Syracuse University

Internet Programming

CSE 686

JaJang: Restaurant Management Platform

Author:
Runbo Zhao

JaJang Team member:
Boyang Chai
Chang Liu
Chunyang Zhan
Hongxin Wu
Jing Tian
Jiecheng Chen
Xinyu Zhang
Yayi Zhang

May 6, 2018

Abstract

In this project, we design an online order platform named Jajang.Cloud. It is a platform which provide interfaces for both merchants and customers. For merchants, they can register for their own restaurants and upload their menus. Regardless of dine in or delivery, our platform can help the merchants deal with customers any request and reply with corresponding response. It can save both time and space for the merchants. And for customers, they can order a delivery online via our platform, or order food and ask for service when dinning in the restaurant by scanning a QR code that the merchant get from our platform. So our purpose is to provide convenience for both merchants and customers. We utilize several techniques to finish the program including AngularJs, Node.JS, TypeScript and so on.

Contents

1 Project Introduction

1.1 Motivation

The catering is highly competitive, new restaurants come out every day. In order to beat the restaurant industry competition, several key factors can be considered. Exquisite menu, customer service, recommends system, restaurants environment information, etc. A well- developed platform is highly needed for those operators to gain advantages. They are able to customize their menu to fit guests needs, make changes to design or update food information through this platform.

Also, online order system increased in popularity during recent years. Customers prefer online food order to avoid fumbling over phone, pressure of long waiting line or bed reception experience. Unsatisfied experience can result in customers spending less or no further visits. Research data shows a group of restaurants with online order system will have larger customers size on average. With online order, customers can make their decision without pressure, take their time and be precise. Furthermore, customers data and order history can be collected for analysis. With customers and orders records, restaurants can make improvement, offer promotions and adjust menu.

JaJang.cloud is such an integrated restaurant management platform that will benefit both restaurants owners and customers. It is also a good assistant to help people effectively manage their restaurants if they are short of employees.

1.2 System Architecture

Our project can be divided into three main parts. They are JaJang (which is our platform), merchants, and customers. JaJang provides service and manages merchants and customers. Customers consume at restaurants (merchants), and at the same time, restaurants provide service to customers. You can see figure 1.1.It clearly represents the relationships among these three components.

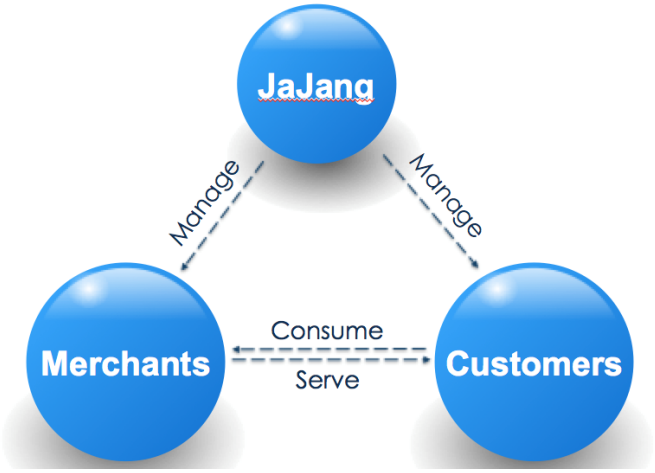


Figure 1.1: JaJang system architecture

The graphs above are class diagrams of our project and they show the relationship among different classes. There are seventeen components in total.

2 Technical Stack

Figure 2.1 shows what kind of technical skill we used in our project.

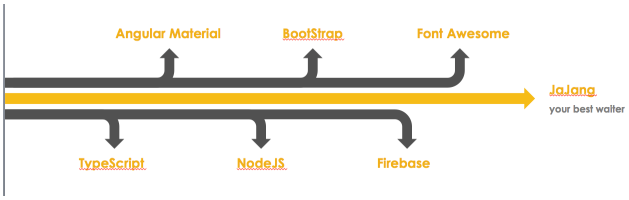


Figure 2.1: Technical stack

Angular Material, BootStrap 4 and ng-bootstrap are used to build beautiful style of pages including some animation and some fashion web component like card, carousel, modal and dropdown button. Also BootStrap 4 and ng-bootstrap is also well designed for responsive pages, which is suitable for using JaJang on both PC and mobile platform.

The combination of Angular 5 and Firebase is used to design a real-time system which is the best choice for JaJang management platform which means you

neednt to refresh the page everytime.

In order to make full use of Angular 5 and Firebase such a wonderful tools, TypeScript is chosen to used. Typescript overcomes a lot of defects of JavaScript and also it includes ES6 features which is good designed for OOP.
Also, to make JaJang more functional, Some third-part modules is imported in JaJang such as: e-ngx-print (https://github.com/laixiangran/e-ngx-print), ngx-qrcode(https://github.com/nacardin/ngx-qrcode),

neu-charts(https://www.npmjs.com/package/neu-charts) and angular firebase develop tools(https://github.com/angular/angularfire2). Thank to these great developers for providing such useful open-source modules.

3 Features

Our project can be basically separated into two main sections. One is merchants section, which includes some major functions of our project such as registration, restaurants management, QR code generating, and statistic analyzing. Every single function has several detailed knowledge points. The other section is about customers. Customer section includes some functions and technology points as well. We will explain it in detail in this report.

3.1 Registration

Our project provides two ways for users to login. One way is to allow people login with their own personal emails. Another way is using Google account to access our project. See figure 3.1.

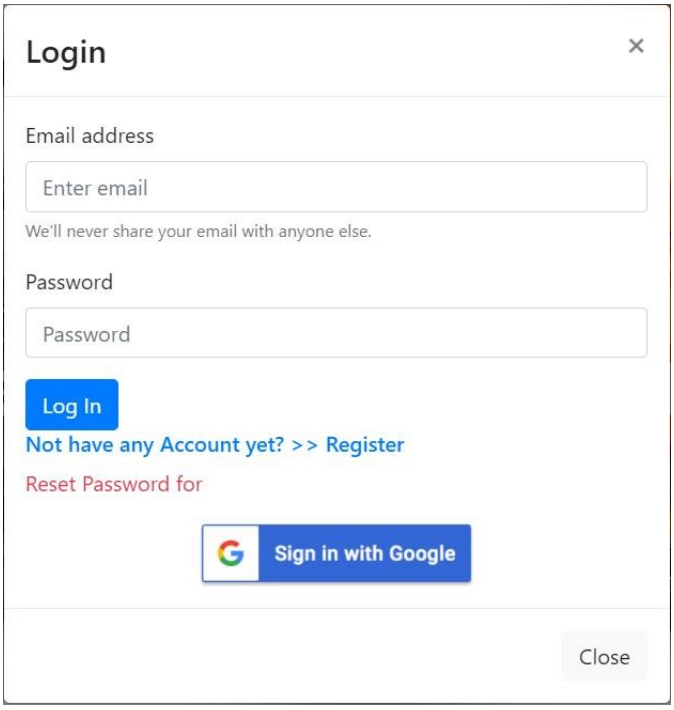


Figure 3.1: Login

If you do not have an account for our website, you can register one first. If you already had one account, but you forgot it, you can reset your password in login page. See figure 3.1 and figure 3.2. After you registered with your personal email, our system will send you a verify email to the email you just resisted. Then you just need to follow the message in the verify email to finish the registration. See Figure 3.3. Whats more, we also add password validation into our registration component. See figure 3.45.

