Twilio makes it simple to integrate telephony — both phone calls as well as SMS and MMS messages — into your code without expensive hardware or manual setup.

This is an in-depth guide to working with the Twilio platform from start to finish, making it easy for any developer to integrate phone calls and SMS messages into their code.

Packed with lots of code examples, this book gets you up and running with Twilio in no time, enabling you to work with messages and calls in a variety of different ways. You'll not only learn how to build basic applications using Twilio, but also how to exploit Twilio's most powerful features, keep your Twilio integration secure, and test and debug the application thoroughly.

This book is the perfect guide from your first steps working with Twilio right up to becoming an expert, giving you all the best practices and top tips you need to build reliable and powerful telephony applications.

Who this book is written for

If you have experience with at least one programming language and are looking to integrate Twilio into your applications, then this book is for you.

What you will learn from this book

- Develop applications that make and receive phone calls and messages
- Access your call and SMS records and much more with the Twilio REST API
- Replace physical phone hardware with calls in your browser or mobile app
- Build powerful applications such as phone menus, call centers, and conferencing services
- Work with Twilio in any programming language
- Secure your application
- Debug and test your Twilio integration thoroughly

Learn how to build powerful real-time voice and SMS applications with Twilio

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Working with TwiML"
- A synopsis of the book’s content
- More information on Twilio Best Practices

About the Author

Tim Rogers is a software engineer and student at the London School of Economics (LSE) and is from London, UK. He currently works at GoCardless, which is a payments start-up that helps businesses accept Direct Debit payments online. Here, he built the company's call center in the cloud, which is documented in a series of popular blog posts.

He also works for a number of freelance clients, helping them use the power of Twilio to do things that range from getting reviews from hotel guests to building scalable customer support operations.

In his spare time, he enjoys drinking coffee and serving in his local church and his university's Christian Union.

I would like to thank everyone at GoCardless (especially Grey Baker), all members of LSESU Christian Union, and Konnaire Scannell for their support.
Twilio Best Practices

The Twilio platform makes it simple to integrate telephony—both calls as well as SMS and MMS messages—into your code without expensive hardware requiring a manual setup. However, getting started on such a powerful platform can be daunting.

This book will teach you how to build powerful real-time applications on the Twilio platform from start to finish, making use of phone calls and SMS messages. You'll be introduced to TwiML, the REST API, and Twilio Client before you start building two real-life applications and diving deeper into issues such as testing and security.

What This Book Covers

Chapter 1, *Working with TwiML*, covers TwiML, which is Twilio's XML-based language, that directs how Twilio handles incoming calls and SMSes and places outgoing calls. The actions in TwiML are called verbs, and we'll look at all of them and how they're used.

Chapter 2, *Exploring the REST API*, takes you through interacting with the data in your Twilio account, as well as placing calls and sending SMS messages. You'll learn how to set up the PHP API library, how to interact directly with the API using a tool called Postman, and how to use the API's most useful features.

Chapter 3, *Calling in the Browser with Twilio Client*, introduces Twilio Client, which is a way of making and receiving phone calls directly within your applications, in a browser or mobile app, without using a physical phone. You'll discover how this works at a high level and how to practically implement it in your code.

Chapter 4, *Twilio in the Real World*, takes you through building two real Twilio applications: a callback request tool for your website and a simple conference calling service with fully explained code samples.

Chapter 5, *Twilio in your language*, goes beyond the PHP code we've used in most of the book, showing how to download and configure Twilio's API libraries for Ruby, Python, C#, Java, Node, and Salesforce.com.
Chapter 6, Securing your Twilio App, helps you keep your application and Twilio credit secure with two-factor authentication on your Twilio account. We will verify that the requests you're receiving really come from Twilio and will set up a circuit breaker to detect bugs and account misuse.

Chapter 7, Testing, Debugging, and Deploying Twilio Apps, explains how to test your apps before you release them into the wild and then how to monitor and maintain them later using the App Monitor and Request Inspector.

Chapter 8, Online Resources, contains some helpful links to help you get the most from this book and follow Twilio's best practices.
TwiML is a Twilio-specific XML-based language that is used within a Twilio application to describe what Twilio should do when an incoming call or message hits one of your Twilio phone numbers. TwiML is our way of asking Twilio to do things. Therefore, it's fitting that the XML elements we use are called verbs. For example, we have verbs such as Play, Dial, Record, and Gather, and they are accompanied by nouns such as Number, Client, and Conference.

In this chapter, you will learn the following topics:

- What TwiML is
- How TwiML fits into your Twilio application
- How to set up an inbound phone number
- The data you get from Twilio for inbound calls
- How to use all of the TwiML verbs including <Play>, <Gather>, and <Dial>
- Best practices and tips for working with TwiML

**Where in my application will I be using TwiML?**

In our Twilio account, we can buy phone numbers which are attached to to a particular URL. Twilio will make a GET or POST request to that URL, in order to fetch TwiML when there's an inbound call or message. The TwiML we return tells Twilio what to do in response to the call or Short Message Service (SMS).

When something happens with our phone number—namely an inbound call or incoming SMS—Twilio sends us a webhook that tells us about what's going on and allows us to direct what happens.
We'll also use TwiML when we place outbound calls using Twilio's REST API. To make a call, we have to specify the URL for a piece of TwiML that will handle that call. Once the dialed party picks up, Twilio will go through that markup, perhaps playing a message or recording its input. As part of its webhooks, Twilio provides information such as the number called and the phone number of the caller in the headers. This means you can customize what you ask Twilio to do based on a variety of different data points related to the call.

For instance, by constructing the right TwiML, you can ask Twilio's text-to-speech engine to read some text, play an MP3, dial a call, and many other things.

You can serve up your TwiML from whichever framework, language, and web server you're using. In my examples, I'll use PHP, but it's equally possible to work in the same way in Ruby (on Rails) or any other language.

PHP makes building powerful and dynamic TwiML very simple, as we can easily change the outputted XML using familiar control structures, such as `if` and `else`, which are embedded right on our page.

