Backend Development Diluted

Abdul Samad Shaikh

Contents

Chapter 1

Introduction

Backend Development Diluted is a book that teaches you about the fundamentals of backend development using Golang.

This book assumes no prior knowledge of backend development but requires that you have some programming experience.

It consists of two parts: The first part contains fundamental concepts that you need to know as a backend developer, and the second part contains a small e-commerce backend application that we will be building with Go alongwith a short introduction to Go itself. I will walk you through every part of the project in the third part, explaining things along the way, but you need to follow along and type with me in order to get the most value out of this book. The code for part II will be in the `src/code/` directory of the repository `https://github.com/schmeekygeek/backend-book`. By the end of it all, you'll have many concepts pertaining to backend development understood, and a shiny new project to show off in front of your friends! ***(Heck yeah!)***.

I have tried my best to not include superfluous details about everything, much of which will be easy to understand with just a little research on your own. This book aims to take away the overwhelm that many beginners face and give you a headstart on your journey.

1.1 Who this book is for

This book is for anyone who is willing to learn and get into backend development. People switching careers may also find this book a great starting guide to get into the field.

With concepts explained such as HTTP, JSON, authentication and authorisation, CRUD and others, it will give you a well-rounded introduction to backend development along with best practices and tips to build solid, performant and secure backend applications in Go.

If you're a backend developer already, this book can also be used to brush up on some concepts, say, before an interview, as well.

Part I

Fundamentals

Chapter 2

Introduction to the Backend

Consider this analogy: You go to a restaurant, the waiter greets you, then sits you at a table. He then brings you a menu for you to decide what you want to eat. You spend some time dilly-dallying, overthinking before finally deciding to go with something basic as noodles. The waiter gives you a look of disappointment, writes down your order, then disappears.

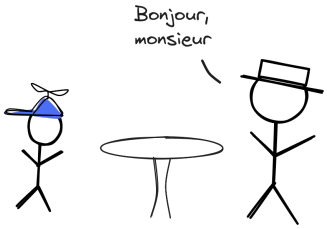

Figure 2.1: Sam visits a restaurant

He returns after some time to give you the ever-saddening news that the restaurant is out of noodles and asks if you would like some rice instead. You agree, and he serves you rice a few minutes later.

What just happened? To put it bluntly, the customer is the user. The

waiter is the website that the user is going to interact with, and the kitchen is the server that the waiter will make requests to. When the server receives the request of noodles, it checks the refrigerator to see if it has the stock for said noodles or not. Here the refrigerator is the database, where all the ingredients of the kitchen are stored. This is obviously an oversimplification, but you get the idea.

Backend development is the server-side development of a web application. It mostly involves writing code that is used by other applications and services; Mainly business logic.

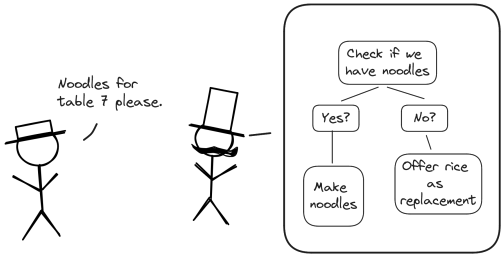

Figure 2.2: Business Logic

2.1 What is an API?

The term API stands for Application Programming Interface. It is essentially what we've been calling a backend application. You can write an API using any popular backend framework or programming language meant to run on the server. We will learn how to build one in the 3rd part of this book using Go.

2.2 Why do we need an API?

When learning about backend development, it is important to understand why we need to have a backend in our application in the first place in order to truly understand its essence and the problems it solves. You might be thinking why the client-side application can't communicate with the database directly. Well, it can, but it **shouldn't**.

Communicating with the database directly is simply not optimal, as we

need to have a central point of data-access that would take a request, know if the request is coming from a person that has the authority to make that request, read some data, run some code, perform some theatrics on that data, read some more data, write some logs to disk, perform other actions to then finally wrap up everything neatly in a response-body to be sent back to the client.

