**UNIVERSIDAD DE LAS AMÉRICAS PUEBLA**

**ESCUELA DE INGENIERÍA**

**DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA, MECATRÓNICA Y COMPUTACIÓN**

# "ParserLintJ: Una librería para manipular información en formato JSON para Swift"

Tesis que, para completar los requisitos del Programa de Honores presenta el estudiante

Pablo de la Rosa Michicol

ID: 150245

Licenciatura:

Ingeniería en Sistemas Computacionales

Mentor:

Dr. Gerardo Ayala San Martin

Santa Catarina Mártir, Puebla

Mayo de 2018-05-06

**Índice de contenido**

Resumen

1. Introducción…………………………………………………………………...1

1.1 Objetivo General ……………………………………………………………….1

1.2 Identificación del problema……………………………………………..................1

1.3 Objetivos específicos……………………………………………………………2

1.4 impacto socio-económico…………………………………………………......3

2. Contexto y situación actual…………………………………………………......4

2.1 JSON (Javascript object notation)……………………………………………......4

2.2 Tipos de datos soportados por JSON…………………………………………….6

2.3 Tipos numéricos………………………………………………………………….6

2.4 Cadenas en JSON…………………………………………………………………...6

2.5 Tipos booleanos en JSON………………………………………………………..7

2.6 Arreglos en JSON………………………………………………………………..7

2.7 Objetos en JSON…………………………………………………………………....7

2.8 JSONSerialization……………………………………………………………......9

2.9 El protocolo embebido Codable…………………………………………………...10

3. Diseño………………………………………………………………………….....14

4. Prototipo……………………………………………………………………….......23

5. Programación………………………………………………………………………..29

6. Compatibilidad de versiones……………………………………………………...32

7. Conclusiones…………………………………………………………………….....33

Referencias

# Resumen

La serialización de información en la actualidad es un proceso que realizan la mayoría de las aplicaciones, ya que, al intentar comunicar una aplicación móvil con un servidor, requiere de este proceso, para llevar esto a cabo, un desarrollador puede optar por usar XML o JSON, como lenguajes de formato estándar, pero ante la complejidad que demuestra XML, la mayoría de los sistemas actuales están optando por usar JSON para realizar este intercambio de información. Para apoyar está postura se ha realizado una librería de parseo de JSON que facilita la lectura de información proveniente de un servidor o de archivos locales, como también la codificación, para también poder emitir información y la creación de vistas dinámicas, todo a partir de archivos en formato JSON.

# 1. Introducción

## 1.1 Objetivo General

La presente tesis es una investigación que tiene por objetivo la creación de una librería para parseo de JSON desarrollada en el lenguaje de programación de Swift. En la actualidad el envío de datos que se realiza entre un cliente y un servidor es mediante el uso de formatos estándares, JSON es una notación literal de objetos del lenguaje de programación JavaScript, que funge como alternativa de XML, el cual es otro formato de transmisión de datos, pero cuya sintaxis es más complicada. De las principales ventajas que tiene JSON sobre XML es la facilidad con la que se puede programar su analizador sintáctico.

## 1.2 Identificación del problema

En Swift existen alternativas para la transmisión de datos usando el formato JSON, la más común es utilizar directamente la clase JSONseralization para enviar o recibir información a un servidor, el inconveniente de esta clase es que al agregar más funcionalidad a la aplicación sobre la cual se usa esta clase, el código llega a ser muy complejo y poco entendible si otro programador desea acceder a él, además de que no ofrece funciones específicas que son muy comunes al momento de realizar una aplicación en Swift, como leer un archivo en formato JSON, otra alternativa es importar una librería externa, la desventaja de esto, es que la mayoría de estas librerías ocupan mucho espacio y al momento de mandar una aplicación a producción, la cantidad de memoria que ocupa está puede aumentar considerablemente, además de que tampoco ayudan a solucionar el problema de tener funciones específicas que ayuden a realizar más rápido la aplicación.

Hoy en día el uso de JSON, cubre más el mundo de las aplicaciones, incorporándose recientemente con Firebase, para el desarrollo de aplicaciones en tiempo real, de tal manera que incluso se han llegado a construir bases de datos.

Cabe destacar que la importancia del desarrollo de esta librería, está centrada en la cantidad de aplicaciones que hoy en día requieren transmitir datos de una manera constante, aplicaciones que usamos todo el tiempo.

## 1.3 Objetivos específicos

Los objetivos específicos que se tenían contemplados al inicio del proyecto, fueron básicamente crear un conjunto de clases que ayudarán al desarrollo de aplicaciones móviles que usen mucho tráfico de información y que buscan serializar de manera correcta dicha información, además de ofrecer una alternativa al momento de la construcción del view de una aplicación móvil, a través de vistas dinámicas, las cuales son manipulables desde un archivo en formato JSON, los objetivos mencionados anteriormente se lograron de manera exitosa, ya que en el desarrollo de la presente tesis, se irán denotando la construcción de dichos objetivos.

## 1.4 Impacto socio-económico

El impacto social que se espera tener es que los desarrolladores puedan hacer uso de la librería para facilitar el desarrollo de sus aplicaciones, asimismo se espera que futuros desarrolladores puedan exhibir nuevas mejoras y propuestas, para las futuras versiones de esta librería ya que como muchas librerías y frameworks, esta será open source, gestionada por GitHub como herramienta para el manejo de versiones.

