



VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Ho Nguyen Vo

ONLINE FOOD SHOP FOR IPHONE

Department of Technology and Communication

2012

VAASAN AMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Software Engineering

ABSTRACT

Author	Ho Nguyen Vo
Title	Online Food Shop for iPhone
Year	2012
Language	English
Pages	115
Name of Supervisor	Ghodrat Moghadampour

Nowadays, mobile phones have become an essential part of a human life. Mobile phones make it possible to listen to music, surf on the Internet, and capture a picture and so on. In recent years Apple's iPhones have gained a lot of popularity and lots of applications have been developed for iPhones.

In this work client-server food shopping application was developed. The main function of the application is to list foods and places in Vaasa area and place order for the food. On the client side, Online Food Shop for iPhone, which is a native iOS application, has been developed by Objective-C (a programming language is used by Apple Inc.). The client application gets the food and place data from the web service, lists them and sends order data to the web service.

The server side application has been developed with Ruby on Rails programming language. The web service allows saving, modifying and searching data on the server, receiving orders from the client and storing client data. The server application has been deployed to cloud application platform, Heroku (<http://www.heroku.com/>), on which it is running too. The application data on the server side is saved on PostgreSQL database.

Keywords	Objective-C, Apple, iOS
----------	-------------------------

ACKNOWLEDGEMENT

My grateful appreciation goes to Professor Ghodrat Moghadampour as my thesis's supervisor. He gave me the golden opportunity to do the wonderful project on the topic Online Food Shop for iPhone, which later has been brought me many various knowledge on the field.

I also would like to thank my parents and friends who helped me a lot in finishing this project within the limited time.

I am making this project not only for marks but also increase my knowledge.

CONTENTS

Contents

1.1 Objectives.....	8
2 TECHNOLOGY OVERVIEW	9
2.1 JavaScript Object Notation (JSON)	9
2.1.1 Data types, syntax and example	9
2.1.2 JSON Parser	11
2.2 Cocoa Frameworks.....	14
2.2.1 Features of a Cocoa Application	14
2.3 iOS Technology Overview.....	15
2.3.1 The iOS Architecture	16
2.3.2 Core OS Layer.....	17
2.3.3 Core Services Layer	17
2.3.4 Media Layer	18
2.3.5 Cocoa Touch Layer	18
2.3.6 iOS Developer Tools.....	19
2.4 Ruby on Rails.....	19
2.4.1 Technical overview	19
2.5 Heroku.....	20
3 ONLINE FOOD SHOP FOR IPHONE	21
3.1 Functional analysis.....	21
3.2 Class Hierachy	23
3.4.1 Application classes.....	24
3.4.2 View controller classes	26
3.4.3 Table view controller classes	28
3.4.4 Core data table view controller	31
3.4.5 UI view classes.....	32
3.4.6 Core data classes	34
3.4.7 Supporting classes.....	36
3.3 Detailed Description of Main Operations	42
3.5.1 badgeValueUpdate function.....	42

3.5.2	fetchWebServiceIntoDatabase function.....	43
3.5.3	sortPlacesByDistanceFrom function.....	44
3.5.4	fetchData function.....	45
3.5.5	addToCart function.....	45
3.5.6	emptyCart function.....	46
3.5.7	PlaceOrder function.....	47
3.4	Component Diagram.....	49
3.5	Architectural Diagram.....	50
4	DATABASE AND GUI DESIGN.....	52
4.1	Design of the database.....	52
4.1.1	Food Table.....	52
4.1.2	Place Table.....	53
4.1.3	Cart Table.....	54
4.1.4	Entity Relationship.....	55
4.2	Design of different parts of GUI.....	56
4.2.1	Food view window.....	57
4.2.2	Location view window.....	62
4.2.3	Map view window.....	65
4.2.4	Cart view window.....	69
4.2.5	More view window.....	71
5	IMPLEMENTATION.....	76
5.1	Get the data from the web service:.....	76
5.2	Initialization for the document and location manager:.....	79
5.3	Adding the data from the web service to the core data:.....	80
5.4	Lazy loading the images from the web service:.....	86
5.5	Add a food item to Cart:.....	90
5.6	Calculate the distance from the user's location to places location:.....	91
5.7	Add the annotation pins to the map view:.....	93
5.8	Loading the images in the left call out accessory view:.....	97
5.9	Remove the cart item from the cart:.....	99
5.10	Empty the cart:.....	100

5.11	Send the order:	102
6	TESTING	108
6.1	The alert of adding more 5 items	109
6.2	Testing working of call out accessory.....	110
6.3	Calculate the price and badge value update	110
6.4	The action sheet of empty cart	111
6.5	The alert of place order	111
7	CONCLUSION	113
8	REFERENCES.....	114

ABBREVIATION

API	Application Programming Interface
GPS	Global Positioning System
XML	Extensible Markup Language
JSON	JavaScript Object Notation
Git	Distributed Revision Control
SCM	Source Code Management
SVN	Apache Subversion
GUI	Graphical User Interface
UI	User Interface
UXD	User Experience Design
PC	Personal Computer
HTML	HyperText Markup Language
URL	Uniform Resource Locator
OS	Operating System
OS X	Mac OS X
iOS	Intelligent Operating System
SDK	Software Development Kit
IDE	Integrated Development Environment
UUID	Universally Unique Identifier

1 INTRODUCTION

There has gone the time when mobile phone was only a simple tool for people to communicate with each other. Only had it call and text message function. Nowadays, a person can connect with the whole world just with a small device. Obviously, a modern mobile phone has improved both capacity and functionality. People not only utilize mobile phone for communicating purpose but also treat them as multi-functional devices, such as camera, the color screen, multimedia player, internet connection, etc. In 2007, Apple Inc., a famous computer company, introduced the most amazing phone in the world: iPhone. Undeniably, iPhone has changed totally the way the world thought about phone. With a combination a new technology in hardware and the stability of software, iPhone possesses much functionality like a small laptop. It is not a weird scene anymore when a person having an iPhone walking on the city street, enjoying the music, surfing the Internet, checking an email, playing thousands of mini-games, etc. In fact, iPhone makes impossible come into possible.

Seeing the advantages and the future of iPhone, the application has been developed based on iOS platform. The iOS is an operating system running on iPhone/iPad.

1.1 Objectives

The application is focused on cuisines and restaurants in Vaasa area. Users can save a lot of time to find ‘what to eat’ as well as ‘where it sells’. An Apple map included in this application will help people to know the nearest restaurant around them and the distance from where they are standing to the nearest restaurant. In addition, the pre-order food function enables users to choose food faster, more accurately and conveniently.

The reason why iPhone was chosen for the development platform as well as the goal of this app will be described. Next, the techniques and the programming languages used for this project will also be introduced.

2 TECHNOLOGY OVERVIEW

Below is a description of the technology programming used in this iPhone application. There are four primary parts: JSON document, Cocoa framework, iOS technology overview and Ruby on Rails document. In each part, there will be a definition, an example, advantages and disadvantages of technology programming.

2.1 JavaScript Object Notation (JSON)

JSON is the format of the documents used as data communication in this thesis application. Also, JSON has some advantages over other data communication like XML, YAML.

JSON is a short form of JavaScript Object Notation, which is a lightweight data-interchange format. JSON is syntax for storing and exchanging text information, which is made to be convenient in reading and writing for users. In addition, JSON also enables computers to prepare and generate information in an easy way. Even though JSON uses JavaScript syntax for describing data objects, it still does not depend on language as well as platform. Instead, C-family of languages, which are familiar to programmers, is utilized quite popularly. Some conventions can be named are Java, JavaScript, Perl, Python, etc. Thus, JSON is considered as an ideal data-interchange language. /3/

2.1.1 Data types, syntax and example

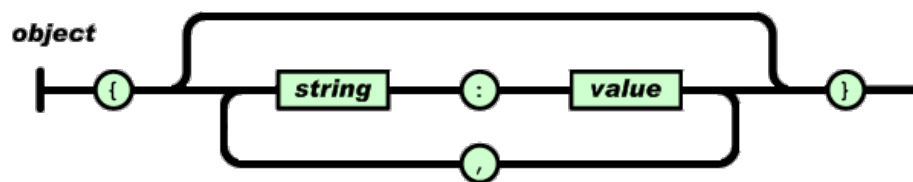
JSON's basic data types are:

Table 1. Basic data types of JSON.

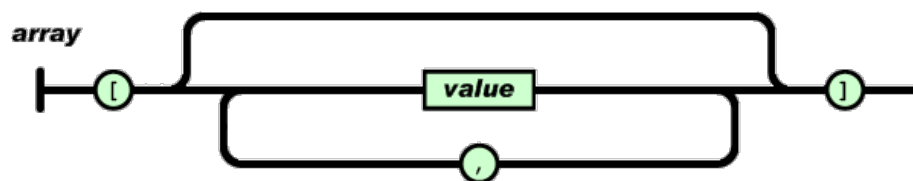
Type	Value
Number	double precision floating point format
String	Unicode (UTF-8)
Boolean	false or true
Array	an ordered list of values
Object	an unordered set of key/value pairs
null	empty

The forms of JSON:

An object begins with { (left brace) and ends with } (right brace). Each key is followed by : (colon) and the key/value pairs are separated by , (comma). (see Figure 1.).

**Figure 1.** Structure of JSON object. /1/

An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma). (see Figure 2.).

**Figure 2.** JSON array handling. /1/

The following example shows the JSON representation of an object that describes a food. The object has string field for name, image_url, ingredient, a number field for price, contains an object representing the place's attributes that have a string field for email, address, image_url, opening time, a number field for lat, log, and phone number.

```

{
  "food_image_url": "images/food/baileys_blended_creme_caramel_lg(1).png",
  "food_ingredient": "2 large Ice cubes, 50ml of Baileys with a hint of Crème Caramel
per person",
  "food_name": "Baileys Blended with a hint of Crème Caramel",
  "food_price": "4.6",
  "id": 1,
  "place": {
    "id": 1,
    "place_address": "Kauppapuistikko 8, 65100, Vaasa",
    "place_email": "info@olivers-inn.fi",
    "place_image_url": "images/place/OliversInn.jpeg",
    "place_lat": "63.097545",
    "place_log": "21.614159",
    "place_name": "Oliver's Inn",
    "place_opening_time_1": "tue: 18-04",
    "place_opening_time_2": "wen, thu: 18-02",
    "place_opening_time_3": "fri, sat: 18-04",
    "place_opening_time_4": "sun, mon: CLOSED",
    "place_phone_number": "+ 358 631 72970"
  }
}

```

Figure 3. This is JSON example from the application.

2.1.2 JSON Parser

In this part, the remote data types will be looked more in detail. Besides, since there are many different data types using common client-server interaction, a quick at those is required. Next, parsing data is also discussed here. The question is raised: how fast is it to parse those different data types? This plays a crucial role in making decision on what kind of data needed to send back and forth.

In a common client-server interaction, there are several different mode data types you can use. Figure 4 is shown the common data types:

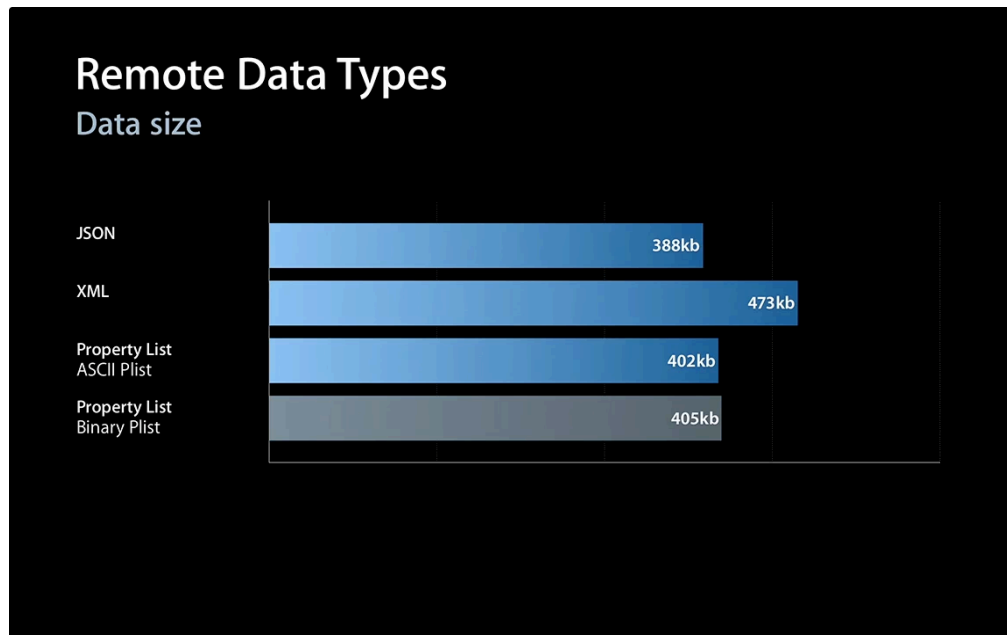


Figure 4. JSON data size remote data types. /5/

Have a look at the JSON. For example, a list of 500 flowers is taken, which takes 388kb to transfer that over the wire using JSON. In comparison with XML, the same data takes 473kb to transfer over the wire. Hence, a property list is used, which is a common format on Mac OS X and iOS, in the ASCII property list format in specific, we got 402kb and lastly, we have got binary property list as an option available and the same data with binary plist is 405kb. Now, we got the data on our client. Besides, one of important considerations is how speedy is that data going to parse. (see Figure 5.).

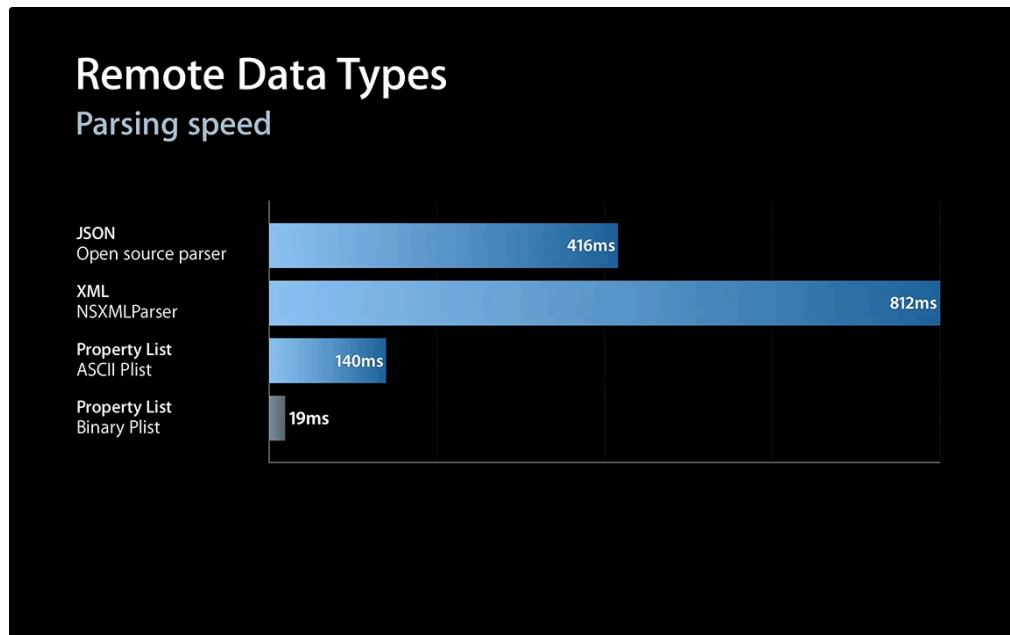


Figure 5. JSON parsing speed. /5/

From this figure, utilizing the same data again with JSON and an open source parser, we're at 416 milliseconds to parse that 500 elements list. With XML and NSXML parser, a whopping 812 milliseconds is reached. That's almost twice as slow as JSON. In property list, ASCII property list is 140 milliseconds, which takes one third in comparison with the time it takes for JSON. Lastly, the binary property list shows a quick blazing at 19 milliseconds to parse the same data on the iOS device. Based on the data above, JSON is obviously the best choice as the data type for sending back and forth.

JSON has some advantages over than XML:

- Small data size
- Very speedily parsing
- Straightforward to create (iOS 5 native support)
 - With Ruby, it is built-in.
 - Ruby on Rails is open-source web framework
- Undemanding to parse (iOS 5 native support)

2.2 Cocoa Frameworks

In order to develop iOS mobile software that the application can run on iPhone, iPad and iPod touch, Cocoa was selected as a primary development framework for this thesis application.

Cocoa is the name of application development environment for Mac OS X and iOS, the operating system used on iPhone, iPad and iPod touch. It was written in Objective-C, Cocoa and categorized in software frameworks. The three primary Cocoa frameworks are Foundation Kit, Application Kit and Core Data. Foundation Kit (mostly know in short form as Foundation) is functioned as supplying main classes that are wrapper classes and data structure classes. Prefix “NS” is required for this framework. Another Cocoa framework is Application Kit (or just AppKit as short form) whose code enables the creativity and interaction in programming graphical user interfaces. As well as the previous framework, prefix “NS” is also required in AppKit. The last one, Core Data, is an object graph and persistence framework supplied by Apple. Those above listed Cocoa frameworks are used commonly in providing common building blocks for all Mac applications. /2/

2.2.1 Features of a Cocoa Application

User interface objects – A wide range of packaged objects are available for your application’s user interface. Almost all these objects are available as the library in Interface Builder, a development tool for creating and designing user interfaces; you simply drag an object from the library and drop it onto the surface of your interface, configure attributes, and connect it to your code. /4/

This is some sampling of Cocoa user interface objects:

- UIView
- UILabel
- UIImageView
- UITextView
- UIButton

- UISegmentedControl
- UIActivityIndicatorView
- UIAlertView
- UISlider
- UIProgressView
- UIWebView

Drawing and imaging – Cocoa includes Quartz framework that helps users create 2D drawing based on the PDF imaging model, procedure curves, lines, animations, transformations, gradients, etc.

Performance – To increase application's performance, Cocoa produces some sample codes for lazy loading, multithreading, concurrency, memory optimizing, and run-loop manipulation.

Internationalization – Cocoa uses Unicode standard is the default. Moreover, it has considerable support for internationalizing applications. Some supporting localized resources such as the text, images and user interfaces.

Networking – Cocoa supports programmatic interface for communicating from client to server, taking advantage of Bonjour. Some of them are: NSURLConnection, NSSStream, CFStream.

Printing – Cocoa on both platforms supports printing. A person can print images, Word documents, PDF document, and the content from the Website with just a little code.

Multimedia – Cocoa support both video and audio player. They can be utilized to play many kinds of media types. In Mac OS X, Cocoa provides for QuickTime player.

2.3 iOS Technology Overview

iOS is the operating system developed by Apple Inc. for iPhone, iPad and iPod touch devices.



Figure 6. iPhone, iPad and iPod touch. /6/

From the knowledge on creation of Mac OS X, Apple built the iOS platform. Almost the tools and technologies used for development in Mac OS X can be used in iOS as well. In the face of its similarities to Mac OS X, iOS doesn't require any experience developing Mac OS X applications. The iOS Software Development Kit (SDK) supports everything you need to create iOS applications. /6/

2.3.1 The iOS Architecture

iOS is divided to kind of four layers. The bottom layer is close to the hardware and the top layer is close to your programming environment for the end-user. Figure 7 is shows layers of iOS architecture.

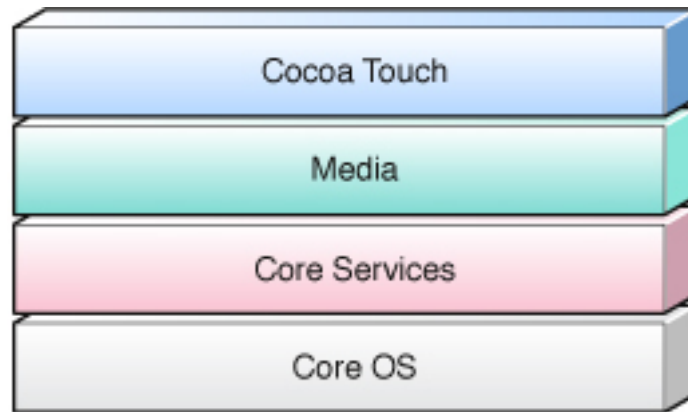


Figure 7. iOS architecture. /7/

2.3.2 Core OS Layer

The bottom layer of software in iOS is basically a Mac 4.x BSD Unix kernel. It's a Mac OS X operating system basically targeted for this device. That will give users a lot of power that enables full multitask unit kernel at the base. It is quite remarkable as that kind of operating system is on. In this layer, many things are included:

- OSX Kernel
- Mach 3.0
- Certificates
- Networking
- Sockets
- Security
- File System
- Bonjour
- Power Management
- Keychain Access

Most of these API is C API (not object oriented) because of kernel Unix code basically.

2.3.3 Core Services Layer

Built on top of those is layer being started more object oriented. It's providing a lot of the same services as layer below but with object oriented API. For example, sockets class is using object-oriented mechanism. This is also where there is a need for some languages runtime support for things like multithreading which is going to be done in this class. The reason is that it's a good interaction build user interface program need use multithread. Also, there are collection classes mentioned such as array, dictionary, all that kind of mechanism is included in

those layers. Multi object oriented layer provided basically a functionality covering Core OS. Core Services layer consists of:

- Collections
- Core Location
- Address Book
- Networking
- File Access
- SQLite
- Threading
- Preferences
- URL Utilities
- Net Services

2.3.4 Media Layer

It's kind of the next layer up away from the hardware. iPhone, iPad and iPod touch are a fundamentally multimedia devices in which are consisted of audio, video combination, FaceTime going, iPad library with the music for the class . Multimedia code runs throughout iOS, it's merely everywhere on build team. Almost API seen from the Core Services layer and up is taken into consideration as designing the minded about multimedia. What kind of media is in need here? How's it plugin? It can be a lot of multimedia going on in this class because this device is fundamentally really good for that. Media layer includes:

- Core Audio
- OpenAL
- Audio Mixing
- Audio Recording
- Video Playback
- JPEG, PNG, TIFF
- PDF
- Quartz (2D)
- Core Animation
- OpenGL ES

2.3.5 Cocoa Touch Layer

Cocoa Touch Layer is the top layer in iOS architecture. This layer is where all button, slider, view, navigation mechanism for navigating user interface, alert, high level media picking like camera takes the picture, photo library, that's all happening on cocoa touch entirely object oriented. This layer enables users to write a few line of code. Cocoa Touch layer consists of:

- Multi-Touch
- Core Motion
- View Hierarchy
- Localization

- Controls
- Alerts
- Web View
- Map Kit
- Image Picker
- Camera

2.3.6 iOS Developer Tools

The primary tool for developing application for iOS platform is Xcode application. Xcode is an integrated development environment (IDE) that supports all of the tools you need to create and manage your iOS project and source file, create and design your user interface, build your code into executable, debug and run your code either in iOS Simulator or on a real device. Apple definitely taking approach of one application does it all. /8/

The secondary tool's Instruments like a plugin let you manage the application do like memory usage, network activity, disc activity, and graphics performance.

