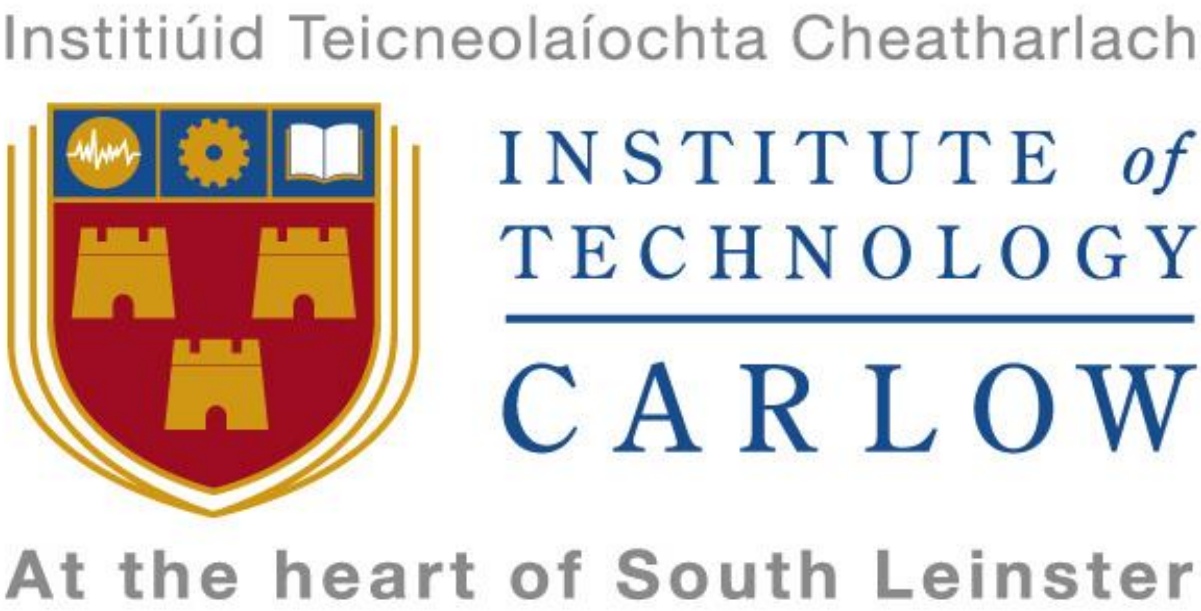
Design Document



Project: Smart Shopping Application

Student: Jason O'Hara

Student Number: C00168956

Supervisor: Paul Barry

Contents

1. Introduction

1.1 Smart Shopping Application

The Smart Shopping Application is a cross-platform, mobile shopping list application. Its key features are:

- Allow a user to create a shopping list in a quick and easy manner.
- Allow a user to save this list to a cloud database
- Allow the user to add information regarding the location of a product in a given shop to a cloud database.
- Allow a user to sort a shopping list based on the location of products in a given shop.

The functionality that separates this application from the many other basic shopping list applications available is the list-sorting functionality. This uses crowd-sourced data to maintain a database describing the location of products across a selection of different shops and supplies users with the ability to integrate this information directly into their personal shopping list.

The intention is to develop this application in a cross-platform manner with a particular focus on mobile platforms. This application will be developed across three iterations from October 2015 until April 2016 by a single developer.

1.2 Purpose of This Document

The purpose of this document is to give an overview of the design of this application on a technical level, broken down on a by-iteration basis. It will do this with the use of sequence-style diagrams to give a system-wide overview of the processes and pseudo-code to give a more detailed description of the logic involved as needed. Many processes have had low-level details simplified for the sake of clarity.

The functionality of this application will be described in greater technical detail in the Functional Specification of this project, the results of the all research done will be discussed in the Research Document and the overall results of this project will be discussed in detail in the Final Project Report.

2. Overview

2.1 System Overview

The basic layout of the system involves a Python application running on the client device. This application has been developed using Kivy and Buildozer in order to be cross platform. The Python application communicates with a web application backend hosted on PythonAnywhere

via HTTP requests. This backend is also written in Python and interacts with the PythonAnywhere MySQL database where all cloud information is stored.

The below diagram summarises the system architecture using Android (the main target platform) as an example.

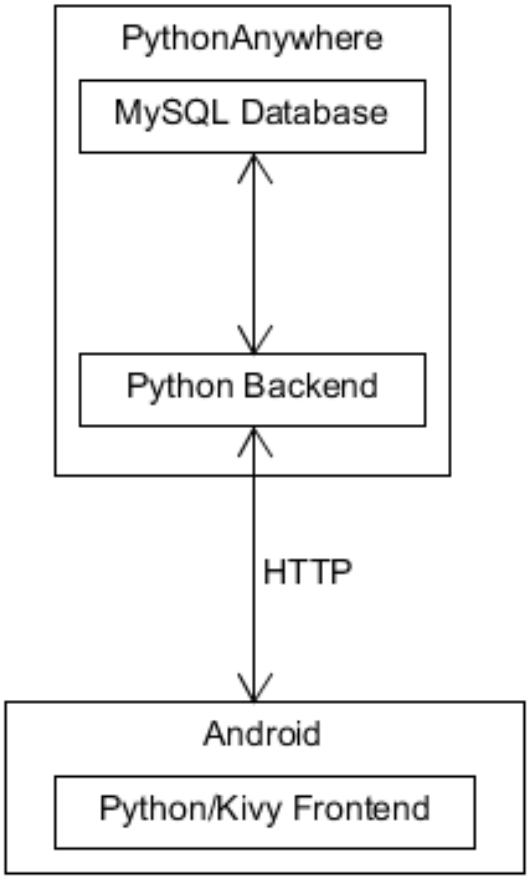


Fig 2.1 System Architecture Overview

2.2.Frontend Overview

This section is an overview of the architecture used for the frontend client application.