El impacto económico esperado es que gracias a esta librería el tiempo de desarrollo de aplicaciones móviles que usen JSON, sea mucho más rápido, por lo consiguiente, la entrega de dichas aplicaciones, tiene una gran probabilidad de que sea

entregada en un tiempo menor, debido a que la tarea de serialización se facilita bastante con la librería, se espera que este sea un atractivo también para los clientes, ya que las entregas de sus aplicaciones móviles serían más rápidas.

# 2. Contexto y situación actual

## 2.1 JSON (Javascript object notation)

JSON, es un formato de lectura para datos estructurados, intercambio de información y almacenamiento, principalmente utilizado para la transmisión de datos entre un cliente y un servidor para el desarrollo de aplicaciones web, móviles, etc. Fungiendo como una alternativa a XML, debido a su simplicidad y fácil lectura.

Actualmente todas las plataformas están reemplazando XML con JSON, está idea nació desde 1980, pese a que se cree que JSON es algo reciente, lo cierto es que muchas personas se dieron cuenta que los objetos en Javascript eran una manera ideal para transmitir información a través de la red. Con la popularidad que logró alcanzar AJAX (Asynchronous Javascript and XML), JSON, comenzó a hacerse más conocido también, este hecho atrajo la atención de más programadores y comenzaron a desarrollar técnicas que se basaban en la creación de simples objetos literales en Javascript< para realizar intercambios de información.

La idea principal sobre la cual, la popularidad de JSON ha ido incrementándose durante los últimos años es que es mucho más fácil cargar información visualizada en un árbol estructurado, que recorriendo un sistema taggeado como lo es en el caso de XML, además de que, a nivel de red, ayuda a salvar ancho de banda y mejora los tiempos de respuesta radicalmente cuando se envían mensajes y también cuando estos mismos son recibidos. El hecho de que grandes compañías ya habían dado marcha a que el envío de información lo hacían mediante el formato estándar taggeado ya mencionado anteriormente.

Sintácticamente, JSON es muy similar a un objeto Literal de Javascript, ya que los objetos en JSON, empiezan con una sintaxis, de llaves de apertura y de llaves de cierre. Dentro de las llaves, es donde se especifican desde cero a más entradas de llaves/valores en parejas, estas parejas son conocidas como miembros. Cada miembro está delimitado por comas, mientras que las comillas se utilizan para demarcar la separación entre una llave y su valor asociado, la llave debe ser una cadena marcada con comillas dobles, he aquí la mayor diferencia entre un objeto en formato JSON y un objeto literal de Javascript, el cual permite el uso de comillas simples, comillas dobles o inclusive no utilizar comillas, en comparación de un objeto en formato JSON que si requiere de comillas dobles, en cuanto al formato del valor, este va a depender del tipo al que se desea hacer referencia.

Listando 2.1 Ejemplo genérico de un objeto en formato JSON.

```
{
“key1”: value1,
“key2”: value2,
“keyn: valueN
}
```

La ruta de un archivo en formato JSON es normalmente un objeto, sin embargo, también puede denotarse como un arreglo.

**2.2 Tipos de datos soportados por JSON**

Una gran ventaja de JSON es que ofrece un soporte de tipos de datos nativos. Especificando JSON soporta datos de tipo:

- Numéricos.
- Cadenas.
- Booleanos.

- Arreglos.
- Objetos.
- Valores nulos.

## 2.3 Tipos numéricos

En JSON, los números no deben tener ceros a la izquierda, deben tener como mínimo un dígito seguido de un punto decimal, si es que es presentado, debido a la restricción de los ceros, JSON sólo es capaz de soportar valores de números en base 10 ya que números en formato octal y hexadecimal requieren de un cero a la izquierda, si se requieren usar números de bases diferentes, primero deben ser convertidos a números en base 10.

Listando 2.2 Ejemplos de datos numéricos en JSON válidos.

```
var number = 100;
var octal = 0144;
var hex = 0x64;
```

Listando 2.3 Ejemplos de datos numéricos en JSON no válidos.

```
var json1 = “{\” key1\”: 0144}”
var json2 = “{\” key2\”: 0x64}”
```

## 2.4 Cadenas en JSON

Las cadenas en JSON son similares a las cadenas en Javascript, la mayor diferencia es que las cadenas en JSON, requieren obligatoriamente ser cerradas entre comillas, si se intentarán usar comillas simples, marcaría un error.

Listando 2.4 Ejemplo de cadena en JSON.

```
var string = “hello”;
```

**2.5 Tipos booleanos en JSON**

Los tipos de datos booleanos en JSON son muy parecidos a los booleanos en Javascript, ya que solo pueden sostener dos valores: true y false.

Listando 2.5 Ejemplo de booleano en JSON.

var jsonBoolean = “{\”foo\”: true, \”bar\”:false}”

**2.6 Arreglos en JSON**

En Javascript, un arreglo es considerado un objeto, se denomina particularmente como objetos de tipo lista de alto nivel. Los arreglos pueden contener tipos de valores muy distintos, siempre y cuando sean tipos de datos soportados en JSON, incluso se pueden construir “nested arrays”.

