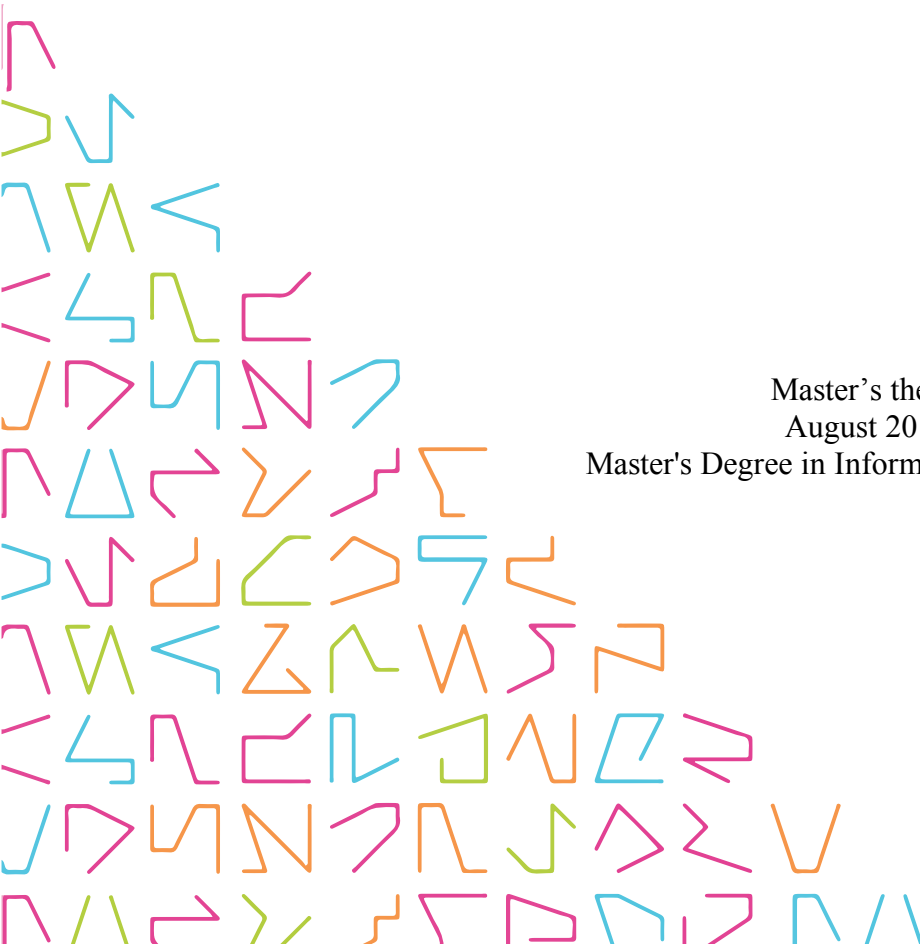


SMART SHOPPING SYSTEM

Ngoc Tuan Le

Master's thesis
August 2018
Master's Degree in Information Technology



ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Master's Degree in Information Technology

Ngoc Tuan LE:
Smart Shopping System

Master's thesis 61 pages, appendices 7 pages
August 2018

Nowadays, the growing of the smart devices and peripherals brings more convenience and wonderful experience into daily life activities. More importantly, these technologies are changing the way we interact with the world and with each other.

Bluetooth Low Energy (BLE) beacon is one of these technologies. It is making daily commute easier, paving the way for smart cities. BLE beacons are tiny transmitters which are now being used widely to provide assistance to both staff and customers wherever possible. In fact, beacons, with their micro-location capabilities can be used to offer great help to a user based on his/her location with regard to an array of things like travel, shopping, parking, choosing a restaurant, and more.

The main objective of this master's thesis was to present one of the applications of BLE beacons which is in retail. Since retail is one of the most experienced and fast-growing industries using beacons today. In fact, Business Insider predicted that in 2016, beacons would directly influence over \$40 billion worth of US retail sales at top 100 retailers. In order to do this, a web system with mobile iOS application were developed which will improve in-store shopping experiences by sending personalized promotion offers to customers while they pass by the store. The application is also capable of providing product and store information to the customer related to the promotion.

This web system and mobile application are prototype which will provide a general idea of how BLE beacons could be used in retail industry to bring better experience for customer while shopping.

Key words: smart shopping, beacon, eddystone, mobile, application

CONTENTS

1	INTRODUCTION	5
1.1	Purpose.....	5
1.2	Objective	5
1.3	Aim	5
2	BEACON TECHNOLOGY	6
2.1	Bluetooth and Bluetooth Low Energy	6
2.2	Beacon Protocols.....	7
2.2.1	iBeacon.....	8
2.2.2	Eddystone	9
2.2.3	Eddystone vs iBeacon	14
3	SYSTEM ARCHITECTURE AND USE CASES	16
3.1	Overview of System	16
3.2	System Architecture	16
3.3	Use-case diagram.....	17
3.4	Use case specification.....	19
3.5	Class diagram	33
4	SYSTEM FEATURE IMPLEMENTATION	35
4.1	Selected technologies.....	35
4.2	Store management system implementation.....	36
4.3	APIs implementation	37
4.4	Mobile Application.....	38
5	SYSTEM DEPLOYMENT AND TESTING.....	40
5.1	System deployment.....	40
5.2	System testing	43
5.2.1	System testing with super admin	44
5.2.2	System testing with store manager.....	45
5.3	Mobile application testing.....	48
6	NEXT STEP	52
7	DISCUSSION.....	53
	REFERENCES.....	54
	APPENDICES	55
	Appendix 1. forms.py of store application.....	55
	Appendix 2. admin.py of store application	57
	Appendix 3. Podfile	60
	Appendix 4. Source code repositories	61

ABBREVIATIONS AND TERMS

BR	Basic Rate
EDR	Enhanced Data Rate
LE	Low Energy
BLE	Bluetooth Low Energy
EDR	Enhanced Data Rate
PDU	Protocol Data Unit
CRC	Cyclic Redundancy Check
TX power	Transmission power
EID	Ephemeral Identifier

1 INTRODUCTION

1.1 Purpose

Researching a solution to help people to find what they need in market / mall, see where other sales for a similar product are, or find better deals or coupons as they walk through the store.

1.2 Objective

- Building an Administration Web Application which allows to manage store along with its beacon devices and provide service to create personalized promotion, sale notifications and send to shoppers.
- Building an iOS application which navigates shoppers to find needed products and receive messages from Store.

1.3 Aim

To bring a better experience to shoppers.

2 BEACON TECHNOLOGY

2.1 Bluetooth and Bluetooth Low Energy

Bluetooth is a short-range wireless communication for exchanging data that allows devices to transmit data. The key features of the Bluetooth wireless technology are robustness, low power consumption, and low cost. There are two forms Bluetooth technology: Basic Rate/Enhanced Data Rate and Low Energy.

The Bluetooth BR/EDR offers synchronous and asynchronous connections enables continuous wireless connections and uses a point-to-point network topology to establish one-to-one device communications. Bluetooth BR/EDR is ideal for providing robust wireless connection between devices ranging from headsets and cars to industrial controllers and streaming medical sensors.

The Bluetooth LE includes features designed to enable products that require lower current consumption, lower complexity and lower cost than Bluetooth BR/EDR. Bluetooth LE enables short-burst wireless connections and uses multiple network topologies. Including *point-to-point topology* to create one-to-one device communications, *broadcast topology* that establishes one-to-many device communications and *mesh topology* for many-to-many communications.

TABLE 1. Comparison between Bluetooth BR/EDR and Bluetooth LE technologies

Factors	Bluetooth BR/EDR	Bluetooth LE
Transfer rate	2-3 Mbps	1 Mbps
Range	Up to 100m	Up to 150m
Large scale network	Weak	Good
Connection set-up speed	Weak	Strong
Power consumption	Good	Very strong
Cost	Good	Strong
Suited for	Require continuous data/voice streaming such as headphones	Involves infrequent data transfers and need to operate on low power consumption

2.2 Beacon Protocols

A beacon is a small Bluetooth radio transmitter. It works like a lighthouse, repeatedly transmits a single signal that other devices can see. Instead of emitting visible light, beacon broadcasts a radio signal that allows other Bluetooth devices like smartphone can see this beacon once it's in range.

BLE allows to transmit data in different modes: connected and advertising modes. Connected mode offers exchanging data in one-to-one connection. With advertising mode, device is able to broadcast data out to any device which is listening, this is one-to-many transfer without guarantees about data coherence.

Taking this advantage of the advertising mode allows BLE Beacons to broadcast structured packets repeatedly. These packets contain information such as Preamble, Access Address, Cyclic Redundancy Check, Protocol Data Unit, etc... Most of these fields are automatically filled by Bluetooth stack, but the advertising payload is controlled by the beacons. Each beacon protocol defines a different data structure for advertising payload in 0-31 bytes long.

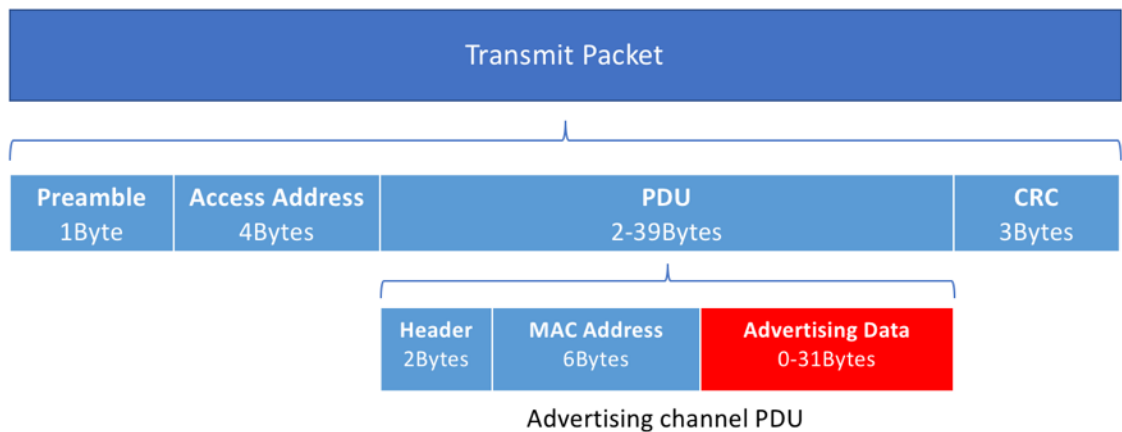


FIGURE 1. BLE packet in advertising mode

There are several beacon protocols that are released, such as: iBeacon, Eddystone, Alt-Beacon, GeoBeacon. They have their own standards and advantages. The most popular protocols are iBeacon and Eddystone that are developed and maintained by Apple and Google. Since one of objectives is building iOS application, so in this thesis I only consider between these two beacon protocols, iBeacon and Eddystone.

2.2.1 iBeacon

iBeacon is BLE advertising protocol designed by Apple. At the Apple Worldwide Developers Conference in 2013, iBeacon was introduced. And it is the first beacon protocol in the market.

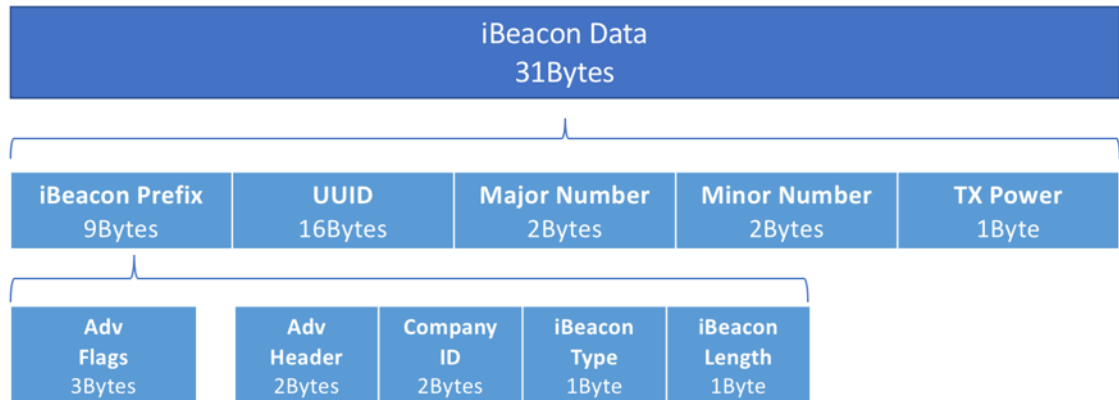


FIGURE 2. iBeacon protocol frame data spec

iBeacon advertising data contains a fixed advertising prefix which is the hex data: 0x0201061AFF004C0215. The detail of this broken-down hex data is shown as in Table 2 below.

TABLE 2. Apple's fixed iBeacon advertising prefix

Field	Size	Value	Description
Adv Flags	3 bytes	0x020106	It specifies this is only for broadcasting, not connecting.
Adv Header	2 bytes	0x1AFF	The following data is 26 bytes long and is Manufacturer Specific Data.
Company ID	2 bytes	0x004C	Apple company identifier code.
iBeacon Type	1 byte	0x02	A secondary ID that denotes a proximity beacon, which is used by all iBeacons.
iBeacon Length	1 byte	0x15	Defines the remaining length to be 21 bytes (the identifying information, 16+2+2+1).

As shown in the figure 1, the most important data of iBeacon protocol are four pieces of information: UUID, Major number, Minor number and TX power level.

The UUID, major number and minor number values provide the identifying information for iBeacon. After receiving these information, mobile application refers them against a database to obtain other information about the beacon. The beacon itself doesn't contain any descriptive data, it requires an external database to store its own data such as: store location, floor level, promotion info.

The TX power field is the measured signal strength to determine how close the smart devices to a beacon. This can be presented either as rough information (immediate/far/out of range) or as a more precise measurement in meters.

TABLE 3. iBeacon profile frame

Field	Size	Description
UUID	16 bytes	A unique ID to distinguish the iBeacons from one another.
Major number	2 bytes	A number from 1 to 65,535 that helps to identify a subset of beacons within a large group.
Minor number	2 bytes	Same as Major number, used to identify a specific beacon.
TX power level	1 byte	This number indicates the signal strength one meter from the device

2.2.2 Eddystone

In July 2015, Google introduced Eddystone protocol and firmware. At first, it was called UriBeacon, then it was announced that the tech had evolved the original specs. To differentiate, they changed the name to Eddystone. The reason perhaps that beacon is often compared to a lighthouse, so Google named their beacons format after the Eddystone Lighthouse in United Kingdom.

