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. The second and final part will involve building the server for a simple chat application. I will walk you through every part of the project in the second 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 Motivation

// TODO: add motivation

1.2 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.

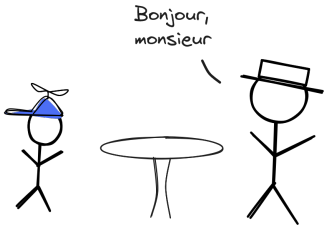

Figure 2.1: Sam visits a restaurant

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.

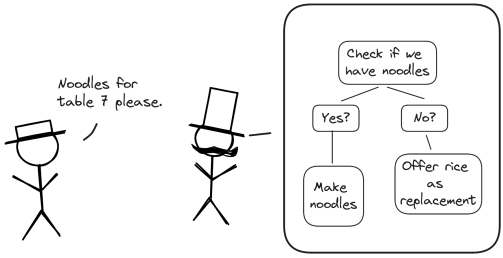

Figure 2.2: Business Logic

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.

When building your application, it is highly recommended you separate your data-access layer and your application layer for a variety of reasons:

- Scaling: If you decide to scale your application to meet growing demands of customers, you would simply need to add more servers running the same application code which won't need change.
- Security: The mobile application code of your product can be decompiled and your database credentials can be placed in the wrong hands.
- Flexibility: If you decide to migrate your database platform from SQL to MongoDB, for example, your client application code won't need to change as it only needs to know how to communicate to the API.

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?

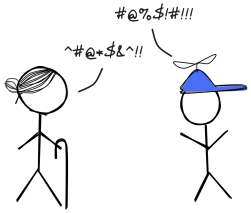

Figure 3.1: Sam has trouble talking to an old woman

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 need 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.

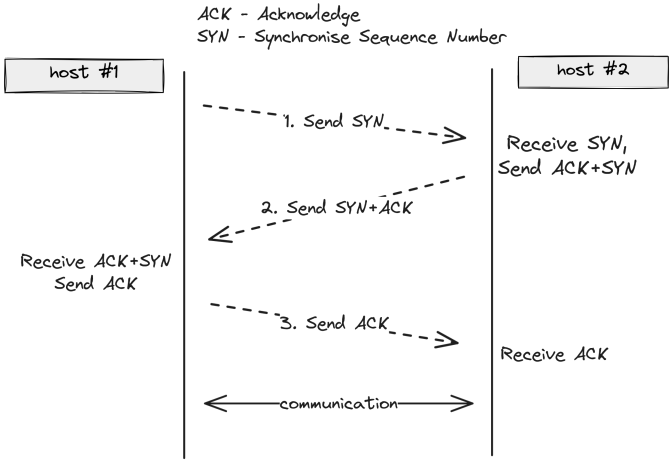

Figure 3.2: The 3-Way Handshake

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.

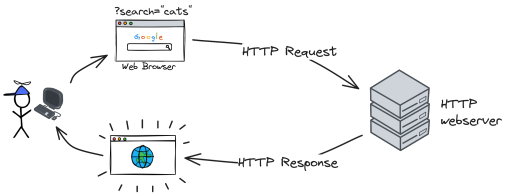

Figure 3.3: Request-Response model

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.

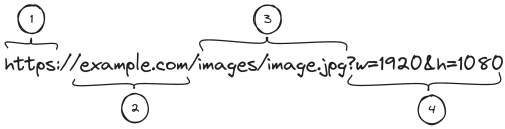

Figure 3.4: A typical URL

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.

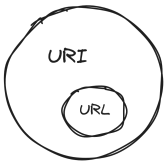

Figure 3.5: URL-URI Venn diagram

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
content-length: 257
server: gunicorn/19.9.0
access-control-allow-origin: *
access-control-allow-credentials: true

