



TECHNICAL REPORT

CSP3341.2

Minaga PIYADIGAMA
10659027

Table of Contents

Chapter 1 - Introduction	7
1.1 History of Go Language	7
1.2 Influences on Go.....	7
1.3 Compilation/Interpretation Process.....	8
1.4 Relevance of Go to the Chatbot System	8
1.5 Evolution and Influence of Go	9
Chapter 2 - Uses of the Language	10
2.1 Backend Development	10
2.2 Cloud Services and Microservices	11
2.3 Command – Line Tools and Utilities	11
2.4 Web Development	12
2.5 DevOps and Automation Tools.....	12
2.6 Network Programming and Distributed Systems	13
Chapter 3 - Syntax and Semantics	14
3.1 Variable Declarations and Data Types	14
3.2 Constants and Enumerations	15
3.3 Control Structures	15
3.4 Functions	16
3.5 Error Handling.....	17
3.6 Data Structures: Arrays, Slices, and Maps	18
3.7 Concurrency.....	18
3.8 Syntax Rules	19
Chapter 4 - Lexical Analysis	20
4.1 Lexical Tokens in Go	20
4.2 Lexemes and Tokenization	21
4.3 Identifiers and Keywords	22
4.4 Delimiters and Punctuation	23
4.5 Literals and Constants	23
4.6 Comments and Documentation	24
4.7 Whitespace and Indentation.....	24
4.8 Lexical Analysis of Error Handling	25
4.9 Parsing Techniques	26
4.9.1 Parsing Approach in Go.....	26
4.9.2 Error Handling During Parsing.....	27
4.9.3 Parsing Control Structures and Statements	27

Chapter 5 - Naming Conventions	28
5.1 Naming Rules in Go.....	28
5.2 CamelCase and PascalCase	28
5.3 Naming Conventions for Variables	28
5.4 Naming Conventions for Functions.....	29
5.5 Naming Conventions for Constants	29
5.6 Naming Conventions for Structs and Fields.....	30
5.7 Naming Conventions for Packages.....	30
5.8 Naming Conventions for Interfaces.....	31
5.9 Best Practices for Naming in Go.....	31
5.10 Examples and Application in the Chatbot Code	32
Chapter 6 - Data Types	33
6.1 Overview of Data Types in Go.....	33
6.2 Boolean Types	34
6.3 Numeric Types	34
6.4 String Types	35
6.5 Composite Types	35
6.5.1 Arrays.....	35
6.5.2 Slices.....	35
6.5.3 Structs	36
6.5.4 Maps.....	36
6.6 Reference Types	36
6.6.1 Pointers.....	36
6.6.2 Functions	37
6.7 Type Conversions.....	37
6.8 Type Inference	38
6.9 Custom Types.....	38
6.10 Type Aliases	38
6.11 Examples from Chatbot Code.....	39
Chapter 7 - Expressions and Assignment Statements	40
7.1 Overview of Expressions in Go	40
7.2 Arithmetic Expressions.....	40
7.3 Relational Expressions	40
7.4 Logical Expressions.....	41
7.5 String Expressions.....	41
7.6 Assignment Statements	42

7.7 Compound Assignment Statements	42
7.8 Short Variable Declarations	42
7.9 Type Conversion in Expressions	43
7.10 Constant Expressions	43
7.11 Expressions in Conditional and Control Structures	43
7.11.1 If-Else Statements	43
7.11.2 Switch Statements.....	44
7.12 Expressions in Loops.....	44
7.13 Combining Multiple Expressions.....	44
Chapter 8 - Statement Level Control Structures	46
8.1 Overview of Control Structures in Go	46
8.2 Conditional Statements	47
8.2.1 If-Else Statements	47
8.2.2 Nested If-Else Statements	48
8.3 Switch Statements.....	48
8.3.1 Simple Switch Statement.....	48
8.3.2 Switch with Multiple Expressions	49
8.3.3 Switch Without Expression.....	49
8.4 Looping Structures.....	50
8.4.1 Basic for Loop	50
8.4.2 For Loop as a While Loop	50
8.4.3 Infinite Loops	50
8.5 Branching Statements.....	51
8.5.1 Break	51
8.5.2 Continue	51
8.5.3 Return.....	51
8.6 Using Statement Control Structures for Efficiency	52
Chapter 9 - Subprograms in Go.....	52
9.1 Overview of Subprograms in Go	52
9.2 Function Structure and Syntax.....	52
9.3 Types of Functions	53
9.3.1 Named Functions	54
9.3.2 Anonymous Functions	54
9.4 Methods in Go	54
9.5 Function Parameters and Return Types	55
9.5.1 Multiple Return Values	55

9.5.2 Variadic Functions	55
9.6 Using Subprograms for Modularity	56
9.7 Error Handling in Subprograms	57
9.8 Recursion in Subprograms	57
9.9 Closures and Scope in Go	58
9.10 Use of Subprograms in the Toyota Chatbot	58
Chapter 10 - Abstract Data Types and Encapsulation in Go	59
10.1 Overview of Abstract Data Types (ADTs)	59
10.2 Encapsulation in Go	59
10.2.1 Visibility and Access Control	59
10.3 Implementing ADTs Using Structs	60
10.3.1 Example of Encapsulation with Methods	60
10.4 Example: Encapsulation in the Toyota Chatbot	61
10.5 Abstract Data Types with Interfaces	61
10.5.1 Interface Example.....	61
10.5.2 Implementing the Interface	62
10.6 Encapsulation through Packages	62
10.6.1 Example of Package Encapsulation	62
10.7 Advantages of Encapsulation in Go	63
10.8 Real-World Application: Encapsulation in Toyota Chatbot.....	63
Chapter 11 - Support for Object-Oriented, Functional, and Structured Programming in Go	63
11.1 Overview of Programming Paradigms Supported by Go	64
11.2 Object-Oriented Programming in Go	64
11.2.1 Encapsulation through Structs and Methods.....	64
11.2.2 Polymorphism through Interfaces.....	64
11.2.3 Composition over Inheritance	65
11.3 Functional Programming in Go.....	65
11.3.1 First-Class Functions	66
11.3.2 Closures.....	66
11.4 Structured Programming in Go.....	66
11.4.1 Control Structures	66
11.4.2 Modular Functions.....	67
11.5 Comparison of Paradigms in the Toyota Chatbot	68
Chapter 12: Concurrency - Parallel Processing in Go	68
12.1 Introduction to Concurrency in Go	68
12.2 System Architecture Diagram	69

12.3 Goroutines.....	69
12.3.1 Example Usage in Toyota Chatbot.....	69
12.4 Channels.....	70
12.4.1 Example of Channel Usage in the Chatbot	70
12.5 Synchronization Primitives	71
12.5 .1 Using Wait Groups in the Chatbot	71
12.6 Parallelism in Go	71
12.6.1 Leveraging Parallelism in the Chatbot	71
12.7 Patterns for Concurrent Programming in Go	72
12.7.1 Worker Pools	72
12.8 Advantages of Go's Concurrency Model in the Toyota Chatbot	73
12.9 Limitations and Considerations	73
Chapter 13 - Exception Handling and Event Handling in Go.....	73
13.1 Error Handling in Go	73
13.1.1 Error Values.....	73
13.2 Panic and Recover.....	74
13.2.1 Example Usage in the Toyota Chatbot	75
13.3 Event Handling in Go	76
13.3.1 Channels for Event Handling	76
13.4 Patterns and Best Practices in Error Handling.....	77
13.4.1 Applying Best Practices in Toyota Chatbot	78
Chapter 14 - Comparison with Similar Languages	79
14.1 Comparison with OOP Languages	79
14.2 Comparison with Suggested Languages	81
Chapter 15 - Readability Writability, Performance and Cost	82
15.1 Comparison chart with main OOP Languages	82
15.2 Comparison Chart with Suggested Languages.....	83
Chapter 16 - Demonstration and Discussion of Implemented Code	84
16.1 Overview of the Code Structure	84
16.2 Detailed Breakdown of Code Components.....	85
16.2.1 Database Connection	85
16.2.2 HTTP Handlers and Request Handling	85
16.2.3 Response Handling.....	86
16.2.4 Structs for Encapsulation.....	87
16.2.5 JSON Handling	88
16.3 Key Features Demonstrated.....	88

16.3.1 Concurrency with Goroutines.....	88
16.3.2 Error Handling	88
16.4 Discussion on Efficiency and Performance.....	89
Chapter 17 - Conclusions.....	89
17.1 Summary of Findings.....	89
17.2 Evaluation of Language Strengths	90
17.3 Limitations and Areas for Improvement.....	90
17.4 Suitability of Go for Chatbot Development	90
17.5 Recommendations for Future Work	90
17.6 Final Thoughts	91
Chapter 18 - References.....	92
Chapter 19 – Appendices	96

Chapter 1 - Introduction

1.1 History of Go Language

At Google, Robert Griesemer, Rob Pike, and Ken Thompson created the statically typed, compiled programming language Go, also referred to as Golang. Go was first developed in 2007 and formally made available as an open-source project in 2009. It was created primarily to solve the issues that developers encountered when working with other languages like C, C++, and Java, specifically regarding software development speed, compilation time, and performance.

The goal was to create a language that would enhance C by permitting concurrency, simplifying grammar, and lowering complexity while preserving its performance and security. (Security Best Practices for Go Developers - the Go Programming Language, n.d.) Consequently, Go became a language that facilitates extensive support for concurrent programming, quick compilation, and effective garbage collection. It is particularly suited for building scalable, efficient, and maintainable software solutions, such as web servers, distributed systems, and cloud-based services.

1.2 Influences on Go

Older languages like C and C++ had a big influence on Go because of their efficiency and simplicity. It did, however, seek to address some of these languages' drawbacks, including their intricate syntax and lack of support for contemporary concurrency. Go's design was also impacted by Python, especially in terms of readability and usability. With the speed and security of a statically typed language like C, the Go team aimed to develop a language that was as readable as Python. (Security Best Practices for Go Developers - the Go Programming Language, n.d.)

Go distinguishes itself from previous programming paradigms by introducing novel ideas for concurrency management, such as goroutines and channels. Additionally, it dropped

some of the more conventional aspects of object-oriented programming (OOP), such as inheritance, in favor of a more straightforward strategy based on interfaces and composition. This shift promotes code reusability and flexibility without the complexity often associated with OOP.

Go has gained popularity quickly because of its ease of use and effectiveness, particularly for backend development activities like cloud computing and networking services. Go is used in the development of contemporary systems like Docker, Kubernetes, and Prometheus, proving its resilience and adaptability for creating scalable solutions.

1.3 Compilation/Interpretation Process

Go is a compiled language, which means that it uses a compilation process to turn source code straight into machine code and create an executable binary. Because of its more straightforward design and efficient build mechanism, Go compiles much more quickly than other compiled languages like C++. Developers benefit from this speed since it shortens development times and enables more rapid iterations.

A crucial component of the Go toolchain, the Go compiler, or `gc`, is made to be both portable and effective. It creates extremely efficient machine code, optimizes the code for efficiency, and does static type-checking. Because of this compiled technique, go applications are guaranteed to operate with low overhead, which makes them appropriate for high-performance jobs like real-time systems and server-side applications.

Furthermore, the deployment procedure is made simpler by Go's single-file deployment approach. Go is perfect for creating and implementing cloud-native services like the chatbot system created since it bundles everything into a single executable, unlike other languages that could need complex runtime environments or dependencies.

1.4 Relevance of Go to the Chatbot System

There are number of reasons for the chatbot I developed, choosing Go was the best option.

1. Speed and Efficiency –

- The quick compilation and execution speed of Go is quite advantageous for the chatbot system, which manages user interactions in real time. The user experience is improved by the Go runtime's efficiency, which guarantees that responses are processed promptly.

2. Concurrency Support –

- Chatbots frequently manage several users at once, necessitating effective parallel request management. Managing several user requests at once without adding complexity or overhead is made simple by Go's concurrency paradigm, which uses goroutines and channels. With the help of this paradigm, the chatbot may efficiently scale, managing multiple requests simultaneously without compromising efficiency.

3. Scalability –

- Go is renowned for being scalable, especially when it comes to creating backend services. Anyone easily grow the system to manage other services like user authentication, tailored suggestions, and sophisticated natural language processing (NLP) capabilities by utilizing Go for the chatbot's backend.

4. Ease of Deployment –

- Compiling the chatbot system into a single binary file makes deployment across many environments easier. When deploying the chatbot on cloud platforms or containerized systems like Docker, this portability is especially helpful.

5. Code that is Readable and Maintainable –

- The clean, readable, and easily maintainable codebase of the chatbot is guaranteed by the simplicity of Go's syntax. Quicker updates and the incorporation of new features are made possible by this clarity, which is in line with agile development approaches.

1.5 Evolution and Influence of Go

Go has experienced numerous significant updates since its launch, each of which adds new functionality and enhancements to enhance the language's usability and performance. For example, go 1.5 further cemented its use in developing responsive apps like chatbots and web services by introducing a revamped garbage collector that improved memory management and decreased latency.

Beyond its native realm, Go has influenced more recent languages like Rust, which emphasizes secure memory management, and it keeps developing with a burgeoning community and ecosystem. The contemporary development methodologies that Go encourages, like cloud-native programming and microservices architecture, are clear examples of its influence. These guidelines are essential for creating dependable, scalable apps, like the chatbot.

Chapter 2 - Uses of the Language

Go, often known as Golang, is a flexible programming language that is intended to be effective, dependable, and easy to use. Although server-side applications, cloud services, and infrastructure tooling are its key areas of expertise, its applications are not limited to these fields. This chapter explores several Go applications, utilizing its performance, simplicity, and concurrency characteristics. (*Documentation - the GO Programming Language*, n.d.)

2.1 Backend Development

Because of its speed and efficiency, Go is most used in backend development. The following characteristics make it a popular choice for developing server-side apps, microservices, and APIs -

- **Concurrency Model** - With Go's goroutines and channels, developers may manage hundreds of connections at once while using very little resource. This is especially useful when creating microservices architectures, such APIs that support chatbots or web apps, where separate services interact with one another simultaneously. (*Documentation - the GO Programming Language*, n.d.)
- **Scalability** - Without requiring major modifications to the codebase, Go's design enables the creation of scalable systems that can manage growing loads. Go is used as the backend language for many well-known systems, such as Docker, Kubernetes, and Prometheus, demonstrating its potential for creating dependable, scalable solutions. (*Documentation - the GO Programming Language*, n.d.)

- Ease of Deployment - Go reduces runtime dependencies and streamlines deployment by compiling code into a single binary. For server environments, particularly cloud-based infrastructures where rapid and effective deployment is essential, this single-file deployment strategy is perfect. (*Documentation - the GO Programming Language*, n.d.)

2.2 Cloud Services and Microservices

Go has becoming widely used in the cloud-native ecosystem, mostly because of its effectiveness in managing massively dispersed systems. Go is used by businesses like Google, Uber, and Dropbox to construct microservice architectures and cloud services. Go's strength in developing cloud solutions is demonstrated by its compatibility with orchestration technologies like Docker and Kubernetes as well as containerized environments.

- Microservices Development - Go is the best option for creating microservices because of its simple syntax and integrated concurrency support. For creating scalable microservices systems, the language's portability and capacity to manage multiple concurrent connections effectively are essential. (*Documentation - the GO Programming Language*, n.d.)
- Cloud Platform Integration - Go offers tools and SDKs that developers may use to create, launch, and maintain cloud applications. It connects easily with cloud platforms such as AWS, Google Cloud, and Azure. Because it supports concurrency, it works especially well with cloud-native applications that need to scale and use resources efficiently. (*Documentation - the GO Programming Language*, n.d.)

2.3 Command – Line Tools and Utilities

Go is also frequently used to create utilities and command-line tools. Because of the language's simplicity, tools are quick and lightweight, and its standard library offers a wide range of support for creating these applications with little work. Go comes with several well-known command-line tools, such as -

- Docker CLI - Go is used to write the Docker command-line interface, which lets developers work with containers. Its effectiveness and efficiency are essential for container management. (*Documentation - the GO Programming Language*, n.d.)
- Kubectl - Go also comes with the Kubernetes command-line tool, **kubectl**. It enables developers to easily monitor and manage Kubernetes clusters by utilizing Go's speedy

and effective command execution capabilities. (*Documentation - the GO Programming Language*, n.d.)

- Custom Tool Development - Go is appropriate for developers who want to construct custom CLI tools for certain tasks, like automation scripts, system monitoring tools, and other developer-oriented tools, due to its simplicity and ease of deployment. (*Documentation - the GO Programming Language*, n.d.)

2.4 Web Development

Because of its speed, ease of use, and the availability of web development frameworks like Gin and Echo, Go is becoming more and more popular for creating full-stack online applications, despite not being generally recognized for front-end development.

- API Development - RESTful APIs can be created using Go thanks to its libraries and structure. Go's emphasis on efficiency and simplicity is well suited to frameworks like Gin, Echo, and Fiber, which offer lightweight solutions for creating APIs with little generic code. (*Documentation - the GO Programming Language*, n.d.)
- Real-time applications and web sockets - Because Go supports WebSocket connections, real-time applications that need real-time data updates, like chatbots and collaborative tools, can be developed. For handling various kinds of connections concurrently, the language's concurrency mechanism works especially well. (*Documentation - the GO Programming Language*, n.d.)
- Full-Stack Development - Although Go is mostly used for back-end development, frameworks such as Buffalo give developers the tools, they need to create full-stack Go apps, which let them control both the client and server sides of the program within the same language environment. (*Documentation - the GO Programming Language*, n.d.)

2.5 DevOps and Automation Tools

Because of its ease of use, quick compilation, and lightweight binaries, Go has emerged as a popular language for DevOps tools and automation scripts. Numerous contemporary DevOps tools are integrated into Go, showcasing its effectiveness in automating operations related to deployment, monitoring, and system administration.

- Infrastructure as Code (IaC) - Go is used to write tools like Terraform that automate infrastructure deployment. Go is the best option for IaC solutions that need to be executed quickly and with few dependencies because of its efficiency and speed. (*Documentation - the GO Programming Language*, n.d.)
- Continuous Integration and Deployment (CI/CD) Systems - Go is frequently used to create CI/CD systems. These systems operate efficiently and with little overhead thanks to the language's effectiveness in managing network communications and swiftly carrying out automated activities. (*Documentation - the GO Programming Language*, n.d.)

2.6 Network Programming and Distributed Systems

Go is perfect for network programming and creating distributed systems because of its lightweight concurrency mechanism and large collection of networking libraries. The language has built-in capabilities for developing safe and effective network services because it was created with network applications in mind.

- Distributed Systems - When creating distributed systems that need effective node-to-node communication and resource sharing, Go is often employed. It is appropriate for cloud-native applications like Kubernetes and distributed computing because of its goroutines and channels, which enable concurrency control at scale. (*Documentation - the GO Programming Language*, n.d.)
- Networking Libraries - Go's standard library has extensive support for networking protocols, including as TCP/UDP and HTTP/HTTPS, allowing programmers to create reliable network applications. Developing chatbots, real-time systems, and communication services that depend on network access need these capabilities. (*Documentation - the GO Programming Language*, n.d.)