2.4 Ruby on Rails

In order to develop the Web Service in the fastest time, easy programming, Ruby on Rails is the best choice for that work.

Ruby is a dynamic, reflective, general-purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk. Ruby was first designed and developed in the mid-1990s by Yukihiro "Mazt" Matsumoto in Japan. /11/

Ruby on Rails is often called as Rails, is an open source full stack web application framework. It lets you develop robust web applications in a matter of days. Ruby on Rails is not to be mystified with Ruby, which is a programming language. /9/

2.4.1 Technical overview

Ruby on Rails is based on Model-View-Controller architecture pattern like other web frameworks.

In a default configuration, in a database a table will be mapped by a model in a Ruby on Rails framework. Following the rule, the name of a database always is the plural form when the model's name is singular form. For example, the model has named Person, but in the database, the table will have name People. Also, the model file will place in /app/models with the filename person.rb.

A controller is a list actions method. In fact, it handles external request from web server to the application, and choose the view file to render. The controller also gets data from model directly and sends them to the view. Common actions in the controller are index, new, edit, update, show and destroy.

A view in Rails application has extension file name erb. It will be automatically converted to html at real time.

2.5 Heroku

Heroku (pronounced her-OH-koo) is a cloud application platform – a new way of building and deploying web apps. /12/

Heroku provide a platform as a service (PaaS) for building, deploying, and running cloud apps using Ruby, Node.js, Clojure, Java, Python and Scala. The architecture of this platform includes tools for deployment and management, a runtime for scalability, fault tolerance, and an add-ons system for extending the capabilities of this platform. /13/

Thousands of companies worldwide use Heroku to deploy their applications into the cloud. From small business to Fortune 500 companies, developers choose Heroku for its ease of use, reliability and performance. /13/

3 ONLINE FOOD SHOP FOR IPHONE

People who are attracted by food in Vaasa mainly use this Online Food Shop for iPhone application. They need to have information of foods and places in order to know where the best food to enjoy. Therefore, the application is made for easily using the information with less interaction with the phone.

User first browses the list of foods, price of foods, and place of foods. And then for each food, you can read information about this food like image, ingredients. Also, you can place an order for it if you want to enjoy it.

In order to know which restaurant is the nearest compared with the position a person is standing, the list of restaurants can be browsed with the display of name, and distance from his own iPhone. Furthermore, more details of destination such as address, phone number, and email are available in this app. User can also determine the user location and place on the map.

The application definitely requires an Internet connection (Wi-Fi or 3G). It connects to the web service to get the data, and load the images from the web service to the application. Also, it uses an Internet connection to display the map and location of places, image of places. Moreover, it also uses GPS signal to specify the location of place and user's location.

3.1 Functional analysis

The primary and the most important feature of the application is display the foods and places. Also, the application can display the distance of places, location of places and the user's location on the map. The application will get the data from the web service.

In addition, the application provides the order operation. The user can use this operation to order the food with only one click.

Table 2. Main features and their priorities, where 1 indicates the highest priority.

Task number	Name	Priority
1	Display food	1
2	Display place	1
3	Show food's detail	2
4	Show place's detail	2
5	Show places on map	2
6	Add food to cart	2
7	Send the order request	2
8	Empty cart	3
9	Delete food from cart	3

The user can browse the list of foods and places. After choosing each food and place, the user can see more detail of it. Also, the user can browse the place on real map, seeing clearly the user's position and the location of places. He can also know the distance between the places and the user's location. Furthermore, if he wants to eat or drink something, they can place an order with one click.

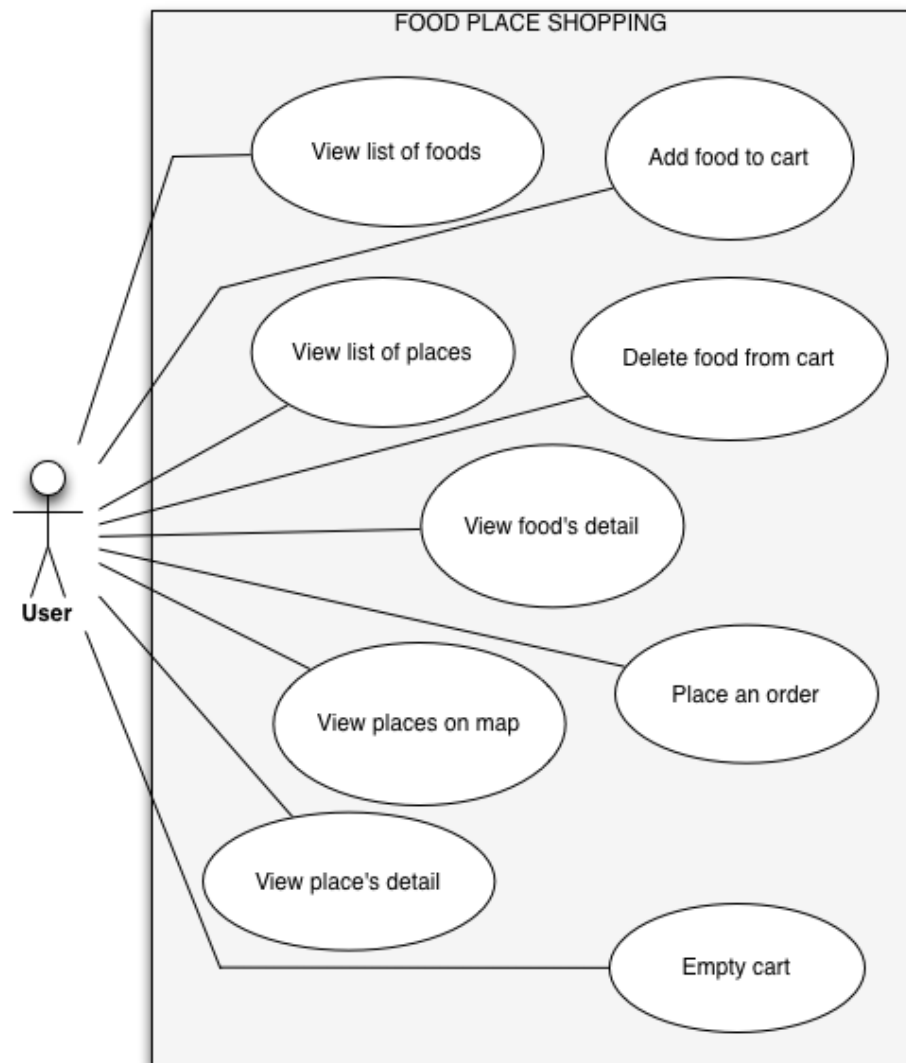


Figure 8. Application main operations.

3.2 Class Hierachy

This application contains a lot of classes that related together. It begins from NSObject (the root class in Objective-C) and every behind classes almost inherit from this class. View controller in iOS architecture needs a class. So if you have eight view controllers, you have to create eight classes supporting for them. This application has eleven view controllers; thus, eleven classes must be created in order to support for them. In addition, other classes for the service, UI, and Core Data are also utilized.

3.4.1 Application classes

The structure of application has a lot of different child classes. In fact, thirty-six classes are created for this application. Eight classes of them are system class and the rest are custom classes. The structure of class is drawn by this diagram.

3.4.2 View controller classes

The basic view controller is view controller class. In this application, view controller is the controller (except the word view is like the UI). In this part, the view controller will be used with the short name is the controller (for easier understanding). There are five UI view windows and following the iOS 5 architecture (storyboard), each UI view window needs at least one controller. Thus, it is a must to create five controllers for them.

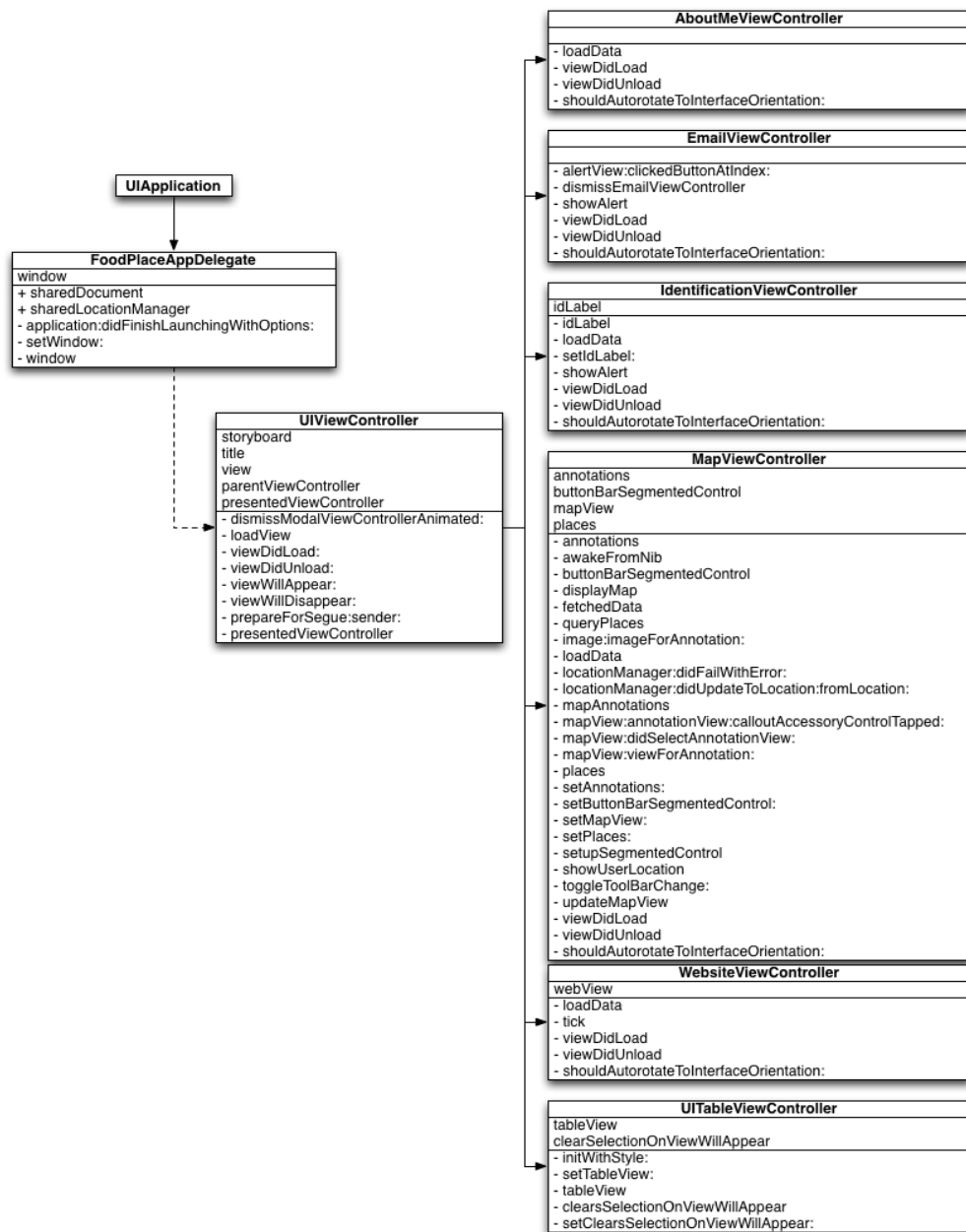


Figure 10. View controller classes.

As be seen from the diagram, there are:

UIApplication: this is a system class and it's one class of the UIKit framework. It inherits from root class NSObject. The UIApplication (or subclass of UIApplication) is a requirement for every application. When the application started, the UIApplication main operation is called. This function will create a singleton UIApplication object, so it is accessible this object by invoking the sharedApplication class method.

FoodPlaceAppDelegate: this is an application delegate. It is normally used to perform tasks on application startup and shutdown, handling URL open request and similar application wide task. This class includes one property:

- **Window** this class from UIWindow class that manages and specifies position the windows an application on the screen.

UIViewController: this is a system class, provides the fundamental view-management model for all iOS apps. It has many properties. The most property used is title. This property has function in setting the title to the UI view window.

AboutMeViewController: this class inherits from UIViewController class. It's responsible for 'about me' UI view window.

EmailViewController: this class is a child of UIViewController class. It's designed for email UI view window.

IdentificationViewController: this class inherits from UIViewController class. It gets the identification to display identification UI view window. It has one property:

- **idLabel:** The identification of the iPhone.

MapViewController: This class is a child of UIViewController class. It designs the map and annotations on the map UI view window. It has four properties:
annotations: the annotations of the places.

buttonBarSegmentedControl: The button bar segmented control on UI view window.

mapView: It inherits from MKMapView, functioned in displaying embedded map interface in the UI view window.

places: All places loaded from the web service.

WebsiteViewController: this class inherits from UIViewController class. It initializes the embedded website in website UI view window. It has one property:

- webView: It displays embedded website to UI view window.

UITableViewController: this system class is a child of UIViewController class. It helps the view display the table view. It has many properties, one of which frequently used is tableView to initialize the view of table.

3.4.3 Table view controller classes

This class is a system class inherits from UIViewController class. It mainly uses for managing the table view. Similar to the UIViewController class, this class will be called with the short name is controller in this part. In the application, there are three table views (inherit from UITableViewController class) and three table view controller classes have to be created to support them.

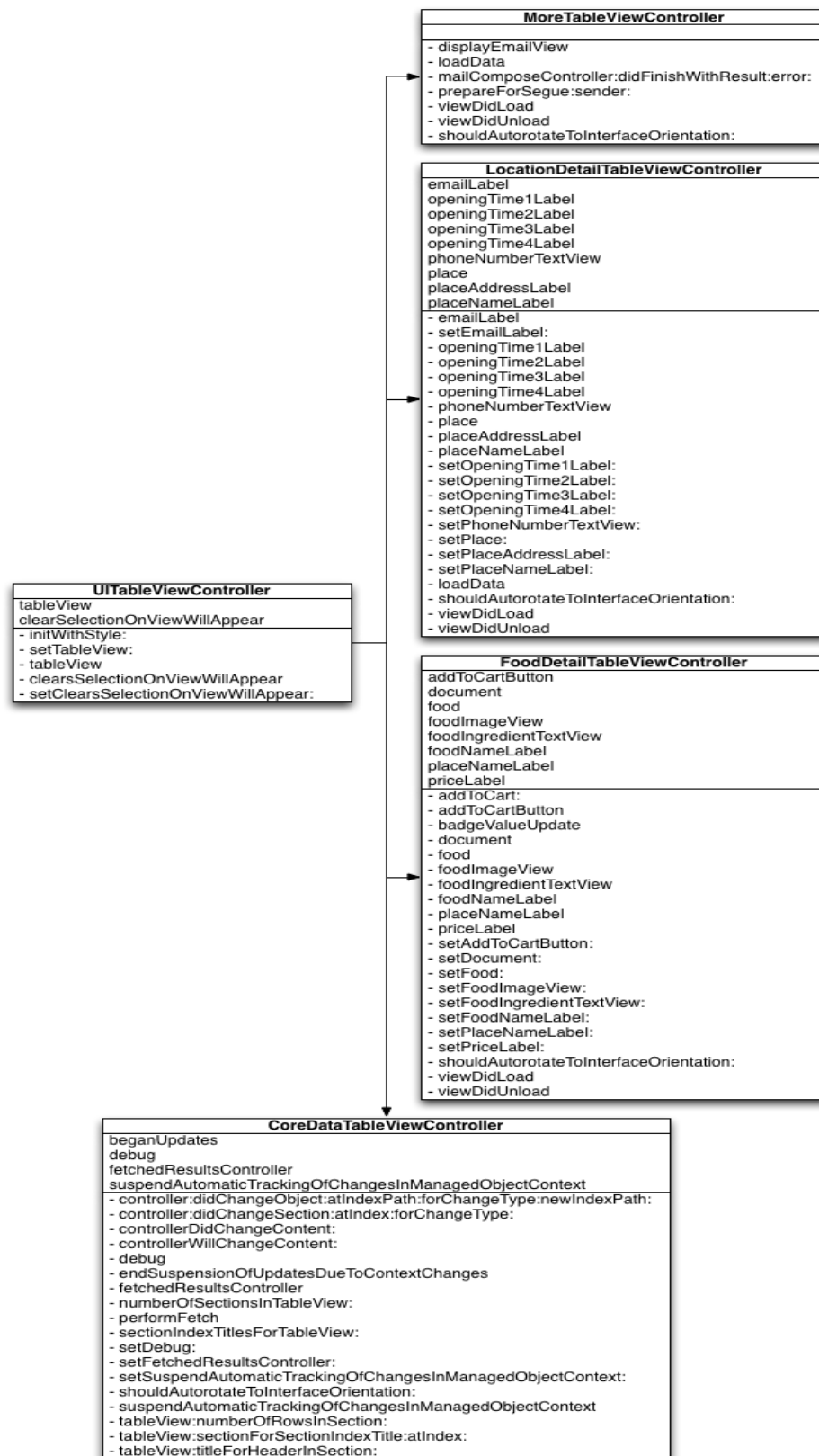


Figure 11. Table view controller classes.

Following the diagram, they are:

MoreTableViewController: this class inherits from `UITableViewController` class. It lists information about this application such as about me, identification, email, web site.

LocationDetailTableViewController: this class is a child of `UITableViewController` class. It lists detail of location like email, opening time, phone, address, name of location. It has nine properties:

- `emailLabel`: an email of the place.
- `openingTime1Label`: an opening time 1 of the place.
- `openingTime2Label`: an opening time 2 of the place.
- `openingTime3Label`: an opening time 3 of the place.
- `openingTime4Label`: an opening time 4 of the place.
- `phoneNumberTextView`: a phone number of the place.
- `place`: the information of current place.
- `placeAddressLabel`: an address of the place.
- `placeNameLabel`: a name of the place.

FoodDetailTableViewController: this class inherits from `UITableViewController` class. It shows the detail of the food. Eight consisted properties are:

- `addToCartButton`: an add to cart button.
- `document`: a document of this table view.
- `food`: the information of current food.
- `foodImageView`: an image of the food.
- `foodIngredientTextView`: an ingredient of the food.
- `foodNameLabel`: a name of the food.
- `placeNameLabel`: a name of place of the food.
- `priceLabel`: a price of the food.

3.4.4 Core data table view controller

CoreDataTableViewController is a custom class inherits from UITableViewController class. It's nearly the same as UITableViewController class but it adds some properties to check and debug the table view. It's a super class for three behind child classes.



Figure 12. Core data table view controller classes.

Following the diagram, three child classes are seen:

CartTableViewController: this class inherits from CoreDataTableViewController class. It lists cart item in the cart UI view window. It has five properties:

- cartLabel: a cart information text, it will display when no cart item in cart view window.
- document: a document of this view.
- emptyCartBarButtonItem: an empty cart bar button item on navigation bar.
- placeOrderBarButtonItem: a place order bar button item on navigation bar.
- totalOrderLabel: a total order text.

FoodTableViewController: this class is a child of CoreDataTableViewController class. It lists food item in food UI view window. It has three properties:

- document: a document of this view.
- spinner: a spinner for spinner view.
- imageDownloadsInProgress: the processing images for the food view.

LocationTableViewController: this class inherits from CoreDataTableViewController class. It lists location item in location UI view window. It only has one property document.

3.4.5 UI view classes

This class is a child of NSObject class. It's a basic UIView class in iOS application. Also, it has a lot of properties and operations. Figure 3.4.5 will show the diagram:

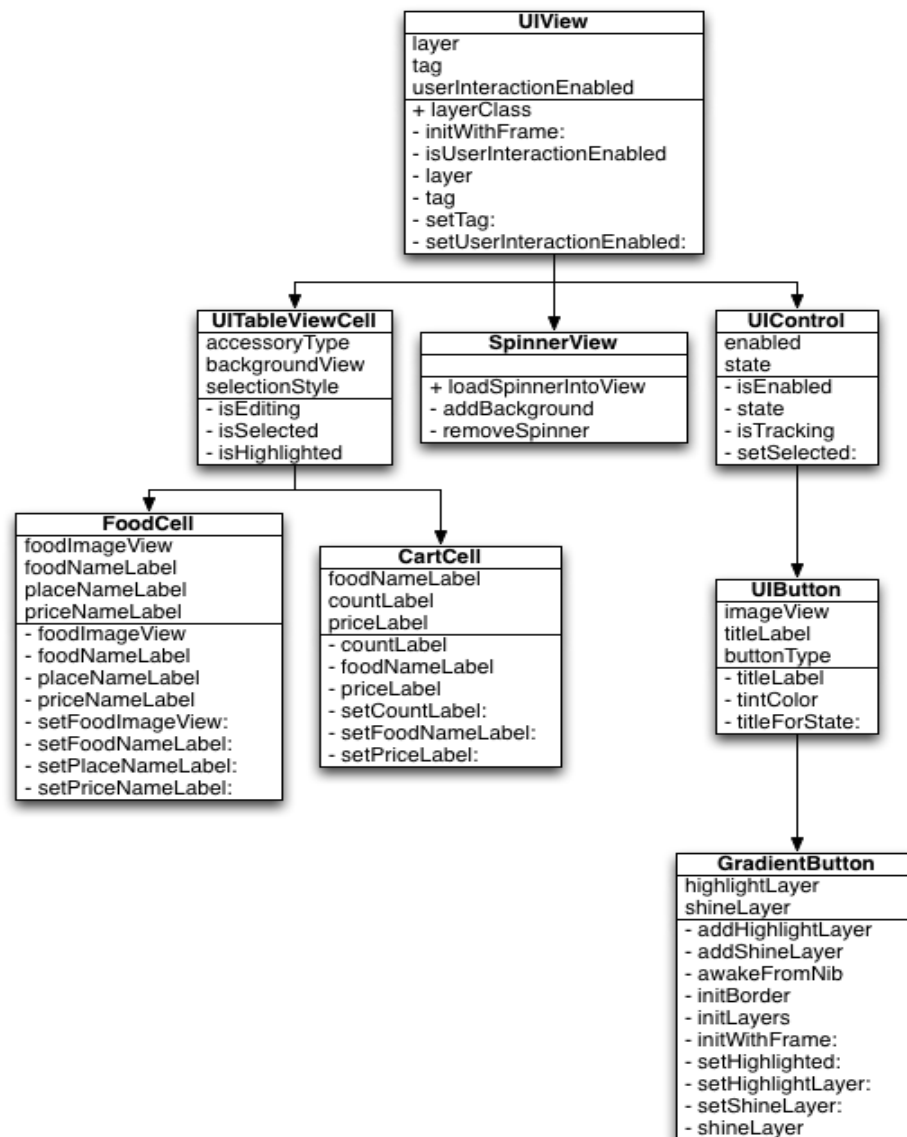


Figure 13. UI view classes.