2.2.1 Scope

The frontend application performs the following tasks:

- Provides an easy-to-use interface for the user to interact with the system.
- Sends information and/or requests to the backend web application.
- Receives and parses information received from the backend web application.
- Stores some information locally between syncing or saving to the cloud.

2.2.2 Kivy

Kivy is core to the structure of all frontend code. Kivy uses a hierarchical structure to order “Widgets” (UI Elements) in an intuitive way.

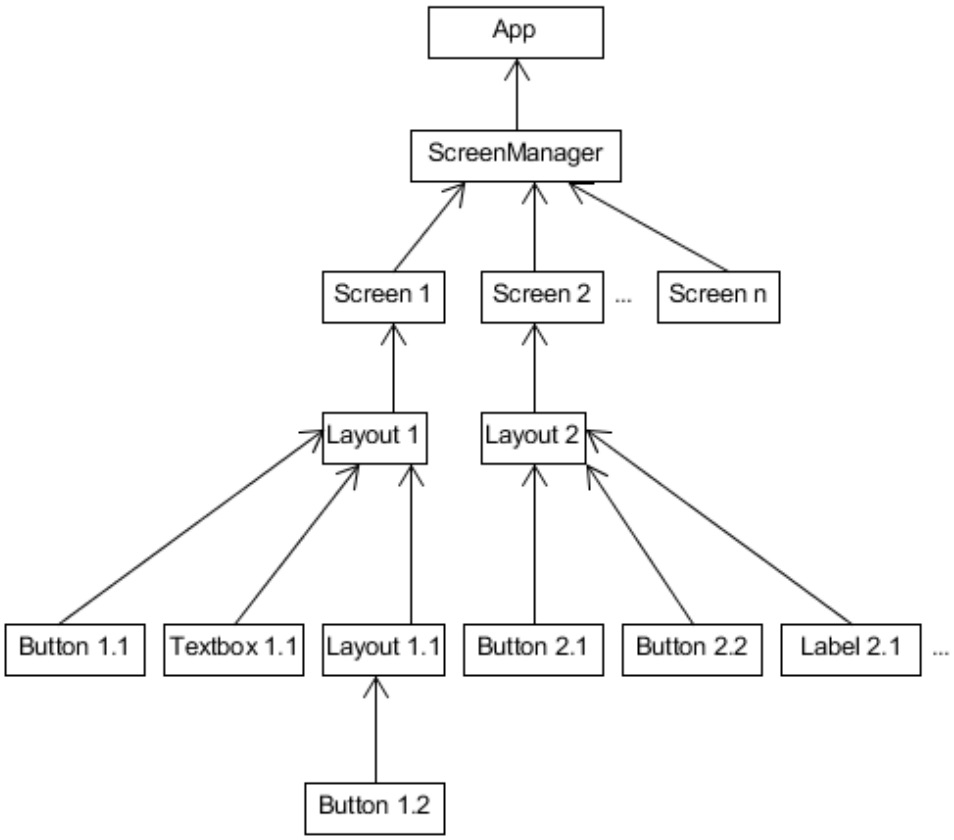


Fig 2.2 A general description of the Kivy widget hierarchy of this project.

In the above diagram the App widget is the root of all other widgets. It can be referenced from any of the other widgets directly and stores global variables such as the username of the user currently logged in and their secret key.

The ScreenManager widget manages which screen is currently visible, the transitions between screens and any processes that involve the creation of screens or the altering of multiple screens at once.

Each Screen widget represents a distinct view the user may have of the application at any one time. There are 12 default screens, with more generated upon start-up. Screen widgets generally have only one direct child widget, a layout widget. This widget defines the manner in which its own child widgets are placed on the screen. These Layout widgets may be nested in

order to define different layouts for different areas on the screen. Each Screen/Layout then has many basic UI elements as child widgets such as buttons, textboxes, labels etc.

Each widget has direct access to its parent, its children and the root widget. In this way widgets may alter and respond to the states of other widgets in the hierarchy.

These widgets, their properties and their behaviour are defined mainly within two .kv files. These files are written in a special Kivy mark-up language with a nested structure that makes it easier to store and manipulate the UI design of the application quickly. *Main.kv* stores the details of each Screen owned by the ScreenManager. *BasicWidgets.kv* stores the details of more low-level widgets such as specific buttons, labels etc.

Widgets may also be created and inserted into this hierarchy at runtime. This is the done in situations such as adding a new item to a shopping list or creating the product/shop menus upon start-up.

2.3 Backend Overview

This section is an overview of the architecture used for the backend server application hosted in the cloud.

