Certainly! Let's expand on the Day 2 Golang learning module, including more details for each learning outcome:

1. Learning Outcomes

1.1 Understand how applications connect to the server via HTTP and TCP

- **Explanation:**
  - Introduce the fundamentals of networking in Go.
  - Discuss how applications establish connections over HTTP and TCP.
  - Explain the role of protocols and how Go simplifies network programming.
- **Activities:**
  - Create a simple HTTP server in Go.
  - Develop a basic TCP server to handle client connections.
  - Explore tools like `curl` or Postman to interact with the servers.

1.2 Understand the concept of writer and response for HTTP

- **Explanation:**
  - Explain the Request-Response cycle in an HTTP server.
  - Discuss the role of the `http.ResponseWriter` and `http.Request` in handling HTTP requests and responses.
  - Emphasize the importance of status codes, headers, and bodies in responses.
- **Activities:**
  - Implement handlers to respond to different HTTP methods (GET, POST, etc.).
  - Customize HTTP responses by setting headers and writing to the response body.

1.3 Ability to grasp simple RESTful API concepts

- **Explanation:**
  - Define RESTful API principles and their importance.
  - Discuss resource identification, representation, statelessness, and uniform interface.
  - Introduce Go packages and tools for building RESTful APIs.
- **Activities:**
  - Design and implement a simple RESTful API using Go.
  - Create endpoints for CRUD operations on resources.
  - Demonstrate handling different HTTP methods and response codes.

1.4 Database Interaction with SQLite

- **Explanation:**

    - Introduce the concept of database interaction in Go.
    - Focus on SQLite as a lightweight embedded database.
    - Discuss basic CRUD operations and database/sql package in Go.

- **Activities**:

    - Connect a Go application to an SQLite database.
    - Perform CRUD operations using the database/sql package.
    - Integrate database interactions into the RESTful API developed using the Chi router.

General Tips for Day 2 Facilitation:

- **Hands-On Exercises:** Continue incorporating hands-on exercises for each learning outcome to reinforce theoretical knowledge.
- **Real-world Examples:** Illustrate concepts with real-world examples of networking, HTTP servers, and RESTful API implementations in Go.
- **Peer Collaboration:** Encourage participants to collaborate and discuss their solutions during activities, fostering a collaborative learning environment.
- **Q&A Sessions:** Allocate time for questions and answers to address any confusion or concerns related to networking, HTTP, and API concepts.
- **Resources for Further Learning:** Provide additional resources and references for participants who want to explore topics in more depth, such as Go's concurrency patterns.
- **Feedback Loop:** Maintain a feedback loop for participants to share their thoughts on the learning materials and session structure.

Adapt the module based on participants' feedback and the pace of the group. The goal is to create an engaging and supportive learning environment for everyone involved in the Golang workshop.

Understanding how applications connect to the server via HTTP and TCP

Introduce the fundamentals of networking in Go

Applications communicate with each other over a network. A network is a group of computers connected to each other that can exchange data. The Internet is the largest network in the world, connecting billions of computers worldwide. The application that initiates the communication is called the client, and the application that receives the communication is called the server. The client and server communicate using a protocol, which is a set of rules that governs how data is exchanged between applications. The client and server must agree on the protocol to use for communication. That protocol is the HyperText Transfer Protocol (HTTP), which is the foundation of data communication on the Internet. HTTP is a request-response protocol, which means that the client sends a request to the server, and the server responds to the request. The client and server communicate over a Transmission Control Protocol (TCP) connection, which is a connection-oriented protocol that ensures reliable communication between the client and server.

This is a simplified overview of how applications communicate over a network. There are many other protocols and concepts involved in networking, but this is enough to get started with Go.

From the client's browser, this is what happens when we navigate to a website:

```
GET / HTTP/1.1
Host: www.example.com
```

The client sends a GET request to the server. The GET request is a request for data from the server. The server responds with a response code, which indicates whether the request was successful or not. The response code is followed by the response body, which contains the requested data.

