Family Tree

Design Document

Jitendra Gandhi

Last Updated: 27th April 2017

**Introduction**

The Software Design Document is a document to provide documentation which will be used to aid in software development by providing the details for how the software should be built. Within the Software Design Document are narrative and graphical documentation of the software design for the project including use case models, sequence diagrams, object behaviour models, and other supporting requirement information.

**Purpose**

The purpose of the Software Design Document is to provide a description of the design of a system fully enough to allow for software development to proceed with an understanding of what is to be built and how it is expected to be built. The Software Design Document provides information necessary to provide description of the details for the software and system to be built.

**Scope**

This Software Design Document is for a base level system which will work as a proof of concept for the use of building a system the provides a base level of functionality to show feasibility for large scale production use. This Software Design is focused on the base level system and critical parts of the system. For this particular Software Design Document, the focus is placed on generation of

the documents and modification of the documents. The system will be used in conjunction with other pre-existing systems and will consist largely of a document interaction facade that abstracts document interactions and handling of the document objects.

**Data Design**

* **Internal Software Data Structure**

Family Tree’s internal structure is divided into two parts: server-side and client-side.

On the client side, data will reside locally in memory and will be organized based on the classes defined later in this document. Since the Family Tree program may be considered data-centric, the classes that handle the data will be isolated and will be accessed by way of a Model-View-Controller system. The data on the local iOS client will be requested from the server based on user actions.

The data structure on the server will essentially mirror the structure of the local iOS client in terms of member fields of the classes. The server is implemented using Python. Permanent storage of user information will be accomplished using a SQLite database.

The server and iOS client will exchange data using the JSON format. JSON is a lightweight object description language that is similar to XML. JSON is being used due to its versatility and because of the fact that an implementation is available for both Python and Swift.

* **Global Data Structure**

The global data structure of this application is best characterized by the database. The database structure shows the data involved in the application in its purest sense. The local iOS client of Family Tree will never access this database directly, it will instead issue requests to the server.

* **Temporary Data Structure**

Temporary data structures, as they relate to Family Tree, refer to the data objects that are created on the local iOS client, and also to the JSON objects that interchanged between the server and the client. The data objects created on the local device will only exist for the duration of time that the application is running, and will subsequently be destroyed.

The JSON objects will only exist for the duration of the transaction between the client and server. The server will destroy the objects after sending them, and the client will destroy the JSON objects once they have been parsed.

**Architectural & Component-Level Design**

* **System Architecture**

The Family Tree system is broken up into two major components: a client-side iOS application and a server-side Django Python application and SQLite database.

The client-side application is also separated into two parts: the functional component (written in Swift), and the graphical component (designed in Storyboard, written in XML). The functional component forms the core of Family Tree. It receives user input and Family tree graph. It performs two type of core operation, adding new relation to graph and deletes existing relation from graph. The graphical component, as the name implies, is simply the graphical user interface. It provides all of the buttons, text fields, and other on-screen elements which allow the user to access all of the features provided by the application.

The server component of Family Tree is comprised of a Python interface, which manages incoming and outgoing messages, and a SQLite database, which provides centralized storage for synchronized data. The server application receives serialized data from iOS devices and parses it into useful information. This data is then stored in the database and subsequently synchronized to other devices (if any) in the same group.

* **Person Class**

The Person class is meant to represent member of family in Family Tree. It includes a unique identifier, a SSN number, user’s name, date of birth, date of death, sex. Person class object are meant to represent family member within the application & there could be many relationship object associated to it.

