservation Data (By Using Sessions)

Use sessions to display the collected reservation data on a summary page after the reservation.

v1.0.70

Lecture 71 – Fast Feedback: Output Alerts As Feedback To The User Via notie

Use `notie` to give the user feedback on their actions.

v1.0.71

Lecture 72 – Alternative Template Engine: Use The Power Of A Jet Engine

Introduction of an alternate template engine in GO.

Templates in the GO Standard Library

Performance Comparison of some GO Template Engines (2020)

The CloudyKit/jet Template Engine

Example from this lesson can also be downloaded in the lecture’s resources.

Code example to use the Jet Template Engine

Section 8 – Putting Your Code To The Test: How Writing Tests Can Save The Day (Or Days!)

> *A good test is one that has a high probability of breaking the code if there is a problem*
>
> *Michael Bolton*

Lecture 73 – Testing In GO: The Why And Wherefore

About the purpose, but also the importance of tests in GO.

My GO/Golang Course For Beginners

Lecture 74 – Testing Success: Mastering Tests For Package Main Of Our Web Application

Learn how to test the package `main` of your web application when the test itself executes a function `main()`.

v1.0.74

Lecture 75 – Handlers Tests I - The Beginning: Initial Setup/Handling GET-Request Handlers

Learn how to test GET request handlers.

v1.0.75

Lecture 76 – Handlers Tests II - Continued: Handling POST-Request Handlers

Learn how to test POST-request handlers.

v1.0.76

Lecture 77 – Render Tests I - Creating A Test Environment And Function Testadddefaultdata()

To test a package under always the same environment, it is necessary to create such an environment.

v1.0.77

Lecture 78 – Render Tests II - Creating Tests For Function Testrendertemplate() And The Rest

Learn how to add render tests for the other functions in the package.

v1.0.78

Lecture 79 – Coverage Of Package Handlers And Package Render Tests

Take a look at the coverage of the test for the packages `handlers` and `render`.

Lecture 80 – Hands-On Exercise: Write A Basic Test For Package "forms"

Write a basic test for the package `forms`.

v1.0.80

Lecture 81 – A Solution: [Solved] Testing For Package "forms"

A solution for testing the functions in the package `forms`.

v1.0.81

Lecture 82 – Final Notes and Tips for Starting Our Web Application

You are welcome to write a batch file or script to simplify the invocation of your application.

Section 9 – Striving for Improvement: Error Handling

> *Better three hours too soon than a minute too late.*
>
> *William Shakespear*

Lecture 83 – Consolidation Of Error Handling In A Package "helpers"

Learn how to combine part of the error code evaluation in a package `helpers` and evaluate it centrally.

v1.0.83

Lecture 84 – Use Of ClientError And ServerError And Updates Of The Relevant Tests

Learn how to evaluate error codes and distinguish between client errors and server errors.

v1.0.84

Section 10 – Database I - Introduction To Database Usage And SQL With PostgreSQL And DBeaver

> *You can have data without information, but you cannot have information without data.*
>
> *Daniel Keys Moran*

Lecture 85 – Brief Section Overview And Download/Installation Of PostgreSQL And DBeaver

- My Free Training on Udemy Running a Simple Webserver Performing CRUD Actions on a PostgreSQL Database Server
- PostgreSQL Server Download
- Postgres.app Download (macOS only)
- DBeaver Community Edition (CE) Download
- PostgreSQL vs mySQL

Lecture 86 – Linux: Installing PostgreSQL And DBeaver And Making A Connection

Here Debian-based distributions, for SUSE-based distributions analogously with package management Yum:

```
sudo apt-get update
sudo apt-get install -y postgresql dbeaver

service postgresql start
service postgresql status
```

And now, as user postgres, we start an administration tool called PSQL. Basically we open a terminal into the postgres server. Ignore the error message here, which basically just says that user postgres doesn't have permissions in our home directory.

```
sudo -u postgres psql
```

PostgreSQL terminal:

```
\password
exit
```

Start DBeaver (here from the terminal, of course you can also choose this from the start menu):

```
dbeaver
```

Create new connection to PostgreSQL using user postgres and the newly created password.

Lecture 87 – macOS: Installing PostgreSQL And DBeaver And Making A Connection

PostgreSQL: Download standalone Postgres.App and install it by drag&drop in Applications.

Postgres.app download

Start Postgressql.App, set password.

DBeaver: Download and run installer, create new connection to PostgreSQL (download any necessary drivers) using user `postgres` and the newly created password.

Lecture 88 – Linux: Installing PostgreSQL And DBeaver And Making A Connection

PostgreSQL: Download and run installer for PostgreSQL. Accept EULA, accept standards, allow access, set password. PostgreSQL server should start in the background.

DBeaver: Download and run installer, create new connection to PostgreSQL (download any necessary drivers) using user `postgres` and the newly created password.

Lecture 89 – CRUD - Now It's Getting Dirty! SQL-Statements In Action

Learn the basic handling of the necessary actions you want to be able to perform in databases:

- **C**reate
- **R**ead/**R**etrieve
- **U**pdate
- **D**elete/**D**estroy

known together as **CRUD**.

