
H@B Twilio Workshop Documentation

Release 0.1

Kyle Conroy

July 18, 2013

CONTENTS

Today we're going to learn how to use Twilio build powerful SMS and voice-enabled applications.

The beginner track will explore the SMS & voice capabilities of Twilio and culminate in deploying an SMS voting application, live to the Internet via Google App Engine. The advanced track will explore Queues, voice over IP using Twilio Client, and finish off with building a distributed call center app. Happy DOing!

INITIAL SETUP

Before we start the workshop, we're going to need to make sure we have a few things. This guide assumes you have nothing currently set up on your computer. Feel free to skip any sections you've already completed.

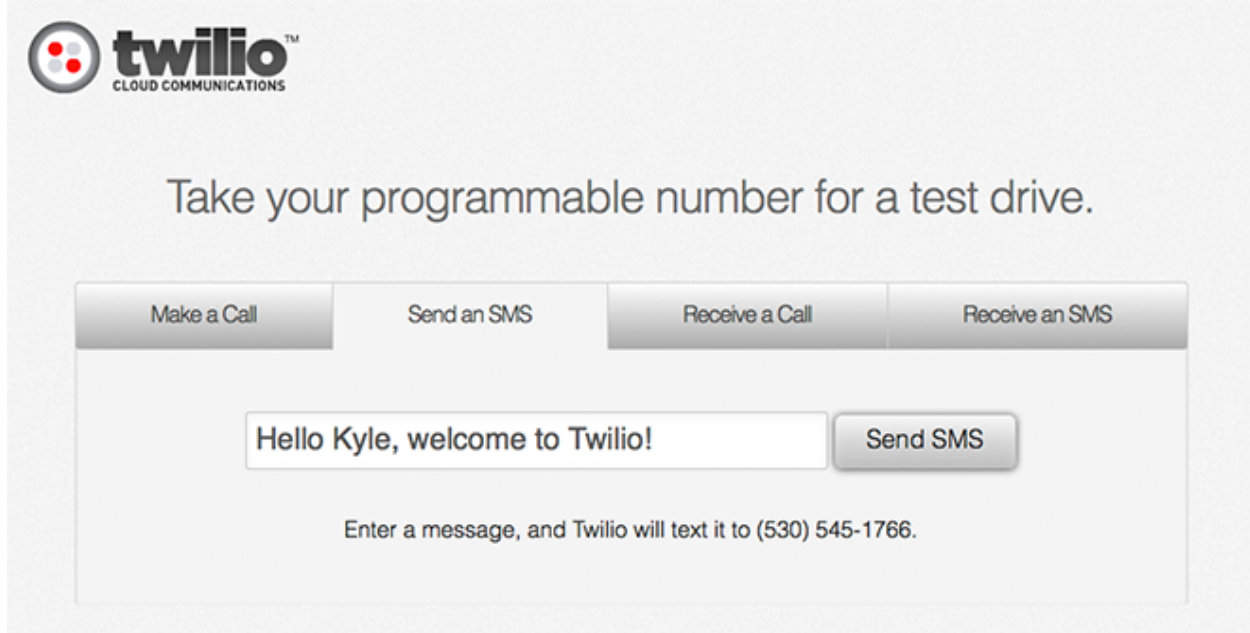
This guide will also setup your computer to build web applications using Google App Engine. If you're planning on using a different development stack during the workshop, please make sure you can easily deploy new code to it during the day.

1.1 Create a Twilio Account

First, [sign up](#) for a free Twilio account. You won't need a credit card, but you will need a phone number to prove you aren't a robot. Once you've signed up, you'll have your own Twilio phone number. We'll use this number for the rest of the workshop.

Make sure to use the promo code you were provided for the workshop.

After you've created your account and verified your phone number, you should end up at a screen that looks like this.



This is your first chance to test out what Twilio can do. Send yourself a text message and receive a call. Congratulations, you've used Twilio for the first time!

1.2 Download Workshop Materials

Download and unzip the [workshop materials](#). If you are familiar with `git`, you can also fork this repository on [Github](#).