In the next chapter we will understand how this communication between two machines over the internet takes place.

Chapter 3

Communication Protocols

Consider this scenario: You see an old woman at the supermarket that is having trouble carrying her basket. So you offer to help, but to your surprise, she doesn't speak the same language as you! How would you ask her if she needs any help?

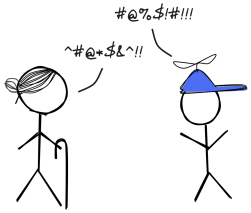

Figure 3.1: Sam has trouble talking to an old woman

In order to verbally communicate with a person effectively, it is necessary that they speak the same language as you. Pretty much the same goes for computers trying to communicate with each other over the internet. If one speaks a language the other doesn't, they won't be able to exchange information at all.

Basically, these 'languages' are called communication protocols. Communication protocols determine who speaks, what is being said, what can be said, how long it should take, etc. Computers over the internet commu-

nicate over a particular protocol. They are a pre-defined set of rules that needs to be followed in order for information to be exchanged successfully.

Let's look at one of the most fundamental protocols of the internet, or, the **Internet Protocol** suite, a set of communication protocols, the transmission control protocol.

3.1 Transmission Control Protocol (TCP)

Transmission Control Protocol, or simply TCP, is a highly-reliable communication protocol which is connection oriented. Many protocols are based on it, such as HTTP, FTP, Websockets, etc. It needs a stateful handshake to occur between two machines before packets of data can be exchanged. Let's learn how a TCP connection is made between two hosts.

3.1.1 The 3-Way Handshake

Two hosts over a network may begin communicating over TCP after they've made a successful handshake. This handshake occurs in 3 steps. Which is why it is called the 3-way handshake.

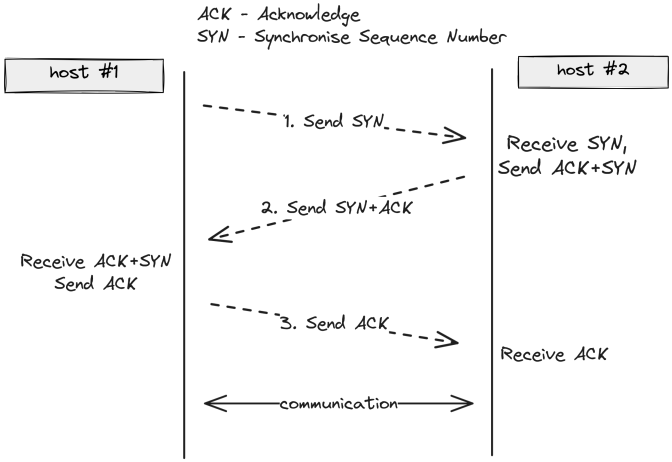

Figure 3.2: The 3-Way Handshake

Step 1: Host #1 sends a message with the SYN (Synchronize Sequence Number) flag set.

Step 2: Host #2 responds with its own message with the SYN flag set, along with an ACK (acknowledgement) that the SYN of the client was received successfully.

Step 3: Host #1 receives the SYN+ACK, then responds back with another ACK.

3.1.2 What makes TCP so reliable?

Every single time a data packet is sent over TCP, an acknowledgement message is sent by the receiver to the sender indicating that the message sent was received successfully. If the sender doesn't get the acknowledgement message after a set period of time, it simply sends the data packet again. This makes TCP highly reliable in environments where data is critical and cannot be lost. There may be instances of TCP not being reliable, but they are exceptionally rare.

Now, for our next protocol based on TCP itself, HTTP.

3.2 Hyper Text Transfer Protocol (HTTP)

HTTP stands for Hyper Text Transfer Protocol. It is the foundational protocol for all types of data exchange on the web. Since HTTP connections are based on the request-response model of the internet, they are unidirectional. Which is why an HTTP connection, or an HTTP request, is always made by the client that is requesting a resource from the server. The server will then respond back with the requested resource, depending on whether the client is supposed to see it or not, in which case it will reject the request for said resource.