Lecture 90 – SQL Queries For Advanced Users - Not Necessarily Complicated, But Complex

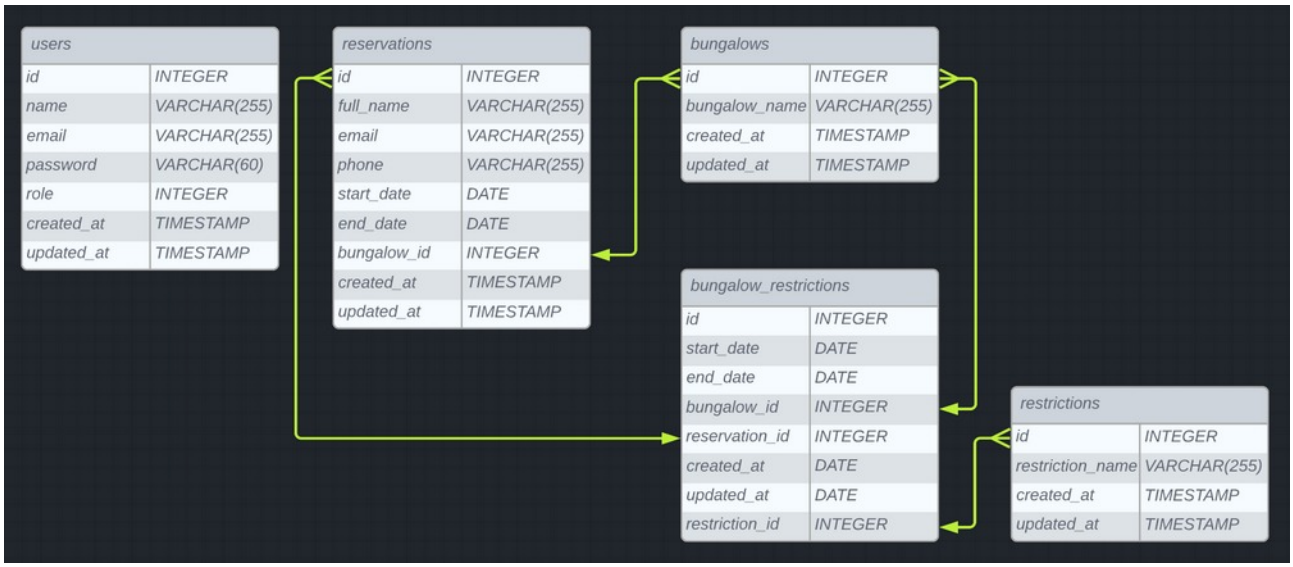
Learn beyond CRUD statements, clauses and options to create more complex database queries.

Section 11 – Database II - Creation And Necessary Structuring Of The Database

> *Structure is more important than content in the transmission of information.*
>
> *Abbie Hoffman*

Lecture 91 – Fascination Database Structure: Creation Of An Entity Relationship Diagram!

A brief look at how to create an entity relationship diagram.



Lecture 92 – *pop* "Want A Soda?" - Installation Of gobuffalo/pop Called Soda!

We compile and install a command line tool for database "pressure fueling" taken out of the web developing framework gobuffalo!

Complete Framework gobuffalo or web development in GO:

Installation of the complete framework (not necessary for the scope of this course).

But we want to use only a part called migrations: automation and "version management" for databases.

And so we don't install the complete framework, we resort to a command line tool included in the Buffalo framework called "pop".

But this "pop" command can also be downloaded or installed as a standalone version, then it is called "Soda". And this soda is available either as a binary for the common operating systems, or if you have GO installed, it is a matter of a few minutes to install it.

You should be able to install your tools - if they are already available as GO source code - yourself. You should also have GO available ready to run.

If you prefer to Homebrew on macOS, or use a binary on whatever, operating system, that's both perfectly fine too!

Windows

Binary download and place it in one of the following folders:

```
C:\Windows
C:\Windows\System
C:\Windows\System32
C:\Users\Username\go\bin
```

Installation:

```
go install github.com/gobuffalo/pop/v6/soda@latest
```

Test:

[WIN + R]

```
cmd.exe
powershell.exe

soda.exe -v
```

macOS

Binary (Note architecture!) and place it in one of the following folders:

```
/usr/local/bin
/usr/local/go/bin
```

Installation:

```
go install github.com/gobuffalo/pop/v6/soda@latest
```

Test:

Terminal:

```
which soda
soda -v
```

Linux

Binary download and place it in one of the following folders:

```
/usr/local/bin
/usr/local/go/bin
```

Installation:

```
go install github.com/gobuffalo/pop/v6/soda@latest
```

Test:

Terminal:

```
which soda
soda -v
```

Lecture 93 – Migrations I - Creation Of The "Users" Table

Database Setup

Migrations/FIZZ

FIZZ

database.yml:

```
development:
  dialect: postgres
  database: mygowebapp
  user: postgres
  password: [PUTYOURPASSWORDHERE]
  host: 127.0.0.1
  pool: 5
```

```
test:
  url: {{envOr "TEST_DATABASE_URL"
"postgres://postgres:postgres@127.0.0.1:5432/myapp_test"}}


production:
  url: {{envOr "DATABASE_URL"
"postgres://postgres:postgres@127.0.0.1:5432/myapp_production"}}
```

soda generate fizz CreateUsersTable

```
UP:
create_table("users") {
t.Column("id", "integer", {primary: true})
t.Column("full_name", "string", {"default": ""})
t.Column("email", "string", {})
t.Column("password", "string", {"size": 60})
t.Column("role", "integer", {"default": 1})
}

DOWN:
drop_table("users")
```