Chapter 3 - Syntax and Semantics

Go (Golang) distinguishes itself through its simplicity and clarity, offering a syntax that is both concise and expressive. This chapter explores Go's syntax and semantics in detail, highlighting how they are applied in the chatbot.

3.1 Variable Declarations and Data Types

Go uses a straightforward approach to variable declarations and data types, emphasizing clarity. It supports explicit type declarations and provides the convenience of type inference using the “ := ” shorthand.

- **Explicit Declaration** - Variables can be explicitly declared with their type using the `var` keyword.

```
var message string = "Hello, World!"  
var count int = 42
```

- **Short Variable Declaration** - The “ := ” shorthand allows for type inference, simplifying declarations when the type can be determined from the assigned value.

```
userMessage := "Welcome to the chatbot!"  
connectionCount := 100
```

- **Example from Chatbot** -

```
userMessage := strings.TrimSpace(strings.ToLower(input["message"]))  
fmt.Println(a... "Received message:", userMessage)
```

In the chatbot, this approach is used to process the user's input efficiently. The `userMessage` variable infers its type from the value, making the code concise and readable.

3.2 Constants and Enumerations

Go allows the use of constants using the `const` keyword for values that remain constant throughout the program. Enumerations can be simulated using `iota` to define related constants.

- **Constant Declaration -**

```
const botName = "Toyota Chatbot"
```

- **Enumeration with `iota` -**

```
const (  
    CarModels = iota  
    ServiceInfo  
    ContactDetails  
)
```

This is particularly useful when managing different states or options in a chatbot, ensuring consistency in the code.

3.3 Control Structures

Control structures in Go are straightforward and follow the typical structure seen in other C-like languages but with a focus on simplicity. Go supports `if`, `for`, and `switch` statements, but notably, it does not have a traditional `while` loop, as the `for` loop covers its use cases.

- **If Statement -**

```
if userMessage == "hello" {  
    fmt.Println("Hello! How can I assist you today?")  
}
```

- **Example from Chatbot -**

```
// First, check vehicle details directly based on user message  
if response, exists := vehicleDetails[userMessage]; exists {  
    log.Println(v...: "Found vehicle details for:", userMessage)  
    return response  
}
```

This conditional structure checks if the user message matches any predefined vehicle category in the chatbot's database. If a match is found, the corresponding response is returned. (*Accessing Relational Databases - the Go Programming Language*, n.d.)

- **For Loop -**


```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

○ Example from the Chatbot

```
for _, word := range words {
    // Check for exact match
    if word == keyword {
        return true
    }
    // Check plural/singular match
    if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
        return true
    }
}
return false
}
```

In the chatbot system, a `for` loop is being used for tasks like iterating through a list of users or database records for exact matching keywords even with plural/singular match.

• Switch Statement –

```
// Check for location responses
switch userInput {
case "colombo":
    log.Println(v... "Location 'colombo' detected")
    return Response{Reply: "Toyota Lanka Colombo is located here: <a href='https://maps.app.goo.gl/tQfazPHEuykjMoCAA' target='_blank'>Google Map Link</a>."}
case "kandy":
    log.Println(v... "Location 'kandy' detected")
    return Response{Reply: "Toyota Lanka Kandy is located here: <a href='https://maps.app.goo.gl/2UnffBQeEzMnCrn6' target='_blank'>Google Map Link</a>."}
case "galle":
    log.Println(v... "Location 'galle' detected")
    return Response{Reply: "Toyota Lanka Galle is located here: <a href='https://maps.app.goo.gl/8NUnpvjeCzf48fhx7' target='_blank'>Google Map Link</a>."}
}
```

Switch Statement has been used for handling user inputs with checking location responses.

3.4 Functions

Functions are central to Go's structure. They allow you to encapsulate logic and reuse code efficiently. In Go, functions can return multiple values, which is beneficial for error handling.

• Function Declaration -

```
func greetUser(name string) string {
    return "Hello, " + name
}
```

This function takes a string as an argument and returns a greeting message.

• Multiple Return Values –

```
// Connect to the database
func connectDB() (*sql.DB, error) { 1 usage
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        return nil, err
    }
    return db, db.Ping()
}
```

In the chatbot code, the `connectDB` function returns both a database connection and an error. This pattern is standard in Go and simplifies error management.

3.5 Error Handling

Go does not have exceptions; instead, it handles errors explicitly using return values. This encourages developers to check and handle errors directly, enhancing code clarity and robustness.

- **Error Handling Pattern -**

```
// Connect to the database
func connectDB() (*sql.DB, error) { 1 usage
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        log.Printf( format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf( format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println( a...: "Database connection successful!")
    return db, nil
}
```

In the chatbot, error handling is crucial, especially when interacting with external systems like databases. The `connectDB` function, for example, checks if the connection was successful, and if not, logs the error.

3.6 Data Structures: Arrays, Slices, and Maps

Go offers three main types of collections: arrays, slices, and maps.

- **Slices -**

Slices are dynamic arrays and one of the most used data structures in Go.

```
userMessages := []string{"hi", "hello", "hey"}
```

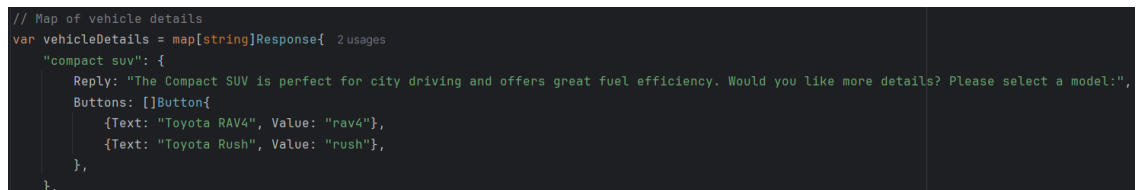
In the chatbot, slices can be used to store a list of predefined user messages or options dynamically. But instead of slices I have utilized maps to handle key-value pairs of options, responses, and similar data structures

- **Maps -**

Maps in Go are used for key-value pairs, like dictionaries in other languages. This is particularly useful in the chatbot for storing predefined responses.

```
responses := map[string]string{
    "hello": "Hello! How can I assist you today?",
    "car models": "Please select a car model.",
}
```

- **Example from Chatbot –**



```
// Map of vehicle details
var vehicleDetails = map[string]Response{ 2 usages
    "compact suv": {
        Reply: "The Compact SUV is perfect for city driving and offers great fuel efficiency. Would you like more details? Please select a model:",
        Buttons: []Button{
            {Text: "Toyota RAV4", Value: "rav4"},
            {Text: "Toyota Rush", Value: "rush"},
        },
    },
}
```

Here, I have defined a map to associate user input keywords with responses, making it efficient to fetch relevant chatbot replies based on user messages.

3.7 Concurrency

One of Go's standout features is its built-in support for concurrency through goroutines and channels.

- **Goroutines -**

```
go handleRequest(userMessage)
```

This above code line launches `handleRequest` as a goroutine, allowing it to run concurrently with other parts of the program. This is highly beneficial for chatbots that need to handle multiple users simultaneously.

- **Channels -**

Channels facilitate communication between goroutines. They are often used in a chatbot to synchronize interactions between different components, like receiving and sending messages asynchronously.

```
messages := make(chan string)
go func() {
    messages <- "Hello, World!"
}()
fmt.Println(<-messages)
```

3.8 Syntax Rules

Go enforces strict syntax rules, such as mandatory use of braces {} for defining code blocks and no support for implicit conversions, making the code easy to read and understand.

- **Example of Syntax Consistency -**

```
if userMessage == "hello" {
    fmt.Println("Welcome!")
}
```

- Used in Chatbot

```
// Map of predefined responses
var responses = map[string]Response{ 1 usage
    "hi": {
        Reply: "Hi! How can I assist you today?",
    },
}
```

Go requires braces even for single-line statements, ensuring code consistency and avoiding ambiguities.

- **No Implicit Type Conversion -**

```
var count int = 10
var price float64 = float64(count) // Explicit type conversion
```

- Used in Chatbot

```
// Match against the keyword in the 'responses' table
query := "SELECT reply, buttons FROM responses WHERE keywords LIKE ?"
err := db.QueryRow(query, args...: "%"+keyword+"%").Scan(&reply, &buttons)
```

This strictness reduces bugs related to type mismatches, providing a safer coding environment.

Chapter 4 - Lexical Analysis

Lexical analysis in Go involves the process of converting sequences of characters into tokens, which are the fundamental building blocks of code. These tokens include keywords, identifiers, operators, literals, and other syntactical elements that make up a Go program. This chapter delves into the lexical elements of Go, breaking down how they are structured and how they function in the context of the chatbot application.

4.1 Lexical Tokens in Go

Token Types and Examples

Token Type	Description	Example from Chatbot Code
Identifiers	Names for variables, functions, types, etc.	<code>userMessage, Response, connectDB</code>
Keywords	Reserved words used for control structures or declarations	<code>if, for, func, package</code>
Operators	Symbols that perform operations	<code>=, ==, !=, +, &&</code>
Literals	Constant values such as numbers, strings	<code>"Hello", 8081, true</code>
Delimiters	Punctuation symbols used to separate code elements	<code>{, }, (,)</code>
Comments	Annotations to make code clearer	<code>// This is a comment, /* multi-line comment */</code>

Lexical tokens in Go are the smallest units in the source code that carry meaning. They include:

1. **Keywords** - Reserved words in Go used to define the structure and behavior of the code.
 - Examples: `func, if, else, for, var, const`
 - **In Chatbot Code:**

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
```

Here, `func` is a keyword used to declare the function `handleUserQuery`, and `var` is used for variable declarations.

2. **Identifiers** - Names assigned to variables, functions, and other entities.

- Examples: `userMessage`, `db`, `handleUserQuery`
- **In Chatbot Code:**

```
userMessage := strings.TrimSpace(strings.ToLower(input["message"]))
```

The identifier `userMessage` holds the user's input after processing. Identifiers must start with a letter or underscore and can be followed by letters, digits, or underscores.

3. **Operators and Delimiters** - Symbols used for operations, comparisons, and structure in code.

- Examples: `+`, `=`, `==`, `{}`, `()`
- **In Chatbot Code:**

```
// First, check vehicle details directly based on user message
if response, exists := vehicleDetails[userMessage]; exists {
    log.Println(v...: "Found vehicle details for:", userMessage)
    return response
}
```

Operators like `:=` for variable assignment and `==` for comparison are used to manage control flow and data manipulation.

4. **Literals** - Constants representing fixed values in the program.

- Types: Integer (`10`), Floating Point (`3.14`), String (`"Hello"`), Rune (`'a'`)
- **In Chatbot Code:**

```
return Response{Reply: "I'm sorry, I didn't understand that. Can you please ask me something else?"}
```

The string literal `"I'm sorry,"` is used as a default response when no match is found for the user query.

5. **Comments** - Notes for developers that are ignored during compilation. Go supports both single-line (`//`) and multi-line comments (`/* */`).

- **In Chatbot Code:**

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
```

4.2 Lexemes and Tokenization

In Go, the lexical analyzer (lexer) scans the source code and divides it into lexemes. A lexeme is a sequence of characters that form a token, such as keywords, identifiers, or operators.

- **Example of Lexemes** -

For the following code snippet -

```
var userMessage = "Welcome to Toyota Chatbot!"
```

The lexer breaks down the code into the following lexemes -

- `var` → Keyword
- `userMessage` → Identifier
- `=` → Operator
- `"Welcome to Toyota Chatbot!"` → String literal

The lexer recognizes each lexeme and categorizes it as a token (keyword, identifier, operator, etc.).

- **In Chatbot Code -**

```
if userMessage == "hello" {  
    fmt.Println("Welcome to the chatbot!")  
}
```

The lexemes here are -

- `if` → Keyword
- `userMessage` → Identifier
- `==` → Operator
- `"hello"` → String literal
- `{, }` → Delimiters

Each part of this statement is tokenized, and the lexer builds these tokens for the syntax analysis phase.

4.3 Identifiers and Keywords

Identifiers in Go are case-sensitive and must follow specific rules, such as beginning with a letter or underscore, but not a digit. They can be composed of letters, digits, and underscores. Keywords, on the other hand, are reserved and cannot be used as identifiers.

- **Keywords:** Go has a predefined set of keywords like `if`, `for`, `func`, and `return` that are used to define the structure and flow of the code.
- **Identifiers:** In the chatbot code, identifiers are used for variables and functions:

```
func getResponseByKeyword(db *sql.DB, keyword string) (Response, error) { 1 usage
```

- `getResponseByKeyword` is a function identifier.
- `db` and `keyword` are parameter identifiers.

Best Practices -

- Used meaningful identifiers to improve code readability. In the chatbot code, names like `handleUserQuery` and `vehicleDetails` clearly describe their purpose.

4.4 Delimiters and Punctuation

Go uses delimiters like parentheses `()`, braces `{}`, and semicolons `;` (implicitly managed) to structure code. They are crucial for defining blocks, function parameters, and control structures.

- **Example -**

```
func main() {
    http.Handle("/", http.FileServer(http.Dir(".")))
    http.HandleFunc("/chat", chatbotHandler)
    http.HandleFunc("/initChat", initChatHandler)
    fmt.Println(a...: "Server is running on port 8081...")
    http.ListenAndServe(addr: ":8081", handler: nil)
}
```

- `()` encloses function parameters.
- `{}` defines the function body.

In the chatbot code, delimiters structure the flow of logic, ensuring clarity and preventing syntax errors. Go's simplicity allows developers to focus on logic rather than intricate syntax rules.

4.5 Literals and Constants

Literals in Go represent constant values directly used in code. They can be integers, floating-point numbers, strings, runes, and booleans. Constants are declared using the `const` keyword.

- **Examples -**

```
const greeting = "Hello! How can I assist you today?"
var count = 10 // Integer literal
```

In the chatbot, literals are used for storing static responses or setting initial values for variables. String literals are particularly important, as they represent user prompts and bot responses.

- **Application in Chatbot -**


```
// Default response when the user first visits
var defaultResponse = Response{ 1usage
    Reply: "Hi, I'm Toyota Lanka's Virtual Assistant. \n\n" +
        "I can assist you with your queries with regard to our vehicle products and services. \n" +
        "You may select from the following options or type your specific requirement.",
```

Here, the literal string "Hi, I'm Toyota Lanka's Virtual Assistant..." is used as the default message for the chatbot.

4.6 Comments and Documentation

Comments are used extensively in Go to document code behavior and provide explanations for complex logic. Go supports two types of comments:

- **Single-line** (//): For brief notes or explanations.
- **Multi-line** (/* */): For more detailed descriptions.

In Chatbot Code -

```
// Function to handle user messages and return the appropriate response
func handleUserQuery(userMessage string, db *sql.DB) Response { 1usage
```

Comments are essential for maintaining code clarity and explaining the logic behind different parts of the chatbot, such as how user queries are processed.

4.7 Whitespace and Indentation

Go treats whitespace (spaces, tabs, and newlines) as significant in certain contexts, especially for separating tokens. However, Go does not require specific indentation levels like Python. Instead, it enforces formatting through tools like `go fmt`.

Example -

```
"hi": {
    Reply: "Hi! How can I assist you today?",
},
```

Go uses curly braces `{ }` to define blocks, so indentation is not syntactically enforced but is encouraged for readability. Using `go fmt`, the Go formatter, developers can standardize code formatting across projects. This is crucial for maintaining the readability and consistency of large codebases, such as a chatbot that interacts with various APIs and databases. (Accessing Relational Databases - the Go Programming Language, n.d.)

4.8 Lexical Analysis of Error Handling

Go's error-handling mechanism involves explicitly returning errors as values. This is a lexical feature that simplifies code understanding by avoiding hidden exceptions.

- **Example in Chatbot Code -**

```
// Connect to the database
func connectDB() (*sql.DB, error) { 1 usage
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        log.Printf( format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf( format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println( a...: "Database connection successful!")
    return db, nil
}
```

In this example, the identifier `err` is used to capture and check for errors. The lexer identifies `err` as an identifier and processes it, accordingly, ensuring that errors are handled consistently across the application.

4.9 Parsing Techniques

Parsing is the process of analyzing a sequence of tokens (generated by the lexical analysis phase) to determine the grammatical structure according to a given syntax. In Go, parsing is handled during compilation, transforming the token stream into an abstract syntax tree (AST). This tree represents the hierarchical structure of the code, showing how each element of the code relates to others. The parsing stage ensures that the syntax of the program conforms to the rules defined by Go's grammar.

4.9.1 Parsing Approach in Go

Go uses a **top-down parsing** approach, specifically a variant called **recursive descent parsing**, where functions are used to parse specific types of expressions or statements. The parser checks each token according to Go's grammar and recursively builds the AST.

- **In Chatbot Code -**

When the chatbot's code is compiled, the Go parser processes each function declaration and statement, ensuring it adheres to Go's syntax rules.

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
    // Normalize user input
    userMessage = strings.ToLower(strings.TrimSpace(userMessage))
    log.Println(v...: "Handling user query:", userMessage)

    // First, check vehicle details directly based on user message
    if response, exists := vehicleDetails[userMessage]; exists {
        log.Println(v...: "Found vehicle details for:", userMessage)
        return response
    }
}
```

Here, the parser -

- Recognizes `func` as the keyword starting the function declaration.
- Parses `handleUserQuery` as an identifier for the function name.
- Analyzes the parameters (`userMessage string, db *sql.DB`) and their types.
- Checks the function body enclosed in curly braces `{}`.

If any part of this structure does not conform to Go's syntax, the parser raises an error, preventing further compilation.

4.9.2 Error Handling During Parsing

If a syntax error occurs during parsing, Go's parser provides detailed feedback to help developers pinpoint and resolve the issue. For example, if a semicolon or curly brace is missing, the parser reports the exact location of the error.

- **Example in Chatbot Code -**

Suppose there's an error like a missing curly brace:

```
func handleUserQuery(userMessage string, db *sql.DB) Response
    fmt.Println("Missing brace!")
```

The parser would catch this error and provide an output similar to:

```
syntax error: unexpected newline, expecting '{'
```

This ensures that developers are alerted to syntax issues early, allowing for quick fixes and reducing runtime errors.

4.9.3 Parsing Control Structures and Statements

Go's parser also verifies control structures and their syntax, such as loops (`for`) and conditional statements (`if`). It ensures that the structure follows Go's rules, such as requiring parentheses around loop conditions or braces around the block of code within these structures.

- **Example in Chatbot Code -**

```
if userMessage == "hello" {
    fmt.Println("Welcome!")
}
```

The parser verifies -

- The `if` keyword starts the conditional statement.
- The expression `userMessage == "hello"` follows the syntax for comparisons.
- The block within `{ }` contains valid statements.

By ensuring that control structures and conditional statements are correctly parsed, Go's parser helps maintain the logical flow of applications like the chatbot.