Eddystone is an open protocol. Eddystone beacons are able to broadcast data with four different frame types: Eddystone-UID, Eddystone-EID, Eddystone-URL and Eddystone-TLM which work with both iOS and Android. Each frame was designed to carry different set of information to fulfil the broadcasting need of most proximity beacon.

Figure 3 is the frame data spec of Eddystone. Similar to iBeacon, Eddystone protocol also has the prefix that contains the advertising flags and header data.

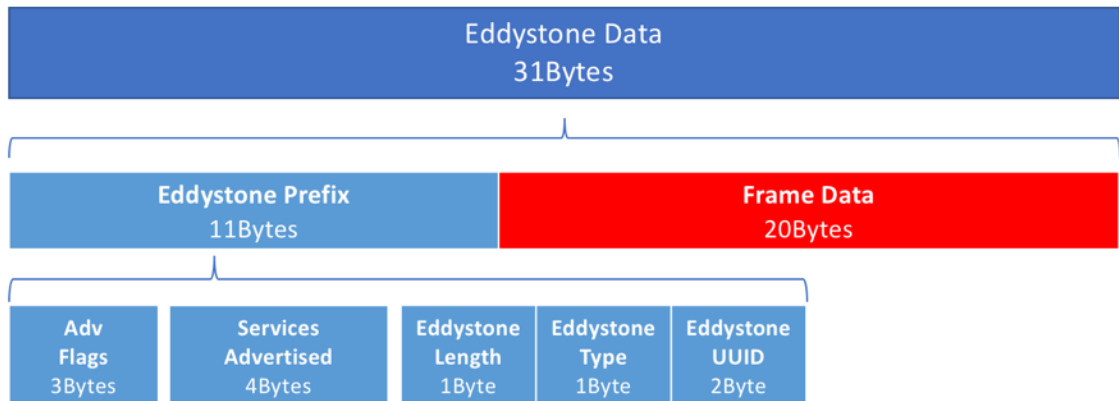


FIGURE 3. Eddystone frame data spec

TABLE 4. Eddystone advertising prefix

Field	Size	Value	Description
Adv Flags	3 bytes	0x020106	Same as iBeacon protocol it specifies this is only for broadcasting, not connecting.
Services Advertised	4 bytes	0x0303AAFE	Refers to the complete list of 16-bit Service UUIDs, that must contain the Eddystone Service UUID of 0xFEAA. This is included to allow background scanning on iOS devices.
Eddystone Length	1 bytes	0x??	The length of the remaining packet, this can vary from 6 bytes to 20. This value depends on the length of frame data.
Eddystone Type	1 byte	0x16	Service Data data type value.
Eddystone UUID	2 bytes	0xAAFE	Eddystone UUID

As mentioned earlier, Eddystone protocol has 4 different frame types for broadcasting different needs, each frame type has its own frame data structure. The individual data frames are listed below.

❖ Eddystone-UID

Eddystone-UID frame specification is very similar to the iBeacon's. As iBeacon protocol, Eddystone-UID allows mobile application to use its identifier to trigger the desired action such as querying data from server. It uses 16 bytes of this frame to store its Beacon ID, that is combination of a 10-byte namespace which is used to group a particular set of beacons (an entity/organization for example), and a 6-byte instance which is used to identify individual beacons.

Frame Type 1Byte	Ranging Data 1Byte	Name Space ID 10Bytes	Instance ID 6Bytes	RFU 2Bytes
---------------------	-----------------------	--------------------------	-----------------------	---------------

FIGURE 4. Eddystone-UID frame data spec

The Ranging Data is the transmission power in dBm which is emitted by the Eddystone beacon at 0 meters. This is different from Apple iBeacon protocol the transmission power is measured at 1 meter. However, Google recommended that the best way to determine the precise value to put into this field is to measure the actual output of Eddystone beacon from 1 meter away and then add 41 dBm to that. 41dBm is the signal loss that occurs over 1 meter.

TABLE 5. Eddystone-UID frame data spec with description

Field	Size	Description
Frame Type	1 byte	Used to specify Eddystone frame type. The value for this UID type is 0x00.
Ranging Data	1 byte	This number indicates the signal strength at 0 meters.
Name Space ID	10 bytes	A unique ID used to identify a group of beacons.
Instance ID	6 bytes	An ID allows to identify an individual beacon.
RFU	2 bytes	Reserved for future use, must be 0x0000

❖ Eddystone-EID

Frame Type 1Byte	Ranging Data 1Byte	Ephemeral Identifier 8Bytes	Not Used
---------------------	-----------------------	--------------------------------	----------

FIGURE 5. Eddystone-EID frame data spec

Eddystone-EID functions same as Eddystone-UID, however it is designed for use in security and privacy-enhanced purposes. It broadcasts an encrypted ephemeral identifier (EID) that changes periodically. This broadcast EID can be resolved remotely by the service which it was registered, but to other observers appears to be changing randomly.

TABLE 6. Eddystone-EID frame data spec with description

Field	Size	Description
Frame Type	1 byte	Used to specify Eddystone frame type. The value for this EID type is 0x30.
Ranging Data	1 byte	This number indicates the signal strength at 0 meters.
Ephemeral ID	8 bytes	A unique ID used to identify the beacon with better privacy.

❖ Eddystone-URL

Frame Type 1Byte	Ranging Data 1Byte	URL Scheme 1Byte	Encoded URL 17Bytes
---------------------	-----------------------	---------------------	------------------------

FIGURE 6. Eddystone-URL frame data spec

Eddystone-URL is designed to store and broadcast an URL in a compressed encoding format, this way enables the beacon broadcasts more information in fewer characters. With any device that has the access to internet will be able to decode the URL and to choose to visit that web page.

TABLE 7. Eddystone-URL frame data spec with description

Field	Size	Description
Frame Type	1 byte	Used to specify Eddystone frame type. The value for this EID type is 0x10.
Ranging Data	1 byte	This number indicates the signal strength at 0 meters.
URL Scheme	1 byte	Encoded Scheme Prefix
Encoded URL	17 bytes	Encoded URL

The URL scheme byte stores the value to defines the identifier scheme. These options are shown in the table 8.

TABLE 8. URL scheme options

Decimal	Hex	Expansion
0	0x00	http://www.
1	0x01	https://www.
2	0x02	http://
3	0x03	https://

The URL address are encoded in ASCII except the URL suffix. The suffix identifier is replaced by the expansion text according to the table 9 below.

TABLE 9. URL suffix options

Decimal	Hex	Expansion
0	0x00	.com/
1	0x01	.org/
2	0x02	.edu /
3	0x03	.net /
4	0x04	.info/
5	0x05	.biz/
6	0x06	.gov/
7	0x07	.com
8	0x08	.org
9	0x09	.edu
10	0xA	.net
11	0xB	.info
12	0xC	.biz
13	0xD	.gov
14..32	0x0E..0x20	Reserved
127..255	0x7F..0xFF	Reserved

❖ Eddystone-TLM

Eddystone-TLM frame is known as the telemetry frame. It broadcasts telemetry information about the beacon itself (encrypted or unencrypted) such as battery voltage, device temperature, and counts of broadcast packets.

Frame Type 1Byte	Ranging Data 1Byte	Battery 2Bytes	Temperature 2Bytes	Advertising PDU Count 4Bytes	Time 4Bytes	Not Used
---------------------	-----------------------	-------------------	-----------------------	---------------------------------	----------------	----------

FIGURE 7. Eddystone-TLM frame data spec

Same as other frame types, the data frame starts with frame type which is “0x20” and followed by the ranging data. The subsequent data are device specific information like battery voltage, Beacon temperature, Advertising PDU count and Time since power-on or reboot. It is not necessary to provide all of the data; the values of obscured data won’t be updated.

TABLE 10. Eddystone-URL frame data spec with description

Field	Size	Description
Frame Type	1 byte	Used to specify Eddystone frame type. The value for this EID type is 0x10.
Ranging Data	1 byte	This number indicates the signal strength at 0 meters.
Battery	2 bytes	Beacon battery voltage is in 1mV per bit resolution.
Temperature	2 bytes	Beacon temperature is in degrees Celsius expressed in 8.8 fixed point representation.
Advertising PDU Count	4 bytes	The number of advertising frame that has been sent since power-on or reboot.
Time	4 bytes	The time since power-on or reboot.

Eddystone-TLM cannot be a standalone frame in the Eddystone protocol. It needs to be used in conjunction with the UID or URL frame.

2.2.3 Eddystone vs iBeacon

Eddystone and iBeacon almost works in the same way. But Eddystone provides some extended functionality. iBeacon broadcasts only one advertising packet that has a unique ID number contains UUID, Minor, Major and TX power level. While Eddystone allows

to broadcast four different packets can be used individually or in combinations to create beacons. More importantly, iBeacon requires for native apps, but building and maintaining such apps are not always feasible for all kinds of businesses. While Eddystone directly works with the Chrome browser on smartphones to deliver Physical Web notification or URL in order to be redirected to the relevant web interface.

Both Eddystone and iBeacon are compatible with Android and iOS. However, iBeacon is native only for iOS while Eddystone is open-source and cross-platform, so it is compatible with any platform that supports BLE beacons. Eddystone is flexible but requires more complicated coding to integration since it sends more packets than iBeacon.

In iBeacon the signal transmitted is a public that can be detected by any device with proper specifications. With Eddystone, it also provides a built-in feature call EIDs that constantly change and allow beacons to broadcast a signal which can only be identified by authorized clients. Besides that, Google has launched Google beacon platform which is a cloud service that provides two APIs (Nearby API and Proximity Beacon API) that makes Eddystone more powerful and beacon management much easier.

In conclusion, Eddystone is more powerful, flexible and secure than iBeacon. Using Eddystone not only allows to manage your beacons and associated data on Google beacon platform, but also to integrate fully with many other Google services such as Google Analytics for getting better understanding about your customer. These are the reasons why Eddystone is better solution.

3 SYSTEM ARCHITECTURE AND USE CASES

3.1 Overview of System

Smart Shopping System is described in three different parts, they are:

1. Store Management System that allows store manager to manage store generally and provides APIs for mobile application to retrieve information.
2. Mobile application will scan the nearby beacon and retrieve the information from Store Management System and display to customer.
3. In-store beacons / Eddystones will be installed in the store and broadcast a UID that link to the information which is stored in Store Management System.

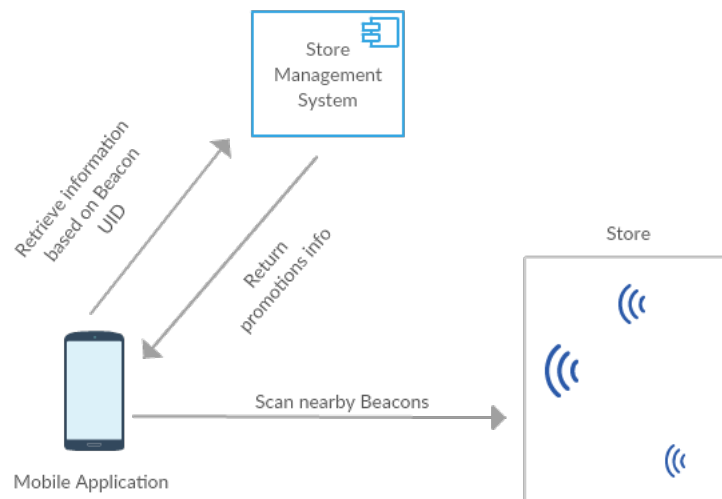


FIGURE 6: Smart Shopping System setup

The core function of system is sending personalized promotions, offers to customer devices in real time (when they are close to or in the shops). To detect when customers are close or in the store, the beacons will be installed in-store which will broadcast a unique identifier which is associated with its location and store information that are stored on web server. Mobile application that is installed on customer's devices will scan those beacons, allows to detect nearby stores, and send request to server to retrieve store information as well as promotions and then notifies customers.

3.2 System Architecture

Figure 7 illustrates a general view of the system. Server side includes Rest API Services and Back Office components. Back Office is actually a web application tool which allows store manager managing the store. In the other side, via Rest API Services, server can provide client side (which is mobile application) API services to authenticate user and fetch store information (such as: product info, notification), this component also takes the responsibly to send the notifications to mobile application.

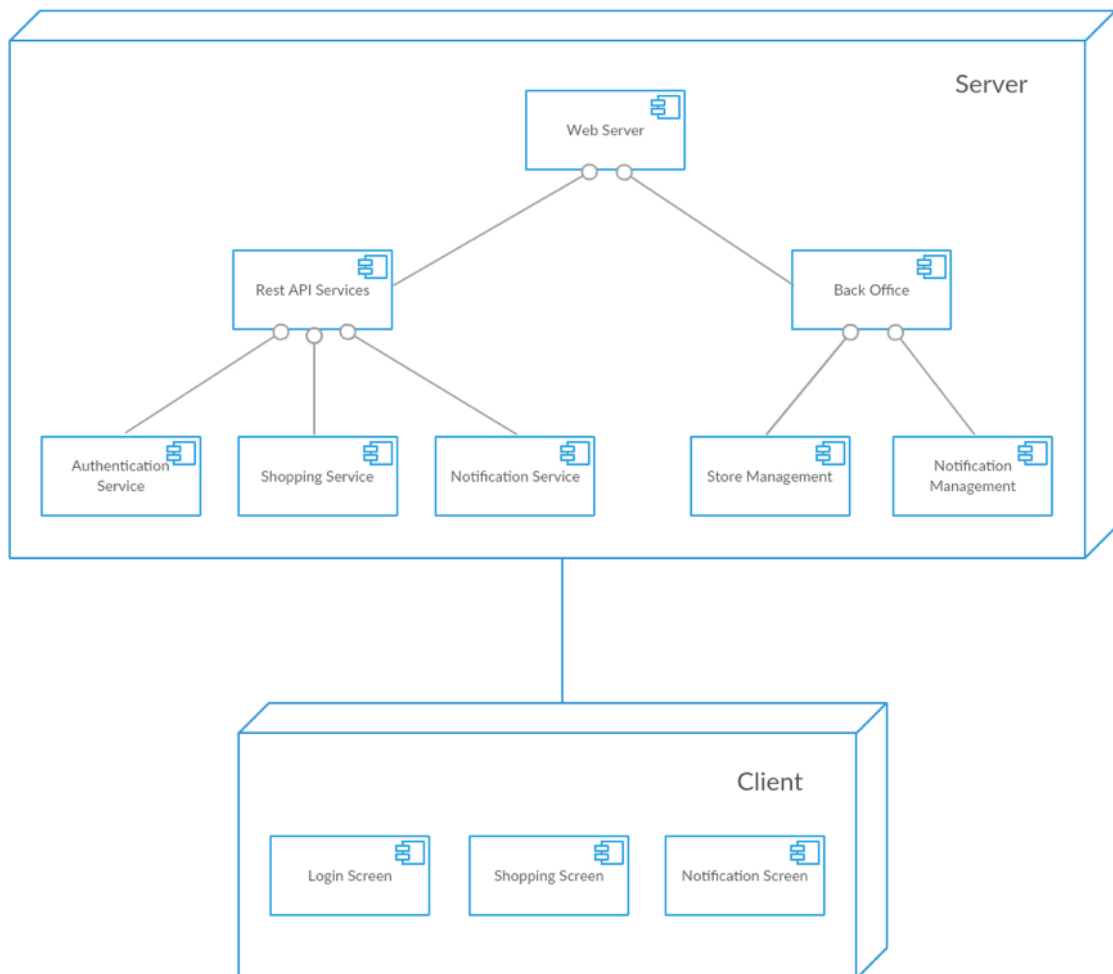


FIGURE 7: Initial design of system

3.3 Use-case diagram

In software and systems engineering, a use case is a list of actions or event steps typically defining the interactions between a role (known as an actor) and a system to achieve a goal. The actor can be a human or a system.

