



Twilio with C#

Succinctly[®]

by Ed Freitas

Twilio with C# Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2017 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	5
About the Author	7
Acknowledgements	8
Introduction	9
Chapter 1 Working with SMS	11
Introduction	11
Setting up a Twilio account	11
Buying a Twilio number	13
Installing the Twilio API helper library	14
Sending SMS	16
Tracking SMS	21
Setting up 2FA using Authy	23
Creating an Authy app	27
Registering an Authy recipient	28
Verification code by SMS	31
Summary	35
Chapter 2 Automation Using SMS	37
Introduction	37
Using Azure Table Storage as our backend	37
Appointment reminders	43
Automated instant lead alerts	55
Summary	63
Chapter 3 Receive and Make Calls	64
Introduction	64
Twiml	64
Receive incoming calls	65
Simple Answering Machine	67
Make outbound calls	77
Summary	78
Chapter 4 Automation Using Voice	79
Introduction	79
Interactive Phone Menu	79
Simple conferencing service	83
Automated voice survey	89
Summary	107

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas works as consultant. He was recently involved in analyzing 1.6 billion rows of data using Redshift (Amazon Web Services) in order to gather valuable insights on client patterns. Ed holds a master's degree in computer science, and he enjoys soccer, running, travelling, and life hacking. You can reach him at Edfreitas.me.

Acknowledgements

My thanks to all the people who contributed to this book, especially Hillary Bowling, Tres Watkins, and Morgan Weston, the Syncfusion team that helped make this a reality. Thanks also to Manuscript Manager Darren West and Technical Editor James McCaffrey, who thoroughly reviewed the book's organization, code quality, and accuracy. Thank you all.

Introduction

Smartphones are now so important to us that they have become almost extensions of ourselves. If you lose your phone today, you are in trouble. Everything from emails, calendar, messaging, banking, and even our wallets are now linked to our phones.

In today's vibrant, dynamic, and ever-connected society, having access to vast amounts of information at our fingertips can be a blessing, but it can also be a curse. Busy professionals nowadays must deal with hundreds of emails on a daily or weekly basis, not to mention the many messages and notifications from social networks such as Twitter and LinkedIn. Keeping up with all these messages can be overwhelming.

But what if phones could actually help us alleviate some of this information overload by notifying us of important things or allowing us to perform customized actions based on [Short Message Service](#) (SMS) or voice commands? Imagine if we could automate certain processes through messaging or voice. That would be great!

Most SMS and voice-enabled solutions are platform-specific and cannot be customized. However, one platform was designed from the ground up with developers in mind and allows anyone with coding skills to create custom messaging and voice-enabled solutions. This platform is [Twilio](#).

Twilio is a messaging, voice, video, and authentication API for every application. It has helper libraries and SDKs in many different programming languages that help developers create apps that can leverage the power of voice and messaging.

These services are accessed over HTTP(S) through a [RESTful](#) API or through helper libraries. Its services are billed based on usage, and the platform is based on Amazon Web Services for hosting Twilio's telephony infrastructure and providing connectivity between HTTP and the Public Switched Telephone Network (PSTN) through its APIs.

Twilio has recently extended its API support to Facebook [Messenger](#), which coincides with the social networking company's introduction of support for bots on its Messenger platform.

In this e-book, we will focus on how to use the C# helper library in order to build connected voice and messaging apps that leverage the power of the Twilio platform. The examples in this e-book require some knowledge of C#, however they are otherwise easy to follow and understand. We'll be using some aspects of the most recent C# language features available. The code samples have all been written using Visual Studio 2015.

By the end of this e-book, you should feel comfortable enough to use the Twilio C# helper library in order to write voice and messaging apps with Visual Studio 2015. We'll explore some common-use cases for voice and messaging apps that are helpful for business automation processes and could prove to be an invaluable resource for any company.

Important: Some of phone numbers provided in the sample code that follows, e.g., 33484464, are actually invalid (not real) phone numbers, and you should provide valid phone numbers with an international format, e.g., +16500000000, if you want to have real interactions on your device.

All the sample code for this e-book is available as a compressed .rar file at this [URL](#).

Have fun!

Chapter 1 Working with SMS

Introduction

Before social networks took off, SMS was the most common way to exchange short messages.

Even though Wikipedia reports that SMS is still strong and growing, social networking messaging services such as Facebook [Messenger](#), [WhatsApp](#), [Skype](#), and [Viber](#), all available on smartphones, are increasingly being used to exchange short messages.

Despite that, SMS is still used widely in enterprise development for marketing, Customer Relationship Management (CRM) automation, real-time alerts notifications, and two-step verification of a user's identity.

The significance of SMS usage in the business world remains incredibly important given that the technology is considered mature, widely used, proven, and reliable.

One of the most prevalent use cases for the adoption of SMS is two-step verification, also known as Two-Factor Authentication (2FA), which consists of strengthening or replacing a username or password with one or more SMS codes. This requires both the primary password and the user's mobile phone in order to properly authenticate and allow a user to get access to a specific resource.

Other uses of SMS technology include sending and receiving standard notifications and alerts.

With the Twilio platform, SMS can be sent to mobile phones anywhere in the world, and message tracking is built into the platform.

In this chapter, we'll explore how to interact with SMS by using the Twilio API helper library, how to implement 2FA, how to set up a Twilio account, and how to purchase a disposable phone number.

We'll start with simple examples of how to send and track SMS, then use that to build a simple but powerful 2FA system.

Setting up a Twilio account

Before we can send SMS using the Twilio API helper library, we'll need to sign up for the service. Twilio is a pay-as-you-go service, which means you'll need to set up an account and provide your credit card details in order to gain credit that will be used to pay for every SMS you send.

You'll also need to purchase a Twilio number, which is a regular but disposable phone number used to send your messages.

Twilio numbers are available for many countries. They look like any other valid phone number, but they are real phone numbers you can dispose when you no longer need them. Pretty amazing.

In order to set up a Twilio account, point your browser to this [URL](#). Once there, click Sign Up. You'll be presented with a screen similar to Figure 1.

A screenshot of the Twilio sign-up page. The page has a light gray background. At the top, it says "It's fast and free to get started." Below this are several input fields: a first name field with "Ed", a last name field with "Freitas", a company name field with "Company Name (optional)", and an email field with "hello@edfreitas.me". There are two password fields, both containing a series of dots. Below the password fields are three dropdown menus: "Which product do you plan to use first?" with "SMS" selected, "What are you building?" with "Other" selected, and "Choose your language" with "C#" selected. At the bottom left is a red "Get Started" button. To its right is the text "By clicking the button, you agree to our legal policies." Below the button is the text "Already have an account? Login".

Figure 1: Twilio's Sign-Up Page

As you can see, the sign-up page is fairly straightforward. You'll need to provide your first name, last name, an optional company name, your email address, and a strong password (which you must enter twice).

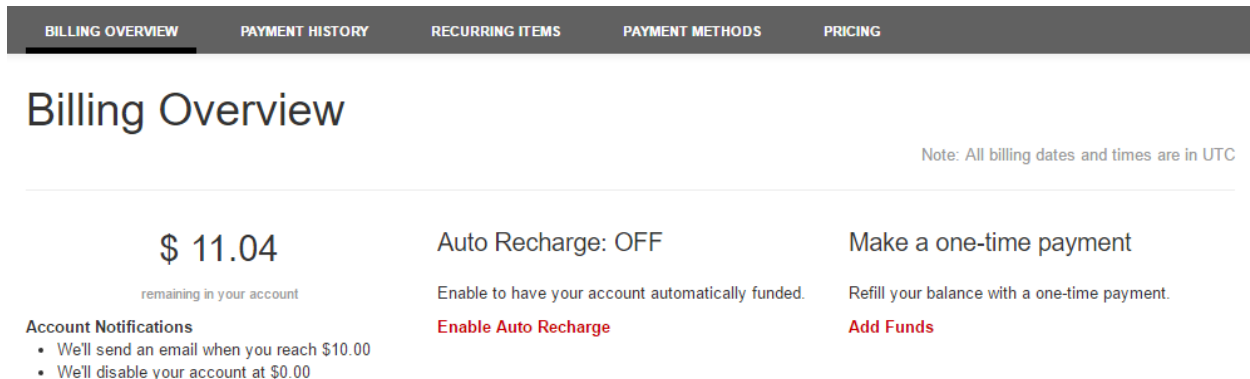
Twilio also asks for additional information in order to get an idea of what kind of apps or project you will initially try with their platform. Once you have filled in the details, all you'll need to do is to click Get Started.

Buying a Twilio number

When your Twilio account is ready, you'll need to purchase a disposal phone number before you can send SMS.

In order to add funds to your account, you'll need to go to this [URL](#). First, click Upgrade to a Full Account. Then, to add funds, click Add Funds, which is linked in red.

You'll have the option to pay using either a credit card or a PayPal account. The minimum starting balance is \$20.

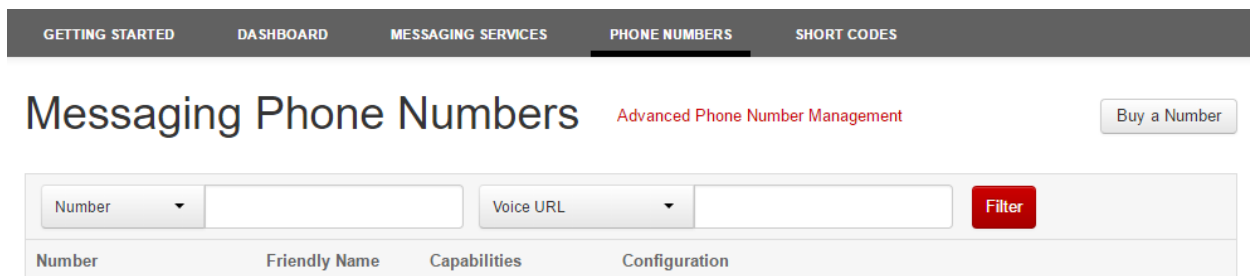


The screenshot shows the Twilio Billing Overview page. At the top is a navigation bar with links: BILLING OVERVIEW, PAYMENT HISTORY, RECURRING ITEMS, PAYMENT METHODS, and PRICING. The main heading is "Billing Overview" with a note: "Note: All billing dates and times are in UTC". Below this, there are three main sections. The first section shows a balance of "\$ 11.04" with the text "remaining in your account" and "Account Notifications" which include: "We'll send an email when you reach \$10.00" and "We'll disable your account at \$0.00". The second section is for "Auto Recharge: OFF", with the description "Enable to have your account automatically funded." and a red link "Enable Auto Recharge". The third section is for "Make a one-time payment", with the description "Refill your balance with a one-time payment." and a red link "Add Funds".

Figure 2: Twilio's Billing Overview Page

With funds in your account, let's see how to get a number. This will be a real phone number that you can dispose at any time. You may choose from which country and city you want your number to belong.

In order to do this, point your browser to this [URL](#), then click Buy a Number.



The screenshot shows the Twilio Phone Numbers Overview page. At the top is a navigation bar with links: GETTING STARTED, DASHBOARD, MESSAGING SERVICES, PHONE NUMBERS, and SHORT CODES. The main heading is "Messaging Phone Numbers" with a red link "Advanced Phone Number Management" and a button "Buy a Number". Below this is a search bar with a "Number" dropdown, a text input, a "Voice URL" dropdown, another text input, and a red "Filter" button. Below the search bar is a table with the following columns: Number, Friendly Name, Capabilities, and Configuration.

Figure 3: Twilio's Phone Numbers Overview Page

When you have clicked Buy a Number, you will be presented with a pop-up screen, as shown in Figure 4.

×

Buy a Number

Country

United States (+1)

Can't find the country you need? [Please let us know.](#)

Location

Search by location e.g. Boston

begins with term

?

Capabilities

☐ Voice
☒ SMS
☐ MMS

Different numbers have different communications capabilities. Select the ones your phone number needs.

Advanced Search

Cancel

Search

Figure 4: Buy a Number Pop-Up Screen

Here you have the option to choose from which country you would like to purchase the number and also a specific location.

You may also indicate if the number will be used solely for SMS or if it will also be used for Voice and MMS.

Once you have purchased your number, you will see a screen with the recently created number.

Messaging Phone Numbers

[Advanced Phone Number Management](#)
[Buy a Number](#)

Number	Friendly Name	Capabilities	Configuration
		Voice SMS MMS	
<div>+1 480-500-7559</div> <div>Phoenix, AZ</div>	(480) 500-7559		<div>✓ Voice</div> <div>✓ Messaging</div>

Figure 5: List of Purchased Twilio Numbers

We are now ready to get the Twilio API helper library installed using Visual Studio.

Installing the Twilio API helper library

In order to start coding, we'll need to create a Visual Studio Console Application. In Visual Studio 2015, go to File, New, then Project. You'll be presented with the screen shown in Figure 6. Choose Console Application and name it TwilioSuccinctly.

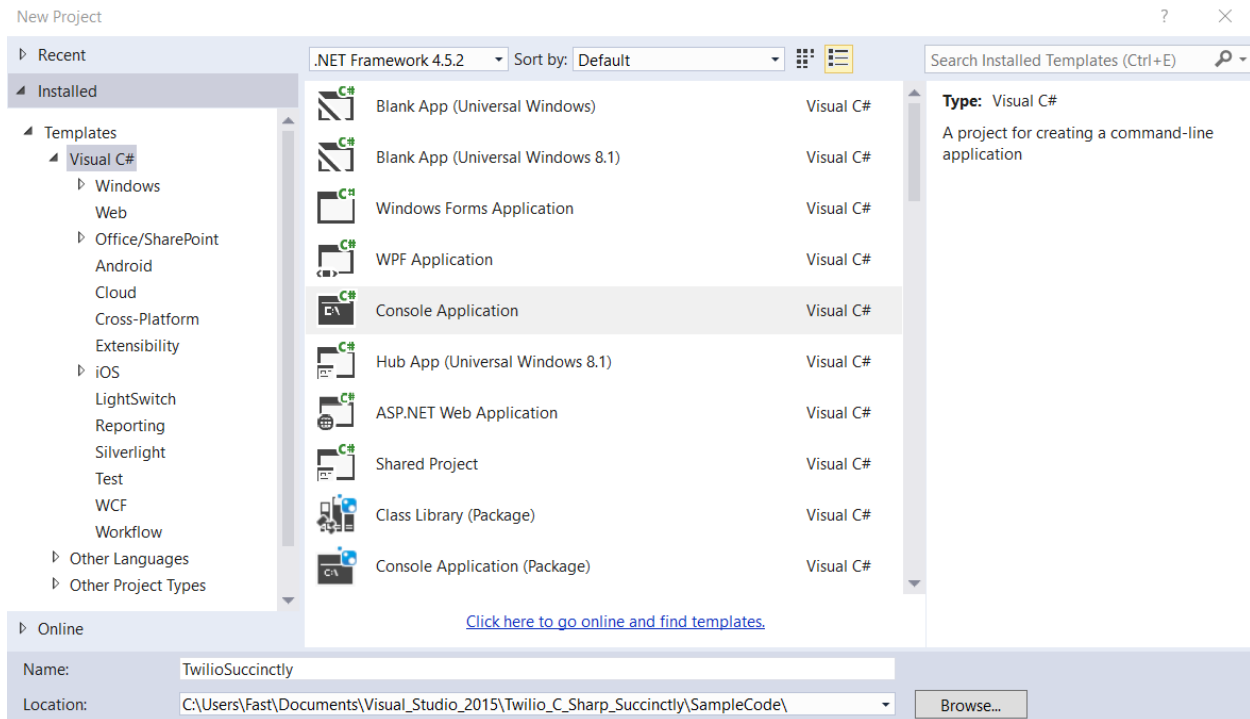


Figure 6: New Visual Studio Console Application

With the Visual Studio project created, let's add two classes—one called **TwilioCore** and another called **TwilioExample** that will serve as a wrapper around **TwilioCore**. The project should look like Figure 7.

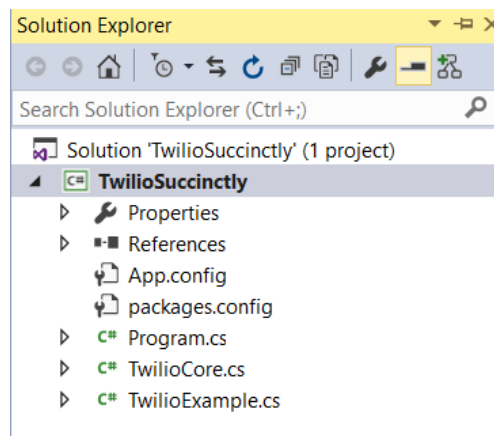


Figure 7: The TwilioSuccinctly Project Structure

Next, let's add the Twilio API helper library, which is available on NuGet. In order to do this, right-click on the **TwilioSuccinctly** project and click Manage NuGet Packages.

Type Twilio on the search box and, from the results returned, choose the Twilio REST API helper library for installation.

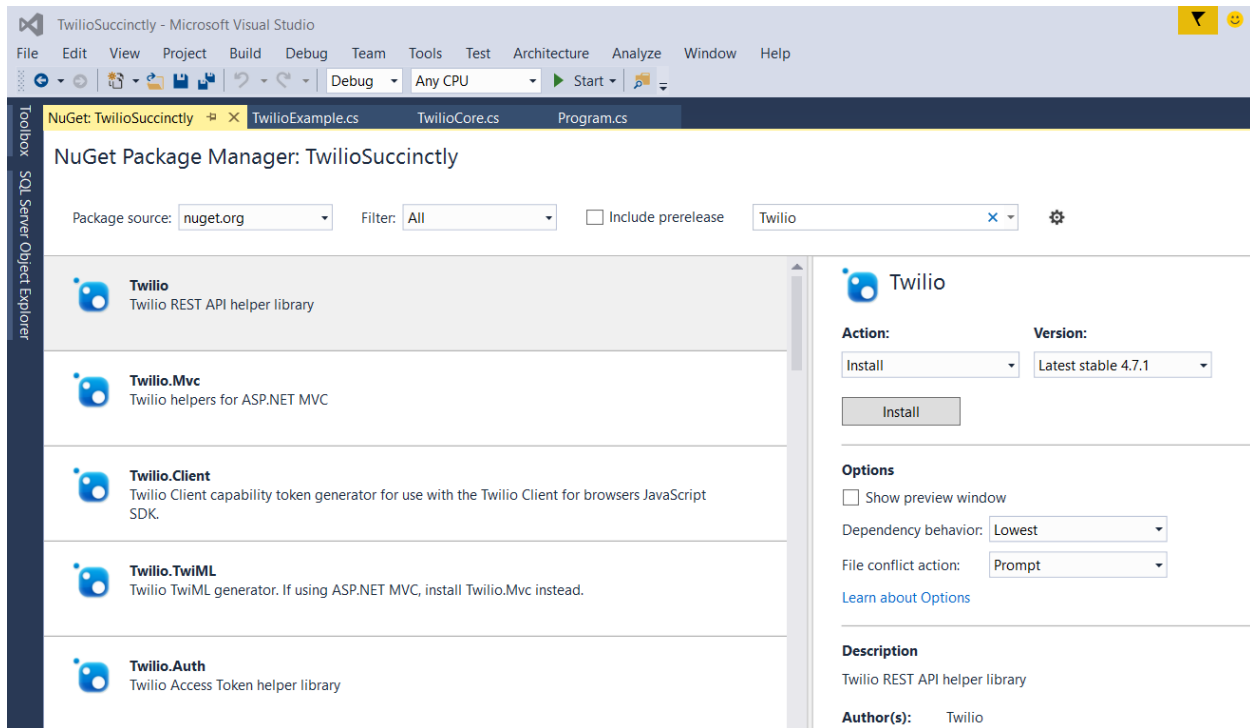


Figure 8: The Twilio REST API Helper Library on NuGet

With the Twilio API helper library installed, we can start writing code.

Sending SMS

The application will have one namespace, called **TwilioSuccinctly**, that will consist of two classes: **TwilioCore** and **TwilioExample**.

The **TwilioCore** class will inherit from **IDisposable** and will contain all the logic that deals directly with the Twilio API helper library.

The **TwilioExample** class is simply a wrapper used to implement uses cases using the **TwilioCore** object.

Let's now implement the **TwilioCore** class. We'll implement the core class functionality and also a method responsible for sending SMS called **SendSms**.

Code Listing 1: The TwilioCore Class

```
using System;
using Twilio;

namespace TwilioSuccinctly
{
```



```

public class TwilioCore : IDisposable
{
    protected bool disposed;

    public string SID
    {
        get;
        set;
    }

    public string aTk
    {
        get;
        set;
    }

    public TwilioCore()
    {
        SID = string.Empty;
        aTk = string.Empty;
    }

    public TwilioCore(string p1, string p2)
    {
        SID = p1;
        aTk = p2;
    }

    ~TwilioCore()
    {
        Dispose(false);
    }

    public virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                SID = string.Empty;
                aTk = string.Empty;
            }

            disposed = true;
        }
    }
}

```

```

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

public Message SendSms(string from, string to, string msg)
{
    if (SID != string.Empty && aTk != string.Empty)
    {
        TwilioRestClient client = new TwilioRestClient(SID, aTk);

        return client.SendMessage(from, to, msg);
    }
    else
        return null;
}
}
}

```

Let's examine this class in detail. In order to use the Twilio API helper library, we need to include the **Twilio** namespace.

TwilioCore inherits from the **IDisposable** interface, which means we can create an instance of **TwilioCore** and later have it automatically disposed if it's called inside a **using** statement. This is very convenient—it helps us write clean code.

In order to authenticate, Twilio requires an Account SID and an Authorization Token (Auth Token).

These have been added to the **TwilioCore** class as parameters to one of the constructors and also as two public properties called **SID** and **aTk** respectively. The Account SID and Auth Token can be found at this [URL](#). In order to view them, you'll need to click Show API Credentials, as shown in Figure 9.