* **Relationship Class**

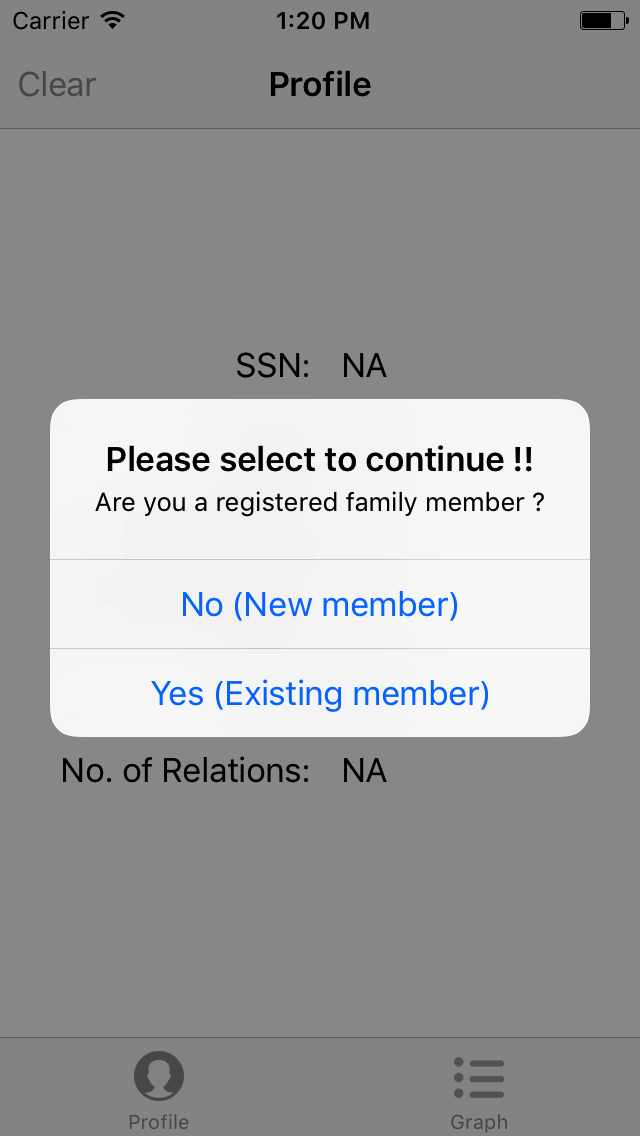
The Relationship class is meant to hold relation type and a person object to which it relates.

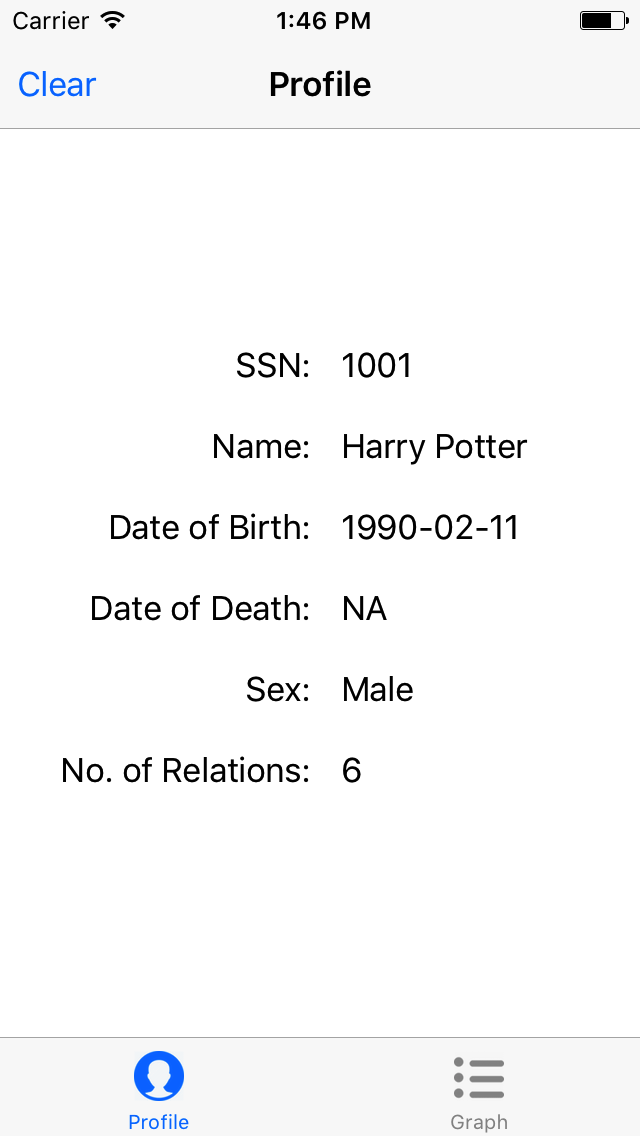
**User Interface Design**

The user interface consists of a set of views through which the user can interact with data on the Family Tree server. These views include a “Profile” view, a “Graph” view & “Add Connection” view. Profile view includes data information for root member, Graph view include family tree graph for root member and Add Connection view is used to register a new root member or to add new relation. The user will interact with the views through the device’s touch screen.