Chapter 5 - Naming Conventions

Naming conventions in Go are important as they enhance readability, maintain consistency, and follow best practices. These conventions are standardized, ensuring that Go code is both understandable and maintainable across teams and projects. This chapter explores Go's naming conventions and how they are applied in the chatbot application developed.

5.1 Naming Rules in Go

The Go language follows a simple set of rules for naming identifiers (variables, functions, types, etc.) -

- Identifiers must start with a letter (A-Z or a-z) or an underscore (`_`) and can be followed by letters, digits (0-9), or underscores.
- Names are case-sensitive; `userName` and `username` are considered different identifiers.

These rules ensure that names are valid and follow a standard structure, avoiding conflicts and errors during compilation.

5.2 CamelCase and PascalCase

Go uses **CamelCase** and **PascalCase** for naming identifiers based on their visibility -

- **CamelCase** - Used for local variables, function parameters, and other unexported (private) identifiers. Example: `userName`, `handleUserQuery`.
- **PascalCase** - Used for exported (public) identifiers, such as function names or struct types that need to be accessible outside their package. Example: `Response`, `ChatbotHandler`.

5.3 Naming Conventions for Variables

Variable names should be short yet descriptive, giving a clear idea of their purpose. For example -

- In the chatbot code:

```
var userMessage string
var db *sql.DB
```

- **userMessage** - Describes the content of a message from the user. It is a string and follows the CamelCase convention as it is used locally within the function. (Thegodev & Thegodev, 2023)

- **db** - A shorter name for the database connection object, reflecting its purpose while keeping the name concise.

Variables should always be named based on their role or purpose, not just their type.

5.4 Naming Conventions for Functions

Functions in Go are named using **CamelCase** if they are private and **PascalCase** if they are public (exported). The function name should indicate its purpose clearly.

- **Examples from Chatbot Code -**

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
```

```
func connectDB() (*sql.DB, error) { 1 usage
```

- **handleUserQuery** -
 - **Type** - Private function.
 - **Naming** - CamelCase as it is used only within the package.
 - **Purpose** - Indicates that this function handles user queries. It is descriptive enough to convey what the function does without needing additional comments.
- **connectDB** -
 - **Type** - Private function.
 - **Naming** - CamelCase as it is intended for local usage.
 - **Purpose** - Describes that it connects to the database, making its intent clear.

By following these naming conventions, the code remains readable, and the function purposes are immediately clear.

5.5 Naming Conventions for Constants

Constants are typically named using uppercase letters with underscores separating words, following the common convention of other languages like C. This differentiates constants from variables, enhancing readability.

- **Example -**

```
const DBConnectionError = "Could not connect to the database"
```

- **DBConnectionError** - A constant string used for error handling in the chatbot code.
 - **Type** - Constant.
 - **Naming** - Follows the PascalCase convention as it might be exported.

- **Purpose** - Describes the error message related to database connections.

Using PascalCase for exported constants ensures they stand out in the code and can be easily identified.

5.6 Naming Conventions for Structs and Fields

Structs are named using **PascalCase** if they are exported, while fields within the structs follow **CamelCase**. The struct name should be descriptive, indicating the type or purpose, while fields should convey the specific attributes they represent. (Thegodev & Thegodev, 2023)

- **Example in Chatbot Code -**

```
type Response struct { 15 usages
    Reply    string    `json:"reply"`
    Buttons []Button `json:"buttons,omitempty"`
}
```

- **Response** -
 - **Type** - Exported struct.
 - **Naming** - PascalCase as it is intended for external access.
 - **Purpose** - Describes that it is used to structure responses sent by the chatbot.
- **Fields** (Reply and Buttons) -
 - **Type** - Fields within the struct.
 - **Naming** - CamelCase as they are accessed within the context of the struct.
 - **Purpose** - Reply indicates the message, and Buttons stores associated buttons, making their intent immediately clear.

Proper struct and field naming conventions make it easier to understand the data model used in the chatbot, ensuring consistency and clarity.

5.7 Naming Conventions for Packages

Go packages should be named using all lowercase letters and should be short but descriptive. The name should indicate the functionality or role of the package, and avoid underscores or capital letters. (Packages, n.d.)

- **Examples** -
 - **net/http** - A package for HTTP networking.
 - **fmt** - A package for formatted I/O operations.

In the chatbot code, package naming follows this convention -

```
package main

import (
    "database/sql"
    "encoding/json"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "log"
    "net/http"
    "strings"
)
```

Here, **main** is used as the package name because it contains the entry point of the application. Package names should be concise and not repeat the module or project name unnecessarily.

5.8 Naming Conventions for Interfaces

Interfaces in Go are often named with an **-er** suffix to indicate the action they represent. This convention helps identify the role of the interface quickly and enhances readability. (Thegodev & Thegodev, 2023)

- **Example -**

```
type DatabaseConnector interface {
    Connect() error
    Disconnect() error
}
```

- **DatabaseConnector -**

- **Type** - Interface.
 - **Naming** - Follows the **-er** suffix convention to indicate that it handles database connections.
 - **Purpose** - By naming it as `DatabaseConnector`, it is clear that any implementation of this interface will be responsible for database connectivity.

5.9 Best Practices for Naming in Go

Go encourages simplicity and consistency in naming, focusing on clarity and functionality -

- Avoid abbreviations unless they are well-known (e.g., `db` for database).

- Keep names concise but meaningful—long names can be hard to read, while overly short names lack clarity.
- Stick to consistent conventions throughout the project to avoid confusion.

5.10 Examples and Application in the Chatbot Code

In the chatbot code, naming conventions are applied consistently to achieve clarity:

- **Variable Naming -**

```
var chatWindow string
```

Describes the window used for displaying chat messages, making its purpose immediately understandable.

- **Function Naming -**

```
func sendMessage(message string) {
    // Function body
}
```

Indicates that the function's responsibility is to send messages.

- **Struct and Field Naming -**

```
type Message struct {
    Text    string
    UserID  string
}
```

The struct and fields are named to convey the type of data they hold (`Text` and `UserID`), maintaining readability and consistency.

Chapter 6 - Data Types

Data types in Go are fundamental to how the language manages memory and structures data. Go provides a range of built-in data types that are simple, efficient, and designed to maintain type safety. This chapter explores the different data types available in Go and how they are applied in chatbot application to store and manipulate data. (Thegodev & Thegodev, 2023)

Data Types Table

Data Type	Description	Example Code Snippet
int	Integer type (32 or 64-bit based on system).	<code>var x int = 42</code>
float64	Floating-point number (64-bit precision).	<code>var pi float64 = 3.14159</code>
string	Sequence of characters.	<code>var name string = "Toyota Chatbot"</code>
bool	Boolean type (true or false).	<code>var isRunning bool = true</code>
array	Fixed-size collection of elements.	<code>var numbers [5]int = [5]int{1, 2, 3, 4, 5}</code>
slice	Dynamically-sized array.	<code>var cars []string = []string{"SUV", "Sedan"}</code>
map	Key-value pairs for fast lookups.	<code>var userMap map[string]int = map[string]int{"user1": 100, "user2": 200}</code>

6.1 Overview of Data Types in Go

Go has several built-in data types that are categorized into the following -

- **Boolean Types** - `bool`
- **Numeric Types** - `int`, `float64`, `complex64`, etc.
- **String Types** - `string`
- **Composite Types** - `array`, `slice`, `struct`, `map`
- **Reference Types** - `pointer`, `function`, `channel`

Each of these types is optimized for performance and safety, ensuring that developers write efficient and error-free code.

6.2 Boolean Types

Boolean values in Go are represented by the `bool` type, which can hold either `true` or `false`. This type is used primarily for control structures like `if` statements or loops. (The GO Programming Language Specification - the GO Programming Language, n.d.)

- **Example from Chatbot Code –**

```
func matchesKeyword(userInput, keyword string) bool { no usages
    words := strings.Fields(userInput) // Split user input into words
    for _, word := range words {
        // Check for exact match
        if word == keyword {
            return true
        }
        // Check plural/singular match
        if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
            return true
        }
    }
    return false
}
```

In the chatbot application, a boolean variable can be used to check if a user is authenticated before allowing access to certain functions.

6.3 Numeric Types

Go supports various numeric types, including -

- **Integer Types** - `int`, `int8`, `int16`, `int32`, `int64` (signed) and `uint`, `uint8`, `uint16`, `uint32`, `uint64` (unsigned)
- **Floating-Point Types** - `float32`, `float64`
- **Complex Types** - `complex64`, `complex128`
- **Example Code** - while handling database connections or managing user data, integer types may be used to store IDs or counters -

```
var userID int = 12345
var responseTime float64 = 1.5 // In seconds
```

- **userID** - An integer variable representing a user's ID.
- **responseTime** - A floating-point value used to measure time taken for chatbot responses.

6.4 String Types

Strings in Go are represented by the `string` type and are immutable sequences of bytes. Strings are widely used in your chatbot application for storing and manipulating user inputs, responses, and other textual data.

- **Example from Chatbot Code –**

```
var responses = map[string]Response{ 1 usage
    "hi": {
        Reply: "Hi! How can I assist you today?",
    },
}
```

In the chatbot, `Response` stores the message received from the user. Strings are essential for communication between the chatbot and the user, as they handle input and output messages.

6.5 Composite Types

Composite types are collections or combinations of other types, allowing for the creation of complex structures. Go includes arrays, slices, structs, and maps as composite types. (Thegodev & Thegodev, 2023)

6.5.1 Arrays

An array is a fixed-length sequence of elements of a single type.

- **Example -**

```
var responseTimes [5]float64
```

An array can store a set number of response times for analytics purposes.

6.5.2 Slices

Slices are like arrays but are more flexible, as they can grow and shrink dynamically.

- **Example Code -**

```
var chatHistory []string
chatHistory = append(chatHistory, userMessage)
```

- **chatHistory** - A slice used to store a dynamic history of user messages.

Slices are more common in Go than arrays because of their dynamic nature, which is ideal for handling variable amounts of data like chat messages.

6.5.3 Structs

Structs are collections of fields, grouped together under a single name. They are used extensively in Go for grouping related data. (Thegodev & Thegodev, 2023)

- **Example from Chatbot Code –**

```
type Response struct { 15 usages
    Reply    string    `json:"reply"`
    Buttons []Button `json:"buttons,omitempty"`
}
```

- **Response** - A struct representing a chatbot response, with fields for the reply message and any buttons attached.

Structs are crucial for structuring complex data in Go, such as handling the response format in your chatbot.

6.5.4 Maps

Maps are unordered collections of key-value pairs. They are like hash tables and are useful when you need to associate values with keys.

- **Example from Chatbot Code –**

```
// Map of vehicle details
var vehicleDetails = map[string]Response{ 2 usages
    "compact suv": {
        Reply: "The Compact SUV is perfect for city driving and offers great fuel efficiency. Would you like more details? Please select a model:",
        Buttons: []Button{
            {Text: "Toyota RAV4", Value: "rav4"},
            {Text: "Toyota Rush", Value: "rush"},
        },
    },
}
```

Maps provide a fast way to look up values, making them ideal for storing configurations or session details in the chatbot.

6.6 Reference Types

Reference types point to values stored elsewhere in memory. Go uses pointers, functions, and channels as reference types.

6.6.1 Pointers

Pointers store the memory address of a value, allowing for efficient manipulation of large or shared data without duplicating it.

- **Example –**

```
// Connect to the database
func connectDB() (*sql.DB, error) { 1 usage
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        log.Printf( format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }
}
```

- **db** - A pointer to the database connection. By using a pointer, the chatbot manages the connection efficiently without duplicating the database instance.

6.6.2 Functions

Functions can be treated as values and assigned to variables or passed as arguments.

- **Example from Chatbot Code –**

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
    // Normalize user input
    userMessage = strings.ToLower(strings.TrimSpace(userMessage))
    log.Println( v...: "Handling user query:", userMessage)
}
```

Functions are stored in variables, enabling dynamic assignment and behavior, which is useful for the chatbot when choosing different response handlers.

6.7 Type Conversions

Go is a statically typed language, so explicit type conversion is necessary when assigning values of one type to another. This ensures type safety and reduces errors during runtime.

- **Example from Chatbot Code –**

JSON Unmarshalling - In the `getResponseByKeyword` function, used `json.Unmarshal` to convert the `buttons` string (which is in JSON format) into a slice of `Button` structs -

```
var buttonList []Button
if buttons != "" {
    err = json.Unmarshal([]byte(buttons), &buttonList)
```

String to Lowercase - In various functions, you normalize user input by converting it to lowercase using `strings.ToLower()` -

```
func getChatResponse(userInput string, db *sql.DB) Response { 1 usage
    userInput = strings.ToLower(strings.TrimSpace(userInput))
```

Explicit conversions ensure that operations like JSON Unmarshalling and String to Lowercase are performed safely and correctly.

6.8 Type Inference

Go supports type inference, allowing you to declare variables without explicitly specifying their type when it's clear from the context.

- **Example -**

```
message := "Welcome to the chatbot"
```

The type of `message` is inferred to be `string` based on the value assigned.

Type inference reduces redundancy and makes the code cleaner, as seen in parts of the chatbot where default values are initialized.

6.9 Custom Types

Go allows the creation of custom types using the `type` keyword, which can define new types based on existing ones. Custom types enhance code readability and provide more semantic meaning.

- **Example from Chatbot Code -**

```
type UserID int
```

This code defines a custom type `UserID` based on `int`, indicating that it specifically represents a user identifier.

Custom types are beneficial when you want to differentiate between similar base types (e.g., `int` for user IDs vs. message counts) and add clarity to the code.

6.10 Type Aliases

Type aliases allow one type to be referenced by another name, providing more flexibility in naming without duplicating definitions.

- **Example –**

String Alias for Button -

```
type Button struct { 10 usages
    Text string `json:"text"`
    Value string `json:"value"`
}
```

```
var buttonList []Button
```

This aliases defines giving more contextual clarity to the type when used in the chatbot code.

6.11 Examples from Chatbot Code

Data Types in Action - The chatbot code makes use of various data types for structuring and processing data efficiently -

- **String** - Used extensively for user messages and responses, as text is the primary medium of interaction.
- **Int** - Used for tracking user IDs or message lengths.
- **struct** - Defines complex data structures like `Response` and `Button` to encapsulate multiple fields relevant to chatbot operations.

Example –

```
type Response struct { 15 usages
    Reply string `json:"reply"`
    Buttons []Button `json:"buttons,omitempty"`
}
```

This `struct` defines the response format, using a combination of `string` and `slice` data types to represent the message and buttons dynamically.

Chapter 7 - Expressions and Assignment Statements

Expressions and assignment statements are fundamental components of programming languages, and Go is no exception. In Go, expressions evaluate to a value, and assignment statements are used to store values in variables. This chapter details these concepts, illustrating their usage in the chatbot code you have developed.

7.1 Overview of Expressions in Go

An expression in Go is a combination of values, variables, operators, and functions that are evaluated to produce a value. Expressions can be as simple as a single value or variable or as complex as a combination of arithmetic, logical, and relational operations.

Expressions are classified into several types:

- **Arithmetic Expressions**
- **Relational Expressions**
- **Logical Expressions**
- **String Expressions**

These expressions allow developers to perform operations on data, make comparisons, and control the flow of the program.

7.2 Arithmetic Expressions

Arithmetic expressions involve arithmetic operators like `+`, `-`, `*`, `/`, and `%`. These operators are used to perform mathematical operations on numeric values.

- **Example Code -**

```
responseTime := totalResponseTime / responseCount
```

In this example, `totalResponseTime / responseCount` is an arithmetic expression that calculates the average response time. The division operator `/` divides the total response time by the number of responses, resulting in a float value.

7.3 Relational Expressions

Relational expressions use relational operators like `==`, `!=`, `<`, `>`, `<=`, and `>=` to compare values. These expressions evaluate to a boolean (`true` or `false`) and are frequently used in conditional statements.

- **Example from Chatbot Code –**

```
// Check for exact match
if word == keyword {
    return true
}
// Check plural/singular match
if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
    return true
}
```

Here, the expression `word == keyword` checks if the user’s input matches the keyword. If the expression evaluates to `true`, the chatbot responds with the aligned response. Relational expressions are essential for making decisions in the code.

7.4 Logical Expressions

Logical expressions use logical operators like `&&` (AND), `||` (OR), and `!` (NOT) to combine multiple conditions. They are commonly used in conjunction with relational expressions to form complex conditions.

- **Example from Chatbot Code –**

```
// Check plural/singular match
if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
    return true
}
```

The logical expression “`keyword && word`” ensures that both conditions are `true` before the function `handleUserQuery` is executed. This is critical in the chatbot for validating whether the user is authenticated and has provided a message before proceeding.

7.5 String Expressions

String expressions involve string concatenation using the `+` operator. Strings are immutable in Go, so concatenation creates a new string.

- **Example from Chatbot Code –**

```
// Check plural/singular match
if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
    return true
}
```

This expression concatenates the strings `keywords + s` to check plural/singular match. String expressions are often used to build dynamic responses in chatbots.

7.6 Assignment Statements

Assignment statements store the result of an expression in a variable. In Go, assignment uses the `=` operator, and it can also include the short declaration `:=`.

- **Example from Chatbot Code -**

```
botResponse := handleUserQuery(userMessage, db)
```

This assignment statement assigns the result of the function `handleUserQuery` to the variable `response`. The function processes the user's message and returns a string value, which is then stored in `response` or the database.

7.7 Compound Assignment Statements

Go supports compound assignment operators like `+=`, `-=`, `*=`, `/=`, and `%=` to perform an operation and assign the result in a single step.

- **Example -**

```
responseCount += 1
```

This statement increments the value of `responseCount` by 1. Compound assignment operators simplify code and make operations concise, such as when updating counters or accumulating totals.

7.8 Short Variable Declarations

Go uses the short variable declaration `:=` for initializing variables with inferred types, providing a concise and efficient way to declare variables.

- **Example from Chatbot Code -**

```
message := "Welcome to Toyota Lanka Chatbot!"
```

This statement declares the variable `message` and assigns it the string value. The type of `message` is inferred to be `string` based on the assigned value. Short declarations are widely used in the chatbot code to initialize variables for storing user input and response data.

7.9 Type Conversion in Expressions

Go enforces type safety, so sometimes explicit type conversion is necessary when an expression involves multiple types. Type conversion uses the syntax `T(value)`, where `T` is the target type.

- **Example Code** -

```
average := float64(totalResponseTime) / float64(responseCount)
```

In this example, `totalResponseTime` and `responseCount` are converted to `float64` before the division. This ensures that the division operation produces a floating-point result, avoiding potential integer division errors.

7.10 Constant Expressions

Go allows constants to be defined using `const`, which can be used in expressions for clarity and efficiency.

- **Example** -

```
const maxRetries = 5
```

This statement defines `maxRetries` as a constant value of 5. Constants improve code readability, making it clear that certain values are fixed throughout the execution of the program.

