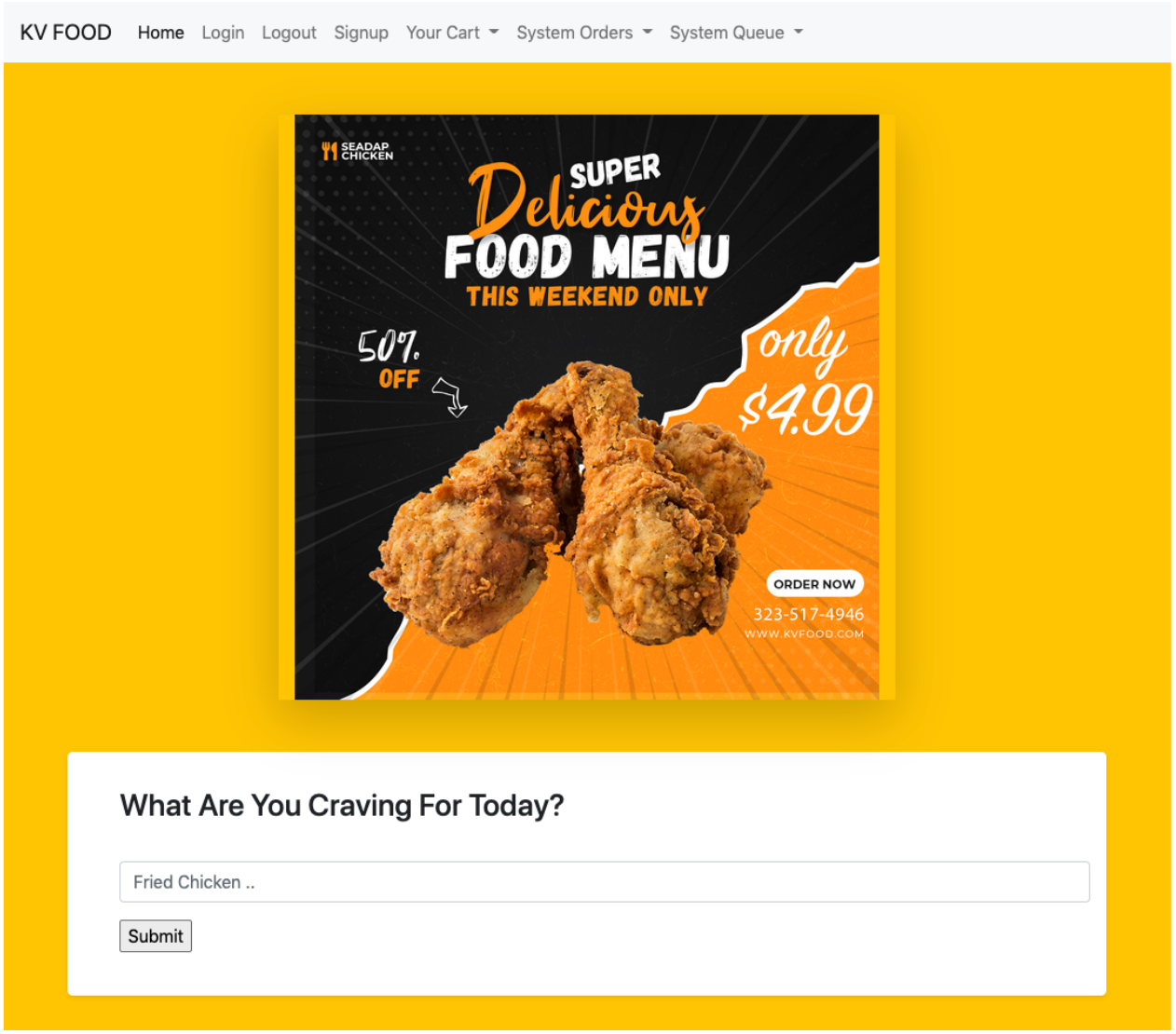
Design Document for KV Food Delivery Service:

Introduction:

The KV Food Delivery server-client project was created in an attempt to solve teething customer service issues that have been affecting customer's average order values, customer satisfaction and ultimate customer loyalty.

While there were alternatives that were created before; there is no single application that can integrate the features that are now available.



Design Considerations:

The web client site should be lightweight and easily deployable to an online server like an aws. The site must be user friendly to encourage early adopters to start feedback / app revisions. The main functionality of the web app must also allow fast searching of certain data especially merchant information, transaction IDs as well as system queue numbers and driver dispatch menus.

Performance Considerations:

Without querying online databases, databases have to be imported locally, created and stored in memory. Utilizing GO routines are mandatory as part of performance control and flow. The GO routines also ensure that the data can be populated accurately with the help of channels confirming routine completions before the system can be utilized.

As a query app loaded with features, the features themselves must also be able to present data fast and correctly.

Error and Panic Considerations:

Error handling and panic cases are to be meticulously tested and addressed for.

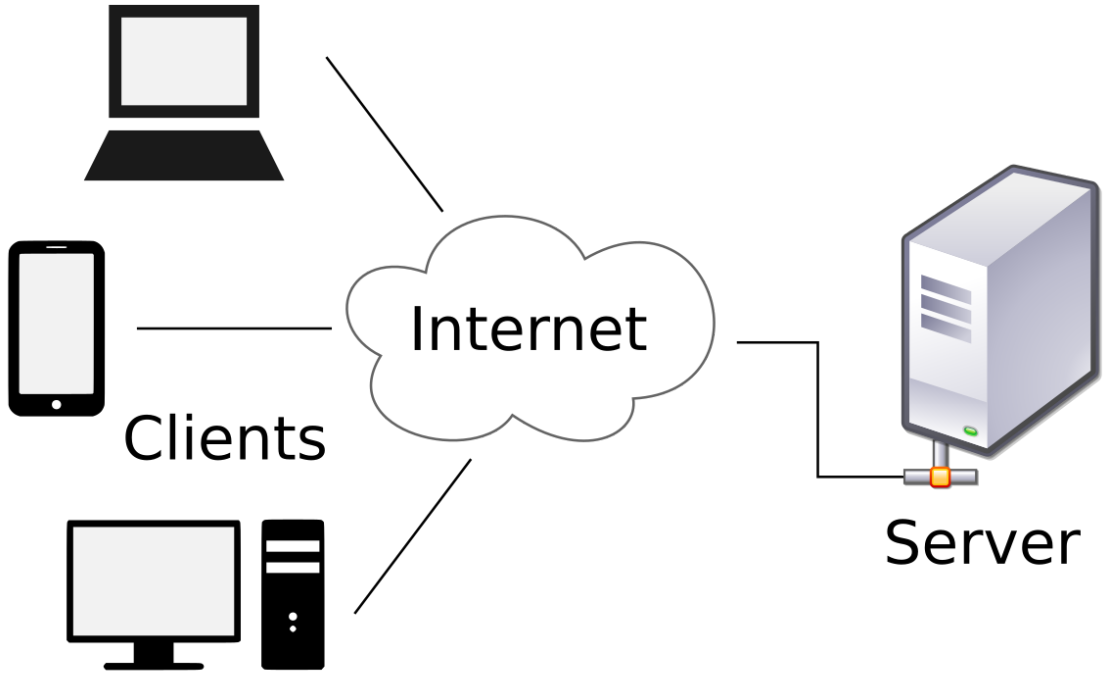
Handling data, writing and reading can cause panic issues especially with databases that are heavy on slices. The most common panic that can happen in the application is due to array or slice length mismatches that occur. The implementation of a prefix trie (discussed more on it later) can only contain certain characters.

Characters that are recognized and stored in the arrays will cause panics. In light of that; deferred recovery functions have to be utilised in the code to mitigate such issues and recover from potential panics.

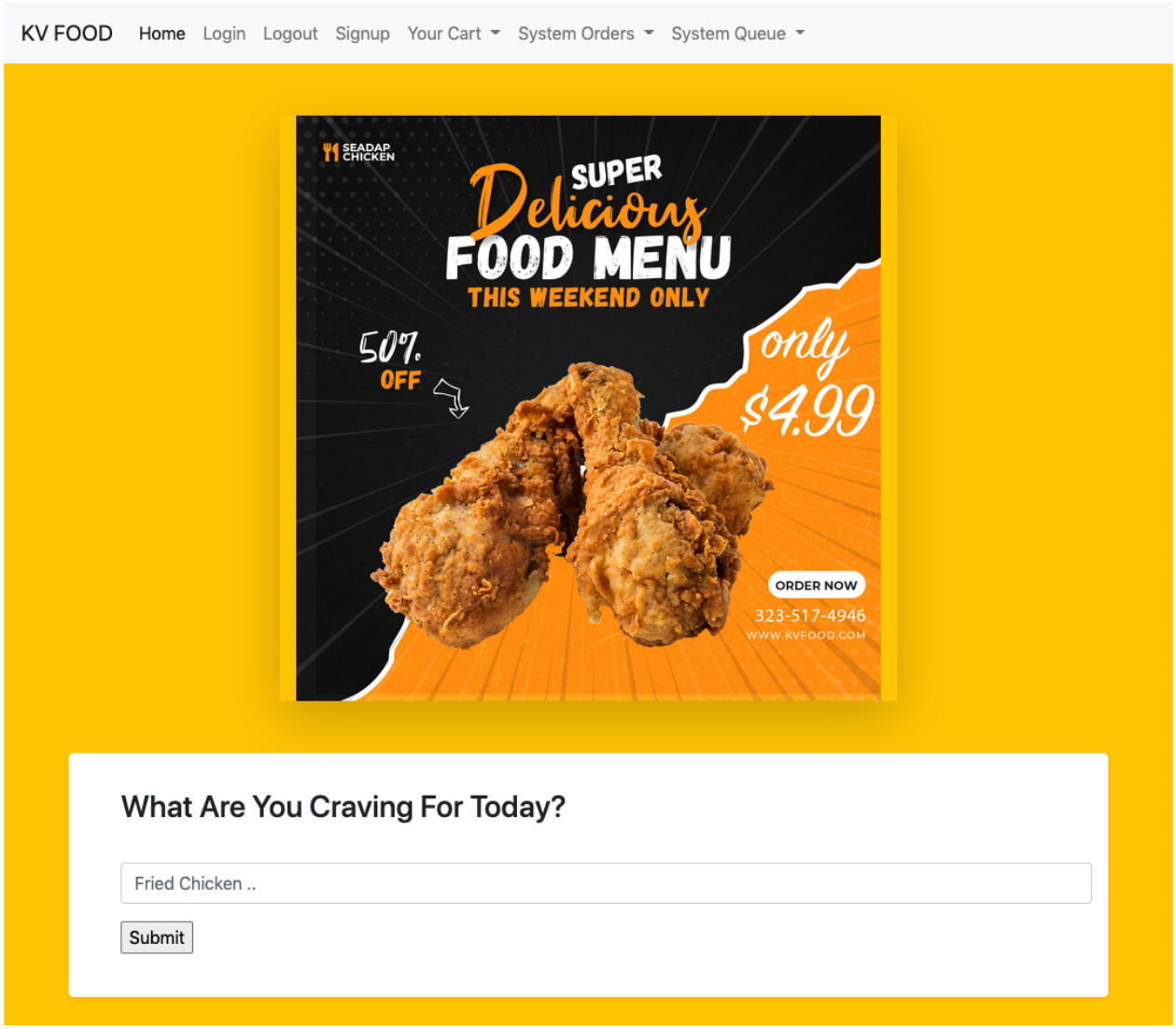
The Solution for an Online Server : ListendAndServe:5221

Previously, the KV Food Delivery backend app was purely offline, made as a CLI application. Now that we need an online push, we need a deployable solution.