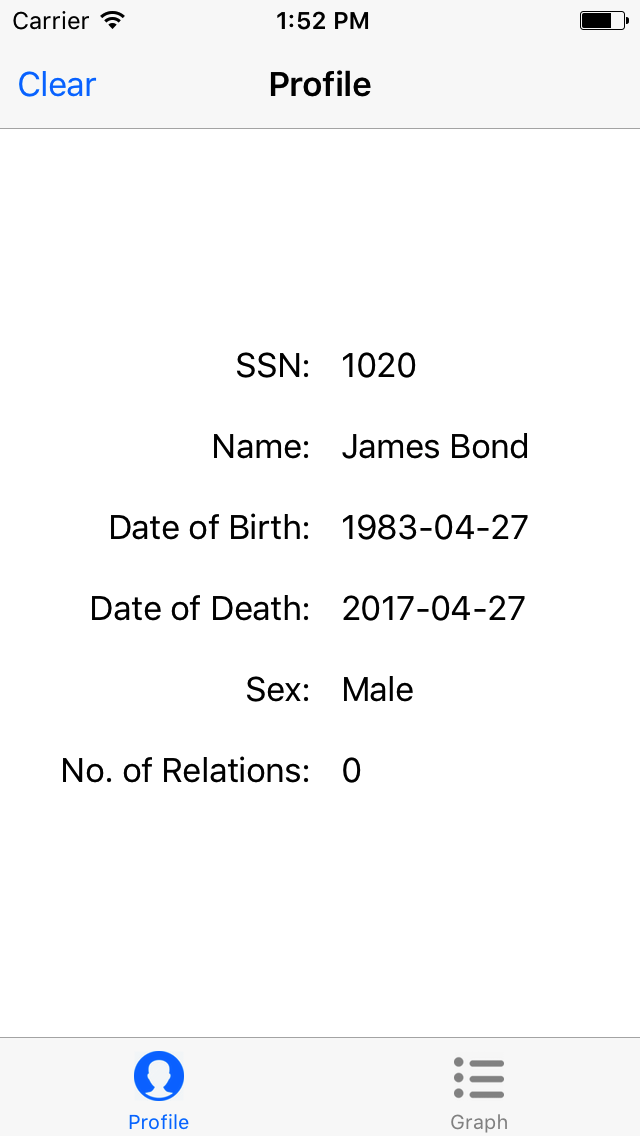
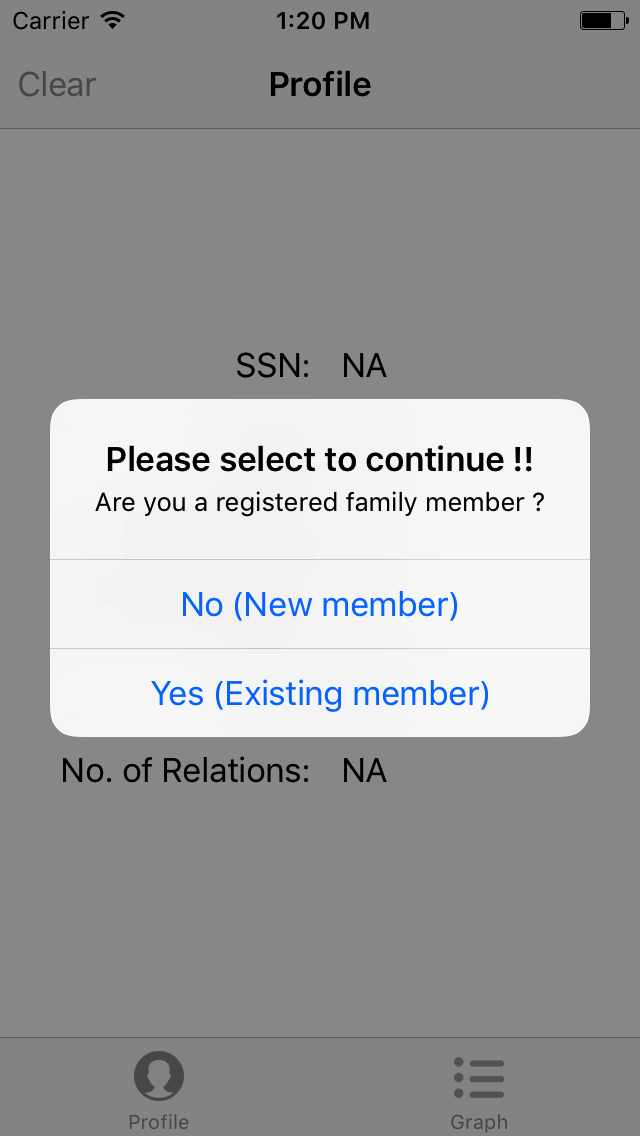