Listando 2.6 Ejemplo de arreglo en JSON.

var jsonArray = “{\”foo\”: []}”

**2.7 Objetos en JSON**

En Javascript, una de sus particularidades, es que las estructuras más complejas que tiene, siempre son construidas como objetos. Un objeto es una colección desordenada de llave/valor, una similitud que tiene con los arreglos es que pueden estar compuestos de cualquier tipo de datos siempre y cuando estén soportados en JSON.

Listando 2.7 Ejemplo de arreglo en JSON.

var jsonObject = “{\”foo\”: {\” bar\”: {\” baz\”: true}}}”;

Los valores nulos en Javascript de igual manera es soportado en JSON.

Listando 2.8 Ejemplo de valor nulo en JSON.

var jsonNull = “{\”foo\”: null}”;

Los tipos de datos que no son soportados en JSON son:

- Undefined

- Objetos en funciones
- Objetos de tipo date.
- Objetos de tipo RegExp.
- Objetos de tipo Error.
- Objetos de tipo Math.

En la mayoría de las ocasiones trabajar sobre un objeto JSON de gran magnitud puede ser tedioso debido a la gran magnitud que puede tener este mismo, por ello es que se han creado funciones que hacen que este trabajo sea más ligero, los más comunes son:

- stringify ().
- parse ().

JSON.stringify() es la forma recomendada para serializar cualquier objeto en Javascript en cadenas JSON

Listando 2.9 Ejemplo del uso del método stringify ().

```
JSON.stringify(value [, replacer [, space]])
JSON.toJSON()
```

En dados casos es posible customizar la forma en la que los datos son serializados de diferentes maneras, durante dicho proceso de serialización, al momento de recorrer el objeto JSON, se busca por el método toJSON (), casos como el objeto Date, estos vienen equipados con una propiedad, que es el método toJSON ().

Listando 2.10 Ejemplo del uso del método toJSON ().

```
var jsonObject = {llave: 0};
obj.toJSON = function () {
var values = {};
for (var key in this) {
```

```
        if (key == "llave") {
        continue;
    } else {
    values[key] = this[key]
    }
    }
    return values;
    };
```

## 2.8 JSONSerialization

La clase JSONSerialization es la que se encarga de recolectar los datos de tipo Data y poder convertirlos en el tipo de datos en el que se desea trabajar, está es una clase nativa de Swift, normalmente al momento de utilizar está clase, se requiere del uso del manejo correcto de opcionales, ya que no en todos los casos las estructuras que se leen en formato JSON, están de manera correcta, para evitar que la aplicación deje de compilar, debemos desenvolver de manera adecuada el opcional.

If let es la manera más común de desenvolver un opcional, provee un valor alternativo al momento de comprobar si el valor es nulo o no, utilizando lo que se denomina como encadenamiento opcional.

Listando 2.11 Ejemplo del uso del if let en el lenguaje de programación Swift.

```
If let newValue = currentValue?  as String {
        print ("New Value", newValue)
}
```

La cadena de opcional va a regresar el valor del tipo que se intenta desenvolver.

La manera más segura de desenvolver un opcional, es utilizando un guard statement como se muestra a continuación.

Un guard es utilizado para transferir valores de manera segura que están fuera del alcance si una o varias condiciones no coinciden

Listando 2.12 Ejemplo del uso de un guard statement let en el lenguaje de programación Swift.

```
guard (condition) else {
    (statement)
}
```

El objetivo de la clase JSONSerialization es convertir objetos en formato JSON en objetos del framework Foundation, para que de esta manera el lenguaje de programación Swift pueda leerlos de manera correcta.

**2.9 El protocolo embebido Codable**

Codable es un protocolo disponible a partir de IOS 11 en el Framework Foundation de Swift, este protocolo puede ser representado ya sea por una clase o por un struct, la idea principal de este protocolo es generar instancias a partir de un archivo en formato JSON, las mayores ventajas de Codable son:

- Permitir cargar información de un archivo en formato JSON a través de instancias.
- Permitir enviar instancias a un archivo en formato JSON.
- No requiere del uso de una librería externa, ya que el protocolo Codable es parte del Framework Foundation.

El uso de este protocolo ayuda a reemplazar el tener que usar un diccionario de manera eventual como sucede cuando se implementa la clase JSONSerialization, ya que, con la implementación de este protocolo, solo es necesario declarar las instancias necesarias que

posee el archivo en formato JSON, con esto además evitamos el uso de opcionales que haríamos con la clase ya mencionada anteriormente.

Listando 2.13. Ejemplo de implementación del protocolo Codable

```
[{“id”: 1, “name”: “user1”, “address”: “address1”},
{“id”: 2, “name”: “user2”, “address”: “address2”},
{“id”: 3, “name”: “user3”, “address”: “address3”}]
struct Person: Codable {
        var id: Int
        var name: String
        var address: String
    }
```

El protocolo Codable está compuesto de otros dos protocolos, los cuales son: Encodable y Decodable, como se había mencionado anteriormente el protocolo Codable nos permite realizar el proceso de serialización a través de clases y structs, para sustituir diccionarios y arreglos.

Si se quiere serializar un struct que implementa el protocolo Codable a un String literal en formato JSON, es necesario implementar el protocolo Encodable (véase Ejemplo 2.14), aquí se introduce la clase JSONEncoder, la cual se va a encargar de realizar el debido proceso de serialización.