In the application, there are three child classes of UIView class.

UITableViewController: this class inherits from UIView class. It designs a UI cell view for the table view. I have created two child classes of UITableViewController class for designing the cell in the table view.

FoodCell: this class inherits from UITableViewController class. It helps us create the custom food table view cell. It has four properties:

- `foodImageView`: an image view of the food.
- `foodNameLabel`: a name of the food.
- `placeNameLabel`: a name of place of the food.
- `priceLabel`: a price of the food.

`CartCell`: this class is a child of `UITableViewCell` class. It uses for designing cart table view cell. It has three properties:

- `foodNameLabel`: a name of the food.
- `countLabel`: a count of the food.
- `priceLabel`: a price of the food.

`SpinnerView`: this class is a child of `UIView` class. It designs spinner for the spinner view.

`UIControl`: this is a parent class of `UIButton` class.

`UIButton`: it's a basic UI button class. It's responsible for UI button on the view. It has one custom child class is `GradientButton` class.

`GradientButton`: this is an extending the UI button class. It helps us create more beautiful button. It has two properties:

- `highlightLayer`: it uses for highlight button pressed.
- `shineLayer`: it uses for normal button but make button looks more shiner.

3.4.6 Core data classes

Core Data is a framework in iOS and Mac architecture. It's a centralization data management in iOS and Mac application. In my application, there are three classes using core data framework. Also, they are child classes of `NSManagedObject` class.

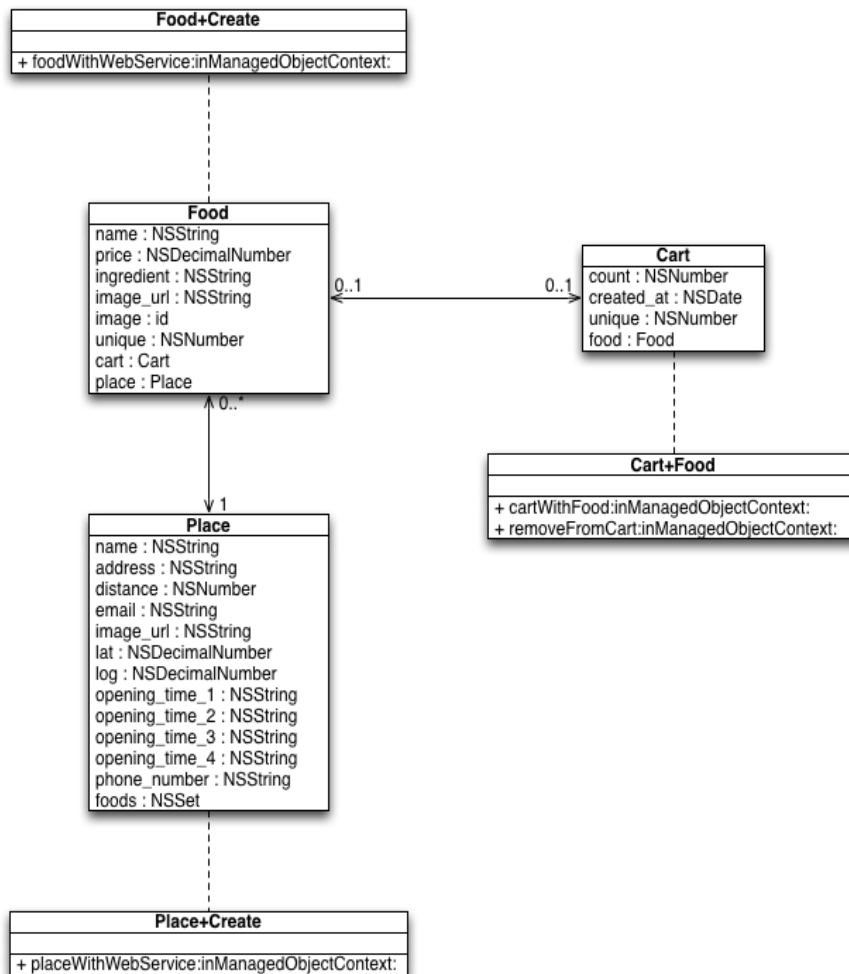


Figure 14. Core data classes.

Food: this class is a persistence class of food entity. It has eight properties: name, price, ingredient, image_url, image, unique, cart, and place.

Place: this class is a persistence class of place entity. It has twelve properties: name, address, distance, email, image_url, lat, log, opening_time_1, opening_time_2, opening_time_3, opening_time_4, foods.

Cart: this class is a persistence class of cart entity. It has four properties: count, created_at, unique, food.

Also, these entity classes are generated by core data framework. There is no point in modifying anything from them. In Objective-C, a category allows the

application to add methods to an existing class. Hence, three categories support are created for three entity classes.

Food+Create: this is a category for food entity class. It has one operation:

- `foodWithWebService:inManagedObjectContext`: this operation copies the data of food from the web service to core data food.

Place+Create: this is a category for place entity class. It has one operation:

- `placeWithWebService:inManagedObjectContext`: this operation copies the data of place from the web service to core data place.

Cart+Food: this is a category for cart entity class. It has two operations:

- `cartWithFood:inManagedObjectContext`: this operation will add food to cart. The action only happens in core data cart.
- `removeFromCart:inManagedObjectContext`: this operation will remove food from cart. The action only happens in core data cart.

3.4.7 Supporting classes

For the purpose of helping the controller, six supporting classes are created.

3.4.7.1 *FoodPlaceFetcher* class

The first one mentioned here is `FoodPlaceFetcher` class stores operations get the data from the web service.



Figure 15. `FoodPlaceFetcher` class.

This class has four operations:

- `getPlaces`: this method will get all data of place from the web service.

- `getFoods`: this method will get all data of food from the web service.
- `urlStringforPlace`: this method will get url string (NSString) from information of place.
- `urlForPlace`: this method will return to url (NSURL) from url string of place.

3.4.7.2 *LazyImageDownloader class*

The second is a `LazyImageDownloader` class, downloads the images from the web service and displaying them on the phone. It's lazy because it doesn't download all the images from the web service. But it's only download necessary image when the user needs it. It helps to save a lot of the bandwidth.

LazyImageDownloader
food indexPathInTableView delegate
- startDownload - delegate - downloadError: - emptyReply - food - imageWithData: - indexPathInTableView - readHttpStatusCodeFromResponse: - setDelegate: - setFood: - setIndexPathInTableView: - timeOut

Figure 16. `LazyImageDownloader` class.

This class has three properties:

- `food`: the information of the food.
- `indexPathInTableView`: the index path in the table view.
- `delegate`: the delegate of this class.

And twelve operations:

- `startDownload`: start download the images.
- `delegate`: getter of delegate.

- `downloadError`: display the error when it occurs.
- `emptyReply`: the empty reply will be shown.
- `food`: getter of food.
- `imageWithData`: return the image with pre data.
- `indexPathInTableView`: getter of index path in table view.
- `readHttpStatusCodeFromResponse`: read the http status code from response data.
- `setDelegate`: setter of delegate.
- `setFood`: setter of food.
- `setIndexPathInTableView`: setter of index path in table view.
- `timeout`: display error time out of network connection.

3.4.7.3 *Helpers class*

The third is a Helpers class.

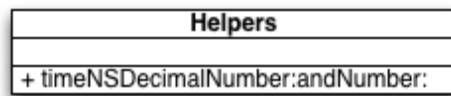


Figure 17. Helpers class.

This class has only one operation:

- `timeNSDecimalNumber:andNumber:` time two number is not the same type.

3.4.7.4 *PlaceAnnotationView class*

The forth is a `PlaceAnnotationView` class, designed for the annotation on the map interface.

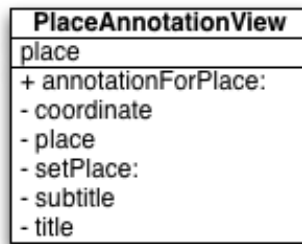


Figure 18. PlaceAnnotationView class.

This class has one property:

- place: is stored the information about restaurants.

And six operations:

- annotationForPlace: specify an annotation for the place.
- coordinate: a coordinate of the place like latitude and longitude.
- place: getter of place.
- setPlace: setter of place.
- subtitle: this is a subtitle for the annotation.
- title: this is a title for the annotation.

3.4.7.5 *OrderUploader class*

The fifth is a OrderUploader class, designs for the uploading the data to the web service.

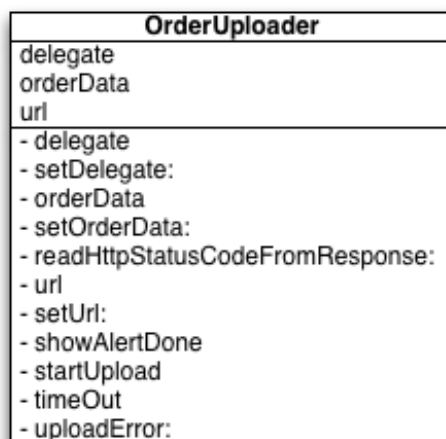


Figure 19. OrderUploader class.

This class has three properties:

- delegate: the delegate of this class.
- orderData: the data to be uploaded.
- url: an url with NSURL type.

And eleven operations:

- delegate: getter of delegate.
- setDelegate: setter of delegate.
- orderData: getter of orderData.
- setOrderData: setter of orderData.
- readHttpStatusCodeFromResponse: read http status code from response data.
- url: getter of url.
- setUrl: setter of url.
- showAlertDone: show alert when done.
- timeOut: display error time out of network connection.
- uploadError: display error when uploading data occurs an error.
- startUpload: start upload the data.

3.4.7.6 *BadgeValue class*

The last class is a BadgeValue class is responsible for show or hidden the badge value on the cart tab bar item.

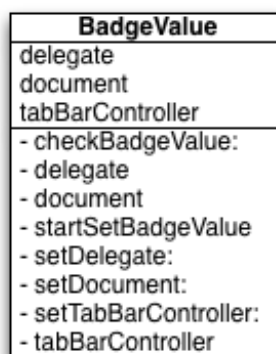


Figure 20. BadgeValue class.

This class has three properties:

- delegate: the delegate of this class.
- document: the document of this class.
- tabBarController: tab bar controller.

And eight operations:

- checkBadgeValue:: check the count and display the count on tab bar item.
- delegate: getter of delegate.
- setDelegate: setter of delegate.
- document: getter of document.
- setDocument: setter of document.
- tabBarController: getter of tab bar controller.
- setTabBarController: setter of tab bar controller.
- startSetBadgeValue: start to set a badge value.

3.4.7.7 Main operations

The controllers have many primary operations that received the data from the web service, processing the core data, and user interaction on the screen. Some of them will be described:

- badgeValueUpdate: check and display the badge value on the tab bar cart item.
- fetchWebServiceIntoDatabase: get data from the web service and add the data to core data on the phone.
- sharedDocument: initialize the document from the beginning.
- sharedLocationManager: initialize the location manager from the beginning.
- sortPlacesByDistanceFrom: calculate the distance from user's location to location of place.
- fetchedData: fetch data from the web service.
- mapAnnotations: add the annotations to places.

- addToCart: add the food item to cart.
- removeFromCart:atIndexPath: remove the cart item from cart at index path.
- emptyCart: emty cart.
- sendOrder: send the order to the web service.
- prepareOrder: prepare the order before sending the order.
- startOrderUpload:withData: order upload information with data (data of the food order).

3.3 Detailed Description of Main Operations

This part will go more detail of main operations.

3.5.1 badgeValueUpdate function

This operation checks the cart item in the cart and display how many the cart items on the tab bar item. It will help the user know how many cart items in the cart even the user is not on the cart view window. The sequence diagram is shown below:

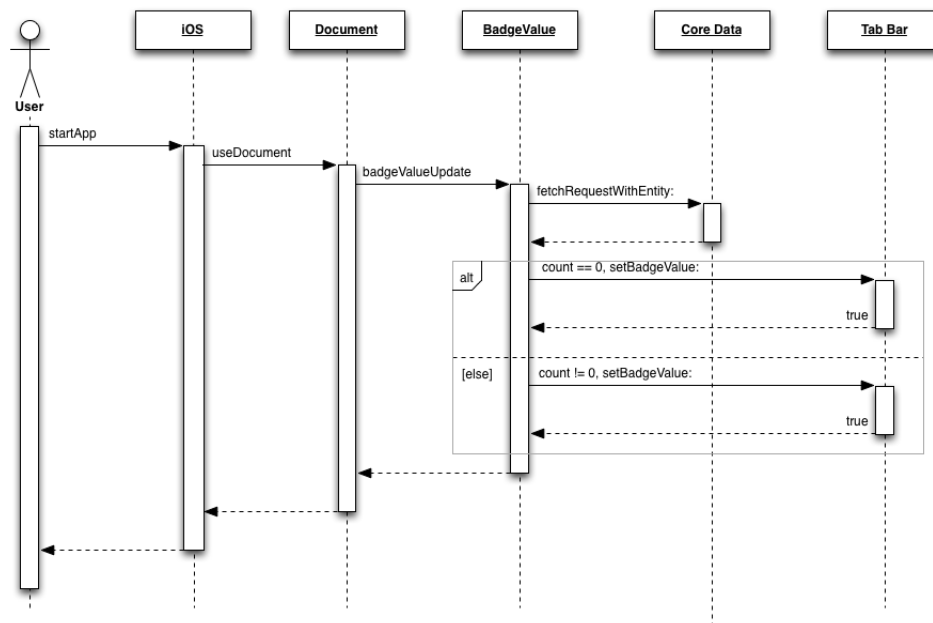


Figure 21. badgeValueUpdate operation.

Explain about some operations in this sequence diagram:

When the user starts the application, the application will use the document (useDocument operation). After this, it will run the operation badgeValueUpdate. This operation has some child operations:

- `fetchRequestWithEntity`: this operation will fetch the cart entity and it will return how many cart items in cart entity.
- `setBadgeValue`: from the cart item, this operation will specify the number cart item on the tab bar item. If no cart item, the number will disappear on tab bar item.

3.5.2 `fetchWebServiceIntoDatabase` function

This operation will fetch the data from the web service and copy it to the core data (the data storage on the phone). The sequence diagram is shown below:

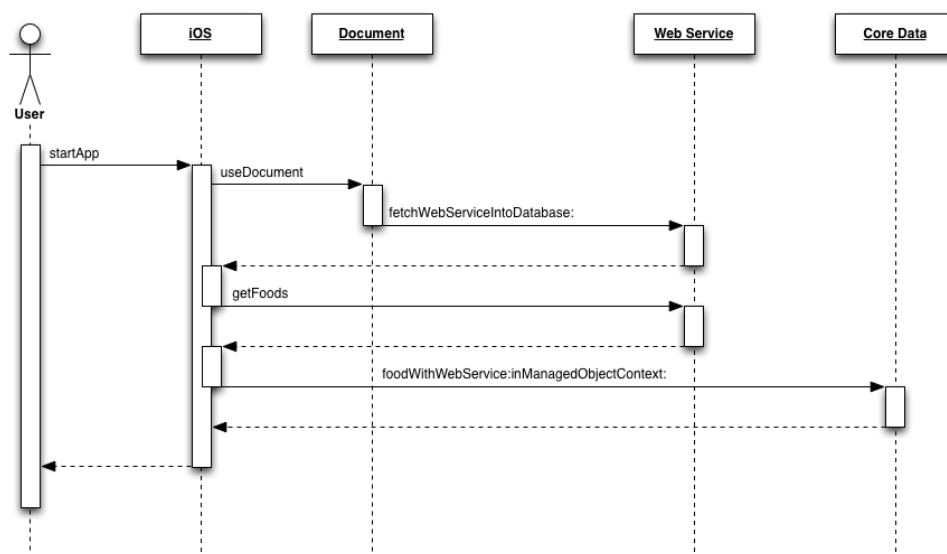


Figure 22. `fetchWebServiceIntoDatabase` operation.

Explain about some operations in this sequence diagram:

When the user starts the application, the application will use the document (useDocument operation). And the main operation `fetchWebServiceIntoDatabase` will be started.

The operation `getFoods` in `FoodPlaceFetcher` class will run to fetch the food data from the web service.

And the operation `foodWithWebService:inManagedObjectContext:` will copy the data (got from the web service) to the core data (the data storage on the phone). The iOS will use these data to display for the user.

3.5.3 `sortPlacesByDistanceFrom` function

This operation will calculate the distance between the user's location and location of wanted restaurant. The below diagram will explain how it works:

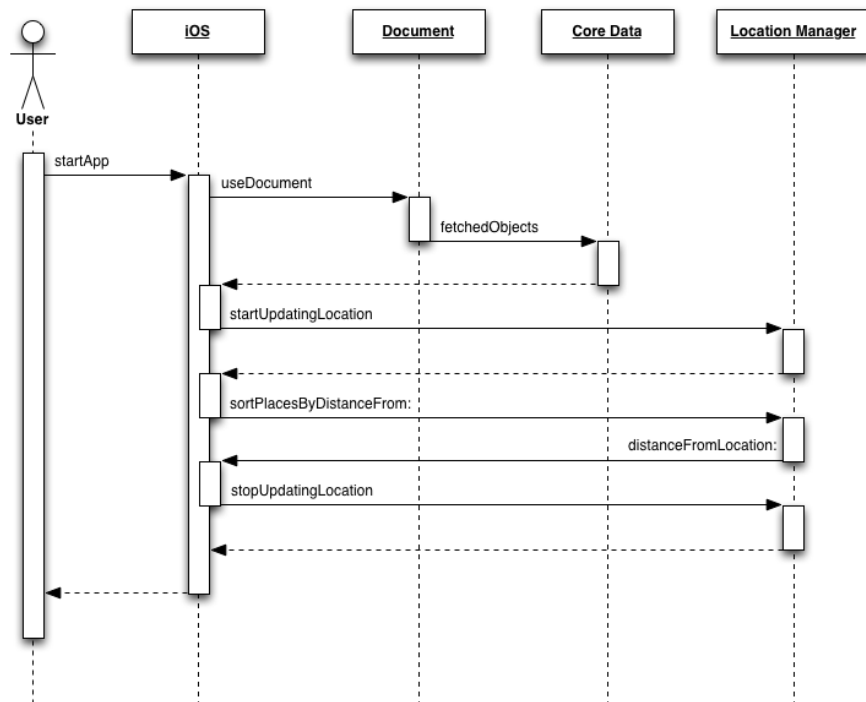


Figure 23. `sortPlacesByDistanceFrom` operation.

The process is the same as previous processes. It will begin with use document operation (`useDocument`) to use document on the application. Next, the operation `fetchedObjects` will fetch the place data from the core data (the data storage on the iOS). Also, the location manager will start by the operation `startUpdatingLocation` to get the user's location. After that, it will use the operation

sortPlacesByDistanceFrom to collect the location of places and calculate the distance from user's location to location of restaurants. The operation distanceFromLocation is used to calculate the distance. Finally, the operation stopUpdatingLocation utilized to stop updating user's location for the reason of saving battery and bandwidth of the Internet.

3.5.4 fetchedData function

This operation is placed in MapViewController class. It will be used for fetching the data of places from the web service. The sequence diagram is shown below:

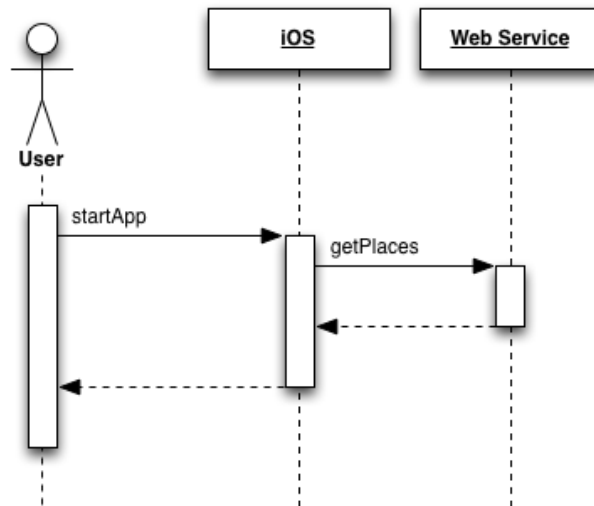


Figure 24. fetchedData operation.

The main operation is getPlaces operation in FoodPlaceFetcher class. It will get the data of places from the web service and copy it to the application (not Core Data).

3.5.5 addToCart function

This operation will add the food item to the cart. The below diagram will explain the process:

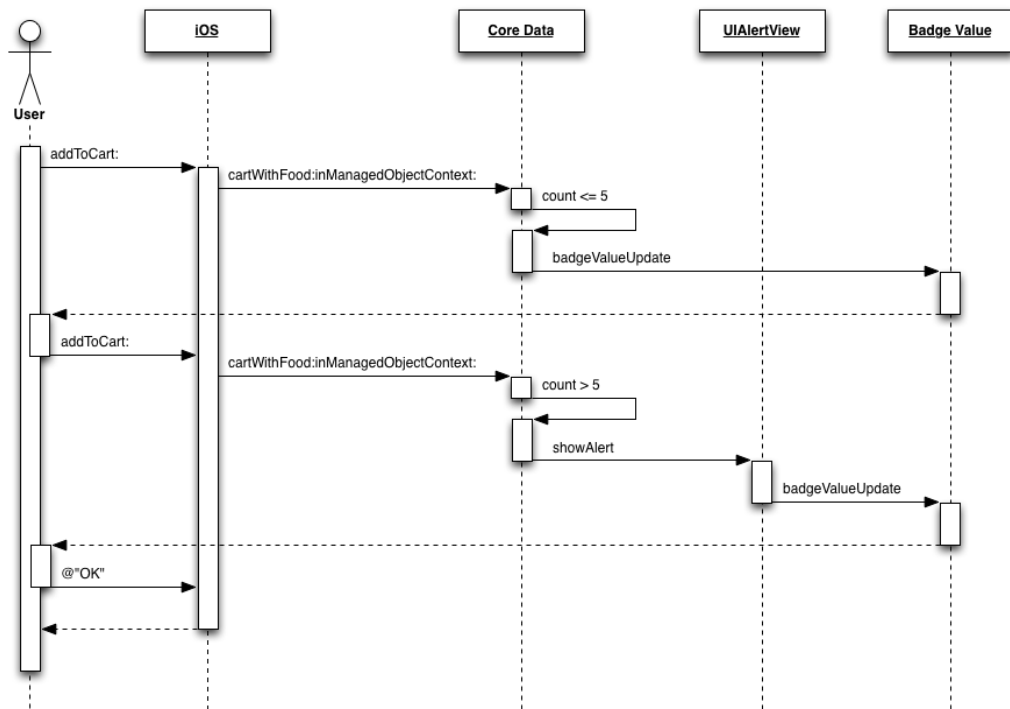


Figure 25. addToCart operation.