2.3.1 Web Application

This is a Python Flask application hosted on PythonAnywhere. It is accessed by the client via sending GET and POST requests to various different URLs. The web application then handles all interaction with the database and returns values to the client application. This application also handles all requests made via the administration website.

2.3.2 Administration Website

This is a simple HTTP website designed for administrative use. It is hosted on PythonAnywhere and managed by the Flask web application. It provides a second interface to many of the web application functions for manipulating database information.

2.3.3 Database Design

The MySQL database supplied as part of the PythonAnywhere service is where all cloud data is stored. It is manipulated by the Flask application based on requests from the client application or the administration website. Below is a brief description of each table and its structure.

ADMINS
This table contains information about the administrators who may use the administration website.

Column	Value	Description
USERNAME	varchar	An administrator's username
PASS	varchar	An administrator's password (encrypted)

LISTS
This table contains the current list for each user, the one last synced with the cloud.

Column	Value	Description
USER	varchar	A user's username
PRODUCT	varchar	A product on a user's current list
QUANTITY	int	The quantity of said product on a user's current list

PRODUCTS
This table contains a categorised list of products that users may chose from for their shopping lists.

Column	Value	Description
PRODUCT	varchar	The name of a product
CATEGORY	varchar	The category of a product

SAVED_LISTS
This table contains all lists users have saved using the "My Lists" functionality.

Column	Value	Description
USER	varchar	A user's username
LIST_NAME	varchar	The name of a user's saved list
PRODUCT	varchar	A product on one of a user's saved lists
QUANTITY	int	The quantity of said product on one of a user's saved lists

SAVED_SHOPS
This table contains the IDs of all shops users have saved using the “My Shops” functionality.

Column	Value	Description
USER	varchar	A user’s username
SHOP_ID	varchar	The ID of one of a user’s saved shops

SHOPS
This table contains layout information for each shop.

Column	Value	Description
SHOP_ID	varchar	A shop’s ID
AISLE	int	A number of an aisle within a shop
AISLE_DESCRIPTION	varchar	A description of an aisle within a shop
PRODUCT	varchar	A product on said aisle

SHOP_DETAILS
This table contains general information for each shop.

Column	Value	Description
SHOP_ID	varchar	A shop’s ID
SHOP_NAME	varchar	A shop’s name
LOCATION	varchar	A shop’s geographic location

USERS

This table contains information about each user. The SECRET_KEY field refers to a key generated for each user that is given to them when they login and used for authentication when making requests to the server.

Column	Value	Description
USER	varchar	A user's username
PASSWORD	varchar	A user's password (encrypted)
SECRET_KEY	varchar	A user's secret key

3. Design

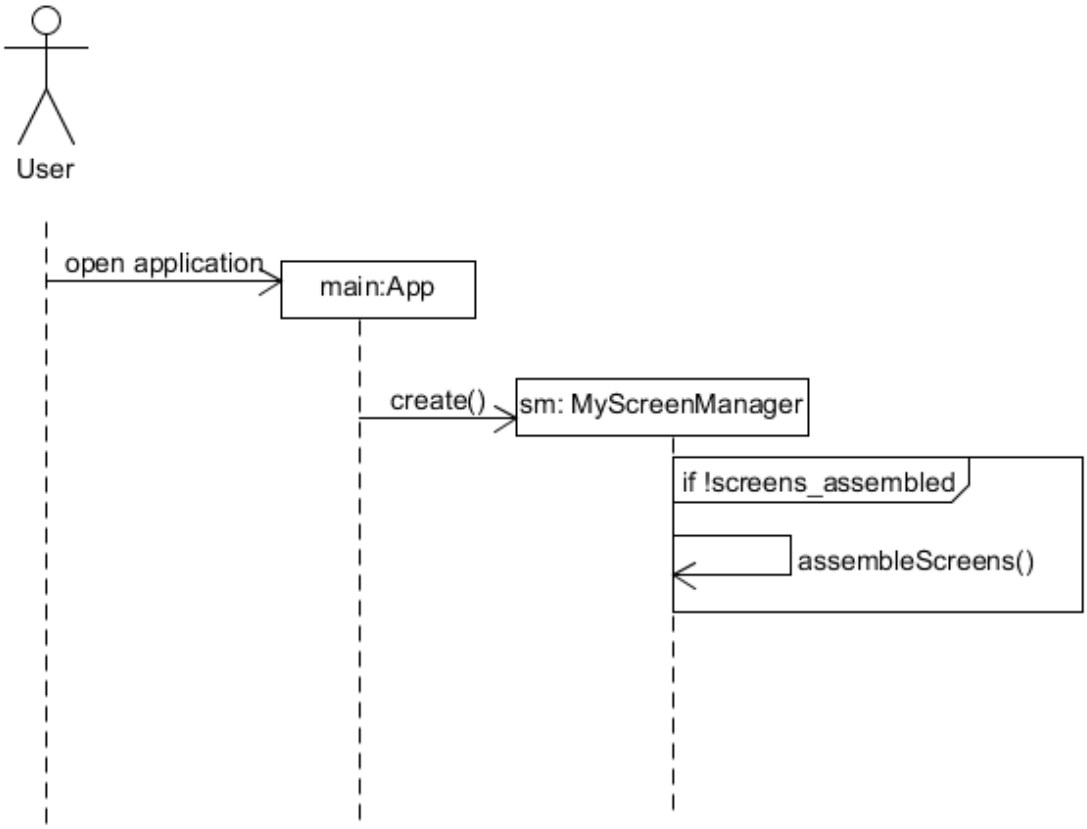
This section gives an overview of the design of all key application functions. It does so using simplified sequence diagrams and pseudocode.