Ejemplo 2.14. Implementación del protocolo Encodable, tomando como referencia el struct Person.

```
let person = Person ("id”: 1, “name”: “Peter”, “address”, “Peter’s address”)
let encoder = JSONEncoder ()
let jsonData = try encoder. encode(person)
```

```
String (data: jsonData, encoding: utf8)
```

Si la información se está recibiendo ya sea de un servidor o de un archivo en formato JSON, lo que se debe hacer es deserializar esa información utilizando el protocolo Decodable (véase Ejemplo 2.15).

Ejemplo 2.15. Implementación del protocolo Decodable

```
let decoder = JSONDecoder()
let decodedData = try decoder.decode(Person.self, from: jsonData)
type(of: decoded)
```

Una vez que la información es recibida de tal manera que se puede leer como una cadena JSON, posteriormente hay que inicializar dichos valores (véase Ejemplo 2.16).

Ejemplo 2.16. Inicializando valores con el protocolo Decodable

```
protocol Mappable: Decodable {
    Init?(jsonString: String)
}
extension Mappable {
    Init?(jsonString: String) {
        guard let data = jsonString.data(using: .utf8) else {
            return nil
    }
    self = try! JSONDecoder().decode(Self.self, from: data)
}
```

# 3. Diseño

## 3.1 Planteamiento de las clases.

La implementación que yo propongo para acelerar el desarrollo de una aplicación móvil a partir del parseo de JSON está ejemplificada en 3 distintas partes:

- Lectura de cualquier archivo en formato JSON.
- Serialización de coordenadas en un archivo de formato JSON.
- Renderización de una vista a partir de información en un archivo de formato JSON.

Para el desarrollo de esta clase se creó una clase especializada en la lectura de cualquier archivo en formato JSON, llamada "ReadJsonFile", la cual va a tener atributos que servirán como contenedores dependiendo del tipo de dato que se vaya leyendo en el archivo. Además de que para leer el archivo va a implementar la clase JSONSerialization, la cual fue descrita anteriormente. Esta clase va a contener sólo una función, la cual se encargará de leer el archivo, esta función fue nombrada como "readAnyJsonFile".

El objetivo principal de la función es tomar cualquier archivo en formato JSON, leer cada llave con su valor asociado y dependiendo del tipo del valor asociado, colocar el valor en su contenedor correspondiente, para posteriormente usar dichos valores en una aplicación móvil dado el uso que se quiera hacer de la información obtenida (véase Ejemplo 3.1).

```
20    func readAnyJsonFile(File:String){
21
22        do {
23            if let file = Bundle.main.url(forResource: File, withExtension: "json") {
24                let data = try Data(contentsOf: file)
25                let json = try JSONSerialization.jsonObject(with: data, options: [])
26                if json is [String: Any] {
27
28                } else if let object = json as? [Any] {
29
30                    for index in 0...object.count-1 {
31                        let getKey = object[index] as! [String : AnyObject]
32                        for (key, value) in getKey {
33                            print("key \(key) value \(value)")
34                            if(value is String){
35
36                                getJsonItems.append(value as! String)
37                            }
38                            if(value is Int || value is NSNumber){
39
40                                getJsonItemsInt.append(value as! Int)
41                            }
42                        }
43
44                    }
```

Ejemplo 3.1. Función “readAnyJsonFile”

Esta clase puede leer cualquier archivo en formato JSON válido, como ya se ha explicado anteriormente, como son los archivos en formato JSON que pueden o no ser válidos (véase Ejemplo 3.2). El ejemplo que utilizamos nos da un mejor panorama de cómo está clase puede ser utilizada para obtener información de un servidor, ya que la respuesta de una petición, sería la misma.

Para una mejor visualización de cómo se pueden interpretar los valores del archivo en formato JSON, se muestra una impresión de los datos con sus respectivas llaves y cómo esta información puede ser manipulada en una aplicación móvil para un fin determinado. El Ejemplo 1.18. Muestra del contenido de un archivo en formato JSON válido.

```
1  [
2  {
3  "id": 1,
4  "name": "Leanne Graham",
5  "username": "Bret",
6  "email": "Sincere@april.biz",
7  "address": {
8  "street": "Kulas Light",
9  "suite": "Apt. 556",
10 "city": "Gwenborough",
11 "zipcode": "92998-3874",
12 "geo": {
13 "lat": "-37.3159",
14 "lng": "81.1496"
15 }
16 },
17 "phone": "1-770-736-8031 x56442",
18 "website": "hildegard.org",
19 "company": {
20 "name": "Romaguera-Crona",
21 "catchPhrase": "Multi-layered client-server neural-net",
22 "bs": "harness real-time e-markets"
23 }
24 },
```


Ejemplo 3.2. Contenido de un archivo en formato JSON válido.

