**Rest API Server in Go**

**API:**

API stands for Application Program Interface. What it actually means is it exposes functionality without exposing internals. An API could be a function you wrote, or a function from a library or method from a framework, or a http endpoint.

**Rest API:**

What is REST

REST is acronym for Representational State Transfer. It is architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000

REST also does have it’s own 6 guiding constraints.

**Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

**Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

**Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

**Uniform interface** –. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

**Layered system** –An architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.

**Code on demand**  – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

**HTTP Verbs**

**GET** method requests a representation of the specified resource. Requests using GET should only retrieve data.

**HEAD** method asks for a response identical to that of a GET request, but without the response body.

**POST** method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

**PUT** method replaces all current representations of the target resource with the request payload.

**DELETE** method deletes the specified resource.

**CONNECT** method establishes a tunnel to the server identified by the target resource.

**OPTIONS** method is used to describe the communication options for the target resource.

**TRACE** method performs a message loop-back test along the path to the target resource.

**PATCH** method is used to apply partial modifications to a resource.

**Status Code:**

1xx Information and more

2xx Success and more

3xx Redirects and more

4xx Client Error and more

5xx Server Error and more

**Terminologies**

The following are the most important terms related to REST APIs

1. **Resource** is an object or representation of something, which has some associated data with it and there can be set of methods to operate on it. E.g. schools and employees are resources and delete, add, update are the operations to be performed on these resources.
2. **Collections** are set of resources, example: Companies is the collection of Company resource
3. **URL** (Uniform Resource Locator) is a path through which a resource can be located and some actions can be performed on it.

**API Endpoint**

This is what a API endpoint looks like.

https://www.github.com/golang/go/search?q=http&type=Commits

protocol subdomain domain path Port query

http/https subdomain base-url resource/some-other-resource some-port key value pair

https www github.com golang/go/search 80 ?q=http&type=Commits

**Protocol**

How the browser or client should communicate with the server.

**Subdomain**

Sub Division of the main domain

**Domain**

Unique reference to identify web site on the internet

**Port**

Port on the server the application is running on. By default its 80. So most cases we don't see it

**Path**

Path parameters in a Rest API represents resources.

**Basic structure is**

top-level-resource/<some-identifier>/secondary-resource/<some-identifier>/...

**Query**

Queries are key value pairs of information, used mostly for filtering purposes.

**https://jsonplaceholder.typicode.com/posts?userId=1**

Parts after the ? is the query parameters. We have only one query here. userId=1

**Headers**

This was not part of the URL itself but header is a part of network component sent by the client or the server. Based on who sends it. There are two kinds of header

Request Header (client -> server)

Response Header (server -> client)

**Body**

You can add extra information to both the request to the server and to the response from the server.

**Response Type**

Usually JSON or XML.

Now a days it's mostly JSON.

**Rest API with GO**

If you are writing Rest API why should you choose go?

* It's compiled. So you get small binaries.
* It's fast. (slower than c/c++ or rust) but faster than most other web programming languages.
* It's simple to understand.
* It works really well in the microservices world for reason no 1.

**net/http**

The standard library in go comes with the net/http package, which is an excellent starting point for building RestAPIs.

**The handler Interface**

type Handler interface {

ServeHTTP(ResponseWriter, \*Request)

} It has one method and one method only.

A struct or object will be Handler if it has one method ServeHTTP which takes ResponseWriter and pointer to Request

**Consuming REST API using Golang:**

**Introduction**:

The API that will be used in this post is icanhazdadjoke as you can probably guess from the URL it will return a joke when called. To get started let’s create a new directory that will hold the project.

**Project setup**

mkdir golang-api

cd golang-api

go mod init github.com/cameronldroberts/golang-api

File name main.go

**Imports:**

In Go a module is a collection of one or more packages which contain related code. In order to call the API we will be using the following four.

package main

import (

"encoding/json"

"fmt"

"io/ioutil"

"net/http"

)

**Example response**

This in an example response that we get from the API when calling it. As we know the structure of the JSON response that we will receive we can map this into a struct.

{

"id": "XgVnOK6USnb",

"joke": "What did the calculator say to the student? You can count on me",

"status": 200

}

**Struct**

The struct is fairly basic so in this instance it wouldn’t be too much work to map from JSON to the struct below.

type Response struct {

ID string `json:"id"`

Joke string `json:"joke"`

Status int `json:"status"`

}

**Code**

Finally for the function that calls the API and handles the response. When developing with Go it would not be advised to have function logic inside main() as this is meant to be the entry point for your program

func main() {

fmt.Println("Calling API...")

client := &http.Client{}

req, err := http.NewRequest("GET", "https://icanhazdadjoke.com/", nil)

if err != nil {

fmt.Print(err.Error())

}

req.Header.Add("Accept", "application/json")

req.Header.Add("Content-Type", "application/json")

resp, err := client.Do(req)

if err != nil {

fmt.Print(err.Error())

}

defer resp.Body.Close()

bodyBytes, err := ioutil.ReadAll(resp.Body)

if err != nil {

fmt.Print(err.Error())

}

var responseObject Response

json.Unmarshal(bodyBytes, &responseObject)

fmt.Printf("API Response as struct %+v\n", responseObject)

}

* Firstly we create a http client and create our http request
* We then add a couple of http headers to our request before sending the http request with `resp, err := client.Do(req)`
* We read in the response and then we unmarshal it into our response struct
* Finally we print out the response

**Running the project**

To run the project use the following command

go run main.go

This will compile and run the main.go file. The output should look something like

go run main.go

Calling API...

API Response as struct {ID:5h399pWLmyd Joke:What did the beaver say to the tree? It's been nice gnawing you. Status:200}

**Complete Code:**

package main

import (

"encoding/json"

"fmt"

"io/ioutil"

"net/http"

)

type Response struct {

ID string `json:"id"`

Joke string `json:"joke"`

Status int `json:"status"`

}

func main() {

fmt.Println("Calling API...")

client := &http.Client{}

req, err := http.NewRequest("GET", "https://icanhazdadjoke.com/", nil)

if err != nil {

fmt.Print(err.Error())

}

req.Header.Add("Accept", "application/json")

req.Header.Add("Content-Type", "application/json")

resp, err := client.Do(req)

if err != nil {

fmt.Print(err.Error())

}

defer resp.Body.Close()

bodyBytes, err := ioutil.ReadAll(resp.Body)

if err != nil {

fmt.Print(err.Error())

}

var responseObject Response

json.Unmarshal(bodyBytes, &responseObject)

fmt.Printf("API Response as struct %+v\n", responseObject)

}

--------------------------------------------------------------------------------------