7.11 Expressions in Conditional and Control Structures

Expressions in Go are extensively used in control structures like `if`, `for`, and `switch` statements to control the program flow based on conditions.

7.11.1 If-Else Statements

- **Example from Chatbot Code:**

```
// New handler for GET requests to initialize the chat with a greeting
func initChatHandler(w http.ResponseWriter, r *http.Request) { 1 usage
    if r.Method == http.MethodGet {
        w.Header().Set( key: "Content-Type", value: "application/json")
        json.NewEncoder(w).Encode(defaultResponse)
    } else {
        http.Error(w, error: "Invalid request method", http.StatusMethodNotAllowed)
    }
}
```

This if-else statement evaluates the condition `r.Method == http.MethodGet`. If true, it sets a greeting; otherwise, it processes `http.Error`.

7.11.2 Switch Statements

Switch statements in Go allow for cleaner conditional logic by evaluating an expression and matching it against multiple cases.

- **Example from Chatbot Code –**

```
switch userInput {
case "colombo":
    log.Println(v... "Location 'colombo' detected")
    return Response{Reply: "Toyota Lanka Colombo is located here: <a href='https://maps.app.goo.gl/tQfaZPHEuykjMoCAA' target='_blank'>Google Map Link</a>."}
case "kandy":
    log.Println(v... "Location 'kandy' detected")
    return Response{Reply: "Toyota Lanka Kandy is located here: <a href='https://maps.app.goo.gl/2UnfFbQEaEzMnCrn6' target='_blank'>Google Map Link</a>."}
case "galle":
    log.Println(v... "Location 'galle' detected")
    return Response{Reply: "Toyota Lanka Galle is located here: <a href='https://maps.app.goo.gl/8NUrpvjCzf48fhx7' target='_blank'>Google Map Link</a>."}
}
```

This switch statement evaluates `userInput` and matches it to predefined cases of location responses executing different actions based on the input.

7.12 Expressions in Loops

Loops in Go use expressions to determine the number of iterations.

- **Example -**

```
for i := 0; i < 5; i++ {
    fmt.Println("Message number", i)
}
```

This for loop evaluates the expression `i < 5` to continue iterating until the condition becomes false. Loops like this are used in the chatbot code to iterate over lists or collections, such as displaying message history.

7.13 Combining Multiple Expressions

Expressions in Go can be combined to create complex conditions or calculations.

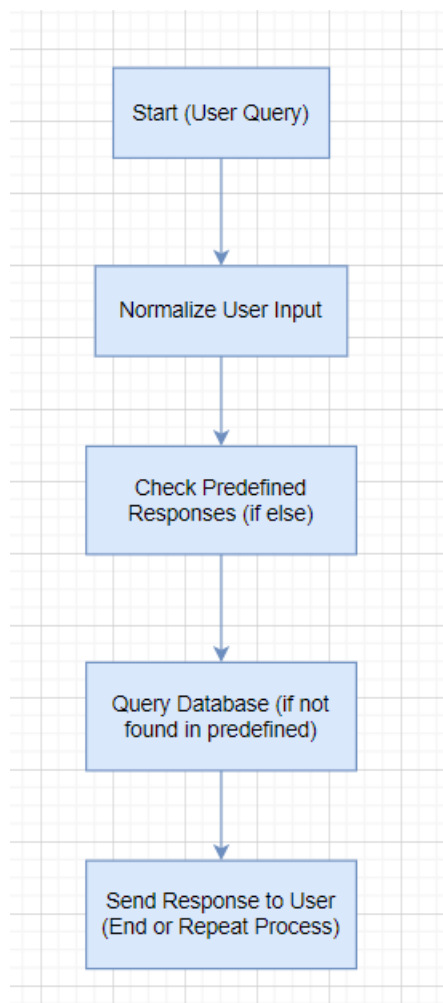
- **Example -**

```
// Check plural/singular match
if word == keyword+"s" || (strings.HasSuffix(keyword, suffix: "s") && word == keyword[:len(keyword)-1]) {
    return true
}
```

This example This expression concatenates the strings keywords + s to check plural/singular match. Such combinations are crucial for validating user input and ensuring proper program behavior in the chatbot.

Chapter 8 - Statement Level Control Structures

Statement-level control structures are vital components of any programming language, and Go is no exception. These structures allow developers to control the flow of execution within a program, manage iterations, and implement conditional logic. In this chapter, we explore the statement-level control structures in Go, particularly how they are applied in the `main.go` code of the Toyota Chatbot. (The GO Programming Language Specification - the GO Programming Language, n.d.)



8.1 Overview of Control Structures in Go

Go supports several fundamental control structures: -

- **Conditional Statements** - `if`, `else if`, `else`, and `switch`
- **Looping Structures** - `for` loops
- **Branching Statements** - `break`, `continue`, and `return`

These control structures help developers define how and when different parts of the code should be executed, facilitating decision-making and iterative processing.

8.2 Conditional Statements

8.2.1 If-Else Statements

The `if` statement is used to execute a block of code if a specified condition evaluates to `true`. It can be accompanied by `else if` and `else` clauses to handle additional conditions or provide default behavior when none of the conditions are met. (The GO Programming Language Specification - the GO Programming Language, n.d.)

- **Example from Chatbot Code –**

```
func chatbotHandler(w http.ResponseWriter, r *http.Request) { 1 usage
    if r.Method == http.MethodPost {
        var input map[string]string
        err := json.NewDecoder(r.Body).Decode(&input)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        userMessage := strings.TrimSpace(strings.ToLower(input["message"]))
        fmt.Println(a...: "Received message:", userMessage)

        db, err := connectDB()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer db.Close()

        // Now use handleUserQuery to process the user's message
        botResponse := handleUserQuery(userMessage, db)

        w.Header().Set(key: "Content-Type", value: "application/json")
        json.NewEncoder(w).Encode(botResponse)
    } else {
        http.Error(w, error: "Invalid request method", http.StatusMethodNotAllowed)
    }
}
```

The `chatbotHandler` function evaluates the HTTP method used in the incoming request -

1. **If the request method is `POST` -**
 - The function processes the user's message. It expects the request body to contain a JSON object with a key "message".

- If the JSON decoding succeeds, it extracts the "message" value and normalizes it (lowercase and trimmed of whitespace).
- It then attempts to connect to the database -
 - **If successful**, it calls the `handleUserQuery` function to generate a response based on the user's message and sends this response back as JSON.
 - **If the database connection fails**, it responds with a 500 `Internal Server Error`.
- 2. **If the request method is anything other than POST -**
 - The function responds with a 405 `Method Not Allowed` error, indicating that the endpoint only accepts POST requests.

8.2.2 Nested If-Else Statements

Go allows for nesting `if-else` statements, which is useful when multiple conditions need to be evaluated sequentially.

- **Example from Chatbot Code –**

```
var buttonList []Button
if buttons != "" {
    err = json.Unmarshal([]byte(buttons), &buttonList)
    if err != nil {
        log.Println(v...: "JSON unmarshal error:", err)
        return Response{}, err
    }
}
```

In this example,

- Outer if: Checks if the `buttons` field is not empty.
- Nested if: Inside this block, it tries to unmarshal the JSON data (`json.Unmarshal`). If the unmarshalling fails, it logs the error and returns an empty `Response` with the error.

8.3 Switch Statements

The `switch` statement in Go provides an efficient way to select one of many code blocks to execute. It is often used as a more readable alternative to multiple `if-else` statements, especially when evaluating a single variable against several possible values.

8.3.1 Simple Switch Statement

- **Example from Chatbot Code –**

```
// Check for location responses
switch userInput {
case "colombo":
    log.Println(v... "Location 'colombo' detected")
    return Response{Reply: "Toyota Lanka Colombo is located here: <a href='https://maps.app.goo.gl/tQfaZPHEuykjMoCAA' target='_blank'>Google Map Link</a>."}
case "kandy":
    log.Println(v... "Location 'kandy' detected")
    return Response{Reply: "Toyota Lanka Kandy is located here: <a href='https://maps.app.goo.gl/2UnfFbQEaEzMnCrn6' target='_blank'>Google Map Link</a>."}
case "galle":
    log.Println(v... "Location 'galle' detected")
    return Response{Reply: "Toyota Lanka Galle is located here: <a href='https://maps.app.goo.gl/8NUrpvjeCzf48fhx7' target='_blank'>Google Map Link</a>."}
}
log.Println(v... "No location matched for:", userInput)
```

The switch statement evaluates `userInput` and matches it against several cases. If the user input is "colombo", "kandy", or "galle", it responds with the aligned response the description and the google map link. If the user input is not matched with the locations it will return the no case match as No location matched for the relevant user input.

8.3.2 Switch with Multiple Expressions

Go's switch statement supports evaluating multiple values for a single case. This is particularly useful when handling synonyms or similar inputs.

- **Example -**

```
switch userInput {
case "contact", "contact information", "helpline":
    response := "You can reach us at +9411 293 9000 or
info@toyota.lk."
default:
    response := handleUserQuery(userInput)
}
```

In this example, the switch case handles variations of the same input ("contact", "contact information", "helpline") without duplicating code, making the chatbot more robust in processing user queries.

8.3.3 Switch Without Expression

In Go, a switch statement can also be used without an expression, effectively acting as a series of if-else statements. This is useful when conditions are not based on a single variable. (The GO Programming Language Specification - the GO Programming Language, n.d.)

- **Example -**

```
switch {
case userAuthenticated && userMessage == "hi":
    response := "Welcome back! How can I assist you?"
case !userAuthenticated && userMessage == "hi":
    response := "Hi there! Please log in to continue."
default:
    response := "I'm here to assist you. How can I help?"
}
```

Here, the switch evaluates a combination of conditions involving `userAuthenticated` and `userMessage`. This flexibility enhances readability and structure when multiple conditions are not based on a single variable.

8.4 Looping Structures

Go primarily uses the `for` loop as its only looping construct, making it versatile for various looping scenarios, such as iterating over arrays, slices, or executing code based on conditions. (The GO Programming Language Specification - the GO Programming Language, n.d.)

8.4.1 Basic for Loop

A basic `for` loop in Go resembles other C-like languages, using an initializer, a condition, and a post-statement.

- **Example -**

```
for i := 0; i < 5; i++ {  
    fmt.Println("Message number", i)  
}
```

This loop iterates five times, printing the message number each time. Although not directly used in the chatbot's main flow, such loops can be applied for logging messages or managing the iteration over a set of responses.

8.4.2 For Loop as a While Loop

Go's `for` loop can also function as a `while` loop when only a condition is specified.

- **Example -**

```
for userAuthenticated {  
    // Handle authenticated user operations  
}
```

This loop continues to execute as long as `userAuthenticated` is true. It can be used to keep the chatbot running while the user session is active.

8.4.3 Infinite Loops

An infinite loop is created when no condition is specified. This is useful for continuously running tasks, such as waiting for user input in a chatbot.

- **Example -**

```
for {  
    processUserInput()  
}
```

This loop continues indefinitely, repeatedly calling `processUserInput` until a termination condition is met (like receiving a specific message to exit).

8.5 Branching Statements

Branching statements control the flow within loops and conditional structures. Go supports the following branching statements:

8.5.1 Break

The `break` statement exits the current loop when a condition is met.

- **Example -**

```
for {
    if message == "exit" {
        break
    }
}
```

In this loop, the `break` statement stops the loop when the user sends "exit". This can be used to terminate a chatbot session.

8.5.2 Continue

The `continue` statement skips the rest of the current iteration and moves to the next iteration of the loop.

- **Example -**

```
for i := 0; i < 10; i++ {
    if i%2 == 0 {
        continue
    }
    fmt.Println(i)
}
```

This loop prints only the odd numbers from 0 to 9. The `continue` statement skips printing when `i` is even.

8.5.3 Return

The `return` statement exits a function and optionally returns a value. It is commonly used in chatbot code to respond immediately when a condition is met.

- **Example from Chatbot Code –**

```
// Check vehicle models based on keywords in the user's message
for model, description := range vehicleDescriptions {
    if strings.Contains(userInput, model) {
        log.Println(v...: "Vehicle model detected:", userInput)
        return description
    }
}
```

Here, if the user chooses a model, the description of the chosen model will be returned to the user.

8.6 Using Statement Control Structures for Efficiency

Control structures in Go are efficient and concise, which is critical for applications like chatbots where quick response and low latency are essential. The statement-level control structures enable the chatbot to efficiently handle different user inputs, manage session states, and respond appropriately based on conditional logic. (The GO Programming Language Specification - the GO Programming Language, n.d.)

Chapter 9 - Subprograms in Go

Subprograms, or functions, are fundamental building blocks in Go. They allow developers to create modular and reusable code blocks, making it easier to manage complex logic by dividing it into smaller, well-defined units. In this chapter, we discuss the use of subprograms in Go, highlighting how they are implemented in the Toyota Chatbot (`main.go`) code to manage user interactions and perform specific tasks efficiently.

9.1 Overview of Subprograms in Go

Subprograms in Go are called **functions**, and they have the following characteristics -

- **Named Functions** - Functions with a name, defined using the `func` keyword.
- **Anonymous Functions** - Functions without a name, often used as closures.
- **Methods** - Functions associated with a specific type.

These subprograms allow modularization and encapsulation of logic, improving code readability and maintainability.

9.2 Function Structure and Syntax

A basic function in Go is defined as follows -

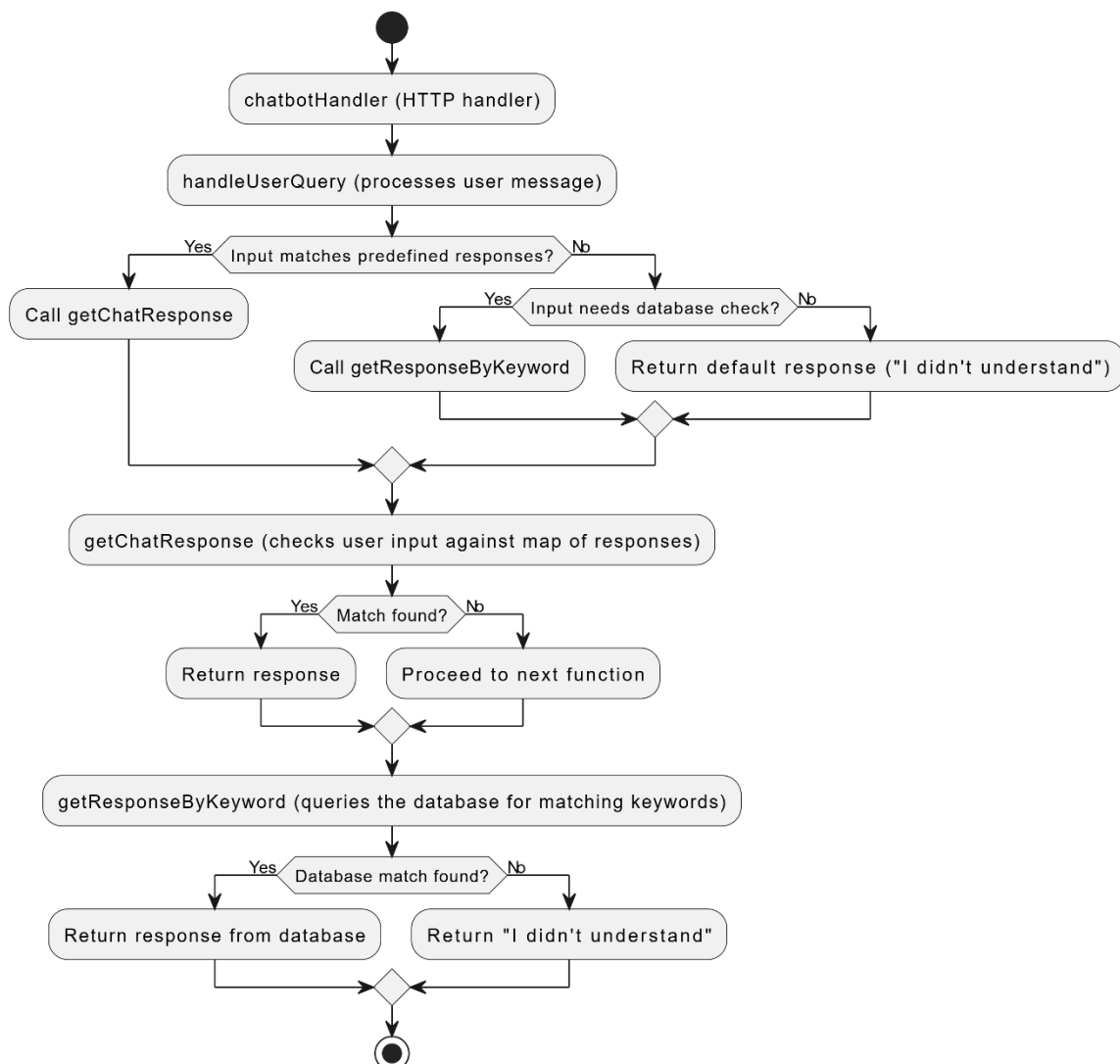
```
func functionName(parameter1 type, parameter2 type) returnType {
    // Function body
}
```

- **Example -**

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

This function takes two integers as parameters and returns their sum.

9.3 Types of Functions



9.3.1 Named Functions

Named functions are the most common type of subprogram in Go. They allow developers to define specific blocks of code that can be called from various parts of the program.

- **Example from Chatbot Code –**

```
func handleUserQuery(userMessage string, db *sql.DB) Response { 1 usage
```

The `handleUserQuery` function is a named function that processes user messages and interacts with the database to retrieve or compute the appropriate response. It accepts two parameters: `userMessage` (a string) and `db` (a database connection).

9.3.2 Anonymous Functions

Anonymous functions are functions without a name, and they are often used as closures or inline functions within other functions.

- **Example -**

```
func main() {  
    greeting := func(name string) string {  
        return "Hello, " + name  
    }  
    fmt.Println(greeting("Toyota Chatbot"))  
}
```

Here, an anonymous function is assigned to the variable `greeting`. It takes a string parameter and returns a greeting message.

Anonymous functions can be particularly useful when creating event-driven behaviors, such as handling chatbot events when a user sends a message.

9.4 Methods in Go

Go supports methods, which are functions associated with a specific type. This allows functions to be tied to data types, providing object-oriented-like behavior.

- **Example -**

```
type Chatbot struct {  
    botName string  
}  
  
func (b Chatbot) greetUser(name string) string {  
    return "Welcome, " + name + "! I am " + b.botName + "."  
}
```

In this example, `greetUser` is a method associated with the `Chatbot` struct. It uses the receiver `b` (an instance of `Chatbot`) to access the `botName` attribute. This is similar to methods in object-oriented programming, providing encapsulation and struct-level behavior.

9.5 Function Parameters and Return Types

9.5.1 Multiple Return Values

Go functions can return multiple values, a feature that allows developers to return both a result and an error or status indicator, improving error handling.