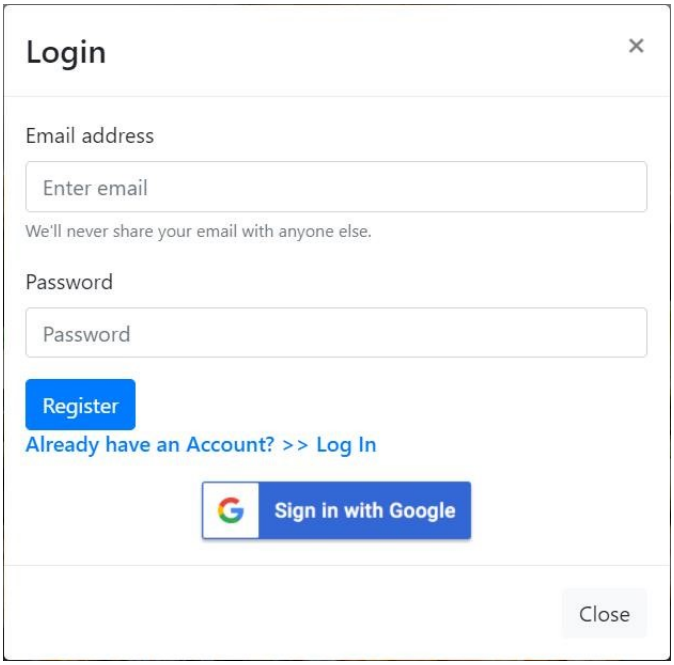


Figure 3.2: Register

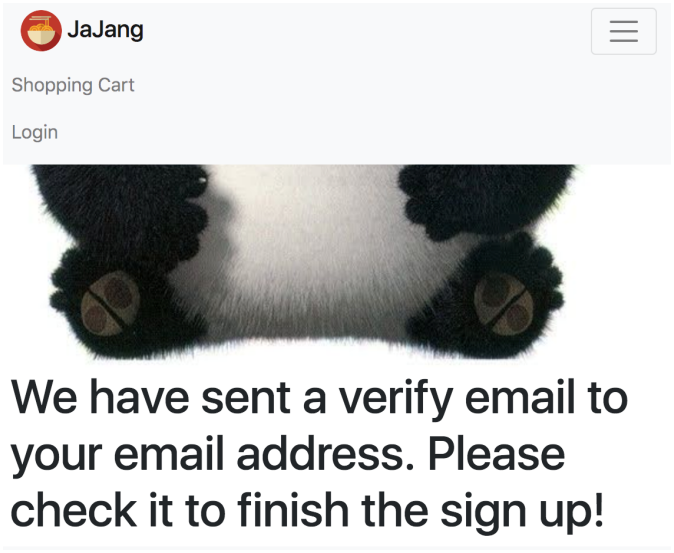


Figure 3.3: Verification email

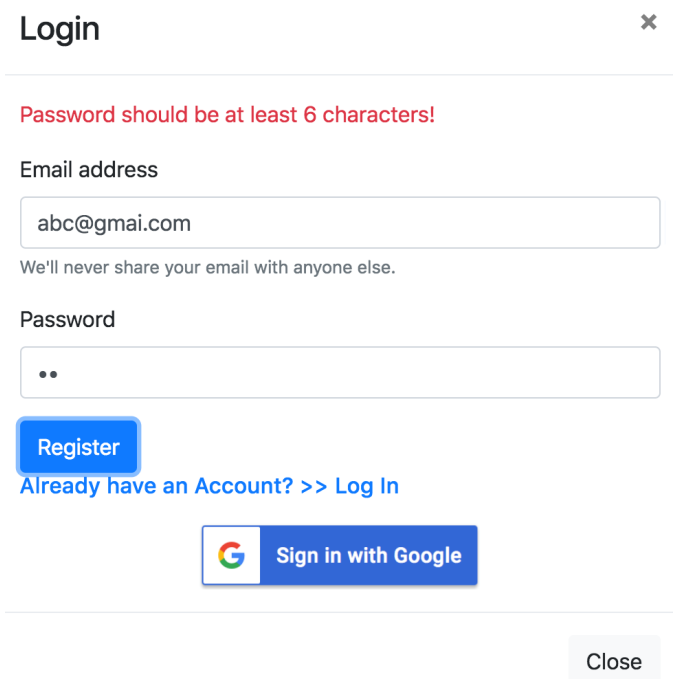


Figure 3.4: Form validation

We realize the login part including login with e-mail address, login with google account and reset password. We set the return url in each function to return to the webpage before the user login. We check the user in our firebase database to verify whether the user exists and get related information back. And we also set the error message when user sign up and login. The log in and sign up is written in the navbar. The following code is the code of BsNavbarComponent. And the detailed implement of log in and sign up is written in the AuthService.

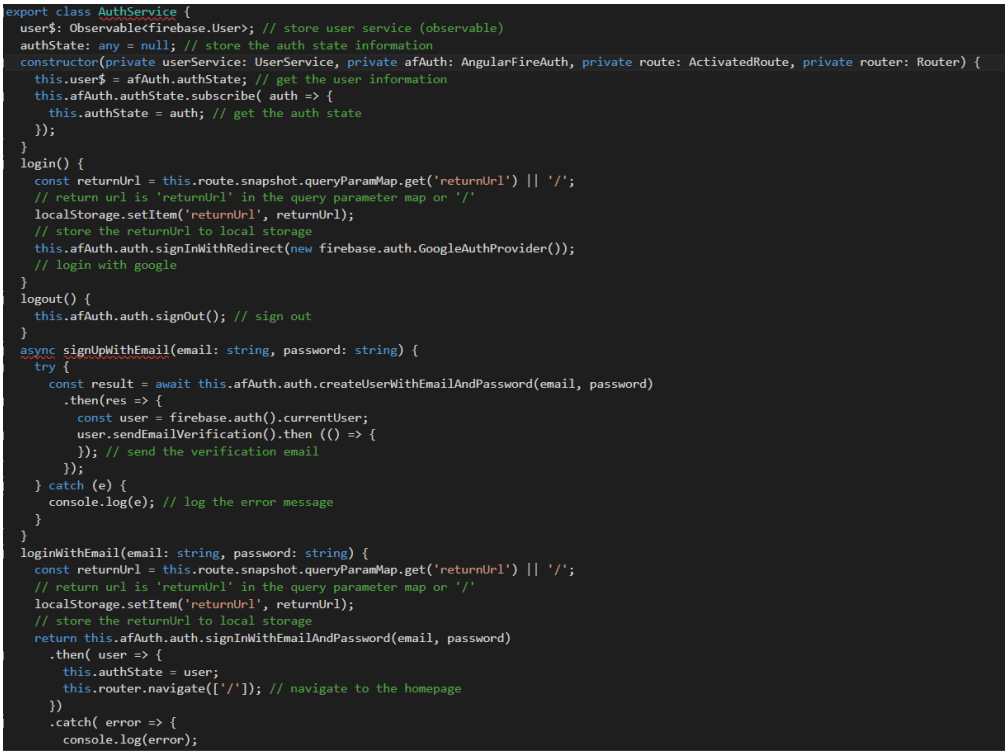
```
export class BsNavbarComponent implements OnInit {
  appUser: AppUser; // store the user information
  cart$: Observable<UserShoppingCart>; // store the user shopping cart information
  tableId: string; // store the table id information
  modal; // used for login form modal

  isNewUser = true; // used for justify if the user is new or not
  email = ''; // login/signup email
  password = ''; // store password
  errorMessage = ''; // error message
  error: {name: string, message: string} = {name: '', message: ''}; // error message
  resetPassword = false; // whether reset the password or not
  public isCollapsed = false; // used for collapse nav-bar when the screen size changed
  constructor(private auth: AuthService,
              private userCartService: UserShoppingCartsService,
              private modalService: NgbModal,
              private route: ActivatedRoute,
              private router: Router) {
    this.route.queryParamMap.subscribe( para => {
      this.tableId = para.get('tableKey'); // get the table key from query parameter map
    });
  }
  login() {
    this.auth.login(); // use auth service login function
  }
  signUpWithEmail() {
    this.clearErrorMessage(); // clear the error message
    if (this.validateForm(this.email, this.password)) {
      this.auth.signUpWithEmail(this.email, this.password)
        .then(() => {
          this.modal.close(); // close the popup form
          this.router.navigate(['/verify-email']); // redirect to the verify email page
        })
        .catch(_error => {
          this.error = _error; // catch the error message
          this.router.navigate(['/']); // navigate to the homepage
        });
    }
  }
```

Figure 3.5: bs-navbar component part 1

```
clearErrorMessage() {
  this.errorMessage = ''; // clear the error message
  this.error = {name: '', message: ''}; // clear the error message
}
changeForm() {
  this.isNewUser = !this.isNewUser; // change the form by changing the isNewUser
}
loginWithEmail() {
  this.clearErrorMessage(); // clear the error message
  if (this.validateForm(this.email, this.password)) {
    this.auth.loginWithEmail(this.email, this.password)
      .then(() => {
        this.modal.close(); // close the modal
        this.router.navigate(['/']); // redirect to the homepage
      })
      .catch(_error => {
        this.error = _error;
        this.router.navigate(['/']); // redirect to the homepage
      });
  }
}
logout() {
  this.auth.logout(); // AuthService logout function
  this.router.navigate(['/']); // redirect to the homepage
}
async ngOnInit() { // initialization
  this.auth.appUser$.subscribe(appUser => this.appUser = appUser); // assign value to appUser
  this.cart$ = await this.userCartService.getUserCart(); // get observable user shopping cart
}
validateForm(email: string, password: string) {
  if (email.length === 0) { // if the email is empty
    this.errorMessage = 'Please enter Email!';
    return false;
  }
  if (password.length === 0) { // if the password is empty
    this.errorMessage = 'Please enter Password!';
    return false;
  }
  if (password.length < 6) { // if the length of password is less than 6
    this.errorMessage = 'Password should be at least 6 characters!';
    return false;
  }
  this.errorMessage = ''; // clear the error message
  return true;
```

Figure 3.6: bs-navbar component part 2



```
export class AuthService {
  user$: Observable<firebase.User>; // store user service (observable)
  authState: any = null; // store the auth state information
  constructor(private userService: UserService, private afAuth: AngularFireAuth, private route: ActivatedRoute, private router: Router) {
    this.user$ = afAuth.authState; // get the user information
    this.afAuth.authState.subscribe( auth => {
      this.authState = auth; // get the auth state
    });
  }
  login() {
    const returnUrl = this.route.snapshot.queryParamMap.get('returnUrl') || '/';
    // return url is 'returnUrl' in the query parameter map or '/'
    localStorage.setItem('returnUrl', returnUrl);
    // store the returnUrl to local storage
    this.afAuth.auth.signInWithRedirect(new firebase.auth.GoogleAuthProvider());
    // login with google
  }
  logout() {
    this.afAuth.auth.signOut(); // sign out
  }
  async signUpWithEmail(email: string, password: string) {
    try {
      const result = await this.afAuth.auth.createUserWithEmailAndPassword(email, password)
        .then(res => {
          const user = firebase.auth().currentUser;
          user.sendEmailVerification().then (() => {
          }); // send the verification email
        });
    } catch (e) {
      console.log(e); // log the error message
    }
  }
  loginWithEmail(email: string, password: string) {
    const returnUrl = this.route.snapshot.queryParamMap.get('returnUrl') || '/';
    // return url is 'returnUrl' in the query parameter map or '/'
    localStorage.setItem('returnUrl', returnUrl);
    // store the returnUrl to local storage
    return this.afAuth.auth.signInWithEmailAndPassword(email, password)
      .then( user => {
        this.authState = user;
        this.router.navigate(['/']); // navigate to the homepage
      })
      .catch( error => {
        console.log(error);
```

Figure 3.7: auth service part 1

```
  loginWithEmail(email: string, password: string) {
    const returnUrl = this.route.snapshot.queryParamMap.get('returnUrl') || '/';
    // return url is 'returnUrl' in the query parameter map or '/'
    localStorage.setItem('returnUrl', returnUrl);
    // store the returnUrl to local storage
    return this.afAuth.auth.signInWithEmailAndPassword(email, password)
      .then( user => {
        this.authState = user;
        this.router.navigate(['/']); // navigate to the homepage
      })
      .catch( error => {
        console.log(error);
        throw error;
      });
  }
  resetPassword(email: string) { // reset the password
    return this.afAuth.auth.sendPasswordResetEmail(email) // send the reset password
      .then( user => {
        this.authState = user; // get the auth state
      })
      .catch( error => {
        console.log(error); // log the error message
        throw error;
      });
  }
  get currentUserName(): string {
    return this.authState['email'];
  }
  get appUser$(): Observable<AppUser> {
    return this.user$
      .switchMap( user => {
        if (user) { // if the user exists
          return this.userService.get(user.uid); // get the app user
        } else {
          return Observable.of(null); // return the null user
        }
      });
  }

}
```