1.3 Install a Text Editor

Now that you've signed up, we need to make sure you can edit the workshop code. **If you already have a text-editor or IDE of choice, skip this section.**

- Windows - Download and install [Notepad++](#)
- OS X - Download and install [Text Wrangler](#)
- Linux - Install `gedit` via your package manager
- Any Platform - Download and install [Sublime Text](#) functional demo

1.4 Install Python

Open up Terminal, or a command prompt window. and type the following command. If you aren't sure how to launch your command prompt, ask a TA or a neighbor for help.

```
python --version
```

If the output contains `Python 2.7.x`, your Python installation is ready to go. If not, download the installer for your operating system:

- [Python 2.7.3 Windows Installer](#)
- [Python 2.7.3 Windows X86-64 Installer](#)
- [Python 2.7.3 OS X Installer](#)
- [Python 2.7.3 compressed source tarball](#)

More downloads are available on the [Python downloads](#) page.

Once you are finished, opening up Terminal (OS X) or Powershell (Windows) and verify the output is now the same

```
python --version
Python 2.7.3
```

1.5 Install the App Engine SDK

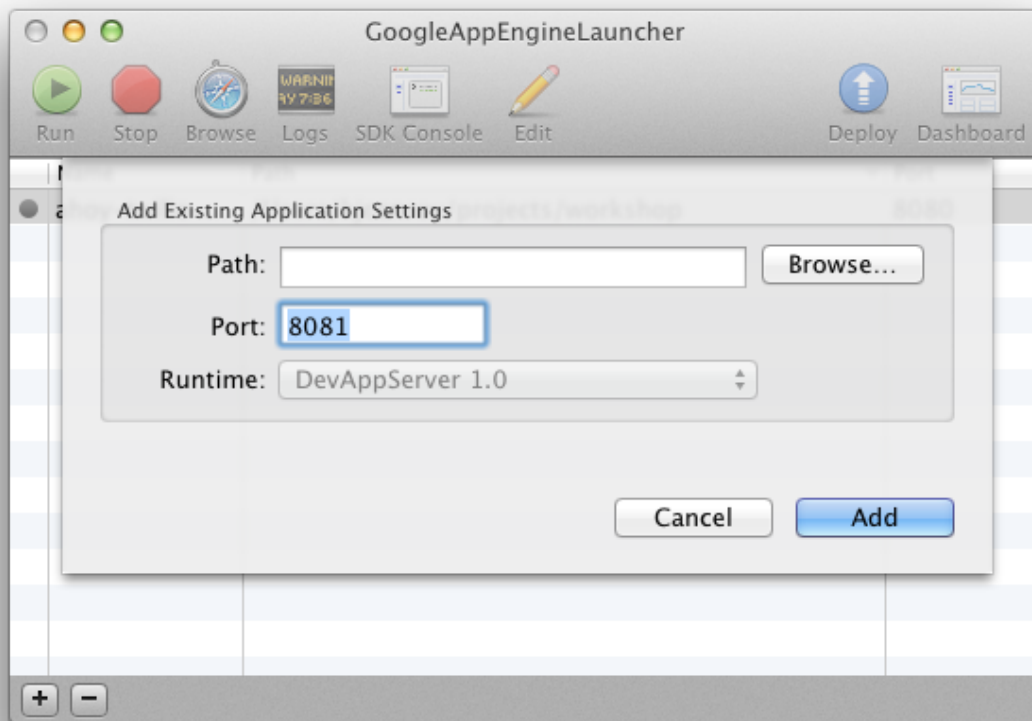
We'll be deploying our Twilio applications to **Google App Engine** during the workshop. **Google App Engine** provides an easy and free way to host your code. Download and install the SDK for your operating system below.

- [App Engine SDK Windows Installer](#)
- [App Engine SDK OS X Installer](#)
- [App Engine SDK Linux/Other Platforms](#)

1.5.1 Basic Application Setup

The workshop directory you downloaded earlier contains a basic web application that we'll be extending during the workshop. We'll use this location to perform local testing, and the code we edit here we'll later deploy to **Google App Engine**. To do this, we need to tell the **Google App Engine Launcher** where to find our files.

