


# Understanding the REST API Server

[KV FOOD](#) [Home](#) [Login](#) [Logout](#) [Signup](#) [Your Cart](#) [System Orders](#) [System Queue](#)



### What Are You Craving For Today?

## 1. Backend? Frontend?

If you've launched both the programs, you should have 2 servers running.

The backend server would run on <http://localhost> where the client server would be running on <http://localhost:8080>

Running them on different ports would ensure they would run without disrupting each other.

The backend server is also called the backend because a user, such as a merchant on KV Food Delivery service, does not have visibility on the buttons being pressed, links being executed etc when the server is running.

Instead, a merchant is presented with the client's interface, to allow them to make changes **to the backend** server without interacting directly with the server.

This is also why an interface, such as a chrome browser is also known as a client.

## 2. RestAPI

Citing from the web, an API is an application programming interface. It is a set of rules that allow programs to talk to each other. A developer creates the API on the server and allows clients to talk to it.

REST determines how the API looks like. It stands for "Representational State Transfer. It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data(called a resource when you link to a specific URL.

Each URL is called a **request** while the data sent back to you is called a **response**.

Source: <https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/>

### 3. Here's how KV Food's generates it's API

### Edit User Profile

Your API Key

Copy

Regenerate API Key

```
99      c := &http.Cookie{
100          Name:      "session",
101          Value:      sID.String(),
102          HttpOnly: true,
103      }
104
105      http.SetCookie(w, c)
106
107      dbSessions[c.Value] = session{un, time.Now()} // i wil store your
108
109      insertSessionsDB(un, c.Value)
110
111      userApiKey := generateApiKey(c.Value)
112
113      bs, err := bcrypt.GenerateFromPassword([]byte(p), bcrypt.MinCost)
114      if err != nil {
```

The code cleverly uses a partial value of the UUID value generated by the Satori package. This reduces the need for a separate package to keep the program clea and lower its dependencies. On top of that; to make it easy to track the "API keys that are generated", we use an algorithm to amend the UUID value before they become usable API keys.

```

96 func generateApiKey(s string) string {
97
98     //time format in DDMMYY
99     var timeNow string = time.Now().Format("010206")
100
101     result := strings.Split(s, "")
102
103     result1 := result[0:26]
104
105     result1WithDate := append(result1, timeNow)
106
107     userApiKey := strings.Join(result1WithDate, "")
108
109     return userApiKey
110 }

```

The function generateAPIKey splits the API key into a slice, retrieves the first **26** values, and appends the date in DDMMYY format which gives us another **6** characters. This results in a unique **32 char** API key and a quick look on an API key can also tell us when it was generated when querying a DB isn't possible.

#### 4. REST : URL = request, data sent back = response

```

186 router.HandleFunc("/api/v1/apivalidation", apiValidation).Methods("Get")
187 router.HandleFunc("/api/v1/nameaddress", nameAddress).Methods("Get")
188 router.HandleFunc("/api/v1/retrieveall", retrieveAll).Methods("Get")
189 router.HandleFunc("/api/v1/additems", addItems).Methods("Post")
190 router.HandleFunc("/api/v1/merchantsetup", merchantSetup).Methods("Post")
191 router.HandleFunc("/api/v1/edititems", editItems).Methods("Put")
192 router.HandleFunc("/api/v1/deleteitems", deleteItems).Methods("Delete")

```

And because each URL sent to the server requires a response, the server also needs to know how to handle them individually. Let's look at the first line of code for example and examine how the server handles it.

```

695 func apiValidation(w http.ResponseWriter, req *http.Request) {
696
697     //query if API key is correct
698     v := req.URL.Query()
699
700     if validateAPIkey(v["api"][0]) {
701         w.WriteHeader(http.StatusOK)
702         w.Write([]byte("Server Response: API Key Validation Successful"))
703     } else {
704         w.WriteHeader(http.StatusNotFound)
705         w.Write([]byte("Server Response: API Key Does Not Exist. API Key Validation failed"))
706     }
707
708 }

```

The handler function simply takes in the API key attached together with the URL which looks something like this:

`http://localhost/api/v1/apivalidation?api=89517e27-810d-425d-a6ff-d1015921`

The handle function retrieves the API key through `req.URL.Query`, fires another function to check against a database.

