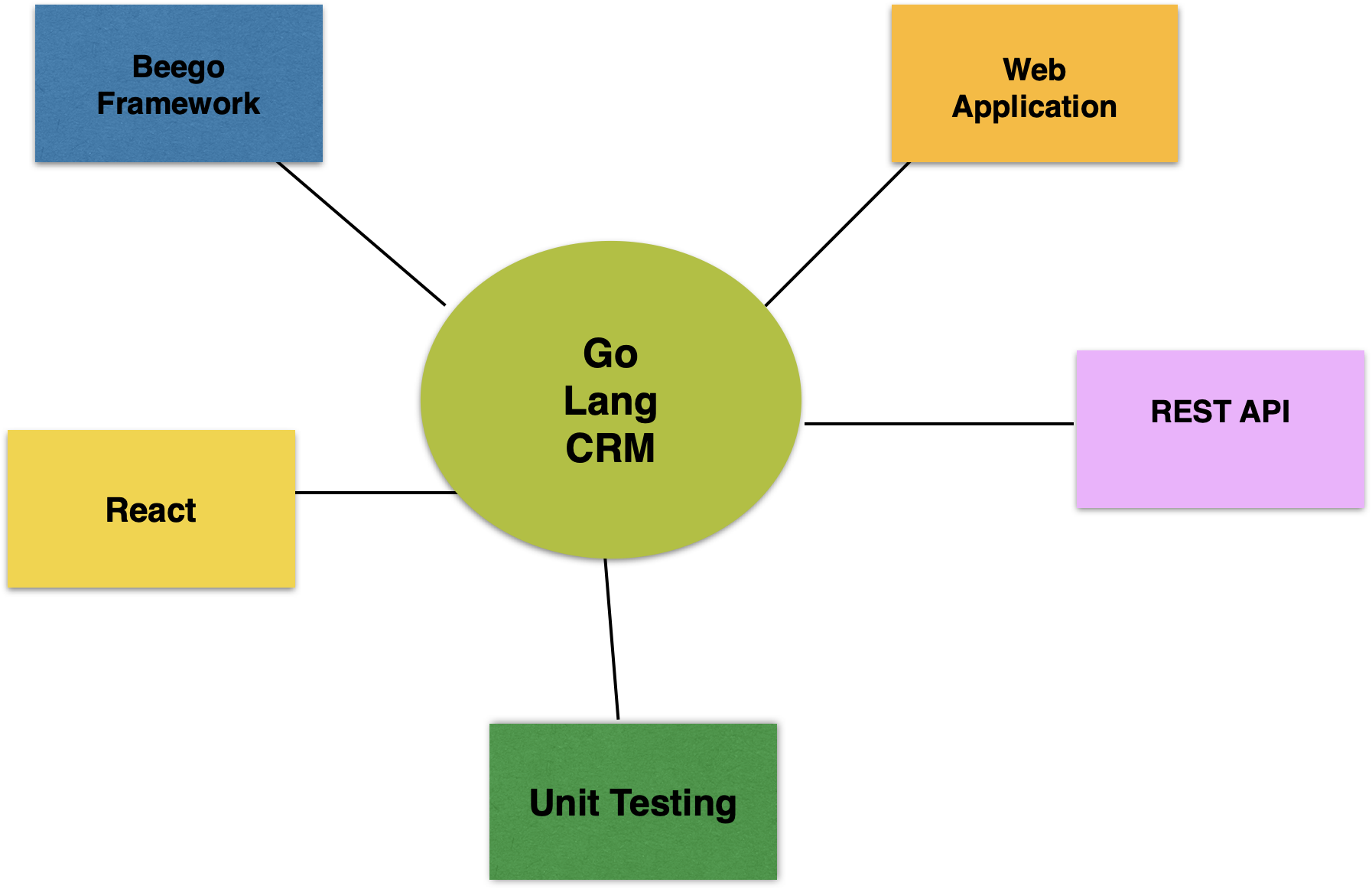
**C H A P T E R 7**

CHAPTER 7

# *My First CRM App in Go: CRM*

# Introduction

In this chapter, reader will know how to build a real-life application Customer Relationship Management. CRM application will be a web application which will have capability to maintain customer data and run campaigns for segment or a list of customers by geography or vertical.