When the user is in the food item view window, the food item view window has a button add to cart. As he presses the button add to cart, the process add to cart begins. It will add currently food to the cart. This operation `cartWithFood:inManagedObjectContext` will connect to the core data and add the food item to cart entity. Furthermore, the operation `badgeValueUpdate` will update the cart item on tab bar item as adding the food item to cart. The application will be set default with limitation of five items per food kind. Thus, if the amount of items is counted larger than five, the alert will show information informing that there is a cheating on the application.

3.5.6 emptyCart function

This operation do empty the cart, it will remove all cart item in the cart. The sequence diagram will explain below:

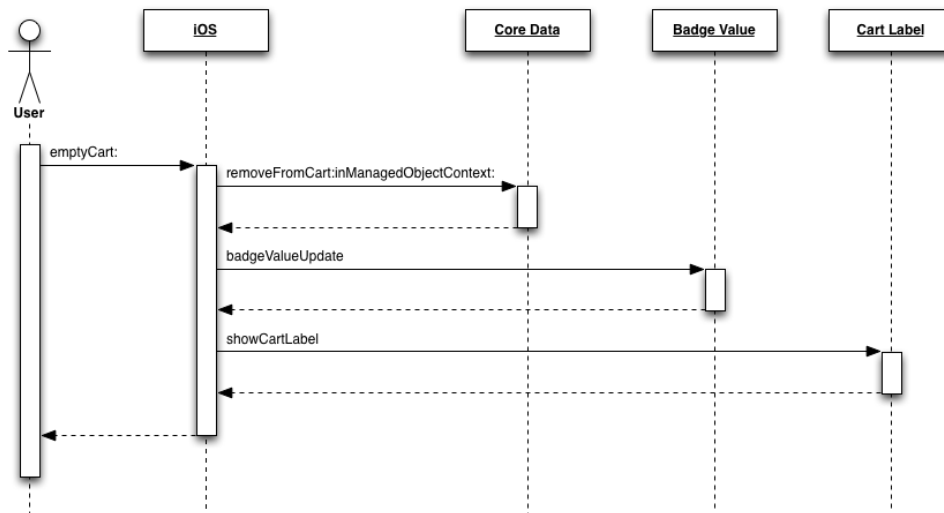


Figure 26. `emptyCart` operation.

When user presses the empty cart button in the cart view window, the cart view will be emptied. The operation `emptyCart` will process some child operations:

- `removeFromCart.inManagedObjectContext`: this operation is responsible for removing the cart item from cart. It connects to the core data, finds cart item and removes it from cart entity.
- `badgeValueUpdate`: after emptying the cart, the badge value should be update to 0. This operation will be updated the badge value. And it will be disappeared the number on the tab bar if nothing's in cart.
- `showCartLabel`: when the cart is empty, the cart will have the label inform the cart is empty.

3.5.7 PlaceOrder function

This is a most important operation in this application. It will get the food data in the cart and send it to the web service to place the order. The sequence diagram is explained below:

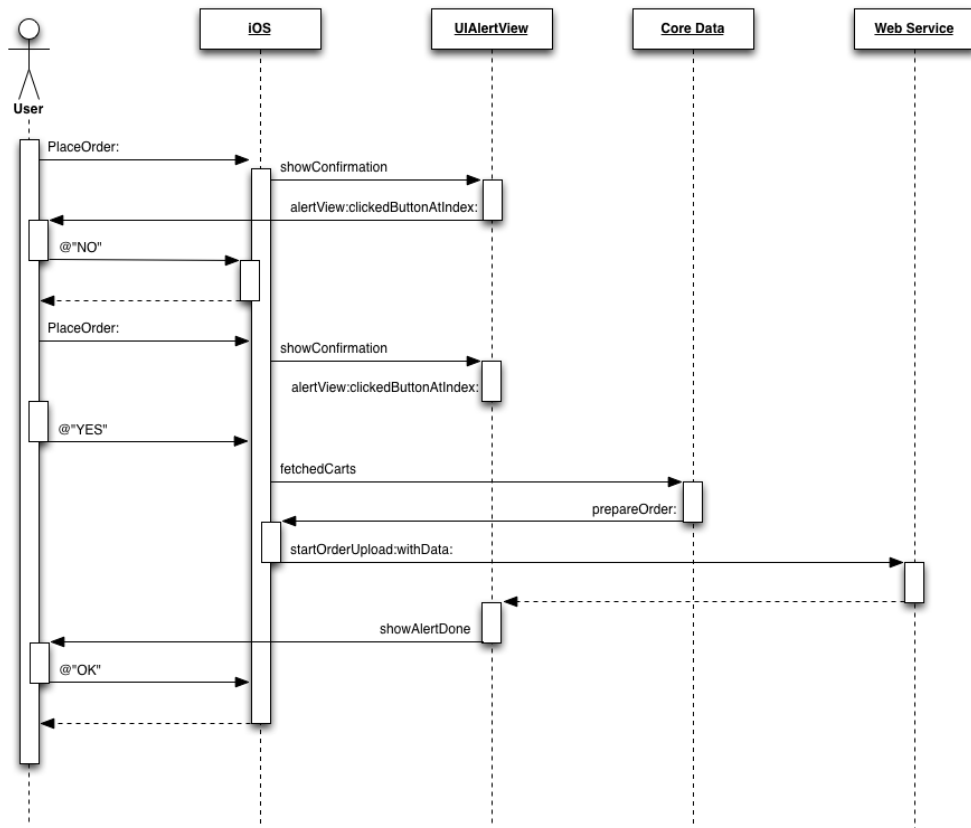


Figure 27. PlaceOrder operation.

When the user finishes adding wanted food item to the cart, he can place the order by pressing the place order button on the right bar system of the cart view window. This button will do the operation PlaceOrder and some child operations as following:

- **showConfirmation:** this is an operation showing the alert when the user press place order button. It confirms whether he really wants or doesn't want to place an order. If the choice is NO, there is no order. Otherwise, as it is YES, that means you want to place an order.
- **alertView.clickedButtonAtIndex:** this is a class of UIAlertView delegate. This operation helps user to set YES or NO operation. In this case, I specify YES to send an order and NO to happen nothing.

- `fetchCarts`: this operation will fetch all cart item in the cart entity and return to a array cart item. It connects the core data and queries the cart entity to get the cart item.
- `prepareOrder`: this operation gets the array cart item and enter the data following the JSON format. Furthermore, it adds some information about the order like uuid, total price of cart items, aorder date. This operation is comparatively complicated because it converts all data to JSON format. It uses a lot of `NSDictionary` and `NSArray` to carry out this process.
- `startOrderUpload:withData`: this operation will upload the data to the web service with the preparing data and the url of the web service. It uses `NSMutableURLRequest` to request the url of the web service and using the `NSURLConnection` to send the request to the web service. Besides, this is `AsynchronousRequest`, so it does not block the main thread of the iOS.
- `showAlertDone`: when the sending data to the web service is successfully, iOS will show the alert inform the success of the order. Otherwise, it will log the fail sending the data.

3.4 Component Diagram

There are nine groups in this iOS application: `UIAlertViewE`, `NSDateE`, `JSONE`, `Cryptography`, `Helpers`, `View Controllers`, `Web Services`, `Core Data`, and `Images`.

- Group `UIAlertViewE`: this is extension of `UIAlertView`.
- Group `NSDateE`: this is extension of `NSDate`.
- Group `Cryptography`: this is cryptography.
- Group `JSONE`: this is a JSON converter.
- Group `Helpers`: this group includes helper classes like `LazyImageDownloader`, `OrderUploader`, `GradientButton`,...
- Group `View Controllers`: store controller of this application.
- Group `Web Services`: responsible for connecting to the web service to get the data.
- Group `Core Data`: the data storage on the iOS application.
- Group `Images`: image for tab bar and about view window.

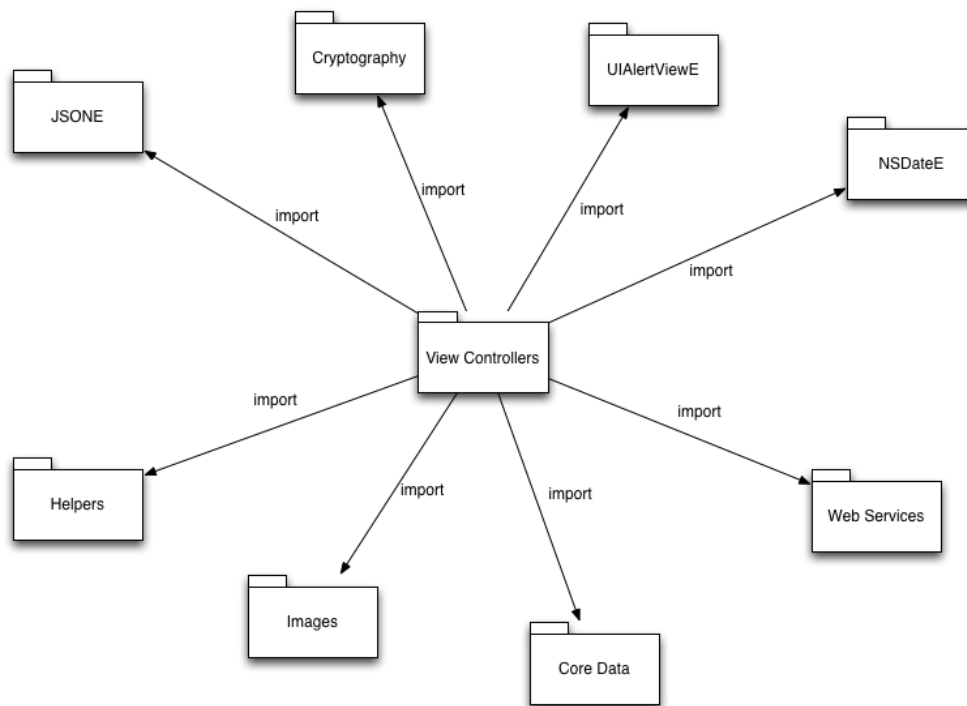


Figure 28. Component diagram.

3.5 Architectural Diagram

The below diagram will describe the architecture of this application.

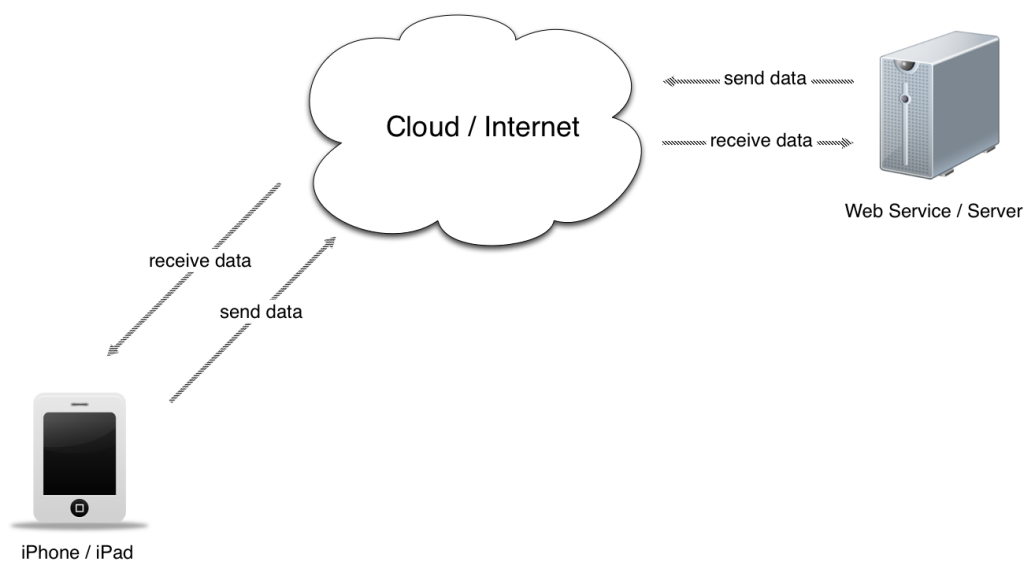


Figure 29. Architecture diagram.

The picture shows the architecture diagram of this iOS application and web service. Basically, almost application needs a data and so does this application. I don't want to place a data in the application because of a size of application. And the data in the application can't be updated in the future. Thus, there is a need for a web service for storing the data supporting data for iOS application. The iOS application can connect to the web service to retrieve the data through a cloud (internet). Nowadays, the Internet becomes more and more popular; hence, the Internet connection for iOS application isn't that much expensive. The application will connect to the web service to get the data and show those data to the user. The benefit of the web service is making the iOS application lighter weight. The iOS application doesn't need to store any data. The data will be gotten from the web service and it will be updating dynamically. In addition, the web service is used for receiving the data from the iOS application. The type of data is in texts, numbers and images. Only the texts and numbers are stored on the iOS application after being received from the web service. Since product images are such big data, it is unnecessary to store them on the iOS application. They will stream directly from the web service to iOS application. In another words, no image is saved on iOS application. It's just streamed from web service.

4 DATABASE AND GUI DESIGN

In the following the procedure of designing the database and the graphical user interface is described in detail.

4.1 Design of the database

The Core Data framework is used in this iOS application in order to save a rarely data changes in this iOS application.

Core Data is a data management of iOS application. It is similar to a database for a web application. However, the Core Data is unlike a typical database. Core Data can be used totally in-memory. Apple designs a Core Data framework used only for Mac and iOS application.

In Core Data, three entities are applied: Food, Place, and Cart to describe working of this application.

4.1.1 Food Table

Food table will store the information of food like image, image_url, ingredient, name, price, and unique.

Attribute	Type
T image	Transformable
S image_url	String
S ingredient	String
S name	String
N price	Decimal
N unique	Integer 16

Figure 30. Food table details description.

The following table shows the structure of the Food table.

Table 3. Food table's structure.

Field	Description	Data Type
unique	an unique number for the food item	integer
price	a price of the food	decimal
name	name of the food item	string
ingredient	an ingredient of the food item	string
image_url	an image url of the food item	string
image	an image of the food item	transformable

4.1.2 Place Table

Place table will store the information of place like address, distance, email, image_url, lat,log, name, opening_time_1, opening_time_2, opening_time_3, opening_time_4, and phone_number.

Attribute	Type
S address	String
N distance	Float
S email	String
S image_url	String
N lat	Decimal
N log	Decimal
S name	String
S opening_time_1	String
S opening_time_2	String
S opening_time_3	String
S opening_time_4	String
S phone_number	String

Figure 31. Place table details description.

The following table shows the structure of the Place table.

Table 4. Place table's structure.

Field	Description	Data Type
address	an address of the place	string
distance	a distance from place's location to user location	float
email	an email of the place	string
image_url	an image url of the place	string
lat	a latitude of the place	decimal
log	a longitude of the place	decimal
name	a name of the place	string
opening_time_1	an opening time 1 of the place	string
opening_time_2	an opening time 2 of the place	string
opening_time_3	an opening time 3 of the place	string
opening_time_4	an opening time 4 of the place	string
phone_number	a phone number of the place	string

4.1.3 Cart Table

Cart table will store the information of cart like count, created_at, unique.

Attribute	Type
N count	Integer 16
D created_at	Date
N unique	Integer 16

Figure 32. Cart table details description.

The following table shows the structure of the Cart table.

Table 5. Cart table's structure.

Field	Description	Data Type
unique	an unique number for the cart item	integer
created_at	an time created for the cart item	date
count	an count for the cart item	integer

4.1.4 Entity Relationship

This diagram described a relationship between three entities: Place, Food and Cart.

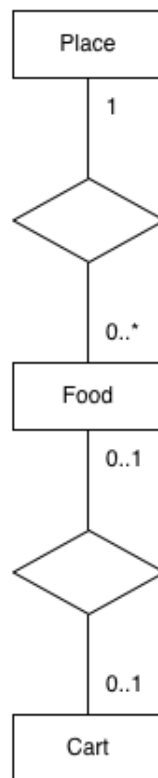


Figure 33. Entities relationship.

Almost relationship in the core data should be a bi-directional (apple developer requirement). Xcode IDE will give a warning about this problem if the application doesn't meet the apple developer requirement.

As being seen on the diagram, the first relationship from food to place is one to one and the opposite is one to many. The second relationship from cart to food is zero to one and the opposite is also zero to one.

The database implementation and design is the most important part in developing iOS application. Thank to apple that created core data framework.

4.2 Design of different parts of GUI

GUI in this application is focused on Storyboard. Storyboard is a new technology to design the user interface of iOS application. It's only available from iOS 5 and later. Previous iOS 5 can't use this technology, that's why this application isn't compatible to iOS 4 or iOS 3. Some advantages of using storyboard are:

- It's a container for all view windows screen (View Controllers, Navigation Controllers, TabBar Controllers).
- It uses the "segue" (the new technology in iOS 5) to manage the connection and transition between these view windows screen.
- Managing the controller to talk to each other.
- It designs a well flow of iOS application.

The initial view controller is tab bar controller. Tab bar controller is applied for this iOS application for the purpose of storing five view controllers in it.

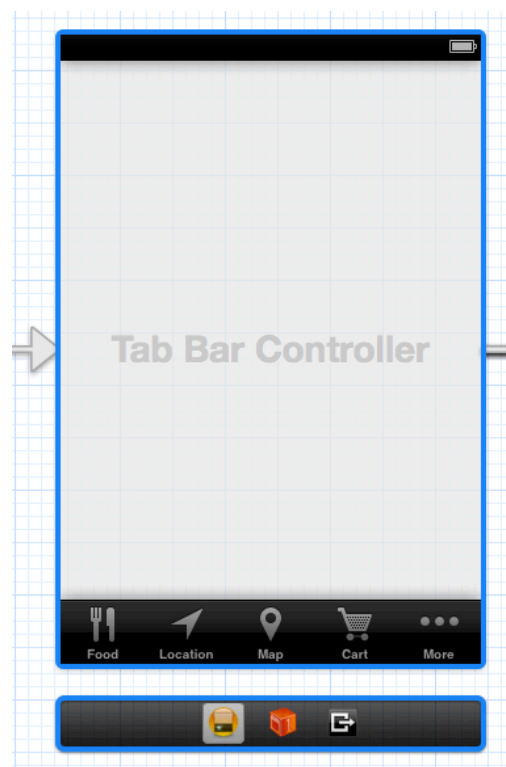


Figure 34. Tab bar controller.

Below, more details about the UI view window screens that are created and designed by Storyboard technology will be described.

4.2.1 Food view window

This is the first view window in the tab bar controller. When the user enters iOS application, this view window is initialized first. In fact, you can specify any view window initialized from the tab bar controller.

In this view, the iOS application has two child view windows: view window one for list of foods, and view window two for details of food. The iOS application needs to use navigation controller in order to connect two view windows.

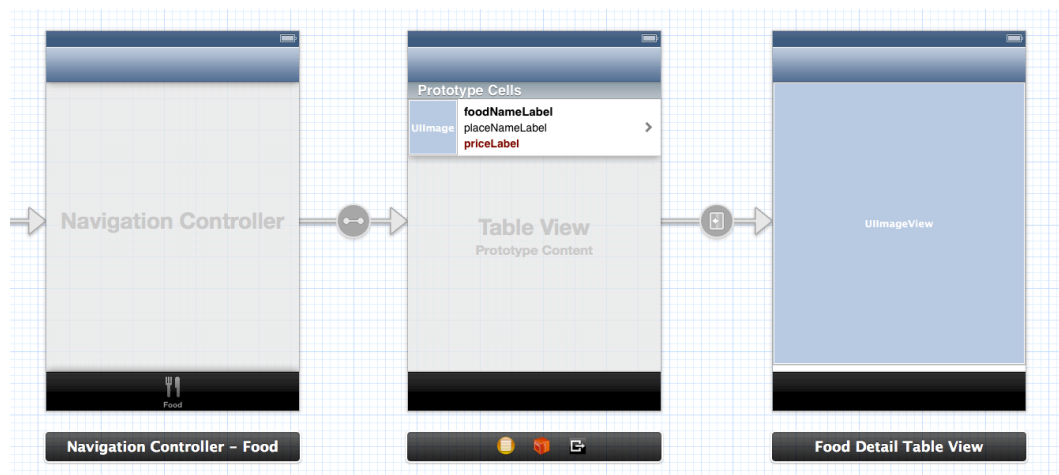


Figure 35. Food view window.

The first view window is for a list of foods, designs by table view controller. The table view controller is a view controller plus a table. So it's so convenient for list of foods. The table view controller has many attributes (see pictures below):

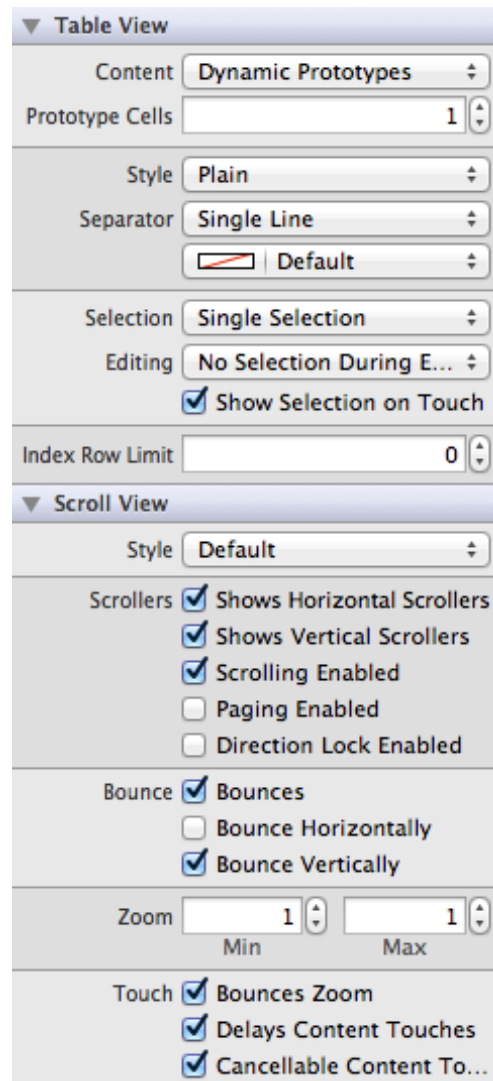


Figure 36. Table view attributes.