This is an example of a response from the server:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 13

Hello, World!
```

The response has three parts: the response code, the response headers, and the response body. The response code is for indicating whether the request was successful or not.

The response headers are for providing additional information about the response, such as the content type and length. Example of content types include text/html, which is used for HTML documents, and application/json, which is used for JSON data. The response body contains the requested data, which is "Hello, World!" in this case.

HTTP is just that. HyperText Transfer Protocol. It is a protocol for transferring hypertext, which is text that contains links to other text. The World Wide Web is built on top of HTTP, which is why we often refer to the World Wide Web as the Web.

There are a few HTTP request methods, but the most common ones are GET and POST. The GET method is used to request data from the server, and the POST method is used to send data to the server. The GET method is used when we navigate to a website, and the POST method is used when we submit a form on a website. Other HTTP request methods include

1. PUT -> to update data on the server,
2. DELETE -> to delete data on the server,
3. HEAD -> to get the headers of a response,
4. OPTIONS -> to get information about the server,
5. and TRACE -> to get information about the request.

Other than 200, there are many other response codes that indicate different things.

For example, 404 means that the requested resource was not found, and 500 means that there was an internal server error. We will learn more about response codes later in this module.

Golang applications are often used as server applications, meaning that they receive requests from clients and respond to those requests. In this module, we will learn how to build a simple HTTP server in Go that can

handle client requests.

Creating a simple HTTP server in Go

```
package main

func main(){
    // specify routes and handlers
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
        fmt.Fprintf(w, "Hello World!")
    })

    // start the server
    http.ListenAndServe(":8080", nil)
}
```

The first part of the code specifies a handler, which is a function that handles requests to a specific route. The handler function takes two parameters: a ResponseWriter and a Request. The ResponseWriter is used to write the response to the client, and the Request contains information about the request, such as the request method and URL.

The second part of the code starts the server on port 8080. The ListenAndServe function takes two parameters: the port to listen on and the handler to use for incoming requests. In this case, we are using the default handler, which is the handler we defined above.

We can run the server by running the following command in the terminal:

```
go run main.go
```

We have just created a simple HTTP server in Go! We can test it out by opening a browser and navigating to http://localhost:8080. We should see the message "Hello World!" displayed in the browser.

1.2 Understand the concept of writer and response for HTTP

The Request-Response cycle in an HTTP server

The Request-Response cycle is the process of sending a request to a server and receiving a response from the server. The Request-Response cycle is used in many different applications, such as web browsers and mobile apps. In this module, we will learn how to build a simple HTTP server in Go that can handle client requests.

The anatomy of an HTTP request

An HTTP request consists of three parts: the request line, the request headers, and the request body. The request line contains the request method, the request URL, and the HTTP version. The request headers contain additional information about the request, such as the content type and length. The request body contains the data that is sent with the request, such as form data or JSON data.

The anatomy of an HTTP response

An HTTP response consists of three parts: the status line, the response headers, and the response body. The status line contains the response code, the response headers contain additional information about the response, and the response body contains the data that is sent with the response.

The role of the `http.ResponseWriter` and `http.Request` in handling HTTP requests and responses

The Go standard library provides a package called net/http that makes it easy to build HTTP servers in Go. The package provides functions for creating HTTP servers, handling requests, and writing responses. In this module, we will learn how to use the net/http package to build a simple HTTP server in Go.

For every handler function, we need to specify two parameters: a ResponseWriter and a Request. The ResponseWriter is used to write the response to the client, and the Request contains information about the request, such as the request method and URL.

The ResponseWriter has a Write function that takes a byte slice as a parameter. The Write function writes the byte slice to the response body. The Request has a URL field that contains the URL of the request. We can use the URL field to determine which handler function to use for the request.

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello World!")
    })

    http.ListenAndServe(":8080", nil)
}
```