3.1 Iteration 1

This iteration began on the 5th of October 2015 and concluded on the 17th of December 2015.

3.1.1 Start-up

Sequence Diagram



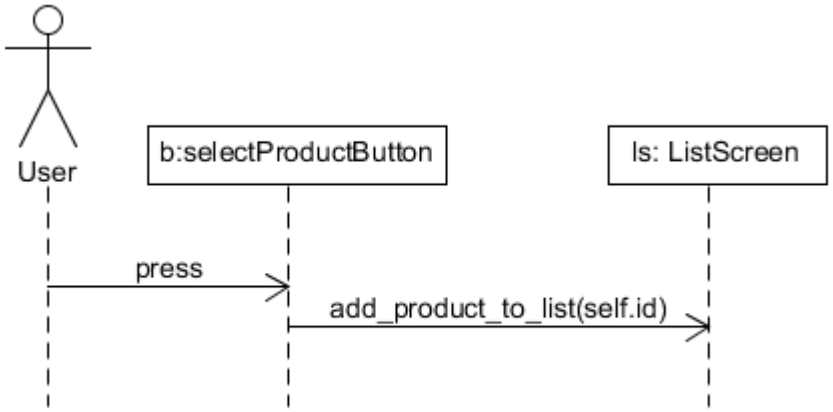
Pseudocode

```
Class MyScreenManager:
	screens_assembled = false
	start-up():
		if user is not already logged in on this device:
			show login screen
		else
			if screens_assembled = false:
				assemble_screens()
				screens_assembled = true
			get user's list from database
			show list screen

	assemble_screens():
		get shops/locations from database
		get products/categories from database
		for each category
			create category screen
			add to category menu
			for each product in category
				add to category screen
		for each location
			create location screen
			add to location menu
			for each shop in location
				add to location screen
```

3.1.2 Add to Shopping List

Sequence Diagram



Pseudocode

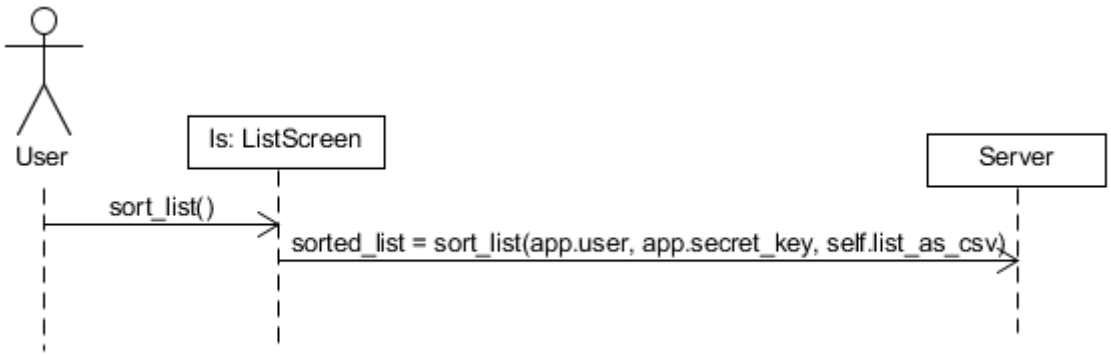
```
class ListScreen:
	add_product_to_list(product):
		if product is already on list
			quantity_dictonary[product] ++
		else
			quantity_dictonary[product] = 1
		refresh list
```

3.1.3 Sync Shopping List to Cloud

Sequence Diagram

3.1.4 Basic Shopping List Sorting

Sequence Diagram



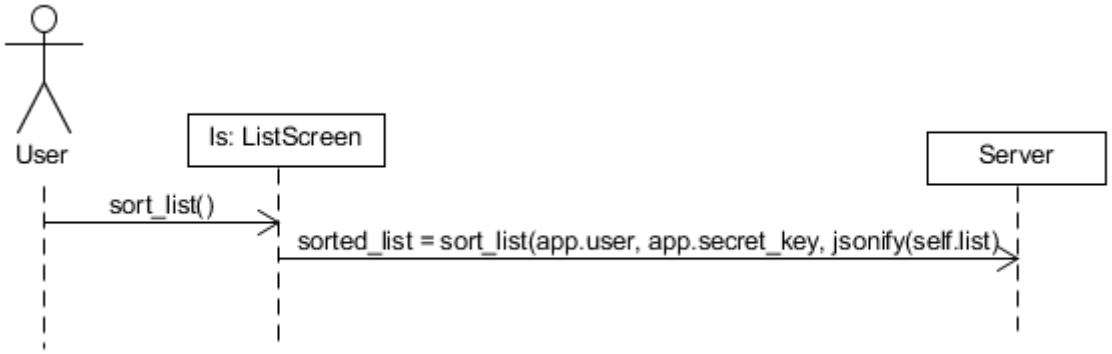
Pseudocode

```
class ListScreen:
	sort_list():
		list_as_csv = current shopping list as csv string
		sorted list = requests.get(<url>, app.user, app.secret_key, list_as_csv)
		for product in sorted list:
			add_product_to_list(product)
```