If you're using a PHP framework, such as Laravel or Cake, or a similar one in other languages, such as Ruby's Ruby on Rails, you'll be able to build XML templates using your usual templating library just as you would for HTML pages.

---

**Getting started with TwiML**

To set the URLs that Twilio will webhook for incoming calls and SMSes, log in to your Twilio account and choose **Numbers** from the navigation bar on top of the screen, as shown in the following screenshot:

![Dashboard](dashboard_image)

If you haven't already, you'll want to buy a phone number. Twilio makes this really easy. You just click on **Buy a number**, which is on the right-hand side, choose your country, and then pick a number of your choice.

Most numbers cost just $1 per month, so cost isn't a huge barrier. Many countries' numbers will support both calls and SMSes, but this is not always the case. Twilio will always tell you what capabilities are supported as part of the buying process.
Once you've got your number, head back to the Numbers screen and click on the one you've just bought.

You'll see that this screen is split into two key sections: Voice and Messaging. You can set separate URLs and HTTP methods for each section. If you're working in PHP, you can usually safely use either GET or POST, but some frameworks and languages will have more specific requirements.

If you click on the optional settings using the link on the right-hand side, you will see a few advanced options which we'll cover. We'll do the same with the powerful Configure with Application setting.

Let's write two quick hello world TwiML snippets, in keeping with programming tradition, using PHP. Start by creating a file called call.php as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Hello world. We love you guys in
  <?php echo $_GET['FromCountry']; ?>.</Say>
</Response>
```
In the preceding sample, you’ll see that this PHP responds with some XML. XML as a language is very similar to HTML, so it’ll look familiar. If you haven’t encountered it previously, don’t worry; you’ll get the hang of it over the course of this chapter.

Inside the `<Response>` block where Twilio looks to find what it should do in response to the incoming call, we use the `<Say>` verb. The text we put within the `<Say>` element is what Twilio’s text-to-speech engine will speak.

We’re already taking advantage of PHP here by looking at some GET data that Twilio provides with the request. In this case, the voice is going to say the name of the country where the caller is located—`FromCountry`. There are lots of other great things you can do, which we’ll cover later.

After the `<Say>` verb, Twilio will hang up, as it has nothing more to do.

We’ve now written a handler for incoming calls, so let’s also write one for SMSes. We can do something very similar indeed; let’s save this as `message.php`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Message>Hello world. We love you guys in
  <?php echo $_GET['FromCountry']; ?>.</Message>
</Response>
```

As you’ll see, we do exactly the same thing here except for using the `<Message>` verb instead of the `<Say>` verb. This means that we’ll text the sender with the message rather than saying it to them over the phone. We’ll cover the `<Message>` verb in more detail later.

You’ll now need to upload these PHP files somewhere where they can be accessed by Twilio. You’ll probably have some hosting set up, but if you do not, there are a range of great options. I’ve included a few recommendations in Chapter 8, Online Resources.

Alternatively, you can use a local server; see my tip on using a great tool called ngrok at the end of the chapter for help with this.

Now that we’ve set up those PHP files, add the URLs of your `call.php` and `message.php` files to your Twilio number, and then hit Save.

Let’s see the magic happen. Try calling and SMSing your number. First, Twilio webhooks our TwiML URL, letting our code know about the call and asking it what to do. We respond with TwiML such that Twilio speaks out loud our "hello world" message if we’re calling in, or SMSes it to us if we’ve sent in a text. You’ve now seen the power of TwiML.
Digging deeper – Twilio's requests

In the preceding example, you've seen that Twilio includes some helpful data in its request when it hits your server to fetch the TwiML it needs in order to handle an incoming call or message.

We pulled out the country where the caller is located, which is stored in the FromCountry parameter. We grabbed this in PHP using $_GET, but you can do the same in any web language.

Alongside the caller's location, Twilio includes a whole lot of useful information.

Here are the highlights:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallSid/MessageSid</td>
<td>This refers to the unique reference for this call or message, as appropriate. This can be particularly useful to store in a database if you might want to look up this call using the REST API later.</td>
</tr>
<tr>
<td>From</td>
<td>This refers to the caller's phone number in the international format (for example, +, then the country code, and then the local number).</td>
</tr>
<tr>
<td>To</td>
<td>This tells you which of your Twilio phone numbers is being dialed in the international format, as described previously.</td>
</tr>
<tr>
<td>CallStatus</td>
<td>This tells you about the current status of the call, which may be queued, ringing, in-progress, completed, busy, failed or no-answer.</td>
</tr>
<tr>
<td>ApiVersion</td>
<td>This refers to the Twilio API version being used for this call; you can safely ignore this parameter.</td>
</tr>
<tr>
<td>Direction</td>
<td>This refers to the direction—or in some sense, the type—of call in progress. This will be inbound for incoming calls, outbound-api for calls initiated with the REST API (we'll learn more about this in Chapter 2, Exploring REST API) and outbound-dial for calls initiated from the TwiML &lt;Dial&gt; verb.</td>
</tr>
<tr>
<td>ForwardedFrom</td>
<td>For some forwarded calls, this will include the number from which the call was forwarded, but this is not supported by all carriers.</td>
</tr>
</tbody>
</table>
### Working with TwiML

<table>
<thead>
<tr>
<th>Parameter</th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallerName</td>
<td>Twilio allows you to enable caller ID lookup on your phone numbers for $0.01 per lookup. If you've enabled this, this will contain the caller's name if a result was found.</td>
</tr>
<tr>
<td>From/To</td>
<td>City refers to the city where the caller/number being called is located (this will not necessarily be provided).</td>
</tr>
<tr>
<td></td>
<td>State refers to the state, province, or country where the caller/number being called is located (this will not necessarily be provided).</td>
</tr>
<tr>
<td></td>
<td>Zip refers to the zip or postal code of the caller/number being called (this will not necessarily be provided).</td>
</tr>
<tr>
<td></td>
<td>Country refers to the country where the caller/number being called is located.</td>
</tr>
<tr>
<td>Body</td>
<td>This refers to the text received in the SMS (SMS only).</td>
</tr>
</tbody>
</table>

This data, which Twilio provides, can help you implement a wide range of dynamic features into your TwiML, such as the following:

- Changing how the call is handled depending on the number being called
- Switching languages based on the location of the caller
- Responding to what someone actually said in an SMS

Further details are available in Twilio's comprehensive documentation at https://www.twilio.com/docs/api/twiml/twilio_request

### The world of TwiML verbs

Verbs are the belt and braces of your TwiML. They're the part that actually tells Twilio what it should do when a call comes in or a text arrives.