There are 3 main actors that will be using this system:

- Super admin who is responsible for system management. He/she will involve doing all the administration tasks like: adding new store into system, editing/removing the store, adding new manager to system to manage a store.

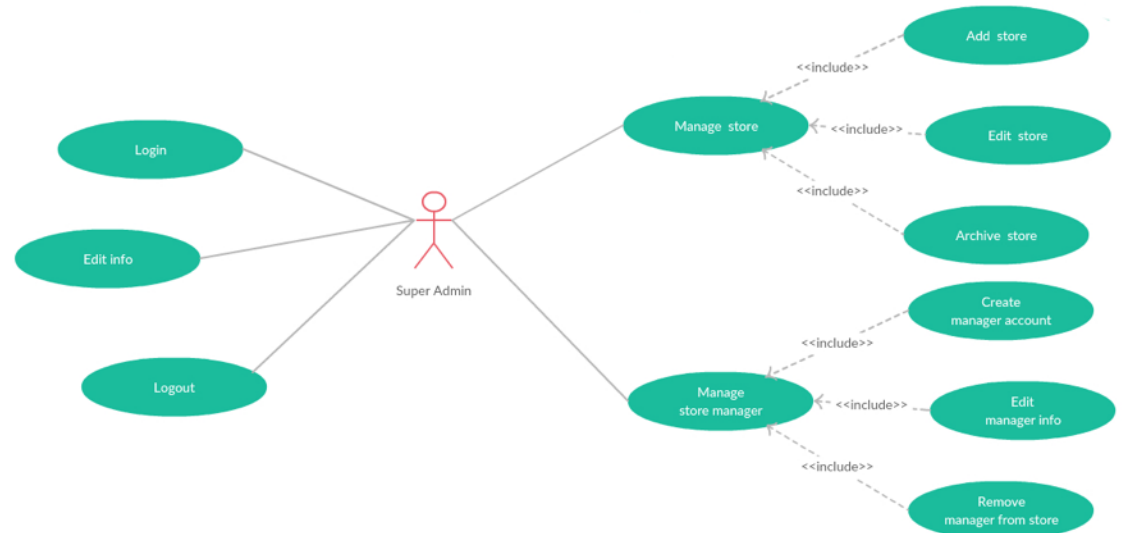


FIGURE 8: Super admin use cases

- Store manager who will be managing the stores. His responsibility is managing everything related to the store such as: catalogues, products, Eddystone beacons which are installed in store and the notification that are linked to them.



FIGURE 9: Store manager use cases

- Shopper who uses the mobile application to view the catalogues, view the products and receive/view the notifications from the store.

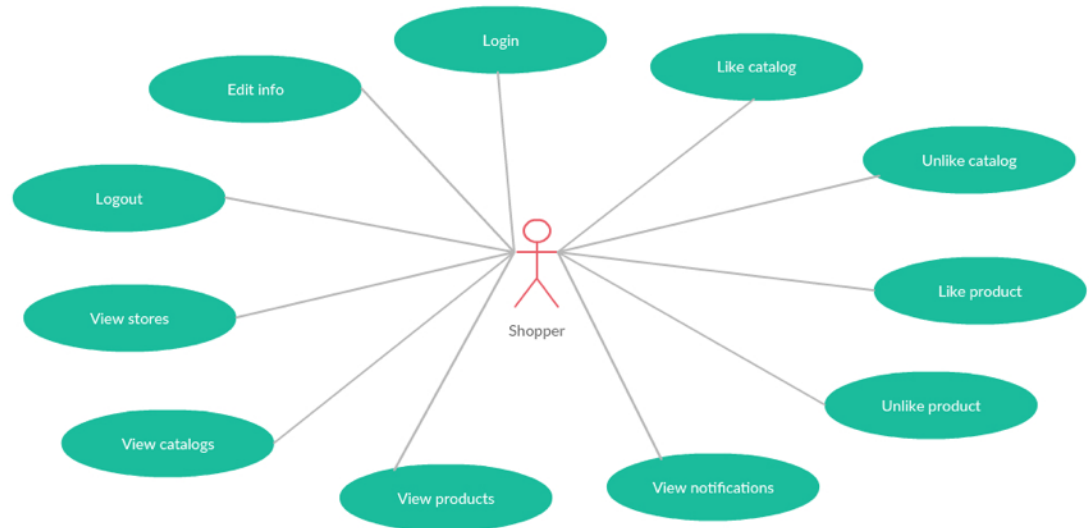


FIGURE 10: Shopper use cases

3.4 Use case specification

The use case specifications below will give a detailed explanation on how each use case works and define the actor's activities and how it interacts with the system.

❖ Use case: Login

- Primary actor(s): Super Admin, Store Manager, Shopper
- Scope: Whole system
- Brief description: This use case describes how the users of system login.
- Basic flow:
 - The use case starts when user starts the application.
 - The system will show the login form.
 - The user input his/her username and password into login form.
 - The system validates user's username and password and log him/her into the system.
 - The system displays the dashboard page and the use case end.

- Alternative flow: If in the basic flow the system cannot find the account with the username or the password is invalid, an error message is displayed. The user can re-type a new username and password or choose to cancel the operation, at which point the use case ends.
- Preconditions: There is no precondition associated with this use case.
- Postconditions: There is no postcondition associated with this use case.

❖ **Use case: Edit account info**

- Primary actor(s): Super Admin, Store Manager, Shopper
- Scope: Whole system
- Brief description: This use case describes how the user edit his/her account info.
- Basic flow:
 - The use case starts when the user opens user info page.
 - The system provides an editor area with user information fields.
 - The user edits the information which she/he wants to.
 - The system validates the inputs, store the data and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The user can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in.
- Postconditions: The entered data is saved in user account.

❖ **Use case: Logout**

- Primary actor(s): Super Admin, Store Manager, Shopper
- Scope: Whole system
- Brief description: This use case happens when the user signs out.
- Basic flow:
 - The use case starts when the user clicks on logout button.
 - The user's information will be removed from local storage.
 - The system leads user to the home page.
- Alternative flow: None.
- Preconditions: The user has logged in.
- Postconditions: There is no postcondition associated with this use case.

❖ Use case: Add store

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case allows the super admin to add a new store into the system.
- Basic flow:
 - The use case starts when the super admin clicks add new store on store management page.
 - The system displays a store information form.
 - The super admin needs to input all of required information.
 - The system validates the inputs, store the data and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The super admin can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has super admin role.
- Postconditions: The entered data is saved in store account.

❖ Use case: Edit store

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case enables the super admin to update a store information.
- Basic flow:
 - The super admin opens store management page.
 - The system displays a list of stores.
 - The use case starts when the super admin selects the store that he/she wants to modify information.
 - The system displays a store information form.
 - The super admin enters the information.
 - The system validates the inputs, store the data and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The super admin can re-type a new input or choose to cancel the operation, at which point the use case ends.

- Preconditions: The user has logged in and has super admin role.
- Postconditions: The entered data is saved in store account.

❖ **Use case: Remove store**

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case enables the super admin to remove a store from system.
- Basic flow:
 - The super admin opens store management page.
 - The system displays a list of stores.
 - The use case starts when the super admin selects the store that he/she wants to delete and click remove button.
 - The system displays the confirmation alert box.
 - If the super admin click Yes button and then the system displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the super admin chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has super admin role.
- Postconditions: The data of selected store will be deleted from system, includes managers, catalogue and product data in that store.

❖ **Use case: Add store manager**

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case allows the super admin to add a new store manager into the system to be able manage a store.
- Basic flow:
 - The use case starts when the super admin clicks add new store manager on manager management page.
 - The system displays a manager information form.
 - The super admin needs to input all of required information.
 - The super admin selects the store which will be managed by this new manager.

- The system validates the inputs and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The super admin can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has super admin role.
- Postconditions: The entered data is saved in manager account.

❖ Use case: Edit store manager

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case enables the super admin to modify a manager information.
- Basic flow:
 - The use case starts when the super admin selects the manager account that he/she wants to modify information on the list of store manager in store manager management page.
 - The system displays a store information form.
 - The super admin enters the information.
 - The system validates the inputs and displays the successful message.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The super admin can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has super admin role.
- Postconditions: The entered data is saved in manager account.

❖ Use case: Remove store manager

- Primary actor(s): Super Admin
- Scope: Back Office
- Brief description: This use case enables the super admin to remove a store from system.
- Basic flow:
 - The super admin opens store management page.
 - The system displays a list of stores.

- The use case starts when the super admin selects the store that he/she wants to delete and click remove button.
- The system displays the confirmation alert box.
- If the super admin click Yes button, the system displays the successful message.
- Alternative flow: If in the basic flow the super admin chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has super admin role.
- Postconditions: The data of selected store will be deleted from system.

❖ Use case: Add catalogue

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case describes how the manager add a new catalogue into the store that he/she manages.
- Basic flow:
 - The use case starts when the manager clicks on add new catalogue button on catalogue management page.
 - The system displays a form with catalogue information fields.
 - The manager enters the required information.
 - The system validates the inputs, displays the successful message and the use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered catalogue data is saved in system.

❖ Use case: Edit catalogue

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case allows the manager to modify a catalogue information.
- Basic flow:

- The use case starts when the manager selects the catalogue that he/she wants to modify information on the catalogue list in store catalogue management page.
- The system displays a form with catalogue information fields.
- The manager enters the information.
- The system validates the inputs and displays the successful message.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered data is saved in catalogue.

❖ Use case: Remove catalogue

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case enables the store manager removes a catalogue from a store which is managed by him/her in system.
- Basic flow:
 - The manager opens catalogue management page of the store.
 - The system displays a list of catalogues in that store.
 - The use case starts when the manager selects the catalogue that he/she wants to delete and click remove button.
 - The system displays the confirmation alert box.
 - If the manager decides to continue the process, the system displays the successful message.
- Alternative flow: If in the basic flow the manager chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The selected catalogue data is removed in system.

❖ Use case: Add product

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case describes how the store manager edit add new product into a store that is managed by him/her.

- Basic flow:
 - The use case starts when the manager clicks on add new product button on product management page.
 - The system provides a form with product information fields.
 - The manager enters the product information.
 - The system validates the inputs and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered product data is saved in system.

❖ Use case: Edit product

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case enables the manager to modify a product information.
- Basic flow:
 - The use case starts when the manager selects the product that he/she wants to modify information on the product list in product management page.
 - The system displays a form with product information fields.
 - The manager enters the information that he/she wants to modify.
 - The system validates the inputs and displays the successful message.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered data is saved in the selected product.

❖ Use case: Remove product

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case allows the store manager to remove a product from a store which is managed by him/her in system.

- Basic flow:
 - The manager opens product management page of the store.
 - The system displays a list of catalogues in that store.
 - The use case starts when the manager selects the product that he/she wants to delete and click remove button.
 - The system displays the confirmation alert box.
 - If the manager decides to continue the process, the system displays the successful message.
- Alternative flow: If in the basic flow the manger chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered catalogue data is removed from system.

❖ Use case: Create notification

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case describes how the manager create a new notification for a store and link to a beacon.
- Basic flow:
 - The use case starts when the manager clicks on add new notification button on notification management page.
 - The system provides a form with notification information fields.
 - The manager enters the notification information and selects the beacon that this new notification will be referred to.
 - The system validates the inputs and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered notification data is saved in system.

❖ Use case: Edit notification

- Primary actor(s): Store manager
- Scope: Back Office

- Brief description: This use case enables the manager to modify an exist notification.
- Basic flow:
 - o The use case starts when the manager selects the notification that he/she wants to modify information on the notification list in notification management page.
 - o The system displays a form with notification information fields.
 - o The manager enters the information that he/she wants to modify.
 - o The system validates the inputs and displays the successful message.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered notification is saved.

❖ Use case: Remove notification

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case describes how the manager remove an expired notification.
- Basic flow:
 - o The manager opens notification management page of the store.
 - o The system displays a list of notification in that store.
 - o The use case starts when the manager selects the notification that he/she wants to delete and click remove button.
 - o The system displays the confirmation alert box.
 - o If the manager decides to continue the process, the system displays the successful message.
- Alternative flow: If in the basic flow the manger chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The selected notification data is removed from system.

❖ Use case: Add beacon

- Primary actor(s): Store manager

- Scope: Back Office
- Brief description: This use case describes how a beacon added to a store.
- Basic flow:
 - o The use case starts when the manager clicks add new beacon to store on beacon management page of the store.
 - o The system provides an editor area with beacon information fields.
 - o The manager needs to enter correctly the information of the beacon.
 - o The system validates the inputs, store the data and displays the successful message. The use case ends.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered beacon data is saved in system.

❖ Use case: Edit beacon

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case enables the manager to modify an exist beacon.
- Basic flow:
 - o The use case starts when the manager selects the beacon that he/she wants to modify information on the beacon list in beacon management page.
 - o The system displays a form with beacon information fields.
 - o The manager enters the information that he/she wants to modify.
 - o The system validates the inputs and displays the successful message.
- Alternative flow: If in the basic flow the input data is invalid, an error message is displayed. The manager can re-type a new input or choose to cancel the operation, at which point the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The entered beacon data is saved.

❖ Use case: Remove beacon

- Primary actor(s): Store manager
- Scope: Back Office
- Brief description: This use case enables a store manager to remove a beacon.