El ejemplo 3.3. Muestra del resultado, luego de serializar la información de un archivo en formato JSON, dividiendo las llaves con sus respectivos valores.


```
key address value {
    city = Gwenborough;
    geo =     {
        lat = "-37.3159";
        lng = "81.1496";
    };
    street = "Kulas Light";
    suite = "Apt. 556";
    zipcode = "92998-3874";
}
key phone value 1-770-736-8031 x56442
key company value {
    bs = "harness real-time e-markets";
    catchPhrase = "Multi-layered client-
server neural-net";
    name = "Romaguera-Crona";
}
key username value Bret
key name value Ervin Howell
key email value Shanna@melissa.tv
key id value 2
key website value anastasia.net
key address value {
    city = Wisokyburgh;
    geo =     {
        lat = "-43.9509";
        lng = "-34.4618";
    };
```

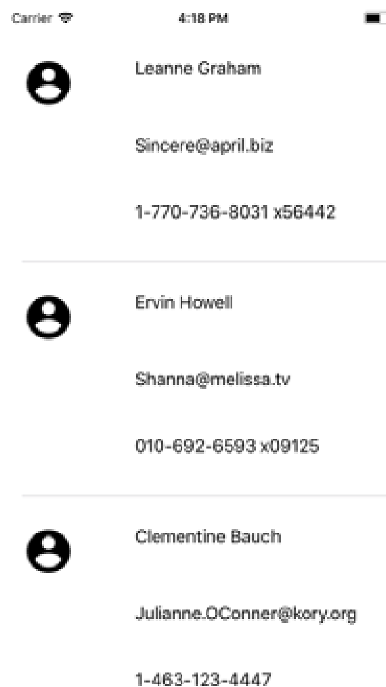
Ejemplo 3.3. Resultado luego de serializar la información de un archivo en formato JSON

Una vez extraída la información hay una gran gama de opciones, para poder representar está información dentro de una aplicación móvil. En IOS, las estructuras visuales más comunes para visualizar información son:

1 Table Views.

2 Collection Views.

Ambas estructuras se encuentran en la mayoría de las aplicaciones que día a día utilizamos los usuarios de Apple. Tomando en cuenta las mejoras prácticas y la experiencia de usuario. Una manera de ejemplificar el uso de información del archivo en formato JSON mostrado anteriormente, es demostrado en un Table View (véase Ejemplo 3.4)




Ejemplo 3.4. Ejemplo de manipulación de la información obtenida del archivo en formato JSON para hacer un directorio.

Un sin fin de aplicaciones actualmente usan mapas, puntos de geolocalización, coordenadas, rastreo de ubicación en tiempo real. Aplicaciones como: Uber, Uber Eats, son

de las más conocidas, si bien el uso de coordenadas puede servir para una infinidad de aplicaciones aún no desarrolladas, la mayoría de las veces, estas coordenadas son extraídas de un servidor ya sea que este mismo este alojado en la nube mediante algún servicio, un servidor compartido o un servidor mismo de la misma empresa que realiza la aplicación. Siempre es necesario deserializar está información en las aplicaciones para que puedan ser utilizadas para el fin determinado de la aplicación.

La mejor manera para implementar el uso de mapas y coordenadas en IOS, es utilizando el framework MapKit, el cual provee de manera gratuita un API para mostrar mapas en aplicaciones móviles, navegar entre ellos, agregar puntos de localización.

A partir de IOS 9, el framework MapKit permite a los desarrolladores la opción de customizar los puntos de localización (pin), además de características de tránsito y flujo, entre otras.

En este caso particular, se utilizarán las coordenadas extraídas de un archivo en formato JSON de una manera tal cual, que el desarrollo de cualquier aplicación que use mapas y coordenadas sea más rápido.

Se desarrolló una clase a la cual se nombró como CoordinatesJson, la cual nos permitirá ya sea decodificar coordenadas que son obtenidas a partir de un servidor, o un archivo local en formato JSON, como también codificar coordenadas y mandarlas a un servidor.

También se creó una clase llamada Location (véase Ejemplo 3.5), la cual contiene un modelo de lo que es una localización, contiene atributos como lo son: latitud, longitud y descripción, dentro de la ya nombrada clase se creó un constructor, cuya función es inicializar los valores de longitud y latitud.

```
11 class Location: Codable, CustomStringConvertible {
12 
13     var latitude: String
14     var longitude: String
15     var description: String{
16         return latitude + "/" + longitude
17     }
18 
19     init(){
20         self.latitude = "0.0"
21         self.longitude = "0.0"
22     }
23 
24     init(latitude: String, longitude: String){
25         self.latitude = latitude
26         self.longitude = longitude
27     }
28 }
```

Ejemplo 3.5. Clase Location

Durante los últimos años, el desarrollo de aplicaciones ha ido incrementando de una manera exponencial y estudios demuestran que esto continuará siendo así por lo que como se mencionó anteriormente, el tiempo de desarrollo de estas aplicaciones debe ser más rápido, pero sin que pierda la calidad o la funcionalidad de este mismo.

Últimamente, una tendencia sobre el desarrollo de aplicaciones híbridas que pueden ser desarrolladas en frameworks Web, ha ido tomando fuerza debido a que su tiempo de desarrollo es rápido, sin embargo, tiene un costo, y es que al realizar aplicaciones de este tipo se pierde tanto funcionalidad del sistema nativo como también características únicas del mismo sistema.

Una alternativa que se propone en la presente tesis para acelerar el desarrollo de aplicaciones nativas, es facilitar el desarrollo de la vista únicamente mediante renderización a través de archivos en formato JSON. En otras palabras, el objetivo es manipular la vista de una aplicación móvil, con tan solo modificar los valores de un archivo ya existente, por

el momento se ha trabajado con los elementos más comunes de una interfaz en IOS como lo son:

- Botones
- Campos de texto
- Etiquetas

Actualmente podemos manipular desde un archivo en formato JSON, propiedades como lo pueden ser: color, opacidad, visibilidad, posición, contenido entre otras.