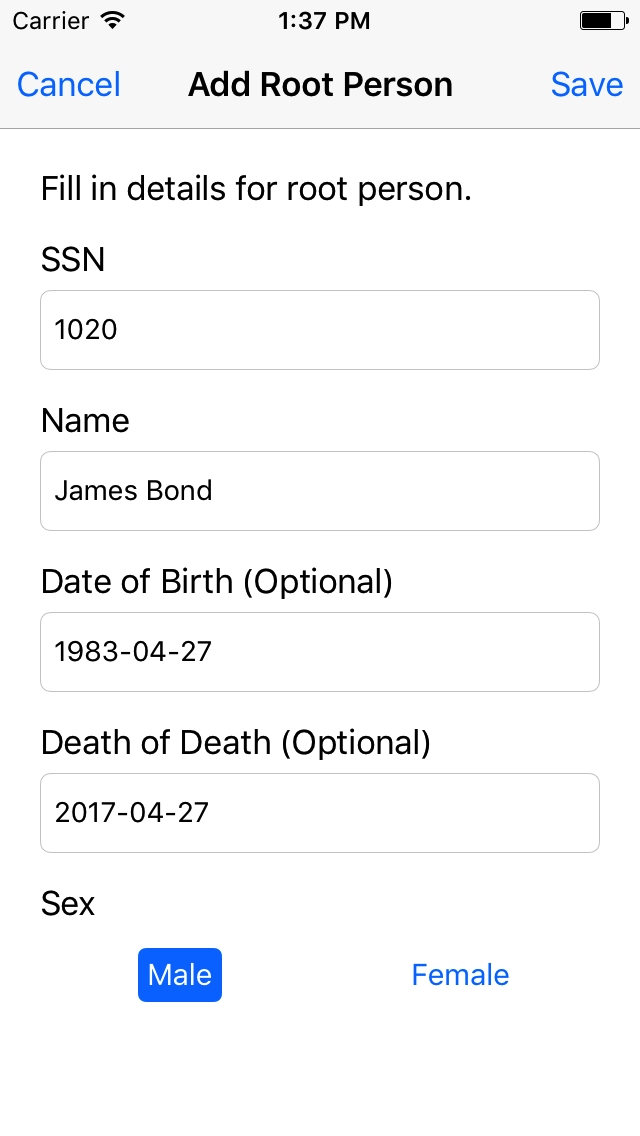
* **Description of User Interface**