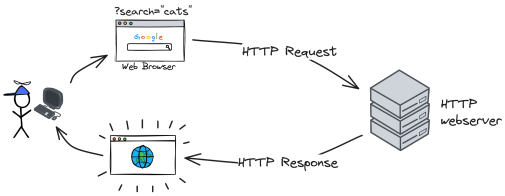

Figure 3.3: Request-Response model

There exists many other protocols such as FTP and MTP, but they are beyond the scope of what's required in this book.

3.2.1 URL and URI

You have most likely seen this long string of *mumbo jumbo*. Let's break it down to try and understand what these frightening symbols and characters actually mean.

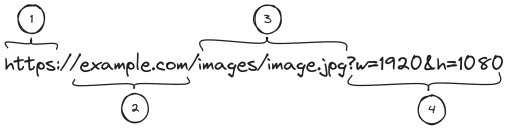

Figure 3.4: A typical URL

1. **Protocol:** The protocol for this URL
2. **Host:** This is the host, or the server that we are making the request to.
3. **Path:** The path, or the location of the resource on the host. Here, `/images` is the path, and `image.jpg` is the resource we're requesting.
4. **Query Parameters:** These are query parameters. Their purpose is to modify variables that the endpoint might expect to fine-tune our request. They begin with a *?* (Question mark). If there are two or more such values, they are separated by an '*&*' (Amper's And).

What we just saw above was an example of a URL. **A URL stands for Uniform Resource Locator**. It *locates* the resource that we're trying to fetch from the host. Another term similar to URL is URI that stands for **Uniform Resource Identifier**. It is an identifier, or a name, for any resource that we're trying to request.

At first glance, it may seem that both of these terms mean the same thing: To identify our resource, which will also be used to locate it. A URI identifies, and a URL locates. But a URL is also a URI, since it is also used to identify. But not every URI is a URL, since we usually have no idea how to locate the identifier.

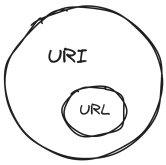

Figure 3.5: URL-URI Venn diagram

In other words, **A URL is a subset of a URI.**

3.2.2 Anatomy of an HTTP request

An HTTP request sent by a client contains many parts that define its structure. Let's try and understand them.

Here is a response that is received upon calling the API *https://httpbin.org/get* using `curl`, a command-line tool used to transfer data from or to a server.

```
    1.    2.
  HTTP/2 200
  date: Sun, 21 Apr 2024 15:23:18 GMT
  content-type: application/json
3. content-length: 257
  server: gunicorn/19.9.0
  access-control-allow-origin: *
  access-control-allow-credentials: true

  {
    "args": {},
    "headers": {
      "Accept": "*/*",
      "Host": "httpbin.org",
4.    "User-Agent": "curl/7.81.0",
      "X-Amzn-Trace-Id": "Root=1-66252f66-1137e90d2c48754965872743"
    },
    "origin": "103.204.132.228",
    "url": "https://httpbin.org/get"
  }
```

Figure 3.6: A basic HTTP request's response

1. **HTTP version:** The protocol version of HTTP being used.
2. **Status Code:** The status code or response code, sent by the server, indicating the status of the request sent.
3. **Headers:** This is the metadata.
4. **Payload:** The data itself. In our case, the format of the data is `json`

The type of request that we sent above was a `GET` request. `GET` is also called an HTTP verb. Let's look at some of these attributes of HTTP requests.