If the user sends something in the body of the request, we can read it using the Request's Body field. The Body field is an io.ReadCloser, which is an interface that has a Read function that takes a byte slice as a parameter and returns the number of bytes read and an error. We can use the Read function to read the body of the request.

```
package main

import (
    "fmt"
    "io"
    "net/http"
)
```

```
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        body, err := io.ReadAll(r.Body)
        if err != nil {
            http.Error(w, "Error reading request body",
                http.StatusInternalServerError)
        }
        fmt.Fprintf(w, "Hello %s!", body)
    })

    http.ListenAndServe(":8080", nil)
}
```

in this example, the fmt.Fprintf function writes the string "Hello" followed by the body of the request to the response body. The body of the request is a byte slice, so we need to convert it to a string before we can write it to the response body. The first argument , *w* is a ResponseWriter, which is used to write the response to the client. The resulting string will be written to the response body.

The multiplexer

In Go, a multiplexer is a function that takes a request and returns a handler function. The multiplexer is used to determine which handler function to use for a request. The multiplexer is often called a router because it routes requests to the appropriate handler function.

The net/http package provides a function called NewServeMux that returns a multiplexer. The NewServeMux function takes no parameters and returns a multiplexer. We can use the multiplexer to register handler functions for different routes.

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    mux := http.NewServeMux()

    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello World!")
    })

    http.ListenAndServe(":8080", mux)
}
```

You can have different multiplexers for different routes. For example, you can have a multiplexer for the /users route and a multiplexer for the /posts route. You can also have a multiplexer for the / route and a multiplexer for the /users route. You can have as many multiplexers as you want.

Although, using just the net/http library can be cumbersome. Third-party developers have developed a library call Chi, which is just a simple routing library for Go

```
package main

import (
    "fmt"
    "net/http"

    "github.com/go-chi/chi"
)

func main() {
    r := chi.NewRouter()

    r.Get("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello World!")
    })

    http.ListenAndServe(":8080", r)
}
```

With Chi, you can specify the HTTP method for each route. For example, you can have a GET route for the /users route and a POST route for the /users route. You can also have a GET route for the / route and a POST route for the / route. You can have as many routes as you want.

Practical Examples

Using our Chi router, let's write a piece of code that receives a request and return a JSON response.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/go-chi/chi"
)

type User struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

func main() {
    r := chi.NewRouter()

    r.Get("/", func(w http.ResponseWriter, r *http.Request) {
```

```
        user := User{
            Name: "John Doe",
            Age:  30,
        }

        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusOK)

        json.NewEncoder(w).Encode(user)
    })

    http.ListenAndServe(":8080", r)
}
```

The `json.NewEncoder(w).Encode(user)` code means that we are encoding the user struct into a JSON object and writing it to the response body.

The response given to the client should be like this :

```
{
    "name": "John Doe",
    "age": 30
}
```

1.4 Database Interaction with SQLite

Introduce the concept of database interaction in Go

What is a database?

A database is a collection of data that is organized in a way that makes it easy to access, manage, and update. There are many different types of databases, such as relational databases, document databases, and graph databases. In this module, we will learn how to interact with a relational database in Go.

What is a relational database?

Relational databases are databases that store data in tables. Each table has a name and a set of columns. Each column has a name and a data type. Each row in a table represents a record, and each record has a value for each column. For example, a table called users might have columns for name, age, and email address. Each row in the table represents a user, and each user has a name, age, and email address.

A relational database management system (RDBMS) is a software that manages relational databases. The RDBMS is responsible for creating, updating, and deleting data in the database. The RDBMS also provides a way to query the database, which is a way to retrieve data from the database.

An example of an RDBMS is SQLite. SQLite is a lightweight embedded database that is used in many applications, such as web browsers and mobile apps. SQLite is a serverless database, which means that it does

not require a server to run. SQLite is also a file-based database, which means that it stores data in a file on the disk.

What is the database/sql package?

The database/sql package is a Go package that provides a generic interface for interacting with relational databases. The database/sql package provides functions for connecting to a database, executing queries, and retrieving results. The database/sql package is used by many other Go packages, such as the SQLite driver.