I'll seek to give you an introduction to what you'll be looking at while using each verb. For more specific details, you'll want to directly refer to the documentation.
Chapter 1

Twilio's documentation is pretty exhaustive and is available online at https://www.twilio.com/docs/api/twiml/

<Say>
The <Say> verb invokes Twilio's text-to-speech engine, that is, it gets Twilio to say things:

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Twilio Best Practices is number one!</Say>
</Response>

Inside the <Say> tag, you provide the text that you want it to speak. In Twilio parlance, what's nested inside the verb is called the noun. In this case, it's just plain text, but for many verbs, it'll be further XML tags.

By setting attributes on the <Say> verb, we can switch the voice from male to female and can also change the language of our text. The full options are in Twilio's documentation, but it works along these lines:

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="alice" language="fr-FR">J'adore ce livre - c'est le meilleur livre que j'ai jamais lu!</Say>
</Response>

In our <Say> verb, we can specify a range of attributes that customize what happens:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>What it means</th>
</tr>
</thead>
<tbody>
<tr>
<td>voice</td>
<td>This sets the voice we want to use while speaking our text; this can be set to man, woman, or alice. The voice you choose affects the options available for the language attribute.</td>
</tr>
<tr>
<td>loop</td>
<td>How many times should our message be read out? If this is not set, the message will not be repeated. Set it to 0 to make it loop forever, or set a specific number of loops.</td>
</tr>
<tr>
<td>language</td>
<td>This sets the language that your message is in. For the man and woman voice, American English, British English, Spanish, French, German, and Italian are available, but the alice voice supports many more languages. It defaults to en (that is, American English). For more details, see <a href="https://www.twilio.com/docs/api/twiml/say#attributes-manwoman">https://www.twilio.com/docs/api/twiml/say#attributes-manwoman</a></td>
</tr>
</tbody>
</table>
There are a few quirks that are worth noting with the `<Say>` verb. Take a look:

- As we can construct our TwiML with PHP, it's easy to dynamically generate the text that is spoken, as we did in `call.php` earlier in the chapter.
- Always test your `<Say>` verbs well by calling in yourself. Twilio might not always pronounce things perfectly, and you should be especially careful to check the annunciation of numbers, dates, and amounts of money:
  - A great example is that if we include 1234—for instance, as a PIN number or password, Twilio will say *one thousand two hundred and thirty four*, rather than *one two three four*. If we wanted it to say the latter, we should write 1 2 3 4, with a space between each number, perhaps also using a `<Pause>` verb between numbers to keep it from being read too fast.
  - With proper nouns, such as place names or products, you might need to be creative in order to have proper pronunciation. One way to do this is to spell things phonetically.

### `<Play>`

The `<Play>` verb lets you play audio. This is useful for things such as holding music and using your own voiceovers where text-to-speech just seems a little too awkward:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Play>https://s3.amazonaws.com/twilio-best-practices/hello.mp3</Play>
</Response>
```

Inside the `<Play>` tag, provide the URL of the audio file to be played. Twilio supports a number of formats, but you'll almost certainly want to use either MP3 or WAV.

You can also use the `<Play>` verb to play DTMF tones (that is, the sound made when you press a number on your phone's keypad) to test other phone systems. We won't cover this, as it's very much an edge use case.

As with the `<Say>` verb, the `loop` attribute is supported; it works exactly the same as for `<Say>`, allowing us to repeat our audio clip as many times as we need or forever.

The most important caveat to remember with `<Play>` is that Twilio will cache the audio file you provide. This means that changing voiceovers or hold music is not necessarily as simple as just changing the file where they're stored.
At the same time, Twilio's caching is useful because it will help you save bandwidth and, therefore, cost as well—especially if you’re hosting your audio with a provider such as Amazon S3 (http://aws.amazon.com/s3/)

Twilio will obey the standard HTTP cache headers to decide when to re-download your audio file and when to keep using a copy it has used previously; see https://www.twilio.com/help/faq/voice/how-can-i-change-the-cache-behavior-of-audio-files for details. So, to change audio files, you’ll need to do one of the following:

• Wait for the caching period to finish
• Re-deploy your application, pointing to a fresh URL for the audio (for instance, uploading your audio files into directories named with the date of the application version, and then updating all the references in your TwiML)

<Pause>

The <Pause> verb waits silently for a specified number of seconds, or one second by default. It's simply used like this, with the length of time for waiting specified in the length attribute:

```xml
<Pause length="5" />
```

Note that the <Pause> verb looks a little different than any other verb because it uses a self-closing tag. It has no noun(s), but it takes an attribute that represents the number of seconds for which you wish to wait. As we'll see later, the <Reject> verb works in a very similar way.

<Gather>

The <Gather> verb allows you to take input from a caller by asking them to enter digits on their phone's keypad. This allows you to build complex, interactive applications.

The <Gather> verb is slightly more complicated to use than the verbs we've seen so far, as you will be nesting other verbs inside it.

So, for example, we might nest a <Say> verb inside our <Gather> block to say a message and then wait for the caller's input.

In this example, we say a message and then wait for 10 seconds for the caller to enter a single digit in response:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Gather timeout="10" numDigits="1" action="digits.php">
</Response>
```
<Say>Choose an option from the menu. Press 1 for sales. Press 2 for customer services. Press 3 for billing.</Say>
</Gather>
<Say>You didn't enter an option. Goodbye.</Say>
</Response>