Figure 3.8: auth service part 2

3.2 Restaurant Upload and Edit

If you are a restaurants owner, and you want to manage your restaurant through our platform, and then you need to register an account, and upload your restaurants basic information into our website. You need to fill out your restaurants name, address, description, owner name, and picture of your restaurant. Please see figure 3.9 and 3.10 After you fill out all the information in our system, you can also edit, delete, and search the information you just uploaded. In addition, our system also provides functions for merchants to upload dishes into their restaurants. After finishing dishes uploading, merchants can edit, delete, and search dishes as well. See figure 317 and 318

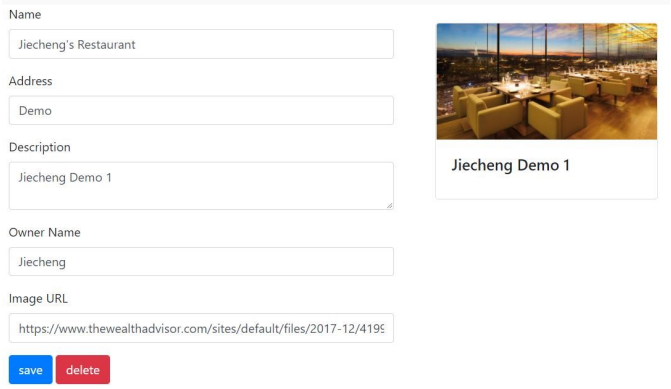


Figure 3.9: Uploading restaurants

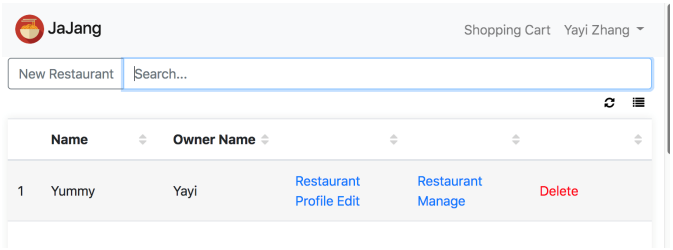


Figure 3.10: Edit, delete, and search restaurants

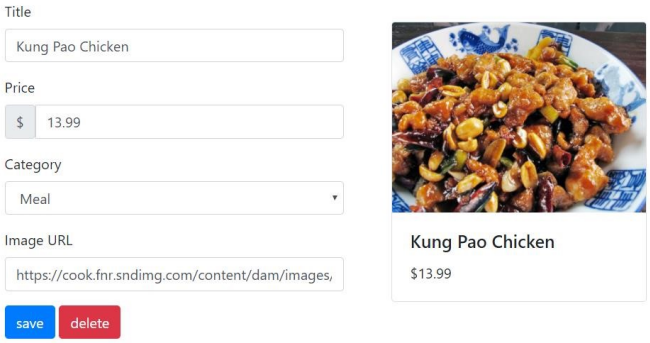


Figure 3.11: Uploading dishes

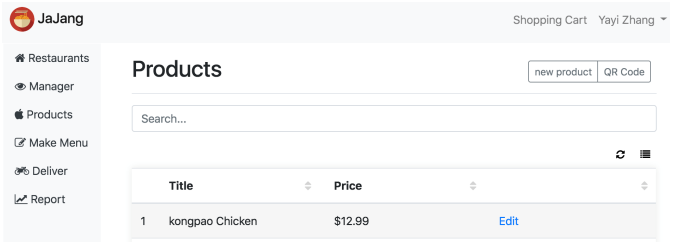


Figure 3.12: Edit delete search dishes

For restaurant, there are mainly two components: ManageRestaurantsFormComponent and ManageRestaurantComponent.

```
{path: 'manage-restaurants-form/new', component: ManageRestaurantsFormComponent, canActivate:
{path: 'manage-restaurants-form/:restId', component: ManageRestaurantsFormComponent, canActiv
{path: 'manage-restaurants', component: ManageRestaurantComponent, canActivate: [AuthGuard]},
{path: 'restaurant-dashboard/:restId', component: RestaurantDashboardComponent, canActivate:
```

Figure 3.13: Root path

In the first component, we define the variables and some fundamental operations like create a new restaurant, save a restaurant, delete a restaurant and so on.

```
export class ManageRestaurantsFormComponent implements OnInit, OnDestroy {
  restaurant: Restaurant = new Restaurant(); // store the restaurant information
  restaurantId: string; // store the restaurant id
  uid: string; // store the user id
  private subscription: Subscription; // used for unsubscribe
  constructor(
    private router: Router, // used for navigation
    private route: ActivatedRoute, // get the information in url
    private auth: AuthService, // auth service
    private restaurantService: RestaurantService) {
  }
  save(restaurant) { // save the restaurant
    if (this.restaurantId) { // if the restaurant id exists
      this.restaurantService.update(this.restaurantId, restaurant); // update restaurant inform
    } else {
      restaurant.ownerId = this.uid; // save the user id to restaurant owner id
      this.restaurantService.create(this.uid, restaurant); // create new restaurant
    }
    this.router.navigateByUrl('/manage-restaurants'); // navigate to the restaurant management
  }
  delete() { // delete the product
    if (confirm('Are you sure you want to delete this restaurant?')) { // need user to confirm
      this.restaurantService.delete(this.restaurantId); // delete the restaurant
      this.router.navigateByUrl('/manage-restaurants'); // navigate to the restaurant managemen
    }
  }
  ngOnInit() {
    this.subscription = this.auth.user$.subscribe( user => {
      this.uid = user.uid; // get the user id
    });
    this.restaurantId = this.route.snapshot.paramMap.get('restId'); // get the restaurant id
    if (this.restaurantId) { // if the restaurant id exists
      this.restaurantService.get(this.restaurantId).take(1).subscribe(restaurant => this.restau
      // get the restaurant
    } else {
      this.restaurant = new Restaurant(); // or create a new restaurant
    }
  }
  ngOnDestroy() {
    this.subscription.unsubscribe(); // unsubscribe
  }
```

Figure 3.14: Manage restaurant form component

In the second component, we define several variables and functions to manage the existed restaurant.

```
export class ManageRestaurantComponent implements OnDestroy, OnInit {
  restaurants: Restaurant[] = []; // store the restaurant information
  tableResource: DataTableResource<Restaurant>; // for data table
  items: Restaurant[] = []; // for data table
  itemCount: number; // the number of items
  uid: string; // store the user id
  private subsription: Subscription; // used for unsubscribe
  constructor(private restaurantService: RestaurantService,
              private auth: AuthService) {
    this.auth.user$.subscribe( user => { // get the user
      this.subsription = restaurantService.getAllForOne(user.uid)
        .subscribe(restaurant => {
          this.restaurants = restaurant; // get the restaurant
          this.initializeTable(restaurant); // initialize the data table
        });
    });

  }
  filter(query: string) { // filter the data table
    const filterRestaurant = (query) ?
      this.restaurants.filter(restaurant =>
        restaurant.name.toLowerCase().includes(query.toLowerCase())) // case unsensative
      : this.restaurants;
    this.initializeTable(filterRestaurant); // initialize the table after filtering the restaurant
  }
  private initializeTable(restaurants: Restaurant[]) { // initialize the data table
    this.tableResource = new DataTableResource<Restaurant>(restaurants); // create a new data table object
    this.tableResource.query({ offset: 0 })
      .then(items => this.items = items); // get the items
    this.tableResource.count()
      .then(count => this.itemCount = count); // get the number of items
  }
  reloadItems(params) { // reload the data table
    if (this.tableResource) { // if the data table exists
      this.tableResource.query(params)
        .then(items => this.items = items); // get the items
    }
  }
  delete(restaurantId) {
    if (confirm('Are you sure you want to delete this product?')) { // ask for confirmation
      this.restaurantService.delete(restaurantId); // delete the restaurant
    }
  }
  ngOnDestroy() {
```

Figure 3.15: Manage restaurant component

All the implementation which is used to CRUD the restaurant data in database is in RestaurantService. Also, the code to CRUD products data in database is in ProductService

```
export class RestaurantService {
  constructor(private db: AngularFireDatabase) { }
  create(uid, restaurant) { // create a new restaurant under a certain user
    this.db.list('restaurants').push(restaurant);
    // push the new restaurant to the database
  }
  getAll() { // get all restaurants
    return this.db.list<Restaurant>('/restaurants').snapshotChanges()
      .map(restaurants => { // get all restaurants
        return restaurants.map(restaurant => { // map to a certain restaurant
          const rs = restaurant.payload.val(); // values of restaurant
          rs.key = restaurant.key; // the key of restaurant
          return rs; // return the restaurant object
        } );
      });
  }
  getAllForOne(uid) { // get all restaurants for a certain user
    return this.db.list<Restaurant>('/restaurants', ref => ref.orderByChild('ownerId').equalTo(uid))
      .snapshotChanges().map(restaurants => { // get all restaurants
        return restaurants.map(restaurant => {
          const rs = restaurant.payload.val(); // the values of restaurant
          rs.key = restaurant.key; // the key of restaurant
          return rs; // return the restaurant object
        } );
      });
  }
  get(restaurantId) { // get a certain restaurant
    return this.db.object<Restaurant>('restaurants/' + restaurantId)
      .snapshotChanges().map(restaurant => { // get restaurant
          const rs = restaurant.payload.val(); // the values of the restaurant
          rs.key = restaurant.key; // the key of the restaurant
          return rs; // return the restaurant object
      } );
  }
  update(restaurantId, restaurant) { // update the information of a certain restaurant
    // this.db.object('users/' + uid + '/restaurants/' + restaurantId).update(restaurant);
    this.db.object('restaurants/' + restaurantId).update(restaurant);
    // update in the database
  }
  delete(restaurantId) { // delete a certain restaurant according to the restaurant id
    // this.db.object('users/' + uid + '/restaurants/' + restaurantId).remove();
    this.db.object('restaurants/' + restaurantId).remove();
    // remove the restaurant from the database
  }
```