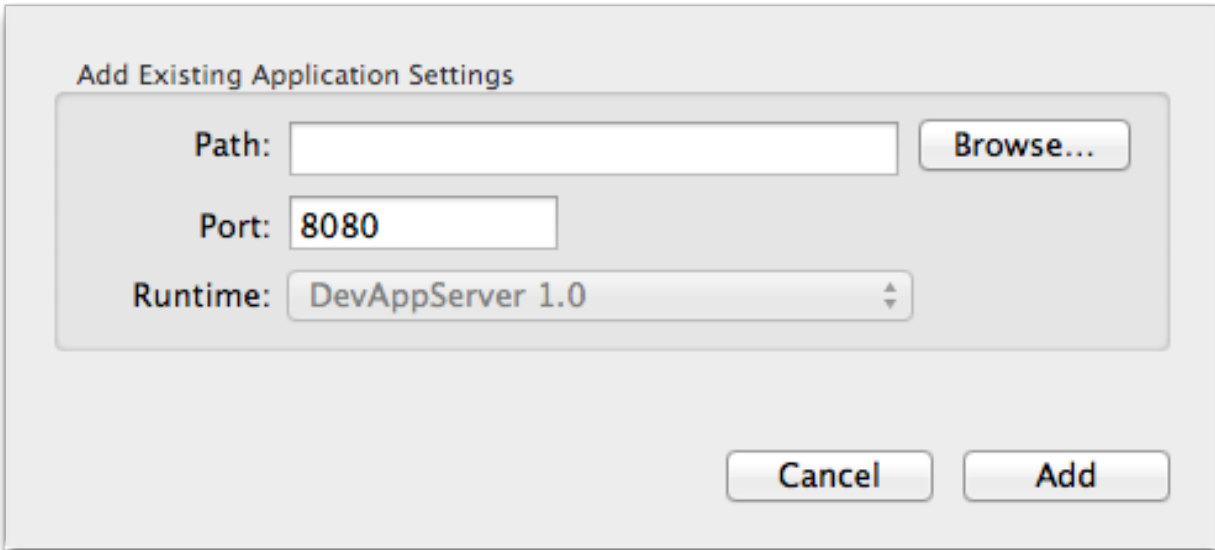
Open the **Google App Engine Launcher**, and from the file menu select "Add Existing Application...". In the next dialog, click the "Browse" button and locate the workshop folder.



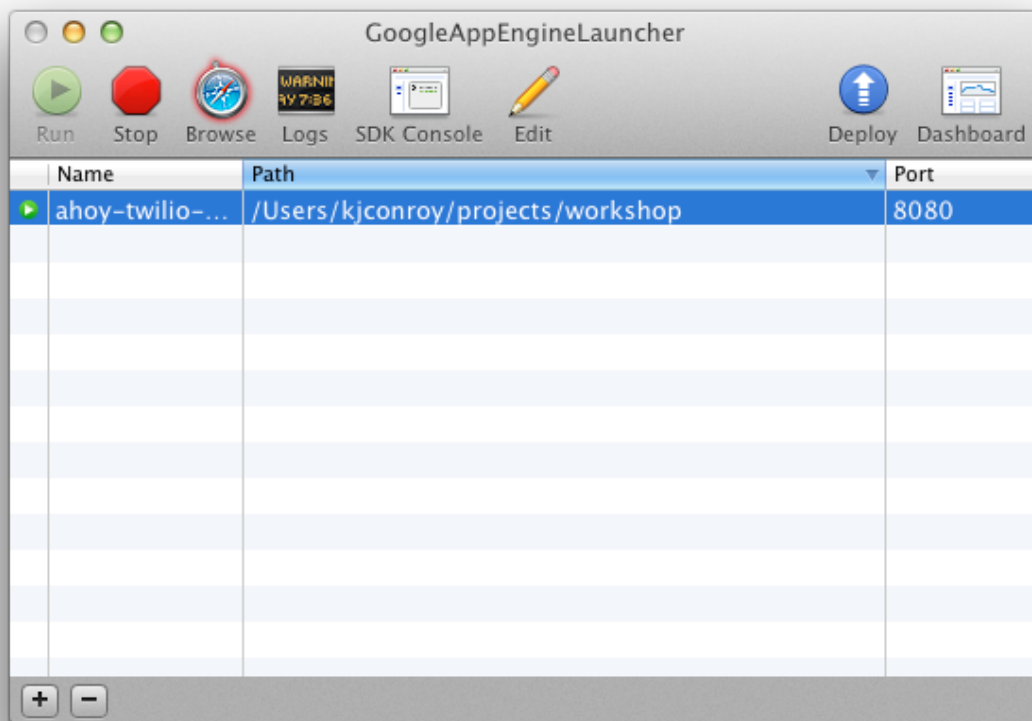
Click "Add" to finish setting up the application.