- Basic flow:
 - The manager opens beacon management page of the store.
 - The system displays a list of beacons in that store.
 - The use case starts when the manager selects the beacon that he/she wants to remove and click remove button.
 - The system displays the confirmation alert box.
 - If the manager decides to continue the process, the system displays the successful message.
- Alternative flow: If in the basic flow the manager chooses to cancel the operation, the system will do nothing and the use case ends.
- Preconditions: The user has logged in and has manager role.
- Postconditions: The selected beacon data is removed from system.

❖ Use case: View store

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case allows shopper to see the list of stores in the system.
- Basic flow:
 - The use case starts when the shopper opens the mobile application.
 - Mobile app displays all of stores in the system. The use case ends.
- Alternative flow: If the mobile can't fetch data from the system, an error message will be displayed.
- Preconditions: The user has logged in.
- Postconditions: There is no postcondition associated with this use case.

❖ Use case: View catalogue

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case allows shopper to see the list of catalogues in a store.
- Basic flow:
 - The use case starts when the shopper selects to view a store.
 - Mobile app displays all catalogues in that store. The use case ends.

- Alternative flow: If the mobile can't fetch data from the system, an error message will be displayed.
- Preconditions: The user has logged in.
- Postconditions: There is no postcondition associated with this use case.

❖ **Use case: View product**

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case allows shopper to see the products in a store.
- Basic flow:
 - The use case starts when the shopper selects to view a catalogue.
 - Mobile app displays all of products in the selected catalogue. The use case ends.
- Alternative flow: If the mobile can't fetch data from the system, an error message will be displayed.
- Preconditions: The user has logged in.
- Postconditions: There is no postcondition associated with this use case.

❖ **Use case: View notification**

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case allows shopper to receive and see the notifications that he/she interests in.
- Basic flow:
 - The use case starts when the shopper selects to view notifications button on the app.
 - Mobile app displays all of notifications that he/she interests in. The use case ends.
- Alternative flow: If the mobile can't fetch data from the system, an error message will be displayed.
- Preconditions: The user has logged in.
- Postconditions: There is no postcondition associated with this use case.

❖ **Use case: Like catalogue**

- Primary actor(s): Shopper

- Scope: Mobile application
- Brief description: This use case describes how the users of system can follow a catalogue of store.
- Basic flow:
 - The mobile app provides a like button for each catalogue.
 - The use case starts when user click on this like button.
 - The app displays the successful message. The use case ends.
- Alternative flow: If in the basic flow there is any error happens, a message is displayed. The use case ends.
- Preconditions: The user has logged in.
- Postconditions: The information of liked catalogue will be stored in system.

❖ **Use case: Unlike catalogue**

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case describes how the users of system unfollow a catalogue of store.
- Basic flow:
 - The mobile app provides an unlike button for each liked catalogue.
 - The use case starts when user click on this unlike button.
 - The app displays the successful message. The use case ends.
- Alternative flow: If in the basic flow there is any error happens, a message is displayed. The use case ends.
- Preconditions: The user has logged in.
- Postconditions: The information of liked catalogue will be removed from system.

❖ **Use case: Like product**

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case describes how the users of system like a product of store.
- Basic flow:
 - The mobile app provides a like button for each product.
 - The use case starts when user click on this like button.
 - The app displays the successful message. The use case ends.

- Alternative flow: If in the basic flow there is any error happens, a message is displayed. The use case ends.
- Preconditions: The user has logged in.
- Postconditions: The information of liked product will be stored in system.

❖ Use case: Unlike product

- Primary actor(s): Shopper
- Scope: Mobile application
- Brief description: This use case describes how the users of system unlike a product of store.
- Basic flow:
 - The mobile app provides an unlike button for each liked product.
 - The use case starts when user click on this unlike button.
 - The app displays the successful message. The use case ends.
- Alternative flow: If in the basic flow there is any error happens, a message is displayed. The use case ends.
- Preconditions: The user has logged in.
- Postconditions: The information of liked product will be removed from system.

3.5 Class diagram

In software engineering, a class diagram in the Unified Modelling Language (UML) is a type of static structure diagram that describes the structure of a system by illustrating the system's classes, their attributes, methods, and the relationships among objects.

The figure 12 is class diagram of Smart Shopping System that allows to see the detail of objects as well as the relationships between them in the system, such as:

- A store has none or many catalogues and products.
- A beacon needs to be installed into a store.
- A notification needs to be linked with a beacon in a store.
- All of user types contains some common information and actions which is inherited from User class.
- Shopper can like/unlike catalogues and product and these data will be saved into Interested Catalogues and Interested Products.

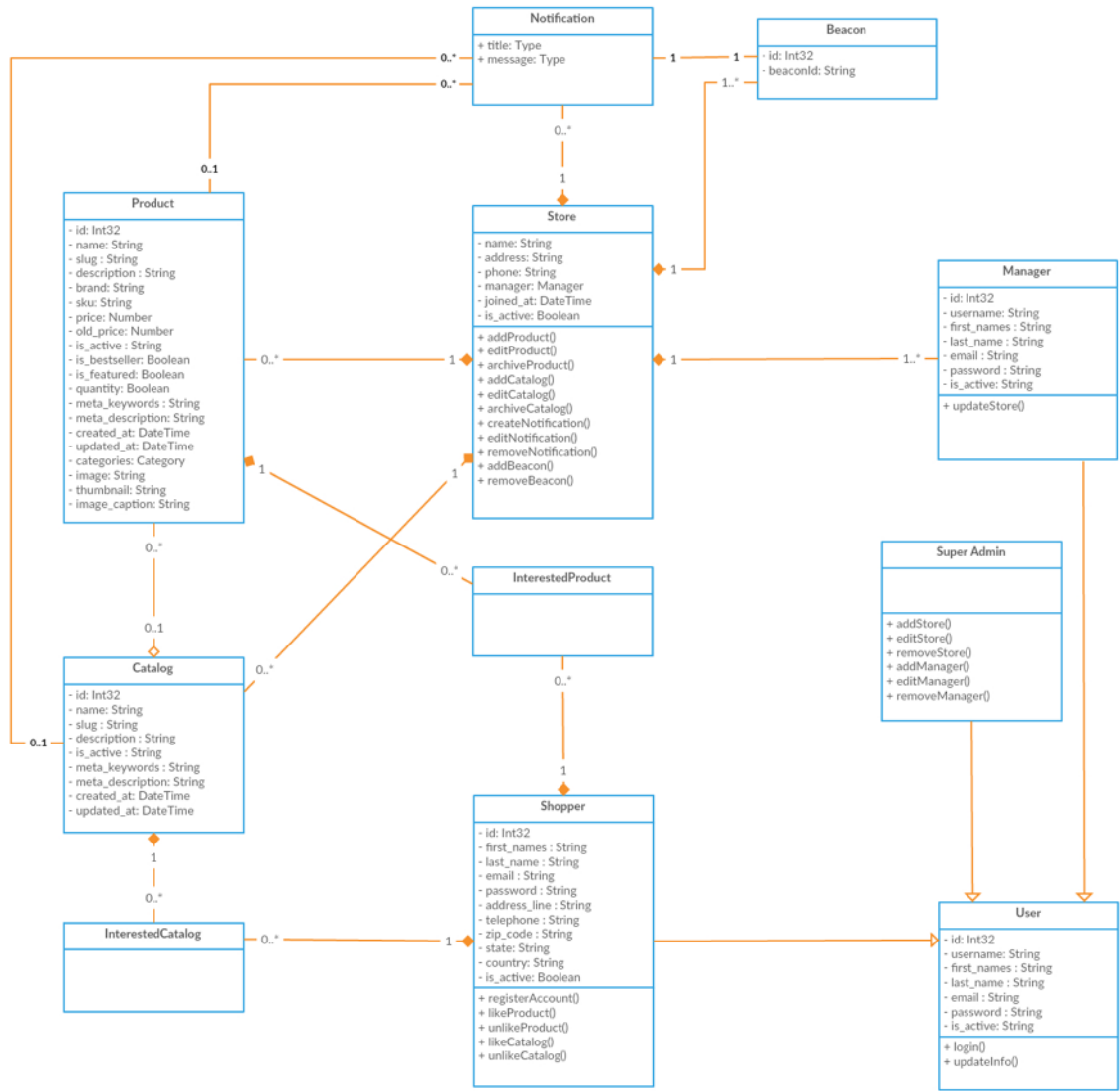


FIGURE 12: Class diagram

4 SYSTEM FEATURE IMPLEMENTATION

As the main purpose of this thesis is providing an implementation plan for a smart shopping system, all of the source code will be uploaded and shared on Github. Therefore, to avoid the repetition of the same information, the coding level won't be explained in this report. In contrary, the technical solution and setting up guideline will be provided.

4.1 Selected technologies

❖ Django Framework

In order to achieve the objectives of this thesis, the selected technology needs to be not only a quick-implementation solution but also efficient and convenient that will help to reduce the development time and cost. That's the reason why Django framework was chosen. Django is an open-source framework powered by Python that provides all stuff needed "out of the box". Furthermore, Django natively supports common database engines such as MySQL, PostgreSQL, SQL Server, DB2 and uses Object Relation Mapper to map project's object with database tables. In this project, PostgreSQL is selected as database solution because of personal preference. Last but not least, Django framework also provides a nice admin panel where the admins and managers can manage the system content.

❖ Django Rest Framework

As mentioned in the system architecture section, the store management system also needs to provide API services for mobile application. However, Django Framework is not made to deal with RESTful API. Therefore, Django REST framework which is a flexible extension of Django is chosen to build Web APIs.

❖ Cookiecutter

Cookiecutter is a utility which helps to create a project from a template. Cookiecutter provides a lot of Python package project templates, includes Django Rest Framework project template. It will help to save time for setting up the environment as well as deployment and allow to focus on researching, implementing the application features and the APIs for mobile application.

❖ Docker

Docker is a program that performs operating-system-level virtualization also known as containerization that allows developers to build, ship and run distributed applications. Docker Compose is a tool for defining and running complex application with Docker. Using Compose allows to define a multi-container application in a single file (docker-compose.yml). Then, all of the application services will be created and started with a single command.

❖ Cookiecutter Django Rest

This is a project template allows creating Django web application which is integrated with Django Rest Framework and PostgreSQL completely. The template also uses Docker Compose to configure and run the application's services from a YAML file.

4.2 Store management system implementation

Although Django framework is not a CMS like Drupal, Symfony or Wordpress, but it provides CMS-like functionality which is admin panel model. These functionalities are actually implemented inside small applications in Django project. An application includes some combination of models, views, templates, static files, URLs, etc. Each application does something specifically such as user management, store management. They're generally wired into projects with the `INSTALLED_APPS` setting and optionally with other mechanisms such as `URLconfs`, the `MIDDLEWARE` setting, or template inheritance.

Based on the requirements, store management system can be divided into 2 applications which are user management app and store management app.

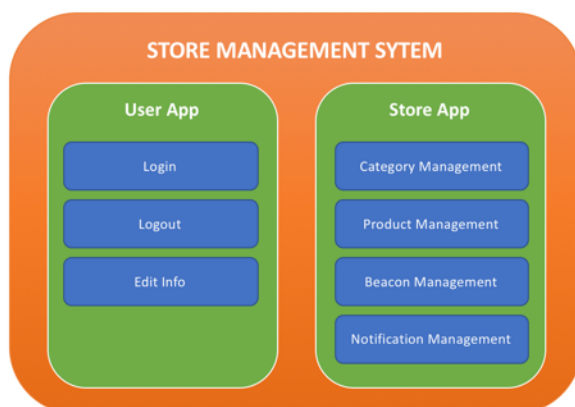


FIGURE 13: Store management system application modules

User application provides login, logout and edit user information features while store application allows the manager to manage the content of the store includes category, product, beacon and notification. These management features can be generated by Django framework easily by declaring all of system classes (same as the design in class diagram section) inside the *model.py* file of each application. However, it will require some extra coding works to customize the default settings to match the requirement of system. These extra works can be seen in *forms.py* (Appendix 1) and *admin.py* (Appendix 2) files of user app and store app in the project.

4.3 APIs implementation

Django framework uses the Model Template View (similar to Model View Controller) pattern, the model represents the data, the template displays the user interface and the view manages the logic and user's interaction. The figure 14 illustrate the data flow of this pattern.

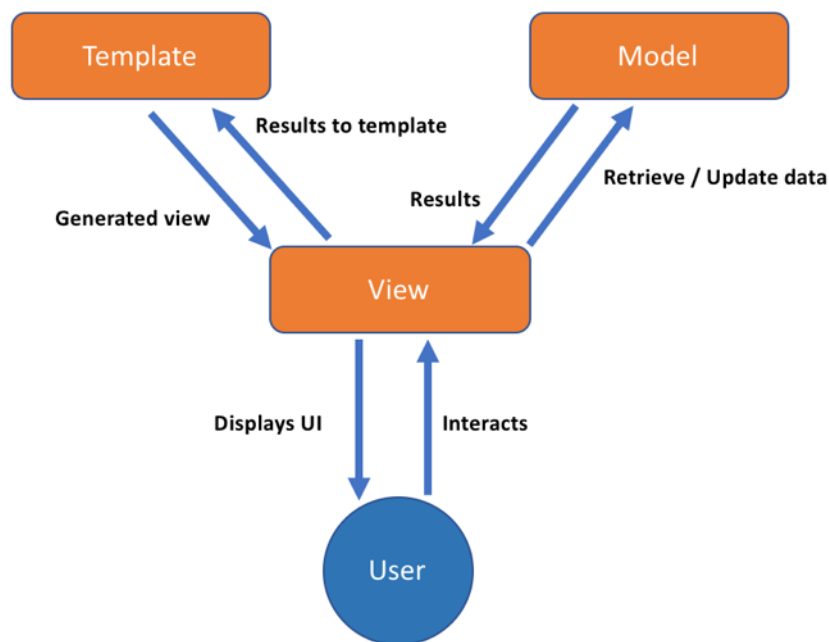


FIGURE 14: Django MTV pattern's data flow

In order to provide a straightforward way to build APIs, Django REST framework transforms Django app into server side and does not display any content but will handle the requests from the client to provide the data or update the data. It means the view no longer renders the model into content view, instead the data is serialized and send it back to

client. With Django REST framework, now the data flow is changed for API implementation. This data flow is shown in figure 15.