Each menu will consist of various GUI components, such as buttons, labels, text fields, and list objects. These components will be arranged in such a way that the user will be able to quickly grasp the purpose of each menu and perform whatever task it is designed for efficiently.

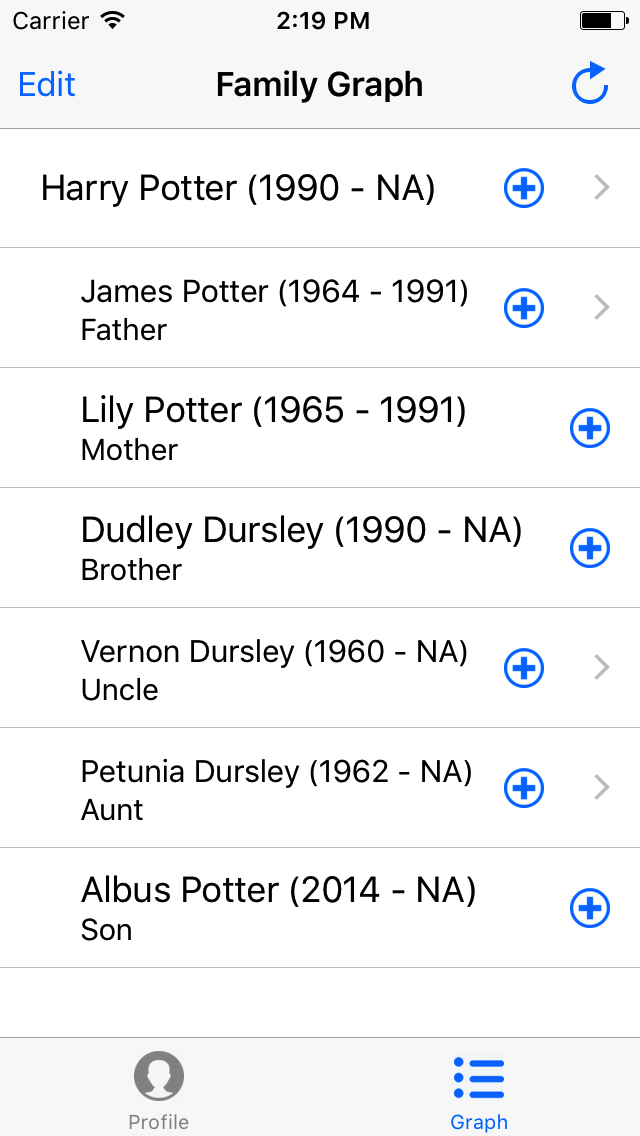
1. Profile View - For existing members

****

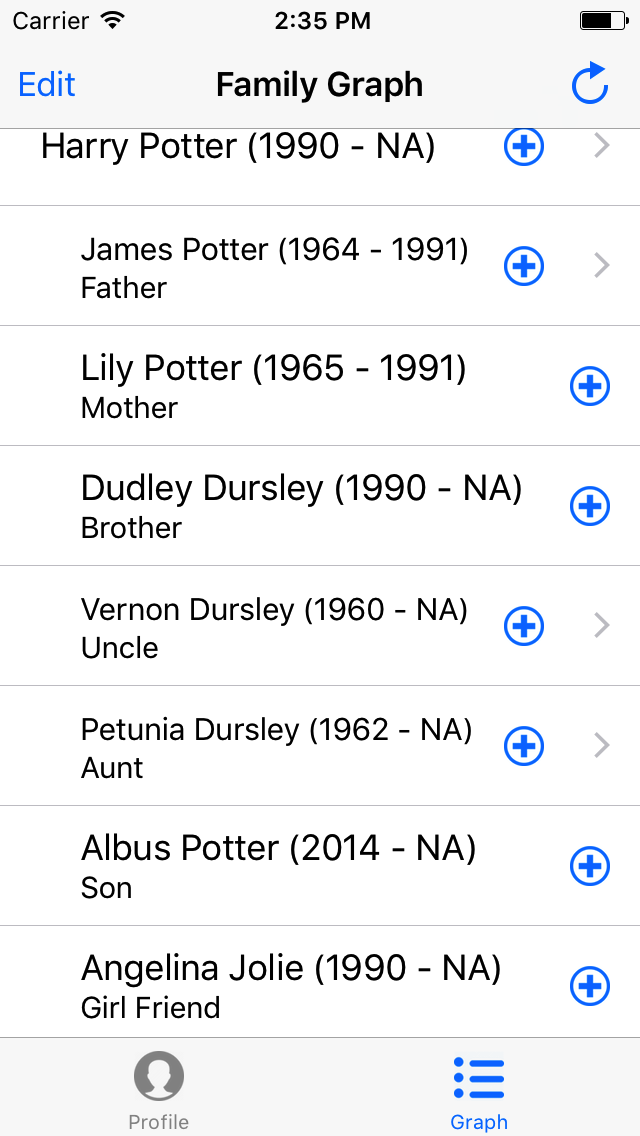
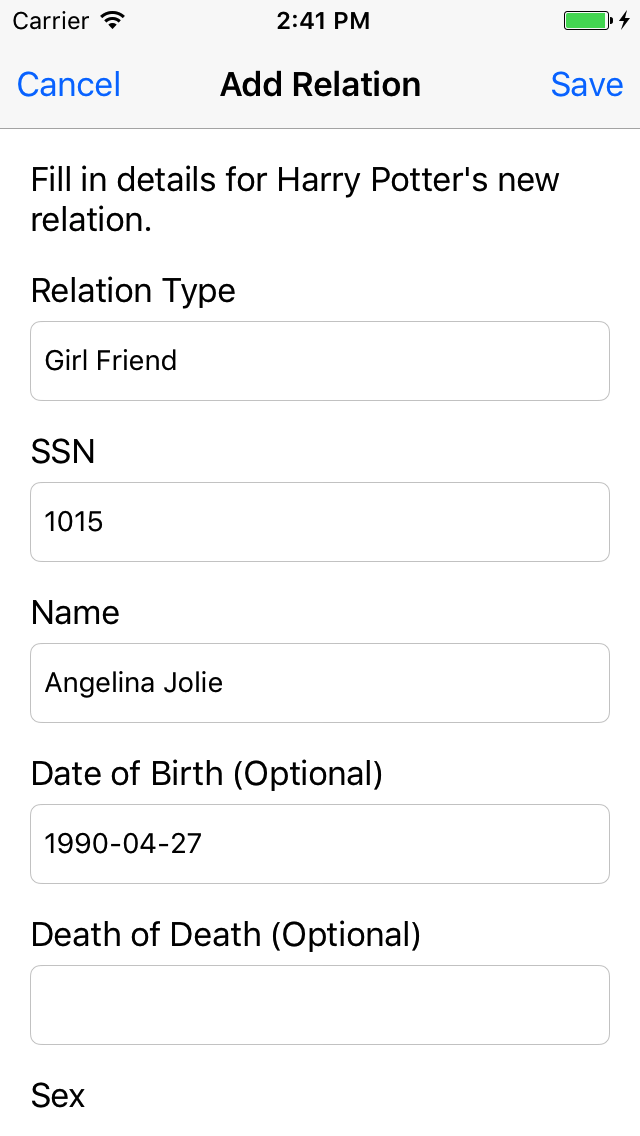
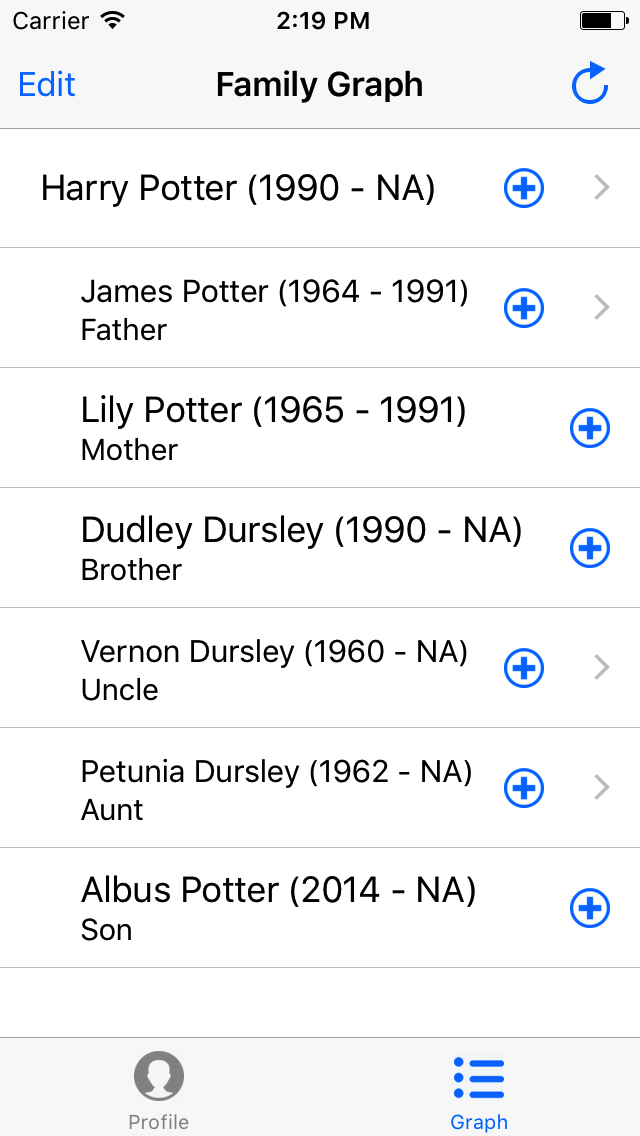
1. Profile View - For new member - Add Connection View (Adding root)