ListenAndServe(":5221", nil) , a single line of code deployed in the code, allows a HTTP server to send responses based on different requests from the clients. The **nil** allows the default golang mux handler to be utilised. This value is to be changed if other 3rd party mux package handlers are employed.

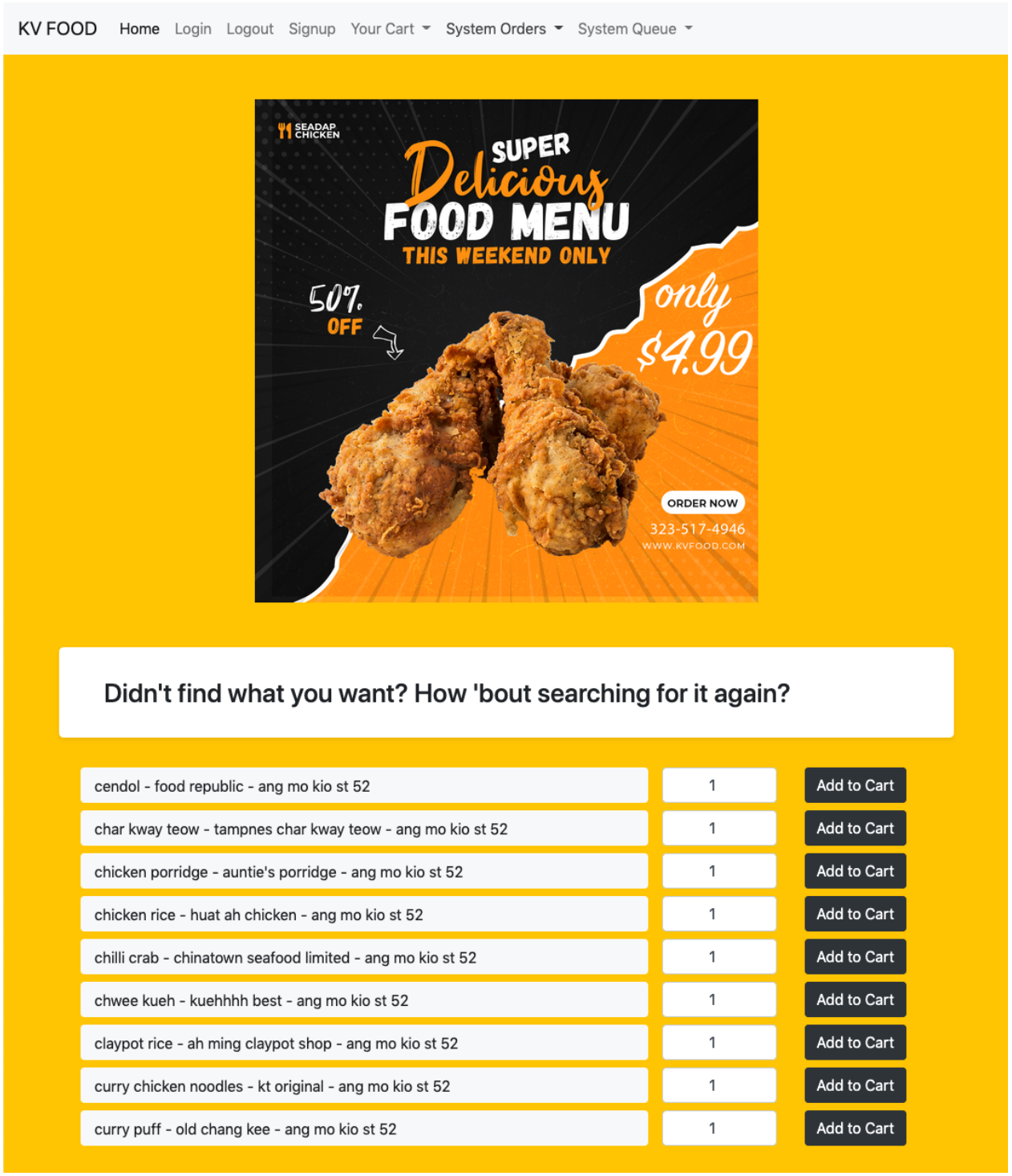


The above image is a clear representation of a client server relationship. Clients from all over the web, will and make requests to the server, and in turn the server will be able to handle the requests by **responding** to them.



On our program , you can see a server accepting and 'listening' to a request. A user who is logged in, has the ability to type something into the search bar. In this case, it is the letter 'c'. Pressing the enter key or the 'search' button is also known as sending a request for the letter 'c'. In layman terms, this is sending a 'search' type of request.

And since a server has to respond, it captures the search request for letter c, and returns the available results.



The server thereby sends the results into a pre-parsed html template allowing results to be displayed correctly. When you think the server is done, that is where you're wrong. The HTTP server **continues** listening for incoming requests and will continue responding and serving.

In other words, if a user decides to modify the number of quantities of the items available or hit the "Add to Cart" button, the server will also respond accordingly.

Main Features That Utilises the Concepts discussed in Go In Action:

1. Sessions

There are two types of main databases that are in the program.

They're called.

1) dbSessions
2) dbUsers

dbSessions is a database that stores sessions of users who are **logged** into the program. For users who are not logged into the program, they only have cookies but not sessions.

dbSessions ensure that as soon as verified users are logged, the other database dbUsers become available for usage.

Here's a quick glance at the code base to see how it looks:

```
11  sID, err := uuid.NewV4() //uuid package from satori
12      if err != nil {
13          fmt.Printf("Something went wrong: %s, err")
14      }
15  c := &http.Cookie{
16          Name:  "session",
17          Value: sID.String(),
18      }
19
20  c := &http.Cookie{
21      Name:  "session",
22      Value: sID.String(),
23  }
24
25  c.MaxAge = sessionLength
26  http.SetCookie(w, c)
27  dbSessions[c.Value] = session{un, time.Now()}
28
29  dbUsers[u.UserName].CartMapData = make(map[string]*cartFullData)
30  dbUsers[u.UserName].CheckoutMapData = make(map[string]*checkoutMapDataFull)
31
```

Assuming a User has already successfully signed up, or logged in, a new UUID value is generated for the user, converted from byte to string at line 17, line 20-23 adds the value into the http package's cookie struct. Thereafter, dbSessions gets a key with the

UUID value assigned and an instance of the *session* struct assigned to it. The *session* struct contains the following:

```
36  type session struct {
37      un           string
38      lastActivity time.Time
39  }
```

This is how dbSessions actually contains the last activity of the user as well as the username of the user.

Subsequently, at certain websites, the program constantly queries the user's browser for cookies that store the session ID and in turn, uses the session ID, to search through dbSessions to query for username.

Once we have the user name, we can utilise dbUsers to extract certain information.

```
45  type user struct {
46      UserName        string
47      Password        []byte
48      First           string
49      Last            string
50      Role            string
51      SearchLogs      *userSearchActivity
52      CartList        []cartData
53      CartDisplay     []cartDisplayData
54      CartMapData     map[string]*cartFullData
55      CheckoutMapData map[string]*checkoutMapDataFull
56      cartTransID     []string
57  }
```

And because dbUsers is a map with key as username, having the username gives up a good overview of the user's login details, search logs, active cart list as well as their checkout History and Record.

On top of regular sessions of logging in and out, the program is also equipped with a duplicate-log-in check using sessions. This ensures that a user can only be **logged in** at any **one** time on any **one** client. I.e. User A, logged in on chrome browser cannot be also logged in on a firefox browser. The implemented duplicate log function checks for duplicates and deletes previous sessions. Forcing that previous session to be logged out.

Here's a quick glance of the codes.

```
60  duplicateLoginCheck(u, w, req)
```

In the same login function, the u - user information, as w - write to HTTP header, req -request to server are sent as arguments to a function called duplicateLoginCheck

```
65  func duplicateLoginCheck(u *user, w http.ResponseWriter, req *http.Request) {
66
67      for k, v := range dbSessions {
68          if u.UserName == v.un {
69              delete(dbSessions, k)
70              break
71          }
72      }
73  }
```

This function takes in the arguments; parses the data in the map to check for dbSessions and if the **logged** username already exists in an existing session, the **previous** session is immediately deleted. To save on memory usage, the loop is immediately broken off with the **break** code as soon as the *if* condition is met.