v1.0.93

Lecture 94 – Migrations II - Mass Production: Creation Of All Other Tables

soda generate fizz CreateReservationsTable

```
UP:
create_table("reservations") {
  t.Column("id", "integer", {primary: true})
  t.Column("full_name", "string", {"default": ""})
  t.Column("email", "string", {})
  t.Column("phone", "string", {"default": ""})
  t.Column("start_date", "date", {})
  t.Column("end_date", "date", {})
```

```
  t.Column("bungalow_id", "integer", {})
}

DOWN:
drop_table("reservations")
```

soda generate fizz CreateBungalowsTable

```
UP:
create_table("bungalows") {
  t.Column("id", "integer", {primary: true})
  t.Column("bungalow_name", "string", {"default": ""})
}

DOWN:
drop_table("bungalows")
```

soda generate fizz CreateBungalowRestrictionsTable

```
UP:
create_table("bungalow_restrictions") {
  t.Column("id", "integer", {primary: true})
  t.Column("start_date", "date", {})
  t.Column("end_date", "date", {})
  t.Column("bungalow_id", "integer", {})
  t.Column("reservation_id", "integer", {})
  t.Column("restriction_id", "integer", {})
}

DOWN:
drop_table("room_restrictions")
```

soda generate fizz CreateRestrictionssTable

```
UP:
create_table("restrictions") {
  t.Column("id", "integer", {primary: true})
```

```
  t.Column("restriction_name", "string", {"default": ""})
}

DOWN:
drop_table("restrictions")
```

v1.0.94

Lecture 95 – Migrations III - Creation Of A Foreign Key For The "Reservations" Table

soda generate fizz CreateFkForReservationsTable

```
UP:
add_foreign_key("reservations", "bungalow_id", {"bungalows": ["id"]}, {
    "on_delete": "cascade",
    "on_update": "cascade",
})

DOWN:
drop_foreign_key("reservations", "reservations_bungalows_id_fk", {})
```

v1.0.95

Lecture 96 – Migrations IV - To Be Continued ... The Remaining Foreign Keys

soda generate fizz CreateFkForBungalowRestrictionsTable

```
UP:
add_foreign_key("bungalow_restrictions", "bungalow_id", {"bungalows": ["id"]}, {
    "on_delete": "cascade",
    "on_update": "cascade",
```

```
})


add_foreign_key("bungalow_restrictions", "restriction_id",
{"restrictions": ["id"]}, {
    "on_delete": "cascade",
    "on_update": "cascade",
})


DOWN:
drop_foreign_key("bungalow_restrictions",
"bungalow_restrictions_restrictions_id_fk", {})


drop_foreign_key("bungalow_restrictions",
"bungalow_restrictions_bungalows_id_fk", {})
```

v1.0.96

Lecture 97 – Hands-On Exercise: Add The Missing Foreign Key To "bungalow_restrictions"

An exercise for you: create the missing foreign key in the bungalow_restrictions table.

Lecture 98 – A Solution: [SOLVED] The Missing Foreign Key For "bungalow_restrictions"

A solution for creating the foreign key in the `bungalow_restrictions` table.

```
UP:
add_foreign_key("bungalow_restrictions", "reservation_id",
{"reservations": ["id"]}, {
    "on_delete": "cascade",
    "on_update": "cascade",
})


DOWN:
drop_foreign_key("bungalow_restrictions",
"bungalow_restrictions_reservation_id_fk", {})
```

v1.0.98

Lecture 99 – Migrations V - Nitro Injection: Index For "users" And "bungalow_restrictions"

A previously created index for frequently queried and extensive data sets speeds up database queries quite considerably!

```
soda generate fizz CreateIndexForBungalowRestrictionsTable
```

```
UP:
add_index("bungalow_restrictions", ["start_date", "end_date"], {})
add_index("bungalow_restrictions", ["bungalow_id"], {})
add_index("bungalow_restrictions", ["reservation_id"], {})

DOWN:
drop_index("bungalow_restrictions",
"bungalow_restrictions_start_date_end_date_idx")
drop_index("bungalow_restrictions",
"bungalow_restrictions_bungalow_id_idx")
drop_index("bungalow_restrictions",
"bungalow_restrictions_reservation_id_idx")
```

v1.0.99

Lecture 100 – Hands-On Exercise: Add Useful Indexes To The "reservations" Table

Consider for which data in the `reservations` table an index would be useful and create them.

Lecture 101 – A Solution: [SOLVED] Useful Indexes For The "reservations" Table

A proposed solution for migrations to create an index for each of the `email` and `full_name` columns.

```
soda generate fizz CreateIndexForReservationsTable
UP:
add_index("reservations", ["email"], {})
add_index("reservations", ["full_name"], {})

DOWN:
drop_index("reservations", "reservations_email_idx")
drop_index("reservations", "reservations_full_name_idx")
```

v1.0.101

Lecture 102 – Migrations VI - "The Sting" For The Development Phase Of The Database

```
soda reset
```

executes all "down" migrations before it executes all "up" migrations. This returns the database to its original well-defined state.