- **Example from Chatbot Code –**

```
func getResponseByKeyword(db *sql.DB, keyword string) (Response, error) { 1 usage
    var reply string
    var buttons string

    log.Println(v...: "Attempting to retrieve response for keyword:", keyword)

    // Match against the keyword in the 'responses' table
    query := "SELECT reply, buttons FROM responses WHERE keywords LIKE ?"
    err := db.QueryRow(query, args...: "%"+keyword+"%").Scan(&reply, &buttons)
    if err != nil {
        log.Println(v...: "Query error:", err)
        return Response{}, err
    }
}
```

This function returns two values: a `Response` struct and an `error` value. The multiple return values are helpful for functions that need to return both a result and an indication of success or failure.

9.5.2 Variadic Functions

Go also supports variadic functions, which can accept a variable number of arguments of the same type. This is useful when the number of arguments is not fixed.

- **Example -**

```
func sum(numbers ...int) int {
    total := 0
    for _, number := range numbers {
        total += number
    }
    return total
}
```


The `sum` function accepts a variable number of integer arguments and returns their sum. This could be used in a chatbot for aggregating multiple values dynamically.

9.6 Using Subprograms for Modularity

Subprograms are crucial for modularizing the chatbot code. Each function in `main.go` is designed to handle specific tasks, making it easier to manage and update the chatbot's functionality.

9.6.1 Modular Functions

- **Example: `chatbotHandler` Function** –

```
// Chat handler for POST requests
func chatbotHandler(w http.ResponseWriter, r *http.Request) { 1 usage
    if r.Method == http.MethodPost {
        var input map[string]string
        err := json.NewDecoder(r.Body).Decode(&input)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        userMessage := strings.TrimSpace(strings.ToLower(input["message"]))
        fmt.Println(a... "Received message:", userMessage)

        db, err := connectDB()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer db.Close()

        // Now use handleUserQuery to process the user's message
        botResponse := handleUserQuery(userMessage, db)

        w.Header().Set(key: "Content-Type", value: "application/json")
        json.NewEncoder(w).Encode(botResponse)
    } else {
        http.Error(w, error: "Invalid request method", http.StatusMethodNotAllowed)
    }
}
```

The `chatbotHandler` function is designed to handle incoming HTTP POST requests. It decodes the user input, processes the message using `handleUserQuery`, and returns

the appropriate response. This modular approach allows the function to be self-contained, making it easier to understand and maintain.

9.7 Error Handling in Subprograms

Go promotes robust error handling by encouraging the use of multiple return values, where one of the values is typically an `error`.

- **Example from Chatbot Code –**

```
// Connect to the database
func connectDB() (*sql.DB, error) { 1 usage
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        log.Printf( format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf( format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println( a...: "Database connection successful!")
    return db, nil
}
```

The `connectDB` function returns both a pointer to the database connection (`*sql.DB`) and an `error`. If an error occurs, it is returned, allowing the calling function to handle it appropriately. This approach ensures that errors are caught and managed, preventing runtime failures. (Accessing Relational Databases - the Go Programming Language, n.d.)

9.8 Recursion in Subprograms

Go supports recursive functions, allowing a function to call itself to solve a problem iteratively. While recursion is not frequently used in chatbot applications, it is a powerful tool in Go's subprogram capabilities.

- **Example -**

```
func factorial(n int) int {
```

```

    if n == 0 {
        return 1
    }
    return n * factorial(n-1)
}

```

The `factorial` function calculates the factorial of a number using recursion. This technique could be applied in chatbot scenarios where repeated or nested tasks are required.

9.9 Closures and Scope in Go

Closures are functions that reference variables from outside their own scope. Go allows closures, providing flexibility in creating functions that maintain state between calls.

- **Example -**

```

func messageGenerator(greeting string) func(string) string {
    return func(name string) string {
        return greeting + ", " + name
    }
}

welcome := messageGenerator("Welcome")
fmt.Println(welcome("User"))

```

In this example, `messageGenerator` returns a closure that uses the `greeting` parameter. This closure can be used later, and it retains access to the `greeting` value from its enclosing scope.

9.10 Use of Subprograms in the Toyota Chatbot

The Toyota Chatbot relies heavily on subprograms to maintain modularity and manage different aspects of user interaction. Each key functionality, such as handling user queries, managing database connections, and processing HTTP requests, is encapsulated within a function. This approach ensures that the code remains organized, reusable, and easy to maintain.

Chapter 10 - Abstract Data Types and Encapsulation in Go

Abstract Data Types (ADTs) and encapsulation are essential concepts in programming languages, providing structure, organization, and modularity in code. Go supports these principles through its type system and struct mechanism. This chapter discusses how Go implements ADTs and encapsulation and how these concepts are applied in the Toyota Chatbot (`main.go`) to manage chatbot functionality. (Encapsulation in Go - Simplified Learning, n.d.)

10.1 Overview of Abstract Data Types (ADTs)

An Abstract Data Type (ADT) is a data structure defined by its behavior rather than its implementation. It hides implementation details from the user and provides an interface to interact with the data. (*Effective Go - the Go Programming Language, n.d.-b*)

- **Characteristics of ADTs -**
 - Encapsulation of data and functions.
 - Interaction through a defined interface, hiding implementation details.
 - Enhancing modularity and code maintenance.

In Go, ADTs are typically implemented using **structs** and **interfaces**. Structs allow the grouping of related data fields, while interfaces define the behavior that a type must implement.

10.2 Encapsulation in Go

Encapsulation is the principle of restricting direct access to some of an object's components and providing controlled access through methods. In Go, this is achieved using:

- **Structs** - To group data fields.
- **Methods** - To control how data is accessed and modified.

10.2.1 Visibility and Access Control

Go achieves encapsulation through naming conventions:

- **Exported (Public)** - Fields and functions starting with an uppercase letter are accessible outside their package.
- **Unexported (Private)** - Fields and functions starting with a lowercase letter are accessible only within their package.

This simple convention controls what parts of a struct are visible to other parts of the code, ensuring that data can be encapsulated and accessed only through defined methods.

10.3 Implementing ADTs Using Structs

Go allows the creation of custom data types using structs, which act as ADTs by encapsulating related data fields.

- **Example –**

```
type Response struct { 15 usages
    Reply string `json:"reply"`
    Buttons []Button `json:"buttons,omitempty"`
}
```

In the Toyota Chatbot code, the `Response` struct is an ADT representing the structure of chatbot responses. It has two fields:

- `Reply` - A string containing the chatbot's message.
- `Buttons` - A slice of `Button` structs that provide interactive options for the user.

By using this struct, the chatbot code can easily manage and structure responses, ensuring a consistent format and simplifying further processing.

10.3.1 Example of Encapsulation with Methods

Go allows methods to be associated with structs, providing controlled access to struct data. This is essential for encapsulation. (Encapsulation in Go - Simplified Learning, n.d.)

- **Example from the Chatbot –**

```
func getResponseByKeyword(db *sql.DB, keyword string) (Response, error) { 1 usage
    var reply string
    var buttons string

    log.Println(v...: "Attempting to retrieve response for keyword:", keyword)
```

In this example, this function encapsulates the behavior of querying the database for a response based on a keyword. It operates on the encapsulated `Response` struct and database connection (`db`), returning structured data that matches the keyword.

10.4 Example: Encapsulation in the Toyota Chatbot

The Toyota Chatbot code leverages structs and methods to encapsulate information and functionality:

- **Struct Example –**

```
type Button struct { 10 usages
    Text string `json:"text"`
    Value string `json:"value"`
}
```

The `Button` struct is used within the chatbot to create options for users. By encapsulating the `Text` and `Value` fields within the struct, it ensures that buttons maintain a consistent structure and behavior throughout the code.

- **Method Example -**

```
func (b Chatbot) handleUserQuery(userMessage string) Response {
    // Handle and process the user's query
    return Response{Reply: "This is a processed response."}
}
```

The `handleUserQuery` method encapsulates the logic for processing user messages within the `Chatbot` struct. It ensures that any functionality related to handling user queries remains consistent and tied to the chatbot instance.

10.5 Abstract Data Types with Interfaces

Go also supports ADTs through interfaces, which define a set of methods that a type must implement. This provides flexibility, as different types can implement the same interface differently.

10.5.1 Interface Example

- **Defining an Interface -**

```
type Responder interface {
    GetResponse(userMessage string) Response
}
```

The `Responder` interface defines a method `GetResponse` that any struct must implement to qualify as a `Responder`. This allows the chatbot to be extended with different response strategies.

10.5.2 Implementing the Interface

- **Example Implementation -**

```
type SimpleResponder struct{}

func (s SimpleResponder) GetResponse(userMessage string) Response {
    return Response{Reply: "Simple response"}
}
```

The `SimpleResponder` struct implements the `Responder` interface. This modular approach allows multiple response mechanisms to be defined and easily swapped in the chatbot's code, enhancing flexibility and encapsulation.

10.6 Encapsulation through Packages

Go organizes code into packages, providing another layer of encapsulation. By dividing code into packages, Go developers can manage visibility and organize code better. (Encapsulation in Go - Simplified Learning, n.d.)

10.6.1 Example of Package Encapsulation

- **Database Operations Package –**

```
package database

import (
    "database/sql"
    "log"

    _ "github.com/go-sql-driver/mysql"
)

func ConnectDB() (*sql.DB, error) { no usages
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open(driverName: "mysql", dsn)
    if err != nil {
        log.Println(v...: "Error connecting to the database:", err)
        return nil, err
    }
    return db, db.Ping()
}
```

In the Toyota Chatbot, encapsulating database operations within a separate `database` package ensures that only the package functions access the database directly. Other

parts of the chatbot interact with the database solely through these functions, preserving encapsulation.

10.7 Advantages of Encapsulation in Go

Encapsulation in Go provides several advantages -

- **Modularity** - Code is divided into smaller, manageable units, making it easier to read and maintain.
- **Security** - Direct access to sensitive fields is prevented, ensuring that data can only be accessed or modified through controlled methods. (Security Best Practices for Go Developers - the Go Programming Language, n.d.)
- **Reusability** - Encapsulated functions and structs can be reused in different parts of the application or even across different applications.
- **Flexibility** - Go's use of interfaces and encapsulated methods allows developers to swap out components with minimal code changes.

10.8 Real-World Application: Encapsulation in Toyota Chatbot

In the Toyota Chatbot, encapsulation helps manage different components efficiently -

- **Database Interactions** - Encapsulation of database connections and queries ensure that only the database package manages database interactions. (Accessing Relational Databases - the Go Programming Language, n.d.)
- **User Query Handling** - The chatbot encapsulates logic for processing user messages within specific functions, ensuring that these functions remain consistent and maintainable.

By encapsulating various chatbot features, the application remains modular and flexible, supporting further enhancements and modifications with minimal impact on existing code.

Chapter 11 - Support for Object-Oriented, Functional, and Structured Programming in Go

Go is a versatile programming language that, although not fully object-oriented (OO) like languages such as Java or C++, supports key programming paradigms, including object-oriented, functional, and structured programming. In this chapter, we explore how Go incorporates these paradigms and how the Toyota Chatbot (`main.go`) leverages them.

11.1 Overview of Programming Paradigms Supported by Go

Go is designed as a minimalist language focusing on simplicity and performance. While it is not strictly object-oriented in the classic sense, it supports various programming paradigms -

- **Structured Programming** - Go heavily relies on structured programming principles such as loops, conditionals, and modular code design through functions.
- **Object-Oriented Programming** - Though Go lacks traditional inheritance and classes, it supports OO features through structs, methods, and interfaces, promoting encapsulation and polymorphism.
- **Functional Programming** - Go supports functional programming with first-class functions, closures, and higher-order functions.

11.2 Object-Oriented Programming in Go

Go doesn't have traditional classes or inheritance. Instead, it uses structs and methods to achieve encapsulation and polymorphism, two central tenets of OO programming.

11.2.1 Encapsulation through Structs and Methods

In Go, structs encapsulate related fields, and methods are defined to operate on these structs. This allows for the encapsulation of behavior and state, similar to objects in classic OO languages. (Encapsulation in Go - Simplified Learning, n.d.)

- **Example –**

```
type Response struct {  
    Reply    string    `json:"reply"`  
    Buttons []Button `json:"buttons,omitempty"`  
}  
  
func (r *Response) Display() string {  
    return r.Reply  
}
```

In the Toyota Chatbot, the `Response` struct encapsulates data related to chatbot responses, and methods such as `Display` can be used to define behavior related to this data. This struct serves as an "object" that combines data and behavior.

11.2.2 Polymorphism through Interfaces

Interfaces in Go allow different structs to implement common behavior, providing a form of polymorphism.

- **Interface Example -**

```
type Responder interface {  
    GetResponse(userMessage string) Response  
}
```

In the chatbot, we could define an interface `Responder` that different structs implement to generate chatbot responses. For example -

```
type SimpleResponder struct{}  
  
func (s SimpleResponder) GetResponse(userMessage string) Response {  
    return Response{Reply: "Simple response"}  
}
```

By defining multiple structs that implement the `Responder` interface, Go supports polymorphic behavior where different types share the same method signature (`GetResponse`), allowing them to be used interchangeably.

11.2.3 Composition over Inheritance

Go embraces composition over inheritance. Instead of class-based inheritance, Go allows embedding structs within other structs, promoting code reuse.

- **Example in Toyota Chatbot –**

```
type Response struct {  
    Reply    string `json:"reply"`  
    Buttons []Button `json:"buttons,omitempty"`  
}  
  
type Button struct {  
    Text  string `json:"text"`  
    Value string `json:"value"`  
}
```

The `Response` struct can include different buttons, leveraging composition to handle different chatbot behaviors based on the embedded struct, promoting modular and reusable code.

11.3 Functional Programming in Go

Go also supports functional programming paradigms, enabling developers to write functions that can be passed around as values and combined to create concise and expressive code. (*Effective Go - the Go Programming Language, n.d.-b*)

11.3.1 First-Class Functions

In Go, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.

- **Example -**

```
func processQuery(query string, handler func(string) Response)
Response {
    return handler(query)
}

response := processQuery("What are your hours?", func(q string)
Response {
    return Response{Reply: "Our hours are 8:30 AM to 5:30 PM."}
})
```

In the chatbot, this approach can be used to dynamically process user queries by passing handler functions, showcasing Go's functional capabilities.

11.3.2 Closures

Go supports closures, which are functions that reference variables outside their own scope. This allows for powerful programming techniques, such as state management within functions. (*Effective Go - the Go Programming Language, n.d.-b*)

- **Example -**

```
func makeResponder(name string) func(string) Response {
    return func(message string) Response {
        return Response{Reply: name + " says: " + message}
    }
}

chatbotResponder := makeResponder("ToyotaBot")
response := chatbotResponder("Hello!")
```

In this example, `makeResponder` returns a closure that maintains access to the `name` variable even after the function call has returned. This feature is useful for creating customizable response handlers in the chatbot.

11.4 Structured Programming in Go

Go's design is heavily based on structured programming principles, emphasizing the use of conditionals, loops, and modular functions to create readable and maintainable code.

11.4.1 Control Structures

Go supports all major control structures, including:

- **If-Else** - For conditional execution.
- **For Loops** - The only looping construct in Go, capable of representing traditional for, while, and do-while loops.
- **Switch Statements** - Used for multi-branch selection.
- **Example from Toyota Chatbot** –

```
switch userInput {
case "colombo":
    log.Println(V{ "Location 'colombo' detected"})
    return Response{Reply: "Toyota Lanka Colombo is located here: <a href='https://maps.app.goo.gl/tQfaZPHEuykjMoCAA' target='_blank'>Google Map Link</a>."}
case "kandy":
    log.Println(V{ "Location 'kandy' detected"})
    return Response{Reply: "Toyota Lanka Kandy is located here: <a href='https://maps.app.goo.gl/2UnfFbQEaEzMnCrn6' target='_blank'>Google Map Link</a>."}
case "galle":
    log.Println(V{ "Location 'galle' detected"})
    return Response{Reply: "Toyota Lanka Galle is located here: <a href='https://maps.app.goo.gl/8NUrpvjeCzf48fhx7' target='_blank'>Google Map Link</a>."}
}
```

This example demonstrates how the chatbot code uses structured programming principles to create a `switch` statement for handling user queries.

11.4.2 Modular Functions

Functions in Go are central to structured programming. They encapsulate logic and promote code reuse. In the chatbot code, functions are used to manage different components, such as user query handling and database operations. (*Effective Go - the Go Programming Language, n.d.-b*)

- **Example from Toyota Chatbot** –

```
// Connect to the database
func connectDB() (*sql.DB, error) {
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open( driverName: "mysql", dsn)
    if err != nil {
        log.Printf( format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf( format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println( a...: "Database connection successful!")
    return db, nil
}
```

The `connectDB` function encapsulates the logic for database connection setup, following structured programming principles to modularize and manage different

aspects of the chatbot application. (Datasource Package - github.com/grafana/grafana-plugin-sdk-go/backend/datasource - Go Packages, n.d.)

11.5 Comparison of Paradigms in the Toyota Chatbot

The Toyota Chatbot (`main.go`) combines structured, object-oriented programming paradigms to create a robust and flexible chatbot system -

- **Structured** - Control structures and modular functions manage the chatbot's logic flow and structure the code for readability and maintainability.
- **Object Oriented** - Encapsulation through structs and polymorphism via interfaces promote modularity and code reuse, making the chatbot adaptable and scalable.

Chapter 12: Concurrency - Parallel Processing in Go

Concurrency is one of Go's standout features. Designed with simplicity and efficiency, Go provides built-in support for concurrent programming using goroutines and channels. Concurrency allows applications to handle multiple tasks at the same time, improving performance and responsiveness

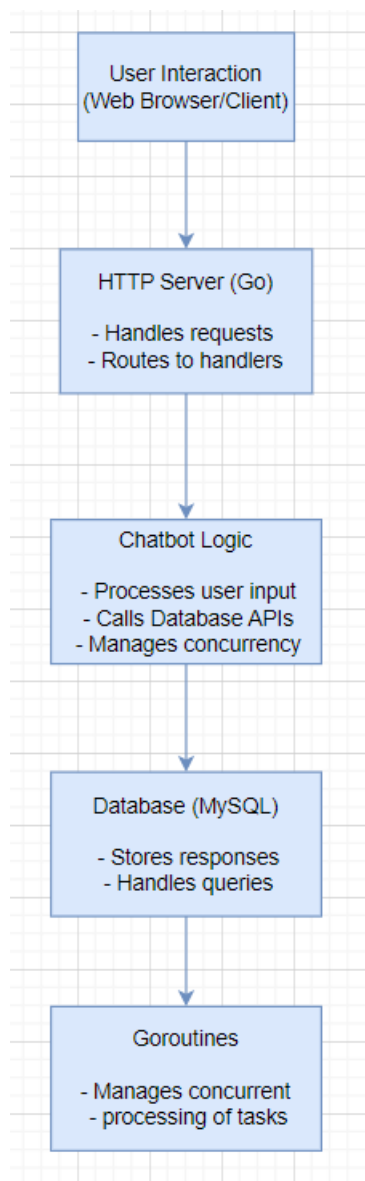
12.1 Introduction to Concurrency in Go

Concurrency refers to the ability of a program to execute multiple tasks simultaneously. Unlike parallelism, where tasks run in parallel on multiple processors, concurrency in Go allows tasks to interleave on a single processor, providing the illusion of simultaneous execution.