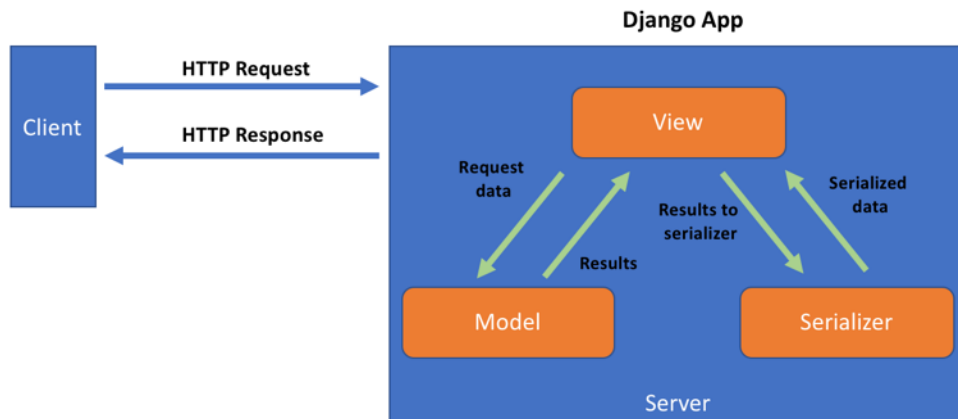


FIGURE 14: Data flow in Django Rest Framework

4.4 Mobile Application

As the objective of this thesis is building an iOS application, the selected technology is iOS SDK. Although there are many frameworks out there which also allow to build iOS application, however, in order to bring a better native experience to shopper iOS SDK is always the best choice.

Another utility is also being used in this iOS project for dependency management is Cocoapods. Cocoapods helps to make installing, uninstalling and updating the third-party libraries for iOS project much easier. All of the libraries just need to be declared inside *Podfile*, then Cocoapods will automatically install and integrate them to iOS project.

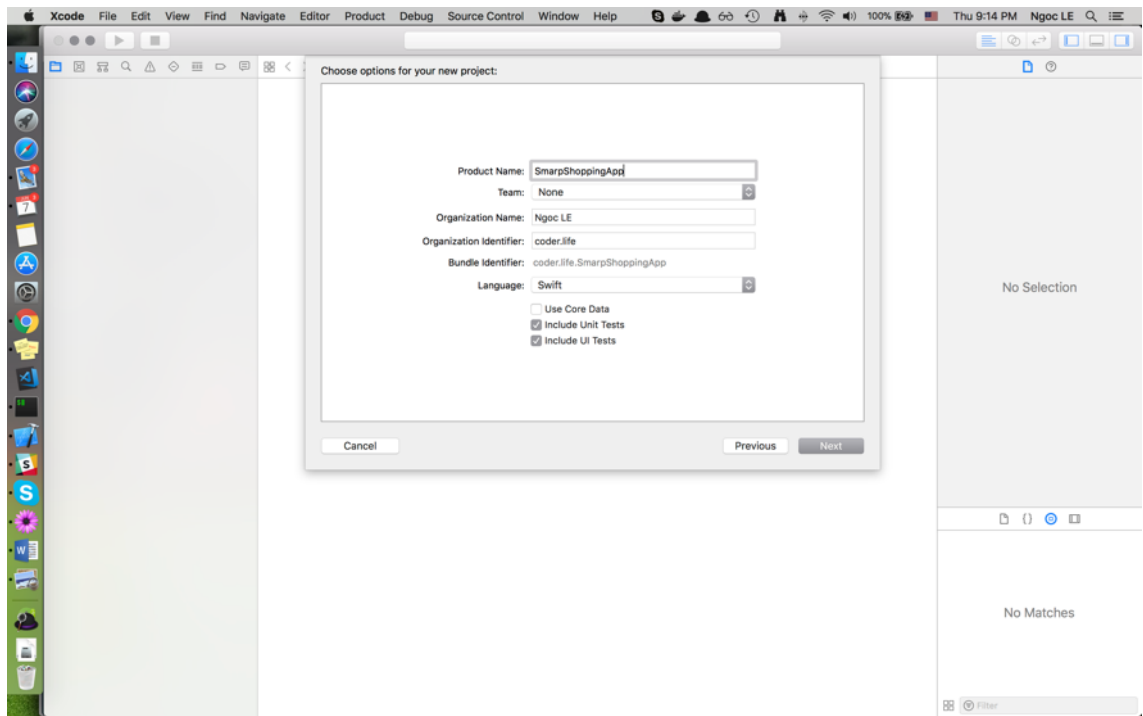
CocoaPods is built with Ruby and is installable with the default Ruby available on OS X. Using this default Ruby install will require to use *sudo*, which will execute the command as root user. The picture 1 below shows how to install Cocoapods.

```

ngocle@Nicks-MacBook-Pro ~/TAMK/smartshoppingbe/smartshoppingsystem master • sudo gem install cocoapods
Password:
Fetching: cocoapods-1.5.3.gem (100%)
Successfully installed cocoapods-1.5.3
Parsing documentation for cocoapods-1.5.3
Installing ri documentation for cocoapods-1.5.3
Done installing documentation for cocoapods after 3 seconds
1 gem installed
  
```

PICTURE 1: Cocoapods installation

After finished installation, the next step needs to be done is creating project from Xcode.



PICTURE 2: Create project with Xcode

To be able to use Cocoapods as a dependency manager in this Xcode project, *Podfile* is required to be created in the root folder of project. It can be created by a text editor or command *pod init*.

After created *Podfile* for project and declare all the libraries which will be used in project in this file (Appendix 3), it requires to run install command to add all of these libraries into project. This step is shown in picture 3 below.

```

ngocle@Ngoc-MacBook-Pro ~ % cd "/Users/ngocle/SmartShoppingApp" && pod install
Analyzing dependencies
Downloading dependencies
Installing Alamofire (4.7.1)
Installing EstimoteDV (4.27.0)
Installing Material (2.14.0)
Installing Motion (1.3.5)
Generating Pods project
Integrating client project
Sending stats
Pod installation complete! There are 3 dependencies from the Podfile and 4 total pods installed.

(!) Automatically assigning platform 'ios' with version '10.0' on target 'SmartShopping' because no platform was specified. Please specify a platform for this target in your Podfile. See 'https://guides.cocoapods.org/syntax/podfile.html#platform'

```

PICTURE 3: Install libraries with Cocoapods

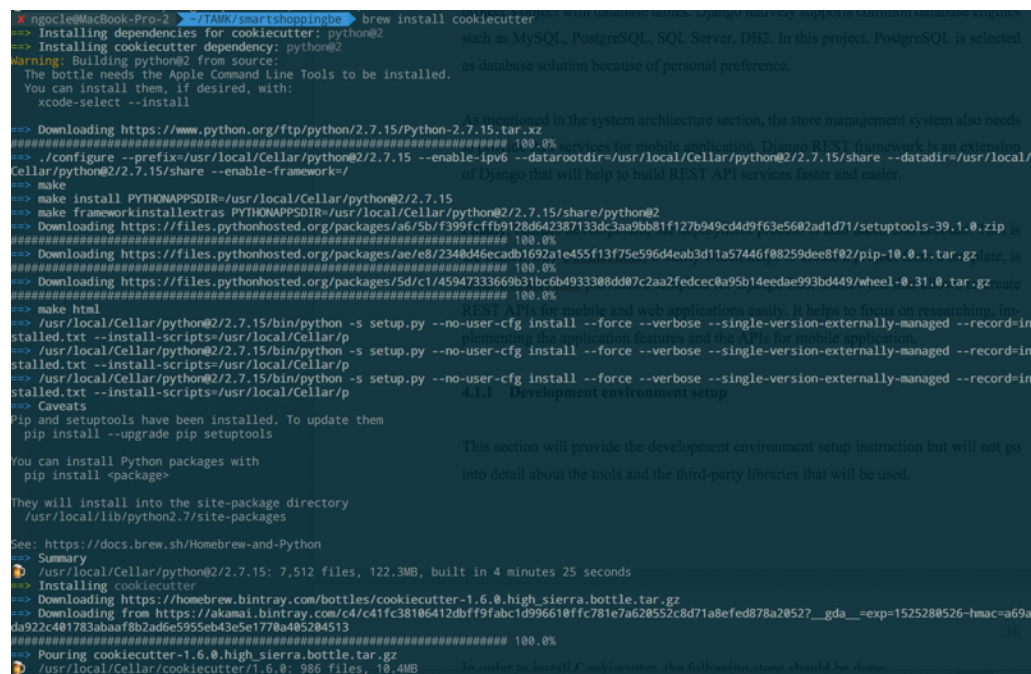
Now, the mobile application features can be started to implementation.

5 SYSTEM DEPLOYMENT AND TESTING

5.1 System deployment

This section will provide the environment setup instruction but will not go into detail about the tools and the third-party libraries that will be used. However, before starting to setup the environment, as Cookiecutter Django Rest requires Python 3.6+ and Docker tool need to be installed. The installation binary files for Docker and Python can be found on their website, so these steps will not be written down here.

First step is using brew to install Cookiecutter.

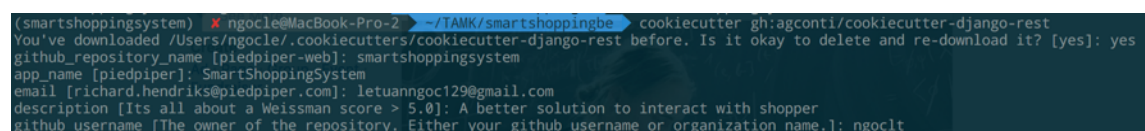


```

ngocle@MacBook-Pro-2 ~/TAMK/smartshoppingbc brew install cookiecutter
--> Installing dependencies for cookiecutter: python@2
--> Installing cookiecutter dependency: python@2
Warning: Building python@2 from sources.
The bottle needs the Apple Command Line Tools to be installed.
You can install them, if desired, with:
  xcode-select --install
--> Downloading https://www.python.org/ftp/python/2.7.15/Python-2.7.15.tar.xz
--> /configure --prefix=/usr/local/Cellar/python@2/2.7.15 --enable-ipv6 --datarootdir=/usr/local/Cellar/python@2/2.7.15/share --enable-framework=/usr/local/Cellar/python@2/2.7.15/share --enable-framework=/usr/local/Cellar/python@2/2.7.15/share --enable-framework=/usr/local/Cellar/python@2/2.7.15/share
--> make
--> make install PYTHONAPPSDIR=/usr/local/Cellar/python@2/2.7.15
--> make frameworkinstallextras PYTHONAPPSDIR=/usr/local/Cellar/python@2/2.7.15/share/python@2
--> Downloading https://files.pythonhosted.org/packages/a6/5b/f399fcfb9128d642387133dc3aa9bb81f127b949cd4d9f63e560ad1d71/setupools-39.1.0.zip
--> Downloading https://files.pythonhosted.org/packages/ae/e8/724046ecadb1692a1e455f13f75e596d4eab3d11a57446f08259dee8f02/pip-10.0.1.tar.gz
--> Downloading https://files.pythonhosted.org/packages/5d/c1/45947333669b31bc6b4933308dd07c2aa2fedcec8a95b14eedae993bd449/wheel-0.31.0.tar.gz
--> make html
--> /usr/local/Cellar/python@2/2.7.15/bin/python -s setup.py --no-user-cfg install --force --verbose --single-version-externally-managed --record=install.txt --install-scripts=/usr/local/Cellar/p
--> /usr/local/Cellar/python@2/2.7.15/bin/python -s setup.py --no-user-cfg install --force --verbose --single-version-externally-managed --record=install.txt --install-scripts=/usr/local/Cellar/p
--> /usr/local/Cellar/python@2/2.7.15/bin/python -s setup.py --no-user-cfg install --force --verbose --single-version-externally-managed --record=install.txt --install-scripts=/usr/local/Cellar/p
--> Caveats
Pip and setuptools have been installed. To update them
  pip install --upgrade pip setuptools
You can install Python packages with
  pip install <package>
They will install into the site-package directory
  /usr/local/lib/python2.7/site-packages
See: https://docs.brew.sh/Homebrew-and-Python
--> Summary
  /usr/local/Cellar/python@2/2.7.15: 7,512 files, 122.3MB, built in 4 minutes 25 seconds
--> Installing cookiecutter
--> Downloading https://homebrew.bintray.com/bottles/cookiecutter-1.6.0.high_sierra.bottle.tar.gz
--> Downloading from https://akamai.bintray.com/c4/c41fc38106412dbff9fabcd1996610ffc781e7a628552c8d71a8efed878a2052?_gda__exp=1525280526-hmac=a69ad922c401783abaa18b2ad6e5955eb43e5e1770a405204513
--> Pouring cookiecutter-1.6.0.high_sierra.bottle.tar.gz
  /usr/local/Cellar/cookiecutter/1.6.0: 986 files, 10.4MB
  
```

PICTURE 4: Cookiecutter installation

After finished installing Cookiecutter, next step is creating the Django Rest project from the boilerplate (on Github). The command-line utility will need some input for project specification such as project name, git repository name.



```

(smartshoppingsystem) * ngocle@MacBook-Pro-2 ~/TAMK/smartshoppingbc cookiecutter gh:agconti/cookiecutter-django-rest
You've downloaded /Users/ngocle/.cookiecutters/cookiecutter-django-rest before. Is it okay to delete and re-download it? [yes]: yes
github_repository_name [piedpiper-web]: smartshoppingsystem
app_name [piedpiper]: SmartShoppingSystem
email [richard.hendriks@piedpiper.com]: letuanguoc129@gmail.com
description [Its all about a Weissman score > 5.0]: A better solution to interact with shopper
github_username [The owner of the repository. Either your github username or organization name.]: ngocle
  
```

PICTURE 5: Creating the project from boilerplate

Cookiecutter will automatically generate a folder with the project name as entered and copy all of the source code in the project template into this new project folder.

TABLE 11. Project directory structure description

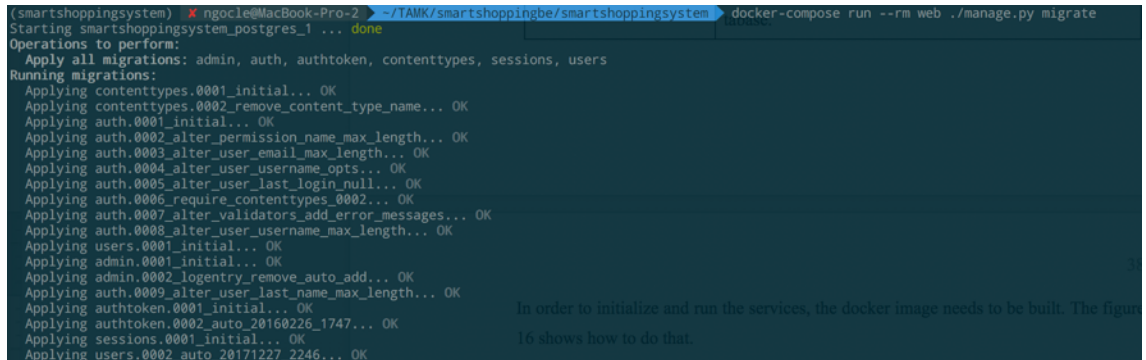
File/Folder	Description
Dockerfile	Is a script, composed of various commands that is used for organizing things and greatly help with deployments by simplifying the process start-to-finish
README.md	Contains instruction shows how to run the local development and how to deploy the code.
SmartShoppingSystem	Contains all of the source code of system.
docker-compose.yml	Docker Compose YAML file, used to configure application's services.
docs	Contains project's document. In this research project, this folder is not used.
manage.py	Is Django's command-line utility for administrative tasks.
mkdocs.yml	Is YAML file contains project settings of MkDocs that is used to build project documentation.
requirements.txt	A text file contains a list of libraries to be installed and used in the project.
setup.cfg	Provides all of a Python distribution's metadata and build configuration
wait_for_postgres.py	Is python code file that is used to connect to Postgres database.