3.2 Iteration 2

3.2.1 Sorting Shopping List in More Complex Manner

Sequence Diagram

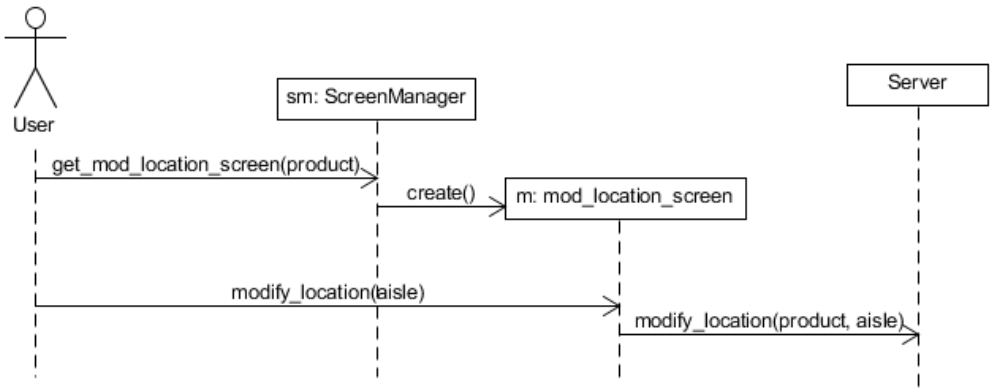


Pseudocode

```
class ListScreen:
	sort_list():
		sorted = requests.get(<url>, app.user, app.secret_key, jsonify(self.list))
		for aisle in sorted:
			add aisle to list
			for product in aisle:
				add_product_to_list(product)
```

3.2.2 Move Product Within Shop

Sequence Diagram



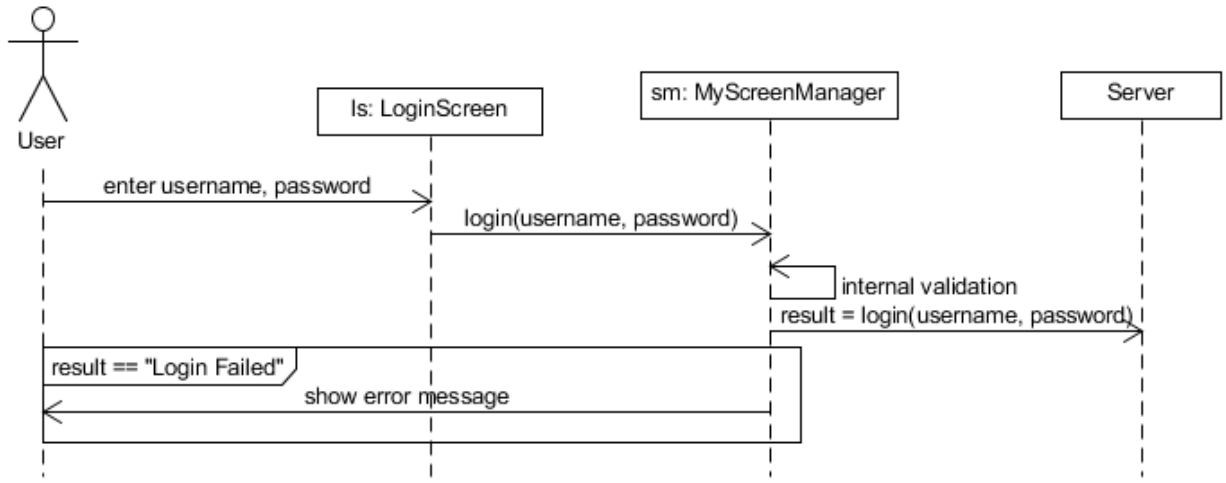
Pseudocode

```
class MyScreenManager:
	get_mod_location_screen(product):
		m = new ModLocationScreen()
		m.product = product
		self.add(m)
		show m

class ModLocationScreen:
	modify_aisle(aisle):
		requests.post(<url>, app.user, app.secret_key, self.product, aisle)
```

3.2.3 Login

Sequence Diagram

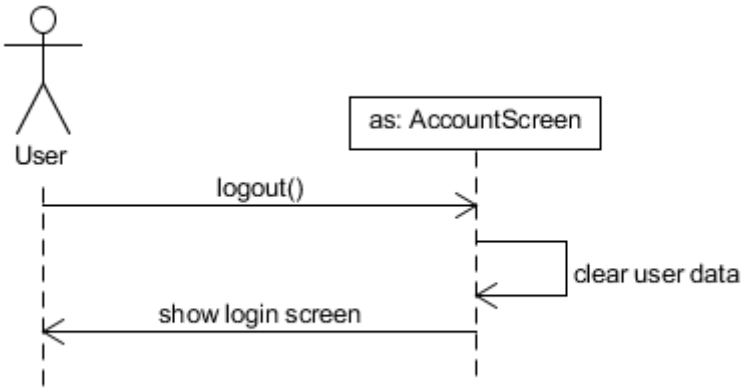


Pseudocode

```
class MyScreenManager:
	login(username, password):
		if username or password do not meet formatting criteria:
			display error message
		else if requests.get(<url>, username, password) == "Login Failed":
			display error message
		else:
			if screens_assembled = false:
				assemble_screens()
				screens_assembled = true
			get users list from database
			show list screen
```