Go offers several mechanisms for handling concurrency:

- **Goroutines** - Lightweight threads managed by the Go runtime.
- **Channels** - Communication mechanisms for goroutines to safely exchange data.
- **Synchronization Primitives**: Mechanisms like mutexes and wait groups for coordinating goroutines.

12.2 System Architecture Diagram



12.3 Goroutines

A goroutine is a function or method that executes concurrently with other goroutines. Goroutines are extremely lightweight; the Go runtime can manage thousands of them simultaneously, making them ideal for concurrent programming.

12.3.1 Example Usage in Toyota Chatbot

In the Toyota Chatbot, we can use goroutines to handle multiple user requests concurrently. When a user sends a query, a new goroutine can be launched to process the request, allowing the chatbot to handle other queries simultaneously.

- **Example Code -**

```
func handleUserQuery(w http.ResponseWriter, r *http.Request) {
    go func() {
        userMessage := r.URL.Query().Get("message")
        response := processMessage(userMessage)
        w.Write([]byte(response))
    }()
}
```

In this example, a new goroutine (`go func() { ... }()`) is used to handle each user query. This allows the server to process multiple queries concurrently without blocking, improving response times and scalability.

12.4 Channels

Channels are the primary mechanism for communication between goroutines in Go. They provide a safe way to send and receive values between concurrently executing goroutines, ensuring data integrity and synchronization. (*Security Best Practices for Go Developers - the Go Programming Language*, n.d.)

12.4.1 Example of Channel Usage in the Chatbot

Channels are useful for coordinating tasks between goroutines, such as processing multiple queries simultaneously. Suppose the Toyota Chatbot needs to perform several database operations (e.g., fetching user information or logging queries) in parallel; channels can coordinate these operations.

- **Example Code -**

```
func fetchUserInfo(db *sql.DB, userID string, ch chan<- string) {
    userInfo := getUserFromDB(db, userID) // Hypothetical function
    ch <- userInfo // Send the result to the channel
}

func main() {
    db, _ := connectDB()
    userChannel := make(chan string)
    go fetchUserInfo(db, "user123", userChannel)

    userInfo := <-userChannel // Receive the value from the channel
    fmt.Println("User Info:", userInfo)
}
```

In this example, the `fetchUserInfo` function fetches user information from the database in a goroutine and sends the result back via the channel `userChannel`. The main function waits for and retrieves the result, ensuring synchronized data access between the goroutine and the main execution flow. (*Effective Go - the Go Programming Language*, n.d.-b)

12.5 Synchronization Primitives

Go also provides synchronization primitives, such as wait groups and mutexes, for coordinating and managing goroutines more explicitly.

12.5.1 Using Wait Groups in the Chatbot

Wait groups are useful when you need to wait for multiple goroutines to complete before proceeding. For example, in the Toyota Chatbot, you might want to handle multiple user requests concurrently but wait for all of them to finish before shutting down the server or logging a summary.

- **Example Code -**

```
func handleRequests(wg *sync.WaitGroup, requestID int) {
    defer wg.Done() // Decrement the counter when the goroutine
    completes
    fmt.Printf("Handling request %d\n", requestID)
}

func main() {
    var wg sync.WaitGroup
    for i := 1; i <= 5; i++ {
        wg.Add(1) // Increment the counter
        go handleRequests(&wg, i)
    }
    wg.Wait() // Block until all goroutines have finished
    fmt.Println("All requests processed")
}
```

In this code, a `WaitGroup` is used to wait for five goroutines (`handleRequests`) to complete before continuing execution. This is useful in scenarios where you want to ensure that multiple tasks (e.g., logging, database updates) are completed before proceeding.

12.6 Parallelism in Go

While Go primarily focuses on concurrency, it also supports parallelism, where multiple goroutines can run simultaneously on separate CPU cores. This is automatically managed by the Go runtime, which schedules goroutines across available processors.

12.6.1 Leveraging Parallelism in the Chatbot

If the Toyota Chatbot is deployed on a multi-core machine, Go's runtime will distribute goroutines across CPU cores, enabling true parallelism. This can be particularly useful when performing compute-intensive tasks, such as processing natural language or querying large datasets from a database. (Datasource Package - github.com/grafana/grafana-plugin-sdk-go/backend/datasource - Go Packages, n.d.)

To maximize parallelism, you can set the number of threads the Go runtime uses by adjusting GOMAXPROCS -

```
func main() {  
    runtime.GOMAXPROCS(runtime.NumCPU()) // Set to the number of available  
    CPU cores  
  
}
```

This setting ensures that the Go runtime makes full use of all available cores, boosting performance when handling multiple concurrent user queries.

12.7 Patterns for Concurrent Programming in Go

12.7.1 Worker Pools

A worker pool is a common pattern for managing multiple tasks concurrently. It allows the creation of a pool of workers (goroutines) that process tasks from a shared channel.

- **Example Code for Worker Pool -**

```
func worker(id int, jobs <-chan string, results chan<- string) {  
    for job := range jobs {  
        result := processQuery(job)  
        fmt.Printf("Worker %d processed job: %s\n", id, job)  
        results <- result  
    }  
}  
  
func main() {  
    jobs := make(chan string, 100)  
    results := make(chan string, 100)  
  
    for w := 1; w <= 3; w++ {  
        go worker(w, jobs, results)  
    }  
  
    for j := 1; j <= 5; j++ {  
        jobs <- fmt.Sprintf("query-%d", j)  
    }  
    close(jobs)  
  
    for a := 1; a <= 5; a++ {  
        <-results  
    }  
}
```

In this example, a pool of three workers is created to process five jobs concurrently. The worker pool pattern is beneficial in the chatbot for efficiently handling multiple user queries or database transactions simultaneously.

12.8 Advantages of Go's Concurrency Model in the Toyota Chatbot

- **Scalability** - Goroutines are lightweight and consume minimal resources, allowing the chatbot to scale and handle a large number of concurrent users efficiently.
- **Simplicity** - Go's concurrency primitives, such as goroutines and channels, provide a simple and intuitive way to implement concurrency without the complexity of traditional threading models.
- **Responsiveness** - By processing user queries in separate goroutines, the chatbot remains responsive, improving user experience.

12.9 Limitations and Considerations

While Go's concurrency model is powerful, it has limitations -

- **Race Conditions** - If shared resources are accessed without proper synchronization, race conditions can occur. Developers must use mutexes or channels carefully to manage shared state.
- **Complex Debugging** - Concurrency bugs, such as deadlocks and data races, can be challenging to debug. Go provides tools like the race detector (`go run -race`) to help identify these issues.

Chapter 13 - Exception Handling and Event Handling in Go

Go does not provide traditional exception handling mechanisms like `trycatch` blocks found in other programming languages such as Java or Python. Instead, Go emphasizes simplicity and control flow management using error values and other patterns like `panic` and `recover`. Event handling, on the other hand, is often managed using channels and goroutines, reflecting Go's design philosophy of concurrency and communication.

13.1 Error Handling in Go

Go's approach to error handling is explicit and simple: functions that might fail return an additional value, typically of the type `error`. This value is checked at the call site, and the program flow is adjusted accordingly. This method ensures that errors are visible and handled in a controlled manner.

13.1.1 Error Values

In Go, errors are represented as values of the built-in `error` type. A common practice is to return an error as the last value in a function. For instance, in the Toyota Chatbot, we often

check for errors when connecting to the database or processing user requests. (Datasource Package - [github.com/grafana/grafana-plugin-sdk-go/backend/datasource](https://github.com/grafana/grafana-plugin-sdk-go/blob/main/backend/datasource) - Go Packages, n.d.)

- **Example Code from Toyota chatbot –**

```
func connectDB() (*sql.DB, error) {
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open(driverName: "mysql", dsn)
    if err != nil {
        log.Printf(format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf(format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println(a...: "Database connection successful!")
    return db, nil
}
```

In this code -

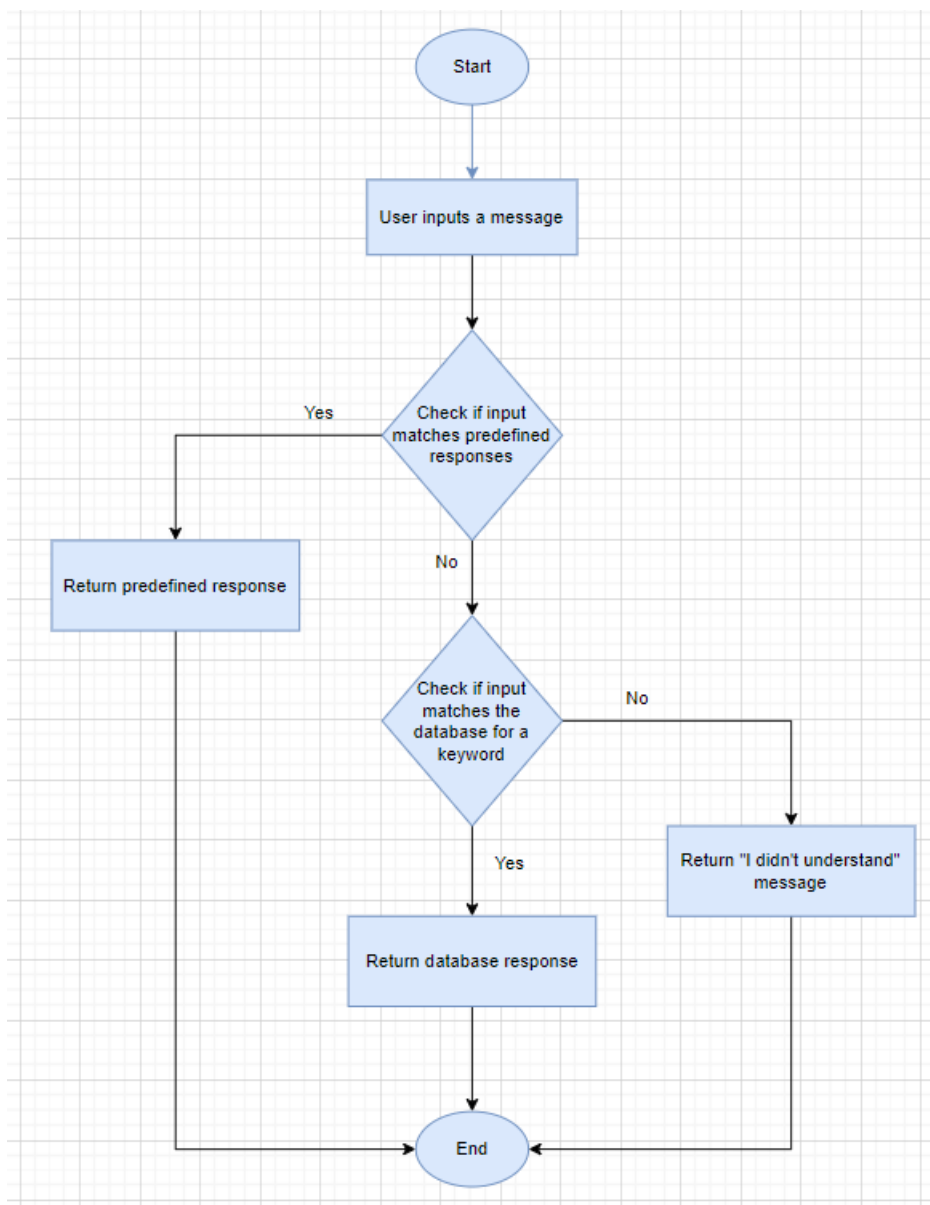
- The `connectDB` function returns an error if the database connection fails.
- The `main` function checks the returned error and logs it if the connection attempt is unsuccessful, gracefully handling the issue.

13.2 Panic and Recover

In addition to error values, Go provides `panic` and `recover` functions for handling unexpected situations -

- **Panic** - Used to stop the normal execution of a goroutine when something goes wrong (like throwing an exception in other languages).
- **Recover** - Used to regain control and recover from a panicking state, allowing the program to gracefully handle the error and continue execution.

13.2.1 Example Usage in the Toyota Chatbot



In the chatbot, we might use `panic` and `recover` to handle unexpected failures, such as a critical database issue or a configuration error.

- **Example Code -**

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            log.Printf("Recovered from panic: %v", r)
        }
    }()

    db, err := connectDB()
    if err != nil {
        panic("Database connection failed!") // Trigger a panic if
        the connection fails
    }
}
```

```

        // Normal chatbot operation continues here
    }

```

In this example -

- The `defer` statement with an anonymous function check for panics using `recover`. If a panic occurs, it logs the error message and allows the program to continue or exit gracefully.
- The `panic` is triggered when the database connection fails, demonstrating how critical errors are handled when they are not recoverable through simple error checks.

13.3 Event Handling in Go

Event handling in Go is typically managed through goroutines and channels, leveraging Go's concurrency model. In the Toyota Chatbot, events such as incoming user messages, database updates, or user interactions can be processed concurrently using these mechanisms.

13.3.1 Channels for Event Handling

Channels provide a safe and efficient way to handle events. For instance, when a new user message is received, the chatbot processes it asynchronously using goroutines and channels.

- **Example Code -**

```

func handleUserQuery(userMessage string, db *sql.DB, responseChannel
chan<- string) {
    response := processMessage(userMessage, db)
    responseChannel <- response // Send the response back via the
channel
}

func main() {
    db, _ := connectDB()
    responseChannel := make(chan string)

    go handleUserQuery("What are your opening hours?", db,
responseChannel)

    select {
    case response := <-responseChannel:
        fmt.Println("Response from chatbot:", response)
    }
}

```

In this code -

- The `handleUserQuery` function runs as a goroutine, processing user messages concurrently.

- The response is sent back using a channel (`responseChannel`), demonstrating how events (user messages) are handled efficiently without blocking the main program flow.

13.4 Patterns and Best Practices in Error Handling

Error Scenario	Description	Handling Approach	Example Code Snippet
Database Connection Failure	Occurs when the chatbot cannot connect to the MySQL database	Attempts to reconnect or sends an error message to the user	<pre>if err != nil { log.Println("Database connection error") }</pre>
Invalid User Input	When user input does not match predefined commands or is empty	Provides a default error message and prompts the user for valid input	<pre>return Response{Reply: "I'm sorry, I didn't understand that."}</pre>
HTTP Request Method Not Allowed	If the chatbot receives an unsupported HTTP method (e.g., GET instead of POST)	Sends an HTTP status code 405 and error message	<pre>http.Error(w, "Invalid request method", http.StatusMethodNotAllowed)</pre>
JSON Parsing Error	When parsing incoming JSON fails due to malformed data	Logs the error and sends a response indicating input error	<pre>json.NewDecoder(r.Body).Decode(&input) with error handling logic</pre>

Go encourages a straightforward approach to error handling with several best practices:

- **Explicit Error Handling** - Always check and handle errors immediately after they occur.
- **Returning Errors** - Functions should return errors instead of panicking unless the error is unrecoverable.

13.4.1 Applying Best Practices in Toyota Chatbot

- **Example Code -**