3.2.3 HTTP Verbs

HTTP verbs are request methods that let us specify certain actions to be performed with our HTTP requests. They essentially are a vague type of the request that we are sending. For example, if we are trying to fetch a resource, we will use the `GET` method in the request, and make our backend accept the same. If we are trying to send some kind of data, we may have to use the `POST` method. There are many HTTP verbs, each having their own significant meaning. However, there are four fundamental HTTP verbs that are most commonly used in backend development. They are:

1. **GET:** Get or fetch a certain resource from an API endpoint.
2. **POST:** Send some data to the server.
3. **PUT:** Update a resource on the server.
4. **DELETE:** Delete a resource on the server.

These HTTP verbs are also called as CRUD operations where CRUD stands for CREATE (POST), READ (GET), UPDATE (PUT) and DELETE. Some other HTTP verbs include, CONNECT, HEAD, OPTIONS, PATCH, etc.

3.2.4 HTTP Headers

HTTP headers let the client and servers share information additional to the payload when communicating over HTTP. They are also called metadata because they contain the data that is used to describe the payload and/or used to set other options and flags. Headers are set as key-value pairs separated by a ':' (colon).

There are 3 types of headers:

1. **Request headers:** Headers set by the client containing information about the client or more information about the resource being requested.
2. **Response headers:** Headers set by the server that contain details about the server or time, etc.
3. **Payload headers:** Client or server independent headers that provide details about the payload.

Here are some of the most common HTTP headers:

- `user-agent`: Client application or web browser identifier.
- `authorization`: Used to set any credential or bearer token that the server might need whenever accessing a protected resource. Usually a JWT (JSON Web Token).
- `content-type`: Indicates the media type of the resource.
- `content-length`: Length of the content (in bytes).
- `cookie`: Sent by the client to help the server recognize a session or a user.
- `accept`: Sent by the client to indicate media types that are acceptable.

3.2.5 HTTP Status Codes

When we interact with APIs, it is necessary for us to know what happened with the request that was sent. Has it completed successfully, did something go wrong, if yes, what went wrong? Is the request being processed, am I being redirected, etc., are many such questions that arise when a request is made. To address this problem, every HTTP response has a status code attached to it that determines if the request was a success or not. These status codes are also called response codes. They exist to inform the client about the status of the request. You have most likely heard about or seen the error: '404 - not found'. This is one of the most common HTTP status codes that informs the client that the resource you were looking for couldn't be found.

A status code is a three-digit number. The first digit signals the general status or success/failure of a response. The next two digits are more specific as to what exactly happened. Here is a range for the status codes.

Range	Message
100 - 199	Informational
200 - 299	Success or OK
300 - 399	Redirection
400 - 499	Client-side error
500 - 599	Server-side error

Here are some of the most common status codes used today:

Code	Description
200	*OK* - The request was successful.

Code	Description
201	*CREATED* - Created a new resource, such as an account.
400	*BAD REQUEST* - Invalid or malformed request
401	*UNAUTHORIZED* - Client is unauthorized, needs authentication.
403	*FORBIDDEN* - Client is forbidden to access the resource.
404	*NOT FOUND* - The resource/s requested couldn't be found
500	*INTERNAL SERVER ERROR* - Something went wrong at the server side

In the next chapter we shall learn about two basic security mechanisms that APIs implement in order to serve the right client.

Chapter 4

Authentication and Authorisation

When we write APIs, it is necessary to know if the client that we are serving, or going to serve, is someone who has the privilege of seeing or modifying any resource on our server. For this reason, we have security mechanisms, where, if someone makes a request, help us identify who we are communicating with and if they have the authority to be making that request.

4.1 Authentication

If you've ever tried to sign into a website, it asks for a few details, one of which is sure to be your password. This is done to make sure that it is indeed *you* that is trying to sign in and not an impersonator. This is pretty much what authentication is. The server is authenticating if it really is you.

Now, a server may have multiple layers and authentication checks before simply handing an account access to you. Let's go over them.

4.1.1 2FA (2-Factor Authentication)

2FA is a type of authentication, where, in addition to providing your password, you need another piece of evidence in order to prove ownership

of an account. You may set an email or phone number when you create your account, and every time you try to authenticate, the API will try and verify that it is really you using your email or phone number, either by sending you a one-time-password that you would need to enter, or an authentication link that you will need to go to.