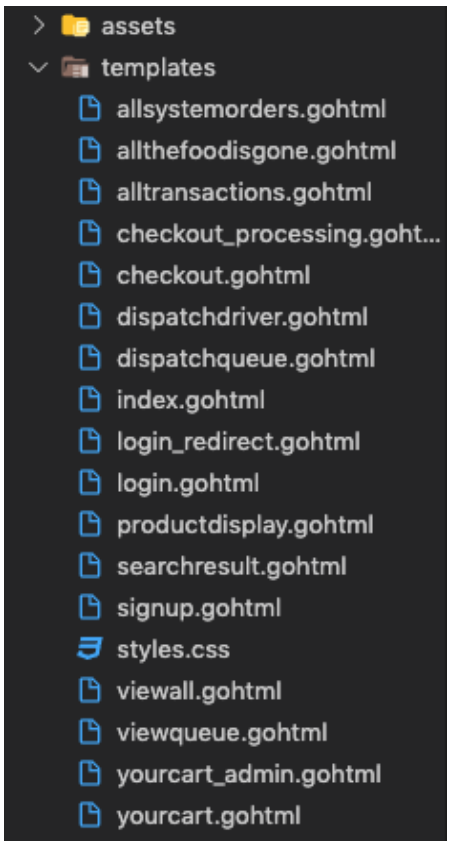
Note: var dbSessions is a global variable; accessible from everywhere within the same package

And since this only deletes the older session, the newer(current) session of the user is not affected.

2. Templates

The program comes with big heavy usage of templates.

With every site available, there is a matching .gohtml file. Here is a screengrab of the available HTML Template within the system.



HTML Templates are essential webpages that have at least a <head>, <body>, <footer> tag and have endless capacities.

Let's take a look at how a HTML template helps us retrieve important information; such as the sign up form for a user who would like to sign up for our program.

```
<form method="post">
  <div class="form-group pt-1">
    <label for="exampleFormControlInput1">Email address </label>

    <input type="username" class="form-control" name="username" placeholder="name@example.com">
  </div><p class="text-danger inline small">{{ .Username}}</p>

    <div class="form-group">
    <label for="exampleFormControlInput1">Password</label>
    <input type="password" class="form-control" name="password" placeholder="Enter your password here...">
  </div><p class="text-danger inline small">{{ .Password}}</p>
    <div class="form-group">
    <label for="exampleFormControlInput1">First Name</label>
    <input type="text" class="form-control" name="firstname">
  </div><p class="text-danger inline small">{{ .FirstName}}</p>
    <div class="form-group">
    <label for="exampleFormControlInput1">Last Name</label>
    <input type="text" class="form-control" name="lastname">
  </div><p class="text-danger inline small">{{ .LastName}}</p>
  <div class="form-group">
    <label for="exampleFormControlSelect1">Type of Account</label>
    <select class="form-control" name="role">
      <option>superuser#1</option>
      <option>Customer Service Officer</option>
      <option>Dispatch Supervisor</option>
      <option>Delivery Partner</option>
      <option>superuser#2</option>
    </select>
  </div>

      <input type="submit">
      <div class ="container pb-3"></div>

</form>
```

This form is designed to be a **post** method and with several field names such as username, password, firstname and last name etc. Now, User A can come along and submit the form, but what happens next?

You might have guessed it, the HTTP server we talked about earlier on is still listening! This new request is processed with signup function here:

```
120    if req.Method == http.MethodPost {
121        // get form values
122        un := req.FormValue("username")
123        p := req.FormValue("password")
124        f := req.FormValue("firstname")
125        l := req.FormValue("lastname")
126        r := req.FormValue("role")
127
128        // dbUser[un] = &users{un, p, f, l, r ..}
129    }
```

Since the form is of **post** method, line 122-126 gets activated, retrieving the user details and together with other fields, can be stored in dbUsers with un as the key to the map's key-value pair requirement.

On top of using HTML templates to store user values, we can also bring error checking to a notch. Let's understand this by following along the code, to see what happens after a value has been **retrieved**.

```
134    var me = make(map[string]string) //make map for error
135    boolresult, mapresult := validateInputs(un, p, f, l, me)
```

In line 134 above, we create a new map called *me*, that will be used for passing values into the function called **validateInputs**

```
141  func validateInputs(un string, p string, f string, l string, me map[string]string) (bool, map[string]string) {
142
143      rx := regexp.MustCompile("^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\\.[a-zA-Z0-9](?:[
144
145      if len(un) == 0 {
146          me["Username"] = "Username is not valid. Please Enter again"
147      } else if !rx.MatchString(un) {
148          me["Username"] = "Username is not valid email address. Please Enter again"
149      }
150      if len(p) == 0 {
151          me["Password"] = "Password is not valid. Please Enter again"
152      }
153      if len(f) == 0 {
154          me["FirstName"] = "First Name is not valid. Please Enter again"
155      }
156      if len(l) == 0 {
157          me["LastName"] = "Last Name is not valid. Please Enter again"
158      }
159      if len(un) != 0 && len(p) != 0 && len(un) != 0 && len(l) != 0 && rx.MatchString(un) {
160          return true, me
161      }
162      return false, me
163  }
```

With this function, we are receiving un, p, f, l , me - username, password, firstname, lastname and the map we created earlier on and returns 2 arguments.

1. Updated map with error code.
2. Boolean value - true or false

Imagine a username that is of 0 value? A new error code is created with key name "Username" is created.

me["Username'] = "Username is not valid. Please enter again"

But what is the username is a string of junk characters that doesn't appear to be an email? Then regexp comes to the rescue.

The rx variable as well as the matchstring function helps to check if the username's email entered is actually valid.

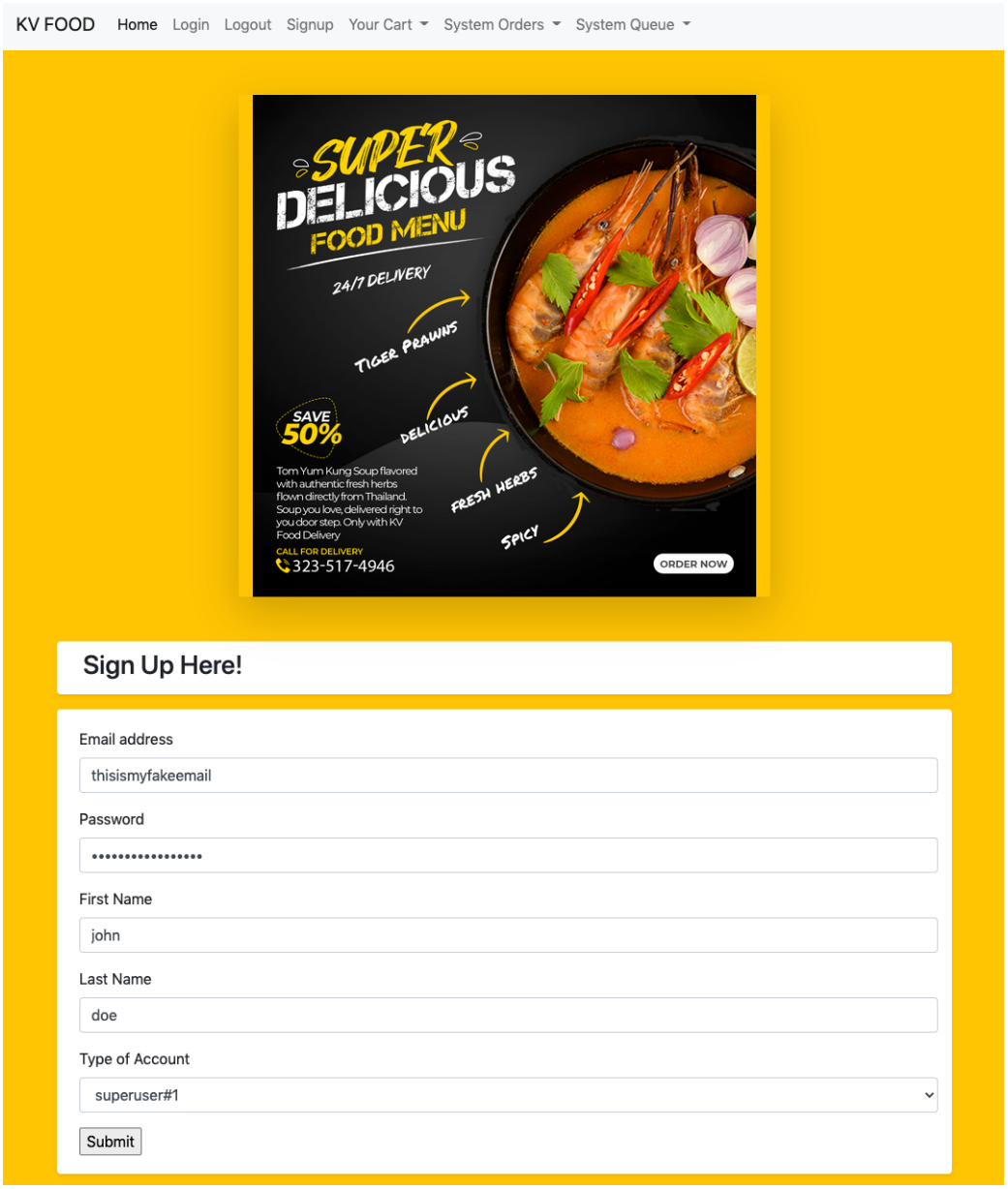
Now if a username of "notanemailaddress" is created instead , a new error code with the key "Username" is created.