## Structure

The chapter covers the following topics:

* CRM application in Go
  + REST API
  + Web App
* Unit Testing the REST API

## Objectives

In this chapter, we are going to build a customer relationship management application. We will be using Go Language. Beego Framework will be used to build the RESTAPI and web application. Readers will be presented with examples of unit testing Rest API in Go Language.

# CRM application in Go

We will be building Customer Relationship Management Application using Go. Customer Relationship Management application consists of creating customers, and managing the customer data. Campaigns will be created and targeted for publishing them to the customers. The leads are gathered as list of customers or segments of customers. Customer segments are created by an attribute like Age, Gender, Salary range, or geography. When the customer likes the product and buys the product, a transaction is logged in the back office of a retail store or an ecommerce company. The transactions are processed to check if the lead has realized and turned into revenue. This process helps in assessing the lead conversion. Transactions can also be checked for aberration to identify the behavior of the customer.

The other important feature is the loyalty points which has different levels like gold, silver, and bronze levels. Customer buys the products and loyalty points are stored in the system to track the progress of the customer. Offers are targeted based on the loyalty to improve the purchasing rate of the customer. The discounted offers are shared based on where the customer is in terms of the loyalty points. Companies bombastically announce frequently about loyal customers and converted leads from CRM portal.

Let us start creating the REST API for this sytem.

## REST API

We will be using Beego web framework to develop the CRM web application and REST API. Beego framework is an opensource framework which has application tools, Object Relationship Mapping framework, and other packages. ORM framework helps in cutting down the ordeal of developing the CRM relational schema and mapping it to object classes and names of the attributes. Beego is used to build CRM REST Apis which can be integrated with opensource JavaScript frontend frameworks. We can build CRM Beego web application using the Beego and SQLite database. Beego framework helps in developing web apps with affordable cost.

First, let us look at the code for Beego REST API.

**main.go**

package main

import (

"fmt"

"github.com/gorilla/handlers"

"go\_beego\_rest\_api/pkg/db"

handler "go\_beego\_rest\_api/pkg/handlers"

"log"

"net/http"

"github.com/gorilla/mux"

)

func main() {

r := mux.NewRouter()

DB := db.InitializeDB()

r.HandleFunc("/customers", handler.GetCustomers(DB)).Methods("GET")

r.HandleFunc("/create", handler.CreateCustomer(DB)).Methods("POST")

r.HandleFunc("/update", handler.UpdateCustomer(DB)).Methods("PUT")

r.HandleFunc("/delete", handler.DeleteCustomer(DB)).Methods("DELETE")

fmt.Println("Server at 8080")

log.Fatal(http.ListenAndServe(":8080", handlers.CORS()(r)))

}

The screenshot below shows the Beego Web app directory structure.

A screenshot of a computer

Description automatically generated with medium confidence

The above main.go has the code for initialization of ORM and registration of the model. Beego is started in the main method.

**db.go**

package db

import (

"fmt"

"github.com/jinzhu/gorm"

\_ "github.com/jinzhu/gorm/dialects/postgres"

)

func InitializeDB() \*gorm.DB {

db, err := gorm.Open("postgres", "user=newuser password=newuser dbname=crm sslmode=disable")

if err != nil {

fmt.Println(err)

} else {

fmt.Println("DB connected!")

}

return db

}

Handler.go has the code for tableName definition and the object class specification.

**handler.go**

package handler

import (

"encoding/json"

"fmt"

"net/http"

"github.com/jinzhu/gorm"

)

type CustomerBody struct {

Name string `json"name"`

}

type Customer struct {

Id int `json:"id"`

Name string `json:"name"`

Mobile string `json:"mobile"`

Address string `json:"address"`

}

func GetCustomers(db \*gorm.DB) http.HandlerFunc {

return func(w http.ResponseWriter, r \*http.Request) {

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

var customers []Customer

\_ = db.Table("customer").Select("id, name,mobile,address").Scan(&customers)

json.NewEncoder(w).Encode(customers)

}

}

func CreateCustomer(db \*gorm.DB) http.HandlerFunc {

return func(w http.ResponseWriter, r \*http.Request) {

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

var RequestBody CustomerBody

json.NewDecoder(r.Body).Decode(&RequestBody)

\_ = db.Table("customer").Create(&RequestBody)

fmt.Println("Created Customer")

json.NewEncoder(w).Encode(RequestBody)

}

}

func UpdateCustomer(db \*gorm.DB) http.HandlerFunc {

return func(w http.ResponseWriter, r \*http.Request) {

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

var PutBody Customer

json.NewDecoder(r.Body).Decode(&PutBody)