1.5.2 Run the Web Application

To run your application, select your application from the list and click the "Run" button. Your app is now running and ready to use.



To use your application, click the “Browse” button. Your app will launch in a browser window.



Your browser will open with the text “Hello World!” on your screen. Let’s take a moment and look at the URL that is loaded in your browser.

The URL says `http://localhost:8080/`. *localhost* is a special URL that tells the browser to make a request to your local computer instead of out to the internet. The `:8080` portion tells the browser to make the request to port

1.5.4 Deploy your Application

It's now time to share your application with the world. To deploy your application on App Engine, we'll need to create an application via your App Engine dashboard (which requires a Google account).

Open the [App Engine dashboard](#) in a new tab and click "Create Application".

Google app engine

ahoy@twilio.com | [My Account](#) | [Help](#) | [Sign out](#)

My Applications

Prev 20 1-1 of 1 Next 20					
Application	Title	Billing Administrator	Storage Scheme	Location	Current Version
ahoy-twilio-workshop	Twilio Workshop		High Replication	US	1 ↗
Create Application Prev 20 1-1 of 1 Next 20					

You'll need to pick a name and title for your application. Names in **Google App Engine** need to be lowercase and unique so I'd suggest a workshop specific name like {lastname}-twilio-workshop.

Google app engine

ahoy@twilio.com | [My Account](#) | [Help](#) | [Sign out](#)

Create an Application

i All applications you create will be billed to the twilio.com Premier Account.

Application Identifier:

.appspot.com

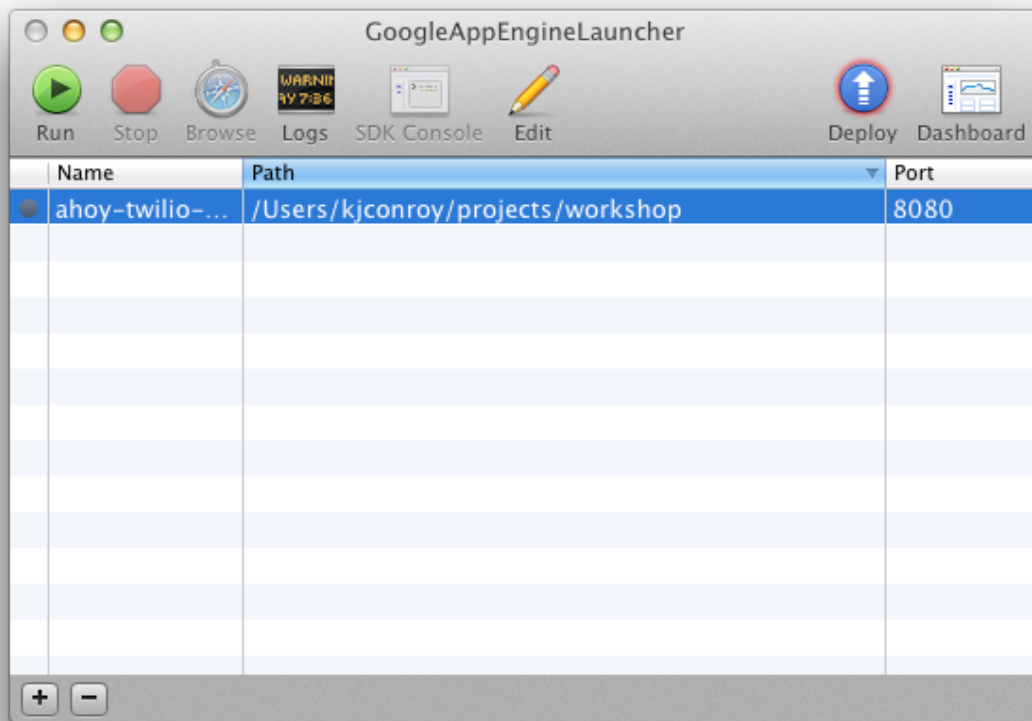
All Google account names and certain offensive or trademarked names may not be used as Application Identifiers. You can map this application to your own domain later. [Learn more](#)

Application Title:

Displayed when users access your application.