Figure 4.1: Sam fails an authentication check

4.1.2 Multi-factor authentication

It is similar to 2FA, but you have to provide more than two pieces of evidence, such as answering security questions that you may have set when you created your account. A business may add as many layers as they desire in order to safeguard critical data.

4.2 Authorisation

After successfully authentication we still need to make sure if a user is allowed to perform a certain action or not. By definition, **authorisation is the process of assigning access rights and privileges to someone.** For example, in most countries, its civilians are not allowed to get on a military base without proper authorisation; They simply don't have the privilege to do so. In other words, they are unauthorised to perform the action of entering a military base.

4.2.1 Role-based authorisation

Role-based authorisation is the most basic method of authorisation. We assign a role to every user in our system, and depending on those roles, we permit or deny a particular request. For example, in an e-commerce application, we may have three roles: **User**, **Seller** and **Admin**.

A **User** would have the authority to browse and buy products, save them to their wishlists, share products, etc.

A **Seller** would have the authority of doing everything a **User** can do, with the added privileges of adding and removing products, etc.

And finally, the **Admin** could be someone who'd have the authority to do many other things such as remove products that are offensive, ban users, etc.

4.3 Authentication vs. Authorisation

Many a times, the terms authentication and authorisation are confused with each other and used interchangeably, but they're two very different terms.

Authentication is the process of determining if a person is the one he claims to be.

Authorisation is the process of determining if a person is allowed to perform a particular action or not.

We can ask ourselves these questions to help understand them better:

Authentication	*Q. Are you really X? Prove it.*
Authorisation	*Q. Are you allowed to do X?*

Chapter 5

Introduction to a Database

If you recall, in chapter I, we discussed briefly what a database is. By definition, **a database is an organized collection of structured information, or data, typically stored electronically in a computer system.**[1] Simply put, it is a place where we store records such as user details.

	Name	Age
1	Bill	14
2	Sam	11
3	Sarah	17

Figure 5.1: A basic database

[1] https://www.oracle.com/database/what-is-database/

5.1 What is a DBMS?

A Database Managment System is a software that allows for interaction with databases.

You might be wondering why we need such a special type of software to store data, when we could accomplish the same thing with tools like Microsoft Excel or Google Drive. Here are a few very basic reasons why tools like these fail compared to a typical database.

1. Not performant.
2. Not secure.
3. Inability to backup data.
4. Limited storage.
5. Not optimal for concurrent operations.

5.2 SQL

SQL stands for Structured Query Language. It is an advanced query language used to query database and perform actions such as read and retrieve data in certain ways, write, back up data, etc.

5.3 Relational vs Non-Relational Databases

5.4 ACID properties

Every independant operation performed on a database, or unit of work done, is called **a transaction**. A DBMS (Database Management System) is a software that helps you perform operations such as read and write on a database. A DBMS typically has a set of properties that ensures that any transaction performed will complete with its intended purpose.

ACID is an acronym that stands for **A**tomicity, **C**onsistency, **I**solation and **D**urability. These are the characteristics of a transaction performed on a database. Databases whose transactions possesses the above characteristics, are said to be *ACID* compliant.

Let's look at these characteristics one by one.

1. **Atomicity:** Guarantees that transactions are either completed or are discarded. If a transaction or operation begins but fails to conclude successfully, the transaction is rolled back and all changes are

discarded.

2. **Consistency:** It refers to the ability of the database to maintain data integrity constraints. So if the constraint of a field is that it should only contain text, writing an integer value to it will fail.
3. **Isolation:** Makes sure that all transactions that run concurrently on a unit of data are isolated from each other. Meaning that if we were to withdraw 100$ and 50$ from an account containing 300$, they both will run in isolation and the end result would be 150$ instead of 250$ or 200$.
4. **Durability:** It is the ability of the database to persist data safely, even in circumstances such as power outages that are usually unexpected and uncalled for.