Section 12 – Database III - Connection of the PostgreSQL Database to the Web Application

> *You don't have to like someone [...] to make a connection.*
>
> *Jennifer Aniston (made up by chatgpt)*

Lecture 103 – Example: How To Connect An Application To A Database In GO

Free Training (Udemy, 2h) to connect a PostgreSQL Server and a WebApp

Lecture 104 – PostgreSQL Connection: Just like Golf! No Driver When You Need One Urgently!

Creation of a package "driver" using the packages github.com/jackc/pgx to create/access PostgreSQL databases.

github.com/jackc/pgx

Reference on pkg.go.dev (Documentation)

v1.0.104

Lecture 105 – Integration Work: Inserting the Driver/Database Connection (Repository Pattern)

Linking the Package "drivers" to the Web Application Using Repository Pattern.

v1.0.105

Lecture 106 – An Easy Time: Creation Of The Necessary Models

Added more models in the form of data types to be able to read and store values from the database.

v1.0.106

Lecture 107 – An Easy Time: Creation Of The Necessary Models

Especially if you are working on your own, always question notation and/or procedure from different points of view and change the source code where it makes sense to you. Always put

yourself in the position of an expert colleague. Could he or she reproduce your code without considerable effort?

v1.0.107

Lecture 108 – Don't Make It Complicated - But You Can If You Like: Object-Relational Mapping

Brief overview of Object-Relational Mapping (ORM), definition, benefits, advantages/disadvantages, and abandonment statement.

Object-Relational Mapping (ORM) – Wikipedia

GORM

upper/db

Lecture 109 – Double Trouble: Reservation Creation And Storage In The Database

Implement the database function to store the generated reservation in the database.

v1.0.109

Lecture 110 – Poking With A Stick: Short Functional Test Of The Reservation Function

Try out your newly implemented functions regularly to avoid unpleasant surprises.

v1.0.110

Lecture 111 – One Small Step For Man... Database Entry In BungalowRestrictions

What LastInsertID in MariaDB or mySQL is, we can use „returning id“ for.

v1.0.111

Lecture 112 – Availability Check: Check Availability For A Specific Date Range Per Bungalow

Create a database function that checks the availability of a bungalow for a given time period.

The database queries in SQL used in the video are:

```
-- check exact start and end date, expected outcome: 1
select
```

```
      count(id)
from
      bungalow_restrictions
where
      '2024-02-01' < end_date and '2024-02-04' > start_date;

-- check start date before restricted date and same end date, expected
outcome: 1
select
      count(id)
from
      bungalow_restrictions
where
      '2024-01-31' < end_date and '2024-02-04' > start_date;

-- check same start date and end date after restricted date, expected
outcome: 1select
      count(id)
from
      bungalow_restrictions
where
      '2024-02-01' < end_date and '2024-02-10' > start_date;

-- search dates outside of restricted dates but the period contains the
restricted period, expected outcome: 1
select
      count(id)
from
      bungalow_restrictions
where
      '2024-01-10' < end_date and '2024-02-15' > start_date;

-- search dates are completely inside and contained in the restricted
period, expected outcome: 1
select
      count(id)
from
      bungalow_restrictions
where
      '2024-02-02' < end_date and '2024-02-03' > start_date;

-- search dates are completely outside and not overlapping with the
restricted period, expected outcome: 0
select
      count(id)
from
      bungalow_restrictions
where
      '2024-02-07' < end_date and '2024-02-15' > start_date;
```

v1.0.112

Lecture 113 – Availability Check: Availability For A Specific Date Range For All Bungalows

Create a database function that checks the availability of a bungalow for a given time period.

The database queries in SQL used in the video are:

```
select
     b.id, b.bungalow_name
from
     bungalows b
     where b.id not in
     (select
          bungalow_id
     from
          bungalow_restrictions br
     where '2024-02-01' < br.end_date and '2024-02-04' > br.start_date);
```

v1.0.113

Lecture 114 – Delicate Ties: Creating Connections Between Database Functions And Handlers

Call your database functions in the handlers and evaluate the information.

Lecture 115 – What Can It Be? Connection Of The Availability Check To The Reservation Page

Establish a connection from the availability check to the reservation.

Lecture 116 – Mission Accomplished: We Successfully Make A Reservation!

Learn how you need to customize the handlers to create a reservation.

v1.0.116

Lecture 117 – Aftermath: Finalize Overview Page, Restrict Date Selection, Debugging

Revise the display of an overview page.

v1.0.117

Lecture 118 – Migrations VII - Preventing "Horsing Around" With Database Entries

Create migrations for different tables with `soda generate`. Use export as SQL statements of already existing data from DBeaver.

soda generate sql SeedBungalowsTable

DBeaver, double-click intended table, choose data tab, select-all-rows, open context menu with right-click, „advanced copy“ → „copy as SQL“

```
UP:
[Paste from Clipboard]
DOWN:
delete from bungalows;
```

soda generate sql SeedRestrictionsTable

DBeaver, double-click intended table, choose data tab, select-all-rows, open context menu with right-click, „advanced copy“ → „copy as SQL“

```
UP:
[Paste from Clipboard]
DOWN:
delete from restrictions;
```

soda generate sql SeedDevReservationsTable

DBeaver, double-click intended table, choose data tab, select-all-rows, open context menu with right-click, „advanced copy“ → „copy as SQL“

```
UP:
[Paste from Clipboard]
DOWN:
delete from reservations;
```

soda generate sql SeedDevBungalowRestrictionsTable

DBeaver, double-click intended table, choose data tab, select-all-rows, open context menu with right-click, „advanced copy“ → „copy as SQL“

```
UP:
[Paste from Clipboard]

DOWN:
delete from bungalow_restrictions;
```

v1.0.118

Lecture 119 – JavaScript On A Date With JSON: Availability Check And A JSON-Processing Handler