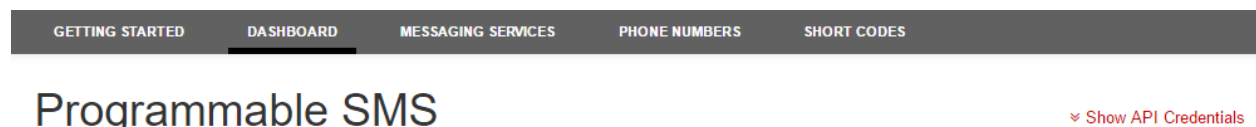


Figure 9: Twilio API Credentials Page

So far, the actual **TwilioCore** class is very simple. There are two constructors, one that has no parameters and doesn't assign any values to the **SID** and **aTk** properties and another one that initializes both properties. The rest consists of the implementation of the disposal mechanism and the class destructor that calls the **Dispose** method.

The **SendSms** method is particularly interesting. This method returns a **Message** object that is part of the Twilio API helper library namespace. In order for the **SendSms** to do anything, the **SID** and **aTk** properties each need a value assigned.

The **SendSms** method creates an instance of the **TwilioRestClient** that is a wrapper of the Twilio API using the [RestSharp](#) library. In order to send an SMS, a call to the **SendMessage** method is executed.

The **SendMessage** method requires three parameters. The first parameter **from** is the number that sends the message. The second parameter **to** is the number that receives it. And the third parameter **msg** is the actual text of the message.

Now that we've seen how easily the **TwilioRestClient** class allows us to send SMS, let's write a proper example. In order to do that, we'll implement the **TwilioExample** class.

Code Listing 2: The TwilioExample Class

```
using System;
using Twilio;

namespace TwilioSuccinctly
{
    public class TwilioExample
    {
        // Your Twilio AccountSID
        private const string cStrSid =
            "<<Your Twilio Account SID goes here>>";

        // Your Twilio AuthToken
        private const string cStrAuthTk =
            "<<Your Twilio Auth Token goes here>>";

        // Change these numbers to the ones you own...
        private const string cStrSender = "<< Your Twilio Number>>";
        private const string cStrReceiver = "+16500000000";

        public static Message SendSimpleSmsExample()
        {
            Message m = null;

            Console.WriteLine("Twilio SendSimpleSmsExample");

            using (TwilioCore tc = new TwilioCore(cStrSid, cStrAuthTk))
            {
                m = tc.SendSms(cStrSender, cStrReceiver,
                    "This is a test SMS");

                // Use C# 6 interpolated string.
                Console.WriteLine($"The status of SMS is: {m?.Status}");
            }
        }
    }
}
```

```

    }
    return m;
}
}
}

```

The static method **SendSimpleSmsExample** creates an instance of the **TwilioCore** class passing the Account SID **cStrSid** along with the Auth Token **cStrAuthTk**, then the method **SendSms** is invoked with the sender **cStrSender** and receiver **cStrReceiver** phone numbers.

We are now missing only the call to **SendSimpleSmsExample** from the main program. This is implemented in Code Listing 3.

Code Listing 3: The Main Program

```

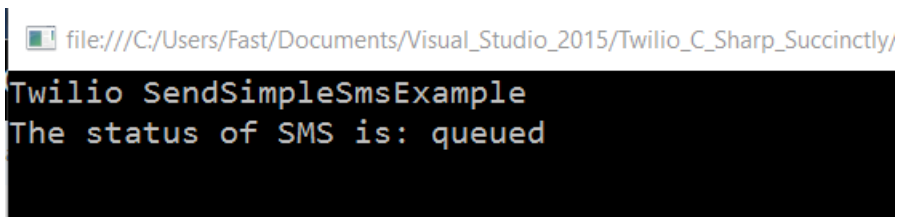
using System;

namespace TwilioSuccinctly
{
    public class Program
    {
        static void Main(string[] args)
        {
            TwilioExample.SendSimpleSmsExample();

            Console.ReadLine();
        }
    }
}

```

Running this code produces the output shown in Figure 10.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/
Twilio SendSimpleSmsExample
The status of SMS is: queued

```

Figure 10: Output of the SendSimpleSmsExample Method

Notice that the output describes that the status of the SMS is queued. This means that Twilio has put the SMS in a queue to be delivered.

If the destination number you will be messaging is an international (non-US) number, you'll need to enable certain permissions in order to allow Twilio to perform that action. These permissions can be checked and configured at this [URL](#).

Tracking SMS

So far, we've explored how to send SMS. But how can we track it?

If we take a look at the implementation of **SendSimpleSmsExample**, we can see that a **Message** object is returned. This object contains a **Status** property that indicates whether the message has been “queued” or “delivered.”

It usually takes a few seconds (between two and five) for the status to change from “queued” to “delivered” even though the SMS might have been received already by the receiver number.

Notice that the **Status** value “queued” or “delivered” is quoted given that it is internally represented as a string.

Twilio is very effective—if your message has been queued, it is very likely to be delivered. Nevertheless, it is still useful to verify that the message has indeed been effectively delivered.

Let's explore how we can check this. Let's add a method to the **TwilioCore** class that checks the status of an SMS.

Code Listing 4: The TwilioCore CheckSmsStatus Method

```
public Message CheckSmsStatus(string sid)
{
    if (SID != string.Empty && aTk != string.Empty)
    {
        TwilioRestClient client = new TwilioRestClient(SID, aTk);

        return client.GetMessage(sid);
    }
    else
        return null;
}
```

The **CheckSmsStatus** method checks for the SMS status by invoking **GetMessage** from an instance of **TwilioRestClient**.

The **GetMessage** method requires the SMS ID **sid** of the message that we want to check. This is the reason why the **SendSms** method of the **TwilioCore** class returns a **Message** object—so that we can obtain the SMS ID from it.

Next, let's implement an example within the **TwilioExample** class by adding a **CheckSmsDelivered** wrapper method around **SendSms**.

Code Listing 5: The TwilioExample CheckSmsDelivered Wrapper Method

```
public static bool CheckSmsDelivered(string mid)
{
```

```

    bool res = false;

    using (TwilioCore tc = new TwilioCore(cStrSid, cStrAuthTk))
    {
        Message m = tc.CheckSmsStatus(mid);

        res = (m?.Status == "delivered") ? true : false;
    }

    return res;
}

```

The functionality is very straightforward. This method creates an instance of the **TwilioCore** class, then invokes **CheckSmsStatus**. The status is determined by inspecting the value of the **Status** property of the **Message** object returned by **CheckSmsStatus**.

If the value of the **Status** property is “delivered,” **CheckSmsDelivered** returns **true**, which confirms that the message has been delivered. Contrary to that, **CheckSmsDelivered** returns **false**.

As we’ve seen, it usually takes somewhere between two to five seconds for Twilio to update the value of the **Status** property from “queued” to “delivered.” Now, let’s implement in our main program a method that takes this into consideration when invoking **CheckSmsDelivered**.

Code Listing 6: The Adjusted Main Program

```

using System;
using System.Threading;
using Twilio;

namespace TwilioSuccinctly
{
    public class Program
    {
        static void Main(string[] args)
        {
            SendSimpleSmsAndCheckIfDeliveredExample();

            Console.ReadLine();
        }

        public static void SendSimpleSmsAndCheckIfDeliveredExample()
        {
            Message m = TwilioExample.SendSimpleSmsExample();

            if (m != null)
            {

```

```

        Thread.Sleep(5500);
        bool res = TwilioExample.CheckSmsDelivered(m.Sid);

        Console.WriteLine($"SMS {m.Sid} delivered: {res}");
    }
}
}
}

```

We've included a new method called **SendSimpleSmsAndCheckIfDeliveredExample** that calls **SendSimpleSmsExample** from the **TwilioExample** wrapper class and then, if the value of the returned **Message** object is not null, waits for 5.5 seconds before calling **Thread.Sleep**, then invokes **CheckSmsDelivered**.

Running this code produces the output shown in Figure 11.

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/SampleCode/TwilioSuccinctly

```

Twilio SendSimpleSmsExample
The status of SMS is: queued
SMS SM4456f8decfd9435aa992dfde9e8d784d delivered: True

```

Figure 11: Output of the *SendSimpleSmsAndCheckIfDelivered* Method

We've seen how easy and straightforward it is to send and track SMS with Twilio. Let's now explore how we can use these features to build a simple 2FA system.

Setting up 2FA using Authy

Authy is a powerful and advanced 2FA solution that strengthens and can even replace the traditional username and password login for websites, SaaS products, and mobile apps.

Authy makes it easy to use 2FA on your online accounts with your smartphone.

The Authy platform generates secure two-step verification codes on your device because using 2FA helps keep your user accounts secure by validating two factors of identity. It works by requiring you to identify yourself using two different things when you login to a site or an app.

Most login systems only validate something your user already knows, like a password. In simple terms, 2FA consists of registering your mobile number once, then subsequently receiving a disposable, one-time code which is used to validate your login credentials when authenticating.

The second factor in the validation is tied to something you own (such as your mobile phone). So, you can think of two-factor validation as something you know (your password) plus something you have (your mobile phone). Again—adding this additional layer of security helps protect your account from hackers and hijackers.

The Authy service is fully integrated into the Twilio platform and can be accessed from your Twilio account by navigating to this [URL](#). When you open that URL, you will see the page shown in Figure 12.

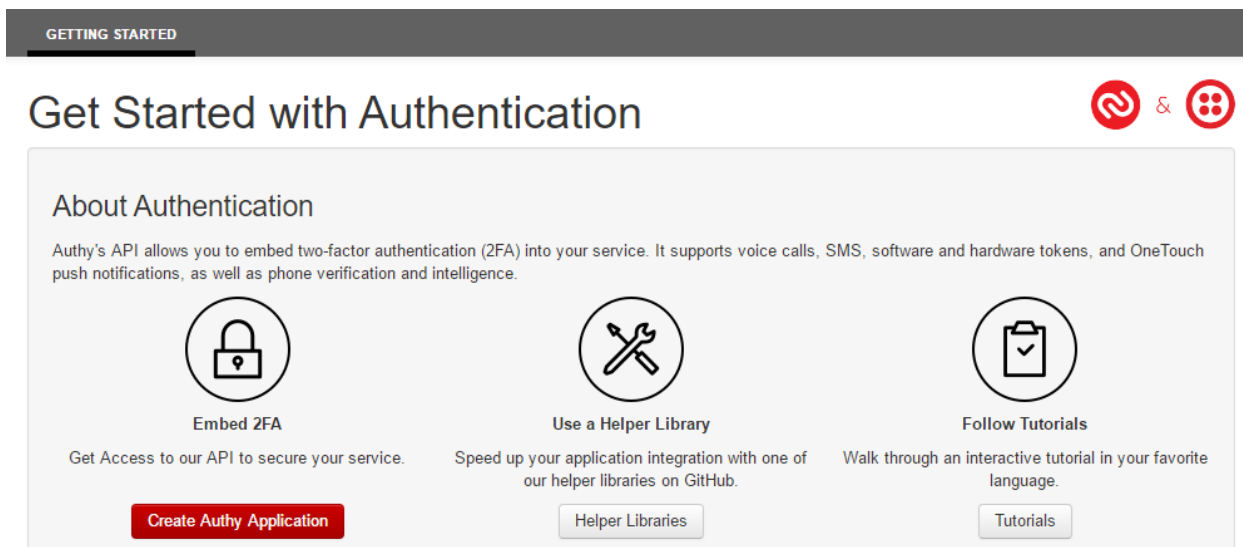


Figure 12: The Twilio Authy Getting Started Page

In order to get started, click Create Authy Application. This will redirect you to the Authy Dashboard in Figure 13.

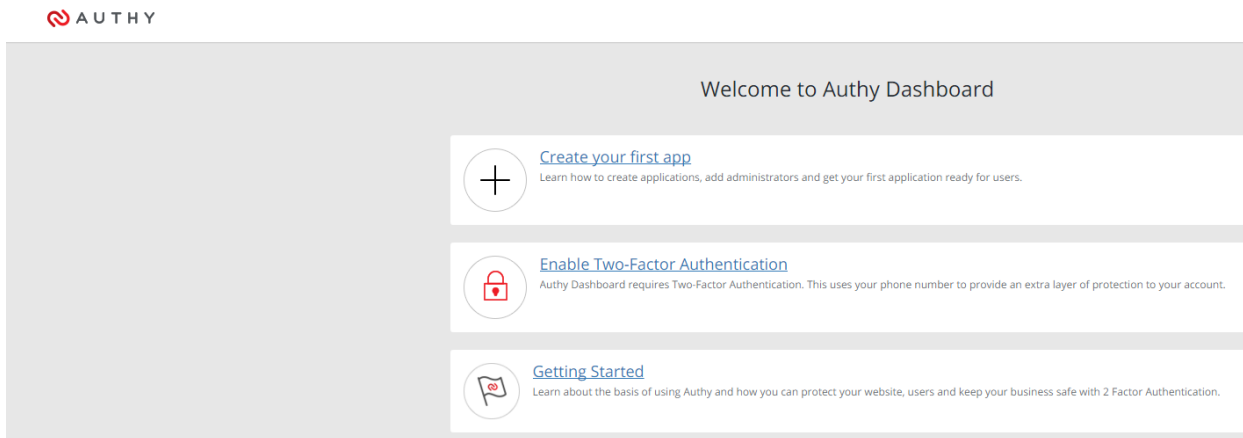


Figure 13: The Authy Dashboard Page

The page is easy to follow—we need only to create our first app in order to get started. This can be achieved by clicking Create your First App.

When you do that, you'll be presented with the following pop-up window.

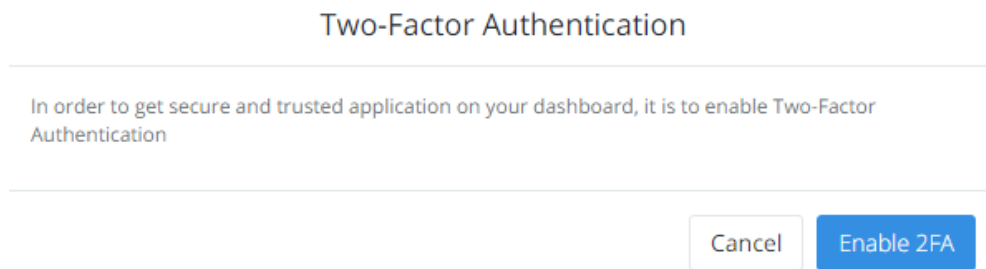


Figure 14: Enable Two-Factor Authentication Pop-Up

In order to enable 2FA, click Enable 2FA. Doing this will display the pop-up in Figure 15.

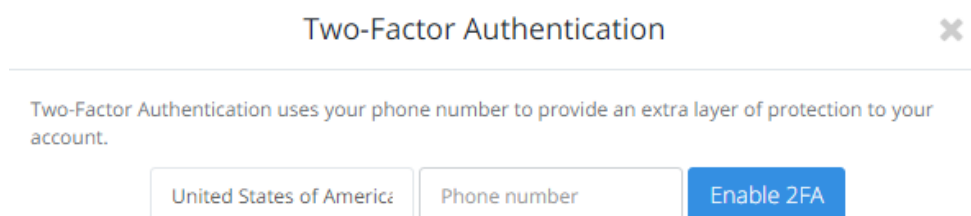


Figure 15: Phone Number for Two-Factor Authentication Pop-Up

Here, you'll have to enter a valid phone number that can receive SMS. You can choose from the list of countries Twilio supports. The country choice is set to United States of America by default, but you can change this.

You can provide the previously created Twilio disposable number that receives SMS, or you can use any number you own that can receive SMS. When you have entered a number, click Enable 2FA, which will verify your identity.

Immediately afterward, Authy will display the pop-up shown in Figure 16.

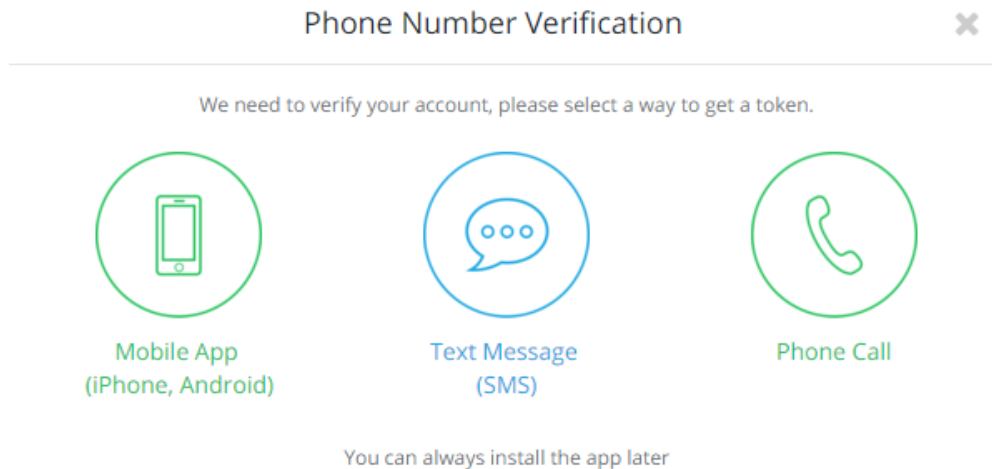


Figure 16: Authy Phone Number Verification

You can choose any preferred method for number verification, but because we are creating a 2FA SMS system, Twilio suggests that you verify your number through a text message.

When you have clicked on the verification method, Authy will display the pop-up shown in Figure 17.

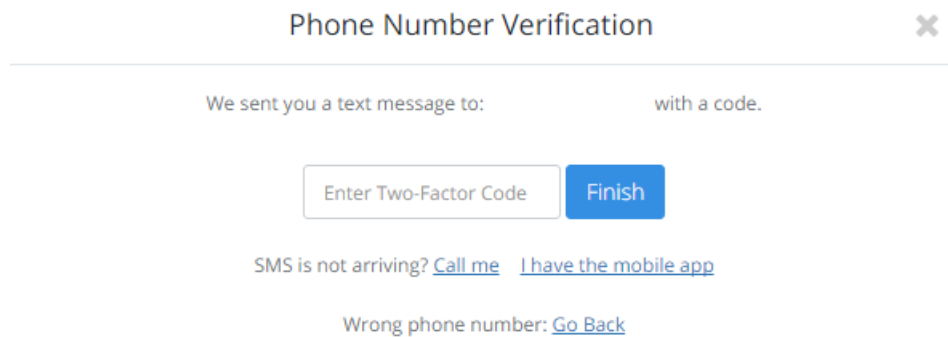


Figure 17: Authy Two-Factor Verification Code Input

At this point, you should have received at least a seven-digit verification code on the phone number you used when registering with Authy. Enter that code and click Finish.

If the SMS did not arrive, Authy provides a link in order to call you or so that you can use their mobile app. In most cases, that will be enough after receiving the SMS.

Once you've received verification, the Authy dashboard should show that 2FA has been enabled.

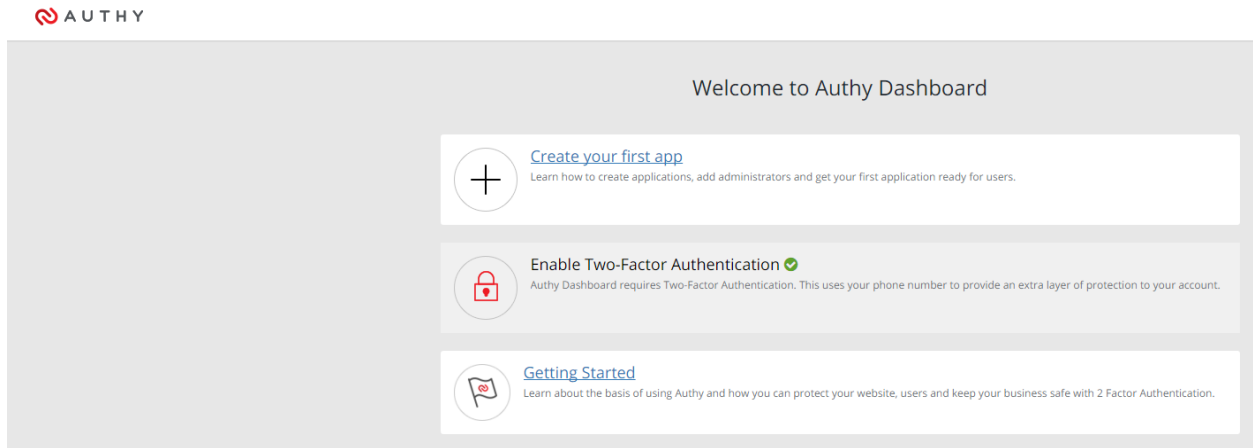


Figure 18: Authy Dashboard with 2FA Enabled

Creating an Authy app

With the Authy setup in place, the next step is to creating the Authy app. You'll first need to give it a name. This can be done by clicking Create Your First App on the dashboard.

You'll then be presented with the following pop-up allowing you to enter the name of your application.

The image shows a 'Create Your First App' pop-up window. It has a title bar with a close button (X). The main text says 'To get started, you need to create an application. Enter the application name and we'll take care of the rest.' Below this is a form with a label 'Application Name' and a text input field containing 'TwilioSuccinctly'. At the bottom right of the form is a blue button labeled 'Create App'.

Figure 19: Create an Authy App Pop-Up

After entering the name of the application, click on Create App. You will be presented with a few more screens, some of which you can skip, or you can simply click Next.

At the end of this process, you will be presented with the API key to use.

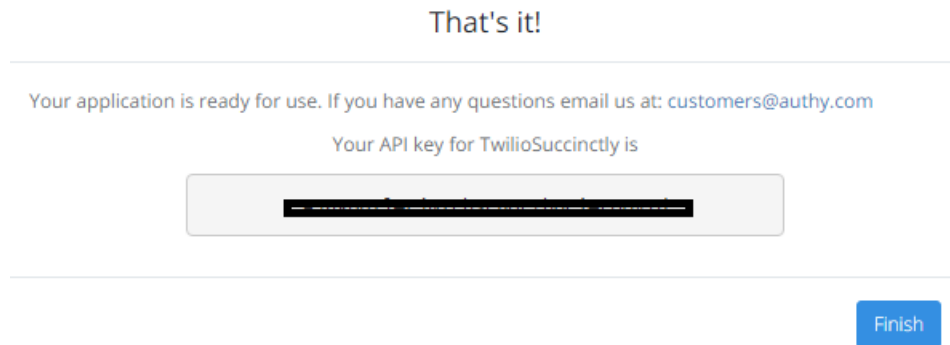


Figure 20: The Authy API Key Pop-Up for Your App

With the API key now available, we can begin setting up Authy with Visual Studio. In order to do this, right-click on your project within Visual Studio and choose Manage NuGet Packages. In the search box, type Authy.Net. You'll be presented with the screen shown in Figure 21.

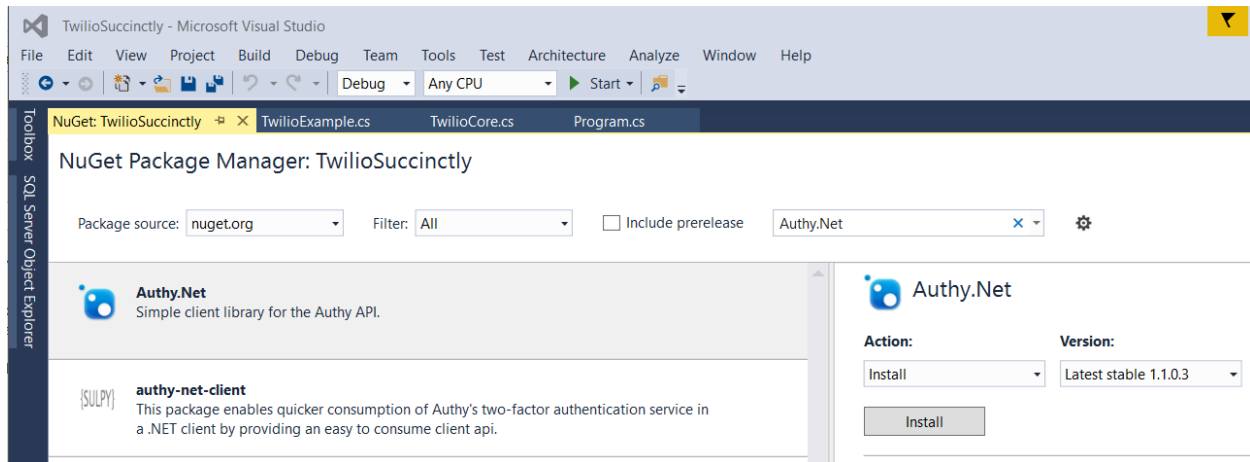


Figure 21: Results of Searching for Authy.Net

Clicking Install will get Authy.Net installed. The Auth.Net package is also available [here](#).

Now it's time to code again. We'll expand the original **TwilioCore** and **TwilioExample** classes and add the Authy 2FA functionality to it.

Registering an Authy recipient

In order to receive a verification token by SMS, it's necessary to register the user's (recipient's) mobile number. You can do this through the Authy dashboard or by code.

Let's modify the **TwilioCore** class in order to add a method to register a user's number.

Code Listing 7: The TwilioCore Class with the RegisterUser Method