Let us now understand what a database schema is.

Chapter 6

Basics of System Design

// system design introduction

Now that we've covered most basic backend development concepts, let us try and understand a little about how backend applications are architected. Writing efficient and fast code is important, but it is also necessary to architect software and have a proper high level design that puts everything together such that components act cohesively and are utilised effectively, and efficiently. Let's learn about this process of designing a good backend solution.

6.1 What is System Design?

System design is the study of the high level design of backend applications. In other words, it is the process of architecting software and putting components and pieces in the right places to be used in the best way possible.

// TODO: figure add system design diagram

6.2 Monolithic or 3-tier applications

An application that consists of the following three parts: The UI, the backend and the database, is known as a 3-tier application.

3-tier applications are the simplest example of a modern-day, full-stack

application. The user pushes a button on a mobile app, it triggers a request to the server, the server reads or writes from the database, depending on the request and returns a response back to the client. Such a form of application that only has a singular API or data access point and is centralized, is called a *Monolith* or a *Monolithic* application. The word 'monolith' means any structure carved out of a singular stone. In the context of programming, it is an application that does all the things such as managing users, writing logs, reading and writing to a database, etc.

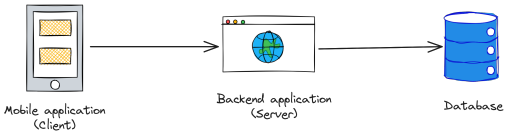

Figure 6.1: 3-tier application

At a glance, this may seem like a backend solution that is good enough and will suffice a lot of business needs, but there are quite a few problems associated with it. Firstly, since these are singular applications, they are not fault-tolerant. Which means that if our application were to go down, every user of the product would have no service. Secondly, it is difficult to scale these kinds of applications on demand. And lastly, they're not very flexible. Let's look at a different architectural approach that solves this problem.

6.3 Microservices

The microservice architecture aims to solve problems with the monolithic architecture by having multiple loosely coupled servers that run independently. Each server or node has its own purpose and multiple nodes communicate with each other via lightweight APIs.

For example, if you have an e-commerce application, you can have multiple services for each task: One for managing users, one for storing application-wide configurations, one for cropping and resizing images, interacting with a database and others. An API gateway sits between these services and will redirect requests to them accordingly.

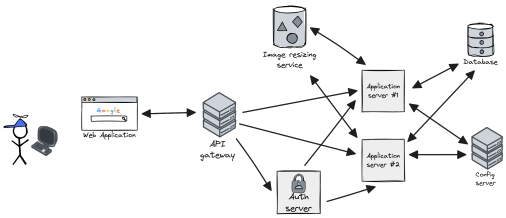

Figure 6.2: Microservice architecture

One of the biggest advantages of the microservice architecture is flexibility. If you have a change to make, you can simply do it in one of the services, instead of having to change it in the entire application and then stopping and starting everything. It also means that your application will be fault-tolerant, meaning if something goes wrong and one of the servers go down, you will still continue to serve users. Even though some of them will be affected by it, it is still better than not being able to serve users at all.

The microservice architecture, if implemented correctly, is a great way to solve the problems with the monolithic architecture. But implementing the microservices architecture without an appropriate reason will lead to unnecessary complexity and maintenance costs.

We briefly talked about how monolithic applications are difficult to scale in Section 6.3. Let's understand what this means.

6.4 Scaling

In the practical world, scaling is the process of increasing supply to meet growing demands. In the context of programming and backend development, it is generally used to describe the process of increasing computational power such that demands like, a growing userbase, are met. The ability of a backend application to scale easily on demand is called its scalability. Scalability is also an important aspect to consider when designing backend solutions. Now, there are two types of scaling, **Vertical** and **Horizontal** scaling, with each of them having their own quirks, advantages and disadvantages. Let's go over them.