In the attribute panel of table view, the chosen content is dynamic prototypes. Custom table view cell is created because the original one doesn't fit for this iOS application. The attribute panel of table view cell is listed in this below picture:

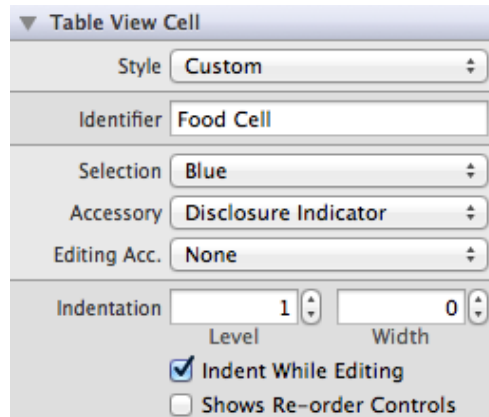


Figure 37. Table view cell attributes.

In the attribute panel of table view cell, a style Custom is chosen for custom table view cell. Also, identifier Food Cell is utilized to identify the unique table view cell. In a custom table view cell, the UI is designed as being seen in the following picture:

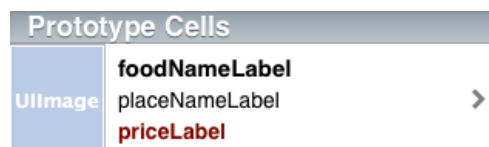


Figure 38. Custom table view cell.

- UIImage will show the image of food.
- foodNameLabel is a name of food.
- placeNameLabel is a name place of food.
- priceLabel is a price of food.

The final view is:

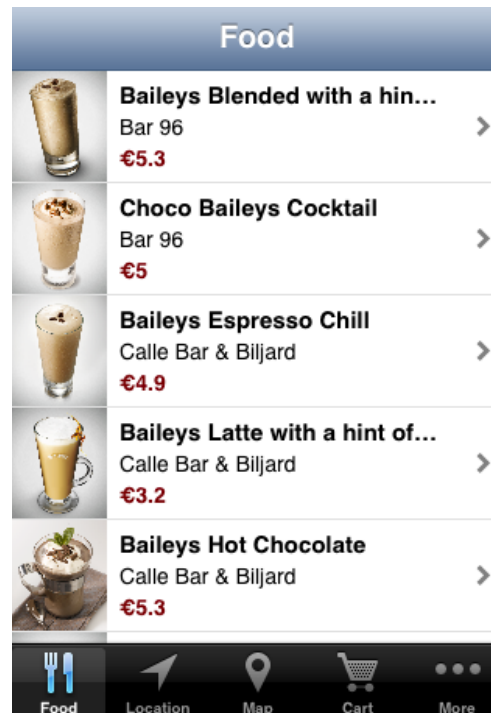


Figure 39. Food table view.

The second view window is for details of food, designs by table view controller. The table view controller has many attributes (see pictures below):

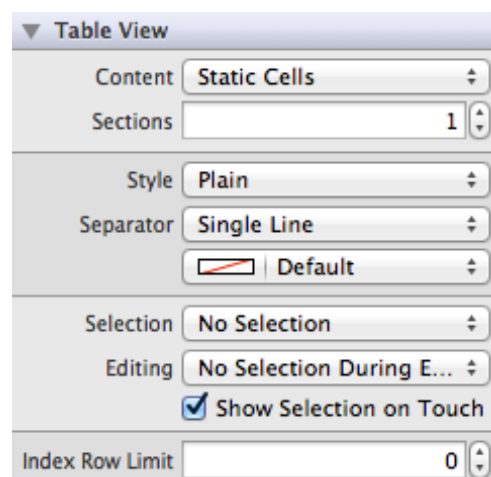


Figure 40. Table view attributes.

In the attribute panel of table view, the content is static cells. The custom table view can be seen as followed:

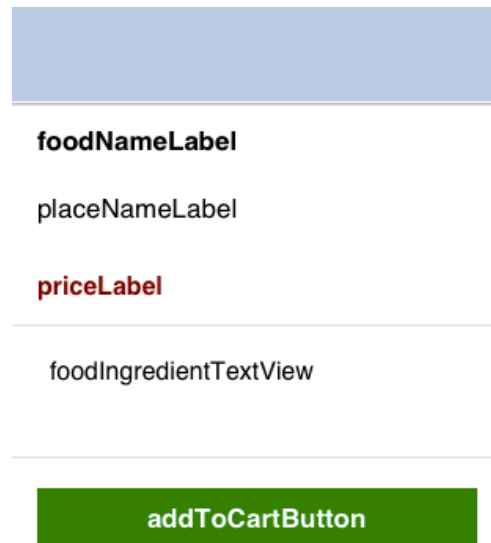


Figure 41. Detail food table view.

- `foodNameLabel` is a name of food.
- `placeNameLabel` is a name of place of food.
- `priceLabel` is a price of food.
- `foodIngredientTextView` is a ingredient of food.
- `addToCartButton` is a Add to Cart button.

The final view is:

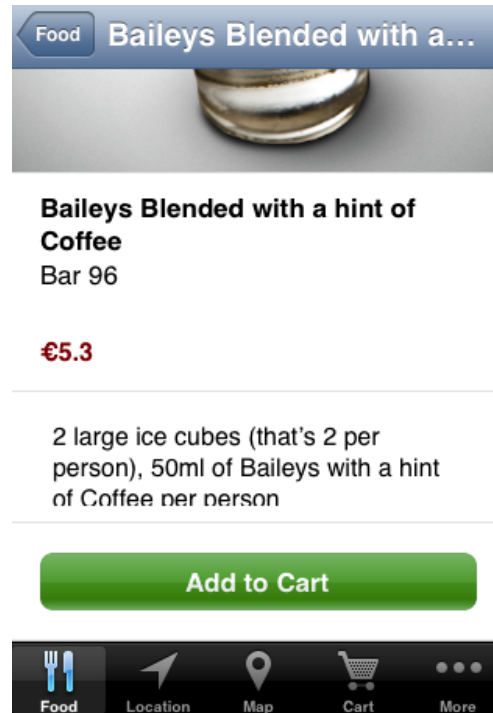


Figure 42. Detail food table view.

4.2.2 Location view window

This is the second view window in the tab bar controller. In this view, as the first view window, the location view will have two child view windows: view window one for list of places included the distance from the users location to location of places, and view window two for detail of place. Also, the application needs to use navigation controller in order to connect two view windows.

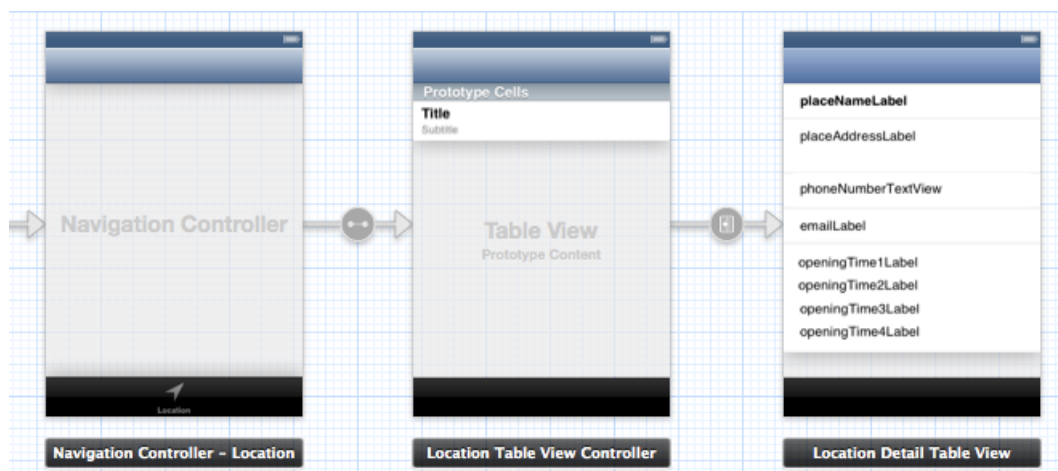


Figure 43. Location view window.

The first view window is for a list of places, designs by table view controller. In this view window, the attribute content of dynamic prototypes is used the same as the first view window for a list of foods. However, in this view window, there is no need for creating a custom table view cell because the original one fits the requirement. The attribute of table view cell is shown in this below picture:

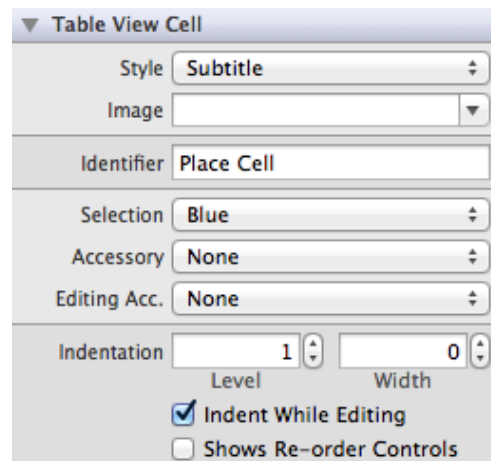


Figure 44. Table view cell attributes.

In the table view cell's attribute panel, a style Subtitle (original table view cell) is chosen. Besides, the identifier Place Cell is utilized to identity the unique table view cell. The UI of original table view cell like this:

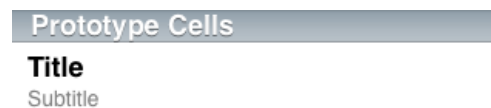


Figure 45. Original table view cell.

- Title is a name of place.
- Subtitle is a distance from location of place to the user's location.

The final view is:

Location	
D.O.M Munkhaus 8339.5 km	>
Oliver's Inn 8339.6 km	>
Calle Bar & Biljard 8339.6 km	>
Office - The Sports Bar 8339.7 km	>
El Gringo Music Saloon 8339.7 km	>
O'Malley's 8339.7 km	>
Bar 96 8339.9 km	>




Figure 46. Location table view.

The second view window is for detail of place, designs by table view controller. The chosen attribute content is static cells the same as previous view window for a detail of food. The UI view window can be seen as following:

placeNameLabel
placeAddressLabel
phoneNumberTextView
emailLabel
openingTime1Label
openingTime2Label
openingTime3Label
openingTime4Label

Figure 47. Detail location table view.

- placeNameLabel is a name of place.

- placeAddressLabel is an address of place.
- phoneNumberTextView is a phone number of place.
- emailLabel is a email of place.
- openingTime1Label is an opening time 1 of place.
- openingTime2Label is an opening time 2 of place.
- openingTime3Label is an opening time 3 of place.
- openingTime4Label is an opening time 4 of place.

The final view is:



Figure 48. Detail location table view.

4.2.3 Map view window

This is the third view window in the tab bar controller. This view is different than two previous view windows. It's embedded the MKMapView and list of the location of places on the map. Also, the user's location is shown on the map. Thus, he can specify the way from the user's location to the location of places.



Figure 49. Map view controller.

The upper picture shows the UI of map view controller. This view has three buttons on the top of the view window. These buttons has a function to switch view map on the map controller. A view map can be switched to standard (default, it's just a map), satellite (the map view is shown like the real) or hybrid (standard plus satellite). Furthermore, this view has one small button at the bottom left of the view. It has function to specify the user's location. When the user press it, it will use GPS signal to specify his location (location of phone).

Apple dropped the Google map on the iOS 6 and the replacement is Apple's map.

The Apple's map has more advantages than the Google map:

- Owned by Apple.
- Siri integration.
- Clearer view from the map
- New 3D map.

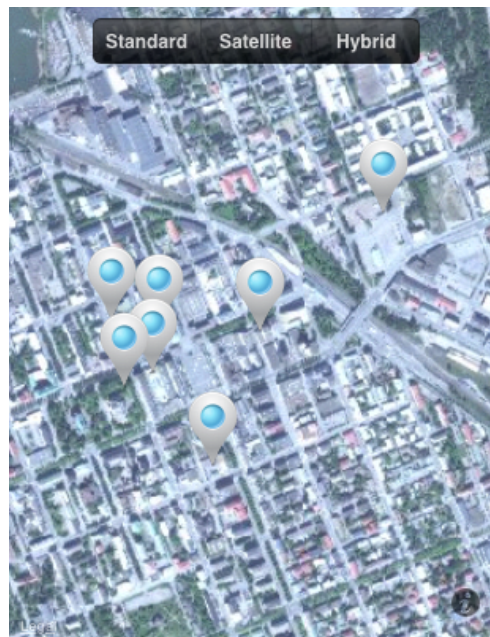


Figure 50. Satellite map view.

In the satellite map view, the map will be shown like the real. The user can choose the Satellite button to enter to this view.

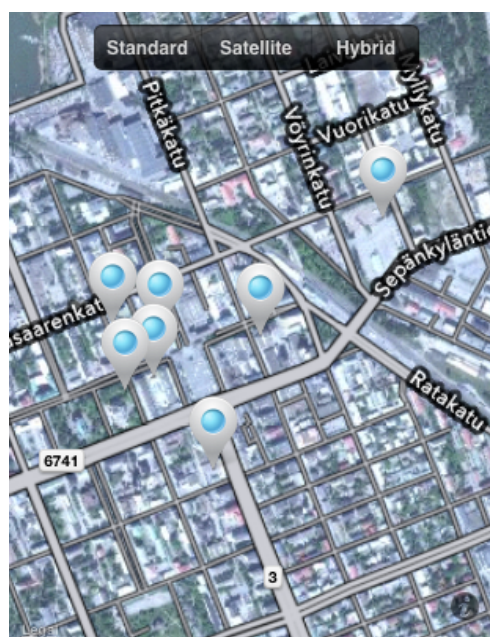


Figure 51. Hybrid map view.

In the hybrid map view, the map not only will be shown like the real but also added the name of the street. The user can choose the Hybrid button to enter to this view.

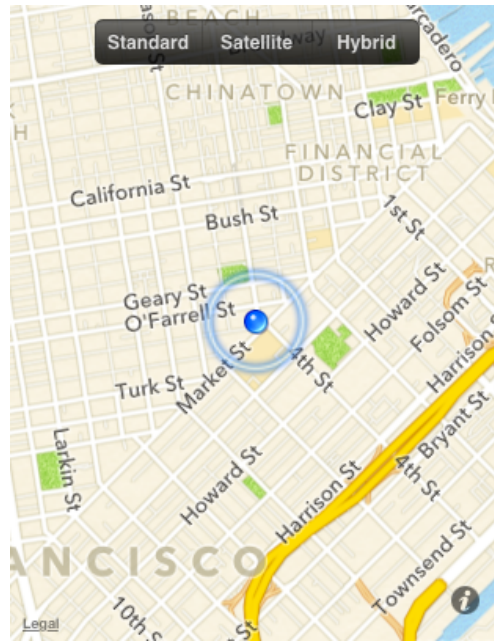


Figure 52. Users location.

When the user chooses the show user's location button, the blue point will be shown on the map.

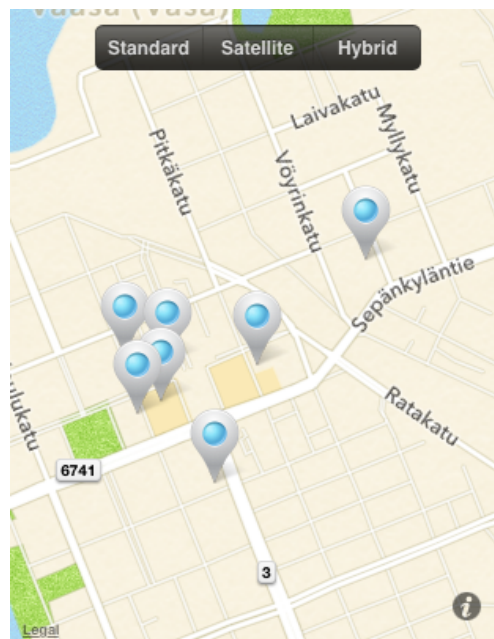


Figure 53. Location of places.

In addition, when user chooses any location of place pin, the call out accessory view will be shown with the name and the address of that place. The UI of this view like this:



Figure 54. Call out accessory view.

4.2.4 Cart view window

This is the fourth view window in the tab bar controller. In fact, this view window uses table view for list of the cart item. In addition, two buttons on the top of the view window make this view more functionality. In order to add two buttons on the top of the view window, there are two ways: (1) using a toolbar and the table view plus table view cell into view controller, (2) using navigation controller. The second way is taken because it's easier and more flexible.

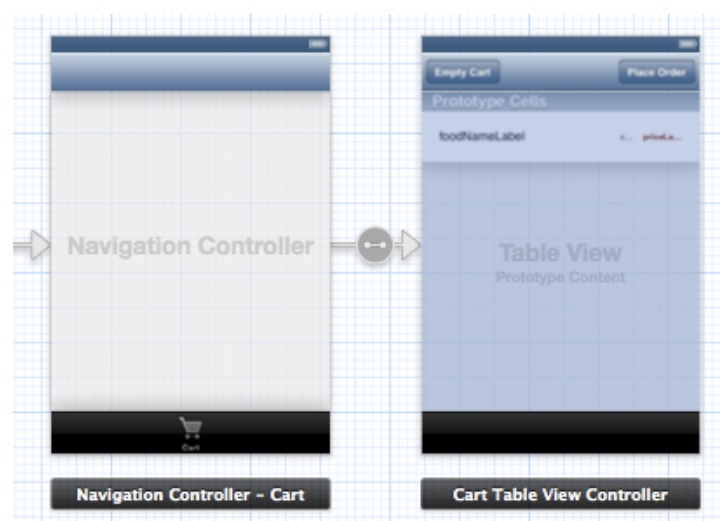


Figure 55. Cart view window.

As usually, this table view cell uses attribute content, dynamic prototypes. Besides, this table view cell is a custom table view cell as the original can't satisfy the requirement. The attribute of table view cell are listed in the below picture:

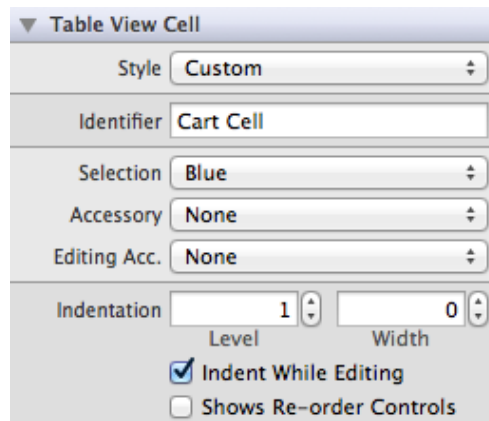


Figure 56. Attribute of table view cell.

In the attribute panel of the table view cell, style Custom is chosen for custom table view cell. Also, identifier Cart Cell is applied to specify the unique table view cell. In a custom table view cell, the UI is designed as the below picture:



Figure 57. Custom table view cell.

- foodNameLabel is a name of food.
- countLabel is a count of food.
- priceLabel is a price of food.

Also, the UI two buttons on the top of the cart view window:



Figure 58. Bar button item.

- Empty Cart is an empty the cart button item.
- Place Order is a place order button item.

The final view is:

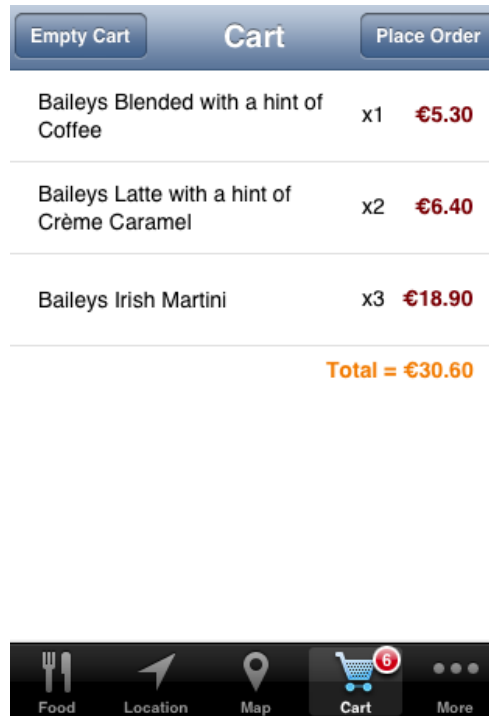


Figure 59. Cart table view.

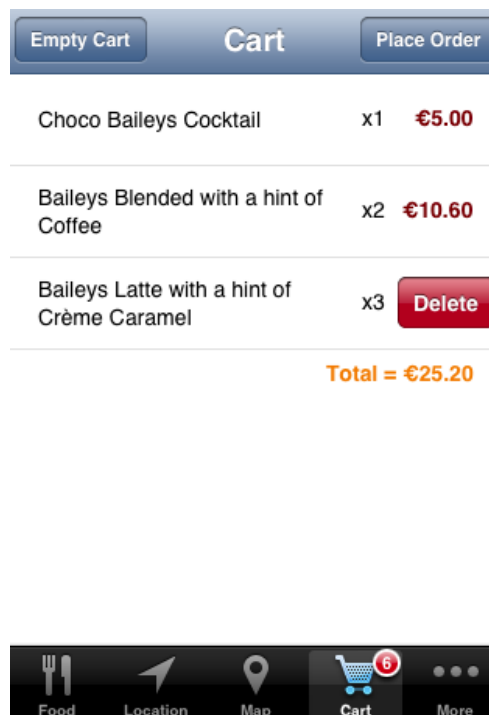


Figure 60. Delete button.

