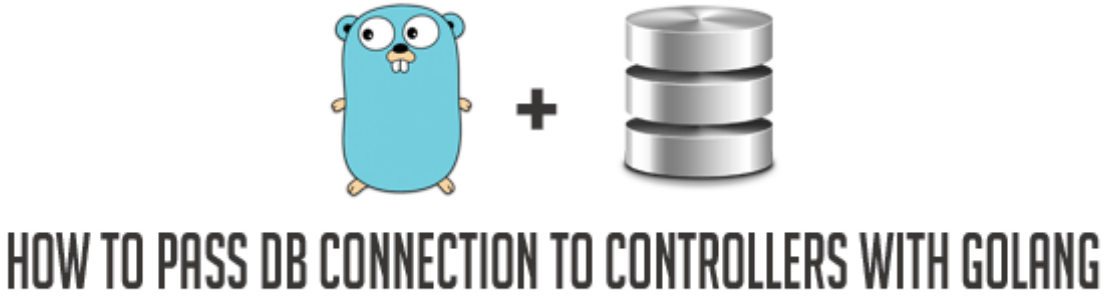
# Different Ways to Pass Database Connection into Controllers in Golang

Praveen

January 15, 2020 · 4 min read



When I started writing backends for web apps in Golang one of the biggest questions I had was – what is the right way to pass the database connection to the controllers? This article covers 3 approaches that can be used based on the application's size and requirement.

## Approach 1: Use Global Variable

Create a file `db.go` under a new subpackage `sqldb`. Declare a global variable `DB` of type `*sql.DB` to hold the database connection. Write a function that will open the connection and assign it to the global variable.

```
package sqldb

import "database/sql"

// DB is a global variable to hold db connection
var DB *sql.DB

// ConnectDB opens a connection to the database
```

```
func ConnectDB() {
        db, err := sql.Open("mysql", "username:password@/dbname")
        if err != nil {
                panic(err.Error())
        }

        DB = db
}
```

Now, where ever you need to access the database, you can simply import the global variable and start using it.

```
package controllers

import (
        "fmt"
        "net/http"

        "github.com/techinscribed/global-db/sqldb"
```

```
)

// HelloWorld returns Hello, World
func HelloWorld(w http.ResponseWriter, r *http.Request) {
        if err := sqldb.DB.Ping(); err != nil {
                fmt.Println("DB Error")
        }

        w.Write([]byte("Hello, World"))
}
```

This is the simplest approach to pass database connection to controllers but not an elegant way to do it. If you want to write a small application then you can go ahead with this approach. Stay away from using global variables if you are looking to write a serious application.

## Pros:

1. Quick and easy to setup.

## Cons:

1. The database can be accessed from any part of the application.
2. Hard to mock the database connection while writing test cases.
3. Extremely difficult to switch over to a different database.

You can find the example code here on Github.

# Approach 2: Create Struct to hold DB Connection

We will update our `db.go` to return the created database connection instead of assigning it to a global variable.

```
package sqldb

import "database/sql"

// ConnectDB opens a connection to the database
func ConnectDB() *sql.DB {
        db, err := sql.Open("mysql", "username:password@/dbname")
```

```
        if err != nil {
                panic(err.Error())
        }

        return db
}
```

In the controllers, we can create a struct `BaseHandler` to hold everything our controller needs to access, including database connection. Then write the handlers as a method of the struct.

```
package controllers

import (
        "database/sql"
        "fmt"
        "net/http"
)
```

```
// BaseHandler will hold everything that controller needs
type BaseHandler struct {
        db *sql.DB
}

// NewBaseHandler returns a new BaseHandler
func NewBaseHandler(db *sql.DB) *BaseHandler {
        return &BaseHandler{
                db: db,
        }
}

// HelloWorld returns Hello, World
func (h *BaseHandler) HelloWorld(w http.ResponseWriter, r *http.Request) {
        if err := h.db.Ping(); err != nil {
                fmt.Println("DB Error")
        }

        w.Write([]byte("Hello, World"))
}
```