```
using System;
using Authy.Net;

namespace TwilioSuccinctly
{
    public class TwilioCore : IDisposable
    {
        protected bool disposed;

        public TwilioCore()
        {
        }

        ~TwilioCore()
        {
            Dispose(false);
        }

        public virtual void Dispose(bool disposing)
        {
            if (!disposed)
            {
                if (disposing)
                {
                }

                disposed = true;
            }
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        public RegisterUserResult RegisterUser(string apikey, string
            uemail, string unumber, int cc, bool tmode)
        {
            if (apikey != string.Empty && uemail != string.Empty &&
                unumber != string.Empty)
            {
                AuthyClient client = new AuthyClient(apikey, tmode);
                return client.RegisterUser(uemail, unumber, cc);
            }
            else
            {
            }
        }
    }
}
```

```

        }
    }
}

return null;

```

The **RegisterUser** method is very simple. It receives the Authy API key **apikey**, the recipient's email **uemail**, the recipient's mobile number **unumber**, and the country code **cc** as parameters. The last parameter **tmode** indicates if test mode will be used (when set to **true**) or if production mode will be used (when set to **false**).

When setting **tmode** to **true**, you'll need to use the Auth API Key for Testing. If you want to use this in production (when **tmode** is set to **false**), you'll need to use the API Key for Production. Both can be found on the Authy dashboard.

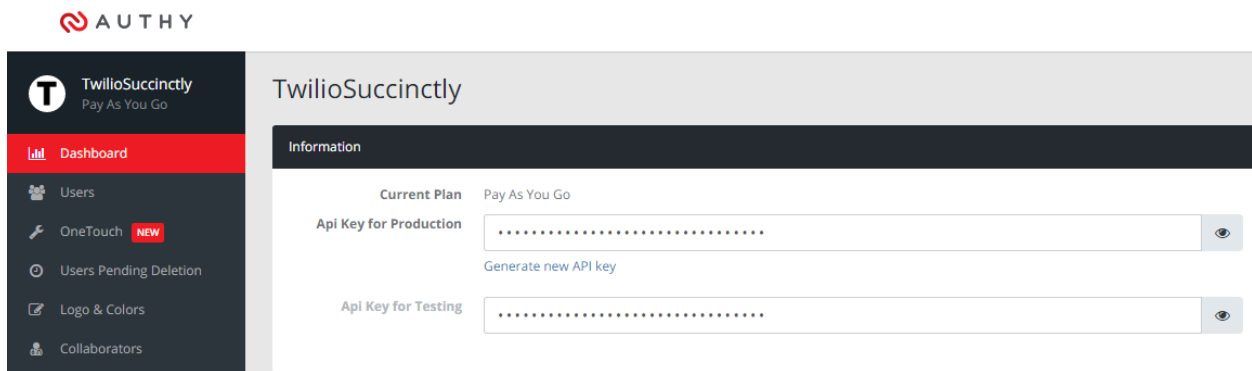


Figure 22: The Authy Production and Testing API Keys

The **RegisterUser** method first verifies that **apikey**, **uemail**, and **unumber** are not empty strings, then it creates an instance of **AuthyClient** by passing **apikey** and **tmode**. In order to register the user, a call to the method **RegisterUser** is invoked that passes **uemail**, **unumber**, and **cc**.

With this in place, let's create a wrapper around it that we can later use within the main program. Let's modify the **TwilioExample** class and add a **RegisterUser** wrapper method.

Code Listing 8: The RegisterUser Wrapper Method within TwilioExample

```

using System;
using Authy.Net;

namespace TwilioSuccinctly
{
    public class TwilioExample
    {
        public static RegisterUserResult RegisterUser(string apikey,
            string uemail, string unumber, int cc, bool tmode)
        {

```

```

        RegisterUserResult res = null;

        using (TwilioCore tc = new TwilioCore())
        {
            res = tc.RegisterUser(apikey, uemail, unumber, cc,
                                tmode);
        }

        return res;
    }
}

```

The implementation of the **RegisterUser** wrapper method is also quite straightforward. Essentially, the same parameters used within the **TwilioCore RegisterUser** method are also used here.

An instance of the **TwilioCore** class is created and a call to the **RegisterUser** method is invoked. The returned result from this call is a **RegisterUserResult** object that will be used later to get the **UserId**. The **UserId** will be required in order to send the verification code.

Verification code by SMS

With the **RegisterUser** method implemented on both the **TwilioCore** and **TwilioExample** classes, the next step is to create a method that we can use to send the verification token as SMS.

We'll also do this in two steps, by creating a **SendSms** method on the **TwilioCore** class and a **SendSms** wrapper method on the **TwilioExample** class.

Let's implement the **SendSms** method of the **TwilioCore** class.

Code Listing 9: The SendSms Token Verification Method of the TwilioCore Class

```

public SendSmsResult SendSms(string apikey, string userid, bool tmode)
{
    if (apikey != string.Empty && userid != string.Empty)
    {
        AuthyClient client = new AuthyClient(apikey, tmode);
        return client.SendSms(userid);
    }
    else
        return null;
}

```

This method takes the **apikey**, **userid**, and **tmode** parameters. If **apikey** and **userid** are not empty strings, an instance of **AuthyClient** is created and, immediately afterward, **SendSms** is invoked by passing **userid**. The result is then returned as a **SendSmsResult** object to the calling method.

Next, let's create the wrapper method within the **TwilioExample** class.

Code Listing 10: The SendSms Wrapper Method of the TwilioExample Class

```
public static SendSmsResult SendSms(string apikey, string userid, bool
    tmode)
{
    SendSmsResult res = null;

    using (TwilioCore tc = new TwilioCore())
    {
        res = tc.SendSms(apikey, userid, tmode);
    }

    return res;
}
```

The wrapper method implementation is quite simple, too. An instance of the **TwilioCore** class is created, and the **TwilioCore SendSms** method is invoked with the **apikey**, **userid**, and **tmode** parameters. An **SendSmsResult** object is returned to the calling method, which will then be used to check if the verification code was sent successfully.

With this in place, in order to test what we have done let's now create a method that registers a recipient within our main program and sends the SMS verification code.

Code Listing 11: The Main Program

```
using Authy.Net;
using System;

namespace TwilioSuccinctly
{
    public class Program
    {
        private const string cStrAuthyKey =
            "<<Your Authy Production API Key>>";

        public static void SendSmsAndVerifyToken()
        {
            RegisterUserResult user =
                TwilioExample.RegisterUser(cStrAuthyKey,
                    "hello@edfreitas.me", "+16500000000", 1, false);
        }
    }
}
```



```

        if (user != null)
        {
            SendSmsResult smsRes =
                TwilioExample.SendSms(cStrAuthyKey, user.UserId, false);

            if (smsRes.Success)
            {
                Console.WriteLine("Please introduce your verification
                    token: ");
                string tk = Console.ReadLine();

                if (tk != string.Empty)
                {
                    AuthyResult res =
                        TwilioExample.VerifyToken(cStrAuthyKey,
                            user.UserId, tk, false);

                    Console.WriteLine($"Status {res.Status} message:
                        {res.Message}");
                }
            }
        }

        static void Main(string[] args)
        {
            SendSmsAndVerifyToken();
            Console.ReadLine();
        }
    }
}

```

The **main** method calls the **SendSmsAndVerifyToken** method that invokes the **RegisterUser** method from the **TwilioExample** class. If the returned **RegisterUserResult** object is not **null**, the **SendSms** method will be invoked.

The **SendSms** method of the **TwilioExample** class returns a **SendSmsResult** object that has a property called **Success**, which, when set to **true**, indicates that the verification token has been sent by SMS.

Notice that the **tmode** parameters on the **RegisterUser** and **SendSms** methods are set to **false**, which indicates that Authy is expecting a production API key.

When we run the program later, the recipient user will be registered and visible on the Authy dashboard.

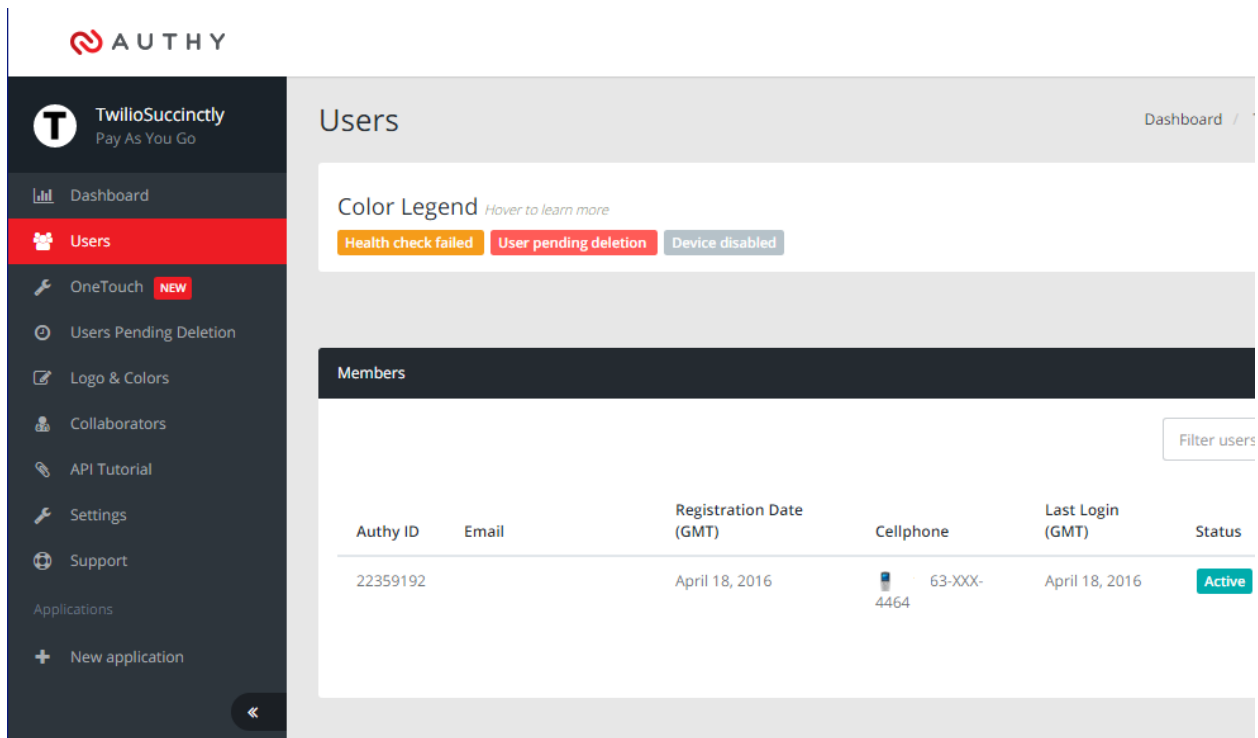


Figure 23: The Register User on the Authy Dashboard

Notice that in our main **program** code, after **SendSms** is called, we request the user to enter the verification code received so that we can check it.

This verification can be done with the **VerifyToken** method. Let's have a look at this method's implementation within the **TwilioExample** class.

Code Listing 12: The TwilioExample VerifyToken Wrapper Method

```
public static VerifyTokenResult VerifyToken(string apikey, string userid,
string token, bool tmode)
{
    VerifyTokenResult res = null;

    using (TwilioCore tc = new TwilioCore())
    {
        res = tc.VerifyToken(apikey, userid, token, tmode);
    }

    return res;
}
```

This method creates an instance of the **TwilioCore** class, then invokes the **VerifyToken** method from that same instance. The result returned is a **VerifyTokenResult** object.

Now let's examine the **VerifyToken** method implementation within the **TwilioCore** class.

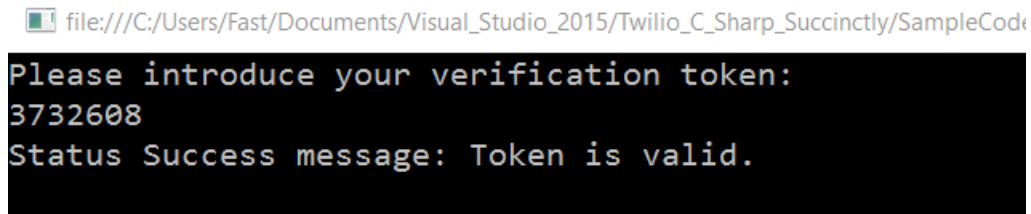
Code Listing 13: The TwilioCore VerifyToken Method

```
public VerifyTokenResult VerifyToken(string apikey, string userid, string
token, bool tmode)
{
    if (apikey != string.Empty && userid != string.Empty && token !=
string.Empty)
    {
        AuthyClient client = new AuthyClient(apikey, tmode);
        return client.VerifyToken(userid, token);
    }
    else
        return null;
}
```

This method checks whether **apikey**, **userid**, **token**, and **tmode** are not empty strings. Notice that **token** is the value that the user received by SMS. We will verify whether or not it is correct—this is the purpose of using Authy with 2FA.

The verification of the token itself is done by creating an instance of the **AuthyClient** class and invoking the **VerifyToken** method from it.

When we run the main **program**, the result shown in Figure 24 is produced. Please notice that when you run your own example, you will receive a different verification code than the one displayed on the output screenshot in Figure 24.