To initialize and run the services, the docker image needs to be built. The picture 6 shows how to do that.

```
(smartshoppingsystem) * ngocle@MacBook-Pro-2 ~/TAMK/smartshoppingbe/smartshoppingsystem: docker-compose build
Building documentation
Step 1/8 : FROM python:3.6.413
--> 6bf7a4fa2d45
Step 2/8 : ENV PYTHONUNBUFFERED 1
--> Using cache
--> c59e2bfc8963
Step 3/8 : COPY ./requirements.txt requirements.txt
--> Using cache
--> 90f4c326db71
Step 4/8 : RUN pip3 install -r requirements.txt
--> Running in 3d765422777c
```

PICTURE 6: Build the Dockerfile

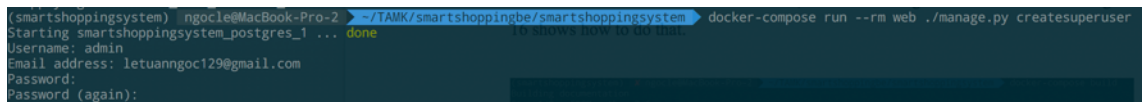
After this step, the needed packages such as Django, Django Rest Framework, PostgreSQL will be installed. However, PostgreSQL still needs to be migrated so all of database table will be initialized. This migration command has to be run every time there is any modification in application model code, then it can be synchronized with the database.



```
(smartshoppingsystem) ngocle@MacBook-Pro-2 ~ -/TAMK/smartshoppingbe/smartshoppingsystem docker-compose run --rm web ./manage.py migrate
Starting smartshoppingsystem_postgres_1 ... done
Operations to perform:
  Apply all migrations: admin, auth, authtoken, contenttypes, sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying users.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying authtoken.0001_initial... OK
  Applying authtoken.0002_auto_20160226_1747... OK
  Applying sessions.0001_initial... OK
  Applying users.0002_auto_20171227_2246... OK
```

PICTURE 7: Migrate database for application

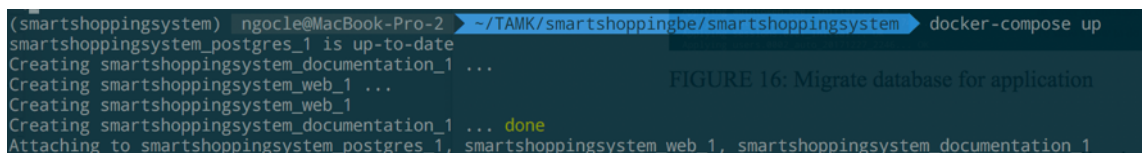
In order to access to our web application (which will be Store Management System), a super admin account will need to be created. This account data will be saved in the PostgreSQL which is created in previous step.



```
(smartshoppingsystem) ngocle@MacBook-Pro-2 ~ -/TAMK/smartshoppingbe/smartshoppingsystem docker-compose run --rm web ./manage.py createsuperuser
Starting smartshoppingsystem_postgres_1 ... done
Username: admin
Email address: letuannoc129@gmail.com
Password:
Password (again):
```

PICTURE 8: Create super admin account for Store Management System

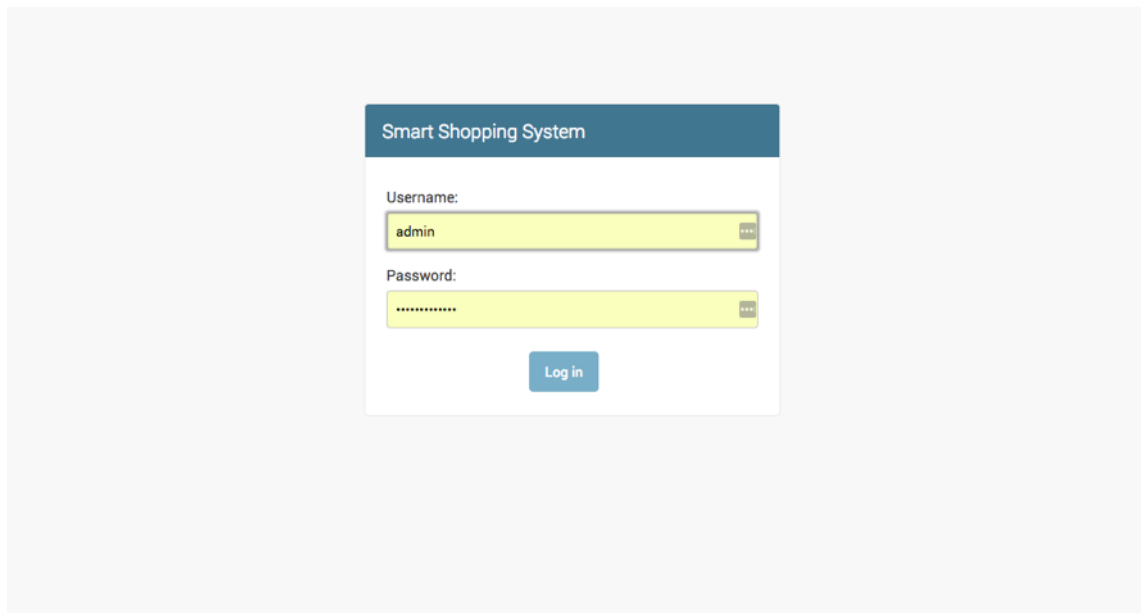
Now the web application can be run locally on the development environment with Docker containers.



```
(smartshoppingsystem) ngocle@MacBook-Pro-2 ~ -/TAMK/smartshoppingbe/smartshoppingsystem docker-compose up
smartshoppingsystem_postgres_1 is up-to-date
Creating smartshoppingsystem_documentation_1 ...
Creating smartshoppingsystem_web_1 ...
Creating smartshoppingsystem_web_1
Creating smartshoppingsystem_documentation_1 ... done
Attaching to smartshoppingsystem_postgres_1, smartshoppingsystem_web_1, smartshoppingsystem_documentation_1
```

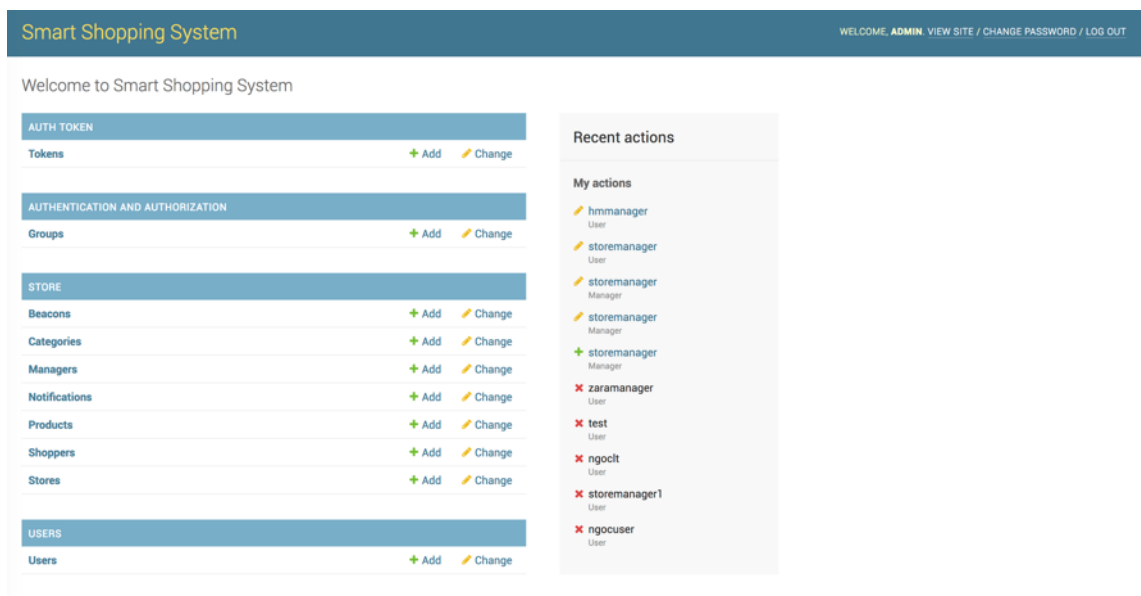
PICTURE 9: Run the system locally with development setting

As stated earlier, Django framework generates an admin panel which allows to manage the content of the application. It can be accessed via address <http://localhost:8000/admin/>, super user account needs to be input to be able to login.



PICTURE 10: Login page

After logged in successfully, the admin dashboard should be displayed as picture 12. This page will be different depends on the user type, admin or store manager. User won't be able to login to this system.



PICTURE 11: Super Admin's dashboard page

5.2 System testing

The goal of this testing is not only to point out and fix the bugs in the system but also to see how the system works in the reality, because all of the system functionalities are designed based on my researching and my working experience. The testing will be divided

into 2 sections for store management system application and mobile application. Since there are 2 user roles will be using store management system, the testing for this web application will be split into 2 small section for each user type, super admin and store manager.

5.2.1 System testing with super admin

As a super admin of the system, he/she will be able to manage all of the content in the system. After logged in successfully, admin panel for super admin will be opened (which is shown in picture 11).

From this page, super admin can access and manage the stores, the managers of stores and their categories, their products. And as described above, super admin will be the one creates store and the manager for that store. So that manager can manage his/her store later on.

The screenshot shows the 'Add store' page in the Smart Shopping System. The page has a dark blue header with the text 'Smart Shopping System' on the left and 'WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT' on the right. Below the header is a breadcrumb trail: 'Home > Store > Stores > Add store'. The main content area is titled 'Add store' and contains the following form elements:

- Name:** A text input field with a small icon on the right.
- Address:** A text input field.
- Phone:** A text input field.
- Manager:** A dropdown menu showing three options: 'hmmanager', 'ngocle', and 'storemanager'. A green plus sign is visible to the right of the dropdown.
- Cover:** A file upload section with a 'Choose File' button and the text 'No file chosen'.
- Is active:** A checkbox that is currently checked.

At the bottom of the form, there are three buttons: 'Save and add another', 'Save and continue editing', and 'SAVE'.

PICTURE 12: Add new store page

The picture 12 above shows the form which allows super admin to create a new store. Admin can select the exist manager or click add new manager button to open create manager form and create a new manager for this store. The picture 13 shows create manager form. Beside filling the information of the manager, the super admin also needs to add the permission for the manager to allows he/she to manage the content in his/her store.

These permissions can be manually added in permission selection box which is shown in picture 14.

Smart Shopping System WELCOME, ADMIN VIEW SITE / CHANGE PASSWORD / LOG OUT

Home · Store · Managers · storemanager2

Change Manager HISTORY

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **No password set.**
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Permissions

Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

PICTURE 13: Add new store manager page

User permissions:

Available user permissions

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
authToken | Token | Can add Token
authToken | Token | Can change Token
authToken | Token | Can delete Token

Chosen user permissions

store | beacon | Can add beacon
store | beacon | Can change beacon
store | beacon | Can delete beacon
store | category | Can add category
store | category | Can change category
store | category | Can delete category
store | notification | Can add notification
store | notification | Can change notification
store | notification | Can delete notification
store | product | Can add product
store | product | Can change product
store | product | Can delete product

Important dates

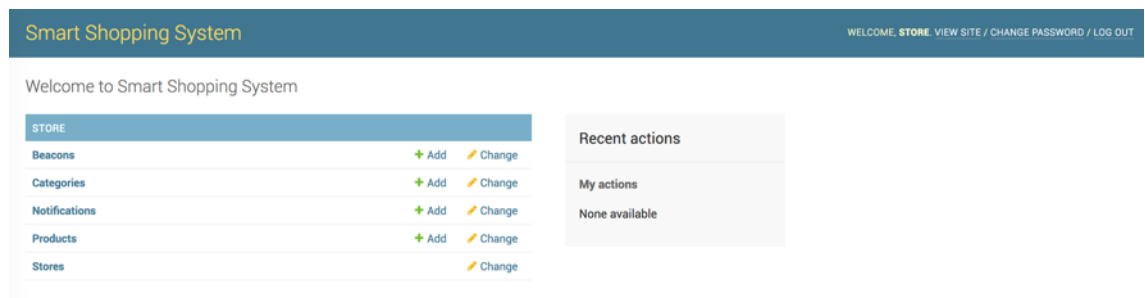
Last login: Date: 2018-06-18 Today
Time: 13:17:15 Now
Note: You are 3 hours ahead of server time.

Date joined: Date: 2018-06-18 Today
Time: 13:17:05 Now
Note: You are 3 hours ahead of server time.

PICTURE 14: User permission selection box

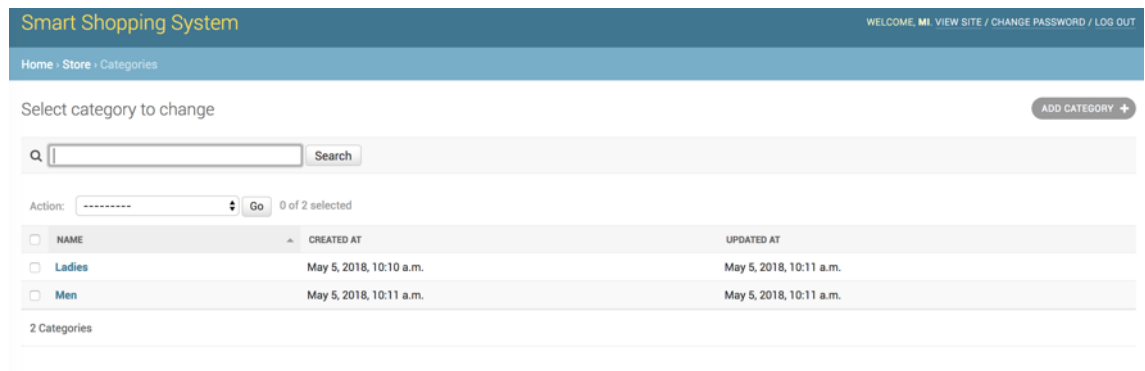
5.2.2 System testing with store manager

After having the login information from super admin, the store manager can login to the system (and change his/her password). The admin panel (dashboard page) for store manager has the same user interface as for super admin. However, it only displays management features for the store that he/she is assigned to.