Figure 3.16: Restaurant service part 1

```
  update(restaurantId, restaurant) { // update the information of a certain restaurant
    // this.db.object('users/' + uid + '/restaurants/' + restaurantId).update(restaurant);
    this.db.object('restaurants/' + restaurantId).update(restaurant);
    // update in the database
  }
  delete(restaurantId) { // delete a certain restaurant according to the restaurant id
    // this.db.object('users/' + uid + '/restaurants/' + restaurantId).remove();
    this.db.object('restaurants/' + restaurantId).remove();
    // remove the restaurant from the database
  }
  growSize(restaurantId: string) {
    // grow the table card size
    this.db.object<number>('/restaurants/' + restaurantId + '/size')
      .valueChanges().take(1).subscribe( s => {
      if (s !== null) {
        this.db.object('/restaurants/' + restaurantId )
          .update({
            size: (((s + 50) < 601) ? (s + 50) : s)
          });
      } else {
        this.db.object('/restaurants/' + restaurantId )
          .update({
            size: 300
          });
      }

    });
  }
  shrinkSize(restaurantId: string) {
    this.db.object<number>('/restaurants/' + restaurantId + '/size')
      .valueChanges().take(1).subscribe( s => {
      console.log(s);
      if (s !== null) {
        this.db.object('/restaurants/' + restaurantId )
          .update({
            size: (((s - 50) > 99) ? (s - 50) : s)
          });
      } else {
        this.db.object('/restaurants/' + restaurantId )
          .update({
            size: 300
          });
      }
```

Figure 3.17: Restaurant service part 2

```
export class RestaurantProductService {
  constructor(private db: AngularFireDatabase) { }
  create(product, restaurantId) {
    return this.db.list('/restaurants/' + restaurantId + '/products/').push(product);
  }
  getAll(restaurantId) {
    return this.db.list<Product>('/restaurants/' + restaurantId + '/products/')
      .snapshotChanges().map( products => {
        return products.map(product => {
          const rs: Product = product.payload.val();
          rs.key = product.key;
          return rs;
        });
      });
  }
  get(productId, restaurantId) {
    return this.db.object('/restaurants/' + restaurantId + '/products/' + productId)
      .snapshotChanges().map( product => {
        const rs = product.payload.val();
        rs.key = product.key;
        return rs;
        }
      );
  }
  update(productId, restaurantId, product) {
    this.db.object('/restaurants/' + restaurantId + '/products/' + productId).update(product);
  }
  delete(productId, restaurantId) {
    this.db.object('/restaurants/' + restaurantId + '/products/' + productId).remove();
  }
}
```

Figure 3.18: Restaurant product service

3.3 Management

After merchants uploading restaurants and dishes information, next step of managing restaurants is to create QR code for each table of the restaurants. After customers scanned the QR code of the table they reserved, this code can exactly target which restaurant they entered and which table they seated. So restaurant can provide accurate service for each customer. See Figure 3.19. The generated QR codes are able to be printed. See Figure 3.20.

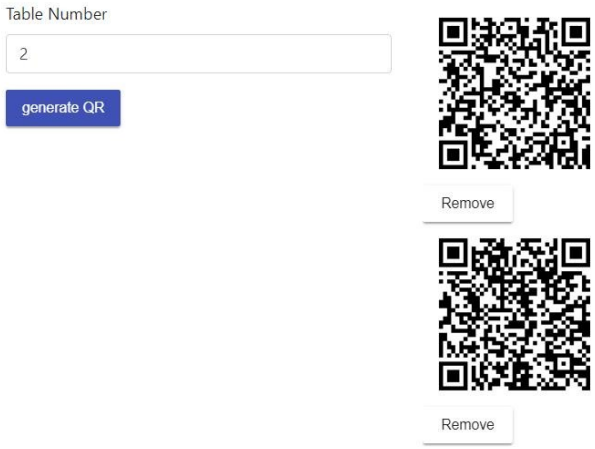


Figure 3.19: Create QR Code

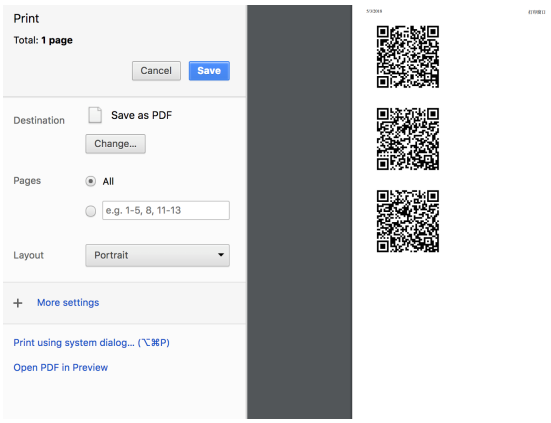


Figure 3.20: Print QR Code

After merchants are done with QR code, they can go back to the home page of management and monitor the status of each table. See figure 3.21 On the top right corner, merchants can grow or shrink this management page. And on the left side, there is menu bar for merchants to manage their restaurant.

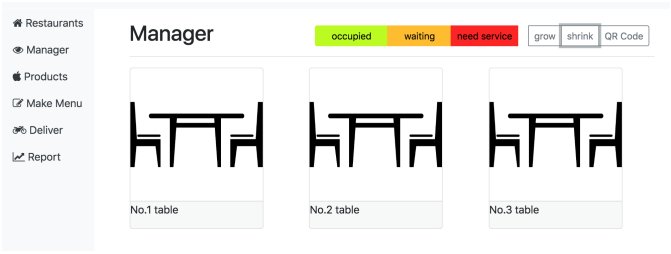


Figure 3.21: Manager page

If the table turns green, which means there is a customer scan the QR code on the table. See figure 3.22.If the table turns yellow, which means customers placed order, and also let the chefs know they could prepare customers food, and cook it. See figure 3.23 If table turns red, which means customers need service. For example, they need more water, or they want to add more dishes. See figure 3.24.

Figure 3.22: Green signal

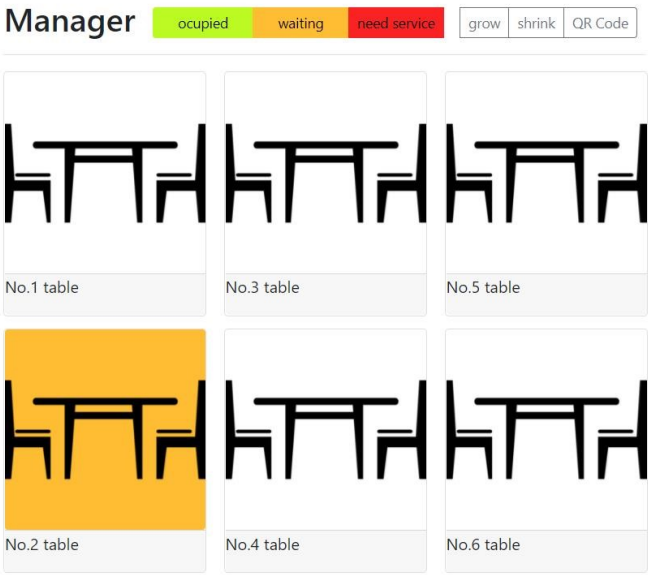


Figure 3.23: Yellow signal

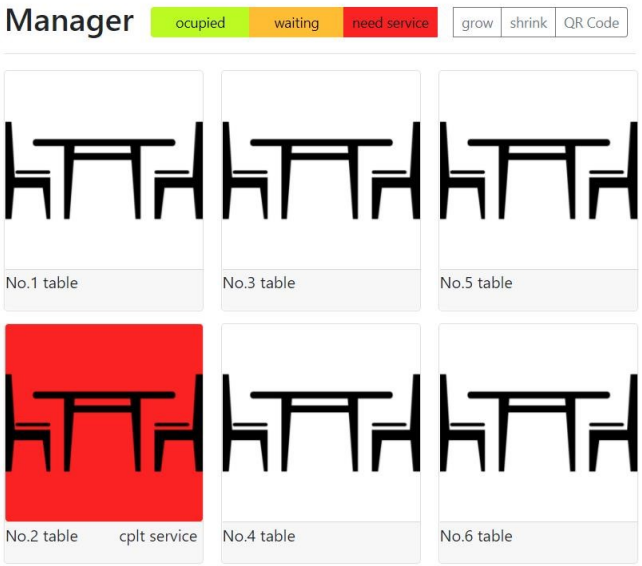


Figure 3.24: Red signal

When customers come into the restaurant, they only need to use their personal smart phone to scan the QR code on the table they want to use. This QR code will lead customers to food-order page. see figure 3.37 And merchants will see the

table you seated turn green on the website where we designed for merchants as we explained it before.