```
file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/SampleCode/
Please introduce your verification token:
3732608
Status Success message: Token is valid.
```

Figure 24: The Output of the Main Authy App

Summary

In this chapter, we've explored how easy it is to get a Twilio and Authy account up and running. We've also seen how we can send SMS using the standard Twilio API and how to do that using Authy in order to do 2FA.

The process and code are very simple and easy to follow because of the way Twilio has wrapped the complexity of these services by creating friendly, easy, and approachable API libraries. Implementing a Twilio solution is a delightful experience.

At this point, you should also have a fairly good understanding of how to use Twilio to send SMS and enable 2FA with Authy by using the C# helper libraries.

In the next chapters, we'll explore more features Twilio offers, including working with calls and other useful voice-enabled applications.

Chapter 2 Automation Using SMS

Introduction

We've seen how Twilio empowers us to implement SMS communication in an intuitive and straightforward way and that it can even be used for enhancing authentication using 2FA.

Using the core principles and examples we've learned so far, in this chapter we'll focus on how we can build on that knowledge and create a couple of small but useful applications that we can use to optimize and automate business processes such as appointment reminders and automated, instant lead alerts.

By the end of this chapter, you should have a broad understanding of how to implement these solutions with SMS and a good appreciation of the overall business logic behind both.

Because some of the samples in this chapter require data storage, we'll be using [Azure Table Storage](#) to keep our data safe and easily accessible on the Microsoft Cloud.

The examples presented in this chapter are intended to be easy to follow and fun to implement while also being useful in a business context. We'll continue to use the **TwilioCore** class we developed in Chapter 1.

Using Azure Table Storage as our backend

As you may know, [Azure](#) is Microsoft's awesome Cloud Services platform. It offers a broad range of services and works across multiple platforms.

Storage is one of its many useful services, and Azure includes a subservice for tables called Table Storage.

The Table Storage service uses a tabular format to store data. Each record represents an entity, and the columns represent the various properties of that entity (fields within a table).

Every entity has a pair of keys (a **PartitionKey** and **RowKey**) to uniquely identify it. Each entity also has a timestamp column that the Table service uses in order to know when the entity was last updated (this happens automatically, and the timestamp value cannot be overwritten—it is controlled internally by the service itself).

Extensive [documentation](#) about how the Storage and Table Storage services work can be found directly on the Azure website, which is an invaluable resource worth checking in order to gain a better understanding of both services.

Nevertheless, we'll quickly explore how we can get up and running with Azure Table Storage.

In order to get started with a Storage instance on Microsoft Azure, you'll need to sign in or sign up for Azure with a Microsoft account. You can do that by visiting <https://azure.microsoft.com>.

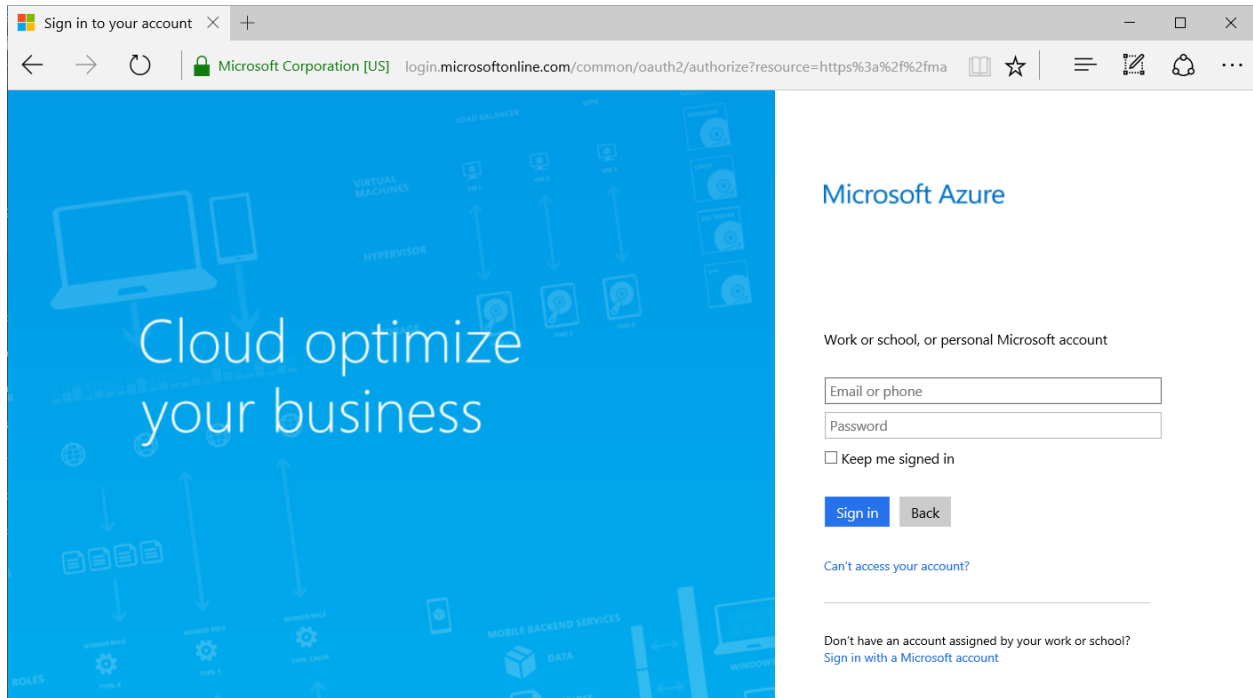


Figure 25: Microsoft Azure Sign-In Screen

Once you've signed up or signed in to the Azure Portal, you can browse through the list of Azure services and select Storage accounts (classic).

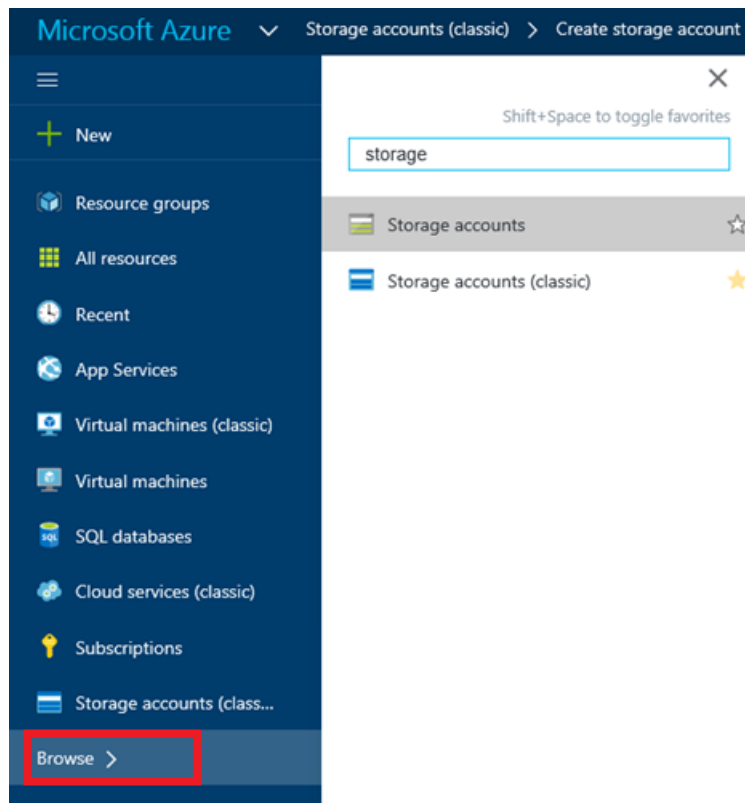


Figure 26: Filtering Through the List of Azure Services

When you select Storage accounts (classic), you'll be presented with the following screen that allows you to add a new Storage account.

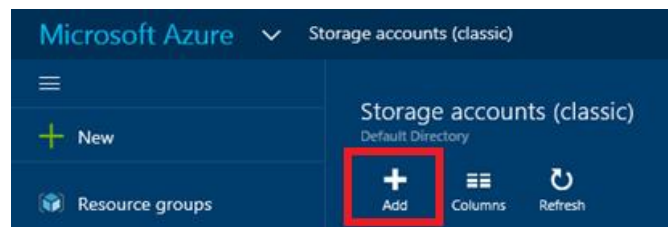


Figure 27: Storage Accounts (Classic) Panel

In order to add a new Storage account (classic), click Add. Doing so will present a screen that allows you to add the account's name and select the Azure location in which the account will be hosted.

Microsoft Azure Storage accounts (classic) > Create storage account

Create storage account

The cost of your storage account depends on the usage and the options you choose below. [Learn more](#)

* Name .core.windows.net

Deployment model
 Resource manager **Classic**

Performance
 Standard Premium

Replication
 Locally-redundant storage (LRS)

Subscription
 BizSpark

* Resource group
 + New

New resource group name

Location
 East US

☐ Pin to dashboard

Create

Figure 27: Create a New Storage Account (Classic) Panel

Once you have chosen a name and selected the location nearest you, click Create. Azure will immediately create the Storage account. Once created, it will look like Figure 28. You'll need your access keys in order to interact with the service from your code or any external tool. The keys can be found by clicking Keys.

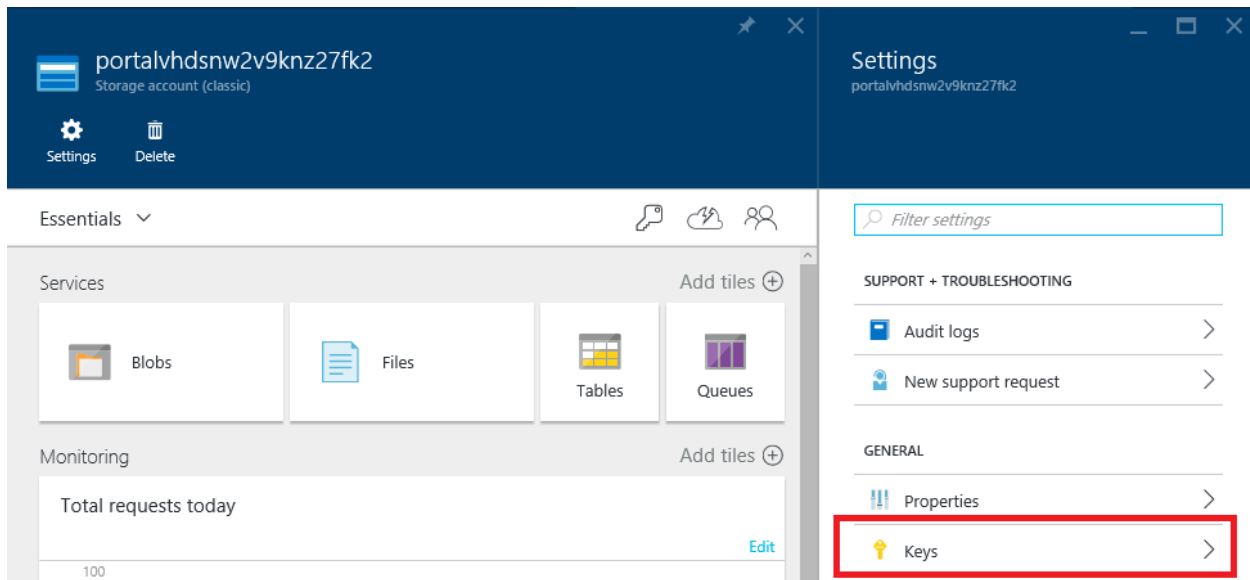


Figure 28: Storage Account (Classic) Panel

With the Azure Storage account now ready, you can use an open source, handy tool called [Azure Storage Explorer](#) that will allow you to easily connect to your Storage account and create, update, delete, and view any storage tables and data.

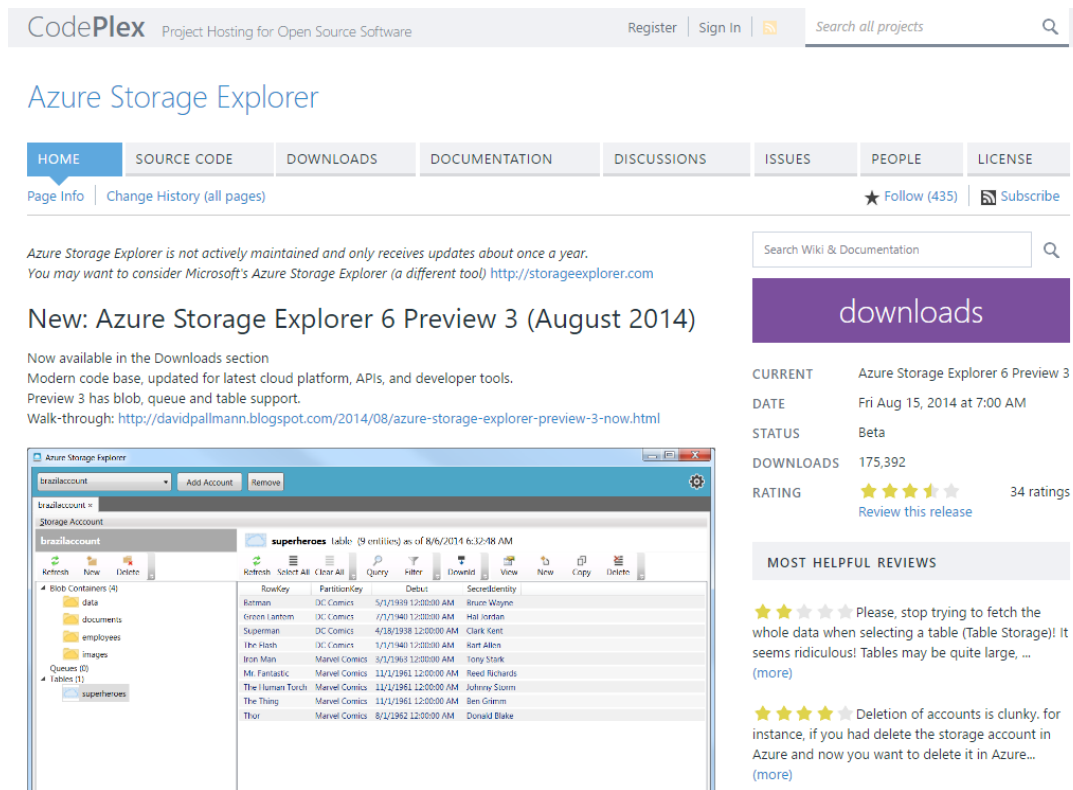


Figure 29: Azure Storage Explorer on CodePlex

After you've downloaded and unzipped the zip file from CodePlex, you'll find an executable that will install the tool.

The installation wizard is easy to follow and self-explanatory, requiring just a few clicks, although please notice that the Azure Storage Explorer application only works on Windows.

Once installed, you'll have the following files on disk.

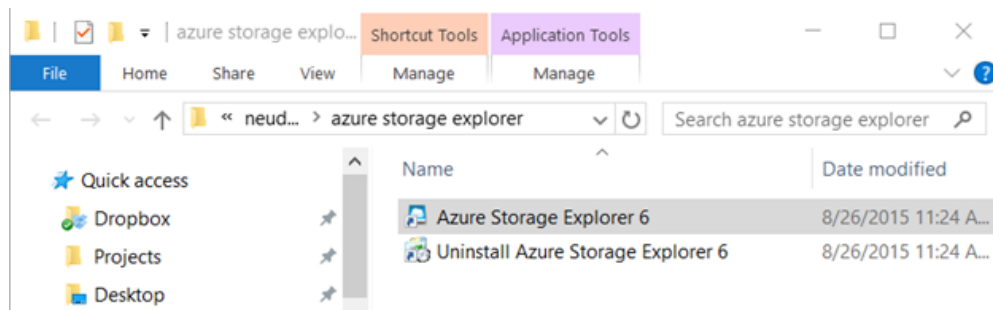


Figure 30: Azure Storage Explorer after Installation

In order to run the tool, double-click on the Azure Storage Explorer shortcut. Once the application is running, you'll need to connect your Azure Storage account to it. This can be done by clicking Add Account.

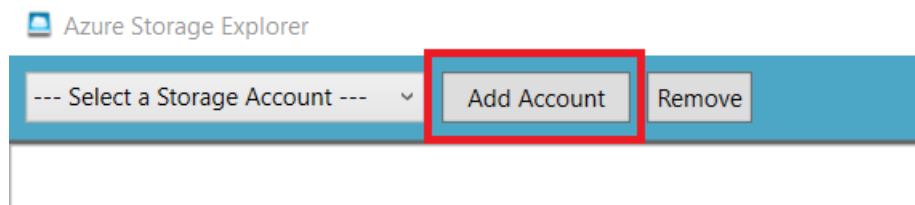


Figure 31: Azure Storage Explorer Add Account

When you click Add Account, you'll be requested to add your Storage account name and key, as shown in Figure 32.

Add Storage Account

☒ Cloud Storage Account ☐ Local Developer Account

Storage account name:
Specify the storage account name

Storage account key:
Specify the storage account key

☒ Microsoft Azure Default ☐ Microsoft Azure China ☐ Other (specify below)

Storage endpoints domain:
core.windows.net

☐ Use HTTPS

Test Access Save Cancel

Enter the credentials for a Microsoft Azure Storage Account

Figure 32: Azure Storage Explorer Add Account Settings

After entering the requested details, you should click Test Access in order to check if the connection works. If that is successful, you can click Save.

With these details stored, the next time you open the Azure Storage Explorer application, you'll be able to access your Storage account from the dropdown next to the Add Account button.

With our Storage account wired up, let's now explore how we can do some SMS automation with Twilio and use Table Storage as our backend.

Appointment reminders

Keeping customers happy is probably the biggest challenge any company faces today. Fortunately, the right attitude and a bit of automation can make customer satisfaction an achievable goal.

Appointment reminders are a simple and effective way of automating the process of reaching out to customers before an upcoming appointment. We can write code that will allow us to accomplish this—we'll use the **TwilioCore** class we've already written.

Let's first think for a bit about how we can architect this solution. We'll need to be able to create appointments with a future **DateTime** that we can store on Azure Table Storage. This future **DateTime** will be the appointment's scheduled date. We'll also need a process in place that watches when an appointment is scheduled, retrieves the information from Table Storage, and sends out an SMS to the registered phone number for that appointment.

We'll initially explore the first action that allows us to interact with Azure Table Storage. In order to do that, let's get the WindowsAzure.Storage NuGet package installed, then create an **Appointment** class.

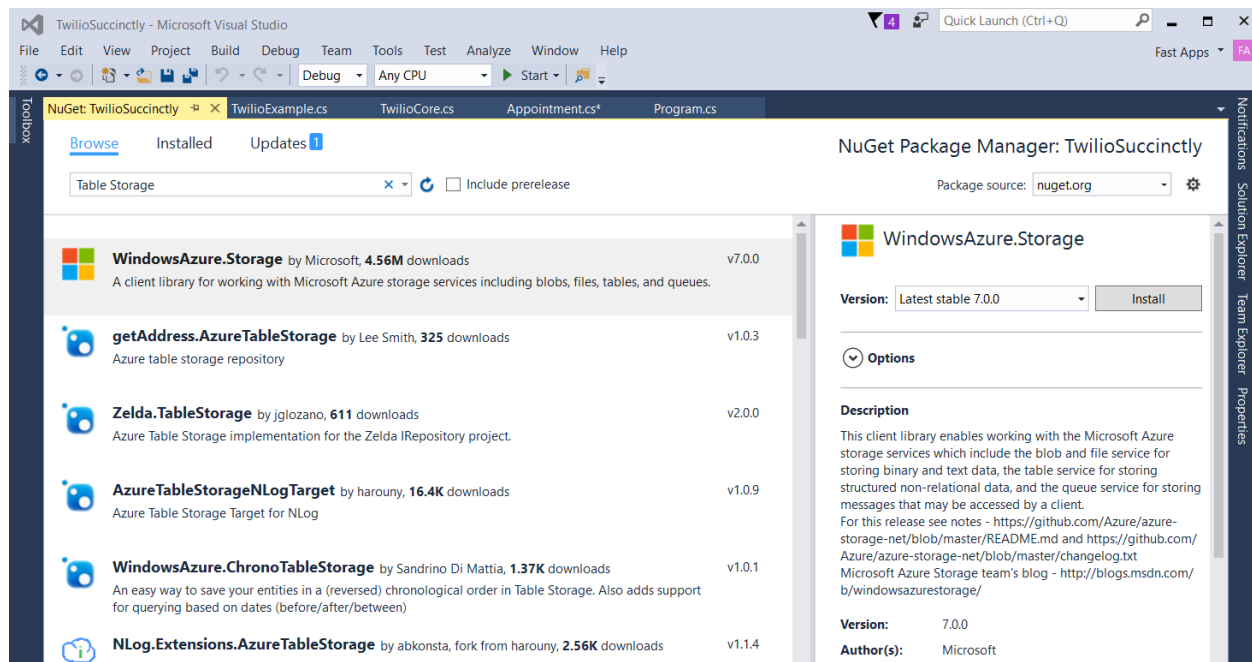


Figure 33: The WindowsAzure.Storage Library on NuGet

In order to get the NuGet package installed, simply click Install. This will add the necessary references to the Visual Studio project.

Before jumping into the code, you'll need to create the storage table using Azure Storage Explorer. Let's name the table AppointmentsSuccinctly.



Figure 34: Creating the AppointmentsSuccinctly Table Using Azure Storage Explorer

With the NuGet package installed, we can create the **AppointmentData** and **Appointment** classes.

Code Listing 14: The Appointment and AppointmentData Classes

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;

namespace TwilioSuccinctly
{
```

```

public class AppointmentData : TableEntity
{
    public AppointmentData(string p, string d)
    {
        PartitionKey = p;
        RowKey = d;
    }

    public AppointmentData()
    {
    }

    public string Id { get; set; }

    public string Name { get; set; }

    public string PhoneNumber { get; set; }

    public string Time { get; set; }

    public bool Sent { get; set; }
}

public class Appointment : IDisposable
{
    private const string cStrConnStr =
        "<< Your Azure Storage Connection String >>";
    private const string cStrAppTable = "AppointmentsSuccinctly";

    protected bool disposed;

    public AppointmentData Data { get; set; }

    public Appointment(string id, string n, string pn, DateTime t)
    {
        string tt = t.Day.ToString().PadLeft(2, '0') +
            t.Month.ToString().PadLeft(2, '0') +
            t.Year.ToString() +
            t.Hour.ToString().PadLeft(2, '0') +
            t.Minute.ToString().PadLeft(2, '0') +
            t.Second.ToString().PadLeft(2, '0');

        Data = new AppointmentData(pn, tt);

        Data.Sent = false;
        Data.Id = id;
        Data.Name = n;
        Data.PhoneNumber = pn;
    }
}

```

```

        Data.Time = tt;
    }

    public Appointment(string pn)
    {
        string tt = DateTime.Now.Day.ToString().PadLeft(2, '0') +
            DateTime.Now.Month.ToString().PadLeft(2, '0') +
            DateTime.Now.Year.ToString() +
            DateTime.Now.Hour.ToString().PadLeft(2, '0') +
            DateTime.Now.Minute.ToString().PadLeft(2, '0') +
            DateTime.Now.Second.ToString().PadLeft(2, '0');

        Data = new AppointmentData(pn, tt);
        Data.PhoneNumber = pn;
        Data.Time = tt;
    }

    ~Appointment()
    {
        Dispose(false);
    }

    public virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                {
                }

                Data = null;
            }
            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    public bool Create()
    {
        bool res = false;

        try
        {
            var storageAccount =

```

```

        CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableOperation retrieveOperation =
            TableOperation.Retrieve<AppointmentData>
                (Data.PhoneNumber, Data.PhoneNumber);

        TableResult retrievedResult =
            table.Execute(retrieveOperation);
        AppointmentData updateEntity =
            (AppointmentData)retrievedResult.Result;

        if (updateEntity == null)
        {
            updateEntity = new AppointmentData(Data.PhoneNumber,
                Data.Time);
            updateEntity.Id = Data.Id;
            updateEntity.Name = Data.Name;
            updateEntity.PhoneNumber = Data.PhoneNumber;
            updateEntity.Time = Data.Time;
        }

        TableOperation ins = TableOperation.Insert(updateEntity);
        table.Execute(ins);

        res = true;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return res;
}

public AppointmentData Get()
{
    AppointmentData res = null;

    try
    {
        int tDay = DateTime.Now.Day;
        int tMonth = DateTime.Now.Month;
        int tYear = DateTime.Now.Year;
        int tHour = DateTime.Now.Hour;
    }

```

```

        int tMin = DateTime.Now.Minute;
        int tSec = DateTime.Now.Second;

        DateTime tdy = new DateTime(tYear, tMonth, tDay, tHour,
            tMin, tSec);

        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableQuery<AppointmentData> query = new
            TableQuery<AppointmentData>().Where(
                TableQuery.GenerateFilterCondition("PartitionKey",
                    QueryComparisons.Equal, Data.PhoneNumber));

        foreach (AppointmentData en in table.ExecuteQuery(query))
        {
            int day = Convert.ToInt32(en.Time.Substring(0, 2));
            int month = Convert.ToInt32(en.Time.Substring(2, 2));
            int year = Convert.ToInt32(en.Time.Substring(4, 4));
            int hour = Convert.ToInt32(en.Time.Substring(8, 2));
            int min = Convert.ToInt32(en.Time.Substring(10, 2));
            int sec = Convert.ToInt32(en.Time.Substring(12, 2));

            DateTime exp = new DateTime(year, month, day, hour,
                min, sec);

            if (!en.Sent && DateTime.Compare(tdy, exp) >= 0)
            {
                res = en;
                break;
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return res;
}

public bool MarkAsSent(AppointmentData ad)
{
    bool res = false;

```



```

try
{
    var storageAccount =
        CloudStorageAccount.Parse(cStrConnStr);
    CloudTableClient tableClient =
        storageAccount.CreateCloudTableClient();

    CloudTable table =
        tableClient.GetTableReference(cStrAppTable);

    TableOperation retrieveOperation =
        TableOperation.Retrieve<AppointmentData>
            (ad.PhoneNumber, ad.Time);

    TableResult retrievedResult =
        table.Execute(retrieveOperation);
    AppointmentData updateEntity =
        (AppointmentData)retrievedResult.Result;

    if (updateEntity != null)
    {
        updateEntity.Sent = ad.Sent;

        TableOperation insertOrReplaceOperation =
            TableOperation.InsertOrReplace(updateEntity);
        table.Execute(insertOrReplaceOperation);

        res = true;
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}

return res;
}

public string FormatTime(string t)
{
    int day = Convert.ToInt32(t.Substring(0, 2));
    int month = Convert.ToInt32(t.Substring(2, 2));
    int year = Convert.ToInt32(t.Substring(4, 4));
    int hour = Convert.ToInt32(t.Substring(8, 2));
    int min = Convert.ToInt32(t.Substring(10, 2));
    int sec = Convert.ToInt32(t.Substring(12, 2));

```

```

        DateTime exp = new DateTime(year, month, day, hour, min,
            sec);
        return exp.ToString();
    }
}

```

Let's analyze this code. We have defined an **AppointmentData** class that represents the data to be stored on Azure Table Storage. The **AppointmentData** class inherits from **TableEntity** and contains a **PartitionKey** and **RowKey** properties used by Table Storage to index and find data records.