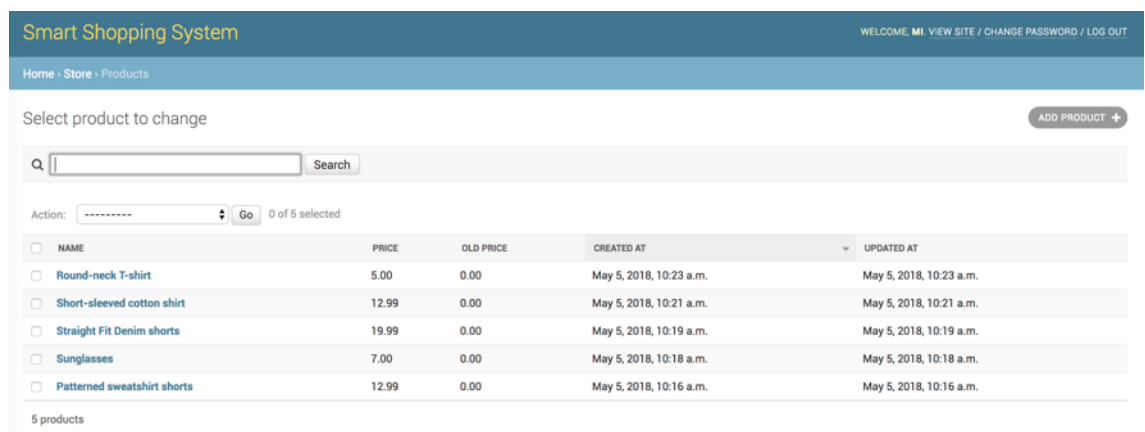


PICTURE 15: Store manager's dashboard page

From this dashboard, admin can access to other subpages to manage his/her store such as: beacon management, notification management, category management, product management. He/she can't access or modify the content of the other stores.

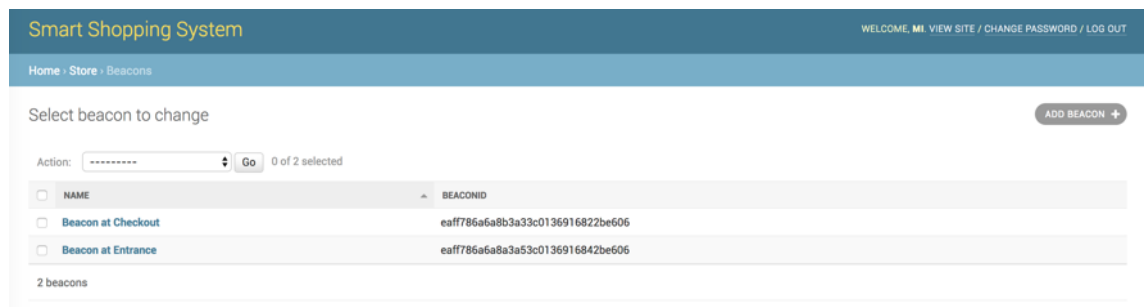


PICTURE 16: Category management page



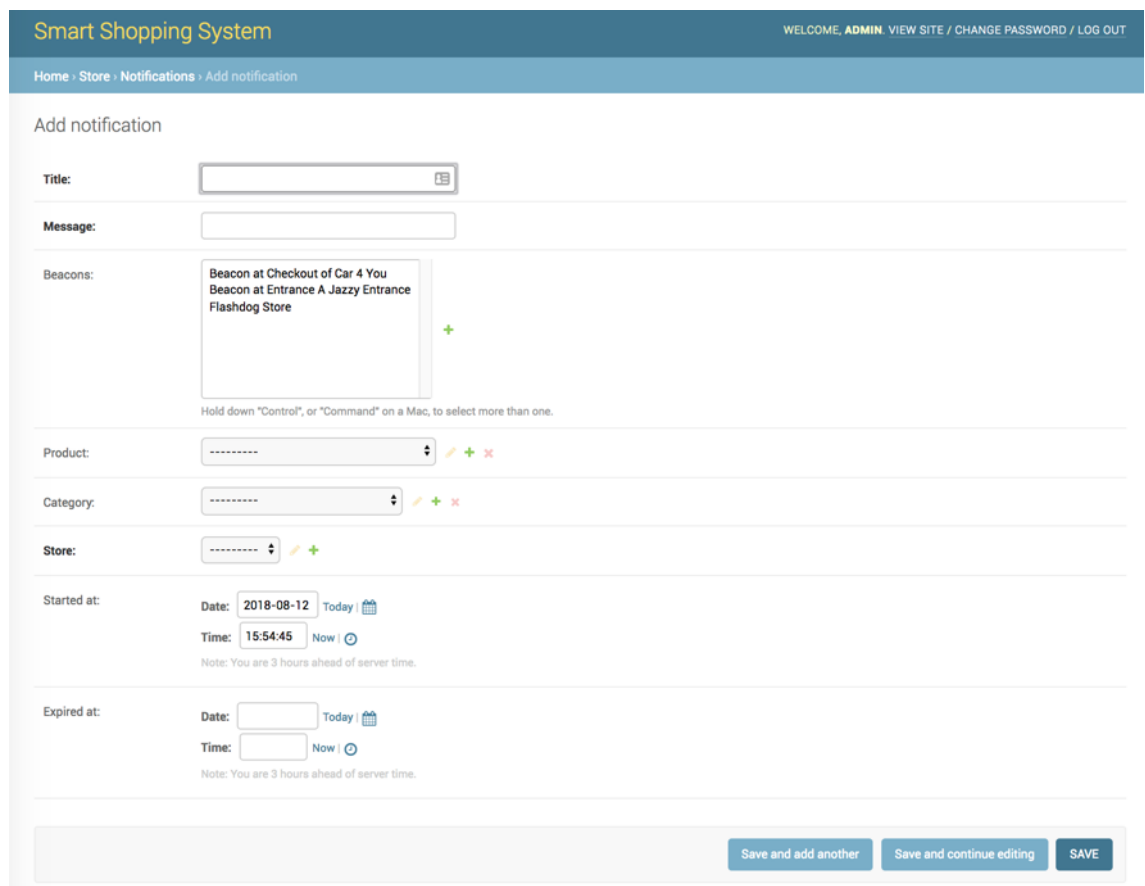
PICTURE 17: Product management page

In order to trigger the notification service and push the new deals to shopper, the beacon devices need to be installed at the store and their IDs have to be added into the system.



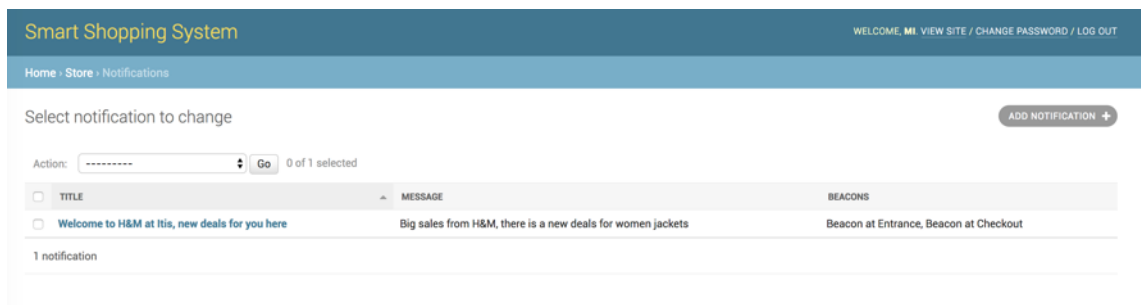
PICTURE 18: Beacon management page

After added the beacons to the systems, now it's able to create the notifications for those beacons. The beacon can be attached into the notification. Besides that, the notification also can be specified for only a product or a category. This allows the shoppers can view the product detail or the whole products in the category after receiving this notification.



PICTURE 19: Notification creation page

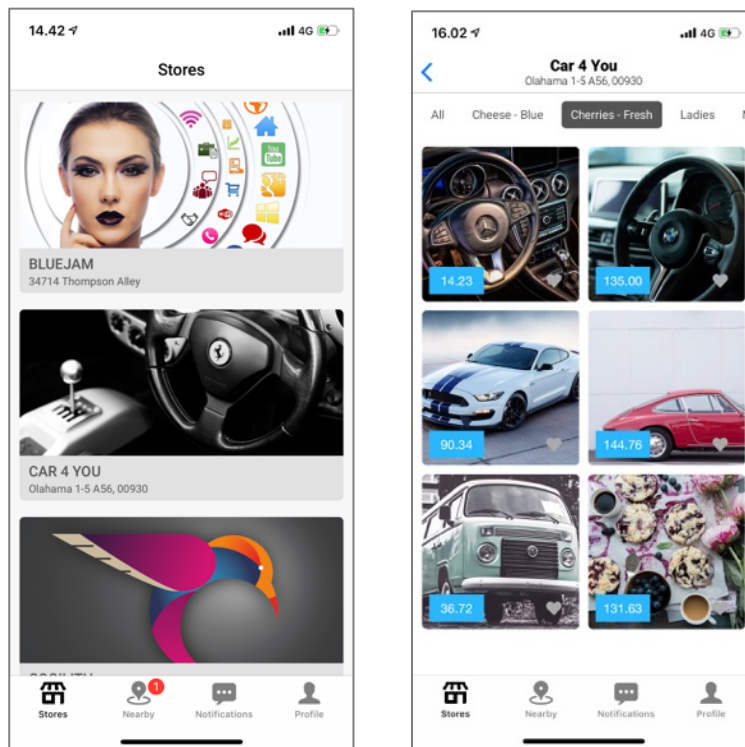
After created the notification successfully, the notification will be shown in the notification management page as shown in picture 20.



PICTURE 20: Notification management page

With these notification settings, every time the shopper's device is in range of beacon, the mobile application will send a request with the beacon ID to Smart Shopping System to inform that there is a user is close to that beacon. The system will query from database to get the information of the beacon and the notifications which are attached to that beacon and then send back all of those notifications to user's device. Finally, user's mobile device will notify user.

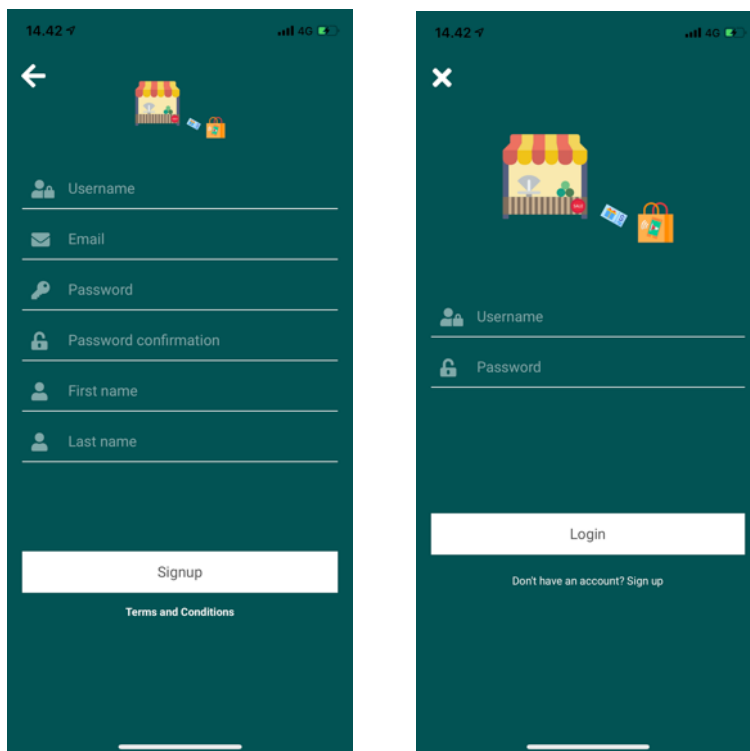
5.3 Mobile application testing



PICTURE 21 & 22: Store list and store detail screens

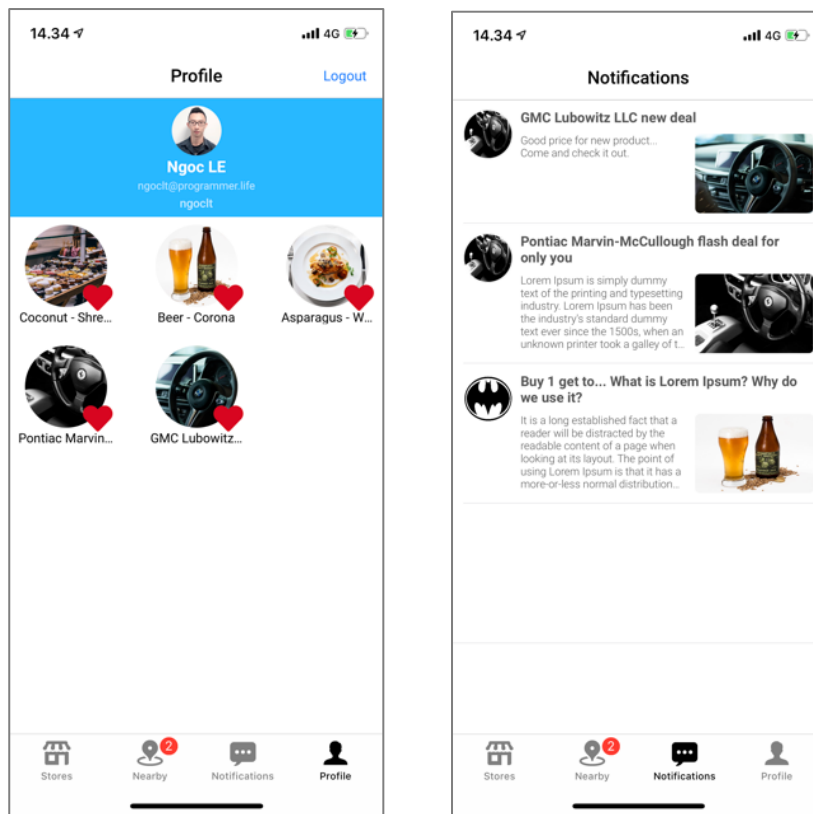
As a shopper, user can use mobile application to check the products on the stores as well as new deal notifications. Without logging in, the shoppers can view the stores and their products only. In the store tab, the shopper can view all of the stores in the system, and they can click on the store to open the store detail screen. From the store screen, the shopper can filter the products list based on the categories and like / unlike the products to add them to their interest collection.