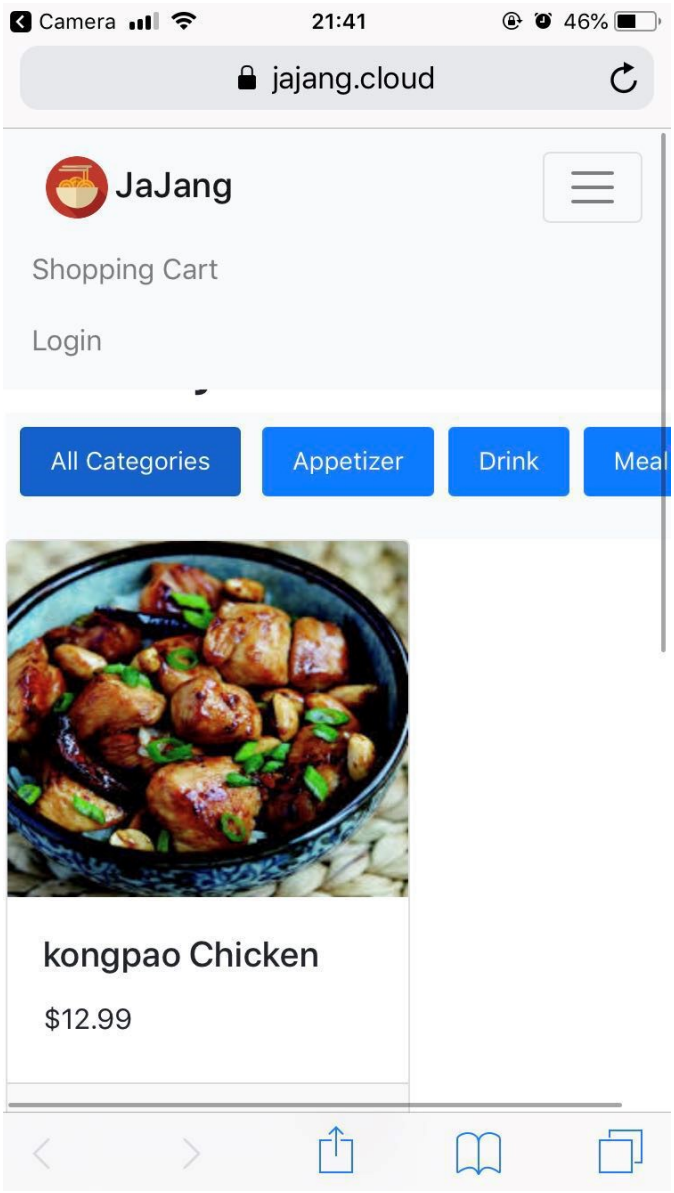


Figure 3.25: Mobile place order page

After you placed order, your screen will change to figure 3.38. And merchants will see the table you seated turn yellow on the website where we designed for merchants.

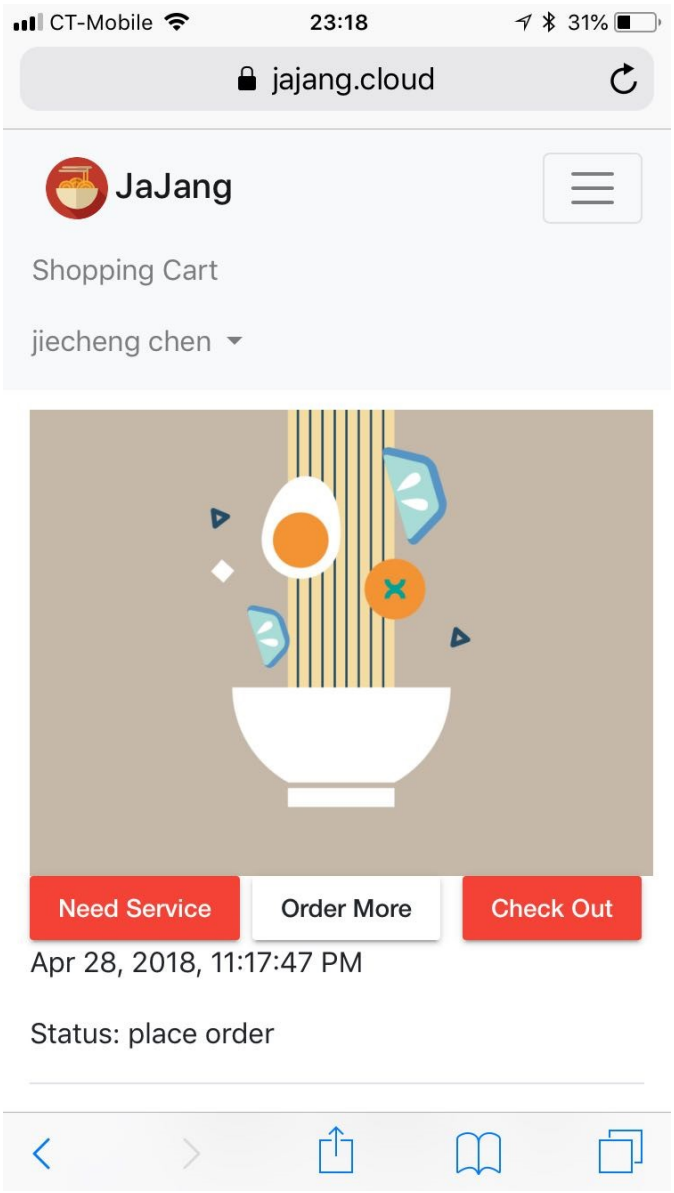


Figure 3.26: Mobile service page

If customers need any service during their meals, they can click on need service button, and their screen will change to figure 3.27. Then our system will let merchants know you need service, so they can come to you and fix your issues.

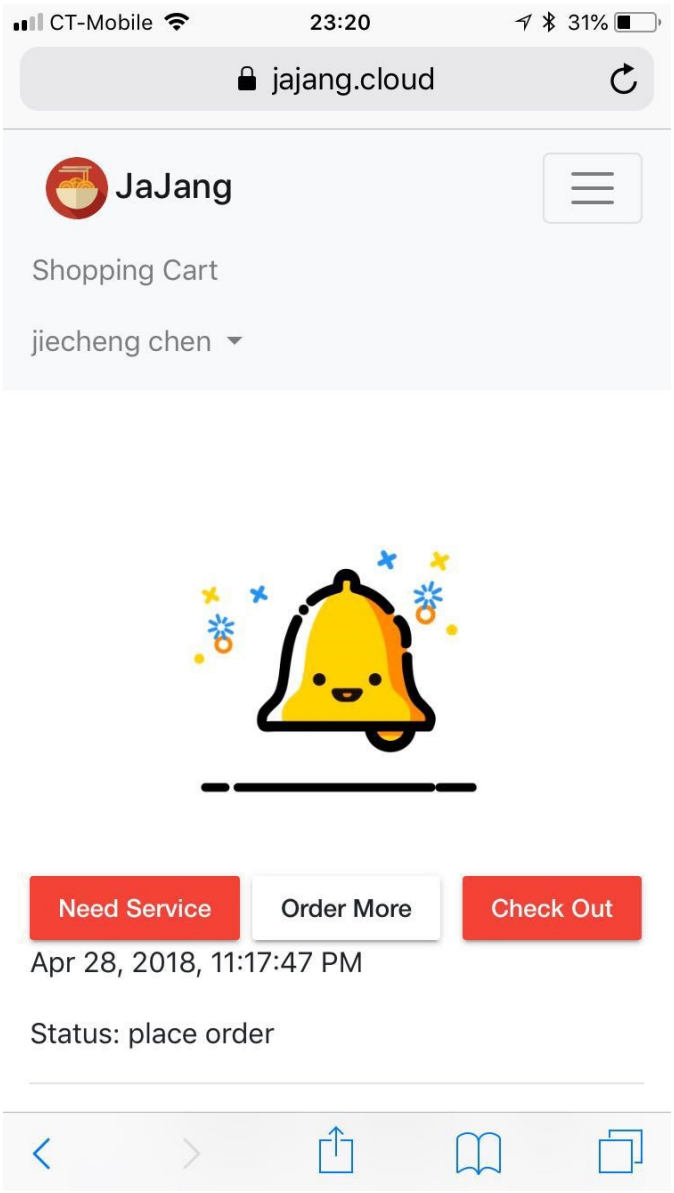


Figure 3.27: Need service signal

If you want to order more food, our system provides this function for customers to add more food. After you click on order more button, it will lead you to food-order page. see figure 3.28, you can see there are two orders of kongpao chicken, and the second one is customer added later.

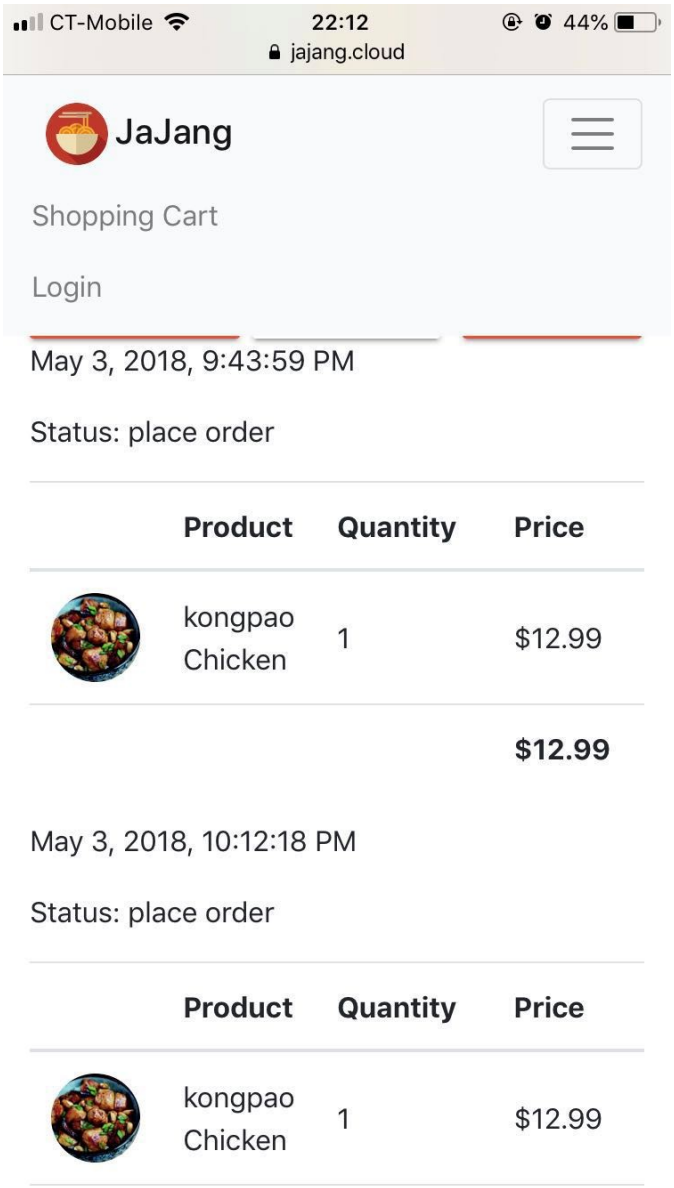


Figure 3.28: Order detail

If customers want to check out, they just need to click on check out button. See figure 3.29 And your phone screen will change to figure 3.29 Merchants will see the table you seated turn blink on the website where we designed for merchants. Our system will notice waiter to help customers check.