Evaluate the data in JSON in your handler generated by your JavaScript.

v1.0.119

Lecture 120 – Displaying the Result Of The Bungalow Availability Query To The User

Added a function call for a result windows and created the correct link for a GET-request with all necessary information to create a reservation.

v1.0.120

Lecture 121 – Session Creation: A Connector Between Availability Check And Reservation

Use sessions by passing reservation data from the availability check to the reservation form.

v1.0.121

Lecture 122 – Data transfer: Copy JavaScript Into Templates, Idea For Code Abstraction

Populated the other holiday home page templates with JavaScript and reorganized JavaScript out of the base layout into a local folder and import from there.

v1.0.122

Section 13 – Checkup: Updating Tests To Keep Your Code Fit And Healthy

> *When debugging, novices insert corrective code; experts remove defective code.*
>
> *Richard E. Pattis*

Lecture 123 – No Database For Your Test Setup? Fake One!

To be able to use the nested repository pattern for the database repository in `test_setup.go` for package `handlers`, create a fake database type, to which you currently add empty database methods.

v1.0.123

Lecture 124 – Repairing The Tests For The Handlers - Reservation In Sessions As Context

Learn to rewrite `test_setup.go` and `handlers_test.go` in package `handlers`. Introduction of a request recorder to make use of session data in tests and have an instance 'faking' a client receiving responses. Basically a full request/response-cycle is simulated in tests for individually customized for each handler.

v1.0.124

Lecture 125 – Improving Test Coverage And Multiple Test Cases For GET-Request Handler

Learn how to increase the coverage of the code under test and implement more test cases.

Lecture 126 – An Example How To Write Tests For POST-Request Handlers

Learn how to test a POST-request handler with an example.

v1.0.126

Lecture 127 – Special Case: Testing POST-Request Handler ReservationJSON

Learn why the POST-request handler `ReservationJSON` needs special testing.

v1.0.127

Lecture 128 – Brief Look At The Rest Of The POST-Request Handlers Tests if you please

Overview of the adjustments made to the POST-request handler tests.

v1.0.128

Lecture 129 – Exchange and Type Change: reqBody Becomes postedData Of Type url.Values{}

Learn how to swap `reqBody` (of type `string`) for `postedData` of type `url.Values{}` and how to create your request body faster using the prebuilt methods.

v1.0.129

Lecture 130 – Houston, We Have A Problem! Emergency Debugging On The Fly!

Learn how to eliminate a bug as soon as you discover it. Addressing a problem early on helps avoid lengthy and difficult troubleshooting and fixes in the future.

v1.0.130

Section 14 – The Postman Always Rings Twice: Integration Of E-Mail Into The Web Application

> *I do love email. Wherever possible I try to communicate asynchronously. I'm really good at email.*
>
> *Elon Musk*

Lecture 131 – What Was That Again? How E-Mail And The SMTP Protocol Work…

Recall how SMTP ensures that email is delivered to the recipient.