{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.81.0",
    "X-Amzn-Trace-Id": "Root=1-66252f66-1137e90d2c487549658727 43"
  },
  "origin": "103.204.132.228",
  "url": "https://httpbin.org/get"
}
```

3. 4.

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`
201	`created`
400	`bad request`
401	`unauthorized`
403	`forbidden`
404	`not found`
500	`internal server error`

3.2.6 The problem with HTTP

Now that we've discussed all the quirks of HTTP, there is still one small caveat about it. HTTP is based on a request-response model, mostly used in RESTful APIs (more on this later) meaning that you can't get data from a server without requesting for it initially. Suppose, you are trying to build a video streaming platform where you can have users stream some kind of audio or video and have clients that are feeded that data. You can't optimally achieve this by using traditional HTTP because you'll have to poll the server and request it to send you the frames of data. Ideally you'd want the server to stream the data to you until the client connection ends. To give you another example, say you want to build a chat application. You can have a server that you send details such as the content of the message, to whom the message is meant for, etc. The server cannot directly send this message to the client for whom the message is meant to go to. The recipient would need to poll the server at regular intervals and ask if there are any new messages, which is not very ideal.

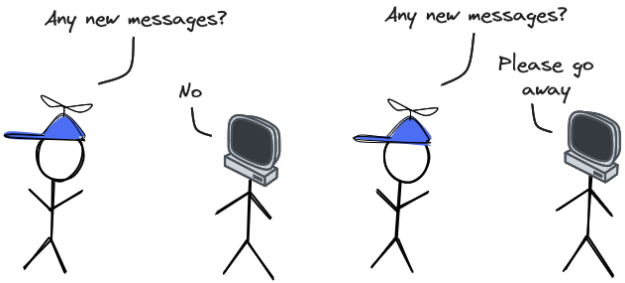

Figure 3.7: The server is annoyed at Sam's requests

You'd be wasting precious resources of the server and the user, requesting data that simply doesn't exist. Another solution would be to make use of a technique called long-polling, where you send a request but instead of receiving a response immediately, you keep the connection open so that the server can respond whenever it receives a message. But all of these are just work-arounds and not actual solutions. This is where a special protocol, Websocket, comes in.

> It is necessary to know that Websocket is not a better protocol than HTTP, it's just that they're both different tools meant for different use-cases. You should pick the right tool for the job and not the other way around.

3.3 Websocket

Websocket is a special type of communication protocol built on top of HTTP that allows for bi-directional communication between the client and the server. A websocket server can have multiple websocket connections open at a time to which it can send and receive data from. You can think of a websocket connection as a pipeline where the data is not synchronized and any party can send and receive data whenever they want, as long as the connection is open. This protocol becomes an ideal choice for applications where the server also has a lot of data to send like a video streaming platform such as YouTube, or doesn't know when new data will be available for a client. Websocket connections are made by establishing a handshake, similar to TCP, between two hosts. It is established after upgrading an initially made HTTP connection to a websocket connection with the use of HTTP headers.

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

Chapter 4

Backend Security and Cryptography

When we write backend applications, it is necessary to have good security around it. Building performant, fault-tolerant applications that are efficient and serve our users is one thing, but it is also necessary to make our application secure so that critical data is not compromised. As the complexity of our application rises, so does the vulnerabilities. Addressing these vulnerabilities and finding solutions to them is vital. Securing a backend application is a formidable task, especially when not equipped with the knowledge of the best practices, severity of attacks, and how someone can use them to gain information that they're not supposed to possess. But thankfully, due to our current understanding of backend application security, there are quite a few things that we can do in order to make sure those pesky attackers stay away. Let's go over them.

4.1 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 we should be doing so.

4.1.1 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 to grant you the access to your account.

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

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

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.

4.1.2 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.

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.

4.1.3 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?*

4.2 Hashing

Hashing in cryptography and backend security, is the process of converting data, or a piece of text into an encrypted, fixed-length string of characters