Once a customer has entered a digit, Twilio will make a POST request to the action, which is `digits.php`, including the digits that the customer entered in the `Digits` parameter. This allows you to build awesome interactive applications. Here’s an example of what we can do in `digits.php`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <?php if ($_POST['Digits'] == "1") { ?>
    <Say>Please wait - we'll put you through now.</Say>
    <Dial>…</Dial>
  <?php } else { ?>
    <Say>Our phone lines are currently closed – please drop us an email at support@twiliobestpractices.com.</Say>
  <?php } ?>
</Response>
```

Inside our `<Gather>` block, we can nest not only `<Play>` verbs, but also `<Say>` and `<Pause>`.

Here, we first check whether the digit 1 has been entered. If it has, we ask Twilio to say a message, and then we add another action afterwards. Here, we use a `<Dial>` verb through which we might add the caller to a queue or dial to them through to a particular number.

If a number other than 1 was entered, we play an alternative message.

![Footnote]
When you're using `<Gather>`, always test all of the paths through your call flow. This means that you try every option; otherwise, it's easy to not pick up serious errors with your application.

`<Record>`

Unsurprisingly, the `<Record>` verb lets you record the caller's voice. This is perfect for things such as building a voicemail service, registering participants' names for a conference call, or gathering feedback from users:

```xml
<Record timeout="30" transcribe="true" action="/recording.php" />
```
In the preceding code snippet, we record for up to 30 seconds, ask Twilio to try to transcribe the audio into text, and then Twilio makes a POST request to recording.php with the URL of the recorded audio as an MP3.

Note that using Twilio's transcription feature costs $0.05 cents per minute transcribed.

Twilio will expect recording.php to also return TwiML in order to let it know what to do next. For instance, you might hang up the call or even play back the caller's recording to them for them to check and confirm.

As you're given the recording URL, it's really easy to do all of this and much more, such as storing our recording in a database:

```php
<?php
    $recordingUrl = $_POST['RecordingUrl'];
    $transcriptionText = $_POST['TranscriptionText'];
    $callSid = $_POST['CallSid']; // The unique identifier for this call
    // Save $recordingUrl to a database against $callSid, and then...
?>
<xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Say>Thank you – here’s the message we just recorded:</Say>
    <Play><?php echo $recordingUrl; ?></Play>
    <Say>Thank you, and goodbye.</Say>
    <Hangup />
</Response>
```

In the preceding code snippet, we take some of the data provided by Twilio in the request (that is, in $_POST) and store it to variables in order to use it later. We then use this to form a TwiML response, which plays a message and then plays the caller's recording back to them. It then says goodbye and hangs up.

The <Record> verb doesn't support nesting. If you want to record an entire call, which is probably the primary example where you'll want to do something else, the flow is slightly different and forms part of what we'll do with the <Dial> verb. We'll cover this later.
Working with TwiML

<Message>
The <Message> verb allows you send a text or Multimedia Messaging Service (MMS) message as part of a phone call's flow. Using this verb is simple. On a basic level, you just nest plain text within it, representing the message that should be sent:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Message>Thanks for calling - if you need anything else, just let us know.</Message>
</Response>
```

Downloading the example code
You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Twilio will automagically, if nothing else is specified, send a text to the current caller with the caller ID as the number that is being called.

Of course, you might want to text someone else. For example, imagine that we have a fault-reporting service where we'll text an available engineer when someone reports a problem:

```php
$engineerPhoneNumber = '+441290211999';

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Thanks for letting us know about this fault - someone will call you back very shortly.</Say>
  <Message To="<?php echo $engineerPhoneNumber; ?>">A caller just reported a fault. Please call them back on <?php echo $_POST['From']; ?>.</Message>
</Response>
```

Here, we play a message to the caller and then send a text to a provided phone number (stored in the $engineerPhoneNumber PHP variable) with the caller's phone number.

Most of the time, this won't be necessary, but we can set up a status callback (statusCallback) on the <Message> verb to have our application notified as to whether an SMS was successfully sent. For details, see Twilio's documentation at https://www.twilio.com/docs/api/twiml/sms/message.
Using the `<Message>` verb, we can also send MMS messages with included images. In order to do this, we'll nest a `<Media>` noun with a URL pointing to an image inside our `<Message>` verb. To include an image and text, we can nest a `<Media>` noun and a `<Body>` noun as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Message>
    <Media>https://demo.twilio.com/owl.png</Media>
    <Body>Owls are excellent.</Body>
  </Message>
</Response>
```

At the moment, MMS messaging is only available with select Twilio phone numbers in the US and Canada, but this will expand in due course.

If you're not interested in sending messages as part of a call, don't worry; we'll cover how to send outbound SMSes via the REST API on an ad-hoc basis in the next chapter.

**<Enqueue>**

As part of offering a full suite for building phone services, Twilio supports the queuing functionality that is very popular for use in call centers and similar applications.

Simply nest the name of a queue that the caller should be joined to within the verb:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait and one of our team members will be with you shortly.</Say>
  <Enqueue>Support</Enqueue>
</Response>
```

On the `<Enqueue>` verb, we can specify a `waitUrl` attribute. This should point to a TwiML that will be repeatedly run through for the caller while they wait in the queue.