4.2.5 More view window

This is a last view window in the tab bar controller. This view window has five child view windows. The first view window uses a table view controller for list of name of child view window. The rest of view window is a child of the first view window. Furthermore, this view window uses navigation controller to connect the rest of view window to the first view window. All of view windows are shown in below picture:

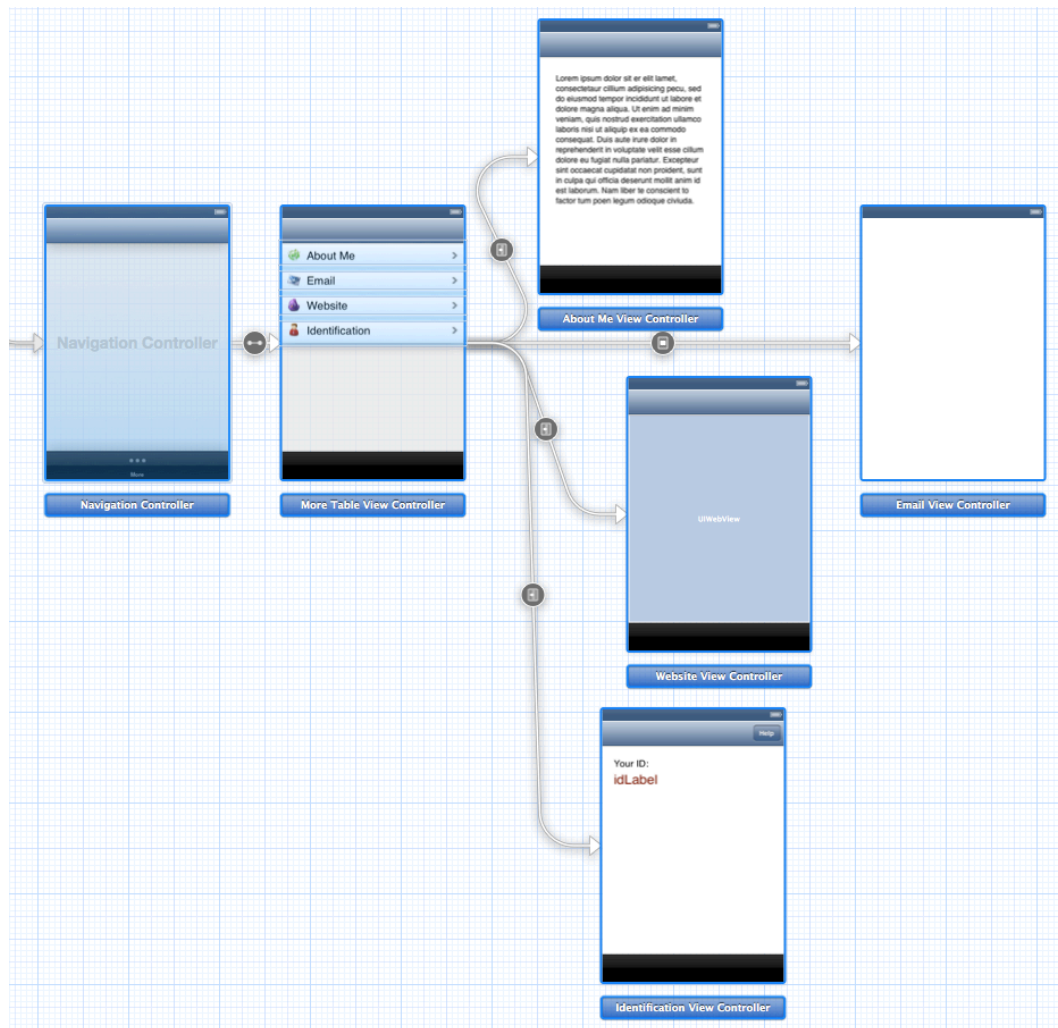


Figure 61. More views and child views.

The first view window for a list of name of child view window is a table view controller. The attribute content of this view window is static cells. The UI of this view window likes below:

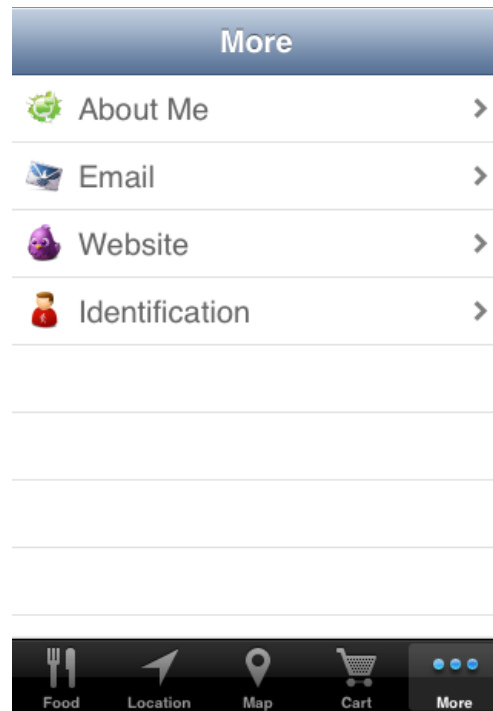


Figure 62. More view.

- About Me is a name of view window about me.
- Email is a name of view window email.
- Website is a name of view window website.
- Identification is a name of view window identification.

The final view is:

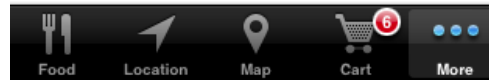
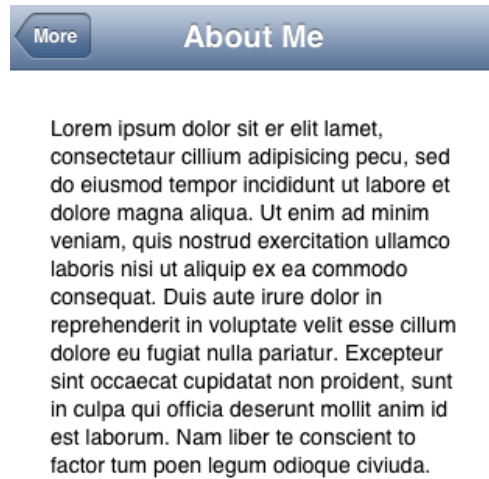


Figure 63. About Me view.

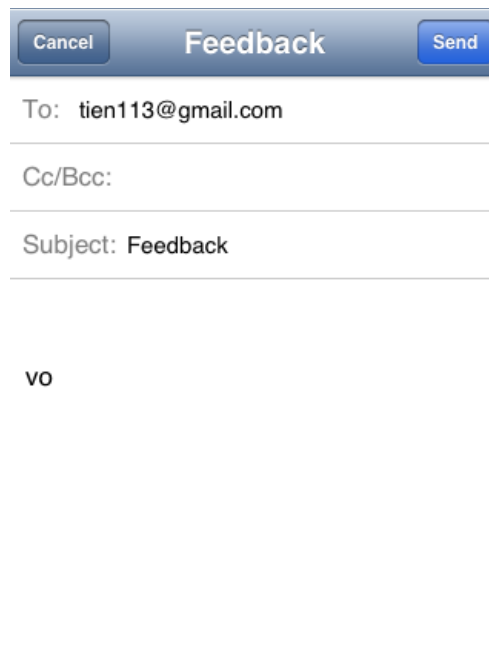


Figure 64. Email view.



Figure 65. Website view.

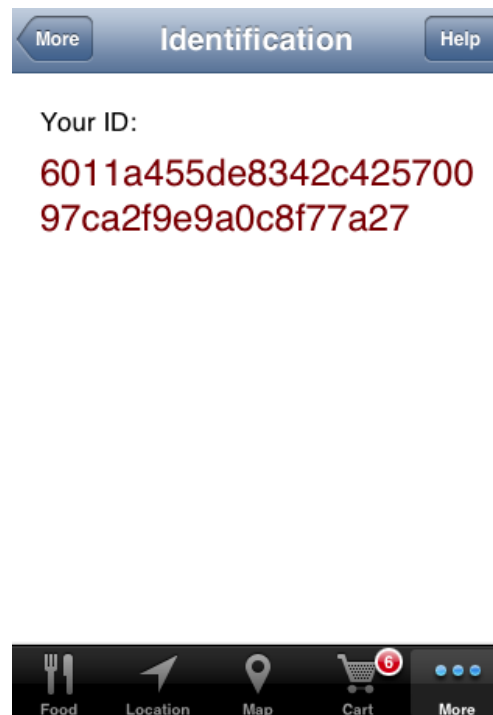


Figure 66. Identification view.

5 IMPLEMENTATION

Every UI view window needs at least one class to support them. The application has so many view windows that a large amount of classes needs to be created in order to support them. All of classes are divided into ten groups:

- UIAlertViewE: this group stores category classes of UIAlertView.
- NSDateE: this group stores category classes of NSDate.
- Cryptography: this group stores cryptography classes.
- JSONE: this group stores category classes of NSData and NSDictionary.
- Helpers: this group stores many helper classes.
- View Controllers: this group stores a view controller classes that support for the view window on storyboard.
- Web Services: this group stores a connection class to the web service.
- Core Data: this group stores core data classes.
- Images: this group stores a lot of images for the UI.
- Supporting Files: this group stores appDelegate classes.

Only important operations and their details are listed above due to huge amount of classes.

5.1 Get the data from the web service:

There are important operations connecting to the web service and getting the data from there to the iOS.

Below is a code for getting the data of foods from the web service:

```
+ (NSArray *)getFoods {
    // init request with url
    NSURLRequest *request = [NSURLRequest
requestWithURL:kFoodPlaceFoodsURL];

    // get data from request
    NSError *error = nil;
    NSURLResponse *response = nil;
    NSData *responseData = [NSURLConnection sendSynchronousRequest:request
returningResponse:&response
error:&error];
}
```

```

// if error, show alert
if (error != nil) {
    dispatch_async(dispatch_get_main_queue(), ^{
        [UIAlertView showWithError:error];
    });
    NSLog(@"getFoods' Error: %@", error.localizedDescription);
    return nil;
}

// reading http status code
NSInteger responseCode = [Helpers
readHttpStatusCodeFromResponse:response];
NSLog(@"%d", responseCode);

// return nil if error happen
if (responseCode != kHTTPRequestOK || responseData.length == 0) {
    NSLog(@"Error!!!");
    return nil;
}

// get foods from JSON
// id foods = responseData.fromJSON;
NSLog(@"Getting foods...");
NSLog(@"%d", responseData.length);
return responseData ? (id)responseData.fromJSON : nil;
}

```

Snippet 1. Get food data from the web service.

This is a code for get the data places from the web service:

```

+ (NSArray *)getPlaces {

    // init request with url
    NSURLRequest *request = [NSURLRequest
requestWithURL:kFoodPlacePlacesURL];

    // get data from request
    NSError *error = nil;
    NSURLResponse *response = nil;
    NSData *responseData = [NSURLConnection sendSynchronousRequest:request
returningResponse:&response
error:&error];

    // if error, show alert
    if (error != nil) {
        dispatch_async(dispatch_get_main_queue(), ^{
            [UIAlertView showWithError:error];
        });
        NSLog(@"getPlaces' Error: %@", error.localizedDescription);
        return nil;
    }

    // reading http status code
    NSInteger responseCode = [Helpers
readHttpStatusCodeFromResponse:response];
    NSLog(@"%d", [Helpers readHttpStatusCodeFromResponse:response]);
}

```

```

// return nil if error happen
if (responseCode != kHTTPRequestOK || responseData.length == 0) {
    NSLog(@"Error!!!");
    return nil;
}

// get places from JSON
// id places = responseData.fromJSON;
NSLog(@"Getting places...");
NSLog(@"%d", responseData.length);
return responseData ? (id)responseData.fromJSON : nil;
}

```

Snippet 2. Get place data from the web service.

Two important operations are totally the same. It uses `NSURLRequest` to make a request with URL of the web service. After that, `NSURLConnection` is used to send a synchronous request to the web service and receiving the data to the phone. This operation also checks the error occurs if it happens. Right after receiving the data from the web service, the last line of operation will convert the received data to JSON or return to nil if there is an error.

This following code makes a request, sending request to the web service and return the data to the phone:

```

// init request with url
NSURLRequest *request = [NSURLRequest
requestWithURL:kFoodPlacePlacesURL];

// get data from request
NSError *error = nil;
NSURLResponse *response = nil;
NSData *responseData = [NSURLConnection sendSynchronousRequest:request
returningResponse:&response
error:&error];

```

Snippet 3. Send the request with URL and receive the data.

The code getting response code from the web service will look like:

```

// reading http status code
NSUInteger responseCode = [Helpers
readHttpStatusCodeFromResponse:response];
NSLog(@"%d", [Helpers readHttpStatusCodeFromResponse:response]);

```

Snippet 4. Get http status code.

In case there is an error, the code will show the alert as following:

```
// if error, show alert
if (error != nil) {
    dispatch_async(dispatch_get_main_queue(), ^{
        [UIAlertView showWithError:error];
    });
    NSLog(@"getPlaces' Error: %@", error.localizedDescription);
    return nil;
}
```

Snippet 5. Show the alert.

This is a code showing return data to JSON or nil if error happens:

```
return responseData ? (id)responseData.fromJSON : nil;
```

Snippet 6. Return JSON or nil.

It will return to nil if an error happens. Otherwise, it will return to id.

5.2 Initialization for the document and location manager:

There are two important operations to initialize the document and location manager. Because the application uses the singleton method, two operations are located in the AppDelegate. When the application needs to initialize for the document or location manager, it easily calls it from the AppDelegate.

Some important line of codes:

```
// singleton init document
+ (UIManagedDocument *)sharedDocument {

    static UIManagedDocument *_document = nil;
    static dispatch_once_t pred;
    dispatch_once(&pred, ^{
        NSURL *url = [[[NSFileManager defaultManager]
        URLsForDirectory:NSDocumentDirectory
                        inDomains:NSUserDomainMask] lastObject];
        url = [url URLByAppendingPathComponent:@"Default Food Place Database"]; //
set the document's name
        *_document = [[UIManagedDocument alloc] initWithFileURL:url]; // init document
    });
    return *_document;
}

// singleton init location manager
+ (CLLocationManager *)sharedLocationManager {

    static CLLocationManager *_locationManager = nil;
    static dispatch_once_t pred;
    dispatch_once(&pred, ^{
```

```

        _locationManager = [[CLLocationManager alloc] init]; // init location manager
        _locationManager.desiredAccuracy = kCLLocationAccuracyBest; // set accuracy
for the GPS
        _locationManager.distanceFilter = 80.0f;
    });
    return _locationManager;
}

```

Snippet 7. Setup singleton document and location manager.

Location manager is initialized by this code:

```

_locationManager = [[CLLocationManager alloc] init]; // init location manager

```

Snippet 8. Initialize location manager.

Location manager has two attributes desired accuracy and distance filter. It is specified by two lines of code:

```

_locationManager.desiredAccuracy = kCLLocationAccuracyBest; // set accuracy for
the GPS
_locationManager.distanceFilter = 80.0f;

```

Snippet 9. Attribute of location manager.

The application uses a lot of document and the location manager. However, the application can't initialize many documents and location managers when the iOS application needed. Thus, the singleton method is the best choice. The singleton will initialize only one time and the application can use many times after that.

5.3 Adding the data from the web service to the core data:

This is the most important operation in this iOS application. Since the data is rarely changed, the data from the web service should be copied to the core data in the iOS application. If the data isn't in the phone, every time the application starts, it will connect to the web service and retrieving the data. Obviously, it takes a lot of time and bandwidth.

This is a code for adding the data from the web service to the core data:

```

// setup fetched result controller
- (void)setupFetchedResultsController {

    // fetch request with entity
    NSFetchRequest *request = [NSFetchRequest
fetchRequestWithEntityName:@"Food"];

```



```

    // [request setFetchBatchSize:5];
    // [request setFetchLimit:7];
    // sort by name or place's name (place.name)
    request.sortDescriptors = @[ [NSSortDescriptor
    sortDescriptorWithKey:@"place.name"
                                ascending:YES

selector:@selector(localizedCaseInsensitiveCompare:)] ];
    self.fetchedResultsController = [[NSFetchedResultsController alloc]
initWithFetchRequest:request

managedObjectContext:self.document.managedObjectContext
                                sectionNameKeyPath:nil
                                cacheName:nil];
}

// fetch data from Web Service into Core Data
- (void)fetchWebServiceIntoDatabase:(UIManagedDocument *)document {

    // create queue for Food Fetcher
    dispatch_queue_t fetchQ = dispatch_queue_create("Food Place Fetcher", NULL);
    dispatch_async(fetchQ, ^{
        NSArray *foods = [FoodPlaceFetcher getFoods]; // get foods from Web Service
to NSArray
        [document.managedObjectContext performBlock:^(
            [foods enumerateObjectsUsingBlock:^(NSDictionary *food, NSUInteger idx,
            BOOL *stop) {
                [Food foodWithWebService:food
                inManagedObjectContext:document.managedObjectContext]; // add foods to Core
Data
            }]);
        [document saveToURL:document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL]; // save
to document
        });
    dispatch_release(fetchQ); // release fetchQ
}

// use document
- (void)useDocument {

    // check document exists or not
    if (![NSFileManager defaultManager] fileExistsAtPath:[self.document.fileURL
path]) {
        [self.document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL success)
{
            [self setupFetchedResultsController]; // fetch data
            [self fetchWebServiceIntoDatabase:self.document]; // fetch data to Document
        }];
        // if document state is closed, open and using it
    } else if (self.document.documentState == UIDocumentStateClosed) {
        [self.document openWithCompletionHandler:^(BOOL success) {
            [self setupFetchedResultsController];
            [self badgeValueUpdate]; // set badge value
        }];
        // if document state is normal, use it
    }
}

```

```

    } else if (self.document.documentState == UIDocumentStateNormal) {
        [self setupFetchedResultsController];
        [self badgeValueUpdate]; // set badge value
    }
}

- (void)setDocument:(UIManagedDocument *)document {

    if (_document != document) {
        _document = document;
        [self useDocument]; // use document
    }
}

```

Snippet 10. Add the data from the web service to the client application.

This is a code for using the document:

```

// use document
- (void)useDocument {

    // check document exists or not
    if (![NSFileManager defaultManager] fileExistsAtPath:[self.document.fileURL
path]]) {
        [self.document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL success)
{
            [self setupFetchedResultsController]; // fetch data
            [self fetchWebServiceIntoDatabase:self.document]; // fetch data to Document
        }];
        // if document state is closed, open and using it
    } else if (self.document.documentState == UIDocumentStateClosed) {
        [self.document openWithCompletionHandler:^(BOOL success) {
            [self setupFetchedResultsController];
            [self badgeValueUpdate]; // set badge value
        }];
        // if document state is normal, use it
    } else if (self.document.documentState == UIDocumentStateNormal) {
        [self setupFetchedResultsController];
        [self badgeValueUpdate]; // set badge value
    }
}
}

```

Snippet 11. Using document.

This operation will check the document existing or not. If the document does not exist, it will create the document. After that, the application starts setup the fetched result controller and fetch the data from the web service in to the core data.

```

// check document exists or not
if (![NSFileManager defaultManager] fileExistsAtPath:[self.document.fileURL
path]]) {

```

```

        [self.document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL success)
{
    [self setupFetchedResultsController]; // fetch data
    [self fetchWebServiceIntoDatabase:self.document]; // fetch data to Document
    }];
    // if document state is closed, open and using it
}

```

Snippet 12. Check the document exists or not.

Otherwise, if the document exists, the operation will check the document is closed state or normal state. If it's closed state, the document will open it and start setup the fetched result controller.

```

else if (self.document.documentState == UIDocumentStateClosed) {
    [self.document openWithCompletionHandler:^(BOOL success) {
        [self setupFetchedResultsController];
        [self badgeValueUpdate]; // set badge value
    }];
    // if document state is normal, use it
}

```

Snippet 13. Check state of document is closed.

Or if it's normal state, the document starts to setup the fetched result controller.

```

else if (self.document.documentState == UIDocumentStateNormal) {
    [self setupFetchedResultsController];
    [self badgeValueUpdate]; // set badge value
}

```

Snippet 14. Check state of document is normal.

This is a code for setup the fetched result controller:

```

// setup fetched result controller
- (void)setupFetchedResultsController {

    // fetch request with entity
    NSFetchRequest *request = [NSFetchRequest
fetchRequestWithEntityName:@"Food"];
    // [request setFetchBatchSize:5];
    // [request setFetchLimit:7];
    // sort by name or place's name (place.name)
    request.sortDescriptors = @[ [NSSortDescriptor
sortDescriptorWithKey:@"place.name"
                                ascending:YES

selector:@selector(localizedCaseInsensitiveCompare:)] ];
    self.fetchedResultsController = [[NSFetchedResultsController alloc]
initWithFetchRequest:request

```

```

managedObjectContext:self.document.managedObjectContext
                                sectionNameKeyPath:nil
                                cacheName:nil];
}

```

Snippet 15. Setup fetched result controller.

This code above is setup result fetched controller. It will make a request with the Food entity with NSFetchedRequest.

```

// fetch request with entity
NSFetchedRequest *request = [NSFetchedRequest
fetchRequestWithEntityName:@"Food"];

```

Snippet 16. Fetch request with entity.

After that it uses NSSortDescriptor to sort the request with a key name of place, following ascending. Also, it will compare the case insensitive.

```

// sort by name or place's name (place.name)
request.sortDescriptors = @[ [NSSortDescriptor
sortDescriptorWithKey:@"place.name"
                                ascending:YES
                                selector:@selector(localizedCaseInsensitiveCompare:)] ];

```

Snippet 17. Sort name of places.

The last line is allocated the fetched result control with the request and managed object context.

```

self.fetchedResultsController = [[NSFetchedResultsController alloc]
initWithFetchRequest:request

managedObjectContext:self.document.managedObjectContext
                                sectionNameKeyPath:nil
                                cacheName:nil];

```

Snippet 18. Fetched result controller is allocated.

This is a code for fetching the data from web service into the core data:

```

// fetch data from Web Service into Core Data
- (void)fetchWebServiceIntoDatabase:(UIManagedDocument *)document {

// create queue for Food Fetcher
dispatch_queue_t fetchQ = dispatch_queue_create("Food Place Fetcher", NULL);
dispatch_async(fetchQ, ^{
    NSArray *foods = [FoodPlaceFetcher getFoods]; // get foods from Web Service
to NSArray
    [document.managedObjectContext performBlock:^(

```

```

        [foods enumerateObjectsUsingBlock:^(NSDictionary *food, NSUInteger idx,
        BOOL *stop) {
            [Food foodWithWebService:food
            inManagedObjectContext:document.managedObjectContext]; // add foods to Core
            Data
        }];
        [document saveToURL:document.fileURL
        forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL]; // save
        to document
    }];
    }];
    dispatch_release(fetchQ); // release fetchQ
}

```

Snippet 19. Fetch data from the web service to core data.

In this operation, the application used the Grand Central Dispatch (GCD). GCD is like the multithreading in the iOS application. The GCD will create the name of dispatch queue, starts by this line:

```

// create queue for Food Fetcher
dispatch_queue_t fetchQ = dispatch_queue_create("Food Place Fetcher", NULL);

```

Snippet 20. Create queue Food Place Fetcher.

And it ended with this line (it will release the dispatch queue from the memory):

```

dispatch_release(fetchQ); // release fetchQ

```

Snippet 21. Release queue.

The code in the body of the dispatch queue will run on the background, so it does not lock the main thread (the UI). The code will begin fetch the data from the web service and return it to the array foods.

```

NSArray *foods = [FoodPlaceFetcher getFoods]; // get foods from Web Service to
NSArray

```

Snippet 22. Get array of food.

And the managed object context in the document will perform the block:

```

[document.managedObjectContext performBlock:^(
);

```

Snippet 23. Document performs block.

The reason that the block in this dispatch queue is performed is the code in dispatched queue can't use directly the document (it's not in right thread). Hence, in order to use the document in this dispatch queue, it is obligatory to perform the block in dispatch queue.

In this block, the loop is run in the foods array to get the data and add it to the core data.

```
[foods enumerateObjectsUsingBlock:^(NSDictionary *food, NSUInteger idx, BOOL
*stop) {
    [Food foodWithWebService:food
inManagedObjectContext:document.managedObjectContext]; // add foods to Core
Data
}];
```

Snippet 24. Food is looping to add to core data.

In this iOS application, the enumeratedObjectsUsingBlock is utilized to run the loops in the foods array. It is a better method than for method, thanks for the Apple Developer. After receiving the data and adding it to the core data, the document will be saved in the core data.

```
[document saveToURL:document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL]; // save
to document
```

Snippet 25. Save the document.