Finally from the main function, we can tie the database and controllers together.

```
package main

import (
        "fmt"
        "net/http"

        "github.com/techinscribed/struct-db/controllers"
        "github.com/techinscribed/struct-db/sqldb"
)

func main() {
        db := sqldb.ConnectDB()

        h := controllers.NewBaseHandler(db)

        http.HandleFunc("/", h.HelloWorld)
```

```
        s := &http.Server{
                Addr: fmt.Sprintf("%s:%s", "localhost", "5000"),
        }

        s.ListenAndServe()

}
```

## Pros:

1. The database can be accessed only from controllers.
2. No global variables.
3. Easy to mock the database connection while writing test cases.

## Cons:

1. Difficult to switch over to a different database.

You can find the example code here on Github.

# Approach 3: Repository Interface per Model

We can define a repository interface for each model. Like so:

```
package models


// User ..
type User struct {
        Name string
}


// UserRepository ..
type UserRepository interface {
        FindByID(ID int) (*User, error)
        Save(user *User) error
}
```

and then instead of having the raw database connection in the `BaseHandler` struct we can have the repository interfaces.

```go
package controllers

import (
        "fmt"
        "net/http"

        "github.com/techinscribed/repository-db/models"
)

// BaseHandler will hold everything that controller needs
type BaseHandler struct {
        userRepo models.UserRepository
}

// NewBaseHandler returns a new BaseHandler
func NewBaseHandler(userRepo models.UserRepository) *BaseHandler {
        return &BaseHandler{
                userRepo: userRepo,
        }
```

```
}

// HelloWorld returns Hello, World
func (h *BaseHandler) HelloWorld(w http.ResponseWriter, r *http.Request) {
        if user, err := h.userRepo.FindByID(1); err != nil {
                fmt.Println("Error", user)
        }

        w.Write([]byte("Hello, World"))
}
```

We can then implement the repository interface, now it doesn't matter what database we use as long as the interface implementation is satisfied!

```
package repositories

import (
        "database/sql"
```

```
        "github.com/techinscribed/repository-db/models"
)

// UserRepo implements models.UserRepository
type UserRepo struct {
        db *sql.DB
}

// NewUserRepo ..
func NewUserRepo(db *sql.DB) *UserRepo {
        return &UserRepo{
                db: db,
        }
}

// FindByID ..
func (r *UserRepo) FindByID(ID int) (*models.User, error) {
        return &models.User{}, nil
}
```

```
// Save ..
func (r *UserRepo) Save(user *models.User) error {
        return nil
}
```

### Finally tying everything together in the main function

```
package main

import (
        "fmt"
        "net/http"

        "github.com/techinscribed/repository-db/controllers"
        "github.com/techinscribed/repository-db/repositories"
        "github.com/techinscribed/repository-db/sqldb"
)

func main() {
```

```
        db := sqldb.ConnectDB()

        // Create repos
        userRepo := repositories.NewUserRepo(db)

        h := controllers.NewBaseHandler(userRepo)

        http.HandleFunc("/", h.HelloWorld)

        s := &http.Server{
                Addr: fmt.Sprintf("%s:%s", "localhost", "5000"),
        }

        s.ListenAndServe()

}
```

Based on the environment (testing, development or production), we can pass different repository implementations to our controller.

**Example:** We can write a separate implementation that uses JSON/XML files that can be used for a testing environment, while development and production environment can use a SQL implementation. Later you can even completely switch over to a NoSQL implementation if required.

## Pros:

1. The database can be accessed only from the controllers.
2. No global variables.
3. Easy to mock the database while writing test cases.
4. Easy to switch over to a different database.

## Cons:

1. More code needs to be written.

You can find the example code here on Github.

# Conclusion

Like I already mentioned, It all depends on the size and requirement of the application. Approach 1 may suit well for small applications, Approach 2 may suit MVC application and application where you know the database won't change and Approach 3 may suit application built based on Domain Driven Design so that you can define one repository per Bounded Context.