This will default to play hold music provided by Twilio, but we can add our own, or even read the caller's position in the queue to them when we specify our own custom file. We can set our own waiting TwiML like this:

```xml
<Enqueue waitUrl="waiting.php">Support</Enqueue>
```
Follow this up by writing your own waiting.php file like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
        ClockworkWaltz.mp3</Play>
    <Say>You are currently in position <?php echo
        $_POST['QueuePosition']; ?> in the queue.</Say>
</Response>
```

Here, we play some of Twilio's wait music (you can see a list of their provided tracks
at http://s3.amazonaws.com/com.twilio.sounds.music/index.xml) and then play the caller's position in the queue back to them.

From our waitUrl attribute, most TwiML verbs (except <Dial> verb) are supported. This means that you can do a range of things in the wait process, from playing a message like we did previously to collecting details from the customer with the <Gather> verb.

A call can be dequeued in three ways:

- By another caller being connected to the call through the <Dial> verb's <Queue> noun
- Via the REST API
- With the <Leave> verb

We'll cover these in detail later, but for now, let's take a look at the <Leave> verb.

**<Leave>**

This verb is a very simple one. It is used from a queue's waitUrl (see the preceding section), and it lets us remove the caller from the queue and run some alternative TwiML instead.

As a crude example, let's add a caller to our support queue but add a we're now closed-style message after our <Enqueue> verb:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Say>Please wait and one of our team member will
        be with you shortly.</Say>
    <Enqueue waitUrl="waiting.php">Support</Enqueue>
    <Say>We're now closed - please call back tomorrow.</Say>
    <Hangup />
</Response>
```
Once the caller is joined to the Support queue, the execution of our TwiML document will be stopped and Twilio will loop over the TwiML in waiting.php, waiting for the call to be dequeued instead.

Only if and when the caller leaves this queue will we continue to execute our TwiML so that the <Say> block gets run.

We might want to remove callers from the queue at 6 p.m. when our customer support lines close. Let's write some TwiML in waiting.php with the help of a little PHP. To do this, take a look at the following code:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <?php if (intval(date("H")) >= 18) { ?><Leave /><?php } else { ?>
  <Say>You are currently in position <?php echo $_POST['QueuePosition']; ?> in the queue.</Say>
  <?php } ?>
</Response>
```

Here, we check whether the hour on a 24-hour clock is more than 18 (that is, 6 p.m.). If it is, we leave the queue (so that our final bit of TwiML in the previous snippet gets run), or else, we play some hold music and then announce the caller's current position. waiting.php will simply be requested again and again while a caller queues.

For <Leave>, we can use a self-closing tag because this verb is never used with a noun. We can write <Leave></Leave>, which would be equivalent to <Leave />, but simply writing <Leave /> is quicker.

**<Dial>**

The <Dial> verb is probably the most important and, equally, the most complex of all the TwiML verbs.

It allows us to place outbound calls and bridge them to our current one, enabling tonnes of powerful applications, from connecting inbound calls to customer support staff to setting up conferences.
For example, as part of a call, we might dial in to one of our support staff:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait while we connect you to one of customer support team. Please note that calls will be recorded.</Say>
  <Dial record="record-from-answer" action="recording.php"/>
    <Number>+441290211999</Number>
  </Dial>
</Response>
```

Here, we play a message and then call our customer support phone number, recording the call from the time it's answered and asking Twilio to post the recording to `recording.php`. When our `<Dial>` verb has an action, as is the case here, no TwiML verbs after it will be accessible (that is, used), as Twilio will move on to the action URL.

As always, Twilio has sensible defaults for the `<Dial>` verb, which can be customized. For instance, it'll set a timeout of ringing for 30 seconds before it gives up, and `callerId` will be set to the number of the current caller. You can discover all of the options in Twilio's documentation at https://www.twilio.com/docs/api/twiml/dial.

When you're using `<Dial>`, what you nest within it is very important. You've already seen the use of `<Number>`, which will call the number of a physical (that is, PSTN) phone. There are a number of nouns you can nest under it in order to make different kinds of calls:

### `<Number>`

The `<Number>` noun lets you call a traditional phone number; nest one or more these under your `<Dial>` verb in order to call it.

One of the most interesting features here is that you can actually try multiple phone numbers. For example, imagine a situation, such as in the following code sample, where you want to try multiple numbers for your customer support phone line:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait while we connect you to one of the customer support team. Please note that calls will be recorded.</Say>
  <Dial record="record-from-answer" action="recording.php">
    <Number>+441290211999</Number>
    <Number>+441290211998</Number>
  </Dial>
</Response>
```
With this TwiML, Twilio will attempt to call both of the numbers. As soon as someone picks up, it'll stop trying the other.

The `<Number>` noun also provides some advanced functionality that lets you control what happens when the dialed party has answered the call, such as the `sendDigits` and `url` attributes:

- With `sendDigits`, you can ask Twilio to send some DTMF tones to a called party when they pick up (for example, to reach a particular extension behind that number).
- With `url`, you can specify the URL or path to a piece of TwiML that can be run against the caller before they're connected to the current call.

Let's go through examples of both of those options.

First, we'll look at `url`; we'll start with our TwiML for the actually incoming call, just like we did previously:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait while we connect you to one of customer support team. Please note that calls will be recorded.</Say>
  <Dial record="record-from-answer" action="recording.php">
    <Number url="intro.php">+441290211999</Number>
    <Number url="intro.php">+441290211998</Number>
  </Dial>
  <!—Let's hang up when the dialed call is over --!>
  <Hangup />
</Response>
```

A `intro.php` file will be played to our customer support agents once they've picked up the call but before the call is actually connected through, letting them reject the call if it's inconvenient:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Gather timeout="5" action="digits.php" numDigits="1">
    <Say>This is an incoming customer support call – press any key on your phone in the next 5 seconds to accept the call, or otherwise hang up.</Say>
  </Gather>
  <Hangup />
</Response>
```
We'll play a message to the customer support agent and then wait for their input. If they press a digit at the prompt, we'll move on to the digits.php TwiML file. Otherwise, the call will be rejected and thus hung up, leaving Twilio to keep trying the other number in the <Dial> block.

Lastly, we'll need to create a digits.php file to deal with the called party's input:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Thanks – you'll now be connected.</Say>
</Response>
```

The agent will be played a quick message, and then Twilio will actually connect the dialed party to the original call.

You'll notice that we need to do nothing to make this bridging of the two calls happen; it's just that in this context, Twilio's default behavior does this when there is no more TwiML left.