La idea principal es tener un conjunto de vistas genéricas, para que un desarrollador pueda descargar dicha vista, y solo tenga que manipular el archivo del formato mencionado para adecuar la interfaz acorde a sus necesidades, de este modo nos aseguramos que el tiempo de desarrollo será más rápido, ya que un problema habitual de los desarrolladores de IOS, el tener que manipular *constraints* para que la vista en todos los dispositivos se vea de manera presentable, ya que inclusive desde el archivo, se pueden manipular las *constraints,* se espera que el tiempo de desarrollo logre bajar de manera considerable (véase Ejemplo 3.6).

```
1   [
2    {
3
4    "UIButton_double_values": {
5    "r": 0.4,
6    "g": 1.0,
7    "b": 0.2,
8    "alpha": 0.5,
9    "x": 200,
10   "y": 400,
11   "width": 100,
12   "height": 100
13
14   },
15   "title": "hello World",
16   "hidden": false,
17   "focused": true,
18   "UILabel": {
19   "r": 1.4,
20   "g": 1.0,
21   "b": 0.2,
22   "alpha": 0.5,
23   "x": 200,
24   "y": 200,
25   "width": 100,
26   "height": 50
```

Ejemplo 3.6 Estructura un archivo en formato JSON para renderizar elementos.

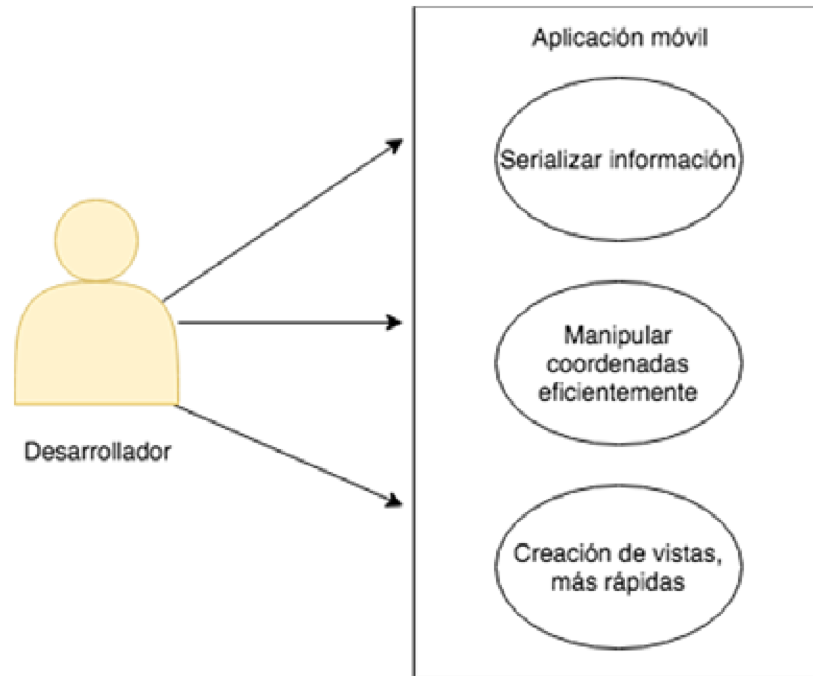
Para lograr los objetivos mencionados anteriormente se desarrolló una clase llamada Templating en la cual se podrán encontrar métodos dependiendo del tipo de objeto de interfaz que se desee agregar.

Hay varias propiedades que todos los objetos de interfaz tienen, puesto que estas mismas propiedades pueden tomarse como globales (véase Ejemplo 3.7)

```
49        let button = template.RederUIButton(File: "templating")
50        let label = template.RederUILabel(File: "templating")
51        let textField = template.RederUITextField(File: "templating")
52
53        self.view.addSubview(button)
54        self.view.addSubview(label)
55        self.view.addSubview(textField)
```

Ejemplo 3.7 Propiedades comunes de los objetos de interfaz

## 3.2 Casos de uso



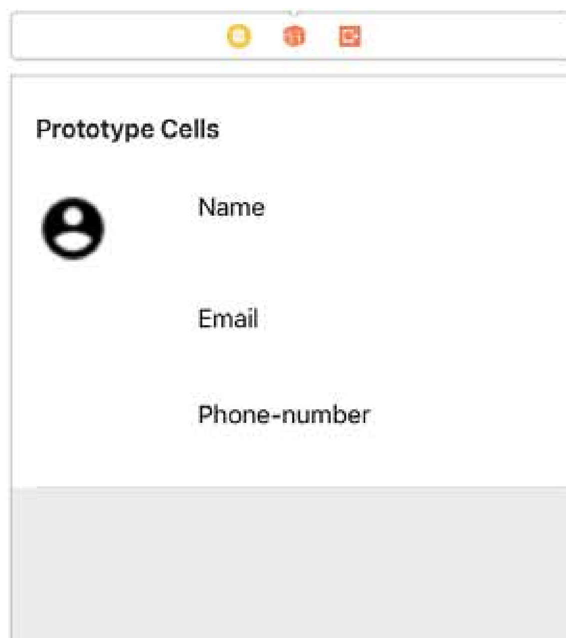
## 4. Prototipo

Los prototipos mostrados a continuación dependen del uso dado de cada clase, para cada caso uso se ejemplifica el funcionamiento de la librería y como efectivamente muestra que ayuda al momento del desarrollo de una aplicación móvil.