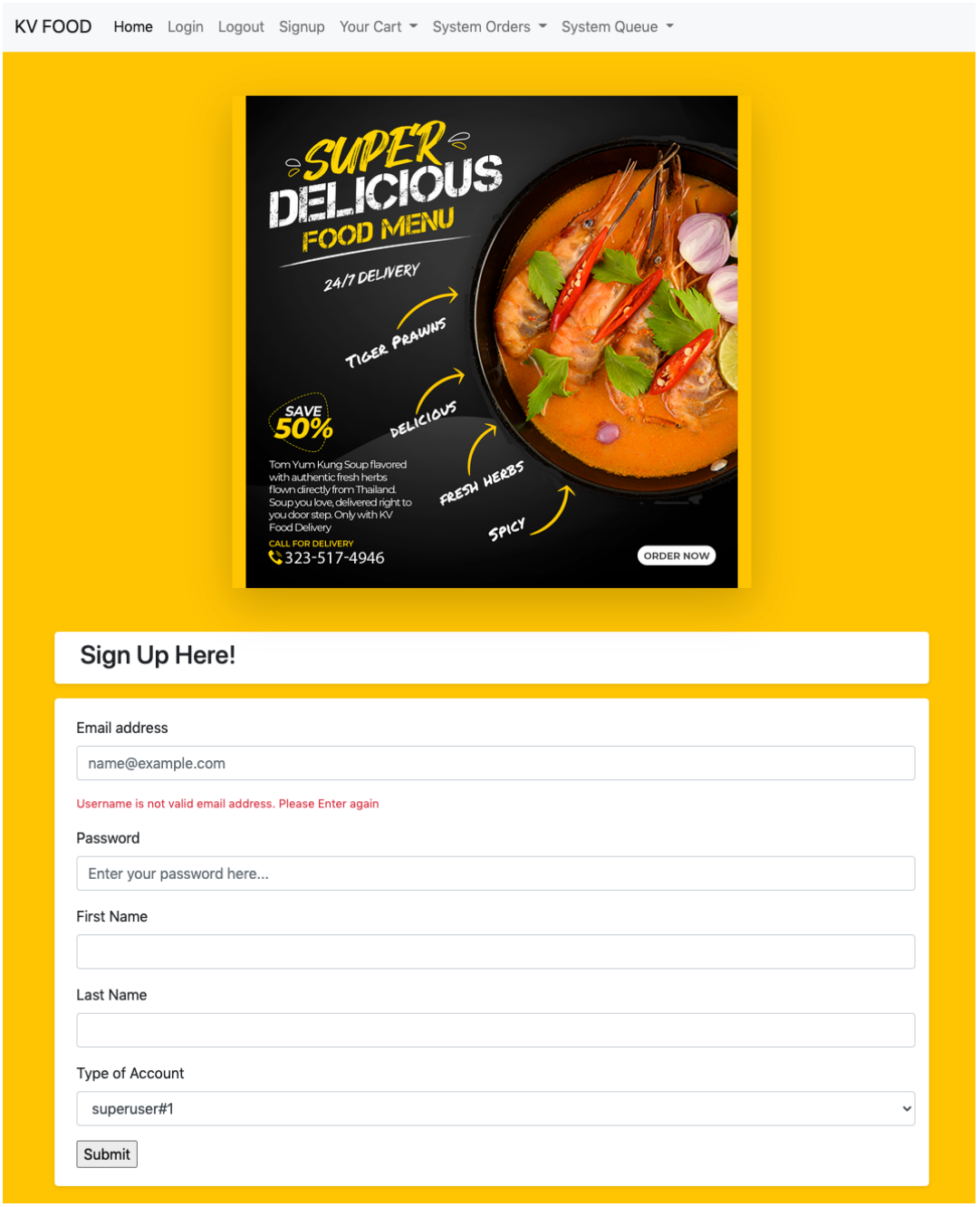
me["Username'] = "Username is not a valid email address. Please Enter again"

In other words, the program can intelligently check what's required and return helpful error messages.

The other parts of the function works similarly and if all the functions equate to be true, the function returns a **true**. Otherwise, a false would be sent back to the function as well as the map of Errors.

Here's a sample form before and after after we do error checking





See the error above? This is the work of function called validateInputs, we created an error message with the help of regex check and then passing the map value into theHTML page and then displaying the value in the sign up form.

```
169    if boolresult == false {
170        tpl.ExecuteTemplate(w, "signup.gohtml", mapresult)
171        return
172    }
```

tpl.ExecuteTemplate(w, "signup.gohtml", mapresult) is responsible for sending the error message into the HTML template, "signup.gohtml"

Last but not least, the HTML requires golang specific *parable* names in order for the server to know where and how to inject the values.

In the diagram below, we've utilised {{ .Username}}, {{ .Password}} {{ .Firstname}} as well as {{ .LastName}} to ensure that the error messages appear accurately.

If these values are not present in the map data(mapresult), then they will not be shown in the form. This ensures that if the username is not valid, an error for the username will be shown. If the username, password, first names are valid choices, then only an error message for the last name will be displayed.

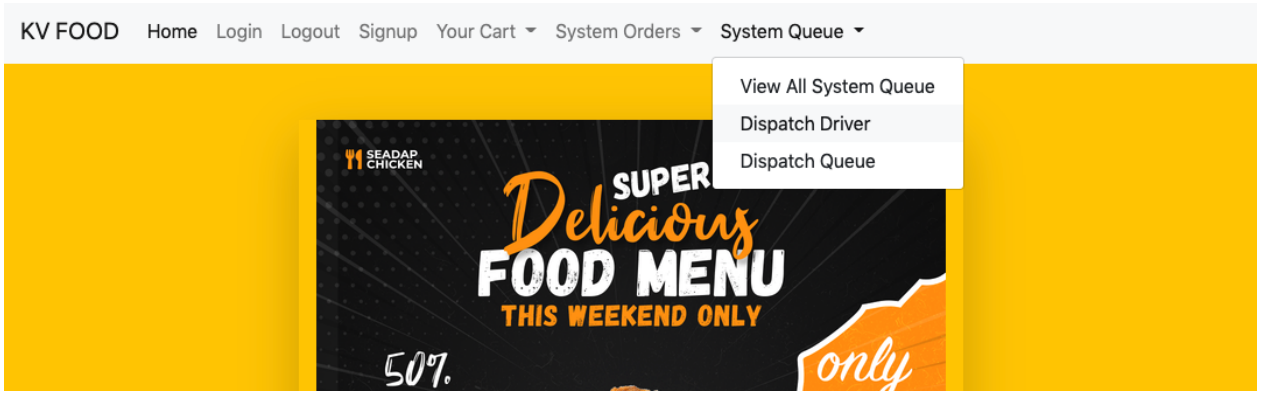
```
<div class="container bg-light ml-10 mr-10 rounded shadow-sm pt-3 pb-1 pl-4 mb-3 bg-white rounded">
<form method="post">
  <div class="form-group pt-1">
    <label for="exampleFormControlInput1">Email address </label>

    <input type="username" class="form-control" name="username" placeholder="name@example.com">
  </div><p class="text-danger inline small">{{ .Username}}</p>

    <div class="form-group">
    <label for="exampleFormControlInput1">Password</label>
    <input type="password" class="form-control" name="password" placeholder="Enter your password here...">
  </div><p class="text-danger inline small">{{ .Password}}</p>
    <div class="form-group">
    <label for="exampleFormControlInput1">First Name</label>
    <input type="text" class="form-control" name="firstname">
  </div><p class="text-danger inline small">{{ .FirstName}}</p>
    <div class="form-group">
    <label for="exampleFormControlInput1">Last Name</label>
    <input type="text" class="form-control" name="lastname">
  </div><p class="text-danger inline small">{{ .LastName}}</p>
  <div class="form-group">
    <label for="exampleFormControlSelect1">Type of Account</label>
    <select class="form-control" name="role">
      <option>superuser#1</option>
      <option>Customer Service Officer</option>
      <option>Dispatch Supervisor</option>
      <option>Delivery Partner</option>
      <option>superuser#2</option>
    </select>
  </div>

      <input type="submit">
      <div class ="container pb-3"></div>

</form>
```

3. Admin Features for Dispatch and Service Recovery

Part of the reason the project was created was to solve the teething issues of customer service and work on ways to improve service recoveries. That said, the program has 2 different parts that are accessible by the general public as well admin users.

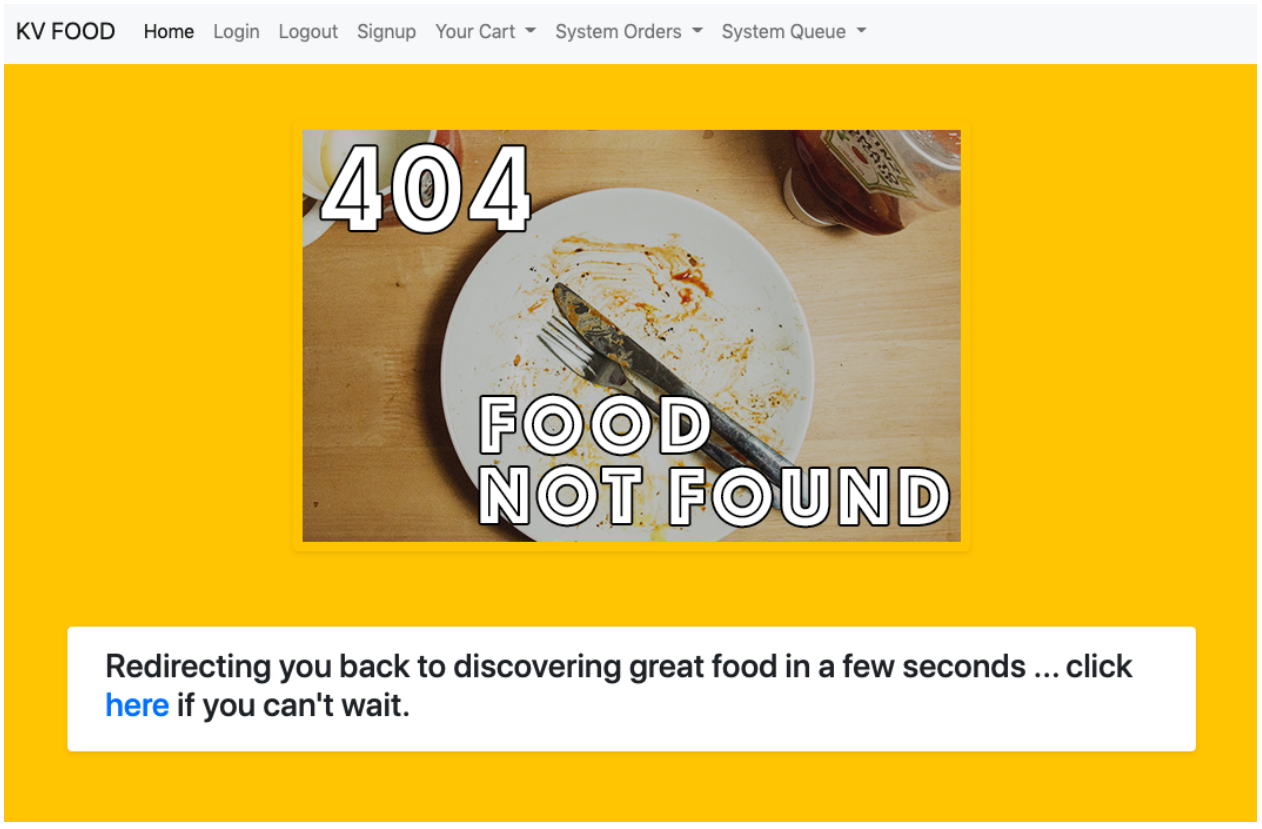
For the testing purposes of the program, an account registered as the default type, superuser#1 is allowed **access** on all menus.