The `sendDigits` attribute is useful when we want to dial some digits once the called party picks up. This is useful for automating other phone menus and services, or for dialing an extension, as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial record="record-from-answer" action="recording.php">
    <Number sendDigits="www100">+441290211999</Number>
  </Dial>
</Response>
```

Here, we'll dial our number, but when they pick up, we'll wait for 2.5 seconds (each `w` character represents a half-second pause) and then dial 100, our imaginary extension.

**<Sip>**

Apart from dialing through to physical phones, we can also make calls on Twilio over Session Initiation Protocol (SIP). SIP is a standard, or perhaps the standard for Internet telephony, connecting together a range of phone networks.

The `Sip` verb effectively acts as a cheaper complement to making calls over PSTN using the `Number` noun at less than half the cost of calling a US phone number.
We'd dial a SIP URI (which identifies a particular client on a particular SIP server) as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial record="record-from-answer" action="recording.php">
    <Sip>sip:me@timrogers.uk</Sip>
  </Dial>
</Response>
```

The `<Sip>` verb works with all of the various `<Dial>` verb options we've seen previously for calls using the `<Number>` noun. For example, we can ask to record calls or set a timeout from the `<Dial>` verb.

The URL attribute is also available on the `<Sip>` noun and works in exactly the same way as it works for `<Number>`, letting us add call screening and other such features with ease.

We can even combine calls to different kinds of destinations under one `<Dial>` verb. For instance, we can simultaneously try to call a member of the staff's mobile phone and the SIP phone on their desk by nesting `<Sip>` and `<Number>` nouns under a `<Dial>` verb.

There are lots of niche options available to you when you're working with SIP in Twilio that aren't worth covering in this book; examples include, forcing the TCP or UDP transport for the connection and sending custom headers with the SIP request.

You can read about these and other customizations at https://www.twilio.com/docs/api/twiml/sip.

**SIP authentication**

Often, SIP servers will have authentication on them to prevent unwanted calls. This will usually work in one of two ways: username and password or IP whitelisting.

**Username and password protection**

As part of our `<Sip>` noun, we can specify a username and password that Twilio should provide when sending the `INVITE` message to the SIP server. To do this, we simply use the `username` and `password` attributes on the noun as follows:

```xml
<Sip username="twilio" password="twiliorocks">sip:me@timrogers.uk</Sip>
```
Working with IP whitelisting
Perhaps a more common (but harder to deal with) form of authentication is IP whitelisting. This is where you'll set up your SIP server to only accept inbound calls from certain IP ranges.

Fortunately, Twilio provides you with a list of the IP addresses from which the SIP traffic may come. You can find them at the bottom of the page at https://www.twilio.com/docs/sip.

You should revisit this page from time to time, as Twilio expects to add additional IPs to enhance scalability and reliability in future.

<Client>
Twilio's <Client> allows you to include voice capabilities within a web page or native app. This means that people can make and take calls from their own devices without using the legacy telephony networks or complicated SIP setups.

This makes it easy to build powerful telephony solutions, for example, browser-to-browser calling within a web application. However, part of the magic is that it's fully connected to the rest of Twilio's platform.

This means that we can set up Twilio Client in a browser and then take incoming calls to it over a traditional phone number. Twilio Client uses TwiML in exactly the same ways as we've already seen for both incoming and outbound calls.

By way of an example, you can imagine a phone conferencing service that uses this to enhance its functionality. Attendees and presenters on a call will be able to join in either through their browser using a headset, from their mobile phone via a custom app, or from any phone of their choice on a local phone number. All of this can be run through Twilio.

Each individual connected to Twilio with Twilio Client will have their own client identifier. It's unique within the scope of our Twilio account and is what we use to connect calls to a particular user. Connecting to a particular client within the context of a <Dial> verb is very simple indeed:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial record="record-from-answer" action="recording.php">
    <Client>the-client-identifier</Client>
  </Dial>
</Response>
```
If Twilio Client sounds great, you’re in luck. Refer to Chapter 3, Calling in the Browser with Twilio Client.

<Conference>
Using Twilio, we can easily build our own custom phone conferencing tool to rival commercial alternatives.

Doing this is a great option as Twilio’s APIs give you the power to perform all sorts of integrations and customizations. The <Conference> verb is really quite a complex noun as it allows you to wield almost all of the features you’d see in professional conferencing tools from your own code.

The <Conference> noun creates or adds a caller to a conference room of your choice. Simply use the noun and place the name of a room inside it. You don't have to create it ahead of time:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial>
    <Conference>Monday Morning Meeting</Conference>
  </Dial>
</Response>
```

Let’s quickly cover the different options—there are a lot of them—and then dive into an example:

<table>
<thead>
<tr>
<th>Option</th>
<th>How it works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>muted</td>
<td>Set this to true or false to determine whether this caller can speak to others in the conference, or if they can only listen in. This is set to false by default.</td>
</tr>
<tr>
<td>beep</td>
<td>This specifies whether this person being added to the conference should hear a beep when people join or leave the conference room. It defaults to true, but it can be turned off with false or set to happen only onEnter or onExit.</td>
</tr>
<tr>
<td>startConferenceOnEnter</td>
<td>This determines whether the conference starts when this caller enters the call (provided there is another participant in the room as well). This defaults to true, but you might wish to set it to false for some callers if, for instance, you want a conference to only start once the presenter arrives.</td>
</tr>
</tbody>
</table>
### Working with TwiML

