Tutorial Chat Simple en Xamarin.iOS

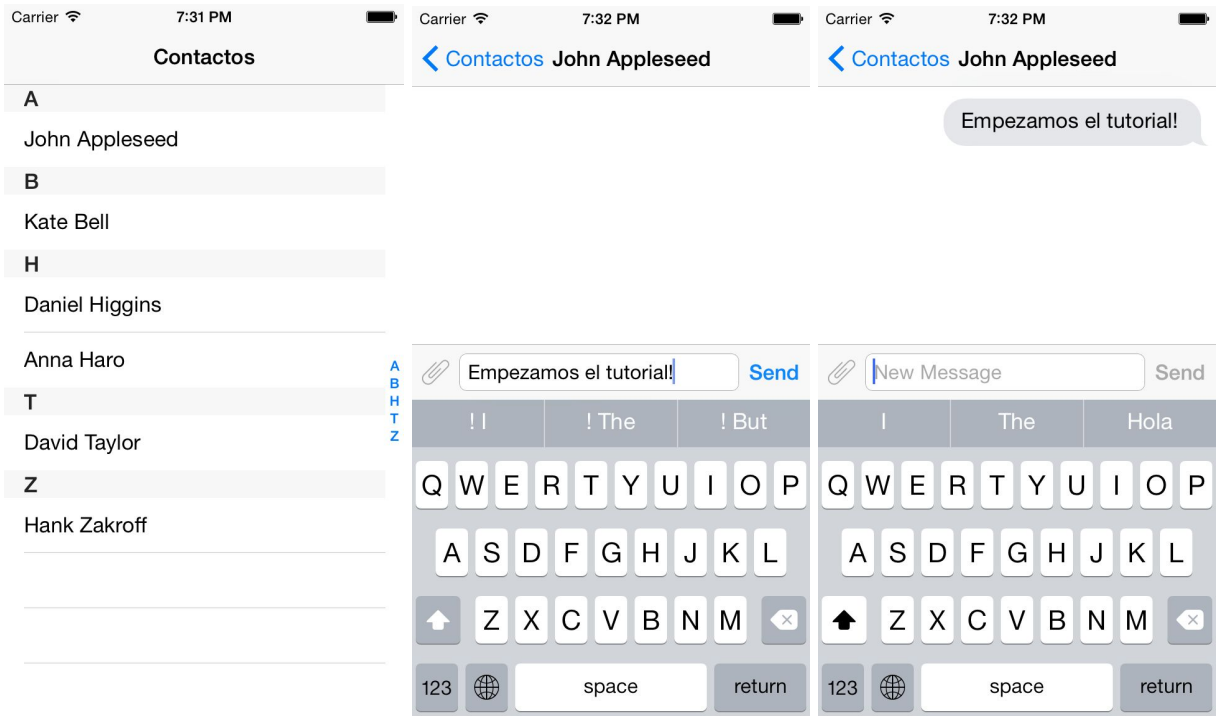
por Luis Hernán Cubillos

Contenidos:

1. Introducción:

En los últimos años hemos observado cómo ha ido creciendo la popularidad de servicios de mensajería instantánea para celulares, siendo WhatsApp el ejemplo más claro de ello. ¿Qué los hace tan especiales? La verdad el servicio de mensajería en sí es bastante simple de hacer (en eso consiste este tutorial de hecho), pero lo que los hace únicos es su tremenda estabilidad y su baja cantidad de *bugs*, junto con varias funcionalidades agregadas.

En este tutorial intentaré explicar de la manera más fácil y simple cómo hacer un Servicio de Mensajería para iPhone, usando Xamarin.iOS. La aplicación va a consistir básicamente en mostrar los contactos del teléfono, ordenados por apellido y nombre, y que cuando uno presione cualquier contacto, pueda iniciar una conversación con él/ella. Al final, podrán tener una conversación con cualquiera de sus contactos, sin problemas.



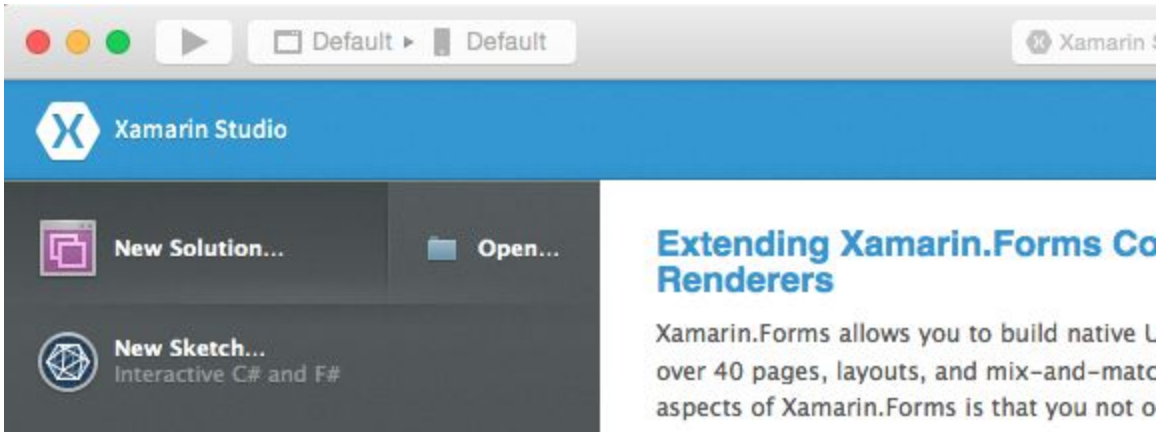
Para los que no sepan, les cuento un poco en qué consiste Xamarin. La empresa surgió en el año 2001 con el afán de hacer la programación de aplicaciones móviles más simple. Es por esto que lograron reunir la programación en Android, iOS, OSX y Windows Phone en tan solo un lenguaje (C#), con una interfaz simple e intuitiva. Hoy, Xamarin tiene oficinas en 4 países, y más de un millón de personas están desarrollando con esto.

Para este tutorial, asumiré creada una API web, ya que escapa un poco del alcance de este tutorial. Esta API tiene que ser capaz de soportar métodos POST y GET desde el código, para obtener y crear usuarios, y para mandar y recibir mensajes. En el desarrollo del tutorial iré contando la API que usé yo, especificando los cambios que tienen que hacer según la API que ocupen[1]. Además, asumo que tienen un nivel medio de aprendizaje de C#.