********

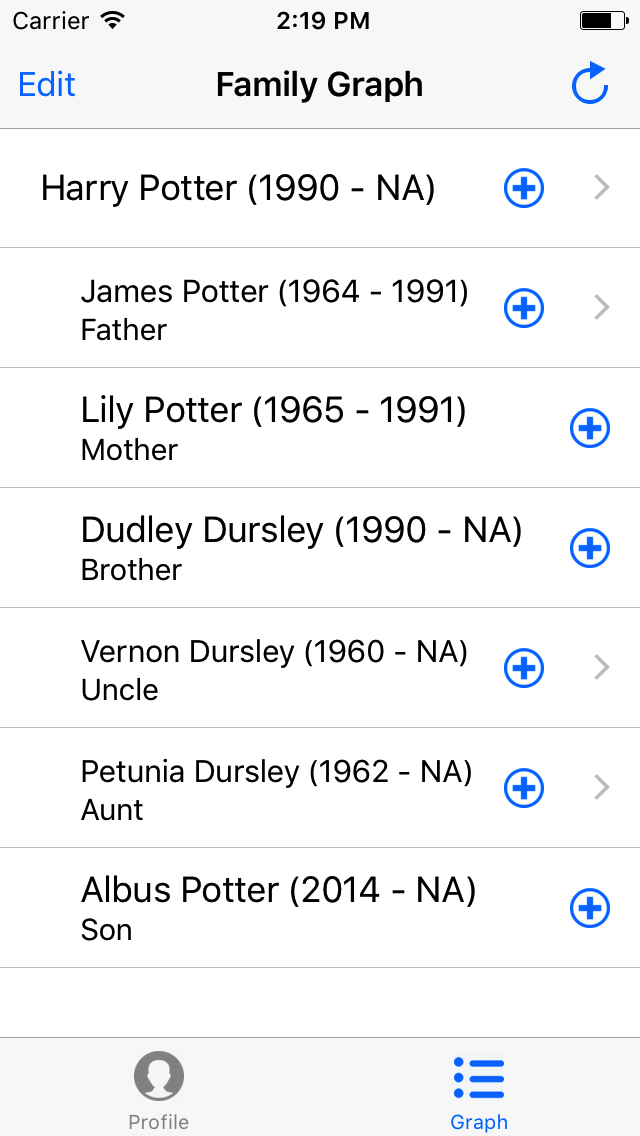
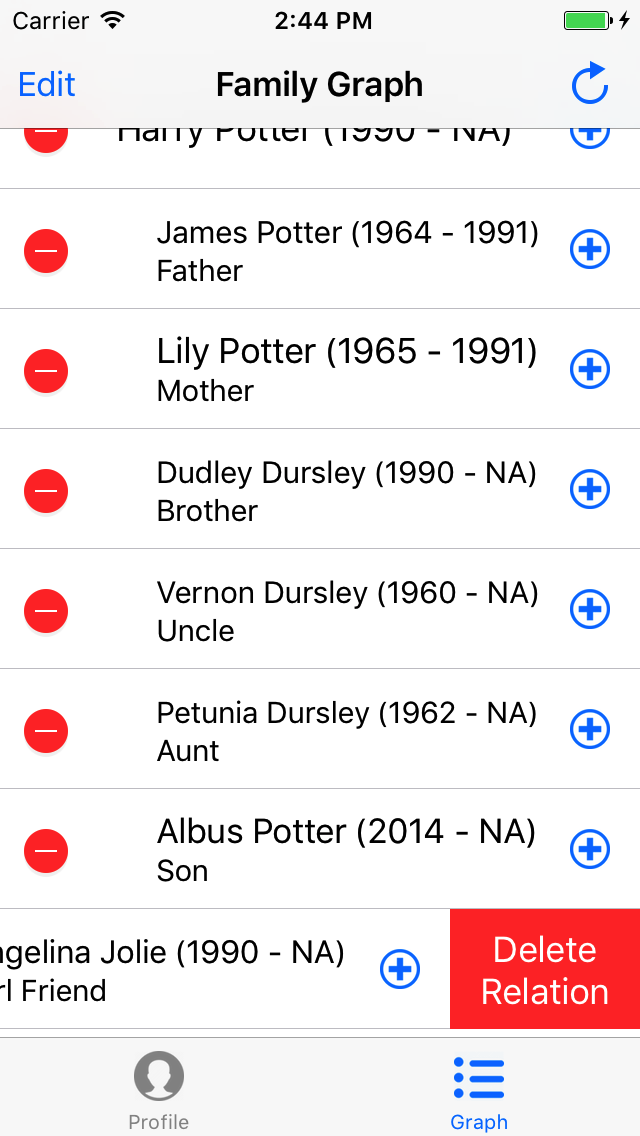
1. Graph View



1. Graph View – Add new relation – Add Connection View (Adding relation)

****

1. Graph View – Delete existing relation

****

**Use Cases**

* **Actors**

1. Root Member

It is the person of the family which belongs to the root level of application, all the connections (i.e. relations) are shown from this person. It can see all relations associated with him, and also relations of relation in tree form. It has ability to add new and delete existing relation to itself or to any other relative. Any person in the family can register itself as root member.

* **List of Use Cases**

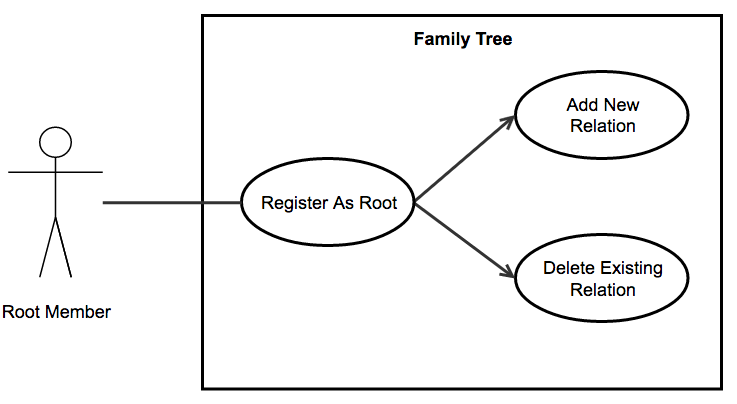
1) Root Member User Use Cases

1.1) Register as root

a) Add new relations

b) Delete existing relation

* **Use Case Diagram**



* **Use Case**

1. Root Member Use Cases – Register as root

Brief Description:

This use case describes the registration of actor as root member in the application which is a key function and entry point of whole system. In this use case, actor’s goal is to successfully register itself as root member.

Flow:

* + If actor is already registered with system, it can add itself as root using its SSN number or else it can proceed with registering as a fresh member of family.
  + Adding root using SSN number, fetches its information from system (including all relations, if exist) and reflects it to actor & draws family graph.
  + Other option, let actor register itself as root by adding its information into system & saving them.
  + Once finished saving, actor can see its detail along with family tree (if any).