<table>
<thead>
<tr>
<th>Option</th>
<th>How it works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>endConferenceOnExit</td>
<td>This sets whether this conference stops once this caller exits. It defaults to false, but if set to true, when this caller leaves, all other participants will be forced out.</td>
</tr>
<tr>
<td>waitUrl</td>
<td>This allows you to specify a TwiML URL that will be looped while the current caller waits for another participant to join the room if they're the first to join. By default, Twilio will play some rather horrible music, but by providing the URL of your own TwiML, you can add your own music or messages.</td>
</tr>
<tr>
<td>waitMethod</td>
<td>This allows you to set the HTTP method with which the waitUrl TwiML will be requested to either GET or POST.</td>
</tr>
<tr>
<td>maxParticipants</td>
<td>This sets the maximum number of participants that will be allowed in this conference, defaulting to 40 (the maximum ever allowed by Twilio).</td>
</tr>
<tr>
<td>record</td>
<td>This specifies whether this conference should be recorded. It defaults to do-not-record, but it can be set to record-from-start in order to make a recording. The recording's URL will be fired over to eventCallbackUrl once the conference is over.</td>
</tr>
<tr>
<td>trim</td>
<td>This specifies whether the conference's recording should have silence at the beginning or end of the conference trimmed off. It defaults to trim-silence, but it can be disabled by setting it to do-not-trim.</td>
</tr>
<tr>
<td>eventCallbackUrl</td>
<td>This is the relative or absolute URL that will receive a POST request when the conference is over, with a RecordingUrl pointing to a recording of the conference if recording is enabled with the record parameter.</td>
</tr>
</tbody>
</table>

Let's build a basic conference where we'll have a passcode for presenters and a passcode for attendees. First, we'll create a TwiML file to handle incoming calls:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Gather timeout="10" action="digits.php" numDigits="6">
    <Say>Enter your six digit pass code to join the conference.<Say>
  </Gather>
  <Hangup />
</Response>
```
This file will play a message and wait for 10 seconds for the caller to enter a six-digit code.

If they fail to enter the code, we'll hang up. In the real world, we'd probably want to do something nicer than this. Otherwise, we'll make a POST request to digits.php with the Digits parameter containing what was entered on the keypad. Let's create the digits.php file:

```php
<?php
$presenterCode = "123456";
$attendeeCode = "654321";
?>
<?xml version="1.0" encoding="UTF-8"?>
<Response>
<?php if ($_POST['Digits'] == $presenterCode) { ?>
    <Say>Thank you, presenter. We'll connect you now.</Say>
    <Dial>
        <Conference endConferenceOnExit="true">Room</Conference>
    </Dial>
<?php } else if ($_POST['Digits'] == $attendeeCode) { ?>
    <Say>Thank you – we'll add you to the conference now. You'll hear hold music until a presenter joins.</Say>
    <Dial>
        <Conference startConferenceOnEnter="false">Room</Conference>
    </Dial>
<?php } else { ?>
    <Say>Sorry, that conference doesn't exist. Goodbye.</Say>
    <Hangup />
<?php } ?>
</Response>
```

Here, we've got a fair bit of logic to go through:

- In the first couple of lines, we set our conference passcodes:
  - In the real world, we'd want to connect this to a database that can handle our different conference code, amongst other things.

- If the caller enters the presenter code, which is 123456, we add them to the conference. By default, the conference will be able to start when they join (as soon as there is another attendee there), but we customize the endConferenceOnExit option so that the conference finishes the moment they leave.
Working with TwiML

- If the caller enters the attendee code, which is 654321, we play a message to them and add them to the conference. However, we customize the startConferenceOnEnter option so that the conference can never start until there is at least one presenter.

- If the caller didn't enter one of the recognized code, we say goodbye and then hang up.

From this, it should be evident that Twilio's conferencing is really quite powerful, especially when you use the various customizable options to deal with things such as recordings, moderation, and the waiting experience.

Twilio conferences only work with a maximum of 40 participants. If you need more callers than this, you'll need to stick with a traditional solution for the time being!

<Queue>
This final noun for <Dial> allows us to pull a call out of a queue to which we've added a call (that is, caller) with the <Enqueue> verb.

As with many of the previous nouns, we can specify a url attribute, pointing to a TwiML that will be played to the queued caller before they're put through to the person being dialed in.

Let's try it for ourselves:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial>
    <Queue url="alert.php">Support</Queue>
  </Dial>
</Response>
```

When this TwiML is executed, the caller will be connected to the next caller in the support queue after the person waiting in the queue has gone through the TwiML in alert.php.

Let's create an example alert.php file now, where we'll tell the person that their call will be recorded:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Thank you for waiting - you're about to be connected to one of our support team members. All calls are recorded.</Say>
</Response>
```
<Hangup>
The <Hangup> verb ends a call. It's just used on its own as a self-closing tag with no nouns or attributes. In the next example, we say goodbye and then hang up:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Goodbye.</Say>
  <Hangup />
</Response>
```

<Redirect>
The <Redirect> verb moves from executing the current TwiML to a different file on a different URL immediately, ignoring the rest of the TwiML in the current file.

Inside the <Redirect> verb, we provide the absolute URL or relative path of the TwiML file to be executed. We can also set the method attribute to GET; it defaults to POST:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Redirect method="GET">../other_twiml.php</Redirect>
</Response>
```

The <Redirect> verb is not only applicable for phone calls. It is the only verb (except <Message> verb) that can be used for phone calls and incoming messages.

Nothing can be nested within the the <Redirect> verb, and any verbs after it are ignored as the redirect takes place right away.

<Reject>
The <Reject> verb, if placed as the very first verb in an incoming call, will prevent the call from being answered and will incur no cost whatsoever.

If placed elsewhere in the call, the call will hang up but we will still be charged up to that point.

The caller will hear an engaged or busy tone that we can customize through the reason attribute. We set the reason to rejected in order to play an engaged tone (which is the default) or set it to busy for a busy tone.
We can use this if we want to screen out certain types of calls. For example, imagine a situation where we're being spammed by a particular number:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <?php if ($_POST['From'] == "+447969123456") { ?>
    <Reject reason="busy" />
  <?php } else { ?>
    <Redirect method="GET">../other_twiml.php</Redirect>
  <?php } ?>
</Response>
```

**Best practices for working with TwiML**

We'll discuss some helpful tips that can be kept in mind when you're working with TwiML to make your application better organized, more easily testable, and more maintainable.

**Test locally using ngrok**

When you're working with and building a Twilio application, you'll need to test it frequently by making and receiving real calls on the platform, which requires Twilio to be able to access your local web server.

