Golang Three-Layer Architecture Notes

1. Overview

The Three-Layer Architecture in Golang is a structured way to organize backend code into three clear layers — Handler, Service, and Repository. Each layer has a specific responsibility and communicates only with the layer directly below it. This pattern improves code organization, testability, and maintainability.

2. Layer Descriptions

2.1 Handler Layer

- The topmost layer that handles HTTP requests and responses. - It receives data from the client, validates it, and passes it to the Service layer. - It does not contain any business or database logic.

```
// Example
type UserHandler struct {
    service *UserService
}

func (h *UserHandler) Register(w http.ResponseWriter, r *http.Request) {
    var user User
    json.NewDecoder(r.Body).Decode(&user)
    h.service.RegisterUser(user)
}
```

2.2 Service Layer

- The middle layer that contains the business logic of the application. - It decides how data flows, applies validations, and calls repository methods. - It does not directly interact with the database.

```
// Example
type UserService struct {
    repo *UserRepository
}

func (s *UserService) RegisterUser(user User) {
    fmt.Println("Validating user data...")
    s.repo.CreateUser(user)
}
```

2.3 Repository Layer

- The lowest layer responsible for interacting directly with the database. - It stores the database connection or query object and exposes methods for CRUD operations.

```
// Example
type UserRepository struct {
    DB *sql.DB
}

func (r *UserRepository) CreateUser(user User) {
    _, err := r.DB.Exec("INSERT INTO users (name) VALUES ($1)", user.Name)
    if err != nil {
        log.Fatal(err)
    }
```

```
}
```

3. Dependency Injection

Dependency Injection (DI) is a technique where you pass dependencies (like database connections or repositories) from outside rather than creating them inside the struct. This keeps layers independent and makes testing easier.

```
// Example of Dependency Injection in main.go
func main() {
    db := ConnectDB()
    repo := &UserRepository{DB: db}
    service := &UserService{repo: repo}
    handler := &UserHandler{service: service}

    http.HandleFunc("/register", handler.Register)
    http.ListenAndServe(":8080", nil)
}
```

In this example, each layer receives its dependency as an argument. The handler gets the service, the service gets the repository, and the repository gets the database connection.

4. Why Use Three-Layer Architecture

- **Clean Code:** Each layer has a single responsibility. - **Testability:** You can easily mock the repository or service layer for testing. - **Maintainability:** Changing one layer (like switching databases) does not affect others. - **Scalability:** Easy to extend with new features or microservices.

5. Flow Summary

Request Flow: 1. Client sends HTTP request to Handler. 2. Handler validates input and calls the Service layer. 3. Service applies business logic and calls the Repository. 4. Repository performs database operation and returns data. 5. Service sends the processed result back to Handler. 6. Handler sends response to the client.

6. Key Points to Remember

- Handler: Handles requests and responses. - Service: Contains business logic and connects to repositories. - Repository: Handles database operations and connections. - Each layer only knows the layer directly below it. - Dependency Injection ensures loose coupling between layers.