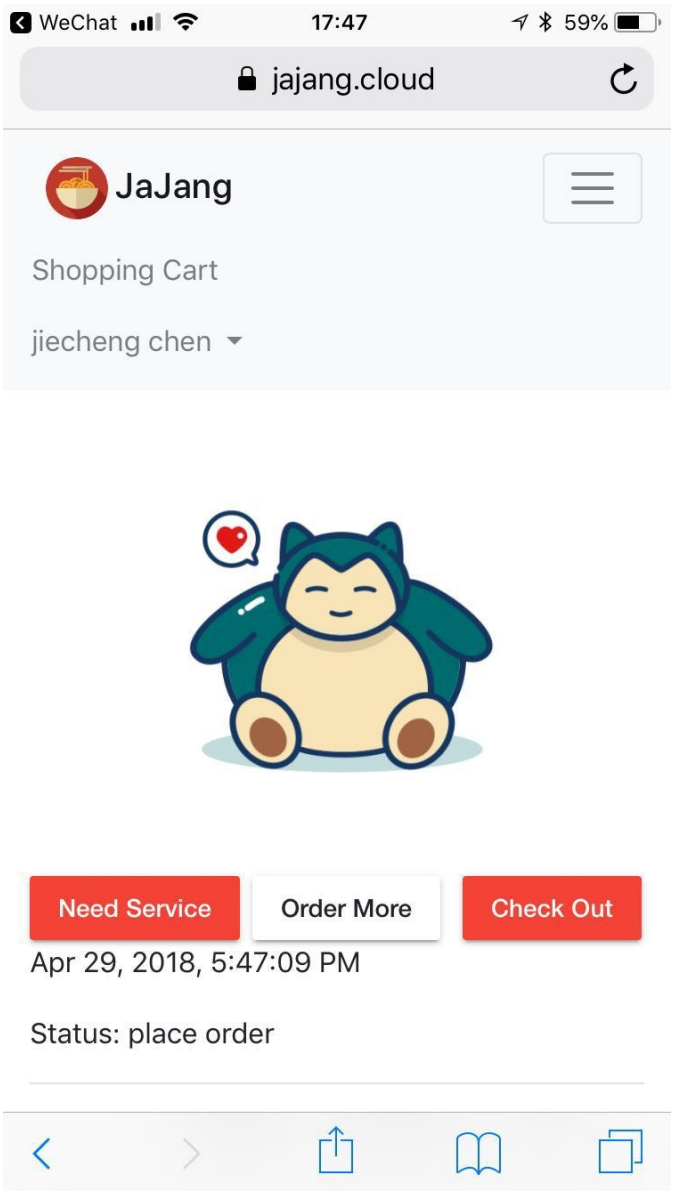


Figure 3.29: Need check out signal

To generate the QR code, we mainly use components: QRcomponent.

```
export class QrCodeComponent implements OnInit {
  numberT = 0; // store the number of table
  urlAddress = 'www.jajang.cloud/restaurants/'; // qr code information: urlAddress + tableKey
  restuarantId: string; // store the restaurant id
  table$; // store table information (observable)
  printStyle = '' + // print style css
    'button' +
    '{' +
    'visibility: hidden;' +
    '}';
  constructor(private hereOrderService: HereOrderService,
              private route: ActivatedRoute) {
    this.restuarantId = this.route.parent.snapshot.url[1].toString(); // get the restaurant id
    this.table$ = this.hereOrderService.getAllTable(this.restuarantId); // get the table(observable)
  }
  generateQR(input) { // generate qr code for table
    this.hereOrderService.generateTable(input.number, this.restuarantId);
  }
  deleteTable(tableId) { // delete the table
    this.hereOrderService.deleteOneTable(tableId, this.restuarantId);
  }

  ngOnInit() {
  }
```

Figure 3.30: QRCode component

To manage the table status, the Manager Component is used:

```
export class ManagerComponent implements OnInit, OnDestroy {
  tableSize = 300; // table size, initialize to 300
  tables$; // store the tables information (observable)
  restaurantId; // store the restaurant id
  private subscription: Subscription; // used for unsubscribe
  constructor(private hereOrderService: HereOrderService, // mobile service
              private restaurantService: RestaurantService, // restaurant service
              private route: ActivatedRoute) { // used for getting information from the url
  }
  completeService(tableId) {
    this.hereOrderService.completeService(tableId, this.restaurantId);
    // complete the service
  }
  status(table: Table): string { // change teh table card color
    if (table.needCheckOut) {
      return '../../assets/table-checkout.gif'; // blink red
    } else if (table.needService) {
      return '../../assets/table-warning.png'; // red
    } else if (table.waitFood) {
      return '../../assets/table-waiting.png'; // yellow
    } else if (table.occupied) {
      return '../../assets/table-idle.png'; // green
    } else {
      return '../../assets/table.png'; // no color
    }
  }
  ngOnInit() {
    this.restaurantId = this.route.parent.snapshot.url[1].toString(); // get the restaurant id
    this.tables$ = this.hereOrderService.getAllTable(this.restaurantId); // get the table
    this.subscription = this.restaurantService.get(this.restaurantId).subscribe(res => {
      this.tableSize = res.size; // get the size of table
    });
  }
  ngOnDestroy() {
    this.subscription.unsubscribe(); // unsubscribe
  }
```

Figure 3.31: Manager component

For the customers, the InRestaurantOrder Component is used:

```
export class InRestaurantOrderComponent implements OnInit, OnDestroy {
  tableId: string; // store the table id
  restaurantId: string; // store the restaurant id
  orders$; // store order information (observable)
  orders: Order[]; // store the orders information
  table$; // store the table information (observable)
  table = new Table(); // store the table information
  private subscription1: Subscription; // used for unsubscribe
  private subscription2: Subscription; // avoid memory leak
  constructor(private route: ActivatedRoute,
              private hereService: HereOrderService,
              private router: Router) {
    this.tableId = this.route.snapshot.queryParamMap.get('tableKey'); // get the table id
    this.restaurantId = this.route.snapshot.queryParamMap.get('restaurantId'); // get the restaurant id
    this.table$ = this.hereService.getTable(this.tableId, this.restaurantId); // get the table (observable)
    this.orders$ = this.hereService.getAllOrdersForTable(this.tableId, this.restaurantId);
    this.subscription1 = this.orders$.subscribe( o => {
      this.orders = o; // get the order information
      if (this.orders === null || this.orders === undefined || this.orders.length === 0) {
        this.router.navigate(['/']); // redirect to the home page
      }
    });
    this.subscription2 = this.table$.subscribe( t => {
      this.table = t; // assign the value to table
    });
  }

  needService() { // change the table status to 'need service'
    this.hereService.needService(this.tableId, this.restaurantId);
  }
  checkOut() { // change the table status to 'check out'
    this.hereService.needCheckOut(this.tableId, this.restaurantId);
  }
  changeGifAccordingToStatus() { // change the gif according the table status
    if (this.table.needCheckOut) { // need check out
      return '../../assets/snorlax.gif';
    }
    if (this.table.needService) { // need service
      return '../../assets/bell.gif';
    }
    for (const order in this.orders) {
      if (this.orders[order].status !== 'cooked') {
        return '../../assets/cooking.gif'; // one of order is not cooked
      }
```

Figure 3.32: In restaurant order component

All the codes about CRUD the table data in database is written in HereOrder-Service:

```
export class HereOrderService {
  constructor(private db: AngularFireDatabase) { }
  generateTable(numOfTable: number, restaurantId: string) { // generate some tables in a restaurant
    let n = 0; // the index of table
    let totalTable = 0; // the number of tables
    this.totalTable(restaurantId).subscribe( ts => {
      totalTable = ts.length;
        while (n < numOfTable) {
          this.db.list('/restaurants/' + restaurantId + '/tables/')
            .push({
              date: new Date().getTime(),
              needCheckOut: false,
              needService: false,
              occupied: false,
              waitFood: false,
              restaurantId: restaurantId,
              index: totalTable + 1 + n,
              size: 300
            });
          n++;
        } // update the table
    }
    );
  }
  generateOrder(tableId: string, restaurantId: string) { // generate the order according to the table id and restaurant id
    return this.db.list('/restaurants/' + restaurantId + '/tables/' + tableId + '/orders/')
      .push({
        date: new Date().getTime(),
        status: 'place order' // update the table information
      });
  }
  changeToOccupy(tableId: string, restaurantId: string) {
    // change the states of table to occupied according to the table id and restaurant id
    this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
      .update({
        occupied: true // update the table information
      });
  }
  getAllTable(restaurantId: string) { // get all tables in a certain restaurant
    return this.db.list('/restaurants/' + restaurantId + '/tables/')
      .snapshotChanges().map( tables => {
        return tables.map(table => {
          if (table !== null) { // if the table exists
```

Figure 3.33: Here order service part 1

```
getTable(tableId: string, restaurantId: string) { // get a certain table according to the table id and restaurant id
  return this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .snapshotChanges().map( table => {
        if (table !== null) { // if the table exists
          return new Table(table.payload.val(), table.key); // return the table (including the values and key)
        } else {
          return new Table(); // or return an empty table
        }
    });
}
private finishTable(tableId: string, restaurantId: string) { // finish the services to a certain table
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId + '/orders/').remove();
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .update({
      needCheckOut: false,
      needService: false,
      occupied: false,
      waitFood: false // update the table information
    });
}
deleteOneTable(tableId: string, restaurantId: string) { // delete a certain table according to the table id and restaurant id
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .remove(); // remove the table from the database
}
needCheckOut(tableId: string, restaurantId: string) { // change a certain table's state to the 'need check out'
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .update(
      {
        needCheckOut: true, // update the table state
      }
    );
}
finishCook(tableId: string, restaurantId: string) { // change a certain table's state to finished
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .update(
      {
        waitFood: false // update the table information
      }
    );
}
needService(tableId: string, restaurantId: string) { // change a certain tables's state to need service
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId)
    .update(
      {
```