3.2.4 Logout

Sequence Diagram

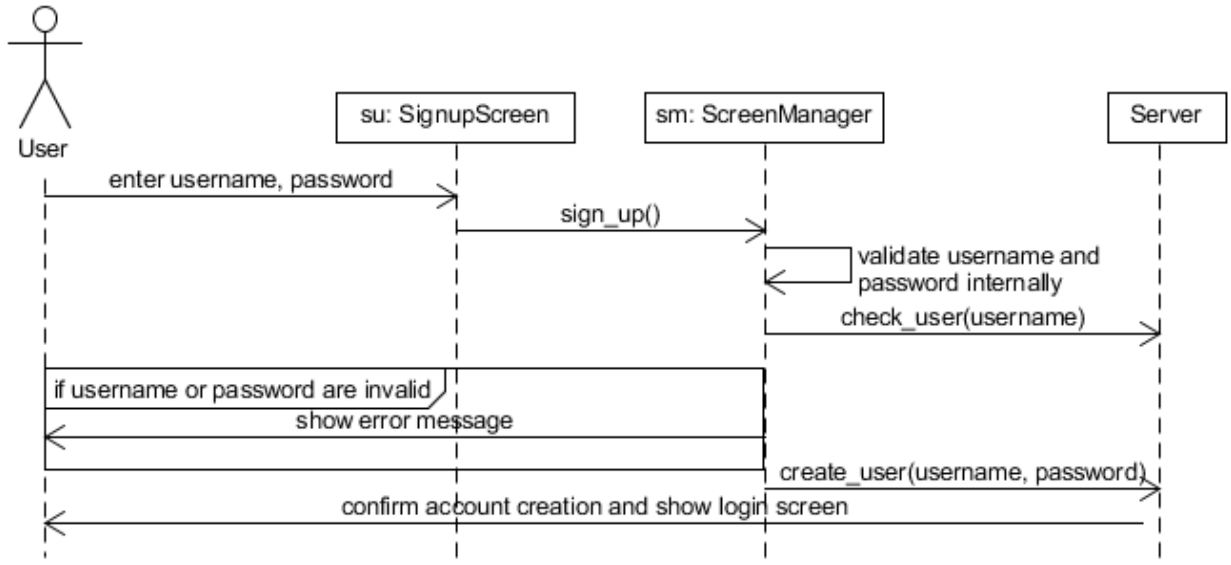


Pseudocode

```
class AccountScreen:
	logout():
		re-initialise variables
		show login screen
```

3.2.5 Create Account

Sequence Diagram



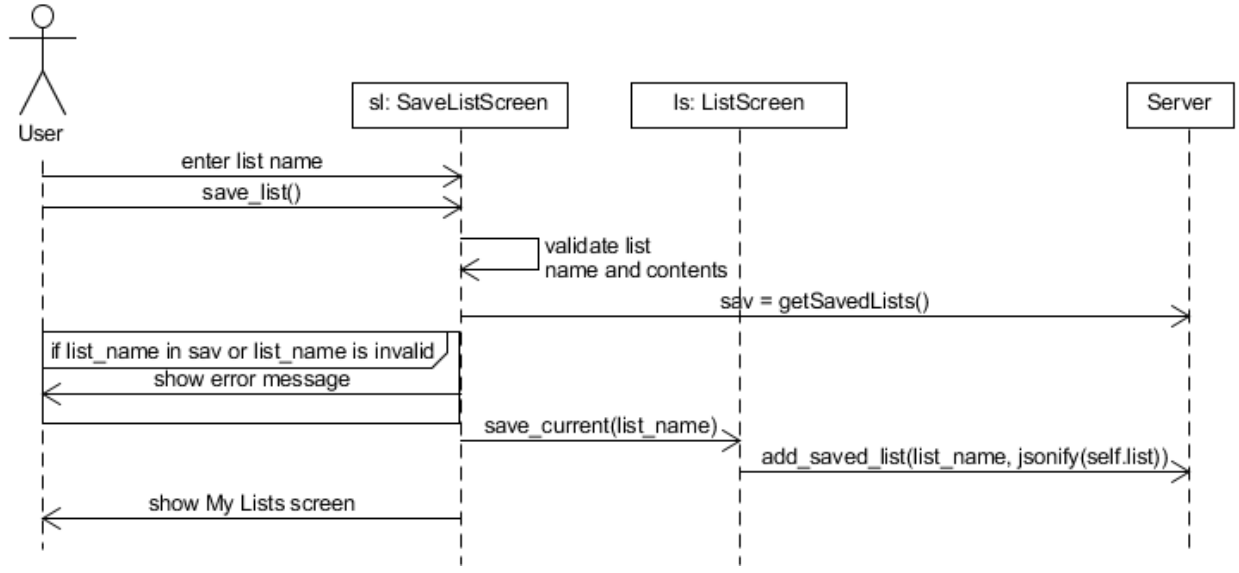
Pseudocode

```
class MyScreenManager:
	signup():
		username = signupScreen.username
		password1 = signupScreen.password1
		password2 = signupScreen.password2
		if password1 <> password2
			display error message
		else if username or password do not meet formatting criteria:
			display error message
		else if requests.get(<url>, username, password) == "Username taken":
			display error message
		else
			requests.post(<url>, username, password)
			confirm account creation
			show login
```