How email works

About how SMTP works

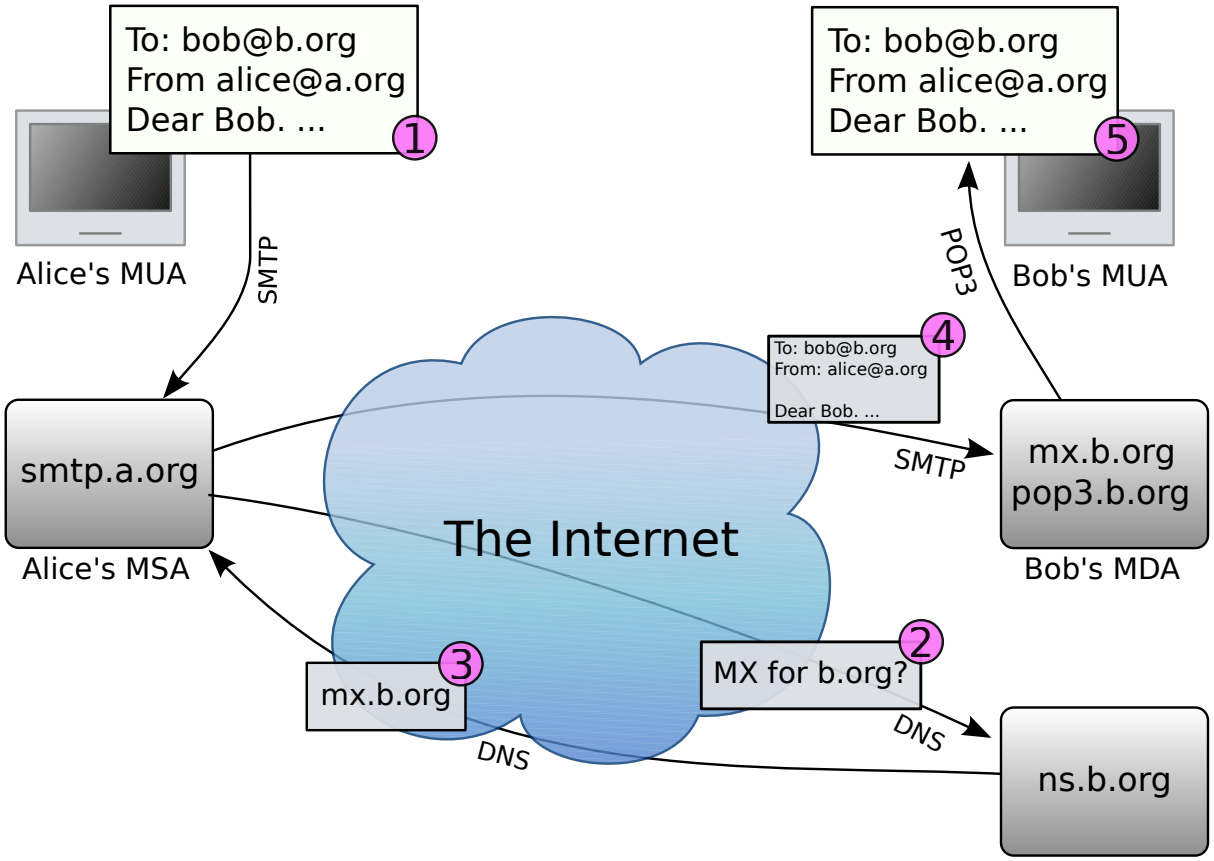


Figure 2: File:Email.svg. (2021, January 6). Wikimedia Commons. Retrieved 03:18, August 26, 2023 from https://commons.wikimedia.org/w/index.php?title=File:Email.svg&oldid=524450166.

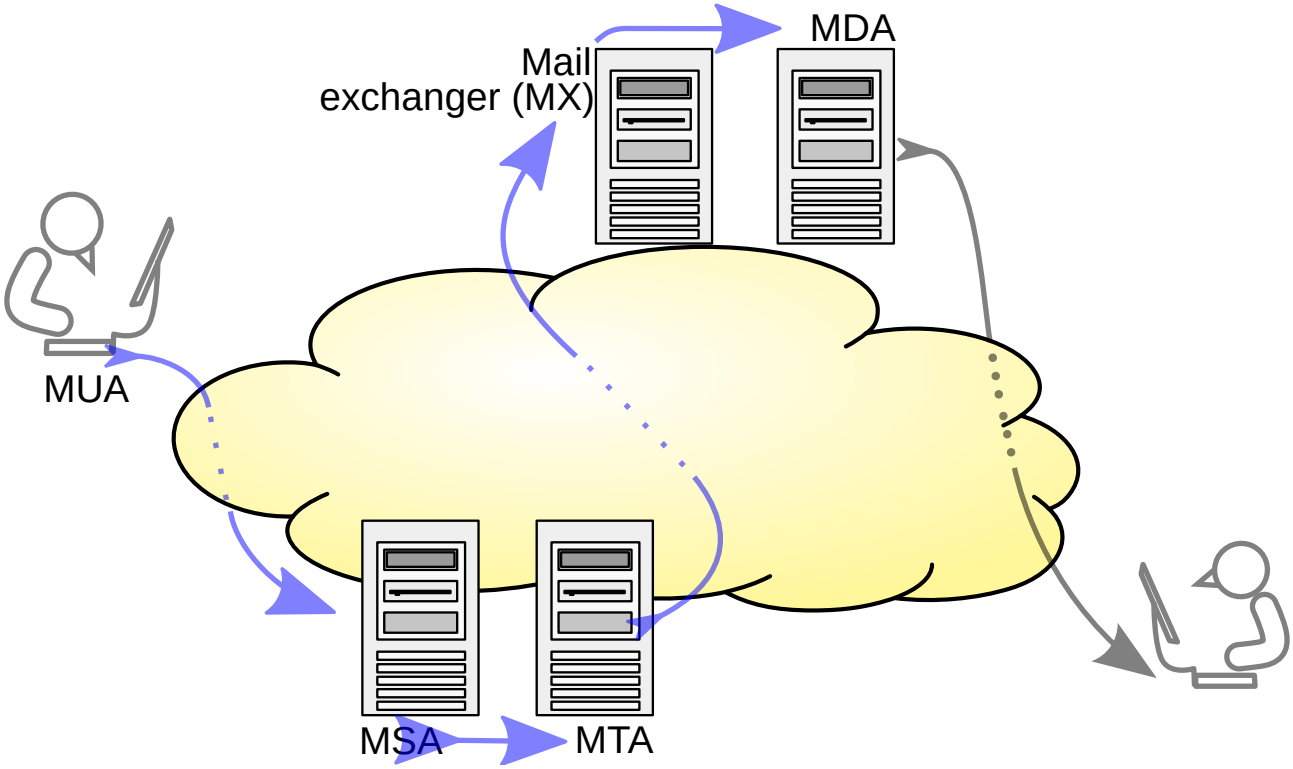


Figure 3: File:SMTP-transfer-model.svg. (2022, May 1). Wikimedia Commons. Retrieved 06:10, August 25, 2023 from https://commons.wikimedia.org/w/index.php?title=File:SMTP-transfer-model.svg&oldid=653069184.

Lecture 132 – MailHog Installation For Testing Purposes: GO The Whole Hog!