\_ = db.Table("customer").Where("id=?", PutBody.Id).Update("name", PutBody.Name).Scan(&PutBody)

fmt.Printf("Updated Customer with id %d\n", PutBody.Id)

json.NewEncoder(w).Encode(PutBody)

}

}

func DeleteCustomer(db \*gorm.DB) http.HandlerFunc {

return func(w http.ResponseWriter, r \*http.Request) {

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

var DeleteBody Customer

json.NewDecoder(r.Body).Decode(&DeleteBody)

\_ = db.Table("customer").Delete(&DeleteBody)

fmt.Printf("Deleted Customer with id %d\n", DeleteBody.Id)

json.NewEncoder(w).Encode(DeleteBody)

}

}

Now let us look at the database script to create a table in postgres crm database. You can use the below commands to create a postgres database crm and users.

psql -h localhost -d postgres

----

ALTER USER postgres PASSWORD 'postgres';

CREATE USER newuser with PASSWORD 'newuser' CREATEDB;

select \* from users;

\du

\l

\q

-----

createdb crm

psql -h localhost -d postgres

\c crm

CREATE TABLE "customer" (

"id" SERIAL,

"name" varchar(200) NOT NULL,

"mobile" varchar(100),

"address" varchar(400) DEFAULT NULL,

"notes" text,

UNIQUE (name)

);

\dt

\du

alter role newuser superuser;

create user newuser with password 'newuser';

grant all privileges on database crm to newuser;

alter role newuser superuser;

**Database**

CREATE TABLE "customer" (

"id" serial,

"name" varchar(200) NOT NULL,

"mobile" varchar(100),

"address" varchar(400) DEFAULT NULL,

"notes" text,

UNIQUE (name)

);

create user newuser with password 'newuser';

grant all privileges on database crm to newuser;

Now let us look at the routes configured in index.go.

The table below shows the routes/paths mentioned in the index.go.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Controller Method** | **Http Method** | **URL Route** | **Description** |
| Create | Add | POST | /create | Create a New Customer |
| Read | getAll | GET | /customers | Retrieve list of customers |
| Delete | Delete | DELETE | /delete | Delete a Customer |
| Update | Update | PUT | /update | Update a Customer |

You can run the beego rest api server by using the commands below:

go mod init go\_beego\_rest\_api

go mod tidy

The output of the rest api server execution is shown below:

(base) apples-MacBook-Air:go\_beego\_rest\_api bhagvan.kommadi$ ./go\_beego\_rest\_api

DB connected!

Server at 8080

Created Customer

Created Customer

Created Customer

Created Customer

Created Customer

Created Customer

# Unit Testing the REST API

In this section, we will investigate how post man is used for unit testing of the REST API. You can do the unit testing of the API using post man.

**Create Customer**

A screenshot of a computer

Description automatically generated with medium confidence

Post man can be used for testing the REST API. In the above screen shot, customer is created with a post method-based rest api call.

**Get Customers**

A screenshot of a computer

Description automatically generated with medium confidence

The above screen shot shows all customers by retrieving the results from a GET method REST api call.