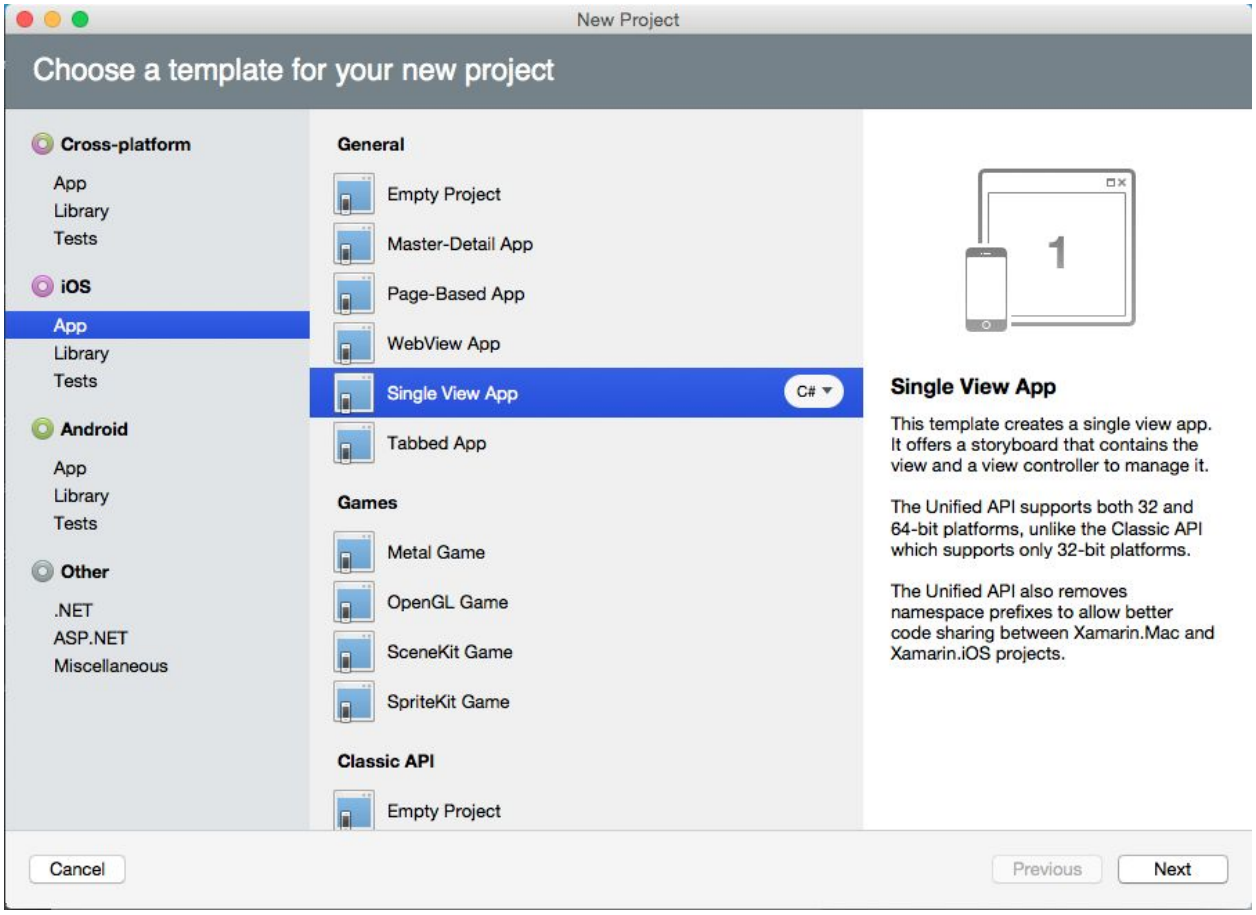
2. Primeros Pasos: Creando una nueva Solución.

Partamos entonces con lo básico. Xamarin trabaja con los conceptos de Solución y Proyecto, tal como se trabaja en .NET. Para crear una nueva solución simplemente presionamos “New Solution”, tal como se aprecia en la siguiente imagen.

[1] TIP: Para los más avanzados, en Heroku pueden encontrar una plataforma que soporta creación de APIs en varios lenguajes, tales como Ruby, Python y otros.



Luego, hay varias opciones. En esta ventana se puede elegir hacer una app para Android o iOS, o ambas, y también trabajar directo con .NET. En nuestro caso en particular, elegiremos la opción iOS -> General -> Single View App, y presionamos Next.



En la siguiente ventana elegimos el nombre de la solución, el identificador, y los dispositivos objetivo. Luego de rellenar con sus propias preferencias, presionen Next, y ya habremos creado la nueva solución.

Una explicación corta de qué significan algunos elementos de la solución, y para qué se usa[2]. La clase Main.cs es la que se llama primero al correr la app, y simplemente llama a AppDelegate para que comience a trabajar en la interfaz gráfica. La clase ViewController es el controlador del View inicial que se crea junto con la solución. Ésta la veremos más en profundidad. Para aclarar este último punto, el funcionamiento de una app en iOS es a través de diferentes Views y segues (transiciones) entre ellas. Cada View tiene asociado un ViewController, que se encarga de actualizar la interfaz gráfica y de llamar al Backend en caso de ser necesario. Además, la solución incluye un archivo llamado Main.storyboard, que nos da la opción de editar la interfaz gráfica de forma simple e intuitiva, relacionándose de manera automática con el designer del ViewController. Xamarin hace todo bastante fácil la verdad.

Ahora, respecto a la clase ViewController, que es en general la que más usaremos, hay una serie de métodos claves, que los iré comentando a medida que vayamos avanzando en el tutorial. Por ahora, diré que el método ViewDidLoad() es el que se corre apenas la View se carga, y es ahí donde conviene realizar los procesos para mostrar distintas cosas en la misma ventana.

```
public partial class ViewController : UIViewController
{
        public ViewController (IntPtr handle) : base (handle)
        {
        }

        public override void ViewDidLoad ()
        {
            base.ViewDidLoad ();
            // Perform any additional setup after loading the view, typically from a nib.
        }

        public override void DidReceiveMemoryWarning ()
        {
            base.DidReceiveMemoryWarning ();
            // Release any cached data, images, etc that aren't in use.
        }
}
```