In order to receive the notifications from the stores ever time there is new deal or offer for the products that they are interested in, they will need to register account and login.



PICTURE 23 & 24: Registration and login screens on app

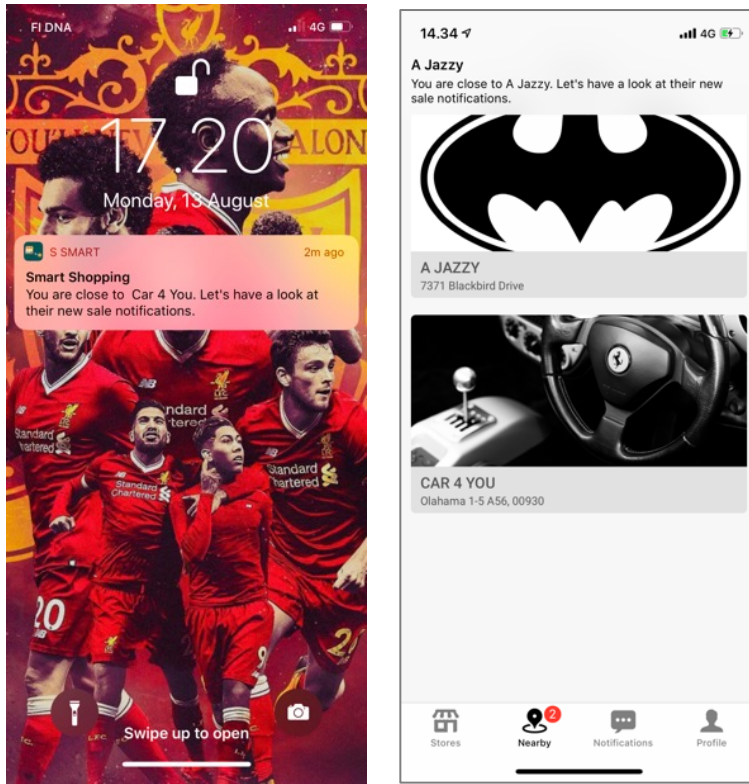
After logged in, the shoppers can like / unlike the products and categories in the system. Their liked (interested) products and categories will be shown in the profile screen. From here, the shopper also can unlike the product / category to remove the interest.



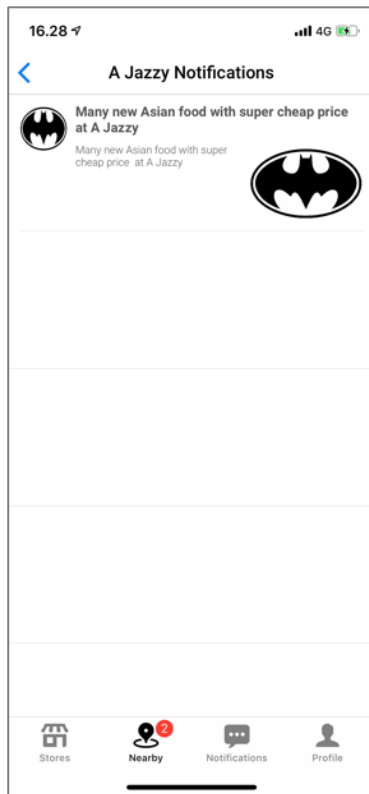
PICTURE 25 & 26: Shopper profile and notifications screens.

Based on these interests, the server will send the notifications to the shopper every time there is new promotion, offer or notification. These notifications will be shown in the notification tab on mobile application. Each notification will be linked to the product or the category and the store, the shopper can click to the notification to see the product or category information in the store detail screen.

As stated above, one of the most important features of this mobile application is receiving the notification in real time, when the shopper is close to the store. Either the shopper is opening the app or not, the application will notify them. The picture 27 and 28 show the notification on the lock screen and the notification which is presented in the app. And at the same time, the app will show the nearby stores in the nearby tab. From here, user can open and check these stores deals by selecting them to open the store notification screen.



PICTURE 27 & 28: Nearby store notification and nearby store list screen.



PICTURE 29: Nearby store notifications screen

6 NEXT STEP

Bluetooth beacon technology is quickly gaining momentum and paving the way for efficient and seamless solutions across multiple domains nowadays. It is being used in many sectors to improve user experience. Sending personalized promotion / deal notifications to shoppers who are nearby the store is just one of many applications which helps to grow retail business. This is the first step that will bring more shopper come to the store.

However, navigating in a big store and shopping mall would be a big problem to shopper. Although many places provide information screens which allow shopper to search the store and see how they can go there, it's still difficult. Leading shopper by mobile application that supports indoor navigation for sure will resolve this problem. That's how beacon technology becomes the best solution in this case, since it can be integrated and inside these store and shopping mall, the signals from these beacon devices will allow shopper's mobile phone knows where they are and navigates them to the place they need with indoor map. And shoppers will easily be navigated right to the store or the shelf to find their needs.



PICTURE 30: Indoor navigation in shopping mall, source: leantegra.com

7 DISCUSSION

Smart Shopping System will be good solution for not only store but also for super market and shopping mall. It provides a solution to deliver a personalized user experience to shopper. It collects user preferences and shopping habits, and then with the assistance from beacon devices, the system will engage shopper via sale, promotions notifications and then make shopping more interactive.

One of challenges of this system is not only about the beacon devices price but also the cost of on-going maintenance for both hardware and software part. However, with the amazing growth of IoT device, especially is Arduino, it makes building these kinds of beacon devices so simple, that's reason why these devices are very cheap today. And it will be even cheaper in the future. Some of popular brands who offers reasonable and affordable beacon devices are Estimote, Kontakt, Gimbal. These brands not only provide the beacon devices but also, they will come with a cloud service which allows to manage these beacons. However, these cloud services are not developed for any specific use, that's reason why another extra system for store management is still needed.

REFERENCES

1. What is Bluetooth, Bluetooth. <https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works>
2. Base knowledge about Bluetooth. <https://iotbreaks.com/base-knowledge-about-bluetooth/>
3. Understanding the different types of BLE Beacons. <https://os.mbed.com/blog/entry/BLE-Beacons-URIBeacon-AltBeacons-iBeacon/>
4. BLE Eddystone Beacon Service, https://os.mbed.com/teams/Bluetooth-Low-Energy/code/BLE_EddystoneBeacon_Service/
5. Eddystone protocol specification <https://github.com/google/eddystone/blob/master/protocol-specification.md>
6. INTRO TO EDDYSTONE – GOOGLE LATEST PROXIMITY BEACON PROTOCOL, OneThesis. <https://www.onethesis.com/2016/01/10/intro-to-eddystonetm-google-latest-proximity-beacon-protocol/>
7. Beacons: all you need to know about them, Appfutura, <https://www.appfutura.com/blog/beacons-all-you-need-to-know-about-them/>
8. Use case, Wikipedia. https://en.wikipedia.org/wiki/Use_case
9. Class diagram, Wikipedia. https://en.wikipedia.org/wiki/Class_diagram
10. Indoor Navigation: Why would Shopping Malls Deploy it? <https://leante-gra.com/blog/indoor-navigation-why-would-shopping-malls-deploy-it>

APPENDICES

Appendix 1. forms.py of store application

```

from django import forms
from .models import Manager, Shopper, Product

from django.contrib.auth.forms import ReadOnlyPasswordHashField

class ProductAdminForm(forms.ModelForm):
    """ ModelForm class to validate product instance data before saving from
    admin interface """

    class Meta:
        model = Product
        fields = '__all__'

    def clean_price(self):
        if self.cleaned_data['price'] <= 0:
            raise forms.ValidationError('Price supplied must be greater than
zero.')
        return self.cleaned_data['price']

class ShopperForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput)

    class Meta:
        model = Shopper
        fields = ('first_name',
                 'last_name',
                 'email',
                 'password',
                 'address_line',
                 'telephone',
                 'zip_code',
                 'state',
                 'country')

class ManagerAdminCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the required
    fields, plus a repeated password."""
    password1 = forms.CharField(label='Password', widget=forms.PasswordInput)
    password2 = forms.CharField(label='Password confirmation',
widget=forms.PasswordInput)

    class Meta:
        model = Manager
        fields = ('username', 'email', 'password', 'first_name', 'last_name',
'gender', 'avatar')

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
            raise forms.ValidationError("Passwords don't match")
        return password2

    def save(self, commit=True):
        # Save the provided password in hashed format
        user = super(ManagerAdminCreationForm, self).save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        if commit:

```

```

        user.save()
    return user

class ManagerAdminChangeForm(forms.ModelForm):
    """A form for updating users. Includes all the fields on
    the user, but replaces the password field with admin's
    password hash display field.
    """
    password = ReadOnlyPasswordHashField(label="Password",
                                         help_text="Raw passwords are not
stored, so there is no way to see "
                                         "this user's password, but
you can change the password "
                                         "using <a href=\"../pass-
word/\">this form</a>.")

    class Meta:
        model = Manager
        fields = ('username', 'email', 'password', 'first_name', 'last_name',
'gender', 'avatar')

    def clean_password(self):
        # Regardless of what the user provides, return the initial value.
        # This is done here, rather than on the field, because the
        # field does not have access to the initial value
        return self.initial["password"]

```


Appendix 2. admin.py of store application

```

# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from copy import deepcopy

from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from .models import Manager, Shopper, Store, Product, Category
from .forms import ProductAdminForm, ManagerAdminCreationForm, ManagerAdmin-
ChangeForm

# Register your models here.

class ShopperAdmin(admin.ModelAdmin):

    form = ManagerAdminChangeForm
    add_form = ManagerAdminCreationForm

    list_display_links = ('username', 'first_name', 'last_name', 'email')
    list_display = ('username', 'first_name', 'last_name', 'email')
    list_per_page = 50
    search_fields = ['first_name', 'last_name', 'username', 'email']
    ordering = ['username']

admin.site.register(Shopper, ShopperAdmin)

class ManagerAdmin(BaseUserAdmin):

    form = ManagerAdminChangeForm
    add_form = ManagerAdminCreationForm

    list_display_links = ('username', 'first_name', 'last_name', 'email')
    list_display = ('username', 'first_name', 'last_name', 'email')
    list_per_page = 50
    search_fields = ['first_name', 'last_name', 'username', 'email']
    ordering = ['username']

admin.site.register(Manager, ManagerAdmin)

class StoreAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone', 'joined_at',)
    list_display_links = ('name',)
    list_per_page = 50
    search_fields = ['name', 'address']
    ordering = ['name']

    def get_fieldsets(self, request, obj=None):
        """Custom override to exclude fields"""
        fieldsets = deepcopy(super(StoreAdmin, self).get_fieldsets(request,
obj))

        # Append excludes here instead of using self.exclude.
        # When fieldsets are defined for the user admin, so self.exclude is
ignored.
        exclude = ()

        if not request.user.is_superuser:
            exclude += ('manager', 'is_active')

        # Iterate fieldsets

```

```

        for fieldset in fieldsets:
            fieldset_fields = fieldset[1]['fields']

            # Remove excluded fields from the fieldset
            for exclude_field in exclude:
                if exclude_field in fieldset_fields:
                    fieldset_fields = tuple(field for field in fieldset_fields
if field != exclude_field) # Filter
                    fieldset[1]['fields'] = fieldset_fields # Store new tuple

            return fieldsets

    def get_queryset(self, request):
        qs = super(StoreAdmin, self).get_queryset(request)
        if request.user.is_superuser:
            return qs

        return qs.filter(manager=request.user)

admin.site.register(Store, StoreAdmin)

class ProductAdmin(admin.ModelAdmin):
    form = ProductAdminForm
    # sets values for how the admin site lists your products
    list_display = ('name', 'price', 'old_price', 'created_at', 'updated_at',)
    # which of the fields in 'list_display' tuple link to admin product page
    list_display_links = ('name',)
    list_per_page = 50
    ordering = ['-created_at']
    search_fields = ['name', 'description', 'meta_keywords', 'meta_descrip-
tion']
    exclude = ('created_at', 'updated_at',)
    # sets up slug to be generated from product name
    prepopulated_fields = {'slug': ('name',)}

    def formfield_for_manytomany(self, db_field, request, **kwargs):
        if not request.user.is_superuser and db_field.name == "categories":
            kwargs['queryset'] = Category.objects.filter(
                store__in=Store.objects.filter(manager=request.user)
            )

        return super().formfield_for_manytomany(db_field, request, **kwargs)

    def get_queryset(self, request):
        qs = super(ProductAdmin, self).get_queryset(request)
        if request.user.is_superuser:
            return qs

        # get products in store that manager is manage
        return qs.filter(categories__store__in=Store.objects.filter(man-
ager=request.user))

admin.site.register(Product, ProductAdmin)

class CategoryAdmin(admin.ModelAdmin):
    # sets up values for how admin site lists categories
    list_display = ('name', 'created_at', 'updated_at',)
    list_display_links = ('name',)
    list_per_page = 20
    ordering = ['name']
    search_fields = ['name', 'description', 'meta_keywords', 'meta_descrip-
tion']
    exclude = ('created_at', 'updated_at',)

    # sets up slug to be generated from category name

```

```
prepopulated_fields = {'slug': ('name',)}

def formfield_for_foreignkey(self, db_field, request, **kwargs):
    if not request.user.is_superuser and db_field.name == "store":
        kwargs['queryset'] = Store.objects.filter(manager=request.user)

    return super().formfield_for_foreignkey(db_field, request, **kwargs)

def get_queryset(self, request):
    qs = super(CategoryAdmin, self).get_queryset(request)
    if request.user.is_superuser:
        return qs

    # get products in store that manager is manage
    return qs.filter(store__in=Store.objects.filter(manager=request.user))

admin.site.register(Category, CategoryAdmin)
```

Appendix 3. Podfile

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'SmartShopping' do
  # Comment the next line if you're not using Swift and don't want to
  use dynamic frameworks
  use_frameworks!

  # Pods for SmartShopping
  pod 'Alamofire', '~> 4.5'
  pod 'Material', '~> 2.12'
  pod 'EstimoteSDK'

  target 'SmartShoppingTests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'SmartShoppingUITests' do
    inherit! :search_paths
    # Pods for testing
  end
end
```

Appendix 4. Source code repositories

Mobile Application source code: <https://github.com/ngoctl/smartshoppingios>

Smart Shopping System source code: <https://github.com/ngoctl/smartshoppingsystem>