The installation of MailHog explained for several operation systems.

Windows

Download **binary** (32 or 64 Bit) and start it with a double click. To exit, close the command line or PowerShell window.

macOS

Download **binary** (Darwin/64 Bit), exec.

Alternative:

Install Homebrew with

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh
)"
```

and in the terminal execute

```
brew update && brew install mailhog
```

Afterwards you can start and stop mailhog with

```
brew service start mailhog
```

resepctively

```
brew service stop mailhog
```

as a service in the background.

Linux

Download **binary** (386/32 Bit, amd/64 Bit or ARM), make executable and run in terminal. Stop with CTRL+C.

Alternative:

```
sudo apt-get -y install Golang-go
go install github.com/mailhog/MailHog@latest
```

Lecture 133 – Sending E-Mails With The Standard Library - Just For The Sake Of Completeness!

You can easily send e-mail with GO already using the standard library:

```
from := "rick@c-137.universe"
auth := smtp.PlainAuth("", from, "", "localhost")
err = smtp.SendMail("localhost:1025", auth, from,
[]string{"morty@lost-in-roy.game"}, []byte("Come here, or  *burp*
I buy a new Morty!"))
if err != nil {
     log.Fatal(err)
}
```

v1.0.133

Lecture 134 – GO Simple Mail: Open An Application-Wide Channel For Sending E-Mails

Learn how to run a concurrent Go program as a listener to a channel where you can drop a mail in from nearly everywhere within your web application.

Watch,

- how to integrate the GO Simple Mail package
- set up a channel that receives mails
- create a listener that waits for mail in parallel with the application
- send mails independently from the web application

v1.0.134

Lecture 135 – #MEGA - Make E-Mail Great Again! - Creating And Sending E-Mail Notifications

Learn how to send an email to the user from your web application.

v1.0.135

Lecture 136 – Stay Informed: A Solution To Send E-Mails To The Operator

Implement sending email to the operator/admin of your web application.

v1.0.136

Lecture 137 – ... Once Again With Feeling: Beautifully Formatted E-Mails With Foundations

Use "Foundations for Emails 2" to optimize the look of your email on any device.

Foundation for Emails 2

v1.0.137

Lecture 138 – Updating The Tests - Doesn't Help, It's Got To Be Done!

An overview of the tests so that your code remains traceable and verifiable.

v1.0.138

Section 15 – Prove It's You: Authenticate Your Identity And Access All The Goods!

> *The most common form of despair is not being who you are.*
>
> *Søren Kierkegaard*

Lecture 139 – Elevate Your App: Craft an Easy Login Screen!

Create a simple login.

v1.0.139

Lecture 140 – Navigate To Success: Crafting A Login Route And Handler

Create route and GET request handler for login.

v1.0.140

Lecture 141 – Unlocking Security: Building Authentication And DB Functions

Create all the necessary database functions to authenticate a user.

v1.0.141

Lecture 142 – After The Form: A Login Handler That Delivers!

Write the POST-request handler for the data that is submitted via the login form.

v1.0.142

Lecture 143 – Getting Middle-witty: Cooking Up Some Middleware Magic!

Implement your own little piece of middleware to authenticate users.

v1.0.143

Lecture 144 – Database Table For Awesome: Crafting A User With Migrations!

Create a database entry for an administrator using `migration`.

`soda generate sql SeedUsersTable`

`UP:`

```
INSERT INTO public.users
(full_name,email,password,role,created_at,updated_at) VALUES
('Patrick Star','patrick@bikini-
bottom.ocean','$2a$12$EfnbNqKehrvw7uUgswFR2OlHcE9XjmYaYH8x02JPXapK
2YFx5IX4i','1','2020-01-01 00:00:00.000','2020-01-01
00:00:00.000');
```

`DOWN:`