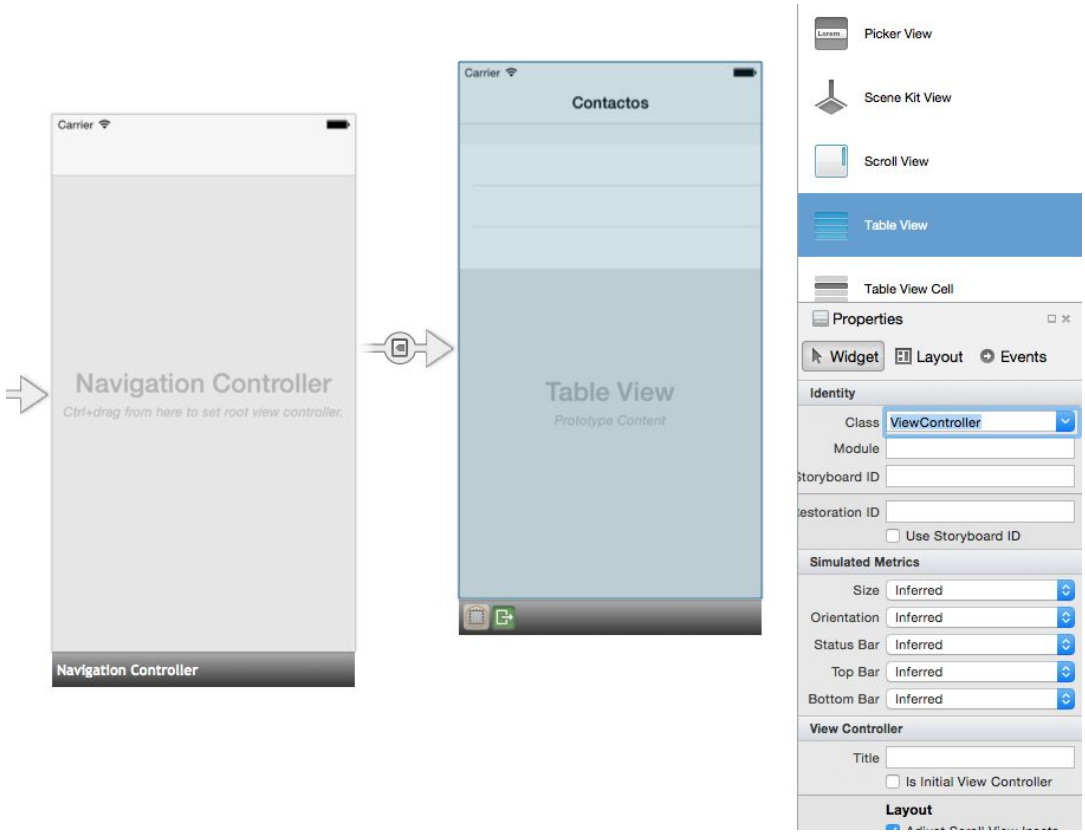
Ahora empieza lo entretenido, comenzamos trabajando con el StoryBoard.

3. Trabajar con el StoryBoard: Crear TableView dentro de ViewController

Para comenzar, abrimos el archivo StoryBoard de nuestra solución. Vamos a encontrarnos con una View predeterminada, que si uno la presiona y ve sus propiedades, puede ver que está controlada por la clase ViewController, de la que hablaba antes. Nuestra app consistirá de múltiples Views, por lo que necesitamos un Navigation Controller que dirija

[2] Para más información, ve la guía de Xamarin sobre el tema.

estas transiciones. Este último no solo controla las segues, sino que también agrega una barra superior a cada vista en la que aparece el título y un botón para devolverse a la vista anterior. Agregamos este último, borramos la View que teníamos antes, reemplazamos la nueva View por un TableView, y ponemos que el controlador de la View nueva sea el ViewController creado antes.



Esta vista con el TableView será nuestra vista inicial y principal, en la que mostraremos todos los contactos del teléfono. Además, debemos hacer una vista extra, que será el chat mismo con cada contacto. La creamos en este momento, pero lo dejamos sin controlador. Empezamos ahora a poblar la tabla con los contactos del teléfono.

4. Poblar la tabla con los contactos del teléfono.

Para poblar las tablas en Xamarin.iOS es necesaria una nueva clase. Ésta, debe heredar de *UITableViewSource*, e implementar 2 métodos por obligación, y varios otros para modificar aún más la presentación de la tabla. Estos métodos son: *GetCell* y *RowsInSection*. El primero, permite explicitar qué contenido se muestra en una celda específica, y cómo es la celda en sí. El segundo, dice cuántas celdas tiene que mostrar la tabla por sección. Además, métodos como *NumberOfSections* y *RowSelected*, permiten trabajar más profundamente en la tabla.

4.1 ContactosTableSource

En este caso llamaré a esta clase, *ContactosTableSource*, que debe heredar de *UITableViewSource*. Con esta clase implementaremos de dónde saca la tabla sus contenidos, y cómo los muestra.

Primero que nada, debemos obtener el *AddressBook* del propio iPhone. Para esto primero debo pedir permiso para acceder a los contactos, y si se permite, obtener una lista de todos ellos. Además, los ordenaremos por apellido y luego por nombre, así que modificamos el método anterior, para que devuelva un *Dictionary<string,List<ABPerson>>*, que separe a todos los contactos por la primera letra de su apellido (que será la key del diccionario) y luego ordene la lista de las personas con el apellido correspondiente.

```
//Fills contacts and initials with the data on the phone.
public void GetAddressBook()
{
    NSError err;
    var addressBook = ABAddressBook.Create (out err);
    //The user must agree to let us use his contacts.
    addressBook.RequestAccess ((bool haveAccess, NSError e) => {
        if (!haveAccess) {
            Console.WriteLine ("No access, you must accept to use the app");
        } else{
            Console.WriteLine ("Access granted");
        }
    });

    contacts = new Dictionary<string, List<ABPerson>> ();
    var persons = addressBook.GetPeople ();

    foreach (ABPerson person in persons) {
        string firstLetterLastName;
        if (person.FirstName != null && person.LastName != null)
            firstLetterLastName = person.LastName [0].ToString ().ToUpper ();
        else if (person.FirstName != null)
            firstLetterLastName = person.FirstName [0].ToString ().ToUpper ();
        else
            continue;
        if (contacts.ContainsKey (firstLetterLastName))
            contacts [firstLetterLastName].Add (person);
        else {
            contacts.Add (firstLetterLastName, new List<ABPerson> ());
            contacts [firstLetterLastName].Add (person);
        }
    }

    foreach (KeyValuePair<string,List<ABPerson>> kp in contacts) {
        initials.Add (kp.Key);
        kp.Value.OrderByDescending (persona => persona.LastName);
    }
    initials.Sort ();
}
```