If you have an account that is of a "Customer Service Officer Type" instead, you will not be able to access menus such as the System Queue Menu > Dispatch Driver Menu



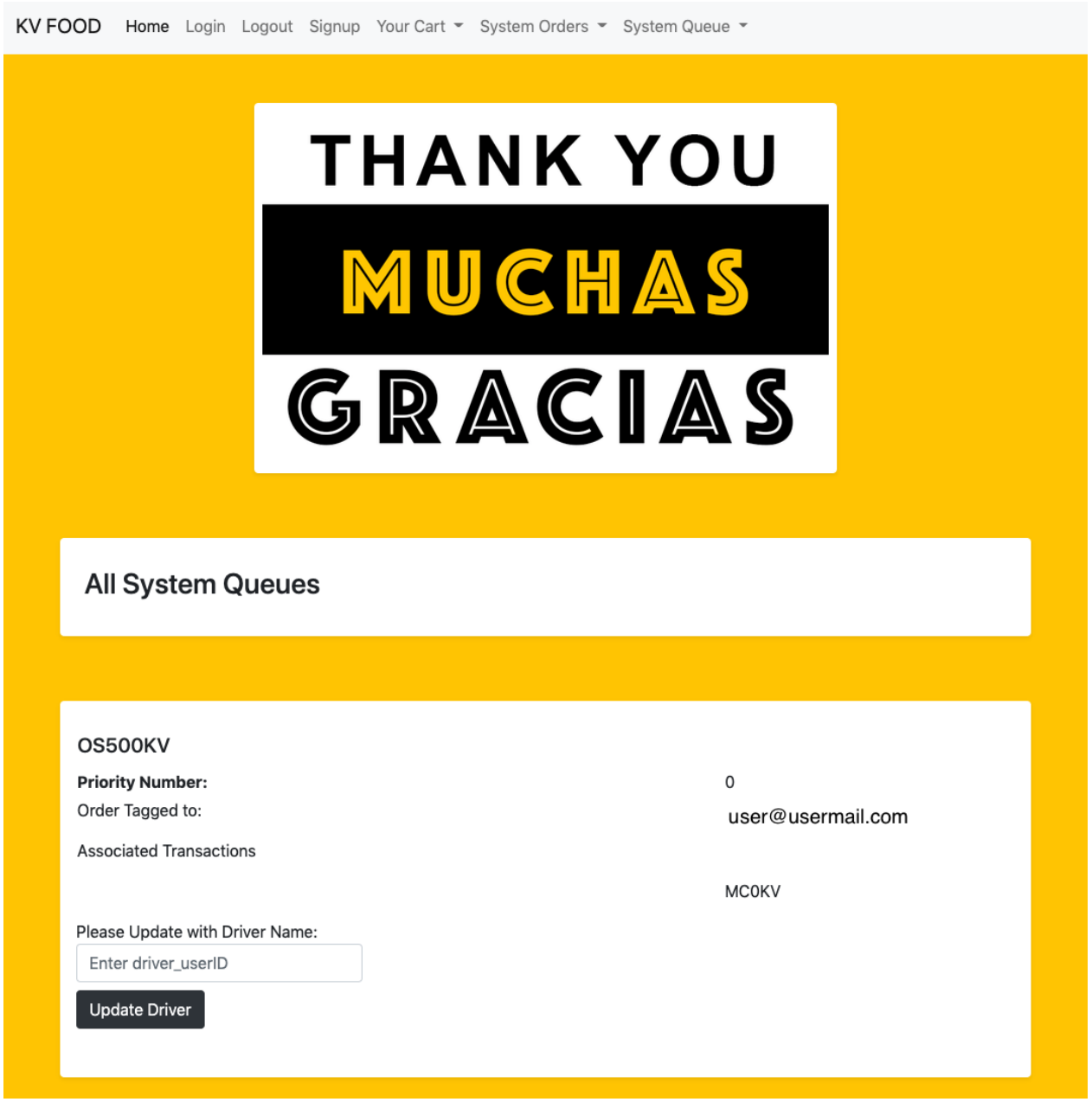
Doing so, should show you an error message, but instead the program tells you that the page cannot be found, and sends you back to the home page.

Here's how it looks like if you have a normal account, without superuser/ Dispatch Supervisor privileges:

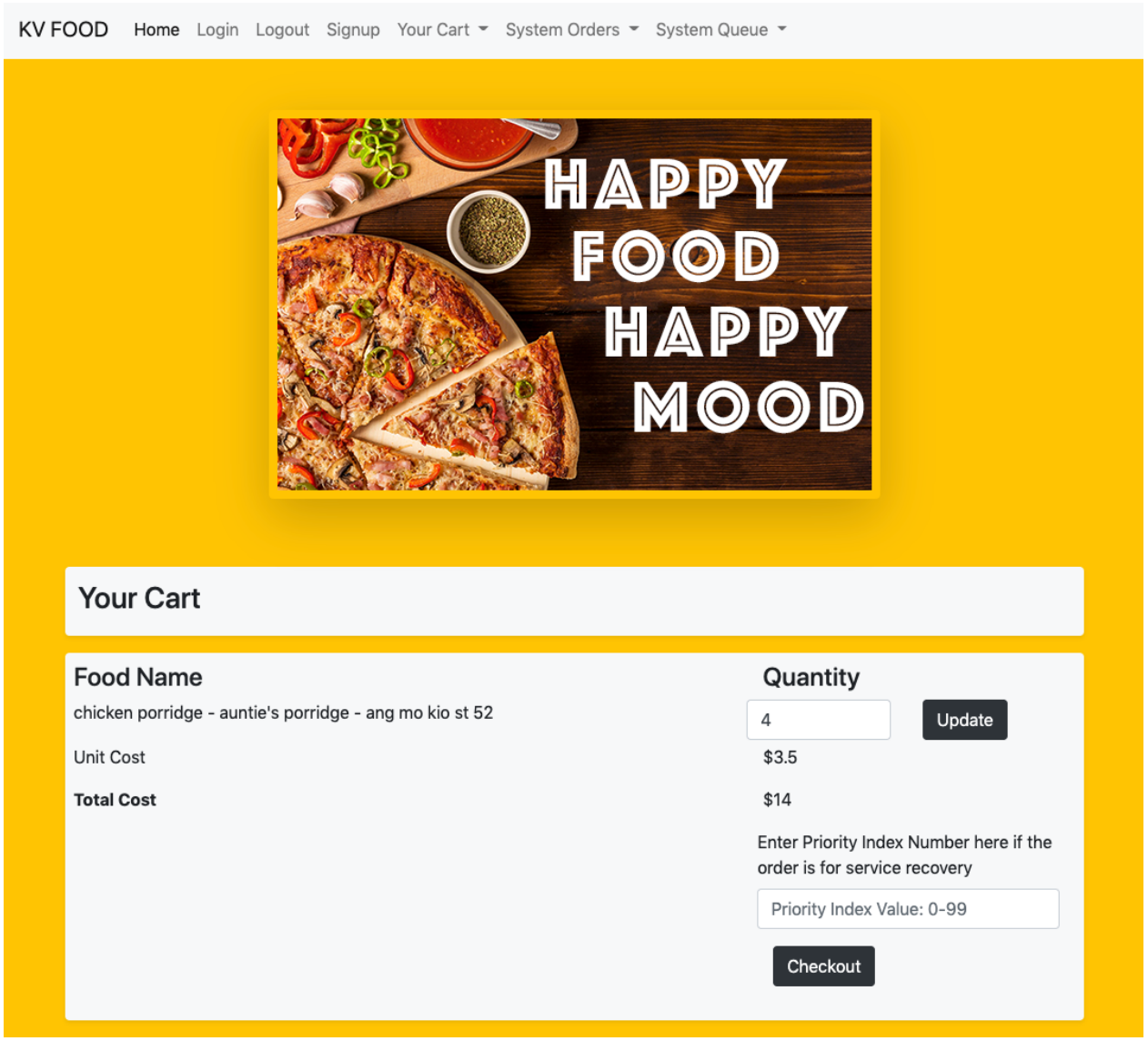


And if you have an account with a role of "Dispatch Supervisor", you will see something else such as this:

Notably, you have the update driver button; where you can assign driver names to specific drivers.



As for service recovery, a special cart page is also only accessible with users that are logged in with roles "Customer Service Officer" or superuser#1



In order to help with service recoveries, orders that are checked out with higher priority values will be placed further up in the queue system employed in the system. Please visit the data structure section (at the end of this document) to learn more about the type of priority queue system employed in the system.

Here's how a regular user without privileges would see their cart page.

KV FOOD Home Login Logout Signup Your Cart System Orders System Queue

Your Cart

Food Name	Quantity	
chicken porridge - auntie's porridge - ang mo kio st 52	1	Update
Unit Cost	$3.5	
Total Cost	$3.5	

Checkout

Channels for Data Management and Performance Enhancement

Implementation of different GO Routines, at the right place of the code allows for data to be initialized and populated accurately.

Let's look at how it is being implemented in the code:

```
212     // usernameDS := InitUsernameTrie() //inits Trie data for username DS
213     go CreateFoodList(ch) //newResult is a slice that is being returned byCreateFoodList function
214     fmt.Println("\nSystem Message :", <-ch)
215     go CreateFoodListMap(ch)
216     go MyFoodListDB.PreInsertTrie(FoodMerchantNameAddress, ch) //populates Trie Data for Food LIst
217     fmt.Println("System Message :", <-ch)
218     fmt.Println("System Message :", <-ch)
219     myPostalCodesDB := InitPostalCode()   //creates PostalCode BST DB
220     myPostalCodesDB.PreInsertPostalCode() //preinset POSTAL Code DB
221     FoodMerchantNameAddressProductID()
222     fmt.Println("System Message : System is Ready", currentTime.Format("2006-01-02 15:04:05"))
223
```

At line 213, the first GO routine has been set to run.

```
go CreateFoodList(ch)
```

This line of code, *CreateFoodList* is solely responsible for populating a slice data, from other forms of data that can be found in the rawData.go file and at the same time, passing a channel argument into the function.