6.4.1 Vertical scaling

Vertical scaling is the process of increasing or upgrading a single machine's or hosts' computing capability and strength to meet the increasing demands. If your machine, having 2 cores and a gigabyte of RAM, is struggling to serve, say, 10 users at the same time, you might want to consider adding another stick of RAM to the machine to make it more capable than it previously was. Vertical scaling is simple, but it can also get expensive as you keep climbing up the ladder.

Pros:

- Simple to perform

Cons: - Limit to how much you can upgrade

// TODO: figure: sam going to the gym

6.4.2 Horizontal scaling

// TODO: figure: many workers

Part II

Basics of Go

Chapter 7

Basics of Go I

In this part, we'll start with learning the basics of the Go programming language. We won't delve into every little detail there is about programming in Go, but we will cover those that are crucial going forward in this book.

Go is a statically typed, compiled high-level programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson.[1] Attributes such as high performance, low memory usage, easy-to-use concurrency model, simplicity and readability, makes Go one of the best choices for backend applications.

There are many guides online for installing Go, so we won't be covering that here. You simply need an editor such as Vim or VSCode, along with a terminal to run your Go files/project from. After successful installation, you should be able to run Go commands in your terminal like so.

```
$ go version
go version go1.21.5 linux/amd64
```

Let's write our first Hello World application using Go next.

7.1 Hello World in Go

Type and save the following code in a file called `main.go`, or any filename of your choice.

[1] https://en.wikipedia.org/wiki/Go_(programming_language)

```go
// main.go
package main // 1.

import "fmt" // 2.

func main() { // 3.
    fmt.Println("Hello, World!") // 4.
}
```

And run it like so:

```
$ go run main.go
Hello, World!
```

Congratulations, you just wrote your first program in Go!

Let's break it down, piece by piece.

```go
1. package main
```

The name of the package the file is in. When dealing with a single Go source file or when the file in the root of the project, the package name will be `main`.

```go
2. import "fmt"
```

This is the import statement, we use it to import packages from our project and other Go modules. When importing more than one package we surround it with parentheses like so:

```go
import (
    "fmt"
    "math"
)
```

```go
3. func main() {...}
```

This is the entry point of your Go application.

```go
4. fmt.Println("...")
```

We call `Println` from the `fmt` package to write 'Hello World' to the `stdout`.

Let's learn about variables in Go next.

7.2 Variables

Variables help store data in our programs. They have a name and a value.

> **Note**: In Go, having unused variables in your code, unlike in other programming languages, is disallowed. Your code simply won't compile if you have a variable that hasn't been used somewhere.

Let's go over the ways to create variables in Go.

7.2.1 Declare first, assign later

You declare a variable in Go by using the following syntax:

```
var <identifier> <type>
```

We make use of the `var` keyword to create variables. If you declare a variable and don't assign a value to it, Go will instantiate it with the default value of its datatype. For example, an unassigned `string` variable will be an empty string, and an unassigned numeric variable such as `int` will be 0. If you have a struct(more on this later) variable, which is simply a way of compositing multiple types, (similar to classes in Java), Go will recurse into the struct until it finds the basic types such as `int` and `string`, and will assign default values to it.

Example:

```
package main

import "fmt"

func main() {
    var name string
    name = "Josh"
    fmt.Println(name)
}
```

7.2.2 Declare and Assign

We can declare as well as assign variables like so:

```
var age int = 32
```

This looks a bit verbose, doesn't it? To solve this problem, Go can also

infer types from variables that are declared and assigned without a type as follows:

```
var age = 12 // the type is int
```

There is an even better way to accomplish this, by making use of a special assignment operator.