Toda la información es obtenida de un servidor, que se encuentra en la nube utilizando los servicios que provee Amazon Web Services como infraestructura, de esta manera podemos asegurar que todas las aplicaciones mostradas a continuación utilizan información de manera persistente.

Las vistas se desarrollaron de acuerdo a la documentación y lineamientos que marca Apple sobre el diseño óptimo que debe tener una aplicación en IOS, de tal manera que se utilizaron Framework muy comunes y objetos de interfaz que son utilizados en la mayoría de las aplicaciones en IOS, de igual manera se puede decir que todas las pruebas mostradas a continuación, están basadas en un modelo de 3 capas de una aplicación, ya que tenemos la parte visual de la aplicación, la parte lógica y la parte de persistencia de información.

Para ejemplificar el uso de lectura de cualquier archivo en formato JSON, se utilizó un TableViewController, con campos correspondientes a la información obtenida del servidor, para posteriormente ser procesados y mostrados de acuerdo a la estructura mostrada en la Figura 4.1.




Ejemplo 4.1 Estructura de la Tabla con celdas reusables.

Al ser información mostrada en un TableViewController, se puede crear una clase que herede de la super clase TableViewController y el método que va a definir lo que debe llevar cada celda reusable es el método "cellForRowAt", de esta manera como se ejemplifica en la Figura 4.2.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
        as? DataTableViewCell
    cell?.label1.text = names[indexPath.row]
    cell?.label2.text = email[indexPath.row]
    cell?.label3.text = phone[indexPath.row]

    // Configure the cell...

    return cell!
}
```

Ejemplo 4.2 Método "cellForRowAt".

Para el caso de manipulación de coordenadas:

Para comenzar si se desea realizar el proceso de decodificación de coordenadas, se precisa invocar la función jsonDecoding (véase Ejemplo 4.3), la cual va a recibir un parámetro de tipo Data y va a regresar un objeto de clase Location, aquí es donde se obtiene la latitud y longitud.

```
71      private func jsonDecoding(jsonData: Data) -> Location{
72          var jsonDecoder: JSONDecoder
73          var stringCoordenates: Location
74          stringCoordenates = Location()
75          jsonDecoder = JSONDecoder()
76          do{
77              stringCoordenates = try jsonDecoder.decode(Location.self, from: jsonData)
78              print("latitude: \(stringCoordenates.latitude)")
79              print("longitude: \(stringCoordenates.longitude)")
80          }catch{
81              print("Error while decoding JSON")
82          }
83          return stringCoordenates
84      }
85
```

Ejemplo 4.3 Función jsonDecoding

En el caso contrario si se requiere enviar información a un servidor, se utilizará la función jsonEncoding, la cual tomará como parámetro un objeto de clase Location y regresará un objeto tipo Data (véase Ejemplo 4.4).

```
55
56      private func jsonEncoding(codable: Location) -> Data{
57          var jsonEncoder: JSONEncoder
58          var jsonData: Data
59
60          jsonData = Data()
61          jsonEncoder = JSONEncoder()
62          do{
63              jsonData = try jsonEncoder.encode(codableObject)
64            // jsonString = String(data: jsonData, encoding: .utf8)
65          }catch{
66              print("Error while encoding JSON.")
67          }
68          return jsonData
69      }
```

Ejemplo 4.4. Implementación de la función jsonEncoding

Para utilizar las coordenadas en una aplicación de manera efectiva y así obtener el punto exacto de la localización a la cual se apunte, se utiliza el método CLLocationCoordinate2DMake. Esta función toma los valores de longitud y latitud y los convierte en una estructura de coordenada que Swift puede interpretar (véase Ejemplo 4.5).

```
63                              annotation.coordinate =
                                    CLLocationCoordinate2DMake( self.routes[i].latlon![0],
                                    self.routes[i].latlon![1])
64                              annotation.title = self.routes[i].title
65                              annotation.subtitle = self.routes[i].title
66
67
68                              self.mapView.addAnnotation(annotation)
69
```

Ejemplo 4.5. Ejemplo de implementación de la función CLLocationCoordinate2DMake ().

Como resultado de las clases y funciones mencionadas anteriormente se puede lograr el desarrollo de aplicaciones que utilicen coordenadas, disminuyendo a la mitad el tiempo de desarrollo (véase Ejemplo 4.6).

Ejemplo 4.6. Aplicación desarrollada a partir de las clases sobre coordenadas.

En el caso de vistas dinámicas, primero es necesario tener listo el archivo en formato JSON, sobre el cual se hará ña manipulación de la vista, como se muestra a continuación:

```
1  [
2   {
3
4   "UIButton_double_values": {
5   "r": 0.4,
6   "g": 1.0,
7   "b": 0.2,
8   "alpha": 0.5,
9   "x": 200,
10  "y": 400,
11  "width": 100,
12  "height": 100
13
14  },
15  "title": "hello World",
16  "hidden": false,
17  "focused": true,
18  "UILabel": {
19  "r": 1.4,
20  "g": 1.0,
21  "b": 0.2,
22  "alpha": 0.5,
23  "x": 200,
24  "y": 200,
25  "width": 100,
26  "height": 50
```

Ejemplo 4.7 Archivo en formato JSON para la creación de vistas dinámicas

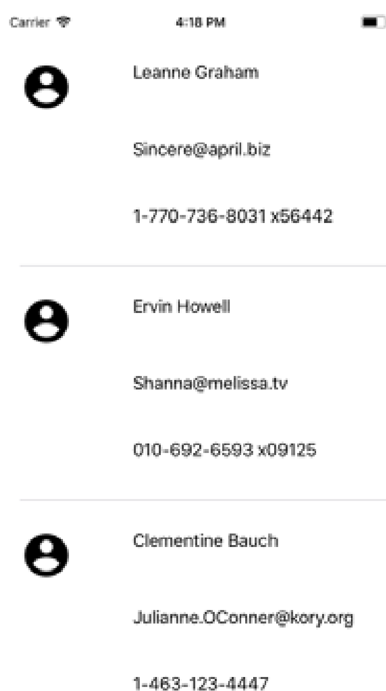
Posteriormente dependiendo del objeto deseado se deberá invocar su método para que se pueda visualizar en la vista de la aplicación (véase Ejemplo 4.8).

```
49        let button = template.RederUIButton(File: "templating")
50        let label = template.RederUILabel(File: "templating")
51        let textField = template.RederUITextField(File: "templating")
52
53        self.view.addSubview(button)
54        self.view.addSubview(label)
55        self.view.addSubview(textField)
```

Ejemplo 4.8 Forma para mostrar un objeto de interfaz

## 5. Programación

Los resultados obtenidos fueron bastante positivos ya que en el caso de la clase sobre lectura de cualquier archivo en formato JSON, hubo una gran mejora debido a que la clase ya obtiene directamente todos los valores relacionados con las llaves, esto hace que al manipular la información y quererla presentarla, se vuelva una tarea más sencilla, en este caso ejemplificamos un directorio de personas, con la información obtenida del archivo, se muestran datos personales como nombre, correo electrónico y número telefónico. Claramente de este ejemplo se pueden partir muchas funcionalidades por demás.




Ejemplo 5.1 Directorio personal

Como se había mencionado anteriormente, se utilizaron objetos de interfaz que son muy comunes en aplicaciones de sistema operativo IOS, esto es debido a que siempre hay

que tener en cuenta a la experiencia de usuario, un usuario de IOS tiene el concepto de experiencia de usuario muy diferente a lo que acostumbra un usuario de Android.

Para el caso de las clases sobre manipulación de coordenadas, los resultaron también fueron positivos ya que al momento de manipular dichas coordenadas en la aplicación se volvió un trabajo más sencillo, debido a que, al momento de recibir la información al servidor, automáticamente se separan las latitudes y las longitudes, puesto que lo único que resta es colocar dicha información en las anotaciones del mapa

Ejemplo 5.2 Información en formato JSON y ejemplo de anotación

En el ejemplo posterior se puede apreciar cómo se pueden agregar varias ubicaciones en un mapa, a cada ubicación le corresponde una descripción, tal cual se había descrito en la clase Location ya mencionada anteriormente

Ejemplo 5.3 Varias ubicaciones mostradas en un mapa.

## 6. Compatibilidad de versiones

El desarrollo de esta librería está realizado en el lenguaje de programación Swift en su versión 4. Si un desarrollador que aún utiliza Swift 3 quiere hacer uso de la librería, en general no tendría ningún problema con el código ya que no hay mucha variación de sintaxis entre una versión y otra. Pero con la llegada de Swift 4 vinieron grandes mejoras y cambios como lo son la creación del protocolo Codable, que es en parte de donde se basa esta librería, también trajo consigo el primer gestor de dependencias nativo de Swift, llamado Swift PM, el cual permite incorporar librerías de terceros que puedan extender del mismo lenguaje Swift, ante estos grandes cambios, Apple siempre recomienda estar lo más actualizado posible en cuanto a la versión de Xcode que se utiliza. Sin embargo, los mayores cambios en cuanto al lenguaje se presentaron cuando hubo la migración de Swift 2 y Swift 3, donde la diferencia de sintaxis era más que evidente, incluso se presentaron varias críticas ya que muchas aplicaciones tuvieron que cambiar más del 50% su estructura.

## Conclusiones

En la actualidad el desarrollo de aplicaciones móviles nativas requiere de un desarrollo más práctico sin perder la calidad de las aplicaciones, gracias a esta librería se ha demostrado que se pueden facilitar varias tareas al momento de realizar una aplicación móvil en IOS, como lo es la lectura de cualquier archivo en formato JSON, separando las llaves de sus valores, como también la codificación de información. En cuanto a aplicaciones que ocupen la implementación de mapas, se ha demostrado una alternativa para serializar las coordenadas de una manera más práctica y eficiente, finalmente demostramos que se pueden crear vistas dinámicas para facilitar la creación solamente de la vista de una aplicación, desde un archivo en el formato ya mencionado.

**Referencias**

Ahmad Jawwad. (2018). IOS 11, Virginia, USA. Editorial: Wanderlich.

Apple, Inc. (2018), Version Compatibility. Consultado el 20 de abril de 2018. Recuperado de:_HYPERLINK "https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Compatibility.html".

Hollemans, Matthijs. (2017). IOS Apprentice, Virginia, USA. Editorial: Wanderlich.

Ng, Simon. (2018). Intermediate IOS Programming with Swift, Hong Kong. Editorial: AppCoda.

Selander Derek. (2018). Apple debugging, Virginia, USA. Editorial: Wanderlich.