If the `validateAPIkey` function returns true, it simply writes **a response back** with a header, as well as a message. In this case, it returns a status 200 (`http.StatusOK`) as well as a message "API Key Validation Successful".

And if it fails, a status 400 (`http.StatusNotFound`) and an accompany message "API Key Does Not Exist"

I hope this explains why each URL is called a **request** while the data sent back to the client is called a **response**.

## 5. API key is also like a lock to a Vault

If someone with ill intentions had keys to your house, they'll break into your house and steal your stuff; much like what a burglar would do.

Similarly, if someone had access to your API key, they could do the same.

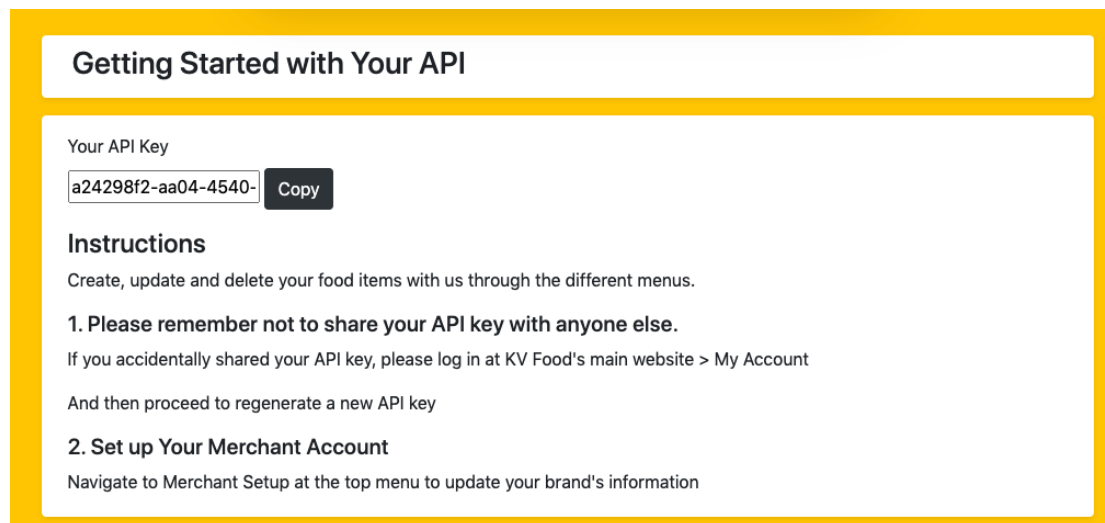
To prevent that, the backend server stores its API keys only in the database and not anywhere else. The API key is the **only way** a client can establish contact with the web server.

In the event that a merchant loses their API key, all they have to do is press the regenerate button to have a new API key

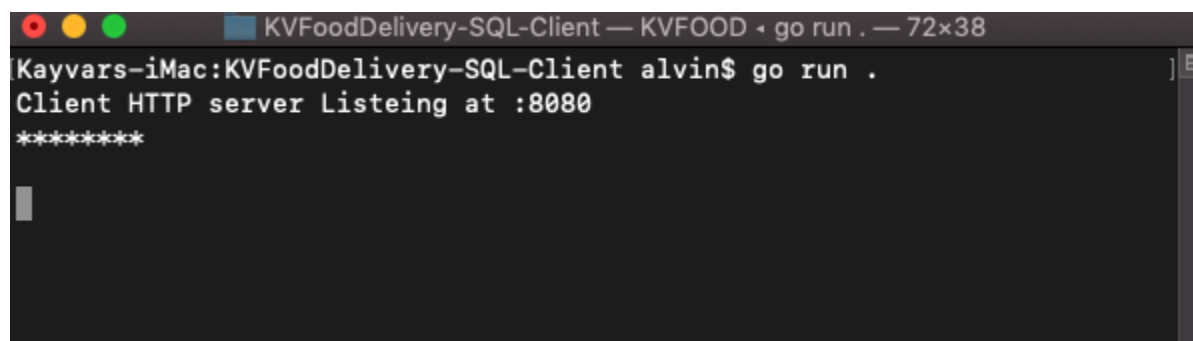


On top of that, the client sends a get request on every page to ensure the API key is validated. To see it in action, try this.