What is a database driver?

A database driver is a software that allows an application to interact with a database. The database driver is responsible for connecting to the database, executing queries, and retrieving results. The database driver is often called a database/sql driver because it implements the database/sql interface.

What is a database connection?

A database connection is a connection between an application and a database. The database connection is used to execute queries and retrieve results. The database connection is often called a database/sql connection because it implements the database/sql interface.

In order for us to retrieve data in a database, we need to connect to the database. We can connect to a database using the database/sql package. The database/sql package provides a function called Open that takes two parameters: the name of the driver and the data source name. The name of the driver is the name of the database driver, and the data source name is the connection string for the database.

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db, err := sql.Open("sqlite3", "test.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    fmt.Println("Successfully connected to database!")
}
```

The first parameter is the name of the driver, which is sqlite3 in this case. The second parameter is the data source name, which is test.db in this case. The data source name is a connection string that contains information about the database, such as the database name and the database file path.

The Open function returns a database/sql connection, which is used to execute queries and retrieve results. The database/sql connection is often called a database/sql connection because it implements the database/sql interface.

RDBMS such as MySQL or PostgreSQL is communicated via TCP/IP. However, SQLite is a file-based database, which means that it stores data in a file on the disk. Therefore, we need to specify the database file path in the data source name.

At a low-level, the library will open the file and read the data from it. The library will then parse the data and return it to us in a format that we can use.

Whereas MySQL for example, the library needs to open a TCP connection to the database server, send the query to the server, and wait for the server to respond. The server will then send the results back to the library, which will then parse the results and return them to us in a format that we can use.

If the client and server are on the same machine, the results will be returned almost instantly. However, if the client and server are on different machines, the results will take longer to return.

The easiness of SQLite is that we don't need to install any database server. We just need to install the SQLite library and we can start using it right away.

Creating a database and table

There are 3 ways that we can come up to create a SQLite database and table. We can either use a software called DB Browser for SQLite, or we can use the command line, or we can use the Go code.

DB Browser Method

DB Browser for SQLite is a free and open-source software that allows us to create and manage SQLite databases. We can download DB Browser for SQLite from the official website.

After installing DB Browser for SQLite, we can create a new database by clicking on the New Database button. We can then create a new table by clicking on the New Table button. We can then add columns to the table by clicking on the Add Field button. We can then add rows to the table by clicking on the Add Record button.

Command Line Method

We can also create a SQLite database and table using the command line. We can create a new database by running the following command in the terminal:

```
sqlite3 test.db
```

Afterwards, to create a table, we can run the following command in the terminal:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
```

```
    age INTEGER NOT NULL
);
```

We can then add rows to the table by running the following command in the terminal:

```
INSERT INTO users (name, age) VALUES ("John Doe", 30);
```

We can then retrieve the rows from the table by running the following command in the terminal:

```
SELECT * FROM users;
```

Go Method

We can also create a SQLite database and table using Go. We can create a new database by running the following code:

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db, err := sql.Open("sqlite3", "test.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    _, err = db.Exec(`
        CREATE TABLE users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            age INTEGER NOT NULL
        );
    `)
    if err != nil {
        panic(err)
    }

    fmt.Println("Successfully created users table!")
}
```

The Go method can be particularly useful if you want to create scripts that can be run on different machines. For example, if you want to create a script that can be run on a server, you can use the Go method to create the database and table.

Database Schema

Database schema is the structure of a database. It defines the tables, columns, and relationships between tables. The database schema is often called the database schema because it defines the structure of the database.

Extracting the schema

To extract the schema for your database, you can use the following command:

CLI

```
sqlite3 test.db .schema
```

Go

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db, err := sql.Open("sqlite3", "test.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    rows, err := db.Query(`
        SELECT sql FROM sqlite_master
        WHERE type='table' AND name='users';
    `)
    if err != nil {
        panic(err)
    }
    defer rows.Close()

    var schema string
    for rows.Next() {
```

```
        err := rows.Scan(&schema)
        if err != nil {
            panic(err)
        }
    }

    fmt.Println(schema)
}
```