As the slice data is of paramount importance, we need the data to be initialized, imported and created correctly. To ensure that the GO Routine **runs** the entire operation, we use a blocking code at line 214

```
fmt.Println("\nSystem Message :", <-ch)
```

<- *ch* in this instance is called a blocking of code because the channel is expecting to receive a message from the function that was called in line 213.

Once the Go routine has completed the function, at the end of the function *CreateFoodList(ch)* found in mainFunctions.go sends the following string data into the channel :

```
ch <- "Mandatory - Food List Data Generated"
```

This is why line 213 code is always executed in the terminal before the other codes.

```
7:goassignment2 admin$ go run database.go databaseMaps.go main.go mainFunctions.
go messageTemplates.go queueManagement.go rawData.go

System Message : Mandatory - Food List Data Generated
```

And the GO Routines found in lines 215, 216 are all set off together, to ensure that the GO Scheduler is able to manage these 2 processes concurrently.

```
212     // usernameDS := InitUsernameTrie() //inits Trie data for username DS
213     go CreateFoodList(ch) //newResult is a slice that is being returned byCreateFoodList function
214     fmt.Println("\nSystem Message :", <-ch)
215     go CreateFoodListMap(ch)
216     go MyFoodListDB.PreInsertTrie(FoodMerchantNameAddress, ch) //populates Trie Data for Food LIst
217     fmt.Println("System Message :", <-ch)
218     fmt.Println("System Message :", <-ch)
219     myPostalCodesDB := InitPostalCode()   //creates PostalCode BST DB
220     myPostalCodesDB.PreInsertPostalCode() //preinset POSTAL Code DB
221     FoodMerchantNameAddressProductID()
222     fmt.Println("System Message : System is Ready", currentTime.Format("2006-01-02 15:04:05"))
223
```

The blocking codes in line 217 and 218 ensure that these other 2 GO routines are run and completed, forcing a return message. This is how the application ensures that the data are all populated before allowing the system to be ready for usage.

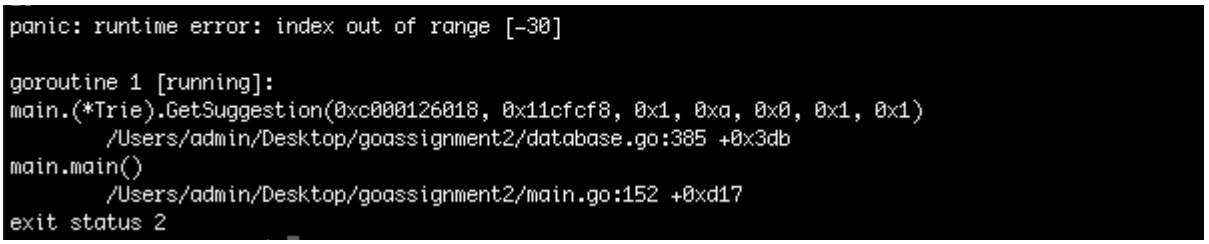
```
System Message : Mandatory - Food List Data Generated
System Message : Food List Map Data Completed
System Message : Food List Search Auto Complete Database Updated
System Message : System is Ready 2021-04-04 17:30:35
********
```

As seen in the screengrab of an actual application usage, system messages received by the channels ensure the data is correctly implemented so features queried and displayed can be accurate at all times

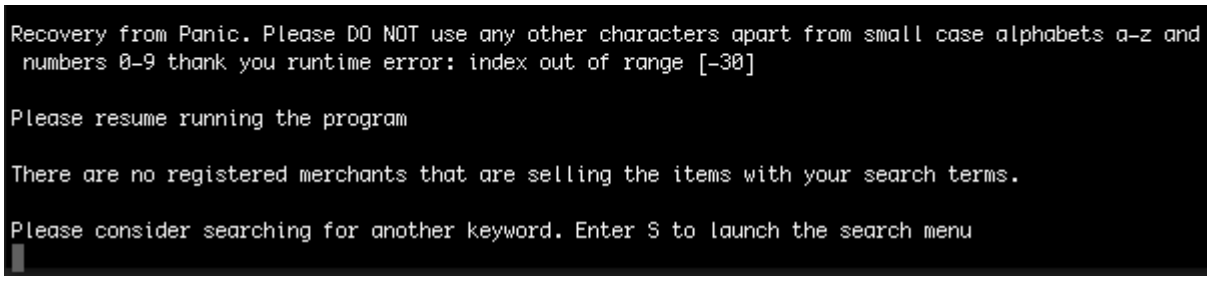
Error, Panic and Recovery Handling

As required by the Error and Panic considerations, error handling in the program was designed by anticipating the possible errors that could happen in code. As briefly described before, the most vulnerable part of the code has to do with retrieving information on data structures that have fixed capacities.

Highlighted in the data structure implementation of the prefix Trie (found at the end of this document) , panics can happen when characters that **do not** belong to the 39 characters get searched for. Here are two screenshots that showcases a screengrab without panic considerations and one with panic considerations.



In the picture above, without proper panic considerations, the program **is forced to exit.**. In an *exited* state, a user cannot continue the usage of the webapp.. Reloading of the app would require time and effort; especially when importing big datas can be very time and memory consuming.



In the above diagram, panic and recovery codes were implemented right in the function that would cause the panic. Below, we'll see how the code is implemented.

```
329  func (t *Trie) GetSuggestion(query string, total int) []string {
330
331      defer func() {
332          if r := recover(); r != nil {
333              fmt.Println("\nRecovery from Panic. Please DO NOT use any
334              fmt.Println("\nPlease resume running the program\n")
335          }
336      }()
337
338      var result []string
339      //move to next position node from the searching character
340      currentNode := t.root //starts from root.
341
342      r := []rune(query)
```

In lines 331 of the function *GetSuggestion()* , we deferred a recovery function. In a nutshell, a deferred function can fire even during a panic, making this code a useful one for serving and recovering from a crash. And since this is the core function that searches for elements that may potentially go beyond the character limit, we know that this function is also the one that will crash.

With the firing of a deferred recovery code, the power is returned to the block of code that is responsible for calling *GetSuggestion()*.

```
28      localsearchResult := MyFoodListDB.GetSuggestion(sr, 50) // you will always append a global variable so you pass data this
29
30      dbUsers[u.UserName].SearchLogs = &userSearchActivity{SearchResults: localsearchResult}
31
32      http.Redirect(w, req, "/searchresult", http.StatusSeeOther)
```

Tracking back to the file *functionhandlers.go*, we see that it is code line 24 that calls *GetSuggestion()*.

Since power returns to the main function thanks to the recovery code, the main function **continues running**. In this case, the next function in line 30 continues.

Since the previous function panicked, there was no value assigned to *localsearchResult*. Regardless of results, line 32 redirects to /searchresult

As you might guess, since there are no search results, there are also no results to display.

In the below image, you'll observe that there are indeed no search results since it panicked. The terminal window also shows the panic recovery mode. Hitting the back button does resume operations so the recovery function did help.

KV FOOD Home Login Logout Signup Your Cart System Orders System Queue

Didn't find what you want? How 'bout searching for it again?

goassignment3_2 — main • go run main.go database.go databaseMaps.go functionhandlers.go mainFunctions.go messageTemplates.go queueManagement.go rawD...

```
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265003298033464 ext:8558569435 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265003299980464 ext:8560516562 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265004267642288 ext:9454437611 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265004269587288 ext:9456382682 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265004993585288 ext:10180381845 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265004995271288 ext:10182067873 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstore@gmail.com lastActivity:{wall:13840265007178881936 ext:12218198552 loc:0x16273c0}}********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

********
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstor2e@gmail.com lastActivity:{wall:13840265623387294912 ext:586099680872 loc:0x16273c0}}********
3d59c7e6-68ff-4210-9f29-0177bc9c08e2 kayvarstor2e@gmail.com
58c64069-5e45-4f6c-81c6-ea1139a2c529 kayvarstore@gmail.com

{un:kayvarstor2e@gmail.com lastActivity:{wall:13840265623389891912 ext:586102277977 loc:0x16273c0}}
Recovery from Panic. Please DO NOT use any other characters apart from small case alphabets a-z and numbers 0-9 thank you runtime error: index out of
range [-61]

Please resume running the program
```

On top of that, the recovery function can only help prompt users in a helpful way regarding the error that occurred. You can see a prompt above that reads *Recovery from Panic. Please DO NOT use any other characters … " etc.*

Apart from Panic and Recovery functions, due care on error handling can also be found in the code. Let's visit an example below:

```
90          sID, err := uuid.NewV4()
91          //err handling
92          if err != nil {
93              fmt.Printf("Something went wrong: %s, err")
94          }
95
96          c := &http.Cookie{
97              Name:  "session",
98              Value: sID.String(),
99          }
```

As mentioned before, the *dbSessions* variable has UUID values as key names. But what happens when the package doesn't work as expected? What happens when the uuid.package fails to fire?

In the ideal situation, uuids are generated; and stored in dbSessions along with the accompanying key-value values. But when it doesn't work, we take care of that error 2 ways.

1. *fmt.Printf("something went wrong: %s", err)*
2. *log.Fatalln("something went wrong", err)*

When using option 1, it prints to the terminal on the error, but the program continues running and since this is not a complete go-production-live program, option 1 is adequate to help in immediate debugging.

Option 2 can and should be used since UUIDs are mission critical when it comes to allowing the program to monitor the active sessions. Without proper UUID values, users can log multiple sessions and constantly overwrite their own database issues/ cart etc.

