Developer

Claro! Aqui estão todas as instruções formatadas como um bloco de código para que você possa copiá-las de uma única vez para o seu editor de texto:

````
### Instruções para Criar um CRUD em GoLang com PostgreSQL

Para desenvolver um CRUD (Create, Read, Update e Delete) em GoLang com o uso de uma API REST e Post

#### Instalando as Dependências:

Abra o terminal e execute os seguintes comandos para instalar as bibliotecas necessárias:

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

Criando a Estrutura do Modelo:

Crie um arquivo `main.go` e adicione o seguinte código:

```
package main

import (
	"gorm.io/gorm"
)

type User struct {
	ID    uint   `json:"id" gorm:"primaryKey"`
	Name  string `json:"name"`
	Email string `json:"email"`
}
```

Criando a Conexão com o PostgreSQL:

Adicione a função de conexão ao seu código:

```
package main

import (
	"github.com/gin-gonic/gin"
	"gorm.io/driver/postgres"
	"gorm.io/gorm"
	"log"
)

```
```

```
var db *gorm.DB

func connect() {
	var err error
	dsn := "<username>:<password>@tcp(127.0.0.1:5432)/<database>?sslmode=disable"
	db, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
	if err != nil {
		log.Fatal("Failed to connect to the database: ", err)
	}
}
```

Criando os Métodos CRUD:

Adicione as funções para manipular os dados:

```
func createUser(c *gin.Context) {
	var user User
	if err := c.ShouldBindJSON(&user); err != nil {
		c.JSON(400, gin.H{"message": "Bad request"})
		return
	}
	db.Create(&user)
	c.JSON(200, gin.H{"message": "User created", "data": user})
}

func getUsers(c *gin.Context) {
	var users []User
	result := db.Find(&users)
	if result.Error != nil {
		c.JSON(404, gin.H{"message": "Users not found"})
		return
	}
	c.JSON(200, gin.H{"message": "Users found", "data": users})
}

func getUser(c *gin.Context) {
	id := c.Param("id")
	var user User
	result := db.First(&user, id)
	if result.Error != nil {
		c.JSON(404, gin.H{"message": "User not found"})
		return
	}
	c.JSON(200, gin.H{"message": "User found", "data": user})
}

func updateUser(c *gin.Context) {
```

```
33          id := c.Param("id")
34          var user User
35          if err := db.First(&user, id).Error; err != nil {
36                  c.JSON(404, gin.H{"message": "User not found"})
37                  return
38          }
39
40          if err := c.ShouldBindJSON(&user); err != nil {
41                  c.JSON(400, gin.H{"message": "Bad request"})
42                  return
43          }
44          db.Save(&user)
45          c.JSON(200, gin.H{"message": "User updated", "data": user})
46  }
47
48  func deleteUser(c *gin.Context) {
49          id := c.Param("id")
50          if err := db.Delete(&User{}, id).Error; err != nil {
51                  c.JSON(404, gin.H{"message": "User not found"})
52                  return
53          }
54          c.JSON(200, gin.H{"message": "User deleted"})
55  }
```

Criando a Rota do Serviço REST:

Adicione o seguinte código para configurar as rotas:

```
1   func main() {
2           connect()
3           r := gin.Default()
4
5           r.POST("/users", createUser)
6           r.GET("/users", getUsers)
7           r.GET("/users/:id", getUser)
8           r.PUT("/users/:id", updateUser)
9           r.DELETE("/users/:id", deleteUser)
10
11          if err := r.Run(":8080"); err != nil {
12                  log.Fatal("Server Run Failed: ", err)
13          }
14  }
```

Executando o Serviço REST:

No terminal, execute o comando:

```
go run main.go
```

Você agora tem um CRUD em GoLang com API REST e PostgreSQL como banco de dados. Você pode testar as rotas utilizando ferramentas como `curl`, `Postman` ou outros clientes HTTP.

Observações Finais:

- Não se esqueça de substituir `<username>`, `<password>` e `<database>` com as credenciais corretas do seu banco de dados PostgreSQL.
- O código foi atualizado para se adequar à nova estrutura do GORM e recomendações do Gin, utilizando práticas de erro mais robustas.