We'll store the user's phone number on the **PartitionKey**, and on the **RowKey** we'll keep the date and time of when the appointment is due as a string, e.g., 29042016203000, where 29 is the day, 04 the month, 2016 the year, 20 the hour of the day (8 pm), 30 the minutes, and 00 the seconds.

The **AppointmentData** class also contains a **string Id** property, a **string Name** property that represents the name of the appointment, a **string PhoneNumber** property that will contain the same data as the **PartitionKey**, a **string Time** property that will contain the same data as the **RowKey**, and a **bool Sent** property that will represent whether or not an SMS has been sent for that appointment.

The **AppointmentData** class will be instantiated by the **Appointment** class in order to store the information on Azure Table Storage. You can think of the **AppointmentData** class as the data layer (Model) and the **Appointment** class as the business logic layer (Controller).

The constructors of the **Appointment** class create an **AppointmentData** instance that will be used to store the appointment data on Azure.

With Table Storage, **DateTime** values should be stored as **string** values. The purpose of the **FormatTime** method is to take a date time **string** such as 29042016203000 and convert it to its equivalent **DateTime** representation (4/29/2016 8:30:00 PM) as a **string** so it can be displayed on the **Console**.

The **Create** method is responsible for creating an appointment reminder on Table Storage. A connection to the Azure instance is established by passing the **cStrConnStr** connection string to the **CloudStorageAccount.Parse** method. Following that, an instance of **CloudTableClient** is created, then used to get a reference to the AppointmentsSuccinctly table.

A call to the **Execute** method of the **CloudTable** instance is invoked with the **AppointmentData** class as **T** and **Data.PhoneNumber** as both the **PartitionKey** and **RowKey**. We do this in order to get a **TableResult** object so that its **Result** property can be cast to an empty **AppointmentData** object. This object will be used to set the corresponding appointment properties and will be inserted on Table Storage by calling the **TableOperation.Insert** method. Finally, the **Execute** method of the **CloudTable** instance commits the **AppointmentData** object to Table Storage.

The **MarkAsSent** methods work in almost exactly the same way as the **Create** method. The main difference is that **MarkAsSent** performs an **InsertOrReplace** rather than an **Insert**, thus updating the **Sent** property of the **AppointmentData** object with the value **true** after the SMS has been sent.

In turn, the **Get** method works almost exactly the same way as the **MarkAsSent** or **Create** methods do, however it creates an instance of a **TableQuery** class, then executes the **Where** method by chaining it with the **GenerateFilterCondition** method. With this in place, a call to **CloudTable.ExecuteQuery** is invoked, returning **IEnumerable<AppointmentData>**.

Because **GenerateFilterCondition** bases its filtering on the **PartitionKey**, all the records with the same phone number will be retrieved. By looping through the **IEnumerable<AppointmentData>** results and by comparing today's date **tdy** with the appointment's due date **exp**, the first expired appointment with a non-sent SMS (**Sent** property set to **false**) will be retrieved and returned to the caller method.

Let's now implement a couple of examples that put all this to work. In Code Listing 15, we'll create an **AppointmentExample** class that will create some appointments and send SMS for due ones.

Code Listing 15: The AppointmentExample Wrapper Class

```
using System;
using System.Timers;

namespace TwilioSuccinctly
{
    public class AppointmentExample
    {
        private const string cStrTwilioSid =
            "<< Your Twilio Account SID goes here >>";
        private const string cStrTwilioAuthTk =
            "<< Your Twilio Auth Token goes here >>";
        private const string cStrTwilioSender =
            "<< Your Twilio Number>>";

        private static bool started = false;

        public static Timer Fetch { get; set; }
        public static string PhoneNumber { get; set; }

        public static void InitFetch(int ms)
        {
            started = false;

            Fetch = new Timer();
            Fetch.Interval = ms;
            Fetch.Elapsed += new ElapsedEventHandler(Fetch_Elapsed);
            Fetch.Start();
        }
    }
}
```

```

    }

    public static void StopFetch()
    {
        started = false;
        Fetch.Stop();
    }

    public static void Fetch_Elapsed(object sender,
    ElapsedEventArgs e)
    {
        if (!started)
        {
            started = true;
            Console.WriteLine(
                "Fetching Oldest Elapsed Appointment...");

            using (Appointment a = new Appointment(PhoneNumber))
            {
                AppointmentData ad = a.Get();

                if (ad != null)
                {
                    using (TwilioCore t = new
                    TwilioCore(cStrTwilioSid, cStrTwilioAuthTk))
                    {
                        t.SendSms(cStrTwilioSender, PhoneNumber,
                            $"Reminder of appointment {ad.Id} -
                            {ad.Name} on: {a.FormatTime(ad.Time)}");

                        Console.WriteLine(
                            $"Reminder of appointment
                            {ad.Id} - {ad.Name} on:
                            {a.FormatTime(ad.Time)}");

                        ad.Sent = true;
                    }

                    a.MarkAsSent(ad);
                }

                started = false;
            }
        }

        public static bool Create(string id, string n, string pn,
        DateTime t)
    }

```

```

    {
        bool res = false;

        using (Appointment a = new Appointment(id, n, pn, t))
        {
            res = a.Create();

            if (res)
                Console.WriteLine(
                    $"Appointment created: {id} - {n} on {t}");
            else
                Console.WriteLine(
                    $"Could not create appointment: {id} - {n} on {t}");
        }

        return res;
    }
}

```

The **AppointmentExample** class does two things. It creates an appointment through its **Create** method, and it also creates a **Fetch Timer** object that executes a **Fetch_Elapsed** event that will retrieve the first expired appointment and send an SMS.

In Code Listing 16, we'll put this into action by invoking it from our **Main program**.

Code Listing 16: Main Program

```

using System;

namespace TwilioSuccinctly
{
    public class Program
    {
        public static void CreateSeveralAppointments()
        {
            AppointmentExample.Create("app01", "BurgerKing",
                "33484464", new DateTime(2016, 4, 29, 20, 30, 00));
            AppointmentExample.Create("app02", "Run", "33484464", new
                DateTime(2016, 4, 29, 21, 20, 00));
            AppointmentExample.Create("app03", "Gym", "33484464", new
                DateTime(2016, 4, 29, 22, 05, 00));
        }

        public static void FetchAppointments()
        {
            Console.WriteLine(

```

```

        "Waiting for Timer (Appointment Fetch)...");

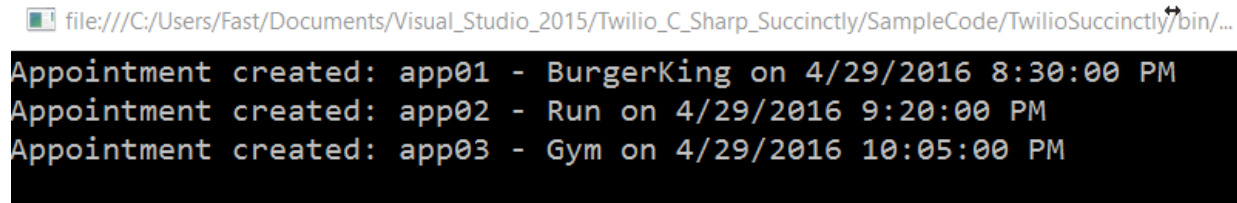
        AppointmentExample.PhoneNumber = "33484464";
        AppointmentExample.InitFetch(1500);
    }

    static void Main(string[] args)
    {
        CreateSeveralAppointments();
        Console.ReadLine();

        FetchAppointments();
        Console.ReadLine();
    }
}

```

When we execute the **Main** program, we get the appointments created in Figure 35.



file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/SampleCode/TwilioSuccinctly/bin/...

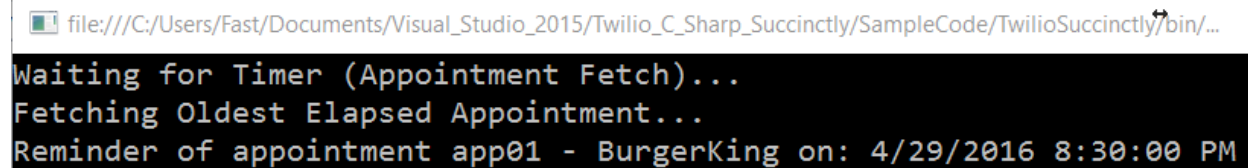
```

Appointment created: app01 - BurgerKing on 4/29/2016 8:30:00 PM
Appointment created: app02 - Run on 4/29/2016 9:20:00 PM
Appointment created: app03 - Gym on 4/29/2016 10:05:00 PM

```

Figure 35: Creation of Several Appointments

After these appointments have been created, we wait for a bit and let the program look for the first expired appointment.



file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/SampleCode/TwilioSuccinctly/bin/...


```










Waiting for Timer (Appointment Fetch)...
Fetching Oldest Elapsed Appointment...
Reminder of appointment app01 - BurgerKing on: 4/29/2016 8:30:00 PM

```

Figure 36: Fetching the First Expired Appointment

We can then inspect the **AppointmentsSuccinctly** table on Azure to see how the data looks after executing the program.

 **AppointmentsSuccinctly** table (3 entities) as of 5/14/2016 2:36:20 AM

 Refresh
  Select All
  Clear All
  Query
  Filter
  Upload
  Downld
  View
  New

PartitionKey	RowKey	Id	Name	PhoneNumber	Sent	Time
33484464	29042016203000	app01	BurgerKing	33484464	True	29042016203000
33484464	29042016212000	app02	Run	33484464	False	29042016212000
33484464	29042016220500	app03	Gym	33484464	False	29042016220500

Figure 37: Appointments Data on Azure Table Storage

We can clearly see the **Sent** property has been set to **True** for the BurgerKing (first expired) appointment.

Automated instant lead alerts

Instant leads are a great way for sales people to close more deals because they allow sales reps to reach out to interested customers moments after they have been identified. Automating the process of reaching out to leads is a fantastic way to put the power of Twilio to work for you.

The process of collecting leads starts with having one or more landing pages in which you collect information from your website visitors about a product or service that your organization offers.

For simplicity's sake, we'll assume that the information being collected is stored on Azure Table Storage. This information will consist of the customer's full name, email address, phone number, and which webinar they are interested in. There will also be a **FollowedUp** property that will be used to indicate whether or not a lead has been followed up on.

We'll create a small application that fetches the lead data from Azure, then sends an SMS using our **TwilioCore** class.


Let's create a **LeadsCore** class responsible for retrieving the leads stored on Azure and send an SMS. We'll manually create the lead records on Table Storage using Azure Storage Explorer.

First, we create a table called LeadsSuccinctly using Storage Explorer. This table will be used to keep track of webinars that potential leads might be interested in. All the fields will have **string** values except for the **FollowedUp** field, which will have a **bool** value.

The table's **PartitionKey** should contain the name of the webinar the lead is interested in, and the **RowKey** should be the string representation of the date and time when the lead was posted on the site.

Just as we did with the appointment's reminder code, we'll use a **Timer** object in order to retrieve the first posted lead.

Let's consider the leads gathered from our website on Azure, as shown in Figure 38.

 **LeadsSuccinctly** table (3 entities) as of 5/4/2016 2:08:18 PM

Refresh Select All Clear All Query Filter Upload Download View

PartitionKey	RowKey	Email	FollowedUp	FullName	PhoneNumber
FPA	29042016210501	vc@vc.com	False	VA Clark	2812816697
SAP	29042016203000	hello@edfreitas.me	False	Ed Freitas	33484464
SAP	29042016203500	ef@ef.com	False	Ed Freitas	33484464

Figure 38: Leads Data on Azure Table Storage

Let's write the code. As with our earlier example, we'll have the core class that interacts with Table Storage, then we'll have a wrapper class around it that actually implements the business logic.

Code Listing 17: LeadsCore Class

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;

namespace TwilioSuccinctly
{
    public class LeadsData : TableEntity
    {
        public LeadsData(string p, string d)
        {
            PartitionKey = p;
            RowKey = d;
        }
        public LeadsData(string p)
        {
            PartitionKey = p;
            RowKey = string.Empty;
        }

        public LeadsData()
        {
            PartitionKey = string.Empty;
            RowKey = string.Empty;
        }

        public string FullName { get; set; }

        public string Email { get; set; }

        public string Webinar { get; set; }
    }
}
```



```

        public bool FollowedUp { get; set; }

        public string PhoneNumber { get; set; }
    }

    public class LeadsCore : IDisposable
    {
        private const string cStrConnStr =
            "<< Your Azure Storage Connection String >>";
        private const string cStrAppTable = "LeadsSuccinctly";

        protected bool disposed;

        public LeadsData Data { get; set; }

        public LeadsCore()
        {
            LeadsData ld = new LeadsData();

            ld.FullName = string.Empty;
            ld.Email = string.Empty;
            ld.Webinar = string.Empty;
            ld.PhoneNumber = string.Empty;
        }

        ~LeadsCore()
        {
            Dispose(false);
        }

        public virtual void Dispose(bool disposing)
        {
            if (!disposed)
            {
                if (disposing)
                {
                    Data = null;
                }
                disposed = true;
            }
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }

```

```

public LeadsData Get(string webinar)
{
    LeadsData res = null;

    try
    {
        int tDay = DateTime.Now.Day;
        int tMonth = DateTime.Now.Month;
        int tYear = DateTime.Now.Year;
        int tHour = DateTime.Now.Hour;
        int tMin = DateTime.Now.Minute;
        int tSec = DateTime.Now.Second;

        DateTime tdy = new DateTime
            (tYear, tMonth, tDay, tHour, tMin, tSec);

        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableQuery<LeadsData> query = new
            TableQuery<LeadsData>().Where(
                TableQuery.GenerateFilterCondition("PartitionKey",
                    QueryComparisons.Equal, webinar));

        foreach (LeadsData en in table.ExecuteQuery(query))
        {
            int day = Convert.ToInt32(
                en.RowKey.Substring(0, 2));
            int month = Convert.ToInt32(
                en.RowKey.Substring(2, 2));
            int year = Convert.ToInt32(
                en.RowKey.Substring(4, 4));
            int hour = Convert.ToInt32(
                en.RowKey.Substring(8, 2));
            int min = Convert.ToInt32(
                en.RowKey.Substring(10, 2));
            int sec = Convert.ToInt32(
                en.RowKey.Substring(12, 2));

            DateTime exp = new DateTime(
                year, month, day, hour, min, sec);

            if (!en.FollowedUp &&

```

```

        DateTime.Compare(tdy, exp) >= 0)
    {
        res = en;
        break;
    }
}
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}

return res;
}

public bool MarkAsFollowedUp(LeadsData ld)
{
    bool res = false;

    try
    {
        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableOperation retrieveOperation =
            TableOperation.Retrieve<LeadsData>(
                ld.PartitionKey, ld.RowKey);

        TableResult retrievedResult =
            table.Execute(retrieveOperation);
        LeadsData updateEntity =
            (LeadsData)retrievedResult.Result;

        if (updateEntity != null)
        {
            updateEntity.FollowedUp = ld.FollowedUp;

            TableOperation insertOrReplaceOperation =
                TableOperation.InsertOrReplace(updateEntity);
            table.Execute(insertOrReplaceOperation);

            res = true;
        }
    }
}

```

```

        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }

        return res;
    }

    public string FormatTime(string t)
    {
        int day = Convert.ToInt32(t.Substring(0, 2));
        int month = Convert.ToInt32(t.Substring(2, 2));
        int year = Convert.ToInt32(t.Substring(4, 4));
        int hour = Convert.ToInt32(t.Substring(8, 2));
        int min = Convert.ToInt32(t.Substring(10, 2));
        int sec = Convert.ToInt32(t.Substring(12, 2));

        DateTime exp = new DateTime(
            year, month, day, hour, min, sec);
        return exp.ToString();
    }
}

```

We have defined a **LeadsData** class that represents the model of the data stored on Table Storage.

This class contains several properties that correspond directly to the fields on the Azure table. These fields are **FullName**, **Email**, **Webinar**, **FollowedUp**, and **PhoneNumber**.

The **LeadsCore** class implements the necessary logic in order to retrieve the first nonfollowed-up lead and to set that lead as followed-up.

The **Get** method receives a string parameter that will be the name of the webinar we are interested in filtering out. This value will be used to query the records on the Azure table using the table's **PartitionKey**.

Following the same logic of the **Get** method of the **Appointments** class, the **Get** method of the **LeadsCore** class connects to Azure, retrieves an **IEnumerable** list of **LeadsData** objects, and checks which have not been followed up (**FollowedUp** set to **false**), and it also checks whether or not the lead's posted date **exp** is older than today's date **tdy**.

The **MarkAsFollowedUp** method is responsible for connecting to the Azure table, for locating the record that corresponds to the lead that was retrieved using the **Get** method, and for setting it as followed-up. The **MarkAsFollowedUp** method basically sets the **FollowedUp** property to **true** so that the same lead cannot be retrieved (followed up) again.

In order to put this into action, let's create a **LeadsExample** class that will wrap around this logic and be responsible for creating the **Timer** object that retrieves the leads.

Code Listing 18: LeadsExample Wrapper Class

```
using System;
using System.Timers;

namespace TwilioSuccinctly
{
    public class LeadsExample
    {
        private const string cStrTwilioSid =
            "<< Your Twilio Account SID goes here >>";
        private const string cStrTwilioAuthTk =
            "<< Your Twilio Auth Token goes here >>";
        private const string cStrTwilioSender =
            "<< Your Twilio Number>>";

        private static bool started = false;

        public static Timer Fetch { get; set; }
        public static string PhoneNumber { get; set; }
        public static string Webinar { get; set; }

        public static void InitFetch(int ms)
        {
            started = false;

            Fetch = new Timer();
            Fetch.Interval = ms;
            Fetch.Elapsed += new ElapsedEventHandler(Fetch_Elapsed);
            Fetch.Start();
        }

        public static void StopFetch()
        {
            started = false;
            Fetch.Stop();
        }

        public static void Fetch_Elapsed(object sender,
            ElapsedEventArgs e)
        {
            if (!started)
            {
                started = true;
                Console.WriteLine
                    ("Fetching Oldest Non-Followed Up Lead...");

                using (LeadsCore l = new LeadsCore())
```

```

        {
            LeadsData ld = l.Get(Webinar);

            if (ld != null)
            {
                using (TwilioCore t = new
                    TwilioCore(cStrTwilioSid, cStrTwilioAuthTk))
                {
                    t.SendSms(cStrTwilioSender, PhoneNumber,
                        $"Please follow up with Lead
                        {ld.PartitionKey} -
                        {ld.Email} - {ld.FullName} on:
                        {l.FormatTime(ld.RowKey)}");

                    Console.WriteLine($"Followed up with Lead
                        {ld.PartitionKey} - {ld.Email} -
                        {ld.FullName} on:
                        {l.FormatTime(ld.RowKey)}");

                    ld.FollowedUp = true;
                }

                l.MarkAsFollowedUp(ld);
            }
        }

        started = false;
    }
}
}
}
}

```

The **InitFetch** method of the **LeadsExample** class creates a **Timer** object (assigned to the **Fetch** property) that will retrieve the leads when the **Fetch_Elapsed** event gets triggered.

The **Fetch_Elapsed** event creates an instance of the **LeadsCore** class that is used in order to execute the **Get** method to retrieve a lead that needs to be followed up. The retrieved lead is then marked as followed-up (**FollowedUp** property set to **true**) and the process repeats itself—another lead is fetched when the **Fetch_Elapsed** event is triggered again by the **Timer** object.

Let's now write the **Main** program in order to invoke the **LeadsExample** wrapper class.

Code Listing 19: Leads Main Program

```

using System;

namespace TwilioSuccinctly

```

```

{
    public class Program
    {
        public static void FollowUpWithLeads()
        {
            Console.WriteLine("Waiting for Timer (Leads Fetch)...");

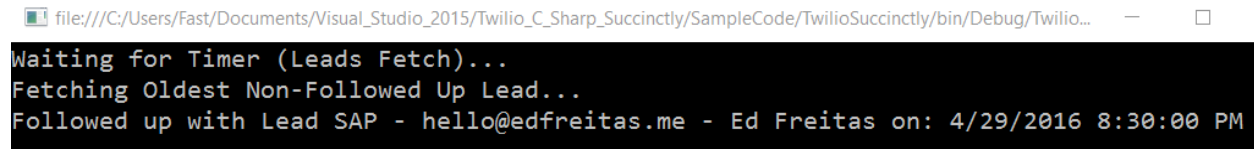
            LeadsExample.PhoneNumber = "33484464";
            LeadsExample.Webinar = "SAP";
            LeadsExample.InitFetch(1500);
        }

        static void Main(string[] args)
        {
            FollowUpWithLeads();

            Console.ReadLine();
        }
    }
}

```

Executing this **program** with the lead data previously posted on the Azure Table Storage table produces the result shown in Figure 39.