by making use of an algorithm, also called as a hash function. A hash of something is guaranteed to not change unless the data itself was modified, making it one of the most reliable methods to send data with no corruption, maintaining integrity. By converting your data into its hash with the help of a hash function, you can safely send it across the internet and have the recipient generate the hash of that data, and compare it with the hash that you generated. If it equals, then the data is unadulterated and safe to use, but if it is not, then it was most likely tampered with. But you might be asking: "What does this have to do with how we store passwords in our databases?"

4.2.1 Password hashing

When we store login information, most of the time, we will have data such as emails, phone numbers, and others. Out of these credentials, one of it is sure to be the password of a user, atleast when you're building an authentication system for your application from the ground up, and not relying on other tools such as Okta, and Google sign-in. This password is crucial to grant access to a user's account. Such a credential is not stored in plain-text as it cannot be compromised. Instead of storing passwords itself as plain-text, we store its hash.

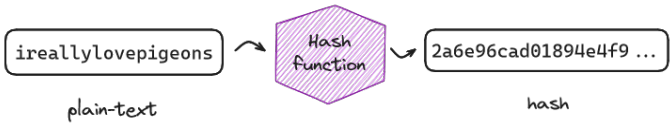

Figure 4.2: Hashing in action

When we get a password to store, we generate its hash, and then store that hash instead of the password in plain-text. So, whenever a sign-in request arrives at our backend, we get an email and a password generally. What we do is, we generate the hash of the password that was sent to us in the request, and then compare that hash with the hash that is stored on our side. If it equals, then the password is correct, and we grant the user the access to the account. A hash function will generate the same hash for the same unmodified data or text. One of the things to note about hashing is that you cannot decode data back from its hash, making it very secure for storing passwords and making sure that data is untouched.

You might be wondering how secure is a popular hashing algorithm such as SHA256 (Simple Hashing Algorithm 256) really is. If you use SHA256, you have a chance of 1 in 2^{256} possibilities to guess the data correctly. 2^{256} is such an unfathombly large number, that it contains 77 digits after its most significant digit!

4.3 Types of attacks

When we design backend applications, it is necessary to protect it from malpractices and certain actions from attackers that may compromise our server's data and our users' confidentiality. Let's go over some of these attacks and ways we can secure our backend application against.

4.3.1 DoS and Brute-force attacks

DoS attacks or *Denial of Service* attacks are attacks that are meant to overwhelm or overpower a remote server so as to render it useless for actual users who rely on it. These kinds of attacks are usually performed by attackers who may have ill intent towards a particular business or a particular person. Denial of Service is achieved by spamming a service with many meaningless requests and worthless data with the help of automated scripts and bot applications. Another similar attack is a brute-force attack meant to steal the password of another user by simply trying multiple possible combinations. All of these attacks whether brute-forcing or Denial of Service can be prevented by utilizing a security mechanism called rate-limiting.

// TODO: image of brute-forcing

Rate limiting

Rate limiting is the process of limiting or throttling the amount of incoming requests so as to prevent the server from being overwhelmed and continue to serve our users.

> You must've experienced something like this when trying to find the pin or password of a mobile phone by trying multiple combinations. The phone would simply prevent you from entering anything for some time or would even lock you out.

There are multiple ways to limit or throttle incoming requests, or, rate-limiting algorithms that are present in practice today. Let's go over a few of the most used ones:

4.3.2 Man in the middle attack

4.3.3 Packet sniffing attack

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

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.

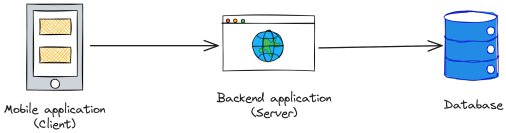

Figure 6.1: 3-tier application

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.

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,

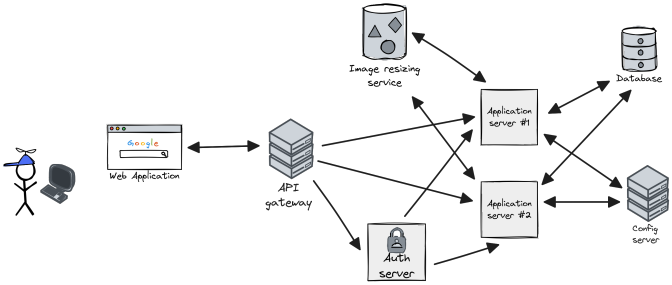

Figure 6.2: Microservice architecture

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 gigabyte to the machine to make it more capable than it previously was. Vertical scaling is simple, but it can also get pretty expensive and pricier as you keep climbing up the ladder. There is also a limit as to how much you can scale vertically, due to our current technological limitations. Scaling vertically can be different for different ways of upgrading that may be required. If you are running a database server, you might consider increasing the storage capacity of the disks to which your data is being written. If you want to run a highly responsive server, you should consider increasing its processing power.

Pros:

- Great for a single quick upgrade