Figure 3.34: Here order service part 2

```
getAllOrdersForTable(tableId: string, restaurantId: string) { // get all orders for a certain table
  return this.db.list<Order>('/restaurants/' + restaurantId + '/tables/' + tableId + '/orders/')
    .snapshotChanges().map(orders => {
      return orders.map( order => {
        if ( order !== null ) { // if the order exists
          return new Order(order.payload.val(), order.key); // return the order (including the values and key)
        } else {
          return new Order(); // or return an empty order
        }
      });
  });
}
getAllFinishedOrders(restaurantId: string) { // get all finished order for a certain restaurant
  return this.db.list<Order>('/restaurants/' + restaurantId + '/finishedOrders/')
    .snapshotChanges().map(orders => {
      return orders.map( order => {
        if ( order !== null ) { // if the order exists
          return new Order(order.payload.val(), order.key); // return the order values and key
        } else {
          return new Order(); // or return an empty order
        }
      });
    });
}
getFinishedOrders( orderId: string, restaurantId: string) { // get finished order
  return this.db.object('/restaurants/' + restaurantId + '/finishedOrders/' + orderId)
    .snapshotChanges().map(order => {
      if (order !== null) { // if the order exists
        return new Order(order.payload.val(), order.key); // return an object (including the values and key)
      } else {
        return new Order(); // or return the new order
      }
    });
}
changeOrderStatusToCooking(orderId: string, tableId: string, restaurantId: string) {
  // change a certain order's status to 'cooking'
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId + '/orders/' + orderId)
    .update({
      status: 'cooking' // update in database
    });
}
changeOrderStatusToCooked(orderId: string, tableId: string, restaurantId: string) {
  // change a certain order's status to 'cooked'
  this.db.object('/restaurants/' + restaurantId + '/tables/' + tableId + '/orders/' + orderId)
```

Figure 3.35: Here order service part 3

```
finishOrders(tableId: string, restaurantId: string) {
  // finish the orders of a certain table
  this.getAllOrdersForTable(tableId, restaurantId).take(1).subscribe( orders => {
    for (const order in orders) {
      if (true) {
        this.db.object('/restaurants/' + restaurantId + '/finishedOrders/' + orders[order].key)
          .update({
            restaurantId: orders[order].restaurantId,
            date: orders[order].date, // update the information of table in database
          });
        const items = orders[order].items;
        for (const itemIndex in items) {
          if (true) {
            this.db.object('/restaurants/' + restaurantId + '/finishedOrders/' + orders[order].key + '/items/' + items[itemIndex].key)
              .update({
                title: items[itemIndex].title,
                imageUrl: items[itemIndex].imageUrl,
                price: items[itemIndex].price,
                quantity: items[itemIndex].quantity
              }); // add these informatio to finished order in database
          }
        }
      }
    }
    this.finishTable(tableId, restaurantId);
  }
  );
}
private totalTable(restaurantId: string) {
  return this.db.list<string>('/restaurants/' + restaurantId + '/tables/')
    .snapshotChanges();
}

}
```

Figure 3.36: Here order service part 4

3.4 Create Menu

The merchants can create their own menu.

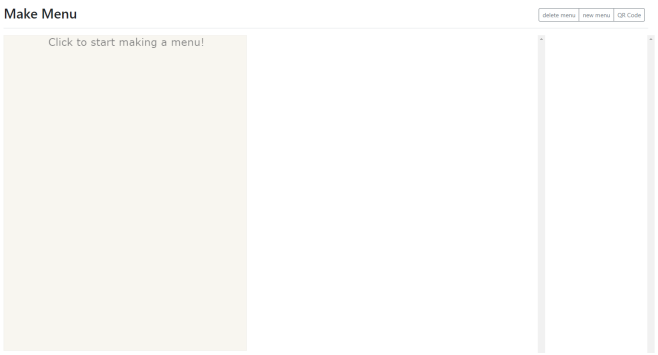


Figure 3.37: Order detail

This is an empty menu and you can just drag the dishes from the list on the right to fill in the menu. The created menus are shown in the right-most list. At the top right-most corner, there are three buttons to delete an existed menu, create a new menu and generate the QR code.

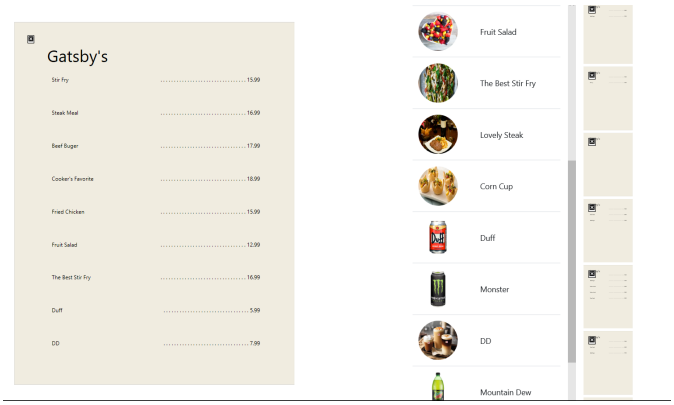


Figure 3.38: Order detail

We used SVG to implement this feature. The menu maker itself is a HTML ¡svg¿ element. And three components are created to provide the demanded menu maker functionality. Code snippets of these components are shown below:

```
@Component({
  selector: 'app-svg-maker-thumbnail',
  templateUrl: './svg-maker-thumbnail.component.html',
  styleUrls: ['./svg-maker-thumbnail.component.css']
})
export class SvgMakerThumbnailComponent implements OnInit, OnDestroy {
  @Input() menuId;
  @Input() size;
  restId: string;
  menu$: Observable<Menu>;
  menuItems: MenuItem[] = [];
  private subscription: Subscription;

  unitLength = 1;
  locations: number[] = [];

  constructor(private route: ActivatedRoute,
              private router: Router,
              private paperMenuService: PaperMenuService) {
  }
  ngOnInit() {
    this.restId = this.route.parent.snapshot.url[1].toString();
    if (!this.menuId) {
      this.menuId = this.route.snapshot.paramMap.get('menuId');
    }
    this.menu$ = this.paperMenuService.getMenu(this.menuId, this.restId);
    this.subscription = this.menu$.subscribe( options: m => {
      this.menuItems = m.menuItems;
      this.locations = Array(this.menuItems.length).fill( value: 1).map( callbackfn: (x, i) => i * this.unitLength);
    });
  }
  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
  createMenu() {
    this.paperMenuService.createMenu(this.restId)
      .then( onfulfilled: u => {
        this.router.navigate(['../menu-maker', u.key], {relativeTo: this.route, queryParams: {section: 'Make Me
      });
  }
  delete(itemId) {
    this.paperMenuService.delete(itemId, this.menuId, this.restId);
  }
```

Figure 3.39: SVG maker thumbnail component

```
@Component({
  selector: 'app-menu-maker',
  templateUrl: './menu-maker.component.html',
  styleUrls: ['./menu-maker.component.css']
})
export class MenuMakerComponent implements OnInit, OnDestroy {
  menuId: string; // store the menu id
  restId; string; // store the restaurant id
  menus: Menu[]; // store the paper menu id
  private subscription: Subscription; // used for unsubscribe
  constructor(private route: ActivatedRoute, // used for getting information from url
              private paperMenuService: PaperMenuService) {
    this.restId = this.route.parent.snapshot.url[1].toString(); // get the restaurant id
    this.menuId = this.route.snapshot.paramMap.get('menuId'); // get the menu id
    this.subscription = this.paperMenuService.getAllMenus(this.restId)
      .subscribe( options: m => {
        this.menus = m; // get the menu
      });
  }
  ngOnInit() {
  }
  ngOnDestroy() {
    this.subscription.unsubscribe(); // unsubscribe
  }

}
```