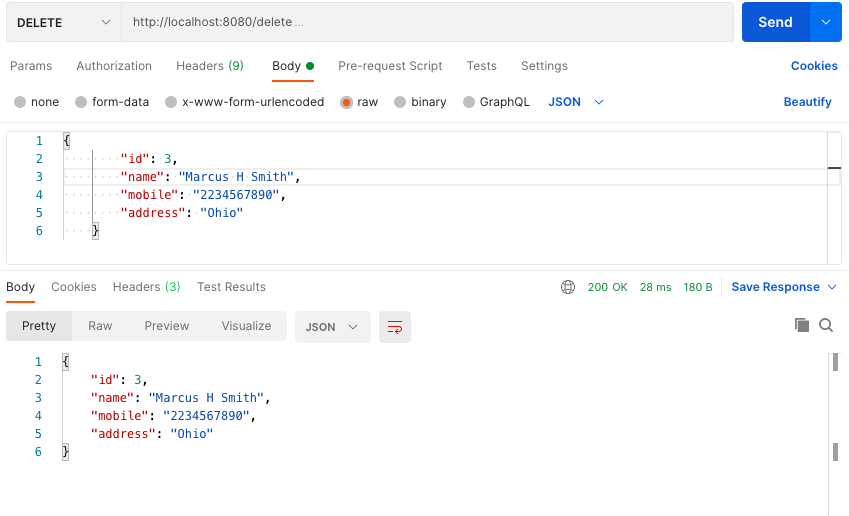
**Update Customer**

A screenshot of a computer

Description automatically generated with medium confidence

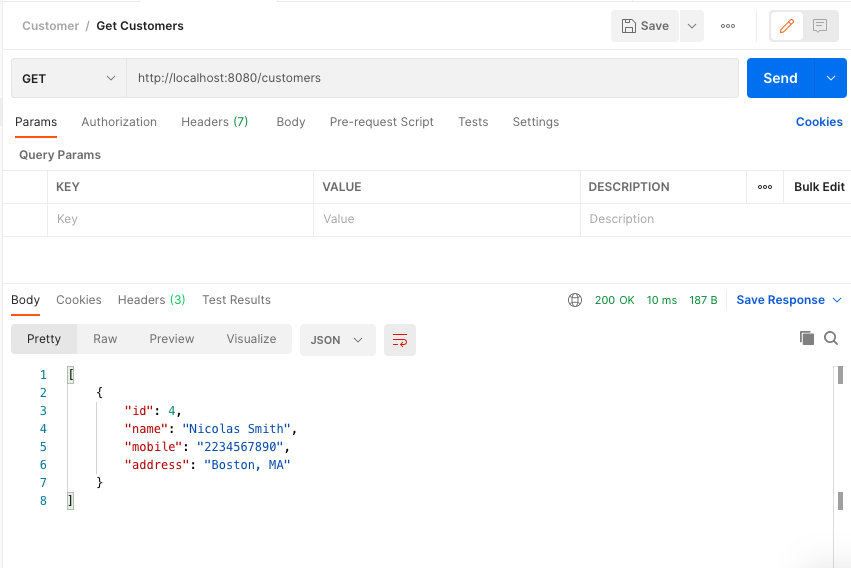
The above screen shot shows how postman is used for updating the customer. PUT method based REST api is used for updating the customer in the CRM portal.

Delete Customer



DELETE Method based rest api is called using postman for deleting a customer which is shown in the screenshot above.

**Get Customers After Delete**



The above screenshot shows the list of customers after update and delete operations.

## Web App

Now let us look at the web application. The web application is built using the React framework. First, let us look at the index.html and the other pages.

**Index.html**

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<meta name="theme-color" content="#000000">

<link rel="manifest" href="%PUBLIC\_URL%/manifest.json">

<link rel="shortcut icon" href="%PUBLIC\_URL%/favicon.ico">

<title>CRM Portal</title>

</head>

<body>

<div id="root"></div>

</body>

</html>

Now let us look at the index.js. This javascript file renders the application by loading the root element defined in example.js.

**index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import App from './example';

ReactDOM.render(

<App />,

document.getElementById('root')

);

**example.js**

import React from "react";

import {

BrowserRouter as Router,

Switch,

Route,

Link

} from "react-router-dom";

import AddCustomer from './AddCustomer';

import Logout from './Logout';

import Home from './Home';

export default function BasicExample() {

return (

<Router>

<div>

<ul>

<li>

<Link to="/home">Home</Link>

</li>

<li>

<Link to="/addCustomer">Add Customer</Link>

</li>

<li>

<Link to="/logout">Logout</Link>

</li>

</ul>

<hr />

{}

<Switch>

<Route exact path="/home">

<Home />

</Route>

<Route path="/addCustomer">

<AddCustomer />

</Route>

<Route path="/logout">

<Logout />

</Route>

</Switch>

</div>

</Router>

);

}

In the above file, different routes are defined for Home, AddCustomer, and Logout. Now let us look at Home page.

**Home.jsx**

import { Link } from "react-router-dom";

import React from "react";

class Home extends React.Component {

constructor(props) {

super(props);

this.state = {

jokes: []

};

this.serverRequest = this.serverRequest.bind(this);

}

serverRequest() {

fetch('http://localhost:8080/customers').then(response => {

return response.json();

}).then(json => {

console.log("json",json)

this.setState({ jokes: json});

});

}

componentDidMount() {

this.serverRequest();

}

render() {

return (

<div className="container">

<br />

<h2>CRM</h2>

<br />

<p>List of Customers</p>

<div className="row">

<div className="container">

<table border="1">

<tr>

<th>Name</th><th> Mobile</th> <th>Address</th>

</tr>

{this.state.jokes.map((item, index) => (

<tr border="1">

<td border="1">

{item.name} </td>

<td border="1">{item.mobile}</td>

<td border="1"> {item.address}</td>

</tr>

))}

<br />

</table>

</div>

<a href="/add">Add New Customer</a>

</div>

</div>

);

}

}

export default Home;

AddCustomer page has the form to create a customer. This page talks to the rest api for creation of customer.