7.2.3 The := operator

The := operator enables us to assign to and declare variables simultaneously.

```
hobby := "Playing video games"
```

Note that when creating variables using this operator, we don't have to use the var keyword. The type will be inferred from the value assigned. When using this operator, the left hand side of the assignment must be a new variable.

```
var age = 31
age := 43 // This fails as age is already defined
```

7.2.4 Declare multiple variables

We can declare multiple variables of the same type in Go using the following syntax.

```
var <identifier>, <identifier>, ... <type>
```

Example:

```
var height, weight, chest int
```

You can assign values too when declaring multiple variables. The number of values on the right hand side must be the same as the number of identifiers on left hand side, though.

Example:

```
var foo, bar, baz = 1, "hello", true
```

Using the := operator:

```
foo, bar, baz := 1, "hello", true
```

7.2.5 Custom types

In Go, it is possible to declare custom datatypes that are based on inbuilt datatypes internally by using the `type` keyword.

Syntax:

```
type <identifier> <datatype>
```

Example:

```
package main

import "fmt"

func main() {
    type Number int
    var age Number = 24
    fmt.Println(age)
} // prints 24
```

Let's learn how to write conditional statements next.

7.3 Conditional statements

7.3.1 If-Else statements

The syntax to write an `if` statement is very similar to other programming languages:

```
if <condition> {
    // todo
} else if <condition> {
    // todo
} else {
    // todo
}
```

Let's look at this example of a program that checks if a person is eligible to vote or not:

```
age := 18
if age ≥ 18 {
    fmt.Println("Eligible")
} else {
```

```
    fmt.Println("Not Eligible")
}
```

You may or may not surround the condition with parentheses. You can also nest multiple `if-else` statements inside each other.

> **Note**: It is necessary to know that the `else if` or `else` keywords must start right after the last block's curly brace ends. This won't compile:

```
if <condition> {

}
else if <condition> {

}
```

Go also provides a special option to initialize a variable before the condition as follows:

```
age := 13
if age = 18; age ≥ 18 {
    fmt.Println("Eligible") // Prints Eligible
}
```

> **Note**: The value assigned to the variable will persist even after the scope of the conditional ends.

Go also supports logical operators such as = (is equal to), and `&&` (logical and).

7.3.2 Switch statements

Here's the syntax of a switch statement in Go:

```
switch <variable> {
case <value>:
    // todo
default:
    // todo
}
```

You can have as many cases as you want.

Example:

```
num := 2
switch num {
case 1:
    fmt.Println("One")
case 2:
    fmt.Println("Two")
default:
    fmt.Println("Not one or two")
}
```

7.4 Functions

You define functions in Go using the `func` keyword.

Syntax:

```
func <function-name>(<identifier1> <type1>, ...) <return-type> {
    // do something
    return <value>
}
```

Note that the datatype comes after the variable name or identifier. You may or may not have a return type to your function. But if you do have one, you must return an appropriate value pertaining to that return type. You also don't need to write the datatype for multiple parameters that have the same datatype. You can just write the parameter names by separating them with commas and adding the datatype at the end. For example:

```
func sum(x, y int) int {
    return x + y
}
```

7.4.1 Passing n parameters

If you're uncertain of the number of the parameters that the function must take, you can use the `...` operator. The resultant value of the passed parameter will be an array that you can iterate over.

Example:

```
func getSum(nums ...int) int {
  sum := 0
  for _, val := range nums {
    sum += val
  }
  return sum
}
```

Note: We will learn about `range` and arrays shortly.

7.4.2 Anonymous functions

Anonymous functions are also called inline functions or lambda functions in some programming languages. They're useful if you're trying to create a short function body and assign it to a variable to perform some small redundant tasks.

Syntax:

```
<identifier> := func (<parameters>) <return-type> {
  return <val>
}
```

You don't need to have a function identifier or name because the variable name that you assign the function to will be used to invoke it.

Example:

```
func main() {

    sum := func (x, y int) int {
      return x + y
    }

    fmt.Println(sum(7, 6)) // prints 13
    fmt.Println(sum(1, 1)) // prints 2
}
```

Part III

Project WeCart