# Golang Seminar: Building an API with Golang and the Gin Framework

## Overview

This document outlines the process of building a RESTful API using Golang and the Gin framework. The goal is to provide a clear and structured flow for developing an API, focusing on best practices and organization. The API will handle basic CRUD operations for a Task Manager application, integrating with PostgreSQL as the database.

## Table of Contents


- Introduction to Building APIs with Golang
- Setting Up Your Project
- Creating the Task Model
- Developing the Service Layer
- Implementing the Controller Layer
- Defining Routes
- Database Integration with PostgreSQL
- Running and Testing the API
- Conclusion


## Introduction to Building APIs with Golang

Building APIs with Golang provides a powerful and efficient way to develop scalable web services. The Gin framework offers a minimalist approach, allowing developers to build fast and robust APIs with ease. This guide will walk you through the key steps involved in setting up and building an API, focusing on structure, best practices, and integration with PostgreSQL.

## Setting Up Your Project

1. **Initialize the Project**: Start by setting up a new Go module for your project. This will help in managing dependencies and organizing your code.

   ```
   mkdir taskmanager-api
   cd taskmanager-api
   go mod init taskmanager-api
   ```

2. **Install Dependencies**: Install the necessary dependencies, including the Gin framework and the PostgreSQL driver.

   ```
   go get -u github.com/gin-gonic/gin
   go get -u gorm.io/gorm
   go get -u gorm.io/driver/postgres
   ```

## Creating the Task Model

1. **Define the Task Model**: The Task model represents the data structure of tasks in your application. It's defined using Golang structs and will be used by GORM for database interactions.

   Example:

```
package models

import "gorm.io/gorm"

type Task struct {
    gorm.Model
    Title       string `json:"title"`
    Description string `json:"description"`
    Completed   bool   `json:"completed"`
}
```

## Developing the Service Layer

1. **Service Layer Overview**: The service layer is responsible for handling the business logic of the application. It interacts with the database and processes data before it's sent to the controller.
2. **Create Task Service**: Develop functions that perform CRUD operations on tasks, such as creating, retrieving, updating, and deleting tasks.

   Example:

```
package services

import (
    "taskmanager-api/database"
    "taskmanager-api/models"
)

func CreateTask(title, description string) models.Task {
    task := models.Task{
        Title:       title,
        Description: description,
        Completed:   false,
    }
    database.DB.Create(&task)
    return task
}

func GetTasks() []models.Task {
    var tasks []models.Task
```

```
    database.DB.Find(&tasks)
    return tasks
}
```

## Implementing the Controller Layer

1. **Controller Layer Overview**: The controller layer handles HTTP requests, calls the appropriate service methods, and returns HTTP responses. It serves as the intermediary between the client and the service layer.

2. **Implement Task Controller**: Develop controller functions to map routes to service methods, handling incoming requests and returning appropriate responses.

   Example:

```
package controllers

import (
    "net/http"
    "strconv"
    "taskmanager-api/services"

    "github.com/gin-gonic/gin"
)

func CreateTask(c *gin.Context) {
    var input models.Task
    if err := c.ShouldBindJSON(&input); err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    task := services.CreateTask(input.Title, input.Description)
    c.JSON(http.StatusCreated, task)
}

func GetTasks(c *gin.Context) {
    tasks := services.GetTasks()
    c.JSON(http.StatusOK, tasks)
}
```

## Defining Routes

1. **Route Definition Overview**: Define routes that map HTTP requests to specific controller functions. This ensures that the API can handle different types of requests and provide appropriate responses.

2. **Set Up Routing**: Define routes in a separate `routes.go` file to keep your application organized.

   Example:

```go
package routes

import (
    "taskmanager-api/controllers"
    "github.com/gin-gonic/gin"
)

func SetupRouter() *gin.Engine {
    router := gin.Default()

    router.GET("/tasks", controllers.GetTasks)
    router.POST("/tasks", controllers.CreateTask)

    return router
}
```

## Database Integration with PostgreSQL

1. **Connect to PostgreSQL**: Set up a connection to a PostgreSQL database using GORM. Ensure that environment variables are used to manage database credentials securely.

2. **Run Database Migrations**: Use GORM's auto-migration feature to create the necessary tables in the database based on your models.

   Example:

```go
package database

import (
    "fmt"
    "log"
    "os"
    "gorm.io/driver/postgres"
    "gorm.io/gorm"
)

var DB *gorm.DB

func Connect() {
    var err error

    dsn := fmt.Sprintf(
```

```
        "host=%s user=%s password=%s dbname=%s port=%s sslmode=disable TimeZone=UTC",
        os.Getenv("DB_HOST"),
        os.Getenv("DB_USER"),
        os.Getenv("DB_PASSWORD"),
        os.Getenv("DB_NAME"),
        os.Getenv("DB_PORT"),
    )

    DB, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})

    if err != nil {
        log.Fatal("Failed to connect to the database:", err)
    }

    fmt.Println("Database connection established.")
}
```

## Running and Testing the API

1. **Run the Application**: Start the Gin server and ensure that the API is running on the specified port.

   ```
   go run .
   ```

2. **Test the API**: Use tools like Postman or curl to test the API endpoints. Ensure that all CRUD operations work as expected.

## Conclusion

This guide provides a structured approach to building a RESTful API with Golang using the Gin framework. By following these steps, you can create a scalable and maintainable API that integrates with a PostgreSQL database. The separation of concerns into models, services, controllers, and routes ensures that the application remains organized and easy to extend.