Accept the terms and conditions and click "Create Application"

You'll need to update your local configuration with your new application name. Open the **Google App Engine Launcher** and highlight your application. Click the "Edit" button.



You'll be asked to sign in with your Google account.

Note: If you use [2-Step Verification](#), you'll need to create an application-specific password to authorize your account. You generate these on the [Authorizing applications & sites](#) page.

The Launcher app will also output progress information in to the Log:

```
*** Running appcfg.py with the following flags:
    --no_cookies --email=user@email.com --passin update
04:42 PM Host: appengine.google.com
04:42 PM Application: my-application-name; version: 1
04:42 PM Starting update of app: my-application-name, version: 1
04:42 PM Getting current resource limits.
04:42 PM Scanning files on local disk.
04:42 PM Cloning 68 application files.
04:42 PM Uploading 4 files and blobs.
04:42 PM Uploaded 4 files and blobs
04:42 PM Compilation starting.
04:42 PM Compilation completed.
04:42 PM Starting deployment.
04:42 PM Checking if deployment succeeded.
04:42 PM Deployment successful.
04:42 PM Checking if updated app version is serving.
04:42 PM Completed update of app: my-application-name, version: 1
Password for user@email.com: If deploy fails you might need to 'rollback' manually.
```

The "Make Symlinks..." menu option can help with command-line work.
*** appcfg.py has finished with exit code 0 ***

Once you see *** appcfg.py has finished with exit code 0 ***. your application is live and ready to view. Open a browser window and go to <http://{your-application-name}.appspot.com> to view your application in action.

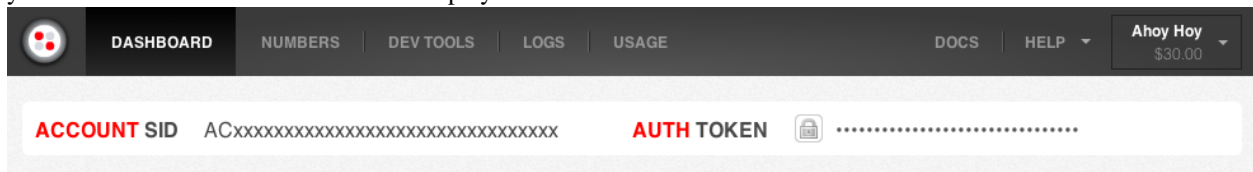
IDEA FACTORY

2.1 Hello World

The first thing we'll do today is explore Twilio's REST API calls. Soon, you'll see how SMS and phone calls can be originated from your web browser with Twilio.

2.1.1 Twilio Account Portal

When you log into your Twilio Account, the first page you come across is your Account Dashboard. This is where your Account Sid and Auth Token are displayed.



These are your account credentials. The Account Sid acts as a username and the Auth Token acts as a password. Twilio uses your Account Sid and Auth Token to authenticate the API requests made by your application.

Analytics about your Voice and SMS application are also shown here. We'll go over these after we've made some calls and sent some SMS messages.

At the bottom of your Account Dashboard is the API Explorer and the Debugger.

2.1.2 Twilio API Explorer

The [API Explorer](#) is a helpful application built into the Account Dashboard that allows you to easily try out Twilio's API without getting into the details of scripting and *HTTP* calls.

You can get to the [API Explorer](#) from the bottom of your Account Dashboard. You may also click on the [Dev Tools](#) tab to access the API Explorer.

API EXPLORER

Make requests to the Twilio API in Zero Lines of Code.

Twilio API Explorer makes it even easier to start building applications with Twilio. With the API Explorer, you can make requests to our APIs, copy code snippets for use in your own applications and see how Twilio responds. The API Explorer is great way to try buying phone numbers, sending text messages, making phone calls and more, all through a nifty web interface.