```
func connectDB() (*sql.DB, error) {
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open(driverName: "mysql", dsn)
    if err != nil {
        log.Printf(format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf(format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println(a...: "Database connection successful!")
    return db, nil
}
```

This code demonstrates:

- Proper handling of errors when opening and pinging the database connection, ensuring that issues such as connection failures are addressed explicitly and logged for debugging.
- The function `connectDB` returns an error if the database connection cannot be established or if the ping operation fails, following Go's explicit error handling practice.
- Proper Handling of Errors: The function uses `log.Printf` to log any error that occurs during the `sql.Open` or `db.Ping` operations, providing clear information for debugging.
- Explicit Error Returning: The function returns `nil` for the database connection and propagates the error to the caller, ensuring that the calling function can handle it appropriately.
- Graceful Failure: If any error occurs, the function returns immediately, preventing further operations on an invalid or uninitialized database connection.

Chapter 14 - Comparison with Similar Languages

14.1 Comparison with OOP Languages

Aspect	Go (Golang)	Java	Python	C++
Type System	Statically typed	Statically typed	Dynamically typed	Statically typed
Compilation	Compiled to native code	Compiled to bytecode (JVM)	Interpreted	Compiled to native code
Object-Oriented Support	Partial support (no traditional classes)	Full support (classes, inheritance, polymorphism)	Full support (dynamic typing, classes, inheritance)	Full support (multiple inheritance, classes)
Error Handling	Uses error values and panic/recover	Uses exceptions (try-catch blocks)	Uses exceptions (try-except blocks)	Uses exceptions (try-catch blocks)
Concurrency Model	Goroutines and channels (lightweight threads)	Thread-based (synchronized methods, threads)	Threading module and async I/O	Thread-based (using OS-level threads)
Memory Management	Automatic garbage collection	Automatic garbage collection (JVM)	Automatic garbage collection	Manual (with support for smart pointers)
Inheritance	No traditional inheritance (composition-based)	Supports single inheritance, interfaces	Supports multiple inheritance and mixins	Supports multiple inheritance
Functional Programming	Supports functions as first-class citizens	Supports lambdas and functional interfaces	Supports lambdas and functional constructs	Supports functional programming (e.g., lambdas, function objects)
Syntax Simplicity	Simple syntax, influenced by C	Verbose syntax, object-oriented structure	Simple, easy-to-read syntax	Complex, closer to hardware-level operations
Cross-Platform Support	Cross-platform, compiles to machine code	JVM-dependent, runs on any JVM-supporting platform	Platform-independent (runs where Python is installed)	Platform-specific, needs recompilation
Standard Library	Extensive support for networking, concurrency	Extensive libraries for various applications	Rich libraries for multiple domains	Standard libraries, but often relies on third-party libraries (STL, Boost)

Use Cases	Best for cloud services, microservices, APIs	Best for enterprise applications, Android development	Best for scripting, data science, AI	Best for system-level programming, game development, high-performance apps
-----------	--	---	--------------------------------------	--

Explanation of Key Differences

- **Object-Oriented Support:**
 - Go uses a composition-based approach rather than traditional classes and inheritance, which is simpler and avoids the complexity of deep inheritance hierarchies seen in Java and C++. Python offers flexibility with dynamic typing and multiple inheritance, while Java enforces single inheritance but supports interfaces to achieve polymorphism.
- **Concurrency Model:**
 - Go's goroutines and channels offer a lightweight and efficient concurrency model that is simpler and more intuitive than traditional thread-based models seen in Java, Python, and C++. This makes Go particularly effective for building concurrent systems like the Toyota Chatbot.
- **Error Handling:**
 - Go's use of error values provides explicit error handling, unlike Java and Python, which use exceptions (`try-catch` or `try-except`). This makes error management straightforward but requires more manual handling.
- **Memory Management:**
 - While Go and Java both use automatic garbage collection, C++ requires manual memory management, and Python has garbage collection but with a different approach suited for scripting. Go's garbage collector is optimized for performance, which is critical in server applications like the Toyota Chatbot.
- **Inheritance:**
 - Go's lack of traditional inheritance simplifies codebases by encouraging composition, which is often more flexible. Java and Python, with their inheritance models, provide a different way to structure object-oriented systems, but this can sometimes lead to overly complex hierarchies.

14.2 Comparison with Suggested Languages

Feature	Go	Rust	Kotlin	Scala
Concurrency Model	Goroutines and channels for lightweight concurrency management.	Ownership and borrowing with threads, ensuring memory safety.	Coroutines, providing a structured concurrency model.	Actors and Futures, supporting asynchronous computations.
Memory Management	Garbage collection (GC) based. No manual memory management.	Ownership model with RAII (Resource Acquisition Is Initialization).	JVM-based garbage collection, memory managed by the JVM.	JVM-based garbage collection with automatic memory management.
Performance	High performance due to lightweight goroutines and optimized runtime.	Comparable to C/C++ with no GC overhead.	Fast, but slightly slower than Go and Rust due to JVM overhead.	Similar to Kotlin, JVM overhead affects performance.
Type System	Static typing, simple and minimalistic.	Static typing with strict type safety and zero-cost abstractions.	Static typing with null safety and expressive syntax.	Advanced static typing with pattern matching and type inference.
Ease of Learning	Simple syntax, easy to learn for beginners.	More complex due to ownership rules.	Beginner-friendly, especially for Java developers.	Complex syntax, steep learning curve for functional programming.

Chapter 15 - Readability Writability, Performance and Cost

15.1 Comparison chart with main OOP Languages

Aspect	Go (Golang)	Java	Python	C++
Readability	High readability with concise syntax; minimalistic design reduces boilerplate code	Moderate readability; verbose, object-oriented syntax	High readability; simple syntax similar to pseudocode	Moderate to low; complex syntax, especially with advanced features like templates
Writability	Easy to write with built-in tools and libraries; supports fast development for concurrent apps	Moderate; requires more setup for complex applications	Very easy to write; highly flexible and dynamic but can lead to less predictable code	High complexity; efficient but can be difficult to write, especially for beginners
Performance	High performance; compiled to native code; efficient concurrency with goroutines	High; JVM-compiled bytecode offers good performance; multi-threading support available	Moderate; interpreted and slower than compiled languages; not ideal for high-performance tasks	Very high; compiled to native code; offers direct memory access for maximum speed
Cost	Open-source; free to use; low development and maintenance costs due to simplicity and speed	Open-source (Oracle's JDK); maintenance cost can be high for large-scale enterprise apps	Open-source; low cost for initial development but may require additional libraries and frameworks	Open-source; initial development can be costly due to complexity; maintenance costs vary based on application size

Explanation of Key Differences

- **Readability:**
 - Go's minimalistic syntax and structure enhance readability, making it easy to learn and maintain. Python's simplicity also makes it highly readable, especially for new developers. Java's verbosity can reduce readability, while C++'s advanced features (like pointers and templates) often complicate code readability.
- **Writability:**

- Python excels in writability due to its flexibility and dynamic nature, making it an ideal choice for rapid prototyping. Go is also easy to write, especially for concurrent programs, due to its native support for goroutines and lightweight syntax. Java's writability is moderate due to its structured, object-oriented approach, which requires more code. C++, while powerful, is the most complex to write effectively.
- **Performance:**
 - C++ and go have the edge in performance, as both compile to native code, providing speed and efficiency. Go's goroutines add efficient concurrency, making it suitable for building fast, scalable services like chatbots. Java offers good performance with its JVM but is generally slower than go and C++ for native execution. Python's interpreted nature makes it less performant for high-speed requirements.
- **Cost:**
 - All the languages are open source, minimizing licensing costs. Go has low development and maintenance costs due to its simplicity and efficiency, especially in building scalable web services. Python also has low initial costs but may require additional frameworks. Java's maintenance costs can rise for enterprise-level applications. C++ may incur higher costs initially due to its complexity, but the maintenance costs depend on the application's structure and size.

15.2 Comparison Chart with Suggested Languages

Metric	Go	Rust	Kotlin	Scala
Compile Time	Fast compilation due to simple syntax and static analysis.	Slower due to complex borrow checker.	Moderate, as it runs on the JVM.	Generally slower, due to JVM and advanced type system.
Execution Speed	High, leveraging lightweight goroutines.	Very high, comparable to C/C++.	Moderate, JVM-related overhead.	Moderate, similar to Kotlin.
Memory Usage	Efficient, but may use more memory due to GC.	Low memory footprint with precise control.	Moderate memory usage, managed by JVM.	Similar to Kotlin, JVM-dependent.
Readability	High, simple and concise syntax.	Medium, can be complex for new users.	High, familiar syntax for Java developers.	Low to medium, complex syntax.
Writability	High, quick code writing due to straightforward syntax.	Medium, more effort due to ownership rules.	High, similar to Java but more concise.	Medium, requires knowledge of functional programming.
Cost (Development)	Low development cost due to simplicity.	Medium due to learning curve.	Low, easy transition for Java developers.	Medium to high, needs skilled developers.

15.3 Performance Analysis

Chapter 16 - Demonstration and Discussion of Implemented Code

This chapter presents a detailed demonstration and analysis of the implemented code for the Toyota Lanka Chatbot, built using Go language (Golang). It explains various components, structures, and functions utilized in the chatbot, highlighting Go's features and how they contribute to the chatbot's efficiency and performance.

16.1 Overview of the Code Structure

The chatbot's code is structured in a way that leverages Go's strengths, such as simplicity, concurrency, and built-in support for web development. The code is organized into several functions and data structures that handle different aspects of chatbot operation -

- **Database Connection** - Establishes a connection with the MySQL database for storing and retrieving responses.
- **Chatbot Initialization** - Sets up default responses and initializes the chatbot on user requests.
- **Response Handling** - Uses predefined responses and database queries to respond to user input.
- **HTTP Handlers** - Manages incoming requests and sends appropriate responses back to the client.

16.2 Detailed Breakdown of Code Components

16.2.1 Database Connection

The function `connectDB()` connects the chatbot to a MySQL database –

```
// Connect to the database
func connectDB() (*sql.DB, error) {
    dsn := "root:mysql@tcp(localhost:3306)/toyota_chatbot"
    db, err := sql.Open(driverName: "mysql", dsn)
    if err != nil {
        log.Printf(format: "Error while opening the database connection: %v\n", err)
        return nil, err
    }

    // Check if the database connection is alive
    err = db.Ping()
    if err != nil {
        log.Printf(format: "Error pinging the database: %v\n", err)
        return nil, err
    }

    fmt.Println(a... "Database connection successful!")
    return db, nil
}
```

- **Explanation** - This function establishes a connection using Go's standard library `database/sql` and the MySQL driver (`go-sql-driver/mysql`).
- **Example** - When the chatbot needs to retrieve a response based on user input, it uses this connection to access the database. This structure demonstrates Go's efficient handling of database connections with error checking.

16.2.2 HTTP Handlers and Request Handling

The chatbot uses HTTP handlers to manage incoming requests from the web client. For instance –

```
// Chat handler for POST requests
func chatbotHandler(w http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodPost {
        var input map[string]string
        err := json.NewDecoder(r.Body).Decode(&input)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        userMessage := strings.TrimSpace(strings.ToLower(input["message"]))
        fmt.Println(a...: "Received message:", userMessage)

        db, err := connectDB()
        if err != nil {
            http.Error(w, err.Error(), http.StatusInternalServerError)
            return
        }
        defer db.Close()

        // Now use handleUserQuery to process the user's message
        botResponse := handleUserQuery(userMessage, db)

        w.Header().Set(key: "Content-Type", value: "application/json")
        json.NewEncoder(w).Encode(botResponse)
    } else {
        http.Error(w, error: "Invalid request method", http.StatusMethodNotAllowed)
    }
}
}
```

- **Explanation** - This function listens for POST requests from the web client and processes user messages. It decodes the JSON payload, connects to the database, and then calls `handleUserQuery()` to generate an appropriate response.
- **Relevance** - This showcases Go's built-in support for web services using the `net/http` package, which allows rapid development of HTTP servers.

16.2.3 Response Handling

The `handleUserQuery` function processes user input and determines the appropriate response

—

```
// Function to handle user messages and return the appropriate response
func handleUserQuery(userMessage string, db *sql.DB) Response {
    // Normalize user input
    userMessage = strings.ToLower(strings.TrimSpace(userMessage))
    log.Println("Handling user query:", userMessage)

    // First, check vehicle details directly based on user message
    if response, exists := vehicleDetails[userMessage]; exists {
        log.Println("Found vehicle details for:", userMessage)
        return response
    }

    // Check for vehicle descriptions directly based on user message
    if description, exists := vehicleDescriptions[userMessage]; exists {
        log.Println("Found vehicle description for:", userMessage)
        return description
    }

    // Check predefined responses
    predefinedResponse := getChatResponse(userMessage, db)
    if predefinedResponse.Reply != "I'm sorry, I didn't understand that. Can you please ask me something else?" {
        log.Println("Found predefined response for:", userMessage)
        return predefinedResponse
    }

    // Query the database if no predefined or specific response was found
    log.Println("Querying database for:", userMessage)
    dbResponse, err := getResponseByKeyword(db, userMessage)
    if err == nil && dbResponse.Reply != "" {
        log.Println("Database response found for:", userMessage)
        return dbResponse
    }

    log.Println("No response found for:", userMessage) // Log unmatched query for future analysis
    return Response{Reply: "I'm sorry, I didn't understand that. Can you please ask me something else?"}
}
```

- **Explanation** - This function demonstrates how Go handles control structures like `if` statements to determine responses based on user input. It first checks predefined responses, then queries the database if necessary.
- **Example** - When a user asks for "car models," the function matches this input with predefined categories and returns a corresponding response.

16.2.4 Structs for Encapsulation

The chatbot uses structs like `Response` and `Button` to encapsulate response details –

```
type Response struct {
    Reply    string `json:"reply"`
    Buttons []Button `json:"buttons,omitempty"`
}
```

- **Explanation** - Go uses structs for grouping related data, similar to classes in object-oriented languages. The `Response` struct contains information about the chatbot's reply and any associated buttons for user interaction.

- **Relevance** - This struct-based approach emphasizes Go's use of lightweight encapsulation for organizing related data, making code simpler and more maintainable.

16.2.5 JSON Handling

JSON is a common format for web-based communication. The chatbot uses JSON encoding/decoding extensively throughout the functions -

```
json.NewDecoder(r.Body).Decode(&input)
json.NewEncoder(w).Encode(botResponse)
```

- **Explanation** - Go's `encoding/json` package allows easy encoding and decoding of JSON data, which is essential for handling HTTP requests and responses.
- **Example** - When the chatbot receives a message, it decodes the JSON data and processes it. The chatbot then encodes its response back into JSON format before sending it to the client.

16.3 Key Features Demonstrated

16.3.1 Concurrency with Goroutines

The Go language is designed with concurrency in mind. In a chatbot system, concurrency could be implemented to handle multiple user requests simultaneously, ensuring fast and responsive interactions. For instance -

```
func main() {
    http.Handle("/", http.FileServer(http.Dir(".")))
    http.HandleFunc("/chat", chatbotHandler)
    http.HandleFunc("/initChat", initChatHandler)
    fmt.Println(a... "Server is running on port 8081...")
    http.ListenAndServe(addr: ":8081", handler: nil)
}
```

- **Explanation** - The `main` function runs the HTTP server concurrently, allowing the chatbot to manage multiple interactions without blocking other operations.
- **Advantage** - This illustrates Go's native support for parallel processing, which is essential for building scalable applications like chatbots.

16.3.2 Error Handling

Go emphasizes explicit error handling, as seen in database operations and HTTP request handling -

```
if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
```

- **Explanation** - Each operation checks for errors, and if an error occurs, an appropriate HTTP status is returned. This systematic approach ensures the chatbot can handle errors gracefully and maintain reliability.

16.4 Discussion on Efficiency and Performance

The Go-based chatbot is highly efficient, thanks to the language's design:

- **Fast Compilation and Execution** - Go compiles directly to native machine code, providing performance benefits that are especially noticeable in real-time applications like chatbots. (*Effective Go - the Go Programming Language, n.d.-b*)
- **Memory Management** - With built-in garbage collection, Go ensures efficient memory usage, crucial for handling multiple chatbot interactions simultaneously.
- **Lightweight Concurrency** - The use of goroutines enables the chatbot to scale effectively, processing numerous requests with minimal overhead.

Chapter 17 - Conclusions

This chapter summarizes the findings and insights gained throughout the development and evaluation of the Toyota Lanka chatbot using Go language. It reviews the key strengths and limitations of the language in the context of the project, and highlights the suitability of Go for building scalable, efficient chatbot applications.

17.1 Summary of Findings

In this section, we encapsulate the major observations and outcomes derived from implementing and analysing the chatbot using Go.

- **Efficiency and Performance** - The Go language demonstrates exceptional performance in terms of speed and memory management, which are critical for real-time chatbot applications. Its ability to compile directly to machine code ensures quick execution, while lightweight goroutines support concurrent processing efficiently.
- **Ease of Development** - Go's simplicity and clean syntax contribute to a smooth development process. The language's comprehensive standard library supports building web servers, handling JSON, and managing databases seamlessly, reducing the need for additional dependencies.
- **Concurrency and Scalability** - By utilizing goroutines, the chatbot efficiently handles multiple user requests simultaneously without compromising performance. This feature showcases Go's concurrency model, which is a standout compared to other programming languages.
- **Error Handling** - Go's approach to error management, which emphasizes explicit handling and readability, allows for robust code that handles unexpected scenarios gracefully.

17.2 Evaluation of Language Strengths

The strengths of Go language highlighted in the chatbot's development are further summarized -

- **Simplified Syntax** - Go's syntax is straightforward, leading to improved code readability and maintainability. This is particularly advantageous in large-scale projects where team collaboration is essential.
- **Lightweight Concurrency** - The ability to spawn thousands of lightweight goroutines showcases Go's effectiveness in building scalable, high-performance applications such as chatbots.
- **Built-in Web Support** - Go's `net/http` package enables rapid development of HTTP servers, providing all the necessary tools to create a robust backend for web-based applications.

17.3 Limitations and Areas for Improvement

Despite its strengths, certain limitations were observed -

- **Lack of Native Object-Oriented Features** - Go's lack of traditional inheritance and its preference for composition over inheritance could be limiting for developers accustomed to other OOP languages like Java or C++.
- **Limited GUI Support** - Go is primarily designed for backend services and systems programming, so developing rich graphical interfaces natively would require additional frameworks or libraries.
- **Complex Error Handling** - While Go's error handling encourages explicit handling, it can also lead to repetitive code, which may reduce readability in large applications.

17.4 Suitability of Go for Chatbot Development

In evaluating Go's applicability for chatbot development, the language proved to be a highly suitable choice -

- **Scalability** - With its efficient concurrency model, Go can handle a high number of concurrent users, making it ideal for scaling chatbot services.
- **Performance** - The compiled nature of Go and its optimized runtime ensures that the chatbot remains responsive, even under heavy load.
- **Maintainability** - Go's clean syntax and structure facilitate maintainable codebases, which is beneficial for long-term project evolution.

17.5 Recommendations for Future Work

Based on the analysis, several recommendations are suggested for enhancing future chatbot projects -

- **Integration of Advanced Machine Learning Models** - Incorporating AI/ML capabilities for understanding natural language more effectively could further enhance user interactions.

- **Expansion of Database and API Usage** - Leveraging external APIs and expanding the database to include richer content would improve the chatbot's informational scope.
- **Optimization of Error Handling Mechanisms** - Streamlining error management in Go code could reduce redundancy and improve code readability, making it even easier for developers to extend the application.

17.6 Final Thoughts

The development of the Toyota Lanka chatbot using Go has proven that the language is a viable and efficient option for building web-based applications that require speed, scalability, and reliability. While Go's simplicity and concurrency model offer significant advantages, its limitations, such as the lack of traditional object-oriented programming constructs, need to be taken into consideration depending on the project requirements. Nonetheless, the chatbot project demonstrates Go's capacity to deliver modern, performant solutions, making it a valuable tool in a developer's arsenal.

Chapter 18 - References

Documentation - the GO programming language. (n.d.). <https://go.dev/doc/>

The GO programming language. (n.d.-b). <https://go.dev/src/go/token/token.go>

Effective Go - the Go programming language. (n.d.-b). https://go.dev/doc/effective_go

Go Wiki: GO Conferences and Major Events - the Go programming language. (n.d.).

<https://go.dev/wiki/Conferences>

The GO programming language. (n.d.-c). <https://go.dev/src/go/token/token.go>

Release History - the GO programming language. (n.d.). <https://go.dev/doc/devel/release>

Dy, A. (2023, May 18). Understanding Golang Object Oriented Programming (OOP)

with Examples. DEV Community. <https://dev.to/adriandy89/understanding-golang-object-oriented-programming-oop-with-examples-15l6>

The GO programming language. (n.d.-d). <https://go.dev/src/runtime/HACKING>

Parthlaw. (2024, April 27). Object-Oriented GO: Unraveling the power of OOP in Golang.

DEV Community. <https://dev.to/parthlaw/object-oriented-go-unraveling-the-power-of-oop-in-golang-49h6>

Tutorial: Developing a RESTful API with Go and Gin - The Go Programming Language.

(n.d.). <https://go.dev/doc/tutorial/web-service-gin>

Concurrency — An introduction to programming in Go | Go Resources. (n.d.).

<https://www.golang-book.com/books/intro/10>

testing package - testing - Go Packages. (n.d.). <https://pkg.go.dev/testing>

Whitney, J., Gifford, C., & Pantoja, M. (2018). Distributed execution of communicating sequential process-style concurrency: Golang case study. The Journal of Supercomputing, 75(3), 1396–1409. <https://doi.org/10.1007/s11227-018-2649-2>

Tutorial: Developing a RESTful API with Go and Gin - The Go Programming Language.

(n.d.). <https://go.dev/doc/tutorial/web-service-gin>

Practical go lessons. (n.d.). <https://www.practical-go-lessons.com/chap-13-types>

Zhao, J., Zhou, X., Chang, S., & Xu, C. (2023). Let It Go: Relieving Garbage Collection Pain for Latency Critical Applications in Golang. Let It Go: Relieving Garbage Collection Pain for Latency Critical Applications in Golang, 169–180.

<https://doi.org/10.1145/3588195.3592998>

rational package - github.com/alex-ant/gomath/rational - Go Packages. (n.d.).

<https://pkg.go.dev/github.com/alex-ant/gomath/rational>

Effendy, F., Taufik, N., & Adhilaksono, B. (2019). Performance comparison of web backend and database: A case study of Node.JS, Golang and MySQL, Mongo DB. Recent Advances in Computer Science and Communications, 14(6), 1955–1961.

<https://doi.org/10.2174/2666255813666191219104133>

Simplicity is Complicated. (n.d.). <https://go.dev/talks/2015/simplicity-is-complicated.slide#1>

Standard library - Go Packages. (n.d.). <https://pkg.go.dev/std>

Arundel, J. (2024, August 4). Functional programming in Go — Bitfield Consulting. Bitfield Consulting. <https://bitfieldconsulting.com/posts/functional>

Chapter 19 – Appendices

main.go – toyota_chatbot