1. Root Member Use Cases – Register as root - Add new relation

Brief Description:

This use case describes the creation of new relation, and show it to the family graph of actor. In this use case, actor’s goal is to successfully create a new relation.

Flow:

* + Actor selects the type of relation he/she want to add to its family tree, followed by entering SSN number of new person in text based input by tapping on plus button in front of every person (node) in tree.
  + If person, actor is trying to add already exist in system, his/her information is reflected to actor if not then actor must manually enter his/her details in appropriate input system.
  + Once finished adding/confirming details, actor can save details, using a save button & it will be reflected to family tree of actor.

1. Root Member Use Cases – Register as root - Delete existing relations

Brief Description:

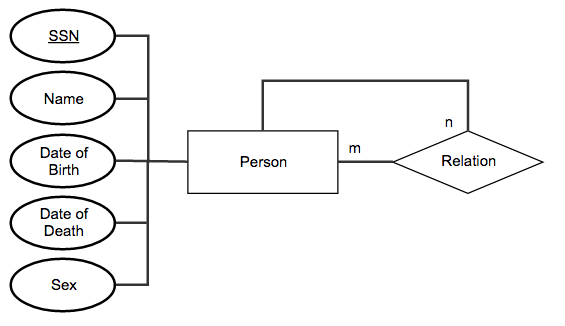
This use case describes the deletion of existing relation from the family graph of actor. In this use case, actor’s goal is to successfully deletes existing relation from itself or from any of its relations.

Flow:

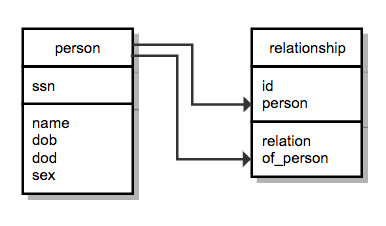
* Actor selects the person whom he/she want to delete from its family tree.
* Actor confirms deletion of person, and it is removed from the family tree of the actor but remains saved in system.

**Data Design Diagram**

* Entity – Relationship Diagram

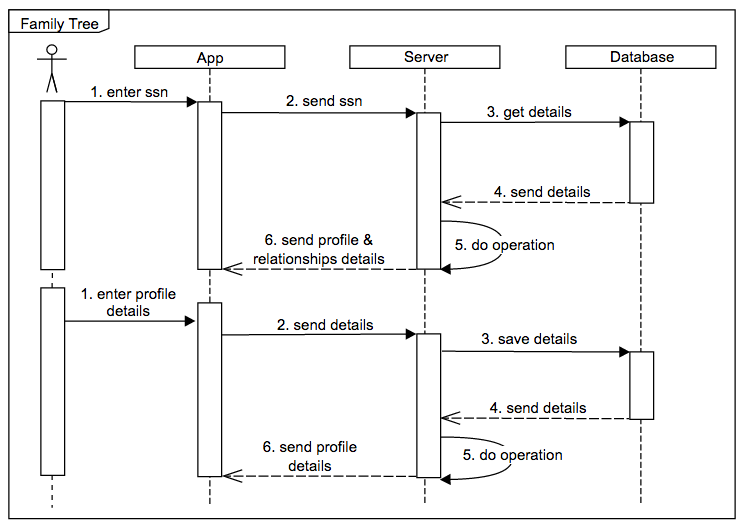


* Database Schema

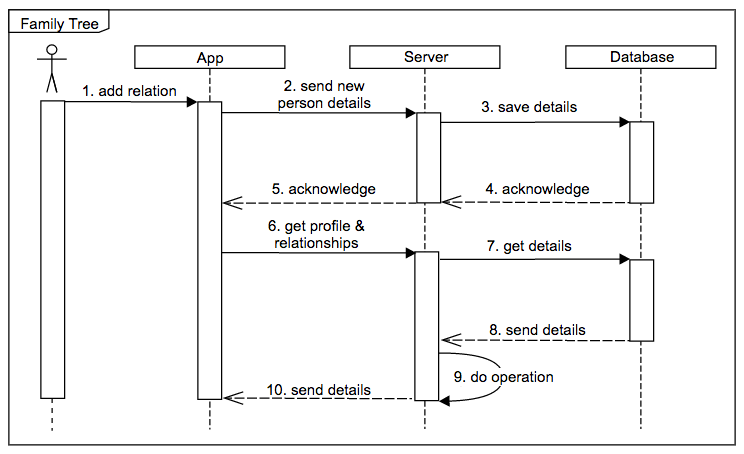


**Sequence Diagram**

* **Adding root member**



* Adding a new relation



* Deleting an existing relation