**AddCustomer.jsx**

import { useState } from "react";

import React from "react"

import "./AddCustomer.css";

function AddCustomer() {

const [name, setName] = useState("");

const [mobile, setMobile] = useState("");

const [address, setAddress] = useState("");

const [message, setMessage] = useState("");

let handleSubmit = async (e) => {

e.preventDefault();

try {

let res = await fetch("http://localhost:8080/create", {

method: "POST",

body: JSON.stringify({

name: name,

mobile: mobile,

address: address

}),

});

let resJson = await res.json();

if (res.status === 200) {

setName("");

setAddress("");

setMessage("User created successfully");

} else {

setMessage("Some error occured");

}

} catch (err) {

console.log(err);

}

};

return (

<div className="container">

<br />

<div className="row">

<div className="container">

<form onSubmit={handleSubmit}>

<div class="form-group">

<label for="name">Customer name</label>

<input value={name}

placeholder="Name"

onChange={(e) => setName(e.target.value)} class="form-control" tabindex="1" />

</div>

<div class="form-group">

<label for="mobile">Customer Mobile：</label>

<input value={mobile}

placeholder="Mobile"

onChange={(e) => setMobile(e.target.value)} class="form-control" tabindex="2" />

</div>

<div class="form-group">

<label for="address">Address：</label>

<input value={address}

placeholder="Address"

onChange={(e) => setAddress(e.target.value)} class="form-control" tabindex="3" />

</div>

<button>Add Customer</button>

<div className="message">{message ? <p>{message}</p> : null}</div>

</form>

</div>

</div>

</div>

);

}

export default AddCustomer;

AddCustomer page has the css defined which is Add Customer.css.

**AddCustomer.css**

.AddCustomer {

display: flex;

justify-content: center;

margin-top: 5rem;

}

input {

display: block;

width: 20rem;

height: 2rem;

padding: 0.5rem;

font-size: 1.1em;

font-weight: 500;

margin-bottom: 2rem;

}

button {

border: none;

padding: 1rem;

width: 21.2rem;

font-size: 1.2em;

border-radius: 0.2rem;

cursor: pointer;

}

button:hover {

background-color: #c5c5c5;

}

.message {

font-size: 1.2em;

text-align: center;

color: #36a936;

}

**Logout.jsx**

import React from "react";

class Logout extends React.Component {

constructor(props) {

super(props);

this.logout = this.logout.bind(this);

}

logout() {

localStorage.removeItem("id\_token");

localStorage.removeItem("access\_token");

localStorage.removeItem("profile");

//location.reload();

}

render() {

return (

<div>

<h1>Logout</h1>

<span className="pull-right">

<a onClick={this.logout}>Log out</a>

</span>

</div>

);

}

};

export default Logout;

You can now compile and run the react application with the command below:

npm install

npm start

You can access the react web app at: http://localhost:3000/

The screenshots are attached below which show the web application.

A screenshot of a computer

Description automatically generated

Above is the screen shot of all customers loaded by the rest api for get All customers.

A screenshot of a computer

Description automatically generated with medium confidence

The above screenshot shows the form for adding new customer to the CRM portal.

A screenshot of a computer

Description automatically generated with medium confidenceAfter adding a new customer, user can come back and see the list of customers in the CRM portal.

A screenshot of a computer

Description automatically generated with medium confidence

After the customer is created, there is message in the page showing the success result of creating an user. This message helps in user to be cognizant of the result because of the action submitted.

# Conclusion

In this chapter we have covered topics related to building CRM application using Go Lang. CRM application is built using Beego Rest API, and React framework. Summary of the chapter is mentioned below.

* CRM application is built using Beego Framework based REST api interacting with sqlite database.
* REST Api can be developed using Beego Framework. Beego Framework has capabilities to have REST APIs, ORM, and developing web application framework.
* Postman can be used for unit testing the REST API for create,update, delete, and get all customers.
* Web application can be built using React Framework with html pages interacting with Beego Framework based REST Api.