Figure 3.40: Menu maker component

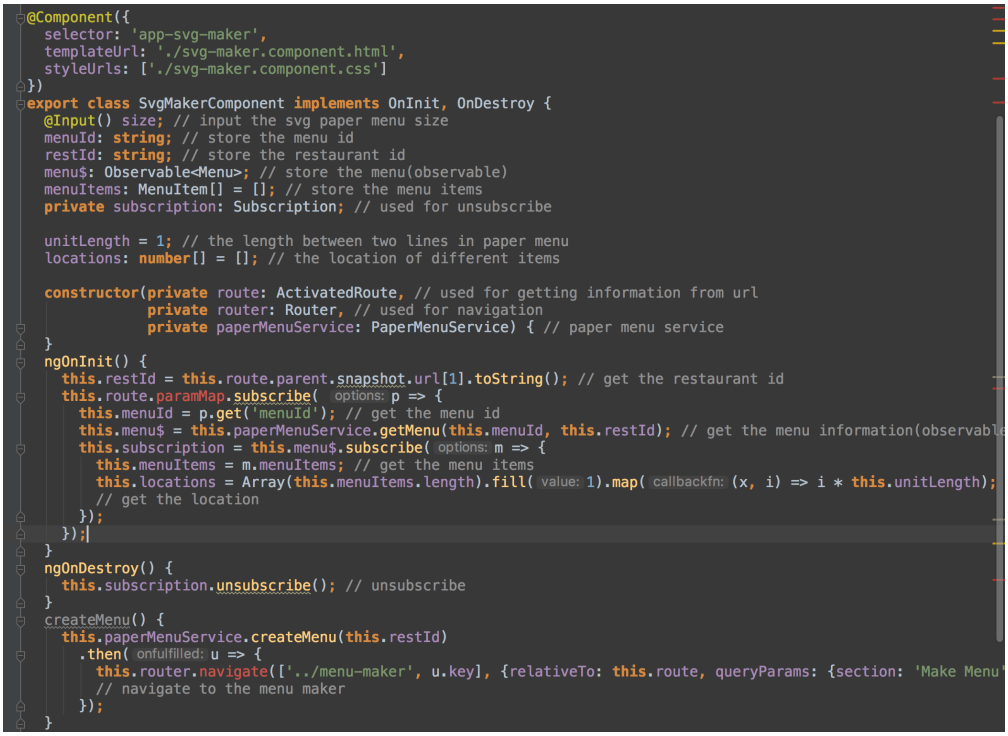


Figure 3.41: SVG maker component

All three components use a PaperMenuService to interact with database. PaperMenuService provides functionalities of creating/deleting menus and items. The code snippet is shown:

```
@Injectable()
export class PaperMenuService {
  constructor(private db: AngularFireDatabase) { }
  createMenu(restaurantId: string) { // create a menu according to the restaurant id
    return this.db.list('/restaurants/' + restaurantId + '/menus/')
      .push({
        date: new Date().getTime() // push the new date to the database
      });
  }
  deleteMenu(menuId: string, restaurantId: string) { // delete a certain menu according to the menu id and restau
    this.db.object('/restaurants/' + restaurantId + '/menus/' + menuId)
      .remove(); // remove the menu from the database
  }
  addFoodItem(menuId: string, restaurantId: string, product: Product) {
    // add food item to a certain menu according the menu id and restaurant id
    this.db.list('/restaurants/' + restaurantId + '/menus/' + menuId + '/menuItems/')
      .push({
        type: 'food', // update the item type
        name: product.title, // update the product title
        price: product.price, // update the product price
      });
  }
  delete(menuItemId: string, menuId: string, restaurantId: string) {
    // delete a certain menu item from the menu
    this.db.object('/restaurants/' + restaurantId + '/menus/' + menuId + '/menuItems/' + menuItemId).remove();
  }
  getMenu(menuId: string, restaurantId: string) { // get a certain menu according to the menu id and restaurant i
    return this.db.object<Menu>('/restaurants/' + restaurantId + '/menus/' + menuId)
      .snapshotChanges().map( menu => {
        if (menu !== null) { // if the menu exists
          return new Menu(menu.payload.val(), menu.key); // return a new menu (including its values and key)
        } else { // if the menu does not exist
          return new Menu(); // return an empty menu
        }
      });

  }
  getAllMenus(restaurantId: string) { // get all menus for the restaurant
    return this.db.list('/restaurants/' + restaurantId + '/menus/')
      .snapshotChanges().map(menus => {
        return menus.map( menu => {
          if (menu !== null) { // if the menu exist
            return new Menu(menu.payload.val(), menu.key); // return the menu object
          } else { // if the menu does not exist
```

Figure 3.42: Paper menu service

Our system can also record the daily turnover and show it with a bar chart. As can be seen from the graph below, the bar chart shows the experiment result we got in April 27, 28, 29 three days. The user can also transfer the chart to table.

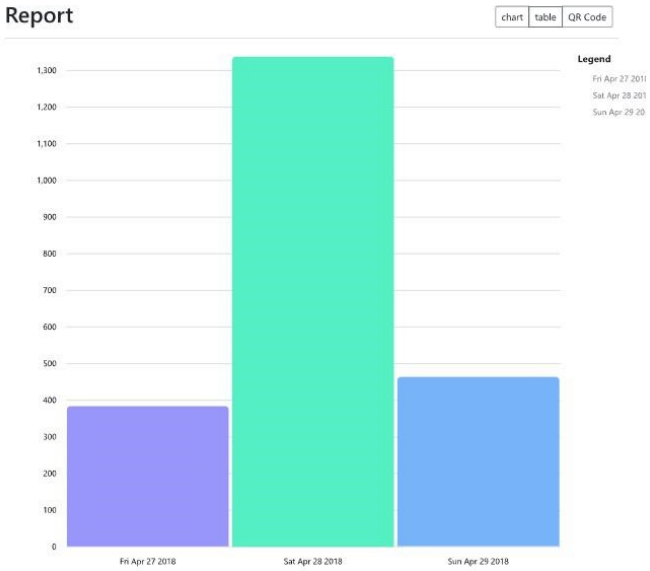


Figure 3.43: Chart

An OrderChartComponent was created to implement this feature.

```
export class OrderChartComponent implements OnInit, OnDestroy {
  data = []; // store the chart data

  orders: Order[] = []; // store the order
  tableResource: DataTableResource<Order>; // used for the data table
  items: Order[] = []; // used for the data table
  itemCount: number; // the number of items
  restuarantId: string; // store the restaurant id
  private subsription: Subscription; // used for unsubscribe
  constructor(private hereService: HereOrderService, // mobile service
              private route: ActivatedRoute) { // used for getting information from the url
    this.restuarantId = this.route.parent.snapshot.url[1].toString(); // get the restaurant id
    this.subsription = this.hereService.getAllFinishedOrders(this.restuarantId)
      .subscribe(  options: orders => {
        this.orders = orders; // get the order information
        this.initializeTable(this.orders); // initialize the data table
        this.chart();
      });
  }
```

Figure 3.44: order chart component

This component generally retrieves all history orders data for a specific restaurant and push this data to an array. Later, the array is passed to ¡neu-charts-bar-vertical¿ HTML tag, and the corresponding chart will be shown in clients browser. Below the code snippet of retrieving order data is shown:

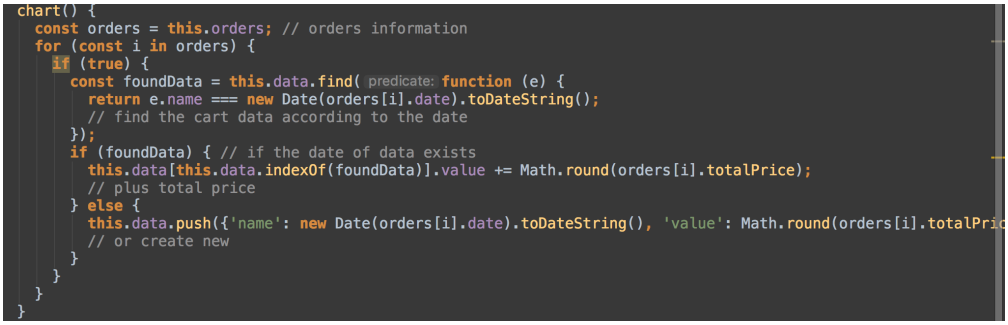


Figure 3.45: order chart component

3.5 Online Order

Our website is not only for customers dine in at restaurants, but also can order food online if they do not want to go out for food. As a customer, once you get in to our website, you can decide to register an account or not since our system does not force customers to register. After you click on the picture on home page, you will be directly taken to restaurant page. See figure 3.46 You can pick any restaurants you like and go head to order your food on your favorite restaurants. See 3.47.

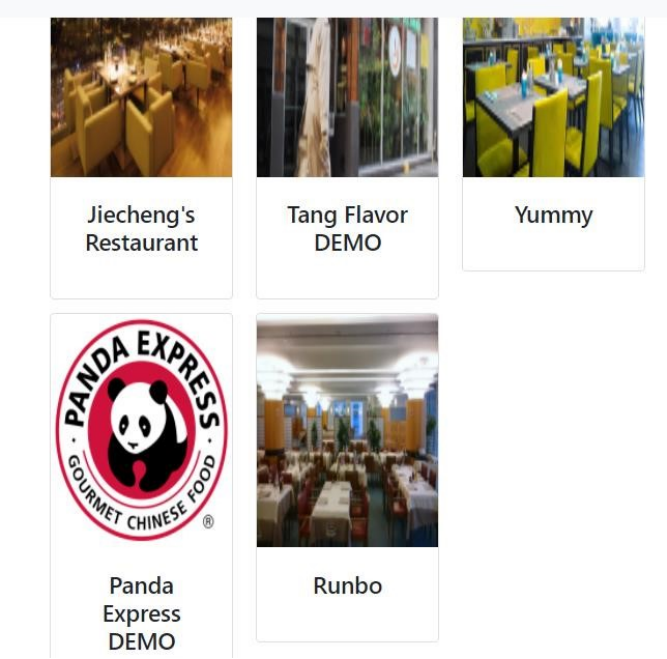


Figure 3.46: online order

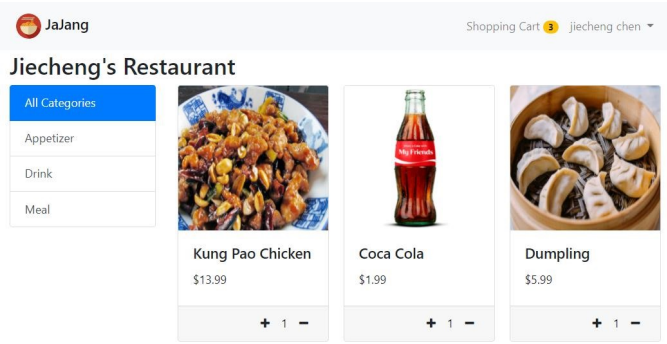


Figure 3.47: online order

After your decision is made, you can go to shopping cart to check and place order. See figure 3.48 and figure 3.49.

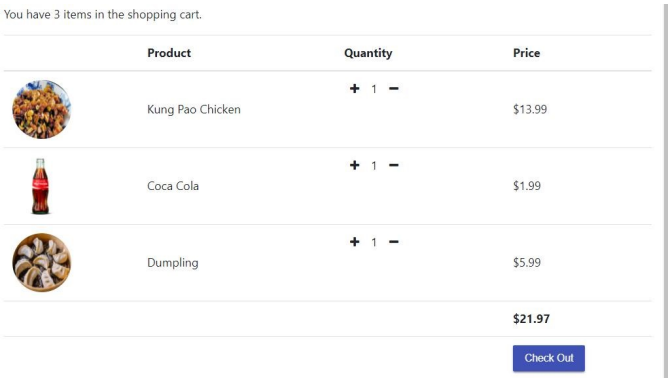


Figure 3.48: online order

Figure 3.49: online order

4 Futures

In the future, our group wants to add rating function for customers to rate for the restaurants and leave the comments. We also plan to integrate Google map in our application so that we can recommend the nearest restaurant according to the customers position and navigate them to the destination. We are in a Big Data era so big data is another important part we want to add. We can analyze customers preference according to their ordering history and recommend suitable restaurants for them. Whats more, we will also keep updating UI and improve the security of our website.