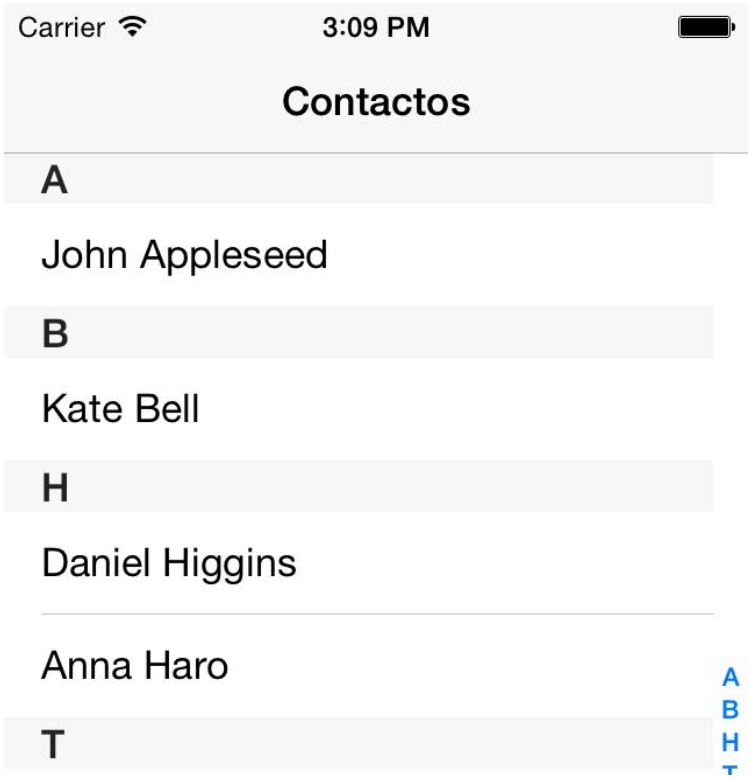
Ahora, para mostrarlo en la tabla, agregamos el código a la clase *ContactosTableSource*. La lista de contactos será la fuente de información para la tabla. En cada celda mostraremos un contacto. Además, los ordenaremos por apellido, tal como se hace en la aplicación de contactos del teléfono. Para esto, debemos implementar los 4 métodos de los que hablaba antes, y también *TitleForHeader* y *SectionIndexTitles*, de manera de poder visualizar el índice de la misma forma que en los contactos. Ejemplo de implementación es la siguiente:

```
public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
{
    //This code snippet allows us to reuse a cell to show content. iOS tables work this way,
there is a fixed number of cells, and
    //you have to reuse the ones that are not on the user interface.
    UITableViewCell cell = tableView.DequeueReusableCell (cellIdentifier);
    if (cell == null)
        cell = new UITableViewCell (UITableViewCellStyle.Default, cellIdentifier);

    //Cell content.
    var person = contacts [initials [indexPath.Section]] [indexPath.Row];
    cell.TextLabel.Text = person.FirstName +" "+ person.LastName;
    return cell;
}

public override nint RowsInSection (UITableView tableview, nint section)
{
    return (contacts [(initials.ToArray()) [section]].ToArray()).Length;
}

public override nint NumberOfSections (UITableView tableView)
{
    return initials.Count;
}

public override string TitleForHeader (UITableView tableView, nint section)
{
    return (initials.ToArray()) [section];
}

public override string[] SectionIndexTitles (UITableView tableView)
{
    return initials.ToArray ();
}

public override void RowSelected (UITableView tableView, NSIndexPath indexPath)
{
    var person = contacts [initials [indexPath.Section]] [indexPath.Row];
    (controller as ContactosViewController).contactToPass = person;
    controller.PerformSegue ("ToChat", controller);

    tableView.DeselectRow (indexPath, false);
}
```

Con esto tenemos listo el contenido que debe ir en la tabla que estamos mostrando en nuestra View principal. Hacemos las modificaciones correspondientes en el View Controller, para que podamos ver los resultados en el simulador.