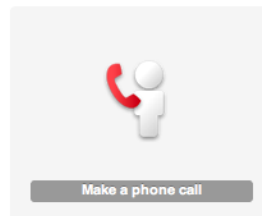
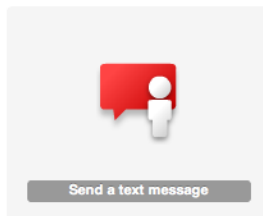
To

From

GENERATE CODE

```
@client = Twilio::REST::Client.new(
  @account = @client.account
  @call = @account.calls.create({:from
puts @call
```

Try it out now!



Getting Started

Welcome

SMS Messages

Calls

Incoming Phone Numbers

Caller IDs

Available Phone Numbers

Recordings

Transcriptions

Conferences

Applications

Notifications

Sandbox

Accounts

Getting Account Information

Let's start with a basic request for your account information. Go to your Twilio Account Portal, click on *Dev Tools*, then click on *Accounts* (on the right near the bottom), then click on *View Account*.

API EXPLORER

View account

Returns a representation of an account.

GET /2010-04-01/Accounts/[AccountSid].[format]

[View Docs](#)

You can see there are a couple of pre-filled fields that make up the request parameters:

Pa- rame- ter	Definition
<i>Format</i>	The <i>Format</i> field tells the API in what format we want the API to respond with. For our purposes right now this doesn't matter, but you can request the response in either XML or JSON format.
<i>Ac- countSid</i>	The <i>AccountSid</i> field tells the API which Account we want to use to send the SMS. This is pre-populated with your account.

Go ahead and click on the *Make Request* button. A response with your Account information should appear at the end of the page.

Ok, so what did we just do?

Let's take a look at the key parts of the request we just made:

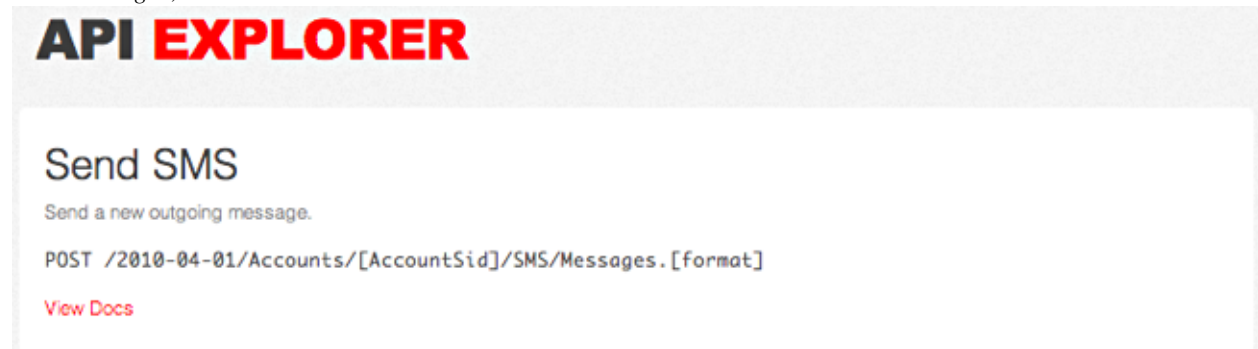
```
GET /2010-04-01/Accounts/AC000.xml
```

- **GET:** this is the type of request to make, a GET request, meaning that the intent of this call is to retrieve information from the server.
- **/2010-04-01/Accounts/AC000.xml:** this is the endpoint for retrieving Account data from the Twilio API. Lets break it down
 - **/2010-04-01/** is the version of the API that we want to request. The version of the API we want to talk to is important because we want to make sure that the way we talk to the API doesn't change. If Twilio makes a major change to how to talk to the API the version will change, but the old version will continue to work the same way so that your application doesn't break.
 - **Accounts/AC000.xml** means that we want to retrieve an Account resource for the account "AC000" and that we want the response in XML format.

Now lets look at the response. The server responded with an XML formatted representation of your account information including, but not limited to, your AccountSid, Account Name, Auth Token, and Resource URIs for the resources available to your account.

Sending an SMS

Let's send a text message using the [API Explorer](#). Go to your Twilio [Account Portal](#), click on *Dev Tools*, then click on *SMS Messages*, then on *Send SMS*.