This is often a challenge with the perils of dynamic IP addresses and port forwarding. Fortunately, there's a great free tool called ngrok (https://ngrok.com), which will help you get around this problem.

It'll give you a special URL (for example, http://123abc.ngrok.com), which Twilio can use to reach your application. So, you can enter this on your numbers in your Dashboard in order to receive inbound calls and SMS messages.

ngrok is simple to use, but you'll need to be at least a little comfortable with the command line:

2. Start your web server. It can be running on port 80, which is the default HTTP port, but many web servers run on non-standard ports such as 8080 or 3000; both are supported.
3. Unzip ngrok, and then copy it to a location from where it is easily accessible. Ideally, it would be in your shell path, but to start with, you could keep it in your Downloads directory or somewhere else which is easy to access.
Chapter 1

The next steps will differ somewhat, depending on whether you're on Windows or another platform.

**Windows**

1. Open a command prompt from the Start menu, and then use `cd` to navigate to the folder where ngrok is stored; see [http://www.wikihow.com/Change-Directories-in-Command-Prompt](http://www.wikihow.com/Change-Directories-in-Command-Prompt) if you're not sure how to do this.

2. Once you're there, run `ngrok` followed by the port your web server is running on (for example, `ngrok 80` if your server runs on port 80), and then hit Return.

**Mac OS X and Linux (and others!)**

Open a terminal and use `cd` to find the directory where the ngrok executable is stored. For help with navigating, see [http://www.linfo.org/cd.html](http://www.linfo.org/cd.html).

Once you're there, run `./ngrok`, followed by the port your web server is listening on, for instance, `./ngrok 3000` if your web server works on port 3000.

Once you've run ngrok, you'll see a screen that looks a little like this:
Just copy the second **Forwarding** URL—the one with HTTPS for additional security and you can use it to build a URL to enter in your Twilio dashboard.

For example, if my TwiML was located at `/voicemail/record.xml`, I’d enter the [https://3625ec81.ngrok.com/voicemail/record.xml](https://3625ec81.ngrok.com/voicemail/record.xml) URL on Twilio in the previous ngrok session. You’ll need to keep your ngrok session running for Twilio to be able to access your application, but once you’re done, just hit Ctrl + C.

## Make your application resilient with a fallback URL

The very nature of hosting an application is that despite your best intentions and efforts, your TwiML will be unavailable from time to time. This might be due to a bug or an electrical outage with your hosting provider.

With something as critical as a phone service, you want to be sure that when this happens, those using your service will see at worst a graceful degradation of the service they receive.

By default, in case of an error, Twilio will speak in its (somewhat robotic) text-to-speech voice:

"An application error has occurred. Goodbye."

However, we want to provide a better experience than this. For example, in the case of a customer support phone system that will usually have complex menus and queuing, we might want to redirect the call to our switchboard if there was downtime. Twilio’s **fallback URL** functionality makes this possible.

A fallback URL will be used by Twilio to fetch TwiML for an inbound call or SMS (and potentially, outbound calls too) if your usual server is unavailable or returns an error. Here, you’d store some default TwiML for use in a worst-case scenario. In the previous example, we might use something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait while we connect you to our switchboard.</Say>
  <Dial>
    <Number>+441290211000</Number>
  </Dial>
</Response>
```
We can set up monitoring using Twilio's App Monitor Triggers so that we're informed as soon as our servers start experiencing problems. See Chapter 7, Testing, Debugging, and deploying Twilio Apps, for details.

It makes sense to make the fallback TwiML as simple as possible in order to avoid potential points of failure when our application is experiencing problems.

Naturally, it's of integral importance that you host your fallback TwiML somewhere separate from your main application. I'd recommend Amazon S3 (http://aws.amazon.com/s3/) as a particularly strong bet from an availability point of view.

We can set a fallback URL from the same Numbers page as where we set up our usual Voice and Messaging URLs. To do this, simply click on the Optional Voice Settings or Optional Messaging Settings button as appropriate, and then enter the location of your TwiML in the Fallback URL field. Finally, click on Save.

Use Twilio's applications to manage your TwiML URLs

At the beginning of this chapter, we set up an inbound phone number by editing that number and setting a URL for Twilio to webhook whenever there's an inbound phone call or message.

This is all well and good, but it makes life much harder if we ever want to conduct maintenance on our application. For example, what if you have hundreds of numbers using the same URL, but you then need to change it?

Twilio provides a great solution for this in the form of TwiML apps. The app functionality lets you predefine sets of URLs that you can then assign to different phone numbers.
Working with TwiML

To create an app, in your Twilio dashboard navigation bar, go to Dev Tools and then go to TwiML Apps in the subnavigation:

From there, click on the red Create TwiML App button. Specify a name, which is how you'll identify your app across Twilio's interface, and then you can set the various request URLs, just as you'd do for an individual phone number. Once you're done, hit Save:

An app supports all of the same settings as an individual number, so it can even be used with a fallback URL for improved reliability.
Once you've created an app, you can easily assign it to phone numbers. To do this, head to the Numbers page, choose a number, and then click on Configure with Application. You'll then be able to choose the application you've just created from the list.

From now on, your phone number will stay updated as you make changes to the app. If, for example, you change the URL for an incoming SMS on your app, it'll propagate to each phone number it's attached to.

**Summary**

In this chapter, we learned what TwiML is and the three ways we'll use it: handling inbound calls, dealing with incoming messages, and telling Twilio what to do when we place outgoing calls.

We bought a phone number and then hooked it up to our TwiML. We also saw the data we get from Twilio in webhooks when there's an incoming call or message.

We respond to the webhooks with TwiML made up of verbs and nouns in order to decide what happens with the call, for instance, playing sound clips or sending SMS messages. For the various verbs, we saw helpful tips and tricks and learned best practices in order to work with TwiML more generally.

In the next chapter, we'll build on what we've learned and will start working with Twilio's REST API that, apart from allowing us to place outgoing calls and messages, will give us access to the wealth of data in our Twilio account.
Where to buy this book
You can buy Twilio Best Practices from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.