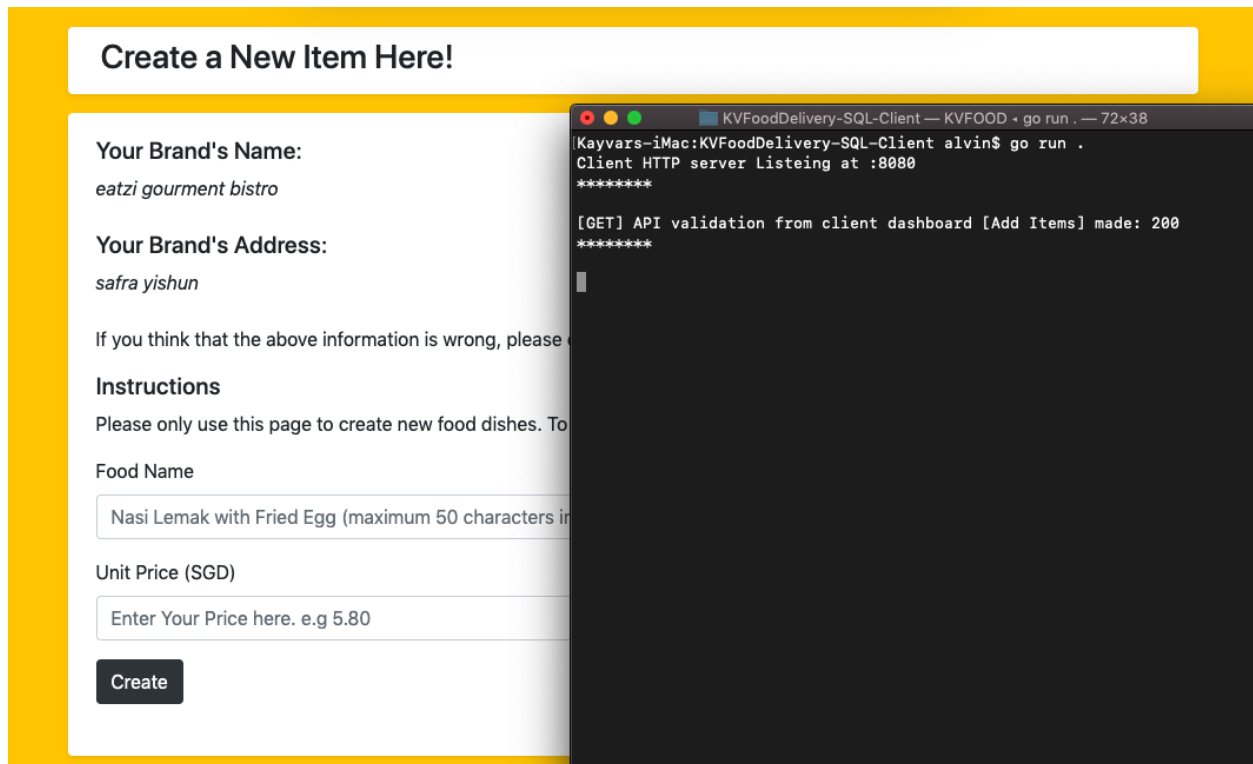
Ensure that your client has it's API key already validated in the My API page.



Now take a look at the terminal that started the client server:



On the client, navigate to Add items:



The screenshot shows a web form titled "Create a New Item Here!" with a yellow header. The form contains the following fields and text:

- Your Brand's Name:** eatzi gourmet bistro
- Your Brand's Address:** safra yishun
- If you think that the above information is wrong, please
- Instructions**  
Please only use this page to create new food dishes. To
- Food Name**  
Nasi Lemak with Fried Egg (maximum 50 characters in
- Unit Price (SGD)**  
Enter Your Price here. e.g 5.80
- Create** button


Overlaid on the right is a terminal window titled "KVFoodDelivery-SQL-Client — KVFOOD · go run . — 72x38". The terminal output shows:

```
Kayvars-iMac:KVFoodDelivery-SQL-Client alvin$ go run .  
Client HTTP server Listeing at :8080  
*****  
[GET] API validation from client dashboard [Add Items] made: 200  
*****
```

As soon as you head to a page that **has to** send a request to the backend server, it sends a GET request to the backend server. The **200** represents the request that is sent back, indicating that the request for API validation was successful. (as mentioned in the previous point)

This is also why the form on the left is visible. If the API validation **failed**, the form would not be visible. Let's try that out.