```
1 package main
2
3 import (
4     "database/sql"
5     "encoding/json"
6     "fmt"
7     _ "github.com/go-sql-driver/mysql"
8     "log"
9     "net/http"
10    "strings"
11)
12
13 type Response struct {
14     Reply string `json:"reply"`
15     Buttons []Button `json:"buttons,omitempty"`
16 }
17
18 type Button struct {
19     Text string `json:"text"`
20     Value string `json:"value"`
21 }
22
23 // Default response when the user first visits
24 var defaultResponse = Response{
25     Reply: "Hi, I'm Toyota Lanka's Virtual Assistant. \n\n" +
26         "I can assist you with your queries with regard to our vehicle products and services. \n" +
27         "You may select from the following options or type your specific requirement.",
28     Buttons: []Button{
29         {Text: "View Car Models", Value: "car models"},
30         {Text: "Locate a Branch", Value: "branches"},
31         {Text: "Current Promotions", Value: "promotions"},
32     },
33 }
```

```
31     {Text: "Current Promotions", Value: "promotions"},
32     {Text: "Services & Maintenance", Value: "service"},
33     {Text: "Make a Payment", Value: "online payment"},
34 },
35 }
36
37 // Map of predefined responses
38 var responses = map[string]Response{
39     "hi": {
40         Reply: "Hi! How can I assist you today?",
41     },
42     "hello": {
43         Reply: "Hello! How can I assist you today?",
44     },
45     "hey": {
46         Reply: "Hey! How can I assist you today?",
47     },
48     "how are you": {
49         Reply: "I'm Toyota Lanka's Virtual Assistant, here to assist you with Toyota-related queries!",
50     },
51     "what is your name, who are you, name": {
52         Reply: "I'm the Toyota Lanka's Virtual Assistant! How can I assist you?",
53     },
54     "contact, helpline, contact number, contact information, how can I contact you": {
55         Reply: `
56         <div style='line-height: 1.5;'>
57             <strong>Tel:</strong> <a href='tel:+94112939080'+9411 293 9080 - 6</a><br>
58             <strong>Fax:</strong> <a href='tel:+94112939005'+9411 293 9005</a><br>
59             <strong>Email:</strong> <a href='mailto:info@toyota.lk'+info@toyota.lk</a>
60         </div>
61     `,
62     },
63 }
```

chatbotHandler(w http.ResponseWriter, r *http.Request)

stbot-toyota > main.go

365.65 LF UTF-8 Tab

```

61 },
62 },
63 "Email": {
64     Reply: "Email Us: <a href='mailto:info@toyota.lk'>info@toyota.lk</a>",
65 },
66 "opening hours, what are your opening hours, showrooms open time": {
67     Reply: "Our showrooms are open from 8:30 AM to 5:30 PM, Monday to Saturday.",
68 },
69 "main office, where is your main office located, headquarters location, address": {
70     Reply: "Our main office is located at 337, Nsagombe Road, Wattala 11500, Sri Lanka.",
71 },
72 "about": {
73     Reply: "Toyota Lanka is committed to quality, innovation, and customer satisfaction. More details can be found on our website: <a href='https://www.toyota.lk/about'>
74 },
75 "promotions, promotion": {
76     Reply: "Toyota Lanka offers several promotions and discounts for members and staff at several organizations. You can check out for more promotion details on our web
77 },
78 "news, newsroom, news room, article, articles, latest": {
79     Reply: "Check out the Latest Events and Occasions at the News Room. You can check out for news on our website: <a href='https://www.toyota.lk/newsroom' target='_bla
80 },
81 "branch, showroom, service center, dealerships, dealership, branches": {
82     Reply: "Locate your nearest branch using our website. You can locate your nearest branch on our website: <a href='https://www.toyota.lk/locate-a-branch' target='_bl
83 },
84 "online payment, payment": {
85     Reply: "Do your payments using our online payment facility through the website: <a href='https://payments.toyota.lk/' target='_blank'>Payment Portal</a>.",
86 },
87 "service, services, maintenance, repair": {
88     Reply: "Maintenance in servicing and repairs through Toyota Lanka. Check out for General Repairs and Periodic Maintenance: <a href='https://www.toyota.lk/maintenanc
89 },
90 "safety, safety sense": {
91     Reply: "Toyota has a long history of creating advancements and innovations, striving to produce vehicles that are safe and reliable: <a href='https://www.toyota.lk/
chatbotHandler(w http.ResponseWriter, r *http.Request)

```

```

92 },
93 },
94 "car models, models, cars, car, suv, pickup, bus, sedan, hatchback, vehicles, vehicle": {
95     Reply: "Please select a car model:",
96     Buttons: []Button{
97         {Text: "Compact SUV", Value: "compact suv"},
98         {Text: "SUV", Value: "suv"},
99         {Text: "Pickup & Bus", Value: "pickup & bus"},
100         {Text: "Sedan", Value: "sedan"},
101         {Text: "Hatchback", Value: "hatchback"},
102     },
103 },
104 "locations, location": {
105     Reply: "Please select a location:",
106     Buttons: []Button{
107         {Text: "Toyota Lanka Colombo", Value: "colombo"},
108         {Text: "Toyota Lanka Kandy", Value: "kandy"},
109         {Text: "Toyota Lanka Galle", Value: "galle"},
110     },
111 },
112 }
113
114 // Map of vehicle details
115 var vehicleDetails = map[string]Response{
116     "compact suv": {
117         Reply: "The Compact SUV is perfect for city driving and offers great fuel efficiency. Would you like more details? Please select a model:",
118         Buttons: []Button{
119             {Text: "Toyota RAV4", Value: "rav4"},
120             {Text: "Toyota Rush", Value: "rush"},
121         },
122     },
123 }
chatbotHandler(w http.ResponseWriter, r *http.Request)

```

```

121 },
122 },
123 "suv": {
124     Reply: "Our SUVs are designed for adventure and comfort. Interested in a specific model?:",
125     Buttons: []Button{
126         {Text: "Toyota Fortuner", Value: "fortuner"},
127         {Text: "Toyota Land Cruiser", Value: "land cruiser"},
128     },
129 },
130 "pickup & bus": {
131     Reply: "You have selected Pickup & Bus. Please select a model:",
132     Buttons: []Button{
133         {Text: "Toyota Hilux", Value: "hilux"},
134         {Text: "Toyota Coaster", Value: "coaster"},
135     },
136 },
137 "sedan": {
138     Reply: "You have selected Sedan. Please select a model:",
139     Buttons: []Button{
140         {Text: "Toyota Camry", Value: "camry"},
141         {Text: "Toyota Corolla", Value: "corolla"},
142     },
143 },
144 "hatchback": {
145     Reply: "You have selected Hatchback. Please select a model:",
146     Buttons: []Button{
147         {Text: "Toyota Yaris", Value: "yaris"},
148         {Text: "Toyota Prius", Value: "prius"},
149     },
150 },
151 }
chatbotHandler(w http.ResponseWriter, r *http.Request)

```

```

153 // Map of vehicle descriptions
154 var vehicleDescriptions = map[string]Response{
155     "rav4": {
156         Reply: "The Toyota RAV4 is a versatile compact SUV that offers an adventurous spirit with comfort and style. For more information, visit: <a href='https://www.toyota.lk/vehicle-rav4'>https://www.toyota.lk/vehicle-rav4</a>."
157     },
158     "rush": {
159         Reply: "The Toyota Rush is a dynamic compact family SUV with 7 seats that combines bold styling with excellent fuel efficiency. For more information, visit: <a href='https://www.toyota.lk/vehicle-rush'>https://www.toyota.lk/vehicle-rush</a>."
160     },
161     "fortuner": {
162         Reply: "The Toyota Fortuner is a tough and reliable SUV built for both urban and off-road adventures. For more information, visit: <a href='https://www.toyota.lk/vehicle-fortuner'>https://www.toyota.lk/vehicle-fortuner</a>."
163     },
164     "land cruiser": {
165         Reply: "The Toyota Land Cruiser is a legendary SUV known for its off-road capabilities and luxury features. For more information, visit: <a href='https://www.toyota.lk/vehicle-land-cruiser'>https://www.toyota.lk/vehicle-land-cruiser</a>."
166     },
167     "hilux": {
168         Reply: "The Toyota Hilux is a rugged pickup designed for heavy-duty performance and durability. For more information, visit: <a href='https://www.toyota.lk/vehicle-hilux'>https://www.toyota.lk/vehicle-hilux</a>."
169     },
170     "coaster": {
171         Reply: "The Toyota Coaster is an ideal bus for transporting passengers with comfort and reliability. For more information, visit: <a href='https://www.toyota.lk/vehicle-coaster'>https://www.toyota.lk/vehicle-coaster</a>."
172     },
173     "camry": {
174         Reply: "The Toyota Camry is a premium sedan that offers an upscale experience with advanced safety features. For more information, visit: <a href='https://www.toyota.lk/vehicle-camry'>https://www.toyota.lk/vehicle-camry</a>."
175     },
176     "corolla": {
177         Reply: "The Toyota Corolla is a compact sedan that stands out for its reliability and efficiency. For more information, visit: <a href='https://www.toyota.lk/vehicle-corolla'>https://www.toyota.lk/vehicle-corolla</a>."
178     },
179     "yaris": {
180         Reply: "The Toyota Yaris is a compact hatchback that offers great maneuverability and efficiency. For more information, visit: <a href='https://www.toyota.lk/vehicle-yaris'>https://www.toyota.lk/vehicle-yaris</a>."
181     },
182     "prius": {
183         Reply: "The Toyota Prius is a hybrid hatchback that represents eco-friendliness and efficiency. For more information, visit: <a href='https://www.toyota.lk/vehicle-prius'>https://www.toyota.lk/vehicle-prius</a>."
184     }
185 }
186
187 chatbotHandler(w http.ResponseWriter, r *http.Request)
188
189 //bot-toyota > main.go

```

Database Connection

```

186
187 // Connect to the database
188 func connectDB() (*sql.DB, error) {
189     dsn := "root:mysql@localhost(3306)/toyota_chatbot"
190     db, err := sql.Open("driverName", "mysql", dsn)
191     if err != nil {
192         log.Printf("Error while opening the database connection: %v\n", err)
193         return nil, err
194     }
195
196     // Check if the database connection is alive
197     err = db.Ping()
198     if err != nil {
199         log.Printf("Error pinging the database: %v\n", err)
200         return nil, err
201     }
202
203     fmt.Println("Database connection successful!")
204     return db, nil
205 }

```

handleUserQuery Function

```

207 // Function to handle user messages and return the appropriate response
208 func handleUserQuery(userMessage string, db *sql.DB) Response {
209     // Normalize user input
210     userMessage = strings.ToLower(strings.TrimSpace(userMessage))
211     log.Println("Handling user query:", userMessage)
212
213     // First, check vehicle details directly based on user message
214     if response, exists := vehicleDetails[userMessage]; exists {
215         log.Println("Found vehicle details for:", userMessage)
216         return response
217     }
218
219     // Check for vehicle descriptions directly based on user message
220     if description, exists := vehicleDescriptions[userMessage]; exists {
221         log.Println("Found vehicle description for:", userMessage)
222         return description
223     }
224
225     // Check predefined responses
226     predefinedResponse := getChatResponse(userMessage, db)
227     if predefinedResponse.Reply != "I'm sorry, I didn't understand that. Can you please ask me something else?" {
228         log.Println("Found predefined response for:", userMessage)
229         return predefinedResponse
230     }
231
232     // Query the database if no predefined or specific response was found
233     log.Println("Querying database for:", userMessage)
234     dbResponse, err := getResponseByKeyword(db, userMessage)
235     if err == nil && dbResponse.Reply != "" {
236         log.Println("Database response found for:", userMessage)
237         return dbResponse
238     }
239
240     chatbotHandler(w http.ResponseWriter, r *http.Request)

```

```

214     if response, exists := vehicleDetails[userMessage]; exists {
215         log.Println(ℳ "Found vehicle details for:", userMessage)
216         return response
217     }
218
219     // Check for vehicle descriptions directly based on user message
220     if description, exists := vehicleDescriptions[userMessage]; exists {
221         log.Println(ℳ "Found vehicle description for:", userMessage)
222         return description
223     }
224
225     // Check predefined responses
226     predefinedResponse := getChatResponse(userMessage, db)
227     if predefinedResponse.Reply != "I'm sorry, I didn't understand that. Can you please ask me something else?" {
228         log.Println(ℳ "Found predefined response for:", userMessage)
229         return predefinedResponse
230     }
231
232     // Query the database if no predefined or specific response was found
233     log.Println(ℳ "Querying database for:", userMessage)
234     dbResponse, err := getResponseByKeyword(db, userMessage)
235     if err == nil && dbResponse.Reply != "" {
236         log.Println(ℳ "Database response found for:", userMessage)
237         return dbResponse
238     }
239
240     log.Println(ℳ "No response found for:", userMessage) // Log unmatched query for future analysis
241     return Response{Reply: "I'm sorry, I didn't understand that. Can you please ask me something else?"}
242 }
243
chatbotHandler(w http.ResponseWriter, r *http.Request)

```

getResponseByKeyword

```

243
244 func getResponseByKeyword(db *sql.DB, keyword string) (Response, error) {
245     var reply string
246     var buttons string
247
248     log.Println(ℳ "Attempting to retrieve response for keyword:", keyword)
249
250     // Match against the keyword in the 'responses' table
251     query := "SELECT reply, buttons FROM responses WHERE keywords LIKE ?"
252     err := db.QueryRow(query, ℳ "%"+keyword+"%").Scan(&reply, &buttons)
253     if err != nil {
254         log.Println(ℳ "Query error:", err)
255         return Response{}, err
256     }
257
258     // Process buttons if present
259     var buttonList []Button
260     if buttons != "" {
261         err = json.Unmarshal([]byte(buttons), &buttonList)
262         if err != nil {
263             log.Println(ℳ "JSON unmarshal error:", err)
264             return Response{}, err
265         }
266     }
267
268     log.Println(ℳ "Database reply found:", reply)
269     return Response{Reply: reply, Buttons: buttonList}, nil
270 }

```

matchesKeyword

```
272 func matchesKeyword(userInput, keyword string) bool {
273     words := strings.Fields(userInput) // Split user input into words
274     for _, word := range words {
275         // Check for exact match
276         if word == keyword {
277             return true
278         }
279         // Check plural/singular match
280         if word == keyword+"s" || (strings.HasSuffix(keyword, "s") && word == keyword[:len(keyword)-1]) {
281             return true
282         }
283     }
284     return false
285 }
```

getChatResponse

```
287 func getChatResponse(userInput string, db *sql.DB) Response {
288     userInput = strings.ToLower(strings.TrimSpace(userInput))
289     log.Println("Checking predefined responses for:", userInput)
290
291     // Split user input into words for better matching
292     userWords := strings.Fields(userInput)
293
294     // Iterate over predefined responses and look for a match
295     for keywords, response := range responses {
296         // Split the keywords string into individual keywords
297         for _, keyword := range strings.Split(keywords, "|") {
298             for _, word := range userWords {
299                 // If any word in the user input matches a keyword, return the response
300                 if strings.EqualFold(word, keyword) {
301                     log.Println("Keyword match found for:", keyword)
302                     return response
303                 }
304             }
305         }
306     }
307
308     log.Println("No predefined response matched for:", userInput)
309
310     // Check vehicle categories based on keywords in the user's message
311     for category, details := range vehicleDetails {
312         if strings.Contains(userInput, category) {
313             log.Println("Vehicle category detected:", userInput)
314             return details
315         }
316     }
317 }
```

```
318 // Check vehicle models based on keywords in the user's message
319 for model, description := range vehicleDescriptions {
320     if strings.Contains(userInput, model) {
321         log.Println("Vehicle model detected:", userInput)
322         return description
323     }
324 }
325
326 // Check for location responses
327 switch userInput {
328 case "colombo":
329     log.Println("Location 'colombo' detected")
330     return Response(Reply: "Toyota Lanka Colombo is located here: <a href='https://maps.app.goo.gl/tQfaZPHEuykMoCAA' target='_blank'>Google Map Link</a>.")
331 case "kandy":
332     log.Println("Location 'kandy' detected")
333     return Response(Reply: "Toyota Lanka Kandy is located here: <a href='https://maps.app.goo.gl/2UnffBQEnEzMcnn6' target='_blank'>Google Map Link</a>.")
334 case "galle":
335     log.Println("Location 'galle' detected")
336     return Response(Reply: "Toyota Lanka Galle is located here: <a href='https://maps.app.goo.gl/BNUrvvjeCzf48fhx7' target='_blank'>Google Map Link</a>.")
337 }
338
339 log.Println("No location matched for:", userInput)
340
341 // If no response was found, return a default message
342 return Response(Reply: "I'm sorry, I didn't understand that. Can you please ask me something else?")
343 }
```

chatbotHandler

```
345 // Chat handler for POST requests
346 func chatbotHandler(w http.ResponseWriter, r *http.Request) {
347     if r.Method == http.MethodPost {
348         var input map[string]string
349         err := json.NewDecoder(r.Body).Decode(&input)
350         if err != nil {
351             http.Error(w, err.Error(), http.StatusBadRequest)
352             return
353         }
354
355         userMessage := strings.TrimSpace(strings.ToLower(input["message"]))
356         fmt.Println("Received message:", userMessage)
357
358         db, err := connectDB()
359         if err != nil {
360             http.Error(w, err.Error(), http.StatusInternalServerError)
361             return
362         }
363         defer db.Close()
364
365         // Now use handleUserQuery to process the user's message
366         botResponse := handleUserQuery(userMessage, db)
367
368         w.Header().Set("Content-Type", "application/json")
369         json.NewEncoder(w).Encode(botResponse)
370     } else {
371         http.Error(w, "Invalid request method", http.StatusMethodNotAllowed)
372     }
373 }
```

initChatHandler and main

```
375 // New handler for GET requests to initialize the chat with a greeting
376 func initChatHandler(w http.ResponseWriter, r *http.Request) {
377     if r.Method == http.MethodGet {
378         w.Header().Set("Content-Type", "application/json")
379         json.NewEncoder(w).Encode(defaultResponse)
380     } else {
381         http.Error(w, "Invalid request method", http.StatusMethodNotAllowed)
382     }
383 }
384
385 func main() {
386     http.Handle("/", http.FileServer(http.Dir(".")))
387     http.HandleFunc("/chat", chatbotHandler)
388     http.HandleFunc("/initChat", initChatHandler)
389     fmt.Println("Server is running on port 8081...")
390     http.ListenAndServe(":8081", handler)
391 }
```