The document will be saved in the document fileURL. It will use saving operation overwriting.

5.4 Lazy loading the images from the web service:

This is an important operation. It helps the application loading the images with lazy technology. It is not a good idea to save the images to the core data because many images will take times for loading. Loading the images from the web service with lazy technology is the best solution.

```
// start download image
- (void)startImageDownload:(Food *)food forIndexPath:(NSIndexPath *)indexPath {

    LazyImageDownloader *imageDownloader =
self.imageDownloadsInProgress[indexPath];
```

```

    if (nil == imageDownloader){
        imageDownloader = [[LazyImageDownloader alloc] init];
        imageDownloader.food = food;
        imageDownloader.indexPathInTableView = indexPath;
        imageDownloader.delegate = self;
        self.imageDownloadsInProgress[indexPath] = imageDownloader;
        [imageDownloader startDownload];
    }
}

// called by our LazyImageDownloader when an image is ready to be displayed
- (void)imageDidLoad:(NSIndexPath *)indexPath {

    LazyImageDownloader *imageDownloader =
self.imageDownloadsInProgress[indexPath];
    if (nil != imageDownloader)
    {
        FoodCell *cell = (FoodCell *)[self.tableView
cellForRowAtIndexPath:imageDownloader.indexPathInTableView];

        // display the newly loaded image
        cell.foodImageView.image = imageDownloader.food.image;
    }
}

// this method is used in case the user scrolled into a set of cells that dont have their
image yet
- (void)loadImagesForOnScreenRows
{
    NSUInteger count = [[self.fetchedResultsController fetchedObjects] count];
    // NSLog(@"%d", count);
    if (count > 0)
    {
        NSArray *visiblePaths = [self.tableView indexPathsForVisibleRows];
        [visiblePaths enumerateObjectsUsingBlock:^(NSIndexPath *indexPath,
NSUInteger idx, BOOL *stop) {
            Food *food = [self.fetchedResultsController objectAtIndex:indexPath];

            if (!food.image) // avoid the image download if the image has already
            {
                [self startImageDownload:food forIndexPath:indexPath];
            }
        }];
    }
}

#pragma mark - UIScrollViewDelegate

// load images for all onscreen rows when scrolling is finished
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
willDecelerate:(BOOL)decelerate
{
    if (!decelerate)
    {
        [self loadImagesForOnScreenRows];
    }
}

```

```
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    [self loadImageForOnScreenRows];
}
```

Snippet 26. Lazy loading images.

This is a code for loading the images at index path:

```
// called by our LazyImageDownloader when an image is ready to be displayed
- (void)imageDidLoad:(NSIndexPath *)indexPath {

    LazyImageDownloader *imageDownloader =
self.imageDownloadsInProgress[indexPath];
    if (nil != imageDownloader)
    {
        FoodCell *cell = (FoodCell *)[self.tableView
cellForRowAtIndexPath:imageDownloader.indexPathInTableView];

        // display the newly loaded image
        cell.foodImageView.image = imageDownloader.food.image;
    }
}
```

Snippet 27. Loading images at index path.

The above code uses lazy image downloader delegate to load the images from the web service. It uses index path in the table view to specify the properly image for the table view cell.

This is a code for loading the images on the screen rows:

```
// this method is used in case the user scrolled into a set of cells that dont have their
image yet
- (void)loadImagesForOnScreenRows
{
    NSUInteger count = [[self.fetchedResultsController fetchedObjects] count];
    // NSLog(@"%d", count);
    if (count > 0)
    {
        NSArray *visiblePaths = [self.tableView indexPathsForVisibleRows];
        [visiblePaths enumerateObjectsUsingBlock:^(NSIndexPath *indexPath,
        NSUInteger idx, BOOL *stop) {
            Food *food = [self.fetchedResultsController objectAtIndex:indexPath:indexPath];

            if (!food.image) // avoid the image download if the image has already
            {
                [self startImageDownload:food forIndexPath:indexPath];
            }
        }];
    }
}
```

Snippet 28. Load images on the screen rows.

This operation will load the images for the visible table view cells. It doesn't load the images from the invisible table view cells. In fact, this lazy technology loading images will help the application run more smoothly as well as save the bandwidth of the network connection. The lag won't be happened.

This is a code for scroll view delegate:

```
// load images for all onscreen rows when scrolling is finished
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
willDecelerate:(BOOL)decelerate
{
    if (!decelerate)
    {
        [self loadImagesForOnScreenRows];
    }
}

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    [self loadImagesForOnScreenRows];
}
```

Snippet 29. Scroll view delegate.

This code will detect the scroll view on screen. The cell only loads the images when it doesn't move. If the cell is moving, the images don't load. The cell will be blank until it doesn't move.

This is a code for starting downloading the images:

```
// start download image
- (void)startImageDownload:(Food *)food forIndexPath:(NSIndexPath *)indexPath {

    LazyImageDownloader *imageDownloader =
    self.imageDownloadsInProgress[indexPath];

    if (nil == imageDownloader){
        imageDownloader = [[LazyImageDownloader alloc] init];
        imageDownloader.food = food;
        imageDownloader.indexPathInTableView = indexPath;
        imageDownloader.delegate = self;
        self.imageDownloadsInProgress[indexPath] = imageDownloader;
        [imageDownloader startDownload];
    }
}
```

Snippet 30. Start download image.

This code allocated the lazy image downloader delegate. Paste the food object, the index path object to the delegate and start download the images from delegate.

5.5 Add a food item to Cart:

This operation will add the food item to the cart. The user will browse the food item in the list of foods. When the user like that food item, they will choose add to cart button to add the food item to the cart, ready for place an order.

```
// add food to Cart
- (void)addToCart:(UIManagedDocument *)document {

    // create queue for Add to Cart
    dispatch_queue_t addQ = dispatch_queue_create("Add to Cart", NULL);
    dispatch_async(addQ, ^{
        [document.managedObjectContext performBlock:^(
            [Cart cartWithFood:self.food
            inManagedObjectContext:document.managedObjectContext]; // add food to Cart
            [document saveToURL:self.document.fileURL
            forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL]; // save
            to document

            // badge value
            [self badgeValueUpdate];
        }]);
    });
}
```

```
    dispatch_release(addQ);
}
```

Snippet 31. Add food to cart.

This upper code uses another operation `cartWithFood` to add cart item to the cart entity with managed object context.

```
[Cart cartWithFood:self.food
inManagedObjectContext:document.managedObjectContext]; // add food to Cart
```

Snippet 32. Add food uses managed object context.

After that, the document will be saved previous operation by this code:

```
[document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL]; // save
to document
```

Snippet 33. Save the document.

If the document wasn't saved, the food item doesn't stay in the cart entity. The last line is updated the badge value on the tab bar item.

```
// badge value
[self badgeValueUpdate];
```

Snippet 34. Badge value updated.

5.6 Calculate the distance from the user's location to places location:

This operation will calculate the user's location to location of places. The unit is kilometer.

```
// distance from places
- (void)sortPlacesByDistanceFrom:(CLLocation *)location {

    // define placeLocation (block way)
    __block CLLocation *placeLocation;
    // get places to NSMutableArray
    NSArray *places = [self.fetchedResultsController fetchedObjects];
    [places enumerateObjectsUsingBlock:^(Place *place, NSUInteger idx, BOOL *stop)
    {
        // init place with latitude and longitude
        placeLocation = [[CLLocation alloc] initWithLatitude:[place.lat doubleValue]
                                                             longitude:[place.log doubleValue]];
        // calculate distance from places
        place.distance = @([placeLocation distanceFromLocation:location] / 1000);
    }];
}
```

Snippet 35. Calculate the distance from places.

This operation has the parameter is location, based on the location of places. The coordinate will use the GPS signal to specify the user's location. This first line of this operation creates the variable to hold the locations of places.

```
// define placeLocation (block way)
__block CLLocation *placeLocation;
```

Snippet 36. Block place location.

It uses the block way. The variable with `__block` before will enter into the block container. The places array is a variable to hold all the place objects in place entity.

```
// get places to NSMutableArray
NSArray *places = [self.fetchedResultsController fetchedObjects];
```

Snippet 37. Get array of place.

After getting all of the place objects in place entity, the application will run the loop through place object to get the location of places. The location of the place is latitude and longitude with type double.

```
[places enumerateObjectsUsingBlock:^(Place *place, NSUInteger idx, BOOL *stop)
{
    // init place with latitude and longitude
    placeLocation = [[CLLocation alloc] initWithLatitude:[place.lat doubleValue]
                                                         longitude:[place.log doubleValue]];
    // calculate distance from places
    place.distance = @([placeLocation distanceFromLocation:location] / 1000);
}];
```

Snippet 38. Location of place is with latitude and longitude.

Running the loop with the block way is easier than the traditional method. That's why the `enumerateObjectsUsingBlock` is always used for the loops. This operation will loops the places array and get the location from the latitude and the longitude.

```
// init place with latitude and longitude
placeLocation = [[CLLocation alloc] initWithLatitude:[place.lat doubleValue]
                                                         longitude:[place.log doubleValue]];
```

Snippet 39. Initialize location of place.

After that, it will calculate the distance from the user's location to the location of places by this operation `distanceFromLocation`:

```
// calculate distance from places
place.distance = @([placeLocation distanceFromLocation:location] / 1000);
```

Snippet 40. Calculate the distance places.

The divided 1000 is converted the unit to kilometer.

5.7 Add the annotation pins to the map view:

These operations will help adding the annotation pins to the map view. The map view will show the location of places like a pin. The user can click on the pin and the pin will show the information about this pin like a name and an address of place.

```
// return NSArray Annotation
- (NSArray *)mapAnnotations {

    NSMutableArray *annotations = [NSMutableArray
arrayWithCapacity:self.places.count];
    [self.places enumerateObjectsUsingBlock:^(NSDictionary *place, NSUInteger idx,
BOOL *stop) {
        [annotations addObject:[PlaceAnnotationView annotationForPlace:place]]; // add
annotation to places
    }];
    return annotations;
}

// view for annotation
- (MKAnnotationView *)mapView:(MKMapView *)mapView
viewForAnnotation:(id<MKAnnotation>)annotation
{
    // check annotation is MKUserLocation class
    if ([annotation isKindOfClass:[MKUserLocation class]]) {
        return nil; // return nil if its
    }

    // check annotation is PlaceAnnotation class
    if ([annotation isKindOfClass:[PlaceAnnotationView class]])
    {
        MKAnnotationView *aView = [mapView
dequeueReusableAnnotationViewWithIdentifier:MY_PIN]; // set name
MKAnnotationView
        if (!aView) {
            aView = [[MKAnnotationView alloc] initWithAnnotation:annotation
reuseIdentifier:MY_PIN];
            aView.canShowCallout = YES; // set call out
            aView.calloutOffset = CGPointMake(0, 0); // set position call out off set
            aView.image = [UIImage imageNamed:MY_PIN_IMAGE]; // set place's image
        }
        return aView;
    }
    return nil;
}

for pin
```

```

        aView.leftCalloutAccessoryView = [[UIImageView alloc]
initWithFrame:CGRectMake(0, 0, 30, 30)]; // set frame for pin's image
        [aView setSelected:YES]; // set selected
    }

    aView.annotation = annotation;
    [(UIImageView *)aView.leftCalloutAccessoryView setImage:nil];

    return aView;
}

return nil;
}

```

Snippet 41. Add annotation to the map.

This is a code to add an annotation to annotations array:

```

// return NSArray Annotation
- (NSArray *)mapAnnotations {

    NSMutableArray *annotations = [NSMutableArray
arrayWithCapacity:self.places.count];
    [self.places enumerateObjectsUsingBlock:^(NSDictionary *place, NSUInteger idx,
BOOL *stop) {
        [annotations addObject:[PlaceAnnotationView annotationForPlace:place]]; // add
annotation to places
    }];
    return annotations;
}

```

Snippet 42. Get array of annotation.

In this operation, the application uses NSMutableArray instead of NSArray because the application wants an array can modify. The different between NSMutableArray and NSArray is NSArray is unmodify array but NSMutableArray is a modify array. The first line is created NSMutableArray annotations:

```

NSMutableArray *annotations = [NSMutableArray
arrayWithCapacity:self.places.count];

```

Snippet 43. Get mutable array of annotation.

This array with capacity is a capacity of places. After that, we will run the loop to the places array (not the NSMutableArray places) to get the data from this array. And I add an annotation to annotations by this code:

```

[annotations addObject:[PlaceAnnotationView annotationForPlace:place]]; // add
annotation to places

```

Snippet 44. Add annotation to places.

The application uses another operation `annotationForPlace` get the information from places (the name, the latitude and the longitude of place). And it uses the operation `addObject` to add the annotation to the annotations array.

This is a code changing the attributes of the annotations:

```
// view for annotation
- (MKAnnotationView *)mapView:(MKMapView *)mapView
viewForAnnotation:(id<MKAnnotation>)annotation
{
    // check annotation is MKUserLocation class
    if ([annotation isKindOfClass:[MKUserLocation class]]) {
        return nil; // return nil if its
    }

    // check annotation is PlaceAnnotation class
    if ([annotation isKindOfClass:[PlaceAnnotationView class]])
    {
        MKAnnotationView *aView = [mapView
dequeueReusableAnnotationViewWithIdentifier:MY_PIN]; // set name
MKAnnotationView
        if (!aView) {
            aView = [[MKAnnotationView alloc] initWithAnnotation:annotation
reuseIdentifier:MY_PIN];
            aView.canShowCallout = YES; // set call out
            aView.calloutOffset = CGPointMake(0, 0); // set position call out off set
            aView.image = [UIImage imageNamed:MY_PIN_IMAGE]; // set place's image
for pin
            aView.leftCalloutAccessoryView = [[UIImageView alloc]
initWithFrame:CGRectMake(0, 0, 30, 30)]; // set frame for pin's image
            [aView setSelected:YES]; // set selected
        }

        aView.annotation = annotation;
        [(UIImageView *)aView.leftCalloutAccessoryView setImage:nil];

        return aView;
    }

    return nil;
}
```

Snippet 45. Attributes of annotation.

This upper code is a part of `MKMapViewDelegate`. It will get the annotation and display the annotation on the map view. The application has two kind of location pin: the user's location and the location of place. And the application uses this code checking what kind of location pin:

```
// check annotation is MKUserLocation class
if ([annotation isKindOfClass:[MKUserLocation class]]) {
    return nil; // return nil if its
}
}
```

Snippet 46. Check annotation is user's location.

This upper code will check the annotation is the user's location (MKUserLocation class). If it is, it will return to nil (the default user's location pin).

```
if ([annotation isKindOfClass:[PlaceAnnotationView class]])
{
    MKAnnotationView *aView = [mapView
dequeueReusableAnnotationViewWithIdentifier:MY_PIN]; // set name
MKAnnotationView
    if (!aView) {
        aView = [[MKAnnotationView alloc] initWithAnnotation:annotation
reuseIdentifier:MY_PIN];
        aView.canShowCallout = YES; // set call out
        aView.calloutOffset = CGPointMake(0, 0); // set position call out off set
        aView.image = [UIImage imageNamed:MY_PIN_IMAGE]; // set place's image
for pin
        aView.leftCalloutAccessoryView = [[UIImageView alloc]
initWithFrame:CGRectMake(0, 0, 30, 30)]; // set frame for pin's image
        [aView setSelected:YES]; // set selected
    }

    aView.annotation = annotation;
    [(UIImageView *)aView.leftCalloutAccessoryView setImage:nil];

    return aView;
}
}
```

Snippet 47. Check annotation is place annotation view.

This upper code will check the annotation is the location of place (PlaceAnnotationView class). If it is, it will create aView (MKAnnotationView) with the identifier MY_PIN:

```
MKAnnotationView *aView = [mapView
dequeueReusableAnnotationViewWithIdentifier:MY_PIN]; // set name
MKAnnotationView
```

Snippet 48. Annotation is with identifier.

After that, the aView is allocated by MKAnnotationView with the identifier MY_PIN. In addition, it specifies some attributes for the annotation pin.

```
aView = [[MKAnnotationView alloc] initWithAnnotation:annotation
reuseIdentifier:MY_PIN];
aView.canShowCallout = YES; // set call out
```



```

aView.calloutOffset = CGPointMake(0, 0); // set position call out off set
aView.image = [UIImage imageNamed:MY_PIN_IMAGE]; // set place's image
for pin
    aView.leftCalloutAccessoryView = [[UIImageView alloc]
initWithFrame:CGRectMake(0, 0, 30, 30)]; // set frame for pin's image
[aView setSelected:YES]; // set selected

```

Snippet 49. Set attribute of annotation.

Some attributes of the annotation pins like show call out, call out offset, image, left call out accessory view, selected. The last attribute of the annotation is set the image to nil. I don't want to setup any images in this operation now.

```

aView.annotation = annotation;
[[UIImageView *)aView.leftCalloutAccessoryView setImage:nil];

```

Snippet 50. Add image to annotation.

The last line is return to aView.

```

return aView;

```

Snippet 51. Return to annotation.

If the annotation is not kind of PlaceAnnotationView class, it will return to nil.

```

return nil;

```

Snippet 52. Return to nil.

5.8 Loading the images in the left call out accessory view:

This operation will load the images from the web service and add them to the left call out accessory view. When the user chooses on the pin, it will show the call out and the image from the left call out accessory view.

```

// image for annotation
- (UIImage *)image:(id)sender imageForAnnotation:(id <MKAnnotation>)annotation {
    PlaceAnnotationView *pav = (PlaceAnnotationView *)annotation;
    // get image's url
    NSURL *url = [FoodPlaceFetcher urlForPlace:pav.place];
    // get data from url
    NSData *data = [NSData dataWithContentsOfURL:url];
    // return image
    return data ? [UIImage imageWithData:data] : nil;
}

// load image when selecting
- (void)mapView:(MKMapView *)mapView

```

```

didSelectAnnotationView:(MKAnnotationView *)aView {

    UIImage *image = [self image:self imageForAnnotation:aView.annotation];
    // load annotation's image
    [(UIImageView *)aView.leftCalloutAccessoryView setImage:image];
}

```

Snippet 53. Load image for annotation.

This is a code for adding the images to the annotation:

```

// image for annotation
- (UIImage *)image:(id)sender imageForAnnotation:(id <MKAnnotation>)annotation {

    PlaceAnnotationView *pav = (PlaceAnnotationView *)annotation;
    // get image's url
    NSURL *url = [FoodPlaceFetcher urlForPlace:pav.place];
    // get data from url
    NSData *data = [NSData dataWithContentsOfURL:url];
    // return image
    return data ? [UIImage imageWithData:data] : nil;
}

```

Snippet 54. Get image for annotation.

It will get the URL from the place data.

```

// get image's url
NSURL *url = [FoodPlaceFetcher urlForPlace:pav.place];

```

Snippet 55. Get URL of image.

The operation dataWithContentsOfURL will get the data of image from the URL.

```

// get data from url
NSData *data = [NSData dataWithContentsOfURL:url];

```

Snippet 56. Get data from URL.

The last line is return to the image with the data or return to nil if error happens.

```

// return image
return data ? [UIImage imageWithData:data] : nil;

```

Snippet 57. Return image data.

This is a code for load the image when selecting the annotation pin:

```

// load image when selecting
- (void)mapView:(MKMapView *)mapView
didSelectAnnotationView:(MKAnnotationView *)aView {

```

```

UIImage *image = [self image:self imageForAnnotation:aView.annotation];
// load annotation's image
[[UIImageView *)aView.leftCalloutAccessoryView setImage:image];
}

```

Snippet 58. Load image when select the annotation.

This operation will load the image when the user touches the annotation pin. The first line uses the previous operation to get the image.

```

UIImage *image = [self image:self imageForAnnotation:aView.annotation];

```

Snippet 59. Load image for annotation.

After that, the last line will setup the image to the call out accessory view.

```

// load annotation's image
[[UIImageView *)aView.leftCalloutAccessoryView setImage:image];

```

Snippet 60. Setup the image to the call out accessory view.

5.9 Remove the cart item from the cart:

This operation will remove the cart item from the cart. When the user wants to delete the cart item, they just have touched the cart item and slide to the right. The delete button will be existed. Pressing the delete button will delete the cart item.

```

// remove from cart
- (void)removeFromCart:(UIManagedDocument *)document
atIndexPath:(NSIndexPath *)indexPath {

    dispatch_queue_t removeQ = dispatch_queue_create("Remove from Cart", NULL);
    dispatch_async(removeQ, ^{
        Cart *cart = [self.fetchedResultsController objectAtIndex:indexPath:indexPath];
        [document.managedObjectContext performBlock:^(
            [Cart removeFromCart:cart
            inManagedObjectContext:document.managedObjectContext];
            [document saveToURL:document.fileURL
            forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL];

            // outlet
            [self showTotalOrderLabelWhenRemove];
            [self showCartLabel];
            [self showPlaceOrderBarButtonItem];
            [self showEmptyCartBarButtonItem];

            // badge value
            [self badgeValueUpdate];
        }]);
    });
    dispatch_release(removeQ);
}

```

Snippet 61. Remove food item from cart.

This upper code will delete the cart item from the cart. The application will get the cart item in the carts array by this line:

```
Cart *cart = [self.fetchedResultsController objectAtIndex:indexPath:indexPath];
```

Snippet 62. Get cart from core data.

The cart item will use the index path to specify the cart item in the carts array. The operation removeFromCart will use the managed object context to remove the cart item.

```
[Cart removeFromCart:cart
inManagedObjectContext:document.managedObjectContext];
```

Snippet 63. Remove food item, using managed object context.

After removing the cart item, the application needs to save the document by this line.

```
[document saveToURL:document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL];
```

Snippet 64. Save the document.

Five last lines operation is setup the outlet on the view window screen.

```
// outlet
[self showTotalOrderLabelWhenRemove];
[self showCartLabel];
[self showPlaceOrderBarButtonItem];
[self showEmptyCartBarButtonItem];

// badge value
[self badgeValueUpdate];
```

Snippet 65. Check outlet on screen.**5.10 Empty the cart:**

This operation will empty the cart. When the user presses this button, the cart will be emptied.

```
// empty cart
- (void)emptyCart:(UIManagedDocument *)document {

    dispatch_queue_t emptyQ = dispatch_queue_create("Empty Cart", NULL);
```

```

dispatch_async(emptyQ, ^{
    NSArray *carts = [self fetchedCarts];
    [document.managedObjectContext performBlock:^(
        [carts enumerateObjectsUsingBlock:^(Cart *cart, NSUInteger idx, BOOL
*stop) {
            [Cart removeFromCart:cart
inManagedObjectContext:document.managedObjectContext];
        }];
    [document saveToURL:document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL];

    // outlet
    [self showCartLabel];
    [self showPlaceOrderBarButtonItem];
    [self showEmptyCartBarButtonItem];

    // badge value
    [self badgeValueUpdate];
    }];
});
dispatch_release(emptyQ);
}

```

Snippet 66. Empty the cart.