Now let's proceed with some simple CRUD operations with SQLite in Go

CRUD Operations

Creating a record

To create a record in a table, we can use the INSERT INTO statement. The INSERT INTO statement takes two parameters: the name of the table and the values to insert into the table. The values to insert into the table are specified using the VALUES keyword.

```
<--! Skipping over some parts -->
func main() {
    db, err := sql.Open("sqlite3", "test.db")
        if err != nil {
        panic(err)
    }
        defer db.Close()

        _, err = db.Exec(`
        INSERT INTO users (name, age) VALUES ("John Doe", 30);
    `)
                if err != nil {
        panic(err)
    }

        fmt.Println("Successfully created user!")
}
```

Reading a record

To read a record from a table, we can use the SELECT statement. The SELECT statement takes two parameters: the columns to select and the table to select from. The columns to select are specified using the SELECT keyword, and the table to select from is specified using the FROM keyword.

```
<--! Skipping over some parts -->

func main() {
    db, err := sql.Open("sqlite3", "test.db")
```

```
        if err != nil {
        panic(err)
    }
        defer db.Close()

        rows, err := db.Query(`
        SELECT * FROM users;
    `)
                if err != nil {
        panic(err)
    }

        defer rows.Close()

        var id int
        var name string
        var age int
        for rows.Next() {
        err := rows.Scan(&id, &name, &age)
        if err != nil {
            panic(err)
        }
        fmt.Println(id, name, age)
    }
}
```

To properly structure your data for complex operations, consider using a struct to represent your data.

```
type User struct {
    ID   int
    Name string
    Age  int
}
```

We now update our code to use the User struct instead of the individual fields.

```
<--! Skipping over some parts -->
type User struct {
    ID   int
    Name string
    Age  int
}
func main(){
    db, err := sql.Open("sqlite3", "test.db")
        if err != nil {
        panic(err)
    }
        defer db.Close()
```

```
        rows, err := db.Query(`
        SELECT * FROM users;
    `)
                if err != nil {
        panic(err)
    }

        defer rows.Close()
        // create a slice of users
        // slices are like arrays but they can grow and shrink
        var users []User
        for rows.Next() {
        var user User
        err := rows.Scan(&user.ID, &user.Name, &user.Age)
        if err != nil {
            panic(err)
        }
        // the append function appends a new element to the end of the slice
        users = append(users, user)
    }
}
```

Alternatively, we can use the Scan function to scan the values into a slice of interface{}.

```
<--! Skipping over some parts -->
func main(){
    db, err := sql.Open("sqlite3", "test.db")
        if err != nil {
        panic(err)
    }
        defer db.Close()

        rows, err := db.Query(`
        SELECT * FROM users;
    `)
                if err != nil {
        panic(err)
    }

        defer rows.Close()

        var users []interface{}
        for rows.Next() {
        var user User
        err := rows.Scan(&user.ID, &user.Name, &user.Age)
        if err != nil {
            panic(err)
        }
        users = append(users, user)
```

```
    }
}
```

Updating a record

To update a record in a table, we can use the UPDATE statement. The UPDATE statement takes two parameters: the table to update and the values to update. The values to update are specified using the SET keyword.

```
<--! Skipping over some parts -->

rows, err := db.Query(`
        UPDATE users SET name = "Jane Doe" WHERE id = 1;
    `)
            if err != nil {
        panic(err)
    }
```

Deleting a record

To delete a record from a table, we can use the DELETE statement. The DELETE statement takes one parameter: the table to delete from. The table to delete from is specified using the FROM keyword.

```
<--! Skipping over some parts -->

rows, err := db.Query(`
        DELETE FROM users WHERE id = 1;
    `)
            if err != nil {
        panic(err)
    }
```