If there is a different way, that you know or use do let me know on the comments.

database go golang

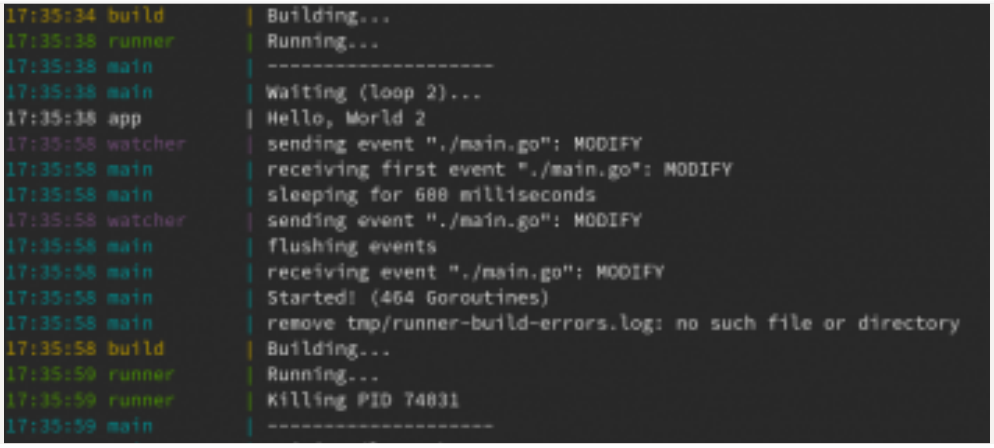
WRITTEN BY

**Praveen**

Programmer | Tech Blogger | Multitasker | Learner Forever | Someone who believes knowledge grows by sharing | Looking forward to using spare time productively

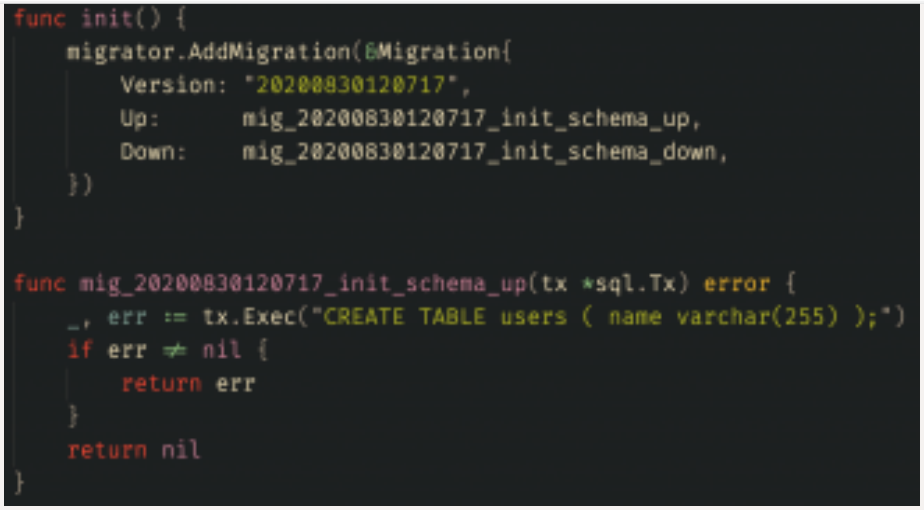
Write the first response

## You Might Also Like




## 5 Ways to Live Reloading Go Applications

Praveen
March 16, 2020 · 4 min read



```go
func init() {
    migrator.AddMigration(&Migration{
        Version: "20200830120717",
        Up:      mig_20200830120717_init_schema_up,
        Down:    mig_20200830120717_init_schema_down,
    })
}

func mig_20200830120717_init_schema_up(tx *sql.Tx) error {
    _, err := tx.Exec("CREATE TABLE users ( name varchar(255) );")
    if err != nil {
        return err
    }
    return nil
}
```

## How to Create DB Migration Tool in Go from Scratch

Praveen
August 30, 2020 · 9 min read

Contact · Write for us · Privacy Policy · Copyright © 2019-2020

象 Name

苓 Email

Submit