3.3 Iteration 3

3.3.1 Add List to My Lists

Sequence Diagram

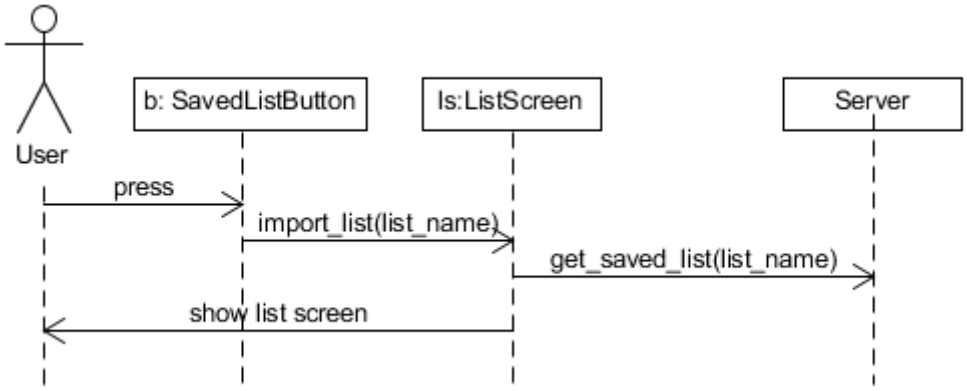


Pseudocode

```
class SaveListScreen:
	save_list():
		list_name = self.list_textbox.text
		if list_name does not meet formatting criteria:
			display error message
		else if requests.get(<url>, list_name) == "List name taken":
			display error message
		else:
			list_screen.save_current(list_name)
			show My Lists screen
class ListScreen:
	save_current(list_name):
		requests.post(<url>, listname, jsonify(self.list))
```

3.3.2 Select List From My Lists

Sequence Diagram

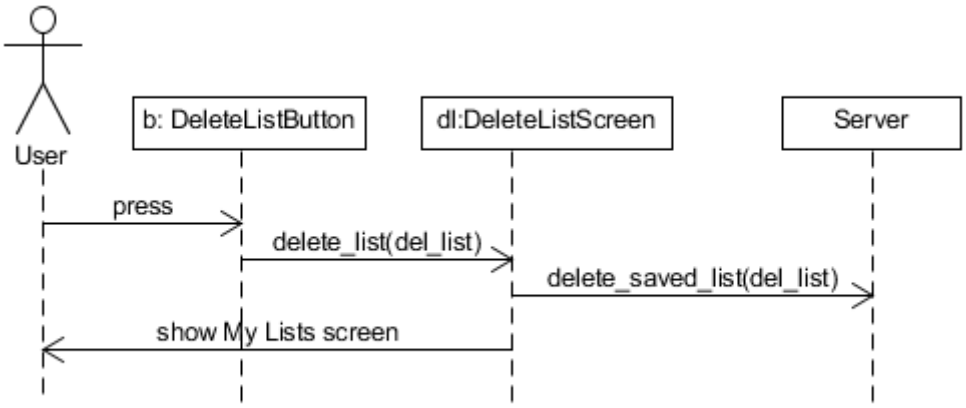


Pseudocode

```
class ListScreen:
	import_list(list_name):
		self.list = requests.get(<url>, app.user, app.secret_key, list_name)
		show list screen
```