```

file:///C:/Users/Fast/Documents/Visual_Studio_2015/Twilio_C_Sharp_Succinctly/SampleCode/TwilioSuccinctly/bin/Debug/Twilio...
Waiting for Timer (Leads Fetch)...
Fetching Oldest Non-Followed Up Lead...
Followed up with Lead SAP - hello@edfreitas.me - Ed Freitas on: 4/29/2016 8:30:00 PM

```

Figure 39: Fetching the First Lead on the Site

Summary

We've seen how easy and straightforward it is to automate a couple of small but useful business processes and how our previously written **TwilioCore** class can be used to send out an SMS. We've also seen how to use Azure Table Storage as a quick and reliable backend for our data needs.

In the following chapters, we'll explore some of the voice features available from Twilio and how they can be used to automate useful business processes.

Chapter 3 Receive and Make Calls

Introduction

We've seen that Twilio is an easy yet powerful platform for enabling SMS communication, for automating certain business processes using SMS, and even for authentication through 2FA.

However, we've only started to scratch the surface of what is possible with Twilio. One of the fundamental and strongest features of this platform is its ability to receive, make, and handle phone calls and build voice-enabled applications.

You are probably familiar with services like [Powwownow](#), [WebEx](#), or [GotoMeeting](#) that allow you to dial a conference call number, set up a virtual meeting room, and speak to other people. You are also likely familiar with automated phone menu systems in which you dial your bank or airline and are asked to press 1, 2, or 3 in order to do X, Y, or Z. We've all been there, haven't we?

Twilio allows us developers to build any of these services specially tailored to suit our own needs—without too much effort and without emptying our pockets. However, we must be cautious about what we are building and what potential costs we might incur. That said, Twilio is almost always cheaper than building an entire telephony infrastructure on our own.

Twilio takes care of the telephony network infrastructure and completely abstracts this complexity away from our development. All we have to do is focus on our business logic and let Twilio deal with the inner workings of telephony systems and protocols.

In this chapter and the next, we'll explore how we can leverage the power of Twilio voice APIs. We'll also use an XML markup language called TwiML that makes it easy to create voice-enabled responses. And we'll look at the basics of using voice features on Twilio, of processing incoming calls and making outbound calls, and of creating a simple answering machine.

By the end of this chapter, you should have the know-how to address these topics and the basics for creating voice-enabled responses and applications with Twilio and TwiML.

In chapter 4, we'll look at more advanced issues with Twilio's voice-enabled features, such as an interactive phone menu, conferencing, and automated voice surveys.

The examples should again be quite easy to follow and fun to implement.

TwiML

In order to easily create voice responses with Twilio, you can use an XML markup language called TwiML, which is simply a set of instructions you can use to tell Twilio what to do when you receive an incoming call or SMS.

When you receive an SMS or a phone call on your Twilio number, Twilio will fetch the URL associated with your Twilio phone number and perform an HTTP(S) request to that URL. This URL will contain an XML response with markup instructions that indicate which tasks Twilio needs to execute. Some of these tasks will record the call, play a message, prompt the caller to enter some digits, etc.

Let's explore a quick example of this in Code Listing 20.

Code Listing 20: TwiML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="man">I'm not available to take your call, please leave a
  message.</Say>
  <Record maxLength="20" />
</Response>
```

As you can see, telling Twilio to execute a specific action using TwiML is relatively simple. The markup language is composed of verbs highlighted in blue that represent actions Twilio will execute.

Some of the TwiML verbs available at the time of this writing are: **Say**, **Play**, **Dial**, **Record**, **Gather**, **Sms**, **Hangup**, **Queue**, **Redirect**, **Pause**, **Conference**, **Reject**, and **Message**.

Complete details on how to use these TwiML verbs can be found [here](#).

Receive incoming calls

Let's walk through the process of handling incoming calls using TwiML.

You'll need to provide a publicly accessible URL where your TwiML XML markup file will be hosted. Twilio allows you to have a default-response TwiML file for both voice and SMS apps. These can be hosted on your own webserver or even on services such as Amazon S3 or Azure Websites that can host static resources.

You can configure these URLs on your Voice Number's [dashboard](#) within your Twilio account, then click your respective Twilio number. This will lead you to the properties of your Twilio number, where you can set the Request URL for Voice responses.

Configure

Event Log

▼ Properties

Friendly Name

SID

Phone Number

Location

Phoenix, AZ US

Capabilities

Voice, SMS, MMS

▼ Voice

Configure with

☒ URL ☐ TwiML App ☐ SIP Trunking

Request URL ?

http://[redacted]/twilio-voice-response.xml

HTTP GET ▼

Figure 40: TwiML URL for Voice Responses

We'll create a quick voice response in Code Listing 21 using TwiML that greets the caller and instructs to send a WhatsApp message or write an email.

Code Listing 21: TwiML Caller Greeting Response

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Pause length="1"/>
  <Say voice="man">Hello.</Say>
  <Say voice="man">You've reached Ed's phone.</Say>
  <Say voice="man">I cannot take your call now.</Say>
  <Say voice="man">
Please send a Whats App message, to this same number.
</Say>
  <Pause length="1"/>
  <Say voice="man">You may also contact me at</Say>
  <Say voice="man">hello, at</Say>
  <Say voice="man">ed freitas dot me</Say>
  <Pause length="1"/>
  <Say voice="man">Thank you.</Say>
  <Say voice="man">Have a great day.</Say>
</Response>
```

Notice that the response has been broken down into several parts by using multiple **Say** verbs and also by using one-second **Pause** verbs in between. This happens because Twilio uses a text-to-speech engine in order to read out the written text to the caller.

In order to make the experience seem as much like a human answering the call as possible, it is important to relay the message in parts and with the necessary pauses. Also notice that the sentences are not long and that punctuation signs are used.

There's no rule of thumb to make a "perfect" TwiML text-to-voice response. In order to get this sounding less like a machine, you'll probably have to play with it for a bit until you get the response sound to your liking.

TwiML is a great resource that allows us to quickly add voice functionality with no programming required. However, because responses are based on XML markup, they will need to be statically hosted or embedded within server-side software such as PHP or ASP.NET.

One of TwiML's limitations as a markup language is that it doesn't allow for the usage of variables, so if we want to have responses that are not predefined, we must construct the XML on the fly on the server and then return it as a response.

We can get around this by creating customized TwiML responses. Let's explore how.

Simple Answering Machine

After having explored some of the basics of TwiML, let's create a Simple Answering Machine that will provide specific instructions to callers based on digits they type in after the call has been answered.

We'll create a **SimpleAnsweringMachine** class that will contain the logic required by our answering machine application.

As TwiML requires a URL endpoint, we'll create a [NancyFX](#) application that we can deploy as an Azure [website](#).

If you would like more in-depth information about NancyFX, Syncfusion has a fully dedicated [e-book](#) that is a great resource about the framework and its capabilities. Feel free to have a look at it.

In order to create a NancyFX application, first make sure to install the [SideWaffle Template Pack for Visual Studio](#). The SideWaffle installer is easy to follow and requires only a few clicks.

After SideWaffle has been installed, you may create a new NancyFX project called *TwilioNancyAspNetHost* by selecting *Nancy empty project with ASP.NET host* as follows in Figure 41.

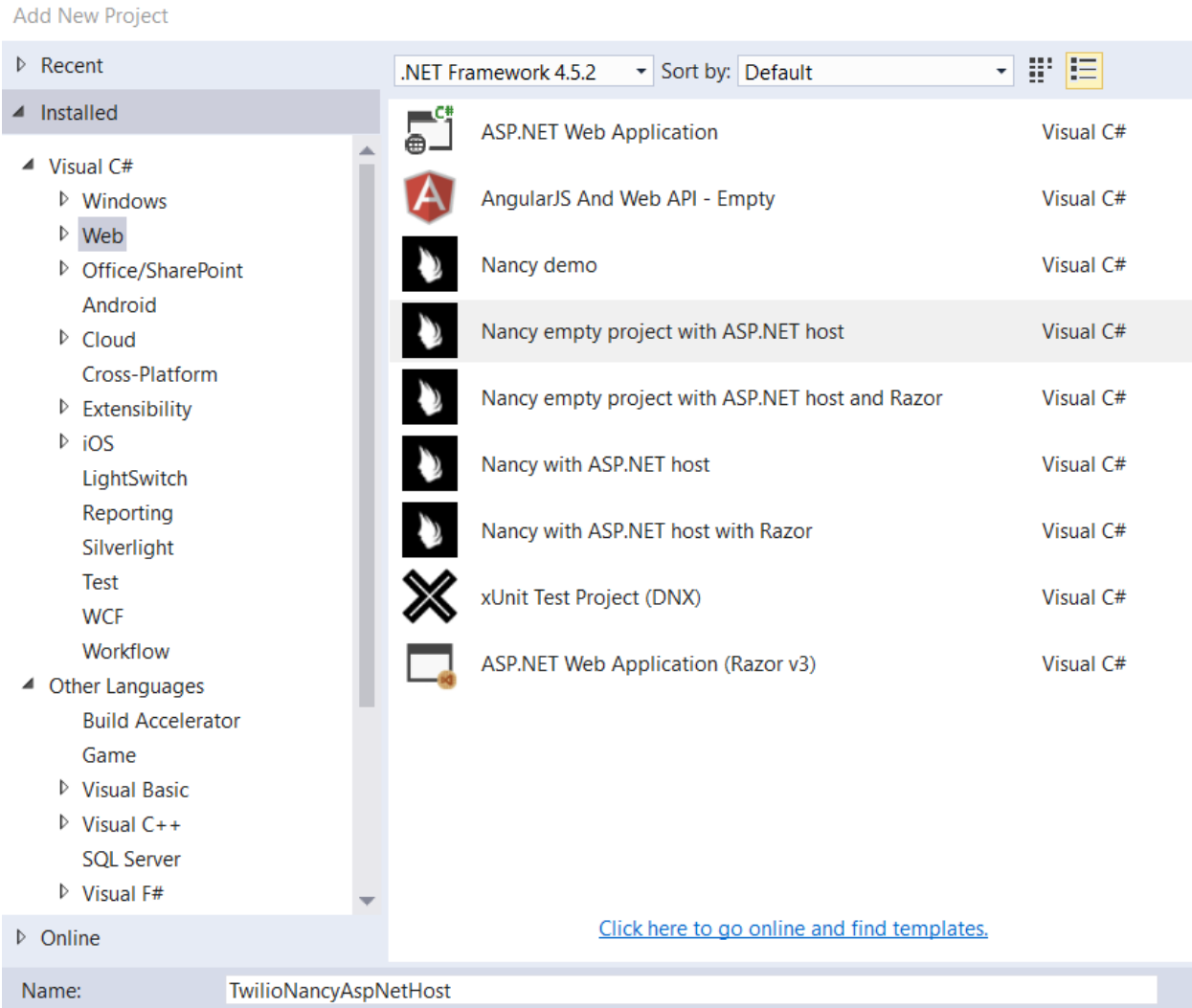


Figure 41: Creating a New NancyFX Project

With the project created, the next step is to create a NancyFX module, which is simply a standard C# class. In Figure 42, we'll call this module *AnsweringMachineMain*.

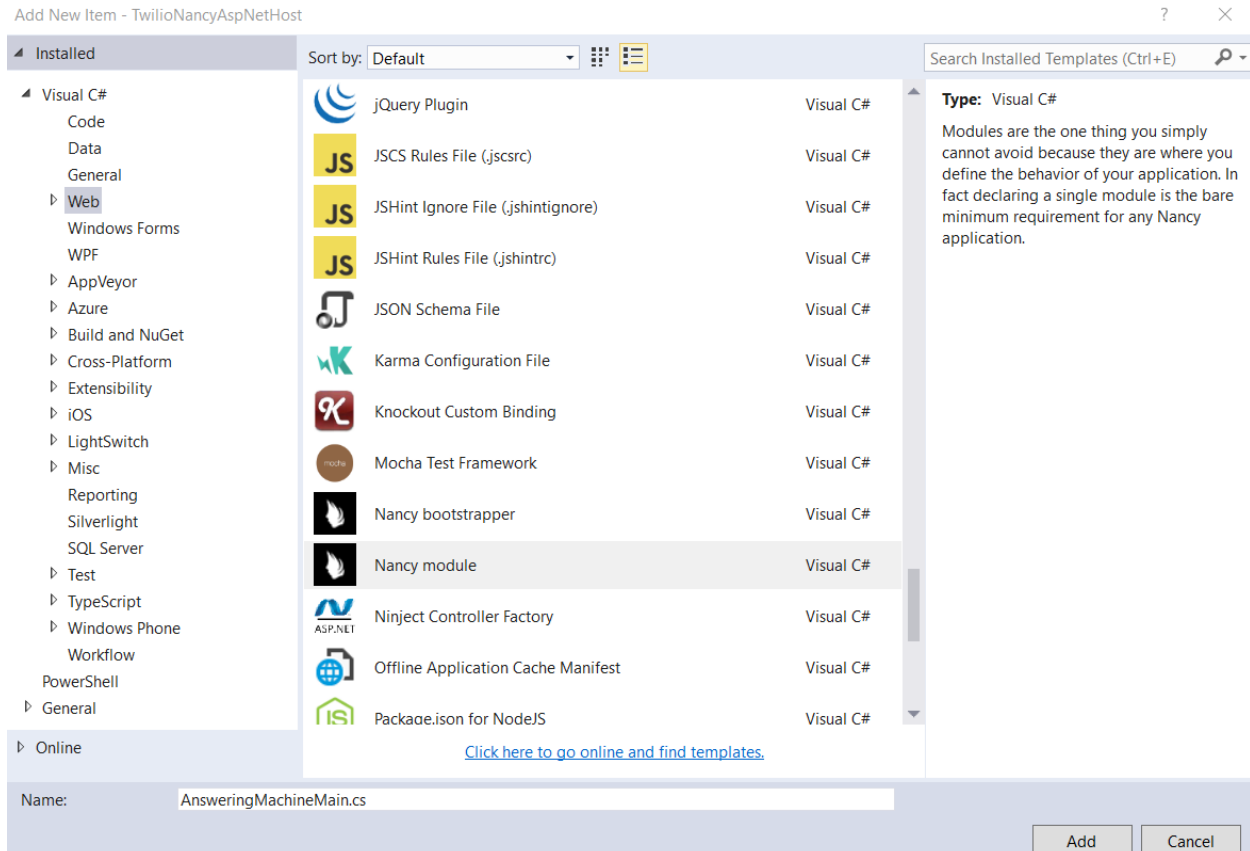


Figure 42: Creating a New NancyFX Module

Although unlikely, if for some reason your NancyFX reference is not found within your *TwilioNancyAspNetHost* project, use the NuGet Package Manager to check for any new updates.

Our *AnsweringMachineMain* module will contain the HTTP REST endpoint(s) that will be used in order send the correct response to the caller. Just to test that this works, we can include the code in Code Listing 22.

Code Listing 22: TwiML Caller Greeting Response

```
using Nancy;

namespace TwilioNancyAspNetHost
{
    public class AnsweringMachineMain : NancyModule
    {
        public AnsweringMachineMain()
        {
            Get["/"] = _ =>
            {
                return Response.AsText(
```

In order to check that our REST endpoint is publicly accessible, we can deploy our NancyFX application to Azure Websites with the Publish option in Visual Studio.

Figure 43: Deploying the NancyFX Application to Azure Websites

Enter the requested details, click Create, and follow the steps through the interactive wizard. In a few clicks, the solution will be deployed to Azure and ready to use.

The site should be accessible through this [URL](#). There, you should see what is depicted in Figure 44.

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="man">
    I'm not available to take your call, please leave a message.
  </Say>
  <Record maxLength="20"/>
</Response>
```

Figure 44: The TwiML Response from Our NancyFX Application

With our NancyFX application up and running, next we'll wire up our Twilio number to point to our NancyFX application running on Azure.

In order to do this, we need to go to our Twilio number's [dashboard](#), then click on the number to which we want to link. You'll be asked to create a new TwiML application.

Create TwiML App

Properties

Friendly Name

Voice

Request URL HTTP POST

Messaging

Request URL HTTP POST

Figure 45: Creating a New TwiML App

Enter the name of the TwiML app (you can specify any name you want) and, most importantly, indicate the Voice Request URL, which is the URL that returns the TwiML response from our NancyFX application. Once these details have been provided, click Save.

In order to test that it all works, simply pick up your phone and dial your Twilio number. If you hear the same message you typed in your NancyFX XML response, it's all fine.

With the setup and wiring in place, we can now focus on adding the features required in order to expand the functionality of our answering machine application.

The application should quickly greet callers and ask them to type in their access code in order to give back a secret code to be stored on an Azure Storage Table called SAMSuccinctly.

Because our NancyFX (Azure website) application will be using Azure Storage, don't forget to install the package from NuGet.

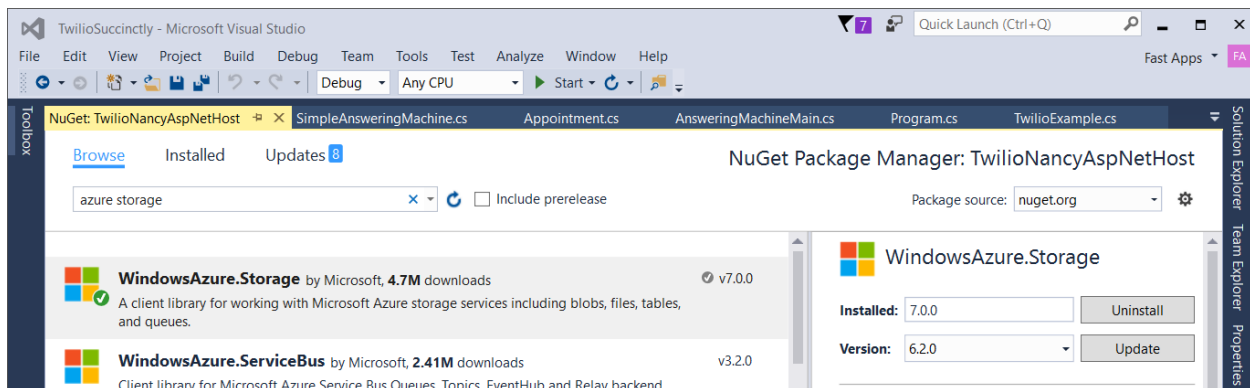


Figure 46: Azure Storage NuGet Package Installed for Our NancyFX Application

We can now create our SAMSuccinctly table using Azure Storage Explorer as follows in Figure 47. Let's store the access code as the **PartitionKey** and the secret code as the **RowKey**.

 A screenshot of the Azure Storage Explorer interface. It shows a table named 'SAMSuccinctly table'. Above the table are buttons for 'Refresh', 'Select All', and 'Clear All'. The table has two columns: 'PartitionKey' and 'RowKey'. There are two rows of data: the first row has '1234' as the PartitionKey and 'aBcDeFg' as the RowKey; the second row has '5678' as the PartitionKey and 'Z-A-B-N-G-2' as the RowKey.

PartitionKey	RowKey
1234	aBcDeFg
5678	Z-A-B-N-G-2

Figure 47: The SAMSuccinctly Table on Azure

We can use the following TwiML markup in order to greet callers and ask them for their access code.

Code Listing 23: TwiML Markup Greeting for Our App

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- page located at http://twiliosuccinctly.azurewebsites.net/ -->
<Response>
  <Gather action="/sam" method="GET">
    <Say>
      Please enter your access code,
      followed by the pound sign.
    </Say>
  </Gather>
</Response>
```



```
</Gather>
<Say>We didn't receive any access code. Bye.</Say>
</Response>
```

Notice that this greeting response, which is responsible for gathering the access code, is hosted on the root of the website (<http://twiliosuccinctly.azurewebsites.net/>). However, the action response is available on /sam (<http://twiliosuccinctly.azurewebsites.net/sam>).

Now, let's implement our **SimpleAnsweringMachine** class.

Code Listing 24: SimpleAnsweringMachine Class

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;

namespace TwilioSuccinctly
{
    public class SimpleAnsweringMachineData : TableEntity
    {
        public SimpleAnsweringMachineData(string p, string d)
        {
            PartitionKey = p;
            RowKey = d;
        }

        public SimpleAnsweringMachineData()
        {
        }
    }

    public class SimpleAnsweringMachine : IDisposable
    {
        protected bool disposed;

        private const string cStrConnStr =
            "<< Your Azure Storage Connection String >>";
        private const string cStrAppTable = "SAMSuccinctly";

        public SimpleAnsweringMachine()
        {
        }

        ~SimpleAnsweringMachine()
        {
            Dispose(false);
        }
    }
}
```

```

public virtual void Dispose(bool disposing)
{
    if (!disposed)
    {
        if (disposing)
        {
            //
        }
    }

    disposed = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

public string Get(string ac)
{
    string res = string.Empty;

    try
    {
        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableQuery<SimpleAnsweringMachineData> query = new
            TableQuery<SimpleAnsweringMachineData>().Where(
                TableQuery.GenerateFilterCondition("PartitionKey",
                    QueryComparisons.Equal, ac));

        foreach (SimpleAnsweringMachineData en in
            table.ExecuteQuery(query))
        {
            res = en.RowKey;
            break;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

```

```

        return (res == string.Empty) ? "Unknown" : res;
    }
}

```

Our **SimpleAnsweringMachine** class is actually very simple. It has a **Get** method that queries the **SAMSuccinctly** table using the access code (represented by the **PartitionKey**) and returns a **SimpleAnsweringMachineData** object that also contains the secret code.

As you can see, querying the **SAMSuccinctly** table looks similar to what we saw in the previous chapter using Azure classes.

Now, in Code Listing 25 let's look at expanding the features of our **AnsweringMachineMain** NancyFX module so that we can implement the functionality exposed by our **SimpleAnsweringMachine** class.

Code Listing 25: Expanded AnsweringMachineMain Class

```

using Nancy;
using TwilioSuccinctly;

namespace TwilioNancyAspNetHost
{
    public class AnsweringMachineMain : NancyModule
    {
        public AnsweringMachineMain()
        {
            Get["@"/"] = _ =>
            {
                return Response.AsText(
                    "<Response>" +
                    "<Gather action=\"/sam\" method=\"GET\">" +
                    "<Say voice=\"woman\">Please enter your access  

                    code, followed by the pound sign.</Say>" +
                    "</Gather>" +
                    "<Say voice=\"woman\">We didn't receive any  

                    access code. Bye.</Say>" +
                    "</Response>",
                    "text/xml"
                );
            };

            Get["@/sam"] = _ =>
            {
                string sc = "Unknown";
                string ac = Request.Query["Digits"].Value;

                string res = "<Response><Say voice=\"woman\">No access

```

```

        code found.</Say>" +
        "</Response>";

    if (ac != null && ac != string.Empty)
    {
        using (SimpleAnsweringMachine sam = new
            SimpleAnsweringMachine())
        {
            if (Request.Query["Digits"] != string.Empty)
                sc = sam.Get(ac);
        }

        res = "<Response><Say voice=\"woman\">You entered: "
            + ac + "</Say>" +
            "<Say voice=\"woman\">Your secret code is: " +
            sc + "</Say>" +
            "</Response>";
    }

    return Response.AsText(res, "text/xml");
};
}
}
}
}

```

Our **AnsweringMachineMain** NancyFX module implements two REST endpoints. The root/endpoint provides the greeting message to the caller and gathers the access code through the **Gather** TwiML verb, which triggers an action on the /sam endpoint.