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    //We give the table on the storyboard some data.
    var tableSource = new ContactosTableSource ();
    Contacts.ReloadData ();
    tableSource.controller = this;
    //Our tableview name is Contacs, and we pass it the source we created.
    Contacts.Source = tableSource;
}
```

Y finalmente corremos el programa y mostramos los resultados en el simulador:



Es necesario agregar que los contactos que estoy mostrando son los contactos que tiene el simulador por defecto.

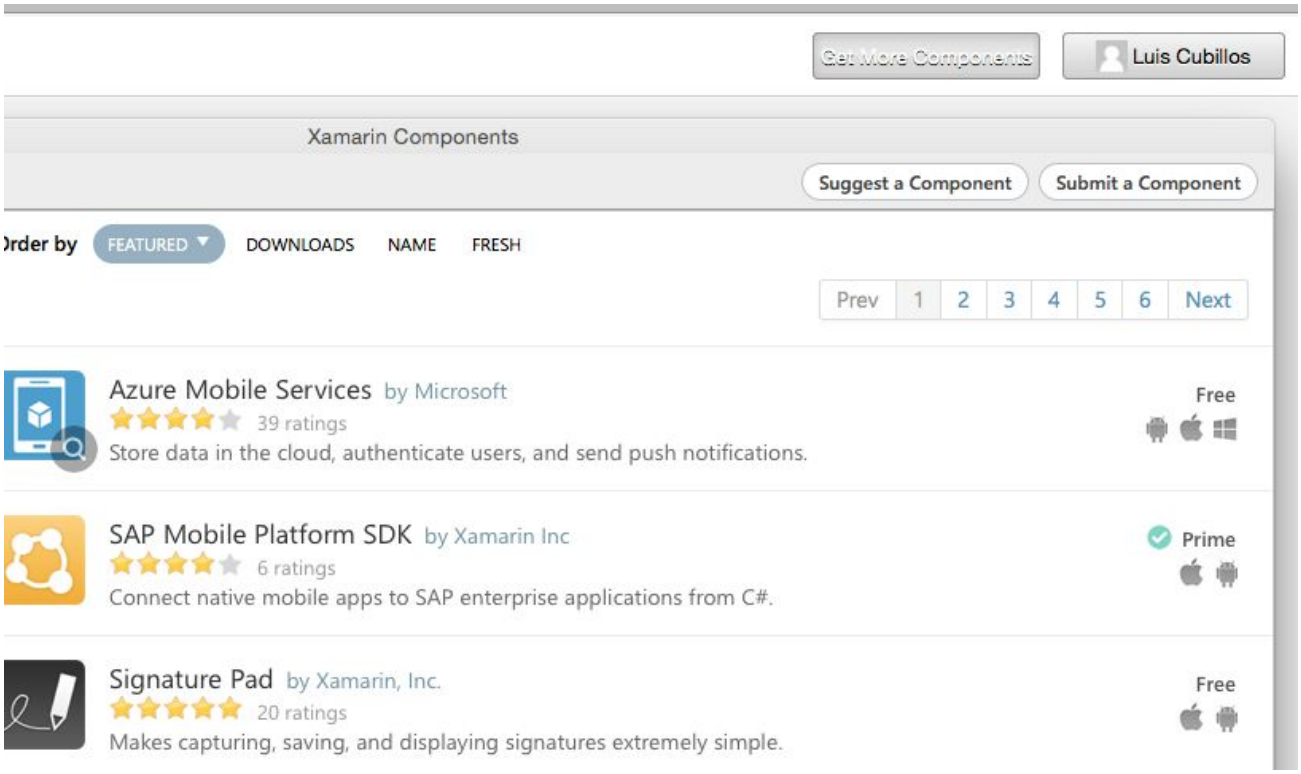
5. Mensajería: JSQMessagesViewController y chat.

Ahora, la parte central de este tutorial, ¿cómo lo hacemos para enviar y recibir mensajes en nuestra app? En esta sección haremos uso profundo de la API que antes comenté, así que si no la has creado, te lo recomiendo hacer ahora. Se hará uso del protocolo REST para hacer distintos request a la página, a través de los métodos GET y POST. Este protocolo es el más usado en aplicaciones móviles principalmente debido al pequeño número de requerimientos técnicos que tiene, lo que lo hace eficiente en el funcionamiento. Además, usaremos una gran herramienta que posee Xamarin para poder agregar Componentes creados por usuarios de todo tipo alrededor del mundo.

5.1 Agregar Componentes: JSQMessagesViewController y Json.NET.

Uno de los puntos altos de Xamarin, es una gran biblioteca de componentes de todo tipo que permiten ocupar códigos hechos por otros, algunos gratis y otros pagados. Así, podemos darnos el lujo de no reinventar la rueda, y tener un punto de partida más o menos alto. En nuestro caso en particular, ocuparemos dos componentes: JSQ Messages ViewController y Json.NET[3].

Agregar estas componentes es bastante simple. Ingresamos a la carpeta “Components” dentro de nuestro proyecto, y hacemos click en “Get More Components”.



Luego, en el buscador de componentes buscamos los dos que nos interesan: Json.NET y JSQMessagesViewController, y los agregamos al proyecto. Un pequeño comentario sobre

[3] El primero lo pueden encontrar aquí, y el segundo aquí.

ambos. El primero, es una librería que nos permitirá serializar y deserializar rápidamente objetos en formato json, lo que nos permitirá trabajar fácilmente con los WebRequest para el API. JSQMessagesViewController, por otro lado, entrega un controlador pre-hecho con una interfaz igual a la de los mensajes del iPhone, y editable a nuestro gusto. Se podría haber hecho una interfaz por nuestra cuenta, o serializar los objetos de la misma manera, pero usando los componentes ahorramos tiempo y dificultades en el proceso. Ambos son gratis, por lo que los podemos ocupar con libre disposición.

5.2 Agregar nuevo controlador: JSQMessagesViewController.

Ahora para crear un nuevo controlador ocupando JSQMessagesViewController, hay una serie de pasos a seguir. Primero, creamos una nueva vista en el StoryBoard, pero sin asignarle un controlador todavía. Luego, creamos un nuevo ViewController, que en este caso se llamará *ChatViewController*, que debe heredar de *MessagesViewController*. Para poder usar esta clase tenemos que incluir el componente en la clase (directiva "using"), y para coordinarla con el Storyboard, tenemos que registrarla como una clase con designer. Además, para ocuparlo normalmente debemos implementar un par de métodos y properties[4].

```
using Foundation;
using JSQMessagesViewController;
using System;