3.3.3 Delete List From My Lists

Sequence Diagram

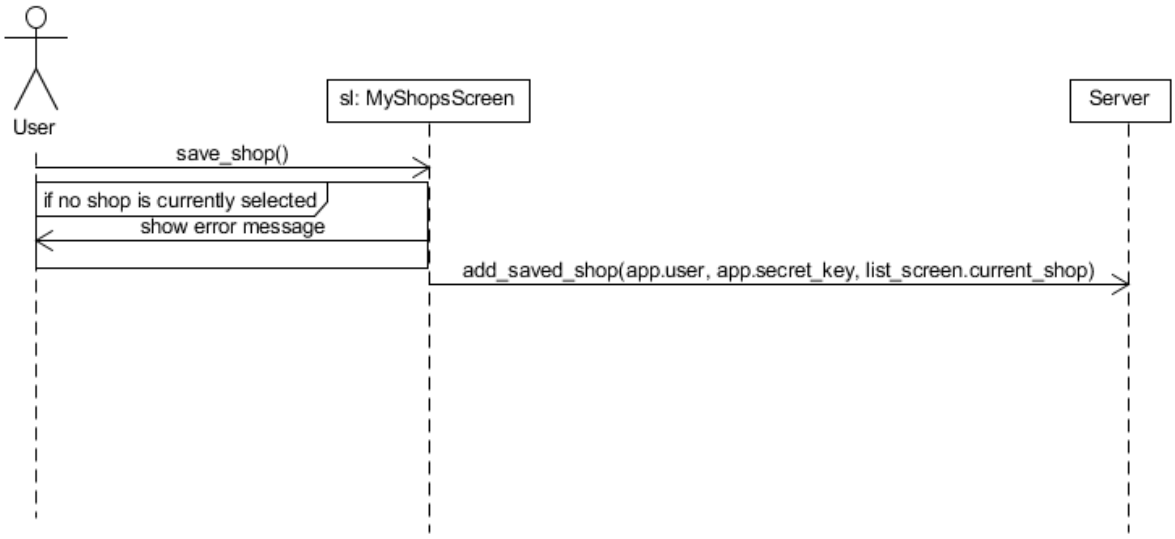


Pseudocode

```
class DeleteListScreen:
	delete_list(del_list):
		requests.post(<url>, app.user, app.secret_key, del_list)
		show My Lists screen
```

3.3.4 Add Shop to My Shops

Sequence Diagram

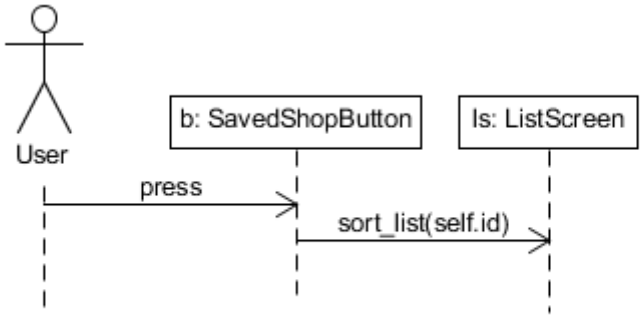


Pseudocode

```
class MyShopsScreen:
	save_shop():
		current = list_screen.current shop
		if current == "":
			show error message
		else:
			requests.post(<url>, app.user, app.secret_key, current)
```

3.3.5 Select Shop From My Shops

Sequence Diagram

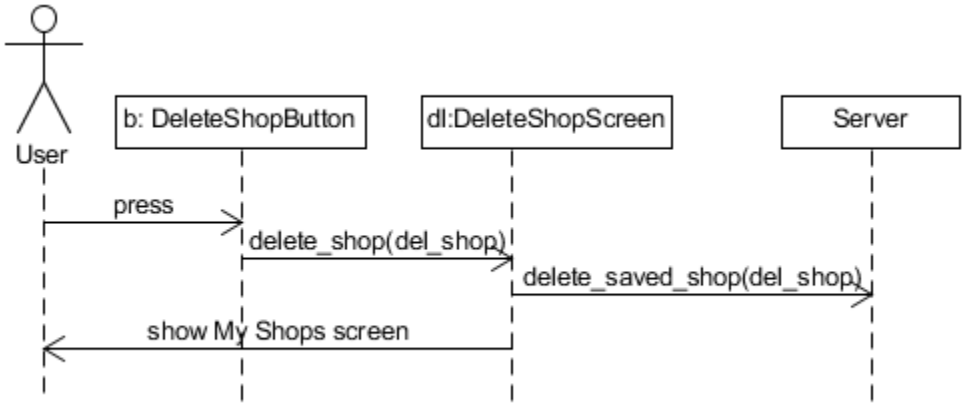


Pseudocode

See 3.2.1

3.3.6 Delete Shop From My Shops

Sequence Diagram

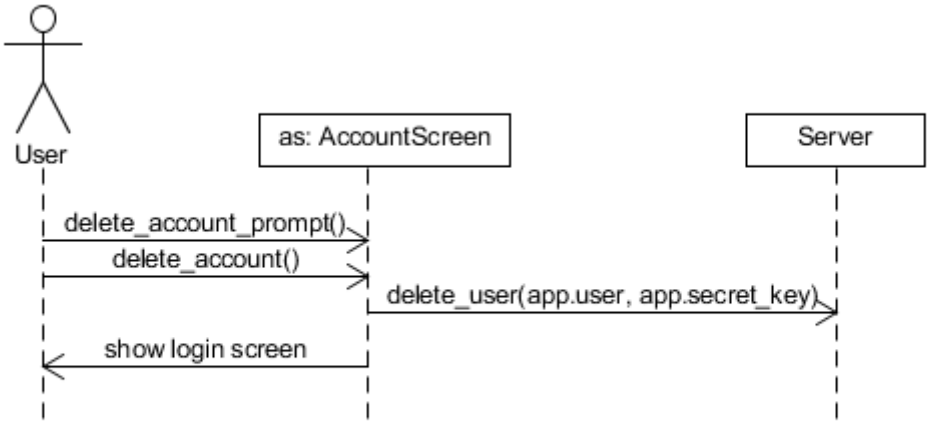


Pseudocode

```
class DeleteShopScreen:
	delete_shop(del_shop):
		requests.post(<url>, app.user, app.secret_key, del_shop)
		show My Shops screen
```

3.3.7 Delete Account

Sequence Diagram



Pseudocode

```
class AccountScreen:
	delete_account_prompt():
		show prompt confirming user wishes to delete account
	delete_account():
		requests.post(<url>, app.user, app.secret_key)
		show login screen
```