When the /sam endpoint gets executed (once the caller has entered the access code digits and pressed the pound sign), an instance of the **SimpleAnsweringMachine** class is created and its **Get** method is invoked, passing the digits entered by the caller (and are stored on the HTTP **Request.Query** object).

The **Get** method then queries the **SAMSuccinctly** table and returns the **RowKey** (which represents the secret code) that matches the **PartitionKey** (access code) being queried.

In order for the NancyFX application to run properly, the *TwilioNancyAspNetHost* project needs to be published on the Azure website. This app can be easily tested by picking up the phone and calling the Twilio number associated with this TwiML application.

You should be able to provide your access code (one of the two existing on the **SAMSuccinctly** table) and get your secret code as a voice response.

Make outbound calls

We've seen how we can receive calls and process voice requests using TwiML. Before closing this chapter and moving into more advanced examples, let's discuss how we can make outbound calls using the Twilio C# helper library.

We'll expand our **TwilioCore** class and add a method to make calls. Let's implement this method in Code Listing 26

Code Listing 26: The TwilioCore StartCall Method

```
public void StartCall(string from, string to, string url)
{
    // SID = Twilio AccountSID, aTk = Twilio AuthToken
    if (SID != string.Empty && aTk != string.Empty && from !=
        string.Empty && to != string.Empty)
    {
        TwilioRestClient client = new TwilioRestClient(SID, aTk);
        client.InitiateOutboundCall(from, to, url);
    }
}
```

As you can see, the **StartCall** method is very simple. It has two parameters: **from** and **to**, which, respectively, represent the called and receiver phone numbers.

The third parameter is a URL resource that the person on the other side of the phone will hear once the call is picked up. This URL needs to return a valid TwiML response.

As we've been doing all along when using our **TwilioCore** class methods, let's implement a wrapper method around **StartCall** that we can invoke from the **Main** program.

Code Listing 27: The StartRecordedCall Wrapper Method

```
public static void StartRecordedCall()
{
    using (TwilioCore tc = new TwilioCore(cStrSid, cStrAuthTk))
    {
        tc.StartCall(cStrSender, cStrReceiver,
            "http://www.edfreitas.me/twilio/sayhello.xml");
    }
}
```

Now, we can call this method from the **Main** program.

Code Listing 28: Calling the StartRecordedCall Wrapper Method from the Main Program

```
static void Main(string[] args)
```

```
{  
    TwilioExample.StartRecordedCall();  
  
    Console.ReadLine();  
}
```

When executing this code, you should hear a recording on the phone number that you have indicated as **cStrReceiver** within your **TwilioExample** class.

As we've seen, making outbound calls with Twilio is easy. In just a few lines of code, we've been able to start a phone call and play a message.

If the destination number you will be dialing is an international (non-US) number, you'll need to enable certain permissions to allow Twilio to perform that action. These permissions can be checked and configured at this [URL](#).

You can also adapt this code to enable the application to give a personalized message or record reactions to it.

Summary

In this chapter, we've started to scratch the surface of what is possible with Twilio's voice-enabled features.

We've seen how to receive incoming calls, give specific feedback based on a user's input, and make outbound calls.

We've also seen how to set up a quick NancyFX Azure website in order to create a simple answering machine.

In the next chapter, we'll dig a bit deeper into using Twilio's voice-enabled features and create some interesting and more advanced samples that can be useful in a business context.

Chapter 4 Automation Using Voice

Introduction

We've explored the basics of Twilio's voice capabilities, which has hopefully given you a good idea of what is possible to do with TwiML and the C# helper library in order to create voice-enabled applications.

Let's now explore how we can expand a bit more and use Twilio's voice-enabled features in order to automate business processes.

We'll mostly use TwiML in order to create an interactive phone menu, a simple conferencing service, and an exciting, automated voice-survey solution.

We'll also be relying on Azure Table Storage and Azure Websites, just as in Chapter 3, and because we are basing the code that follows on Chapter 3's code, the examples should again be easy to follow and fun to implement.

Interactive Phone Menu

An Interactive Phone Menu is an automated phone software system that can facilitate and coordinate the communication between callers and businesses. If you've ever dialed your insurance company to file a claim and responded to a series of automated prompts, you've used one of these systems.

Twilio makes it easy to build this. We've already explored some of the basics by building a simple answering machine in Chapter 3. Now, let's create a phone menu system for a company that has departments located in different regions. This system will connect the caller with a specific department based on the digit entered by the caller.

In order to start the phone menu, we need to configure our Twilio number so that it will send our web application an HTTP request when we receive an incoming call. We'll use and modify the existing NancyFX app we created in the previous chapter.

First, we [configure](#) our Twilio number to point to a specific URL that our web app will use for processing incoming calls. Next, we modify the TwiML app we called TwilioSuccinctly to point to our new incoming URL. This URL should be the location of our NancyFX app on Azure Websites followed by `/ipm/incoming`.

Using Visual Studio, let's publish our NancyFX application with this change to Azure Websites. If we then call our Twilio number, we should now hear this response. Hurray!

Moving forward, the **Gather** verb on the TwiML response points to another endpoint: `/ipm/response`. We'll use this endpoint to create our menu response based on the input entered by the caller.

Note that we can still use and expand our **SimpleAnsweringMachine** class in order to add any specific logic required for our phone menu.

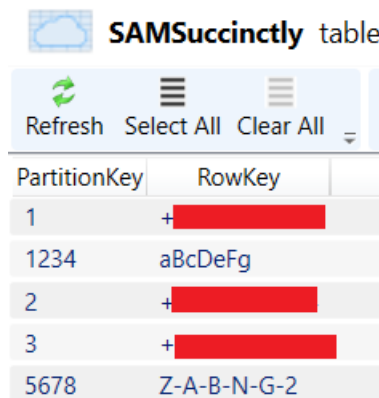
Having a method that inserts a space after each char within a string is also useful. This is done so that Twilio can read the phone number digit-by-digit rather than by reading the entire number at once. We'll explore that in a bit.

First, let's process the caller's input and create our `/ipm/response` endpoint on our **AnsweringMachineMain** NancyFX module.

Just as we did with our simple answering machine example in Chapter 3, we'll query the **SAMSuccinctly** table on Azure and retrieve a value.

However, this value will be a phone number that corresponds to the option the user entered. The digit the user entered will be the **PartitionKey** and the phone number retrieved will be the **RowKey**. The phone number needs to be in an international format, i.e. `+14800001111`.

Let's add three more rows to our **SAMSuccinctly** table using Azure Storage Explorer.



PartitionKey	RowKey
1	+ [REDACTED]
1234	aBcDeFg
2	+ [REDACTED]
3	+ [REDACTED]
5678	Z-A-B-N-G-2

Figure 49: The Three Options and Numbers for Our Menu System

This phone number will be used to connect the caller with the corresponding department within the company.

Code Listing 30: The Interactive Phone Menu Response Endpoint

```
Get["@/ipm/response"] = _ =>
{
    string num = "Unknown";
```

```

string ac = Request.Query["Digits"].Value; // Access Code

string res = "<Response><Say voice=\"woman\">No phone number found
             for that option.</Say>" +
             "</Response>";

if (ac != null && ac != string.Empty)
{
    if (ac == "1" || ac == "2" || ac == "3")
    {
        using (SimpleAnsweringMachine sam = new
                SimpleAnsweringMachine())
        {
            if (Request.Query["Digits"] != string.Empty)
            {
                num = sam.Get(ac);

                res = "<Response><Say voice=\"woman\">You entered: "
                    + ac + "</Say>" +
                    "<Say voice=\"woman\">I'll transfer you now to: "
                    + sam.InsertSpace(num) +
                    ", please hold on.</Say>" +
                    "<Dial timeout=\"10\" record=\"false\">" + num
                    + "</Dial>" +
                    "</Response>";
            }
        }
    }
}

return Response.AsText(res, "text/xml");
};

```

As you can see, we are again building upon what we have already learned in Chapter 3.

The main difference is that instead of using the verb **Say** to provide the answer (as we did in the previous chapter), we use the verb **Dial** to connect callers with the department that corresponds to the option they entered in the phone menu.

Another difference is that we now have a program-defined **InsertSpace** method within our **SimpleAnsweringMachine** class that inserts a space after each char within a string. This is applied on the phone number so that Twilio can read it digit-by-digit rather than as a complete number.

The implementation of the **InsertSpace** method within our **SimpleAnsweringMachine** class looks like Code Listing 31.

```
public string InsertSpace(string s)
{
    string res = string.Empty;

    int count = 1;
    foreach (char c in s)
    {
        if (char.IsDigit(c) && count < s.Length)
            res += c + " ";
        else
            res += c;

        count++;
    }
    return res;
}
```

If you now publish this application within Visual Studio to Azure Websites, when you dial your Twilio number you should now hear the phone menu, be able to enter 1, 2, or 3, and get connected to the phone number that corresponds to the option selected. This is possible thanks to the data stored on the Azure table.

This example could be further improved with more interactivity, but that's beyond the scope of what we'll cover here. However, you are encouraged to check out the Twilio [documentation](#) and keep adding more features.

Let's build some other cool examples.

Simple conferencing service

Services such as [Powwownow](#), [WebEx](#), or [GotoMeeting](#) that allow you to dial a conference call number, set up a virtual meeting room, and speak to other people are known as conferencing services.

Twilio also allows us to build these kind of services. We'll explore the basics of how we can set up a small conferencing app.

First, let's expand our NancyFX app in order to include this functionality.

In the same way we configured our phone menu, let's [configure](#) our Twilio number to point to a specific URL that our web app will use for processing incoming calls.

Next let's modify our TwiML app called TwilioSuccinctly to point to our new incoming URL. This URL should contain the location of our NancyFX app on Azure Websites followed by `/conf/incoming`.

The Voice Request URL for our TwilioSuccinctly TwiML app should look like Figure 50.

Voice

Request URL ⓘ HTTP GET ▾

Figure 50: Voice Request URL for TwilioSuccinctly TwiML App

Now, let's create our conferencing service greeting REST endpoint on our **AnsweringMachineMain** NancyFX module, as shown in Code Listing 32

Code Listing 32: The Conference Service Greeting

```
Get["@"/conf/incoming"] = _ =>
{
    string res = "<Response>" +
        "<Gather action=\"/conf/connect\" method=\"GET\">" +
        "<Say voice=\"woman\">Welcome to the coolest conference!</Say>" +
        "<Say voice=\"woman\">Press 7 to join as a listener.</Say>" +
        "<Say voice=\"woman\">Press 8 to join as a speaker.</Say>" +
        "<Say voice=\"woman\">Press 9 to join as the moderator.</Say>" +
        "</Gather>" +
        "<Say voice=\"woman\">We didn't get any response. Bye.</Say>" +
        "</Response>";

    return Response.AsText(res, "text/xml");
};
```

This looks no different than what we've previously seen. However, things are about to change a bit. In order to create a conference, we'll need to use the TwiML **Conference** verb with a few attributes.

We do this in order to indicate if the person joining will be as a listener (on mute), a speaker (not on mute), or the moderator. The conference doesn't actually begin until a moderator has joined, and this is determined by the verb's attributes.

We'll do this in a two-step process. After we greet callers, we'll ask them to choose how they want to join. The **Gather** verb does this by sending a request to the `/conf/connect` endpoint.

Once the caller has chosen how they want to join the conference call, an access code is requested.

Code Listing 33: The Conference Service Connection

```
Get["@"/conf/connect"] = _ =>
{
```

```

string val = "Unknown";
string moderator = false.ToString();
string muted = false.ToString();

string ac = Request.Query["Digits"].Value;

string res = "<Response><Say voice=\"woman\">No valid option or
            access code found.</Say>" +
            "</Response>";

if (ac != null && ac != string.Empty)
{
    using (SimpleAnsweringMachine sam = new SimpleAnsweringMachine())
    {
        if (ac == "7" || ac == "8" || ac == "9")
        {
            if (Request.Query["Digits"] != string.Empty)
            {
                val = sam.Get(ac);

                res = "<Response><Say voice=\"woman\">You entered: "
                    + ac + "</Say>" +
                    "<Gather action=\"/conf/connect\"
                    method=\"GET\">" +
                    "<Say voice=\"woman\">Please enter your access
                    code.</Say>" +
                    "</Gather>" +
                    "<Say voice=\"woman\">We didn't get any
                    response. Bye.</Say>" +
                    "</Response>";
            }
        }
        else if (ac == "77" || ac == "88" || ac == "99")
        {
            if (Request.Query["Digits"] != string.Empty)
            {
                val = sam.Get(ac);

                if (val == "conf-muted")
                    muted = true.ToString();

                if (val == "is-moderator")
                    moderator = true.ToString();

                string typ =
                    (Convert.ToBoolean(moderator)) ? " the moderator" :
                    (Convert.ToBoolean(muted)) ?
                    " a listener" : " a speaker";
            }
        }
    }
}

```

```

        res = "<Response><Say voice=\"woman\">You entered: " +
            ac + "</Say>" +
            "<Say voice=\"woman\">You're being connected to" +
            "the conference now as: " + typ +
            ", please hold on.</Say>" +
            "<Pause length=\"2\"/>" +
            "<Say voice=\"woman\">You have joined the" +
            "conference.</Say>" +
            "<Dial>" +
            "<Conference startConferenceOnEnter=\"\" +
            moderator + "\" endConferenceOnExit=\"\" +
            moderator + "\" muted=\"\" +
            muted + "\" waitUrl=\"http://bit.ly/20gwxz4\"\" +
            ">CoolestConf</Conference>" +
            "</Dial>" +
            "</Response>";
    }
}
}

return Response.AsText(res, "text/xml");
};

```

This logic is easy to follow. After the access code has been gathered, the **Gather** verb sends a request to itself (again to the /conf/connect endpoint), then the access code is verified.

If the access code exists on the Azure table, the conference is initiated by returning a response that contains the **Conference** verb with the correct attribute values set, which will depend on the menu options chosen (which in turn is determined by the value of the access code provided).

In order to make this work, create the values on the SAMSuccinctly table shown in Figure 51.

7	77
77	conf-muted
8	88
9	99
99	is-moderator

Figure 51: Conference Service Options and Access Codes

The menu option (**PartitionKey**) with a value of 7 has an access code (**RowKey**) of 77.

The menu option (**PartitionKey**) with a value of 8 has an access code (**RowKey**) of 88.

The menu option (**PartitionKey**) with a value of 9 has an access code (**RowKey**) of 99.

The access code (**PartitionKey**) with a value of 77 indicates that the caller joins the conference on mute (**RowKey**).

The access code (**PartitionKey**) with a value of 99 indicates that the caller joins the conference as a moderator (**RowKey**).

The code assigns the corresponding values to the correct **Conference** verb attributes, such as **startConferenceOnEnter**, **endConferenceOnExit**, and **muted**.

Because the value of **startConferenceOnEnter** and **endConferenceOnExit** is set to the value of the moderator variable, the conference starts only when the moderator joins, and it ends when the moderator leaves.

The **waitUrl** attribute points to background music that plays until the conference starts.

If the person joining is a listener, the XML response generated by the app will look like Code Listing 34.

Code Listing 34: The TwiML Markup Response for a Listener

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="woman">You entered: 7</Say>
  <Say voice="woman">You're being connected to the conference now as,
  please hold on.</Say>
  <Pause length="2"/>
  <Say voice="woman">You have joined the conference.</Say>
  <Dial>
    <Conference startConferenceOnEnter="false" endConferenceOnExit="false"
    muted="true" waitUrl="http://bit.ly/20gwzz4">CoolestConf</Conference>
  </Dial>
</Response>
```

Notice that with a listener, the **startConferenceOnEnter** and **endConferenceOnExit** values are set to **false** and **muted** is set to **true**.

Let's now examine the XML response generated for a person who joins as a speaker.

Code Listing 35: The TwiML Markup Response for a Speaker

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="woman">You entered: 7</Say>
  <Say voice="woman">You're being connected to the conference now as,
  please hold on.</Say>
  <Pause length="2"/>
  <Say voice="woman">You have joined the conference.</Say>
  <Dial>
```

```
<Conference startConferenceOnEnter="false" endConferenceOnExit="false"
muted="false" waitUrl="http://bit.ly/20gwxx4">CoolestConf</Conference>
</Dial>
</Response>
```

Notice that with a speaker, the **startConferenceOnEnter** and **endConferenceOnExit** values are set to **false** and **muted** is set to **false**.

Finally, let's examine the XML response generated for a person who joins as a moderator.

Code Listing 36: The TwiML Markup Response for a Moderator

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
<Say voice="woman">You entered: 7</Say>
<Say voice="woman">You're being connected to the conference now as,
please hold on.</Say>
<Pause length="2"/>
<Say voice="woman">You have joined the conference.</Say>
<Dial>
<Conference startConferenceOnEnter="true" endConferenceOnExit="true"
muted="false" waitUrl="http://bit.ly/20gwxx4">CoolestConf</Conference>
</Dial>
</Response>
```

Notice that with a moderator, the **startConferenceOnEnter** and **endConferenceOnExit** values are set to **true** and **muted** is set to **false**.

In order to test this, simply deploy to Azure Websites by using the publish option within Visual Studio. Call your Twilio number and you are ready to go.

You'll need at least two phones in order to test this by simulating two users joining the conference. It's pretty cool what Twilio can do with a few lines of code and a couple of verbs and attributes.

Obviously, more functionality could be added to this conferencing demo, such as announcing the names of the callers, recording the conference, and other interactive voice options. The possibilities are endless.

I encourage you to visit the Twilio [documentation](#) to learn more about all the attributes that the **Conference** verb offers and to explore other possibilities.

Let's move on and look at our final example. We'll see how we can implement an automated voice survey solution, which should be quite interesting and a lot of fun.

Automated voice survey

An automated survey allows end users to call a phone number that will be used to conduct a survey that asks questions and prompts the user to provide answers to each question.

Phone surveys are a great way to gather valuable data from our users, and they have many business applications, especially in regards to brand awareness, product insights, and customer satisfaction.

First, let's explain how a voice survey should work using Twilio.

Twilio gets the call and makes an HTTP request to our TwilioSuccinctly TwiML application—e.g., asking for instructions on how to respond.

The NancyFX app instructs Twilio (using TwiML) to **Gather** the user's voice input. The user will be asked to provide a unique ID, and we'll identify them by that.

After each question, Twilio will make another request to our NancyFX app with the user's input, which our application stores on an Azure table.

After storing the answer, the app will continue instructing Twilio to redirect the user to the next question until the survey is finished. In a nutshell, that's what the voice survey app should do.

We'll expand our **SimpleAnsweringMachine** class to include a method that saves data on an Azure table we'll call ResponsesSuccinctly.

Our ResponsesSuccinctly table has three fields. On the **PartitionKey**, we'll store the question's ID. On the **RowKey**, we'll store the user's ID. Finally, we'll have a third field called **Response** that will also be a string and will contain the user's response.

We'll also need two other tables—one called Questions that will contain the ID of the question (**PartitionKey**) and the actual question itself as a string. We'll also need a PredefinedAnswers table that will contain possible answers for each question.

So, our survey app will read from the Questions and PredefinedAnswers tables and store the user's input on the ResponsesSuccinctly table on Azure.

Let's create the Questions and PredefinedAnswers tables using Azure Storage Explorer. We'll start with the Questions table and populate it with sample data.

Questions table (3 entities) as of 5/21/2016 2:07:44 PM

Refresh Select All Clear All Query Filter Upload Download

PartitionKey	RowKey	Question
1	BelieveUFOs	Do you believe in UFO's?
2	SeenUFOs	Have you seen any UFO's?
3	WhenUFOs	When have you seen a UFO?

Figure 52: The Questions Table on Azure

You can see that the **RowKey** of the table is a short ID representation of the question itself, and the actual question is stored on the **Question** field, which is also a string.

Let's now create the PredefinedAnswers table with Azure Storage Explorer and populate it with this sample data.

PredefinedAnswers table (9 entities)

Refresh Select All Clear All Query Filter

PartitionKey	RowKey	Answer
a1	q1	Yes
a1	q2	Yes
a1	q3	Never
a2	q1	No
a2	q2	No
a2	q3	Cannot remember when
a3	q2	Not sure
a3	q3	In the last 6 months
a4	q3	None of the previous

Figure 53: The PredefinedAnswers Table on Azure

On the PredefinedAnswers **PartitionKey**, we store the answer number and on the **RowKey** we store the question number. The **Answer** field, which is a string, contains the actual answer itself.

We can see all questions have multiple predefined answers, some more than others, and these will be the options presented to the users. Their answers will then be stored on the ResponsesSuccinctly table.

With the theory behind us, let's write some code. First, let's [point](#) our TwiML app to our new endpoint, which we will call /survey/incoming.

Voice

Request URL ?	<input type="text" value="http://twiliosuccinctly.azurewebsites.net/survey/incoming"/>	HTTP GET ▾
---------------	--	------------

Figure 54: Voice Request URL for Our TwilioSuccinctly TwiML App

Now, let's define our survey greeting. We'll modify our **AnsweringMachineMain** NancyFX module to accommodate for this.

Code Listing 37: The Automatic Voice Survey Greeting