namespace ChatSimple
{
    //This line is MANDATORY, without it you can't use this controller on the storyboard.
    [Register ("ChatViewController")]
    public class ChatViewController : MessagesViewController
    {
        //This line is also mandatory to allow the use of this class on the storyboard.
        public ChatViewController(IntPtr handle):base(handle)
        {
        }
    }
}
```

Con esto, ya tenemos agregado nuestro *ChatViewController*, que nos permitirá mostrar de manera elegante e intuitiva los mensajes que mande y que reciba. Ahora, eso sí, tenemos que personalizar la vista para que cumpla con lo que queremos que haga.

De partida, haremos que muestre como título del Chat el nombre del contacto al que queremos enviar el mensaje. Además, agregaremos un botón que usaremos para ver si hay algún mensaje nuevo para mí en el servidor, y que luego los descargue. Sé que es una forma bastante burda de recibir los mensajes, pero es la más simple de las que existen. Para recibir los mensajes instantáneamente, habría que trabajar con Background Fetch, que está fuera del alcance de este tutorial[5].

[4] Ver la aplicación final para detalles de estos métodos.
[5] Xamarin tiene un tutorial bastante completo, para los que estén interesados.

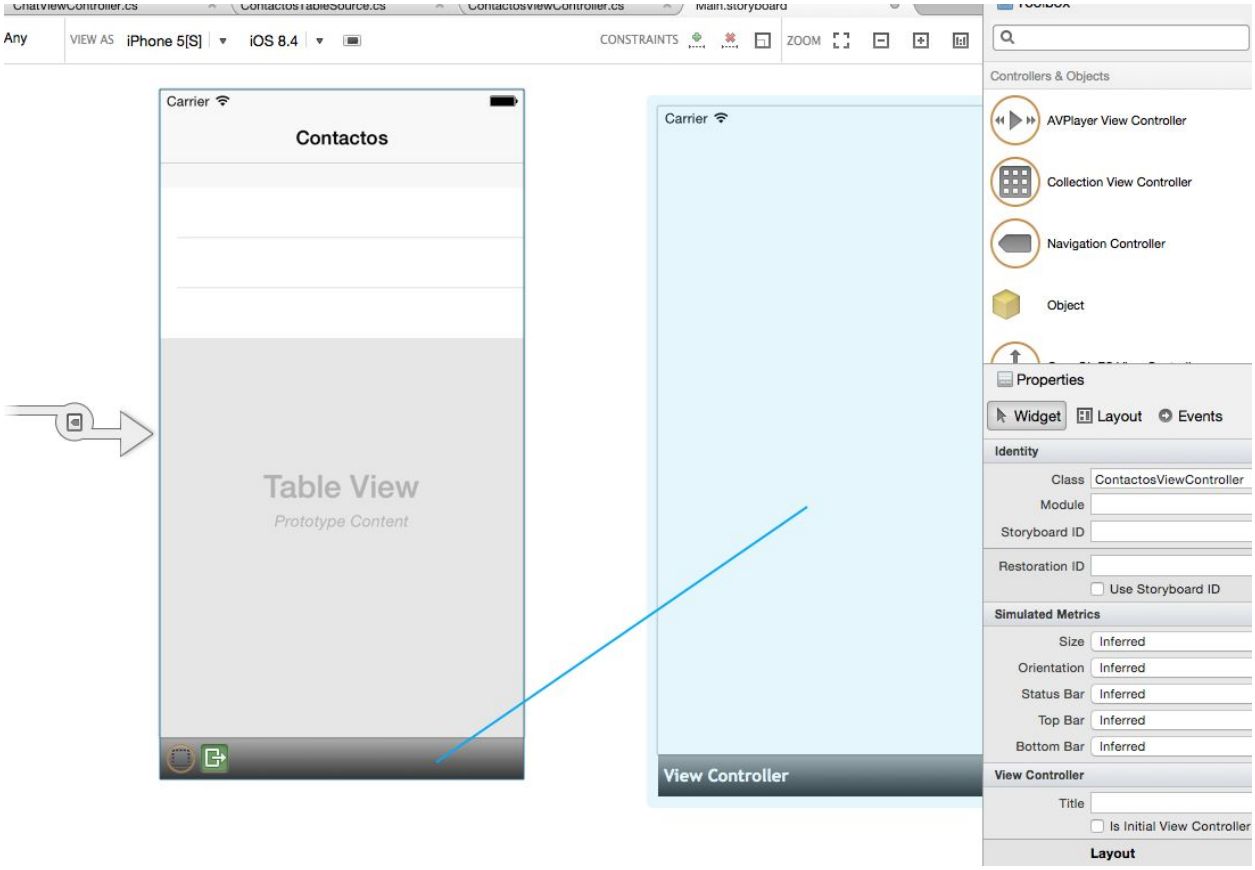
Lo que haremos entonces, es que el controlador del Chat tenga como atributo una *ABPerson*, que definiremos al realizar el *segue* (transición) entre el controlador principal y este. Además, dentro de la clase implementaremos el botón que va en la *InputToolBar*, que nos permitirá actualizar los mensajes.

Ahora, eso sí, hay un par de vacíos que hay que rellenar: Funcionamiento de la API, y características de los segues. El funcionamiento de la API se puede encontrar en el Anexo. Veamos los segues.

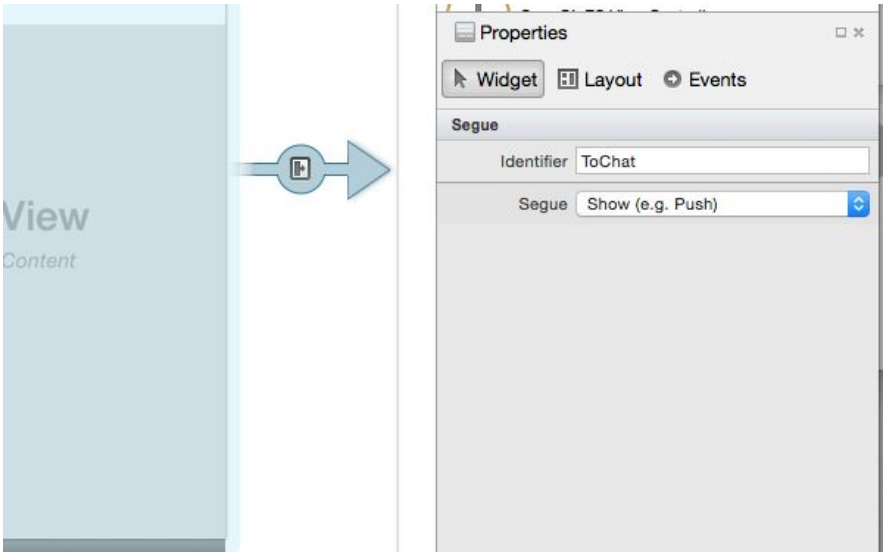
5.3 Segue:

La palabra *segue* significa “transición” en inglés, y es exactamente eso lo que se hace en las segues en iOS. Sirven para realizar un cambio entre una vista y otra. Ahora, existen varios tipos de Segue: *Show, Show Detail, Present Modally* y *Present as Popover*. En este caso, ocuparemos el tipo *Show*, que tiene el mismo efecto que antes de iOS 7 tenía el segue *Push*. Básicamente consiste en una transición simple entre una vista y otra, que permite devolverse fácilmente, ya que agrega un botón de retorno en la barra de navegación. Es muy ocupada en las aplicaciones básicas de iOS como la de Contactos.

Crear un segue es bastante simple con el Storyboard de nuestra aplicación. Abrimos el storyboard, y al tiempo que apretamos *ctrl*, arrastramos el mouse desde la zona inferior del ViewController de inicio, hasta cualquier parte del ViewController de llegada.



Luego, presionando el símbolo del Segue, podemos elegir el nombre (identificador) y el tipo de segue. En este caso, le pondremos *"ToChat"* y de tipo *Show*.



Los segues, además, permiten pasar información de una vista a otra con un método llamado PrepareForSegue(), que se define en el ViewController de origen. En este podemos usar el ViewController de destino, definiendo variables o llamando métodos.

En definitiva, todo segue a través del cuál uno quiere traspasar información, consiste en dos partes: la llamada al segue en sí, y la preparación del segue. Como algunos se darán cuenta, ya vimos la parte de la llamada al segue, que se realizaba en *ContactosTableSource* cuando una celda era presionada. Ahora, falta el método para prepararse para el segue. En nuestro caso, nosotros queremos pasarle al *ChatViewController* el contacto que presionamos, para poder realizar acciones con él. Para esto, definimos un atributo de *ABPerson* en ambos ViewControllers. Visto desde *ContactosViewController*:

```
partial class ContactosViewController : UIViewController
{
     ...
   public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)
   {
     base.PrepareForSegue (segue, sender);
     (segue.DestinationViewController as ChatViewController).abContact = this.contactToPass;
   }
}
```

donde *ContactoAPasar* es una variable de transición. De esta manera, y en conjunto con lo hecho en el método *RowSelected* de *ContactosTableSource*, logramos traspasar la información sobre el contacto que presionamos.

5.4 Iniciando una Conversación

En esta sección se usarán mucho los tipos de request específicos para la API que estoy usando, por lo que se recomienda leer el Anexo antes de seguir con el tutorial.

Veamos qué tenemos hasta el momento. Abriendo la aplicación, nos aparece una lista de los contactos que tenemos guardados en el teléfono, ordenados por apellidos y separados por distintas secciones según la inicial del apellido. Luego, podemos presionar cualquiera de ellos, y nos aparece una ventana como ésta:

¿Qué falta todavía? Bueno, lo más importante y lo central de este tutorial: iniciar una conversación y comenzar a chatear fácilmente. Partamos por lo primero. ¿Cómo iniciamos una conversación con el contacto? Aquí es donde entra en juego la API que debes tener.

En mi caso, la verdad no fue nada fácil. Por cada contacto que aprieto, debo verificar primero si existe como usuario en la API o no. Si no existe, lo debo crear. Luego, reviso si ya tengo una conversación creada con él. Si no la tengo, la creo y guardo el ID de la conversación. Si la tengo, la debo buscar entre mis conversaciones para obtener el ID. En fin, todo un trámite. Sin embargo, esa es la estructura de la API, y es de esa forma para luego sostener conversaciones grupales, que están fuera del alcance de este tutorial, desgraciadamente. Básicamente, es necesario crear un método que haga todas estas cosas antes de que el chat se abra, para tener el ID de la conversación y así poder enviar fácilmente mensajes. En mi caso, lo hice en un Thread aparte, para que no detuviese la interfaz gráfica.