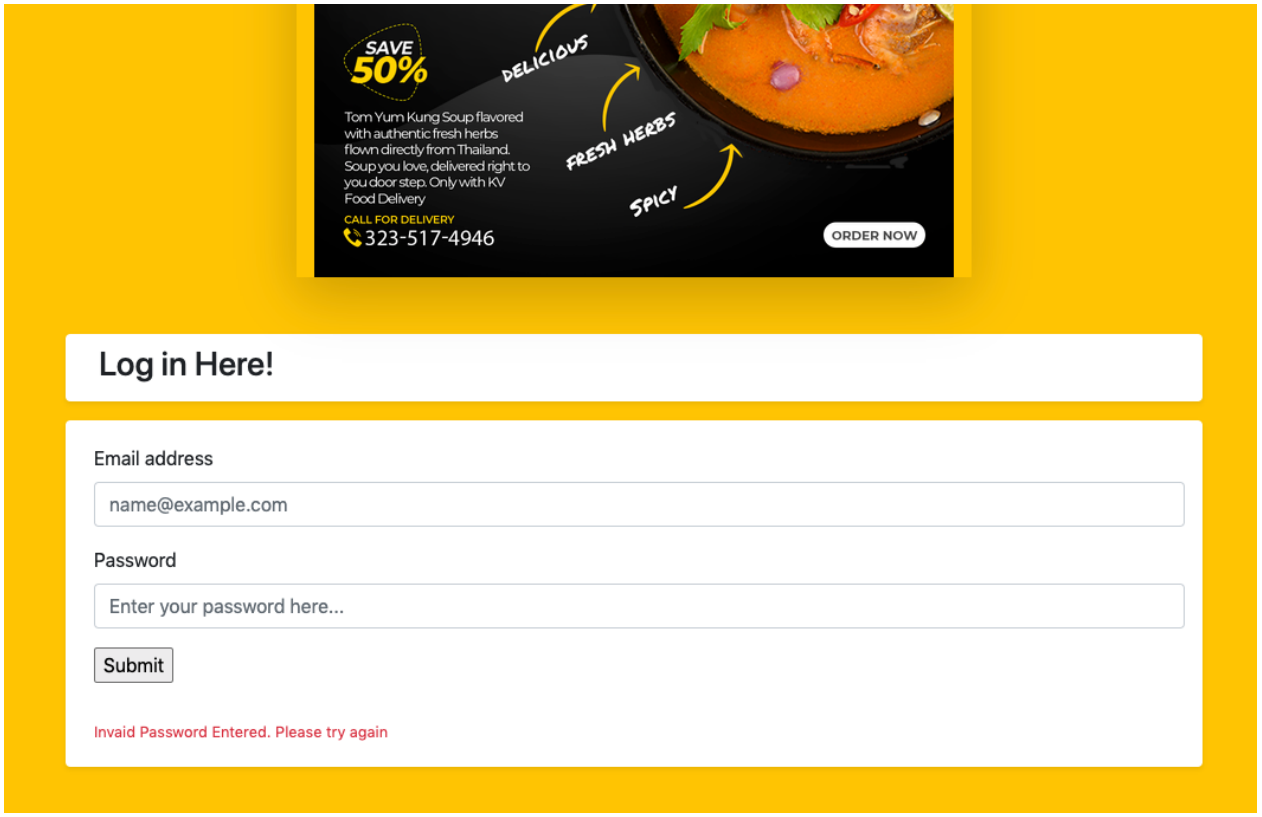
Other notable error handling mechanism incorporated in the code includes the usage of the bcrypt package

```
169         err := bcrypt.CompareHashAndPassword(u.Password, []byte(p))
170         if err != nil {
171
172             me["Password"] = "Invaid Password Entered. Please try again"
173             tpl.ExecuteTemplate(w, "login.gohtml", me)
174
175             return
176         }
177
```

In the above image, we can see the code snippet from func login, in functionhandlers.go in line 169. When a user attempts to login and right after the username has been verified to be a correct one (that's available in the dbUsers), the program now needs to authenticate the user's passwords.

Utilising the bcrypt package and the accompany method *CompareHashAndPassword* that takes in u.Password *[u= user]* and the entered password of []byte, the method returns an error if the password is of a **wrong** value

We can similarly create an error message using the map data structure and send it right into the login.gohtml form if the password is wrong.



Data Structure Implementation

Prefix Trie - Employed for Search Results

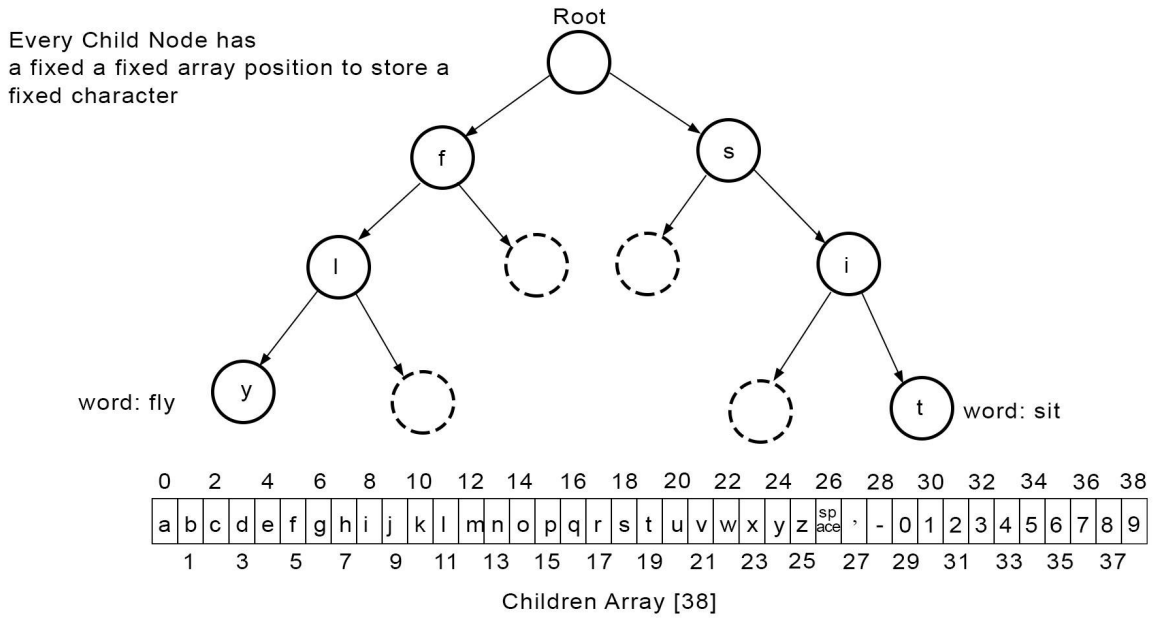


Diagram for a Prefix Trie

A prefix trie, also known as a trie is a tree-like data structure where all the nodes store letters of an alphabet, or a set of characters predetermined by codes.

The array size for each child is **39** where tries that only hold alphabets in either lower or upper case alphabets excluding symbols and numerical values can have an array size of 26.

While tries often take up more space in the memory, they are also very efficient in handling search data. In the application, we don't just store single-word keywords in the food list database. In fact, the trie data structure is also used to store the usernames in the application.

This is an example of what is being stored in the customized Trie structure.

roti prata - ah sik's curry - tampines ave 7

In the above example, you will realise that if we chose a regular 26 array sized Trie, the program would not be able to accommodate the other characters. These characters would not be able to be inserted into the trie.

The *space* , the *apostrophe*, the *hyphen* and the number *7* will be the affected characters.

And because each child node is fixed in size, it is guaranteed that as long as the characters that we are trying to store are within the 39 characters, the trie data structure **will not cause panics** but be a reliable, searchable data structure.

Should a panic occur, the deferred recovery function should also be able to take care of it.

Please refer to the diagram above to understand the 39 characters that can be stored

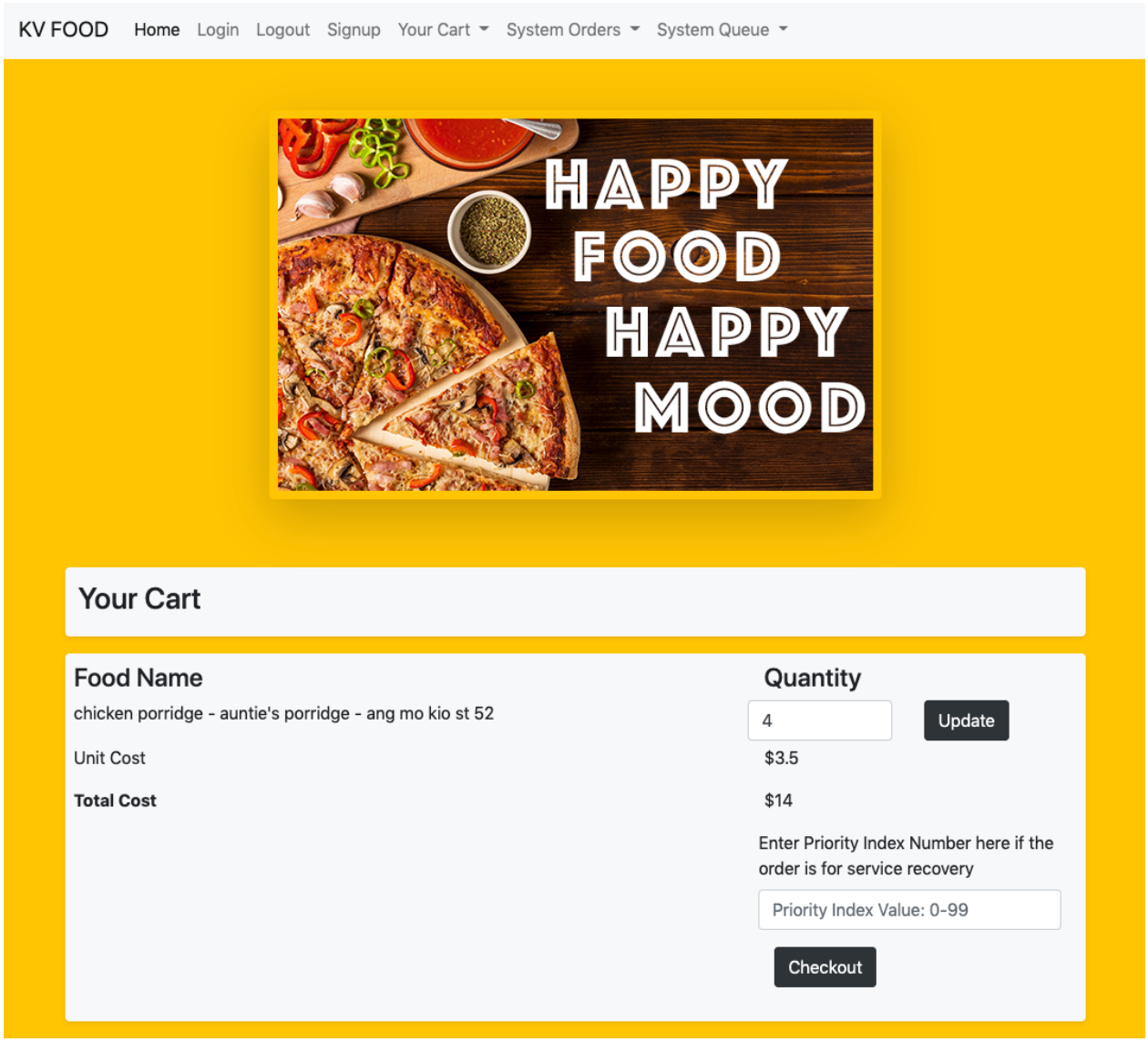
Priority Queues

The core business requirement of the KV Food Delivery business requires orders placed by customer service officers as well customers to be served in a first in first out approach.