```
Get[@"/survey/incoming"] = _ =>
{
    string res = "<Response>" +
        "<Gather action=\"/survey/connect\" method=\"GET\">" +
        "<Say voice=\"woman\">You are about to join the UFO survey.</Say>" +
        "<Say voice=\"woman\">What is your Social Security Number?</Say>" +
        "</Gather>" +
        "<Say voice=\"woman\">We didn't get any response. Bye.</Say>" +
        "</Response>";

    return Response.AsText(res, "text/xml");
};
```

The greeting requests that the caller provide a Social Security Number for identification. Keep in mind that in a nondemo scenario, you'd never ask a user for a Social Security Number! When the information has been gathered, a request will be sent to the `/survey/connect` REST endpoint, which will handle most of the remaining survey logic. We'll later need to slightly modify this code in order to initialize state (we will cover this later).

The Social Security Number value will be stored on the `ResponsesSuccinctly` table and will identify the user with the corresponding answers provided.

The application will not do any validation on the Social Security Number (in order to verify if it is a valid or unique number), as the purpose is to simply demonstrate how the information can be gathered through interactive options. The application can always be improved upon and expanded with more specific logic that can validate this or other aspects.

Before we dig into the details of the `/survey/connect` endpoint, we need to add extra features to our **SimpleAnsweringMachine** class to include a method that saves data to the `ResponsesSuccinctly` table, and we need to add a couple of other methods that fetch information from the `Questions` and `PredefinedAnswers` tables.

Let's now create a method that reads from the `Questions` table and another that reads from the `PredefinedAnswers` table.

We'll need to create a **QuestionsData** class that corresponds to the structure of the `Questions` Azure table. Code Listing 38 shows how we can define this class.

Code Listing 38: The QuestionsData Class

```
public class QuestionsData : TableEntity
{
    public QuestionsData(string p, string d)
    {
        PartitionKey = p;
        RowKey = d;
    }

    public QuestionsData()
    {
    }

    public string Question { get; set; }
}
```

We'll also need to include Code Listing 39's **using** statement on our **SimpleAnsweringMachine** class, as we'll be using **List<string>** in order to get the questions from the Questions table.

Code Listing 39: Using System.Collections.Generic

```
using System.Collections.Generic;
```

And, we'll need to include Code Listing 40's constants to our **SimpleAnsweringMachine** class.

Code Listing 40: Additional Constants for Our SimpleAnsweringMachine Class

```
private const string cStrSurveyResponses = "ResponsesSuccinctly";
private const string cStrSurveyQuestions = "Questions";
private const string cStrSurveyAnswers = "PredefinedAnswers";
```

Now, let's implement the **GetQuestions** method within our **SimpleAnsweringMachine** class.

Code Listing 41: The GetQuestions Method

```
public string[] GetQuestions()
{
    List<string> q = new List<string>();

    try
    {
        var storageAccount = CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
```

```

        tableClient.GetTableReference(cStrSurveyQuestions);

        TableQuery<QuestionsData> query = new
            TableQuery<QuestionsData>().Where(
                TableQuery.GenerateFilterCondition("PartitionKey",
                    QueryComparisons.NotEqual, "-1"));

        foreach (QuestionsData en in table.ExecuteQuery(query))
        {
            q.Add(en.Question);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return q.ToArray();
}

```

Let's examine this logic quickly.

Because we want to retrieve all questions stored on the Questions table, we simply query all records that have a **PartitionKey** different than -1, which means all records will be returned (because we didn't enter any record on the Questions table for a **PartitionKey** equal to -1, when we created it with Azure Storage Explorer).

Just as with Questions, we'll also need to define a class for PredefinedAnswers and another for Responses Succinctly. We'll name these classes **PredefinedAnswersData** and **ResponseData**, respectively. Let's do this now in Code Listing 42.

Code Listing 42: The PredefinedAnswersData and ResponseData Classes

```

public class PredefinedAnswersData : TableEntity
{
    public PredefinedAnswersData(string p, string d)
    {
        PartitionKey = p;
        RowKey = d;
    }

    public PredefinedAnswersData()
    {
    }

    public string Answer { get; set; }
}

```

```

public class ResponseData : TableEntity
{
    public ResponseData(string p, string d)
    {
        PartitionKey = p;
        RowKey = d;
    }

    public ResponseData()
    {
    }

    public string Response { get; set; }
}

```

Next, we implement a method within our **SimpleAnsweringMachine** class to retrieve the records from the PredefinedAnswers table, which we will call **GetPredefinedAnswers**.

Code Listing 43: The GetPredefinedAnswers Method

```

public string[] GetPredefinedAnswers(string qn)
{
    List<string> pa = new List<string>();

    try
    {
        var storageAccount = CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrSurveyAnswers);

        TableQuery<PredefinedAnswersData> query = new
            TableQuery<PredefinedAnswersData>().Where(
                TableQuery.GenerateFilterCondition("RowKey",
                    QueryComparisons.Equal, "q" + qn));

        foreach (PredefinedAnswersData en in table.ExecuteQuery(query))
        {
            pa.Add(en.PartitionKey + "|" + en.Answer);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

```

```

        return pa.ToArray();
    }

```

This method queries the PredefinedAnswers table by having the **RowKey** look for records that match the question being asked to the caller. Those answers are then returned as a **string** array to be further processed.

Now, let's create a method that will populate the ResponsesSuccinctly table. We'll also add this method to our **SimpleAnsweringMachine** class in Code Listing 44.

Code Listing 44: The InsertSurveyResponse Method

```

public void InsertSurveyResponse(string qId, string uId, string aId)
{
    try
    {
        var storageAccount = CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrSurveyResponses);

        TableOperation retrieveOperation =
            TableOperation.Retrieve<ResponseData>(qId, uId);

        TableResult retrievedResult = table.Execute(retrieveOperation);
        ResponseData updateEntity = (ResponseData)retrievedResult.Result;

        if (updateEntity == null)
        {
            updateEntity = new ResponseData();
            updateEntity.PartitionKey = qId;
            updateEntity.RowKey = uId;
            updateEntity.Response = aId;
        }

        TableOperation insertOrReplaceOperation =
            TableOperation.InsertOrReplace(updateEntity);
        table.Execute(insertOrReplaceOperation);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

```

Looking at this method, we can see that the **ResponseSuccinctly** table is queried with the question ID (**qId**) as the **PartitionKey** and the user ID (**uId**) as the **RowKey**.

If no record is found (**updateEntity** is **null**), values are assigned to the properties of **updateEntity** and this object is inserted into the table by invoking the **Execute** method.

Let's now look at the full updated code for our **SimpleAnsweringMachine** class and .cs file.

Code Listing 45: The Updated SimpleAnsweringMachine.cs File

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using System.Collections.Generic;

namespace TwilioSuccinctly
{
    public class SimpleAnsweringMachineData : TableEntity
    {
        public SimpleAnsweringMachineData(string p, string d)
        {
            PartitionKey = p;
            RowKey = d;
        }

        public SimpleAnsweringMachineData()
        {
        }
    }

    public class QuestionsData : TableEntity
    {
        public QuestionsData(string p, string d)
        {
            PartitionKey = p;
            RowKey = d;
        }

        public QuestionsData()
        {
        }

        public string Question { get; set; }
    }

    public class PredefinedAnswersData : TableEntity
    {
        public PredefinedAnswersData(string p, string d)
        {

```



```

        PartitionKey = p;
        RowKey = d;
    }

    public PredefinedAnswersData()
    {
    }

    public string Answer { get; set; }
}

public class ResponseData : TableEntity
{
    public ResponseData(string p, string d)
    {
        PartitionKey = p;
        RowKey = d;
    }

    public ResponseData()
    {
    }

    public string Response { get; set; }
}

public class SimpleAnsweringMachine : IDisposable
{
    #region "private declarations"
    protected bool disposed;

    private const string cStrConnStr =
        "<< Your Azure Storage Connection String >>";
    private const string cStrAppTable = "SAMSuccinctly";
    private const string cStrSurveyResponses = "ResponsesSuccinctly";
    private const string cStrSurveyQuestions = "Questions";
    private const string cStrSurveyAnswers = "PredefinedAnswers";

    public SimpleAnsweringMachine()
    {
    }

    ~SimpleAnsweringMachine()
    {
        Dispose(false);
    }

    public virtual void Dispose(bool disposing)

```

```

{
    if (!disposed)
    {
        if (disposing)
        {
        }
    }

    disposed = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

public string Get(string ac)
{
    string res = string.Empty;

    try
    {
        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrAppTable);

        TableQuery<SimpleAnsweringMachineData> query = new
            TableQuery<SimpleAnsweringMachineData>().Where(
            TableQuery.GenerateFilterCondition("PartitionKey",
            QueryComparisons.Equal, ac));

        foreach (SimpleAnsweringMachineData en in
            table.ExecuteQuery(query))
        {
            res = en.RowKey;
            break;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return (res == string.Empty) ? "Unknown" : res;
}

```

```

    }

    public string[] GetQuestions()
    {
        List<string> q = new List<string>();

        try
        {
            var storageAccount =
                CloudStorageAccount.Parse(cStrConnStr);
            CloudTableClient tableClient =
                storageAccount.CreateCloudTableClient();
            CloudTable table =
                tableClient.GetTableReference(cStrSurveyQuestions);

            TableQuery<QuestionsData> query = new
                TableQuery<QuestionsData>().Where(
                    TableQuery.GenerateFilterCondition("PartitionKey",
                        QueryComparisons.NotEqual, "-1"));

            foreach (QuestionsData en in table.ExecuteQuery(query))
            {
                q.Add(en.Question);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }

        return q.ToArray();
    }

    public string[] GetPredefinedAnswers(string qn)
    {
        List<string> pa = new List<string>();

        try
        {
            var storageAccount =
                CloudStorageAccount.Parse(cStrConnStr);
            CloudTableClient tableClient =
                storageAccount.CreateCloudTableClient();
            CloudTable table =
                tableClient.GetTableReference(cStrSurveyAnswers);

            TableQuery<PredefinedAnswersData> query = new
                TableQuery<PredefinedAnswersData>().Where(

```

```

        TableQuery.GenerateFilterCondition("RowKey",
            QueryComparisons.Equal, "q" + qn));

        foreach (PredefinedAnswersData en in
            table.ExecuteQuery(query))
        {
            pa.Add(en.PartitionKey + "|" + en.Answer);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return pa.ToArray();
}

public void InsertSurveyResponse(string qId, string uId,
    string aId)
{
    try
    {
        var storageAccount =
            CloudStorageAccount.Parse(cStrConnStr);
        CloudTableClient tableClient =
            storageAccount.CreateCloudTableClient();
        CloudTable table =
            tableClient.GetTableReference(cStrSurveyResponses);

        TableOperation retrieveOperation =
            TableOperation.Retrieve<ResponseData>(qId, uId);

        TableResult retrievedResult =
            table.Execute(retrieveOperation);
        ResponseData updateEntity =
            (ResponseData)retrievedResult.Result;

        if (updateEntity == null)
        {
            updateEntity = new ResponseData();
            updateEntity.PartitionKey = qId;
            updateEntity.RowKey = uId;
            updateEntity.Response = aId;
        }

        TableOperation insertOrReplaceOperation =
            TableOperation.InsertOrReplace(updateEntity);
    }
}

```

```

        table.Execute(insertOrReplaceOperation);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

public string InsertSpace(string s)
{
    string res = string.Empty;

    int count = 1;
    foreach (char c in s)
    {
        if (char.IsDigit(c) && count < s.Length)
            res += c + " ";
        else
            res += c;

        count++;
    }

    return res;
}
} // end class SimpleAnsweringMachine
} // end ns TwilioSuccinctly

```

With our up-to-date SimpleAnsweringMachine logic, let's move our attention to our **AnsweringMachineMain** NancyFX module so that we can write the logic for our /survey/connect REST endpoint.

The survey app should fetch the list of all the questions, loop through each one, present the caller with the predefined answers available, then store each response.

In order to keep the state on our **AnsweringMachineMain** NancyFX module and know which question callers will be asked, we'll need to create some protected **AnsweringMachineMain** class-level properties.

We need to maintain state because the /survey/connect endpoint invokes itself for each new question and answer, thus we need to know what has been answered.

Code Listing 46: Protected Class-Level Properties to Maintain the AnsweringMachineMain State

```

// The state member variables are declared using protected modifier just
// in class you later inherit from the class.
protected static string[] questions = null;
protected static int currentQuestion = 0;

```

```
protected static string userId = string.Empty;
```

The **questions** **string** array holds the name of the questions that the caller will be requested to answer.

The **currentQuestion** **int** variable holds the state for the question currently being prompted for the user to answer.

Finally, the **userId** **string** indicates the value of the Social Security Number (unique identifier) that will be used to identify the answers provided by the caller for each question.

For the sake of simplicity, and in order to avoid the unique identifier being mistaken for an answer to a question, the only requirement for the Social Security Number in our demo is that the user enters any value that contains at least two characters.

As you'll remember, we earlier addressed the need to accommodate the `/survey/incoming` endpoint in order to initialize state. We'll modify it in Code Listing 47.

Code Listing 47: The `/survey/incoming` REST Endpoint with State Initialization

```
Get["@survey/incoming"] = _ =>
{
    userId = string.Empty;
    currentQuestion = 0;
    questions = null;

    string res = "<Response>" +
        "<Gather action=\"/survey/connect\" method=\"GET\">" +
        "<Say voice=\"woman\">You are about to join the UFO survey.</Say>" +
        "<Say voice=\"woman\">What is your Social Security Number?</Say>" +
        "</Gather>" +
        "<Say voice=\"woman\">We didn't get any response. Bye.</Say>" +
        "</Response>";

    return Response.AsText(res, "text/xml");
};
```

With our state initialization in place, we can create the `/survey/connect` endpoint logic, which is where all the magic will happen.

Code Listing 48: The `/survey/connect` REST Endpoint

```
Get["@survey/connect"] = _ =>
{
    string res = "<Response><Say voice=\"woman\">No valid option entered.</Say></Response>";
}
```

```

string val = Request.Query["Digits"].Value;

if (val != null && val != string.Empty)
{
    using (SimpleAnsweringMachine sam = new SimpleAnsweringMachine())
    {
        if (userId == string.Empty)
        {
            if (val.Length > 1)
            {
                userId = val;

                if (questions == null)
                    questions = sam.GetQuestions();

                res = ProcessQuestion(sam, string.Empty);
            }
            else
            {
                res = "<Response><Say voice=\"woman\">No valid  
option entered.</Say></Response>";
            }
        }
        else
        {
            string[] pa =
                sam.GetPredefinedAnswers(currentQuestion.ToString());

            foreach (string p in pa)
            {
                string[] parts = p.Split('|');

                if (parts[0].ToLower().Contains(val.ToLower()))
                {
                    sam.InsertSurveyResponse("q" +
                        currentQuestion.ToString(), userId, parts[0]);

                    string nxtQ =
                        (currentQuestion < questions.Length) ?
                        "<Say voice=\"woman\">Please hold for the next  
question...</Say>" :
                        "<Say voice=\"woman\">The survey is complete.  
Thank you.</Say>";

                    if (currentQuestion < questions.Length)
                    {
                        res = "<Say voice=\"woman\">Answer... " +
                            parts[0][1] + " saved for question... " +
                            currentQuestion.ToString() + "</Say>" +
                            nxtQ;
                    }
                }
            }
        }
    }
}

```

```

        res = ProcessQuestion(sam, res);
        break;
    }
    else
    {
        res = "<Response>" + nxtQ + "</Response>";
        break;
    }
}
else
{
    res = "<Response><Say voice=\"woman\">No valid
        answer entered.</Say></Response>";
    break;
}
}
}
}
}

return Response.AsText(res, "text/xml");
};

```

Let's examine this in order to understand what is going on.

Because we have initialized state by setting the **questions** **string** array to **null**, the **currentQuestion** **int** value to 0, and the **userId** variable to an empty **string**, the first check we perform is to verify if indeed **userId** is empty. Then we check if the variable **val** contains a value with at least two characters (which, at this stage, represents the Social Security Number entered on the /survey/incoming endpoint).

We want to know if the variable **val** is at least two chars because we want to differentiate it from any other response, which will be one character long and will represent an answer to a question.

When we are sure that **val** is indeed the Social Security Number we have requested the caller to enter, we assign the state of the **userId** variable to the value of the **val** variable.

If the **questions** **string** array is **null**, we fetch the list of questions available from the Questions Azure table by calling the **GetQuestions** method from the **SimpleAnsweringMachine** instance.

With the list of questions available, we next request the app to process the first question by calling the **ProcessQuestion** method. The **ProcessQuestion** method is a static method of the **AnsweringMachineMain** NancyFX module that we will explore shortly.

The purpose of the **ProcessQuestion** method is to get Twilio to ask the question and describe the possible answers. However, Twilio doesn't process the response to that question. The actual response to the question is processed when the **ProcessQuestion** method invokes the **/survey/connect** endpoint.

Essentially, **/survey/connect** is self-repeating, in the sense that in order to ask multiple questions and get an answer for each, the **ProcessQuestion** method triggers a call to the same endpoint that executes it. This is why maintaining state is important.

When the first question has been asked, the next time **/survey/incoming** is executed (when the answer to the question is processed), the control flow passes to the principal **else** statement, which makes the code fetch all the possible predefined answers by calling the **GetPredefinedAnswers** method from the **SimpleAnsweringMachine** instance.

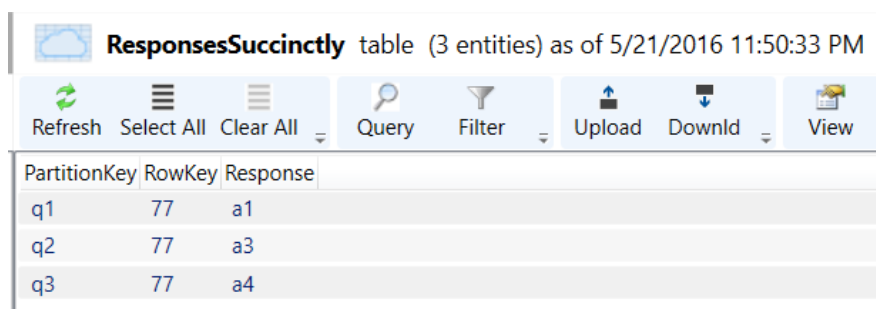
The answer received from the caller, contained within the **val** variable, is checked in order to make sure it exists within the list of predefined answers retrieved from the PredefinedAnswers Azure table. It does this by calling the **GetPredefinedAnswers** method.

If the answer exists within the list of predefined answers for the question being asked (the current question being asked is known because the state is kept by the **currentQuestion** property), the answer is stored along with the question number and **userId** on the ResponsesSuccinctly table by calling the **InsertSurveyResponse** method.

If an incorrect answer to a question is provided, one that is not within the list of predefined answers for the question being asked, the survey is terminated. This feature can be improved upon in the future, but for now works well enough.

When all the questions have been asked and the answers gathered and processed, the information can be viewed on the ResponsesSuccinctly table on Azure.

Figure 55 shows what the results of a completed automatic survey look like (after publishing the app with Visual Studio and calling our Twilio number).



PartitionKey	RowKey	Response
q1	77	a1
q2	77	a3
q3	77	a4

Figure 55: The ResponsesSuccinctly Table after Running Our Survey

Let's explore what the **ProcessQuestion** method does.

Code Listing 49: The ProcessQuestion Method

```
protected static string ProcessQuestion(SimpleAnsweringMachine sam,
string r)
{
    currentQuestion++;

    string res = "<Response>" +
        "<Gather action=\"/survey/connect\" method=\"GET\">" + r +
        "<Say voice=\"woman\">Question " + currentQuestion + " .</Say>" +
        "<Say voice=\"woman\">" + questions[currentQuestion - 1] + "</Say>" +
        GetPossibleAnswers(sam, currentQuestion) +
        "</Gather>" +
        "<Say voice=\"woman\">We didn't get any response. Bye.</Say>" +
        "</Response>";

    return res;
}
```

As we've seen, the **ProcessQuestion** method invokes the /survey/connect endpoint in order to process the answer to the question being asked.

Notice that this method internally calls the **GetPossibleAnswers** method that simply returns the TwiML XML markup code with the predefined answers for the question being asked.

Let's have a look at the **GetPossibleAnswers** method.

Code Listing 50: The GetPossibleAnswers Method

```
protected static string GetPossibleAnswers(SimpleAnsweringMachine sam,
int q)
{
    string res = string.Empty;
    string[] pAnswers = sam.GetPredefinedAnswers(q.ToString());

    if (pAnswers.Length > 0)
    {
        string tmp = string.Empty;

        foreach(string pa in pAnswers)
        {
            string[] parts = pa.Split('|');
            tmp += "<Say voice=\"woman\">Enter... " + parts[0][1] +
                " for... " + parts[1] + "</Say>";
        }

        res = "<Say voice=\"woman\">Please choose from one of the
            possible answers.</Say>" +
    }
```

```

        "<Pause length=\"1\"/>" + tmp;
    }
    else
        res =
            "<Say voice=\"woman\">No predefined answers found...</Say>";

    return res;
}

```

The **GetPossibleAnswers** method gets the list of predefined answers for the current question being asked and returns the TwiML markup to the calling **ProcessQuestion** method in order to give the correct instructions to the caller, which allows the user to answer the question.

This ends our survey application. It's been a lot of fun, even if it was a bit technically challenging due to the self-repeating nature of the `/survey/connect` endpoint.

Summary

We've reached the end of this chapter and this e-book. I hope it's been a wonderful journey of exploring SMS, 2FA, and some of the voice features of the amazing Twilio platform.

The examples I've presented are intended to give you a good glimpse of what is possible with Twilio and should encourage you to further explore other possibilities using the platform.

One of the great things about Twilio is its support of the most common and important programming languages that exist today. The examples written in C# could be easily converted to any other language, as the Twilio helper libraries between the languages are very similar (with minor differences), which makes Twilio both platform and language independent.

Thank you for your interest in this e-book. I hope it's only a starting point for further exploration into the world of SMS and voice-enabled applications with Twilio.

Cheers!