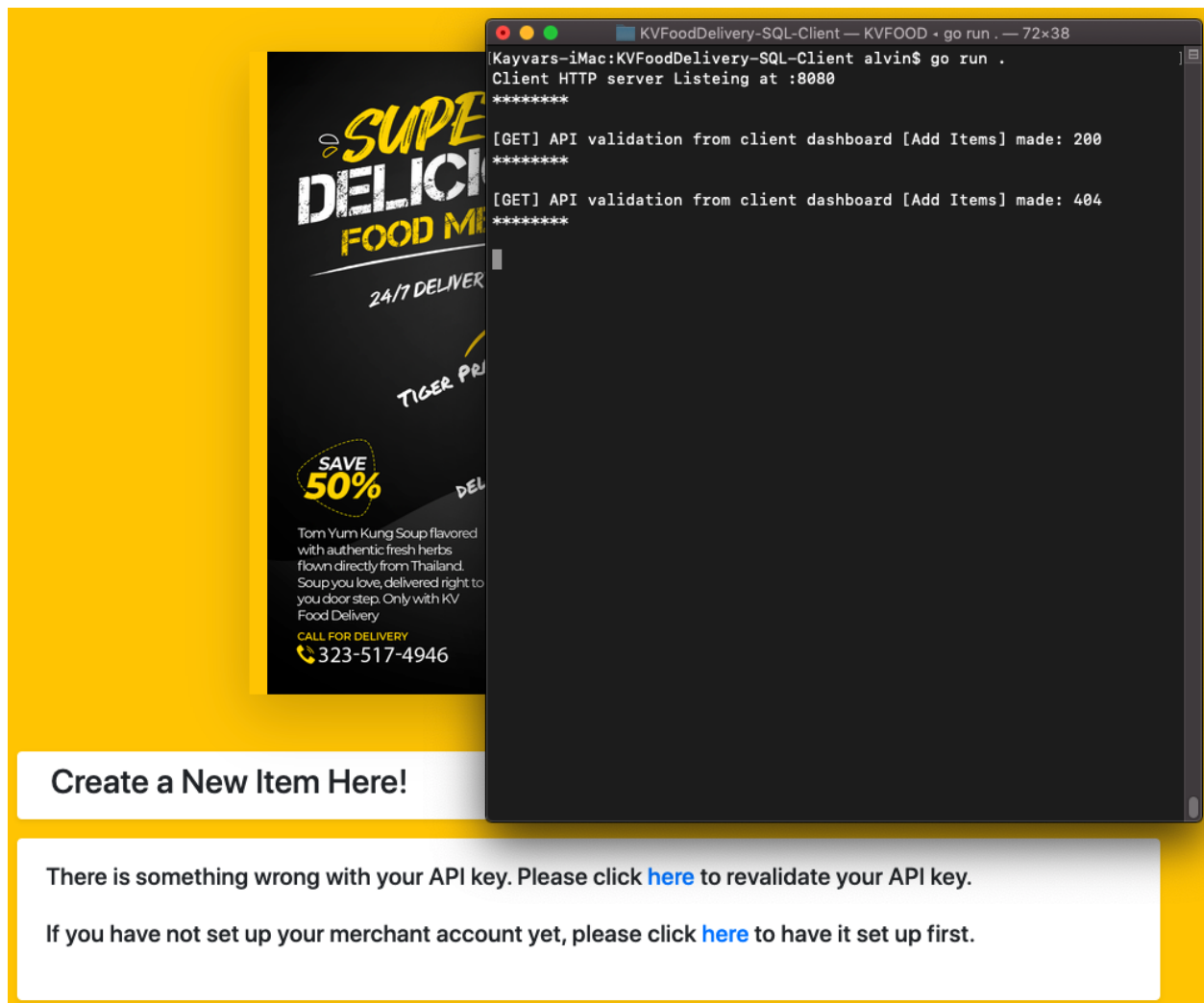
Head over to the server's page and click on Regenerate API



The screenshot shows a web form titled "Edit User Profile" with a yellow header. The form contains the following fields and text:

- Your API Key**  
a24298f2-aa04-4540- Copy
- Regenerate API Key** button

Now head back to the client's page and refresh the page. Again, look at your terminal and see if you observe the same.