```
delete from users;
```

Create a hash value for a password

v1.0.144

Lecture 145 – Putting The Login Page To The Test: Success Awaits!

Test if the login page works as you expect it to.

v1.0.145

Lecture 146 – Unveiling Authenticated Users and Log Out in Style!

Create a way to distinguish between authenticated and unauthenticated users that is available to be evaluated in the templates.

v1.0.146

Lecture 147 – Fortify Your App: Building A Secure Admin Zone With The Middleware!

Create a protected area by using middleware to protect a route against unauthenticated access.

v1.0.147

Lecture 148 – Cleaning up Your Code: Smaller Cleaning Actions - Sweep Through Again Quickly!

Clean up your code regularly to avoid potential problems in the future.

v1.0.148

Section 16 – Home, Sweet Home: A Customized Backend For Easy Maintenance With Security

> *Home is a place you grow up wanting to leave, and grow old wanting to get back to.*
>
> *John Ed Pearce*

Lecture 149 – Creating An Admin Dashboard In A Ready-Made Way - Choosing A Template

Learn how to quickly and efficiently create an administration area using a third-party package.

You can find overworked `admin-layout.tpml` and `admin-dashboard-page.tpml` files in the resources of this lecture on Udemy.

https://github.com/BootstrapDash/RoyalUI-Free-Bootstrap-Admin-Template

v1.0.149

Lecture 150 – Like On An Assembly Line: Bulk Creation Of Routes, Handlers And Templates

Create all routes, handlers and templates for already planned functionalities in one go.

v1.0.150

Lecture 151 – Displaying All Reservations: Where DB Records Get A Seat At The Stylish Table

You will learn how to implement a database function to read out all existing reservations.

Simple DataTables

v1.0.151

Lecture 152 – A Copy & Paste Orgy: Creating A List With Only New Reservations

Learn how to use already created source code to quickly and efficiently implement another functionality.

v1.0.152

Lecture 153 – Interlude: Makeover And Iron Out Small Mistakes

Face small errors and negligence early and ensure sufficient entries in the database during the development phase.

v1.0.153

Lecture 154 – Displaying A Single Reservation: Preparation For More To Come

Learn how to implement database functions to store or delete reservation data.

v1.0.154

Lecture 155 – New Possibilities: Creating the Database Access Functions

Create more database functions.

v1.0.155

Lecture 156 – Very Concrete: Implementation Of The Editing Function

Create the POST-request handler to process the form containing the reservation to be processed.

v1.0.156

Lecture 157 – Shift Up A Gear: Change The Status Of A Reservation

Create a Set-To-Processed button with confirmation prompt to change the status of a reservation.

v1.0.157

Lecture 158 – Delete a Reservation: Is This Art Or Can It GO Away?

Create a Delete button with confirmation prompt to permanently remove a reservation from the database.

v1.0.158

Lecture 159 – Reservation Calendar I: Heading and Navigation

Create a heading with year and month and build a small navigation to scroll back and forth month by month.

v1.0.159

Lecture 160 – Reservation Calendar II: Bungalows, Days and Checkboxes

Build a simple table to display the days of a month for each bungalow and use check boxes as placeholders.

v1.0.160

Lecture 161 – Reservation Calendar III: Reservations and Blocked Days

Add existing reservations as well as checkboxes for free and blocked days to your calendar.

v1.0.161

Lecture 162 – Reservation Calendar IV: Render That! Display The Calendar!

Teach the package `render` to display the calendar.

v1.0.162

Lecture 163 – Reservation Processing I: POST-Request, Route, And Handler

Create the POST-request handler and the associated route and handler.

v1.0.163

Lecture 164 – Reservation Processing II: Correct Return After Processing

You must ensure that the POST-request handler returns to the correct page after processing a reservation.

v1.0.164

Lecture 165 – Reservation Editing III: Handlers To Perform Actions

Learn how the two handlers work to manage blocked days in the reservation calendar.

v1.0.165

Lecture 166 – Reservation Processing IV: Database Functions for Actions

You write the two database functions to block a tag for a bungalow in the database and to unblock a blocked tag.

v1.0.166

Lecture 167 – Quo Vadis? Correction Of Redirects After Editing

You will learn how to adjust the redirects, routing and URL parameters so that the user is correctly redirected back from processing a reservation.

v1.0.167

Lecture 168 – Making the Handler Tests Run Again and a Few Tests

Get an overview of what needs to be done to get your `handler` tests working again.

v1.0.168

Section 17 – Going Live: Deploying Your Web Application To A Server On The Internet!

> *Tame birds sing of freedom. Wild birds fly.*
> *attributed to John Lennon*

Lecture 169 – Launch Your Web Application Flexibly: Use Command Line Flags

You will learn how to pass your parameters as arguments to command line flags and thus flexibly start your web application. You will use the `flag` package from the standard library.

Cobra - A Framework for Modern CLI Apps in GO

Understanding Golang Command Line Arguments

v1.0.169

Lecture 170 – Note On Using .env Files For Your Web Application

Learn how you can also use .env files to define your environment with a third-party package.

Use a .env File with GO

Lecture 171 – Text Editors Nano And Vi/Vim: Short Operating Remarks

On a Linux server, you probably only have text-based editors like `nano` and `vi/vim` available via a terminal/shell. You might be able to use some operating instructions.

Nano Editor Cheat Sheet

Vi Editor Cheat Sheet

Lecture 172 – Get Server And Set Up The Necessary Server-Side Software

This lesson will show you ways to rent a server and set up Linux as an operating system with a few tools. The server software `Caddy2` is also part of it.

Service Provider

- Digital Ocean

- Vultr
- Linode
- Azure
- AlibabaCloud (intl) / Aliyun (China)
- Amazon AWS

Server Software

- Nginx
- Apache
- Lighttpd
- Caddy
- Caddy Install (Binaries/Packages/Docker Image)

Lecture 173 – Install GO And Get The Web Application On The Server

You install GO on a Linux operating system and use git/github.com or a simple download to get your source code onto your server.

Lecture 174 – No Mail Transfer Agent (MTA) Available? Fake It Till You Make It!

Without a Mail Transfer Agent (MTA) on the target system, your application keeps crashing. To make things work at all, install `postfix` as a transitional measure.

Lecture 175 – Supervisor - Someone Has To Watch While You're Away

Use the `supervisor` application (under Linux) to start, monitor and if necessary restart processes.

Lecture 176 – Logo, Footer Content: Final Touches Before The Curtain Rises!

You will learn how to easily include a logo, fill the footer and some pages with content.

Section 18 – Farewell & How It Could Go On From Here (And Room For Bugfixes)

> *Education is the most powerful weapon which you can use to change the world.*
>
> *Nelson Mandela*

Lecture 177 – Goodbye And How It Could GO On With Your Web Application

Time to say goodbye - you will get a little inspiration on how to improve your application.

Latest Release