```
public void DoRequest()
{
    if (!IsUser ())
    {
        AddUser ();
        CreateConversation ();
    }
    else
    {
        if (!WeHaveConversation ())
        {
            CreateConversation ();
        }
    }
}
```

No voy a entrar en detalles con los métodos, ya que se pueden ver en la aplicación final. Sin embargo, sí quiero dejar claro los dos tipos de request que hice. Ambos requests GET y POST tienen una estructura bastante similar. Lo único que los diferencia es que en el primero solo se pide algo al API, y en el segundo se escribe algo en este, y además se recibe otra cosa.

Primero con el método GET:

```
public bool IsUser()
{
    //request URL
    string URL = "http://guasapuc.herokuapp.com/users.json";
    try
    {
        var request = WebRequest.Create(URL) as HttpWebRequest;
        request.Method = "GET"; //request type
        //Headers
        request.ContentType = "application/json";
        request.Headers.Add("Authorization", "Token token="+token); //safetymeasure on the
API
        //The answer to the request.
        HttpWebResponse Httpresponse = (HttpWebResponse)request.GetResponse();
        StreamReader sr = new StreamReader (Httpresponse.GetResponseStream ());
        string answerJson = sr.ReadToEnd ();
        sr.Close (); //We close the reader.
        //We deserialize the users already on the API
        var users = JsonConvert.DeserializeObject<List<APIUser>>(answerJson);

        //Check if the contact is or isn't there.
        foreach(APIUser u in users){
            if (u.phoneNumber == this.contact.phoneNumber)
                return true;
        }
        return false;
    }
    catch(Exception e){
        UIAlertView error = new UIAlertView ("Error", e.Message, null, "Ok", null);
        error.Show ();
        return false;
    }
}
```

Y ahora un ejemplo del método POST:

```
public void AddUser()
{
    //Request URL.
    string URL = "http://guasapuc.herokuapp.com/api/v2/users";
    try
    {
        var request = WebRequest.Create(URL) as HttpWebRequest;
        request.Method = "POST";

        //same headers as on GET method.
        request.ContentType = "application/json";
        request.Headers.Add("Authorization", "Token token="+token);

        //Instantiate the stream writer, in charge of modifying the data on the server.
        var sw = new StreamWriter(request.GetRequestStream());

        //We create the contact and serialize it to send it to the server
        var user = new APIUser(){phoneNumber=contact.phoneNumber, name=contact.name,
password="123456"};
        string json = JsonConvert.SerializeObject(user);
        //the new user is written on the API
        sw.Write(json);
        sw.Flush();
        sw.Close(); //The writer is shut.

        //Necessary to read the full response.
        HttpWebResponse Httpresponse = (HttpWebResponse)request.GetResponse();
        StreamReader sr = new StreamReader (Httpresponse.GetResponseStream ());
        sr.ReadToEnd ();
        sr.Close ();
    }
    catch(Exception e)
    {
        //Captures errors.
        Console.WriteLine("Error: "+e.Message);
    }
}
```

Esa es la forma de realizar un típico GET o POST *request*. Si se fijan, ocupé el método *SerializeObject* y *DeserializeObject* del objeto *JsonConvert*, que viene del componente que instalamos de *Json.Net.* Para poder ocupar estos métodos no olviden incluir la librería, haciendo:

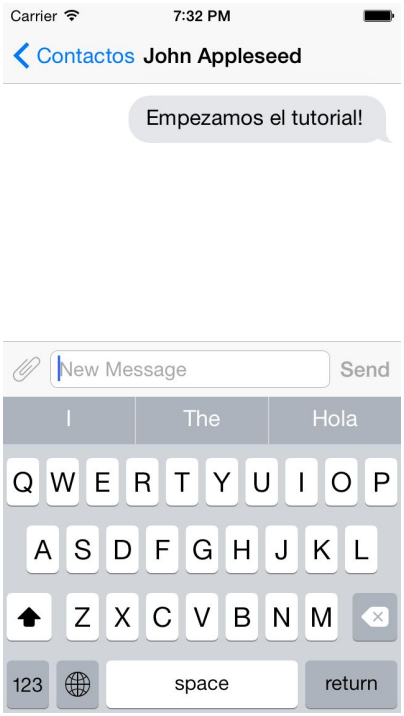
```
using Newtonsoft.Json;
```

Así, podemos ver que el método POST escribe y lee información del servidor, y el método GET solo lee. Además, ocupamos esta información para distintas cosas después.

5.5 Enviar y Recibir Mensajes.

Ahora, lo interesante finalmente llega: ¿cómo lo hacemos para enviar y recibir mensajes? La verdad, con todo lo que hemos hecho, esto es bastante simple, y yo creo que ya lo podrían hacer solos. Sin embargo, lo muestro por si las moscas.

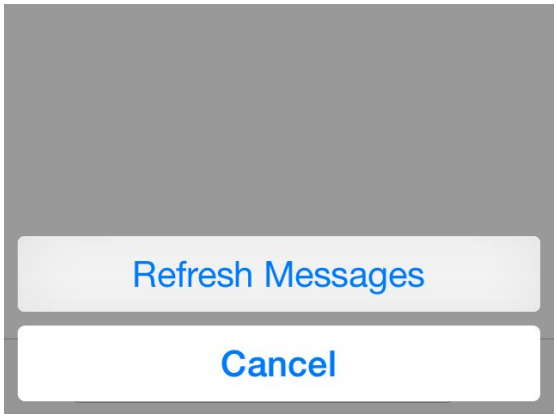
Primero, para enviarlos. Como vemos en el Anexo, para enviar mensajes ocupando la API actual, necesito tener el ID de la conversación. Éste, lo obtuve al hacer el segue y revisando si tenía alguna conversación con el contacto, por lo que ya lo tenemos. Por lo tanto, es cosa de hacer un simple POST request con el ID, mi número y el contenido del mensaje. JSQMessagesViewController tiene un método llamado *FinishedSendingMessage* que se debe llamar al terminar de enviar el mensaje. Éste limpiará la *InputToolbar*, y recargará todos los métodos que tienen que ver con el relleno de la vista. Así, se agrega el mensaje que uno manda a la vista, de esta forma:



Segundo, para recibirlos. Como contaba antes, haremos una forma bastante burda, pero simple, de recibir los mensajes que estén para mí en el servidor. Para esto, haremos uso del *AccesoryButton* que incluye nuestro ChatViewController, para mantenerlo más elegante. Además, nos valdremos de un ViewController que viene por defecto, que es el *AlertViewController*. Éste nos permite mostrar de forma rápida una alerta o una *ActionSheet* de manera de mostrar opciones extras. En este caso ocuparemos esta última modalidad, y se ocupa así:

```
public override void PressedAccessoryButton (UIButton sender)
{
    base.PressedAccessoryButton (sender);

    //A button to refresh.
    var refreshMessages = new Action<UIAlertAction> (RefreshMessages);
    var alerta = UIAlertController.Create(null, null,UIAlertControllerStyle.ActionSheet);
    alerta.AddAction(UIAlertAction.Create("Refresh Messages", UIAlertActionStyle.Default,
```

```
refreshMessages));
    //Cancel button.
    alerta.AddAction(UIAlertAction.Create("Cancel", UIAlertActionStyle.Cancel, null));
    //Shows the alert.
    PresentViewController (alerta, true, null);
}
```



Ahora, el BackEnd de esto se encuentra en el anexo, donde sale como obtener los mensajes de una conversación con un GET request. Sin embargo, esto no es tan simple como parece. La solución no es incluir todos los mensajes que están en el servidor, pues se repetirían con los que ya tengo en el Chat. Es por eso que hacemos un request de los mensajes, y luego agregamos a nuestra lista *Mensajes* todos los que cumplan que no están previamente en la lista. Finalmente, al igual que al enviar los mensajes, llamamos al método *FinishedReceivingMessages* para recargar las fuentes de información y mostrar los nuevos mensajes en la vista.

```
void RefreshMessages(UIAlertAction obj)
{
    var messagesConversation = GetMessagesConversation ();
    foreach (MessageServer m in messagesConversation) {
        if (!messages.Contains (m))
            messages.Add (m);
    }
    FinishReceivingMessage (true);
    ScrollToBottom (true);
}
```

¡Y estamos listos! Así de simple. Tenemos un servicio de mensajería instantánea con todos los contactos de nuestro teléfono. Ahora, eso sí, para que podamos recibir respuesta, tendremos que hacer que descarguen nuestra app cuando en el futuro esté en la App Store.

Cualquier duda, consulta o sugerencia, no tengan problemas en escribirme a lhcubillos@uc.cl, y voy a hacer lo posible por ayudarlos.

6. Anexo: Funcionamiento de la API web

Como conté durante el desarrollo de la app, es necesaria una API web para completar este interesante desafío. En mi caso específico, doy los créditos a Andrés Matte (aamatte@uc.cl) por la creación de la que ocupé durante este tutorial. Ahora, les voy a dejar las indicaciones de cómo usar esta API en específico, para que entiendan cómo hice los distintos request, y qué significa cada uno.

De partida, para hacer cualquier tipo de request, se necesita un token, entregado por la API al momento de crear un usuario. En mi caso en específico, el token se encuentra en el código de la aplicación. Así, a todo request había que agregar dos *headers*: Content-Type: application/json y Authorization:Token token=*token*.

Para probar los distintos request, recomiendo usar PostMan, un add-on de Chrome.

6.1 Obtener usuarios API

Para obtener los usuarios de la API se hace un GET request con el siguiente URL:

```
URL = “http://guasapuc.herokuapp.com/users.json”
```

lo que retorna un arreglo de la forma:

```
{
        "id": 1,
        "phone_number": "56961567267",
        "password": "123456",
        "api_key": "1MQK9QG5as8nu6yw9ENKJAtt",
        "url": "http://guasapuc.herokuapp.com/users/1.json"
}
```

6.2 Crear usuario

Para crear un nuevo usuario en la API, se hace un POST request con el siguiente URL y parámetros:

```
URL = "http://guasapuc.herokuapp.com/api/v2/users"
```

JSON:

```
{ "name":"Luis", "phone_number":"56961527267", "password":"pass"}
```

Lo que retorna un arreglo con toda la información del usuario, como el link y el token:

```
{
 "id": 66,
 "name": "name",
 "phone_number": "numero",
 "api_key": "a3Ce8PR3fCMMRLBffAWpXgtt"
}
```

6.3 Crear Conversación con usuario

Para crear una conversación con el usuario que nos interesa, hacemos un POST request con el siguiente URL y parámetros:

```
URL = "http://guasapuc.herokuapp.com/api/v2/conversations/create_two_conversation"
```

JSON:

```
{ "first":"56912345678", "second":"56987654321", "title":"Titulo (puede ser el nombre con que tengan al contacto)"}
```

6.4 Obtener conversaciones de usuario

Para obtener todas las conversaciones en las que está participando un usuario, hacemos un GET request con el siguiente URL:

```
URL =
"http://guasapuc.herokuapp.com/api/v2/users/get_conversations?phone_number=numero_telefono"
```

Lo que retorna el id de la conversación y otros:

```
{
        "id": 112,
        "title": "Titi4",
        "group": false,
        "admin": "56983362592"
}
```

6.5 Enviar mensaje

Para enviar un mensaje a través de la API, se hace un POST request con el siguiente URL y parámetros:

```
URL = "http://guasapuc.herokuapp.com/api/v2/conversations/send_message"
```

JSON:

```
{ "conversation_id": "20",  "sender":"56912345678", "content": "Hola a todos"}
```

Retorna:

```
{
        "id": 581,
        "sender": "56983362592",
        "content": "Hola",
        "conversation_id": 74,
        "created_at": "2015-05-26T03:42:20.039Z",
        "file": null,
        "url": "http://guasapuc.herokuapp.com/messages/581.json"
}
```

6.6 Recibir mensajes de una conversación

Y finalmente, para recibir mensajes, se hace un GET request con el siguiente URL:

```
URL = "http://guasapuc.herokuapp.com/api/v2/conversations/get_messages?conversation_id=74"
```

Retorna:

```
[
    ...
    {
        "id": 581,
        "sender": "56983362592",
        "content": "Hola",
        "conversation_id": 74,
        "created_at": "2015-05-26T03:42:20.039Z",
        "file": null,
        "url": "http://guasapuc.herokuapp.com/messages/581.json"
    },
    ...
]
```