Backend Development for Beginners (using Golang)

Abdul Samad Shaikh

Contents

Chapter 1

Introduction

Backend Development for Beginners (using Golang) 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 three parts: The first part contains fundamental concepts that you need to know as a backend developer, the second part is a short introduction to the Go programming language. And the third part contains a small e-commerce backend application that we will be building with Go. 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. 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 overwhelmingness that many beginners face and provide you a headstart in 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.

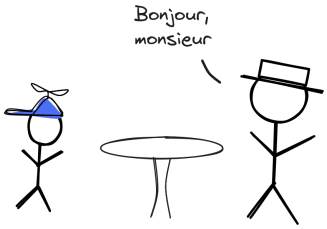

Figure 2.1: Sam visits a restaurant

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.

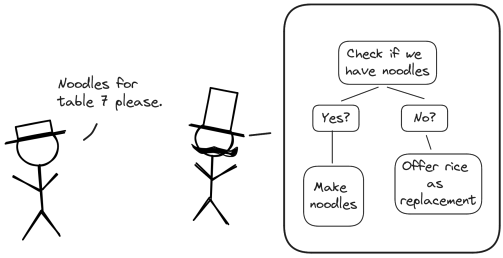

Figure 2.2: Business Logic

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?

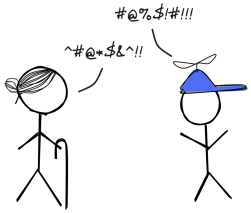

Figure 3.1: Sam has trouble talking to an old woman

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.

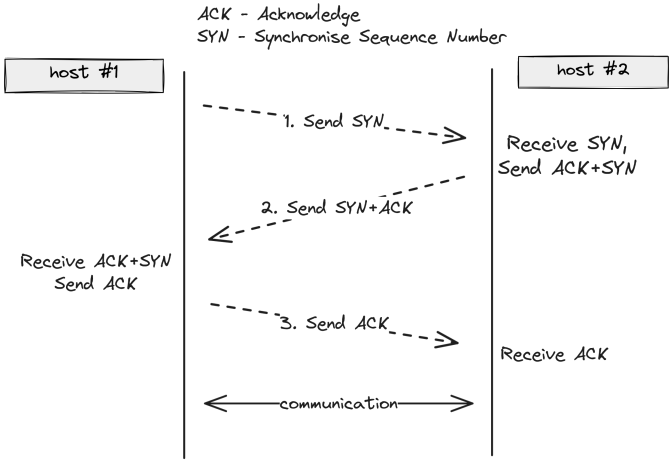

Figure 3.2: The 3-Way Handshake

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.

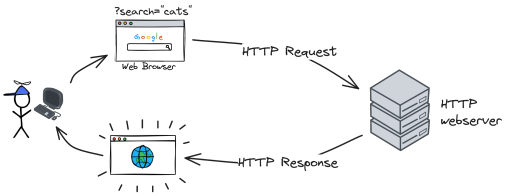

Figure 3.3: Request-Response model

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.

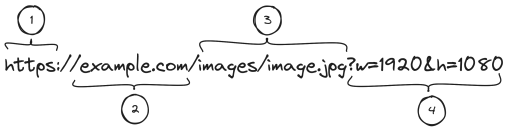

Figure 3.4: A typical URL

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 endpoint, 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.

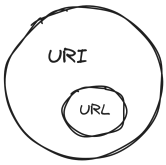

Figure 3.5: URL-URI Venn diagram

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

3.2.2 Anatomy of an HTTP request

An HTTP request contains many parts that define

3.2.3 HTTP Headers

3.2.4 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.
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

Backend Architecture

5.1 Architectural styles

5.1.1 REST

5.1.2 GraphQL

5.1.3 SOAP

5.1.4 Hateos

Now that we've covered most basic backend development concepts, let us try and understand a little about how backend applications are architectured. Writing efficient and fast code is important, but it is also necessary to have a proper high level design that puts everything together such that components are cohesive and utilised effectively.

5.2 3-tier or monolithic 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, 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.

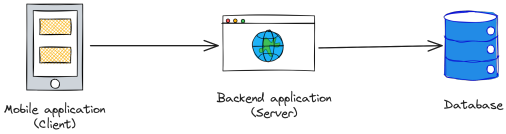

Figure 5.1: 3-tier application

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 the server 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. You can scale vertically, but that can only help you for so long. And lastly, they're not very flexible. Let's look at a different architectural approach to solves this problem.

5.3 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 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.

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 restarting everything. It also means that your application will be

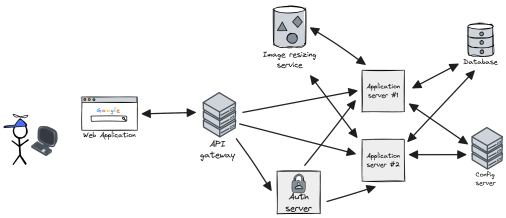

Figure 5.2: Microservice architecture

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 doing so incorrectly will lead to unnecessary complexity and maintenance costs.