You would have realised the page disappeared but instead asks you to revalidate your key. The terminal also warns you that the API validation failed since a 404 was returned back to the server.

This means that even if a merchant loses their API key, a simple regeneration of a new API key would render all connected clients almost instantaneously.

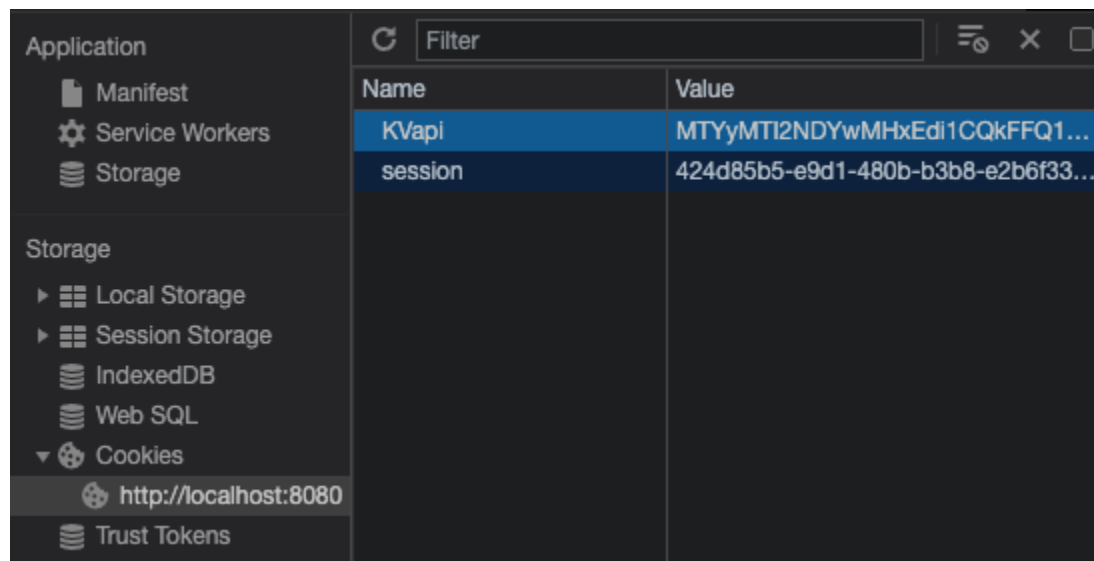
To solve this error, all you have to do is copy the new API key, re-validate your API key and you'll be back on track.

## 6. API key is cleverly stored on the client's browser

A client is meant to be lightweight; so we made the decision to store API keys on the client's browser instead. This means that their API key will persist for as long as the same browser is used.



Here's where we store the API Key:



The screenshot shows the Chrome DevTools Application tab. On the left, the 'Storage' section is expanded, showing 'Local Storage', 'Session Storage', 'IndexedDB', 'Web SQL', 'Cookies', and 'Trust Tokens'. The 'Trust Tokens' section is selected, showing a single item for 'http://localhost:8080'. The main pane displays a table of storage items:

Name	Value
KVapi	MTYyMTI2NDYwMHxEdi1CQkFFQ1...
session	424d85b5-e9d1-480b-b3b8-e2b6f33...

And in case the cookie information gets stolen, the thief will get it's encrypted values instead.

Also, since the API key persists in the local browser, a regeneration of the key will also not be detected. Unless, we made it detect it so that's what we did.