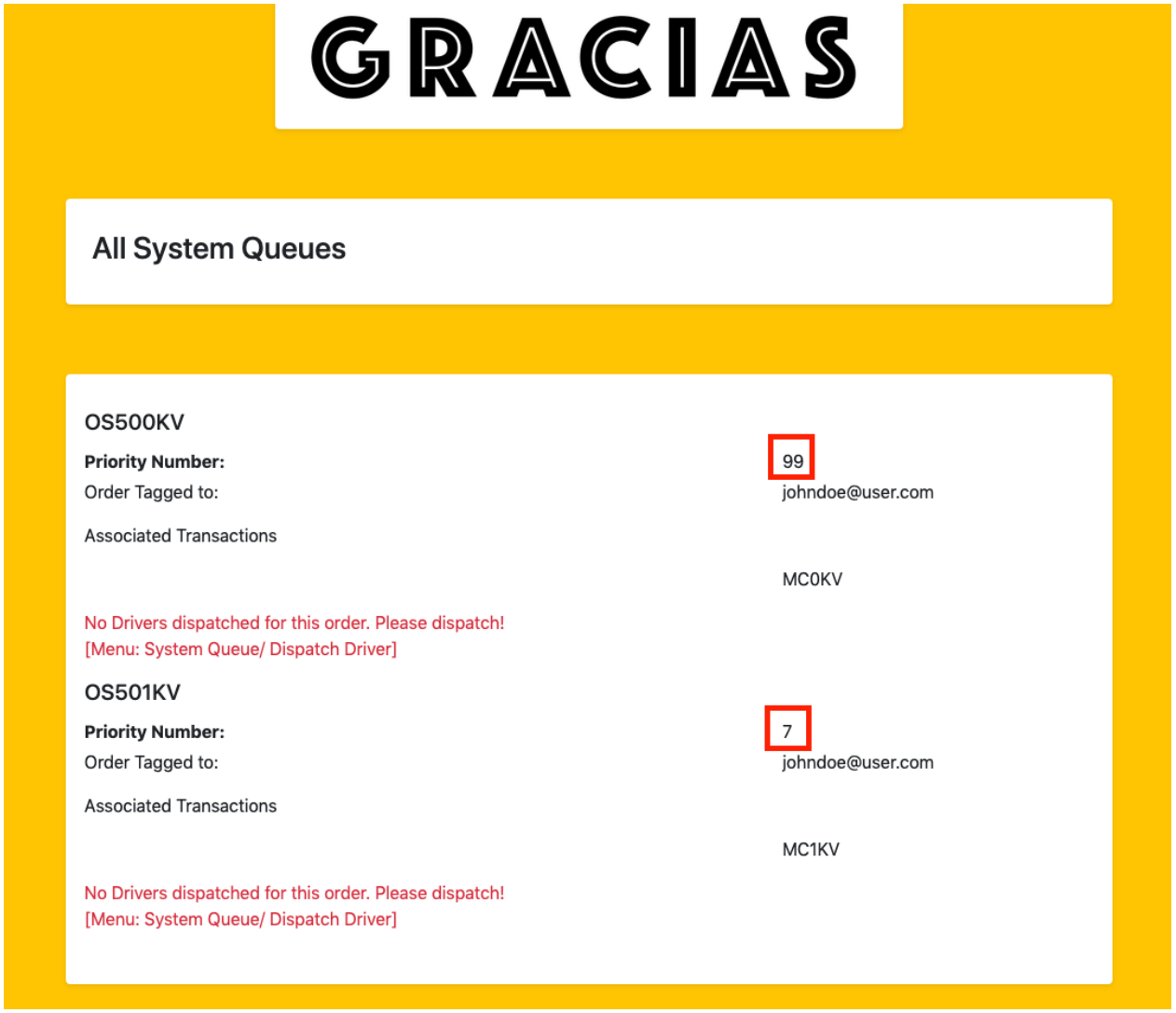
In other words; if an order, named **order A** is made now, and another order, **order B** is made a second later, order A should be processed first.

While that is correct, the aim of the KV app is to improve customer satisfaction and loyalty. In the event that there is a service recovery required, an admin that is logged in, is able to make changes to the queue by making an order of a higher priority than the others.

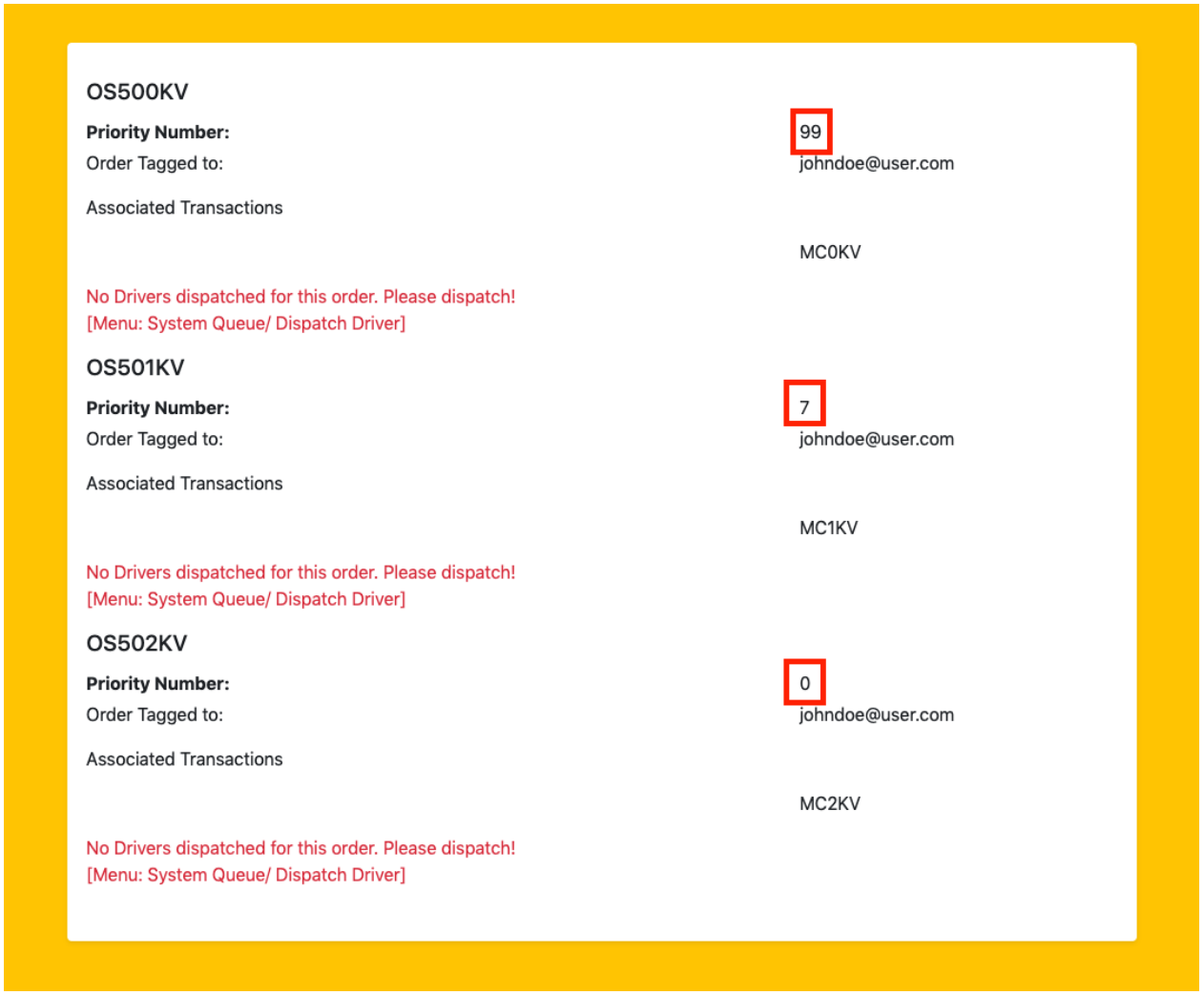
This can be achieved easily by setting numbers that are more than 0, when a user of "custome service officer" or "superuser#1" role is logged in, in the cart section.



If a number such as 99 is added to portray an order of significant importance such as an order meant for service recovery, you can visually see that the queue is amended, placing this order on the first of the list. (right before an order of priority 7) This ensures that delivery partners will see the order first; thereby starting the service recovery process.



However, if another priority value such as 0 is set, which is the default value, the order gets added to the last of the queue, automatically.



Queues can be also removed automatically when the dispatch menu is selected. You can imagine a dispatch supervisor with the correct privileges to be able to add an order or dispatch one if he/she thinks that there is a delivery partner/rider that is able to commit and fulfill the order.

Here's how a dispatch supervisor dispatches a system order for collection:

All System Queues

Please ensure drivers are dispatched for system orders before dispatching for queue.

Orders on the top of the liste are the first to be dispatched.

Dispatch OS500KV

OS500KV

Priority Number: 99

Order Tagged to: johndoe@user.com

Associated Transactions

MC0KV

Delivery Driver ID: KVR25

OS501KV

Priority Number: 7

Order Tagged to: johndoe@user.com

Associated Transactions

MC1KV

No Drivers dispatched for this order. Please dispatch!
[Menu: System Queue/ Dispatch Driver]

OS502KV

Priority Number: 0

Order Tagged to: johndoe@user.com

Associated Transactions

MC2KV

No Drivers dispatched for this order. Please dispatch!
[Menu: System Queue/ Dispatch Driver]