- Cost effective, for smaller upgrades.

Cons:

- There are limitations to how much you can upgrade.
- Can get pricey after a certain threshold.
- May require you to upgrade other parts in order to upgrade the one you want to.

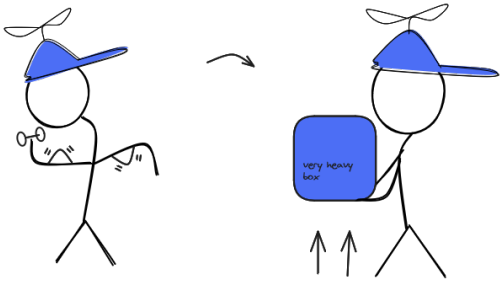

Figure 6.3: Scaling vertically: Sam increases his strength

6.4.2 Horizontal scaling

Horizontal scaling is the process of increasing the number of host machines or servers instead of increasing a singular machine's computing capacity. A load balancer is another machine used when scaling horizontally. It sits in front of the host machines usually running a software that distributes the workload evenly such that one machine or host is not overwhelmed completely. Horizontal scaling has many advantages since you can have as many computers as you like, doing work. It is also a much viable option in the long run because you can simply add another machine to the group of servers you may be running to ease the workload of the others.

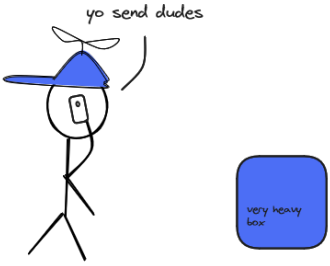

Figure 6.4: Scaling horizontally: Sam requests for more manpower

In conclusion, horizontal and vertical are two scaling methods that should be used after considering tradeoffs such as performance, cost effectiveness, etc.

Part II

Project BlogSpace

Chapter 7

Project Introduction and Setup

BlogSpace is a simple blogging platform that we will be building together in this book. Building a project after you've learnt some concepts is a great way to cement the knowledge and grok the true meaning and purpose of a concept inside your mind. We are going to build *BlogSpace* step-by-step, covering each detail, feature implementation and code, line-by-line. Most of the endpoints that we'll build, will be simple CRUD operations so that you pick them up easily. Note that we won't be building the frontend of the entire application as that is beyond the scope of this book. However, we will be testing and trying out our application's API endpoints using tools such as *Postman* (An API testing application) and `curl`. Let's go over the features that we will be implementing in *BlogSpace*.

> You may choose to add your own features on the project or skip the ones that you feel are redundant. I will also be leaving a few exercises for you to complete after the project is concluded.

7.1 Features

- **User login and signup** - We'll implement the total flow of creating a user account using details such as email, verifying the email of the account and adding login. Before the user is logged in and their email is verified, we will reject all requests from them.

- **Creating blog posts** - We will allow users to write as many blogs as they would like.
- **Like blog posts** - Blog posts can be liked by another user other than the author himself.
- **Search for blogs** - We will do keyword matching and search blog posts for the user.
- **Delete blog posts** - Blog posts can be deleted by the author of the post.

7.2 Features summary

- *User account creation*
- *User email verification*
- *User login*
- *User profile edit*
- *Blog creation*
- *Blog editing*
- *Like blogs*
- *Search for blogs*

For this project we'll be using Go (or Golang), specifically version `1.21`. We won't be using any library or framework in our project except for a tiny one called `sqlx` which is a superset of the standard `database/sql` library in Go. Meaning it has everything that `database/sql` has and a few extra extensions such as prepared statements. That's it! We won't be needing any massive framework or library, or install huge dependencies to get things done, we just need an editor where we can write your code and the command-line where we'll execute the project. The reason for doing this is to help you understand how applications work at a lower level instead of abstracting everything away to a framework. Frameworks are great when building applications that just work, but when your purpose is to learn, explore and understand, you should do so with as little help as possible.

7.3 Why Go?

You might be wondering why I chose Go for this project over something like Python, or maybe JavaScript.

The simple answer is, **I just like Go.**

It is an extremely simple programming language that is very readable even if you're not familiar with it. One of the biggest perks of Go is that it has pretty much everything you'd need to build applications in its standard library. You mostly don't need to install huge and bulky dependencies to setup a simple project to try some things out. Also, did I mention that it is one of the very few programming languages that are *blazingly fast*? Writing Go is fun and I would highly recommend anyone passionate about programming to learn it. Let's begin!

7.4 Setting the scene

Let us begin by creating an empty directory called 'blogspace', you can name this whatever you want. Head into it and initialize an empty `git` repository:

```
$ mkdir blogspace
$ cd blogspace
$ git init .
Initialized empty Git repository in blogspace/
```