Remember status 200 vs 400 when we moved from an existing API key to a new key? Here's what happens when a status code of 400 is generated:

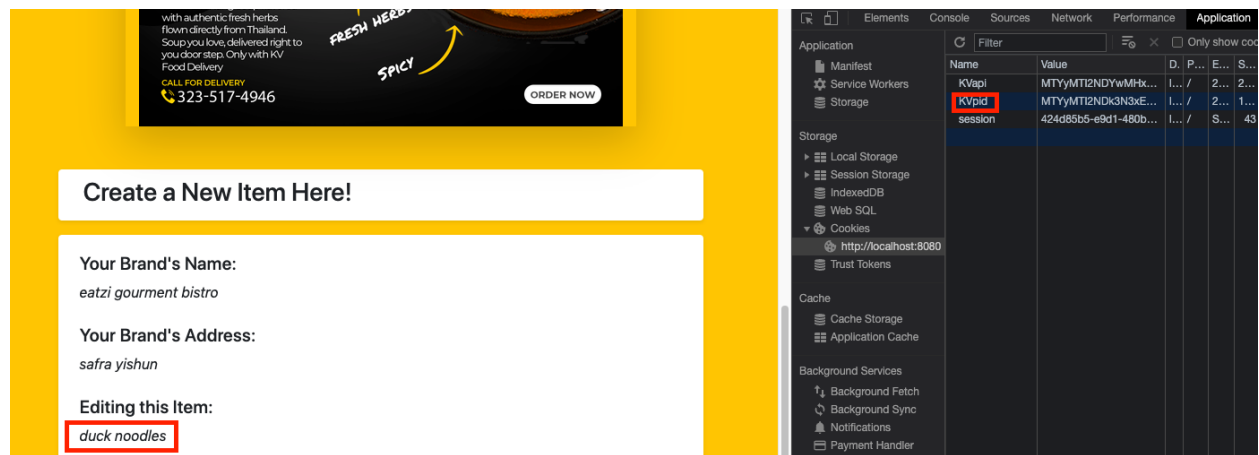
```
339 |         if response.StatusCode == 200 {
340 |             err = json.Unmarshal(data, &result)
341 |             if err != nil {
342 |                 fmt.Println("error processing json files")
343 |             }
344 |         } else {
345 |             session.Options.MaxAge = -1
346 |             err = session.Save(req, w)
347 |             if err != nil {
348 |                 fmt.Printf("Failed to delete session", err)
349 |             }
350 |         }
351 |     }
352 | }
```

We forced the API key to decay using MaxAge -1 . When the API key disappears from the cookies on the local browser, My Api page will again ask for the new API key to be re-validated.

## 7. Food-to-be-edited is also stored on the browser

Based on the same theory, we decided to also store another variable on the browser.

This variable *pid* represents the food item to be edited. This allows the value to be passed from one page to another, without incurring extra space complexity.



When the merchant decides that a new item is to be edited instead; then this value changes accordingly.

## 8. GET, POST, PUT, DELETE

Everything the client does, it does it with these few requests.

Here's a quick summary of what the client does and the type of request it represents.

API validation - GET

Retrieving brand name and address - GET

Adding new food items - POST

Editing existing food items - PUT

Deleting an existing food item - DELETE

And because we know it can be hard to keep track of the codes, you can find out when the client does a certain type of request when you look at the terminal.

```
KVFoodDelivery-SQL-Client — KVFOOD - go run . — 72x38

*****

*****

[GET] API validation from client dashboard [Add Items] made: 200
*****

[GET] API validation from client dashboard [Add Items] made: 200
[POST] Request from client dashboard [Add Items] made: 201
Your new food item has been created succesfully and is pending for approval.
[GET] API validation from client dashboard [Edit Items] made: 200
*****

[GET] API validation from client dashboard [Edit Items] made: 200
*****

[GET] API validation from client dashboard [Edit Item] made: 200
*****

[GET] API validation from client dashboard [Edit Item] made: 200
nasi lemak with fried egg nasi lemak with fried egg 12.35

{OldFoodname:nasi lemak with fried egg NewFoodname:nasi lemak with fried egg Price:12.35}
[Put] Request from client dashboard [Edit Item] made: 200
The price of your menu item has been updated successfully
[GET] API validation from client dashboard [Delete Items] made: 200
*****

[GET] API validation from client dashboard [Delete Items] made: 200
*****

[DELETE] Request from client dashboard [Delete Items] made: 200
200
The food menu item has been succesfully deleted
```

This marks the end of this report. We hope you've enjoyed learning about the program as much as I've spent the time building it.