This above code will empty the cart. The application will get all of cart item and return them to carts array.

```

NSArray *carts = [self fetchedCarts];

```

Snippet 67. Get array of cart.

After that, the application will run loop through the carts array and remove cart item in the carts array.

```

[carts enumerateObjectsUsingBlock:^(Cart *cart, NSUInteger idx, BOOL
*stop) {
    [Cart removeFromCart:cart
inManagedObjectContext:document.managedObjectContext];
}];

```

Snippet 68. Loop the cart and remove food item.

After removing the cart item, the document should be saved by this line.

```

[document saveToURL:document.fileURL
forSaveOperation:UIDocumentSaveForOverwriting completionHandler:NULL];

```

Four last lines will setup the outlet in the view window.

```

// outlet
[self showCartLabel];

```

```

[self showPlaceOrderBarButtonItem];
[self showEmptyCartBarButtonItem];

// badge value
[self badgeValueUpdate];

```

Snippet 69. Check outlet on window.

5.11 Send the order:

This is an operation sending the order. In order to send the order to the web service, the application has to combine three operations: `sendOrder`, `prepareOrder:`, `startOrderUpload`.

```

- (void)sendOrder {

    dispatch_queue_t sendQ = dispatch_queue_create("Send Order", NULL);
    dispatch_async(sendQ, ^{
        NSArray *carts = [self fetchedCarts];
        [self performSelectorOnMainThread:@selector(prepareOrder:) withObject:carts
        waitUntilDone:YES];
    });
    dispatch_release(sendQ);
}

- (void)prepareOrder:(NSArray *)carts {

    NSString *orderId = [MacAddress getMacAddress].toSHA1; // get UUID
    NSString *orderTotal = [NSString stringWithFormat:@"%0.2f", [self totalOrder]];
    NSString *orderDate = [[NSDate date] toString];
    NSString *orderDone = FALSE_VALUE; // set order to FALSE

    __block NSMutableArray *orderDetailParents = [NSMutableArray array]; // init
    array
    __block NSMutableArray *keyOrderDetailParents = [NSMutableArray array];

    [carts enumerateObjectsUsingBlock:^(Cart *cart, NSUInteger idx, BOOL *stop) {
        NSString *foodName = cart.food.name;
        NSString *foodCount = [cart.count stringValue];
        NSString *foodPrice = [NSString stringWithFormat:@"%0.2f", [Helpers
        timeNSDecimalNumber:cart.food.price andNumber:cart.count]];
        NSString *foodPlace = cart.food.place.name;

        // NSArray *foodObjects = @[ foodName, foodCount, foodPrice, foodPlace ];
        // NSArray *foodKeys = @[ FOOD_NAME, FOOD_COUNT, FOOD_PRICE,
        FOOD_PLACE ];
        // NSDictionary *orderDetailChild = [NSDictionary
        dictionaryWithObjects:foodObjects forKeys:foodKeys];
        NSDictionary *orderDetailChild = @{ FOOD_NAME : foodName,
        FOOD_COUNT : foodCount,
        FOOD_PRICE : foodPrice,
        FOOD_PLACE : foodPlace };

        [orderDetailParents addObject:orderDetailChild]; // add order detail child to
        orderdetailparents
    }
}

```

```

        [keyOrderDetailParents addObject:[NSString stringWithFormat:@"%d", idx]]; //
add key orderdetailparent
    };

    // alloc order detail parent
    NSDictionary *orderDetailParent = [NSDictionary
dictionaryWithObjects:orderDetailParents forKeys:keyOrderDetailParents];

    // NSArray *orderObjects = @[ orderUuid, orderTotal, orderDate, orderDone,
orderDetailParent ];
    // NSArray *orderKeys = @[ ORDER_UUID, ORDER_TOTAL, ORDER_DATE,
ORDER_DONE, ORDER_DETAILS_ATTRIBUTES ];
    // NSDictionary *orderChild = [NSDictionary dictionaryWithObjects:orderObjects
forKeys:orderKeys];

    NSDictionary *orderChild = @{ ORDER_UUID : orderUuid,
ORDER_TOTAL : orderTotal,
ORDER_DATE : orderDate,
ORDER_DONE : orderDone,
ORDER_DETAILS_ATTRIBUTES : orderDetailParent };

    NSDictionary *orderParent = @{ ORDER : orderChild };
    // NSDictionary *orderParent = [NSDictionary dictionaryWithObjectsAndKeys:
orderChild, ORDER, nil];

    NSData *orderData = orderParent.toJSON; // convert nsdictionary to nsdata

    [self startOrderUpload:kFoodPlaceOrdersURL withData:orderData];
}

// uploader delegate
- (void)startOrderUpload:(NSURL *)url withData:(NSData *)data {

    if (nil != data) {
        OrderUploader *orderUploader = [[OrderUploader alloc] initWithURL:url
delegate:self
orderData:data];

        [orderUploader startUpload];
    }
}

```

Snippet 70. Send an order to the web service.

This code allows to starting to order upload:

```

// uploader delegate
- (void)startOrderUpload:(NSURL *)url withData:(NSData *)data {

    if (nil != data) {
        OrderUploader *orderUploader = [[OrderUploader alloc] initWithURL:url
delegate:self
orderData:data];

        [orderUploader startUpload];
    }
}

```

Snippet 71. Start order upload.

This code will use the delegate to start the upload data. It will allocate and initialize with the URL, the delegate is self and the order data is data.

```
OrderUploader *orderUploader = [[OrderUploader alloc] initWithURL:url
                                delegate:self
                                orderData:data];
```

Snippet 72. Initialize order upload.

After that, it will use startUpload operation to start uploading the data to the web service.

```
[orderUploader startUpload];
```

Snippet 73. Order uploader start to upload.

This is a code for preparing an order:

```
- (void)prepareOrder:(NSArray *)carts {

    NSString *orderId = [MacAddress getMacAddress].toSHA1; // get UUID
    NSString *orderTotal = [NSString stringWithFormat:@"%0.2f", [self totalOrder]];
    NSString *orderDate = [[NSDate date] toString];
    NSString *orderDone = FALSE_VALUE; // set order to FALSE

    NSMutableArray *orderDetailParents = [NSMutableArray array]; // init
    array
    NSMutableArray *keyOrderDetailParents = [NSMutableArray array];

    [carts enumerateObjectsUsingBlock:^(Cart *cart, NSUInteger idx, BOOL *stop) {
        NSString *foodName = [cart.food.name];
        NSString *foodCount = [cart.count stringValue];
        NSString *foodPrice = [NSString stringWithFormat:@"%0.2f", [Helpers
timeNSDecimalNumber:cart.food.price andNumber:cart.count]];
        NSString *foodPlace = [cart.food.place.name];

        // NSArray *foodObjects = @[ foodName, foodCount, foodPrice, foodPlace ];
        // NSArray *foodKeys = @[ FOOD_NAME, FOOD_COUNT, FOOD_PRICE,
FOOD_PLACE ];
        // NSDictionary *orderDetailChild = [NSDictionary
dictionaryWithObjects:foodObjects forKeys:foodKeys];
        NSDictionary *orderDetailChild = @{ FOOD_NAME : foodName,
                                           FOOD_COUNT : foodCount,
                                           FOOD_PRICE : foodPrice,
                                           FOOD_PLACE : foodPlace };

        [orderDetailParents addObject:orderDetailChild]; // add order detail child to
orderdetailparents
        [keyOrderDetailParents addObject:[NSString stringWithFormat:@"%d", idx]]; //
add key orderdetailparent
    }];

    // alloc order detail parent
```



```

NSMutableDictionary *orderDetailParent = [NSMutableDictionary
dictionaryWithObjects:orderDetailParents forKeys:keyOrderDetailParents];

// NSArray *orderObjects = @[ orderUuid, orderTotal, orderDate, orderDone,
orderDetailParent ];
// NSArray *orderKeys = @[ ORDER_UUID, ORDER_TOTAL, ORDER_DATE,
ORDER_DONE, ORDER_DETAILS_ATTRIBUTES ];
// NSDictionary *orderChild = [NSMutableDictionary dictionaryWithObjects:orderObjects
forKeys:orderKeys];

NSMutableDictionary *orderChild = @{ ORDER_UUID : orderUuid,
ORDER_TOTAL : orderTotal,
ORDER_DATE : orderDate,
ORDER_DONE : orderDone,
ORDER_DETAILS_ATTRIBUTES : orderDetailParent };

NSMutableDictionary *orderParent = @{ ORDER : orderChild };
// NSDictionary *orderParent = [NSMutableDictionary dictionaryWithObjectsAndKeys:
orderChild, ORDER, nil];

NSData *orderData = orderParent.toJSON; // convert nsdictionary to nsdata
[self startOrderUpload:kFoodPlaceOrdersURL withData:orderData];
}

```

Snippet 74. Prepare data for upload.

This code will prepare the information for the order like the name, the total price, the cart item. After that, this code will collect the information from the phone and order data such as ordering total price, order date and order done.

```

NSString *orderUuid = [MacAddress getMacAddress].toSHA1; // get UUID
NSString *orderTotal = [NSString stringWithFormat:@"%f", [self totalOrder]];
NSString *orderDate = [[NSDate date] toString];
NSString *orderDone = FALSE_VALUE; // set order to FALSE

```

Snippet 75. Get information for order.

Then, the application will create two block array order detail parents and the key of order detail parents.

```

__block NSMutableArray *orderDetailParents = [NSMutableArray array]; // init
array
__block NSMutableArray *keyOrderDetailParents = [NSMutableArray array];

```

Snippet 76. Block two array order and key order.

The application will run the loop through the array carts to get the cart item, for instance, name of food, count of food, price of food, and name of place of food.

```

[carts enumerateObjectsUsingBlock:^(Cart *cart, NSUInteger idx, BOOL *stop) {

```

```

NSString *foodName = cart.food.name;
NSString *foodCount = [cart.count stringValue];
NSString *foodPrice = [NSString stringWithFormat:@"%0.2f", [Helpers
timeNSDecimalNumber:cart.food.price andNumber:cart.count]];
NSString *foodPlace = cart.food.place.name;

// NSArray *foodObjects = @[ foodName, foodCount, foodPrice, foodPlace ];
// NSArray *foodKeys = @[ FOOD_NAME, FOOD_COUNT, FOOD_PRICE,
FOOD_PLACE ];
// NSDictionary *orderDetailChild = [NSDictionary
dictionaryWithObjects:foodObjects forKeys:foodKeys];
NSDictionary *orderDetailChild = @{ FOOD_NAME : foodName,
                                   FOOD_COUNT : foodCount,
                                   FOOD_PRICE : foodPrice,
                                   FOOD_PLACE : foodPlace };

[orderDetailParents addObject:orderDetailChild]; // add order detail child to
orderdetailparents
[keyOrderDetailParents addObject:[NSString stringWithFormat:@"%d", idx]]; //
add key orderdetailparent
}];

```

Snippet 77. Get food item information.

In the loops, the application will add these data to the key FOOD_NAME, FOOD_COUNT, FOOD_PRICE, FOOD_PLACE to the dictionary order detail child. Finally, in the loops, the application will add the data item in to the array:

```

[orderDetailParents addObject:orderDetailChild]; // add order detail child to
orderdetailparents
[keyOrderDetailParents addObject:[NSString stringWithFormat:@"%d", idx]]; //
add key orderdetailparent

```

Snippet 78. Add order detail to order.

This is a code for sending an order:

```

- (void)sendOrder {

    dispatch_queue_t sendQ = dispatch_queue_create("Send Order", NULL);
    dispatch_async(sendQ, ^{
        NSArray *carts = [self fetchedCarts];
        [self performSelectorOnMainThread:@selector(prepareOrder:) withObject:carts
        waitUntilDone:YES];
    });
    dispatch_release(sendQ);
}

```

Snippet 79. Send order function.

This code is connected with the sending order button. It sends the order to the web service. It will fetch all the cart items in the cart.

```
NSArray *carts = [self fetchedCarts];
```

Snippet 80. Get array of cart.

And it uses `performSelectorOnMainThread` operation to connect and perform the selector `prepare order and carts` object.

```
[self performSelectorOnMainThread:@selector(prepareOrder:) withObject:carts  
waitUntilDone:YES];
```

Snippet 81. Perform prepare an order.

6 TESTING

When testing, the beginning of view window will load the list of foods first. It will be delayed if your network connection is slow because of a delay loading the images on the list of foods. In order to solve this problem, there is a need for a strong network connection like Wireless network connections.

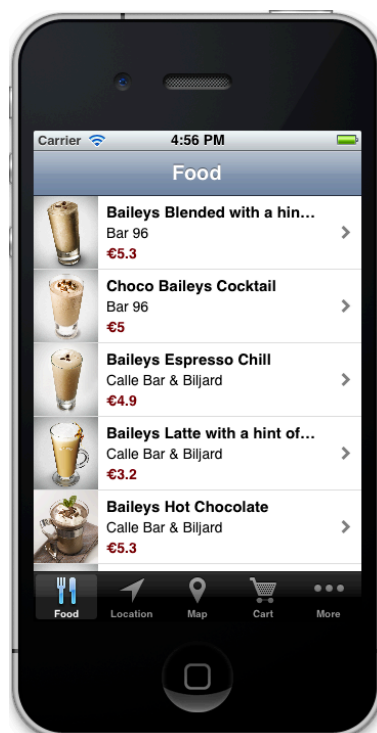


Figure 67. Users interface.

Every implemented operation was tested during coding and after finishing. Any bug was immediately found and corrected.

- The application is lagged.
- The image is overloaded.
- Show alert when no network connection.
- Show alert when adding more 5 food items.
- Show action sheet when empty the cart.
- Show alert when place an order.
- An error is happened when add food item to the cart (bug in iOS 6 beta 4).

In addition, the application has covered with a lot of the alert and logging. The GUI has been tested by simple performing operation. Besides, the application has been tested with the web service and it's performing well.

Below are the testing results and screenshots of this iOS application.

6.1 The alert of adding more 5 items

When adding the food item to cart, the application is only limited five food items per kind. So if the user adds more food items, it will show the alert to inform the user is cheating.

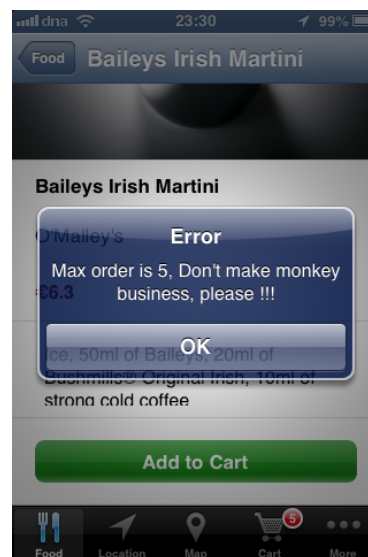


Figure 68. Show the alert.

6.2 Testing working of call out accessory

When the user chooses on the pin, it will show the call out view to inform the name of place and address of place. Also it will show the small picture of the place.

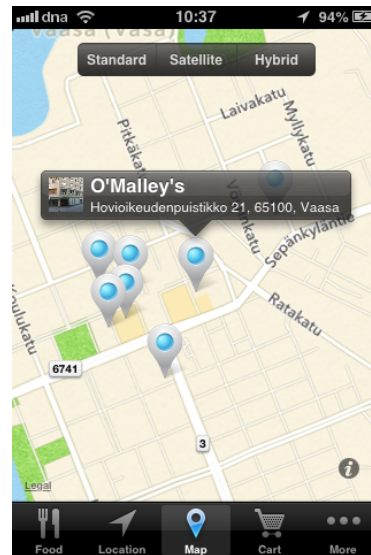


Figure 69. Call out view.

6.3 Calculate the price and badge value update

When the user adds the food item to the cart, it will show the food item like this:

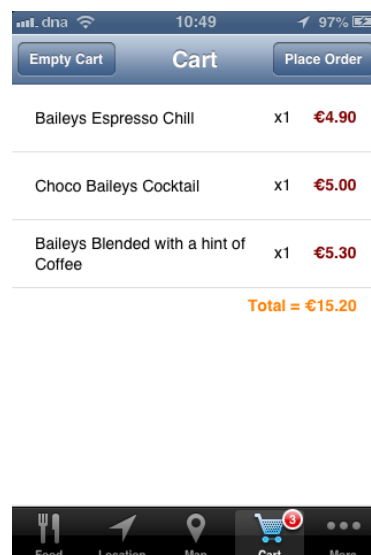


Figure 70. Food item in cart view.

In the cart, the food will show the name of food, the count of food, the total price of each food and the total price for all of the foods. When the food is added in the cart, the empty cart button and the place order button will be enabled. Also, the badge value will be displayed.

6.4 The action sheet of empty cart

When the user chooses the empty cart button, the action sheet will show to ask to confirmation. Do you really want to do it or cancel?

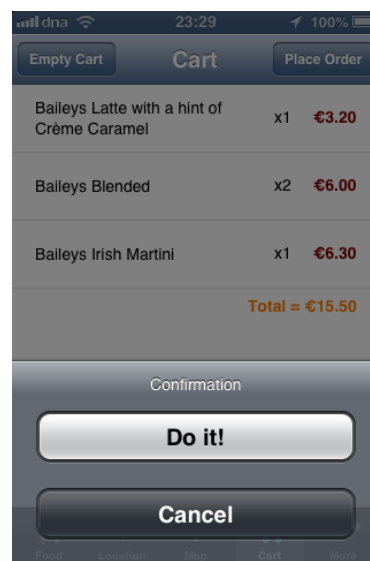


Figure 71. Show the action sheet.

6.5 The alert of place order

When the user chooses the place order button, the alert will show to ask confirmation. Do you really want to place an order or not.

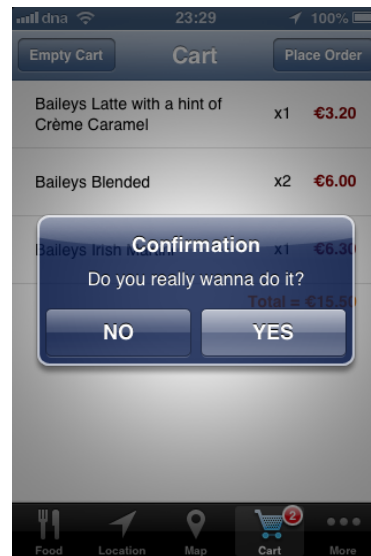


Figure 72. Show the alert view.

When the user chooses no, the alert will disappear and come back to the cart view. The action place order doesn't activate. When the user chooses yes, the place order will be activated and the alert will show the order is reserved.

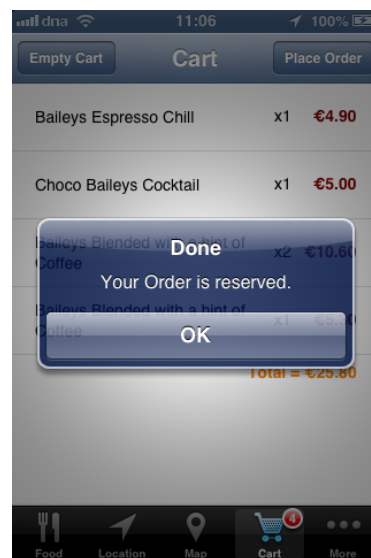


Figure 73. Show alert view done.

7 CONCLUSION

This application consists of two parts: client side is an iPhone application, which can run on iOS 5 or later versions. The server side is a web service. The client application can receive data from the web service and send data to the web service.

The client application can get a list of food data, a list of place data, a map of places and a shopping cart to store food data. It can also calculate price of foods in the shopping cart and send the order to the web service. On the server side, the web service has keeps and handles data of foods, places and orders.

And the implementation has been done on server side are create, update, remove and edit on food and place, get list of order and order detail and remove order.

The most challenging part in my application is to send the data from the client to the web service. And the function upload the data has also challenges me.

The application can be further developed in the following ways: the data from mobile phone can be synchronized with the web service, the view list of food can be displayed by collection view (collection view is a new view controller in iOS6), the passbook application will be integrated.

8 REFERENCES

- /1/ JSON – Introducing JSON (2003). [referred 15.05.2012] Available on the Internet: <URL:<http://www.json.org/>>
- /2/ Wikipedia – Cocoa (API) (2012). [WWW]. [referred 16.05.2012] Available on the Internet: <URL:[http://en.wikipedia.org/wiki/Cocoa_\(API\)](http://en.wikipedia.org/wiki/Cocoa_(API))>
- /3/ Wikipedia – JSON (2012). [WWW]. [referred 16.05.2012] Available on the Internet: <URL:<http://en.wikipedia.org/wiki/JSON/>>
- /4/ Apple Developer – What Is Cocoa? (2010). [WWW]. [referred 16.05.2012] Available on the Internet:
<URL:<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html>>
- /5/ WWDC 10 Videos – Session 117 – Building a Server-driven User Experience (2010). [WWW]. [referred 16.05.2012] Available on the iTunes:
<URL:<https://developer.apple.com/itunes/?destination=adc.apple.com.4092349126.04109539109.4144345611?i=1991648002>>
- /6/ Apple Developer – iOS Technology Overview (2011). [WWW]. [referred 16.05.2012] Available on the Internet:
<URL:<http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>>
- /7/ Apple Developer – About iOS Development (2011). [WWW]. [referred 16.05.2012] Available on the Internet:
<URL:http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSOverview/iPhoneOSOverview.html#//apple_ref/doc/uid/TP40007898-CH4-SW1>
- /8/ Apple Developer – iOS Developer Tools (2011). [WWW]. [referred 16.05.2012] Available on the Internet:
<URL:http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSDeveloperTools/iPhoneOSDeveloperTools.html#//apple_ref/doc/uid/TP40007898-CH7-SW1>

- /9/ Wikipedia – Ruby on Rails (2012). [WWW]. [referred 17.05.2012]
Available on the Internet:
<URL:http://en.wikipedia.org/wiki/Ruby_on_Rails>
- /10/ Wikipedia – Ruby (programming language) (2012). [WWW]. [referred
23.10.2012] Available on the Internet: <URL:
[http://en.wikipedia.org/wiki/Ruby_\(programming_language\)](http://en.wikipedia.org/wiki/Ruby_(programming_language))>
- /11/ Heroku – Cloud Application Platform (2012). [WWW]. [referred
25.10.2012] Available on the Internet: <URL: <http://www.heroku.com>>
- /12/ Heroku – Business (2012). [WWW]. [referred 25.10.2012] Available on the
Internet: <URL:<http://www.heroku.com/business>>