Here we can try out the Twilio API for sending SMS Messages. All the fields required to send an SMS are visible. This request will add 2 new parameters:

Parameter	Definition
<i>From</i>	The <i>From</i> field tells the API which phone number to use to send the Message. This can only be one of the phone numbers you've purchased or ported into Twilio.
<i>To</i>	The <i>To</i> field tells the API where to send the message. The phone number should be in E.164 format. Twilio will assume that <i>To</i> phone numbers without a "+" will have the same country code as the <i>From</i> phone number.
<i>Body</i>	The body is a freeform field to enter your message. You can enter a message up to 160 characters long.

Enter your cell phone number in the *To* field along with a text message *Body*, and click the *Make Request* button at the bottom of the page. This will send the information you've just entered to the Twilio API. You will be prompted to

confirm the use of funds from your account. Aren't you glad you got the Promo credit?

Twilio will process the information you have submitted and your phone will receive a text message shortly.

So how was that different from our Accounts information request?

Lets take a look at the key parts of this request:

```
POST /2010-04-01/Accounts/AC000/SMS/Messages.xml
```

There are a few key differences to note:

- **POST** this time we're making a POST request, meaning that the purpose of this request is to pass data in to the API for the purposes of modifying the SMS Messages resource.
- **Parameters:** if you look at the *Code Example* right above the *Make Request* button you see **-d 'From=xxx'*** et al... these tell cURL what data to pass to the API. You can see each entry you modified in the form is represented here. You'll also notice that each entry contains special characters (ie: %2B instead of +). This is called [Url Encoding](#) and is required to make sure that special characters are properly transmitted to the API.

Now lets examine the response. You'll see that the message was given a *Sid*, a unique identifier, how Twilio interpreted the information you sent, and you can see that it was queued for delivery.

Click on the Message Sid and you'll be taken to another page where we can get information about the message.

Now click on *Make Request* to see the current status of the message. You can see, among other information, that the message was sent and how much it cost to send.

Making a Phone Call

Now let's make a phone call using the [API Explorer](#). Click on the Calls link on the left hand sidebar, then on the sublink "Make call".



The request parameters should look familiar by now. This request replaces the *Body* field with a *Url* field:

Pa- rame- ter	Definition
<i>Url</i>	The <i>Url</i> field tells the API where to load TwiML instructions for handling the call. TwiML is a set of instructions that tells Twilio what to do. Don't worry, we'll get more into TwiML later.

Enter your cell phone number in the *To* field. To make things easy, we're going to use a [Twimlet](#) for the *Url*. We'll get into the details of building TwiML later on. For now, copy the URL below into the *Url* field.

```
http://twimlets.com/message?Message=Hello+World
```

Click on the *Make Request* button at the bottom of the page. Again, the information you've submitted is sent off to Twilio and your phone should start ringing momentarily.

How was that different from our SMS request?

In this request we replaced the *Body* parameter with a *Url* parameter. The URL is required to be an endpoint that returns TwiML. This TwiML will tell Twilio how to handle the phone call.

If you load the URL we supplied directly in to a web browser you can see the TwiML that was used to handle the phone call. Don't worry about understanding it right now, we'll get in to that in the next section.

If you examine the API response you'll notice it looks much like the response we got from sending the SMS, but with a few different values. Click on the *Sid* in the response to be taken to a page where we can request call details.

Click on *Make Request* to see the details on the completed call.

Have any questions? Ask your TA!

2.1.3 Additional Information

- [TwiML: the Twilio Markup Language](#)
- [Twilio REST API - Calls Resource](#)
- [Twilio REST API - SMS/Messages Resource](#)

2.2 Introduction to TwiML

We've successfully made a phone ring, but how do we actually control call flow? [TwiML](#) is the answer. TwiML is a set of instructions you can use to tell Twilio what to do when you receive an incoming call or SMS. TwiML instructions are formatted in XML and are case sensitive.

When someone makes a call or sends an SMS to one of your Twilio numbers, Twilio will look up the URL associated with that phone number and make a request to that URL. Twilio will read TwiML instructions at that URL to determine what to do: record the call, play a message for the caller, prompt the caller to press digits on their keypad, etc.

TwiML is an XML-based language and consists of five basic verbs for Voice calls.

- [Say](#)
- [Play](#)
- [Gather](#)
- [Record](#)
- [Dial](#)

And one basic verb for SMS messaging.

- [Sms](#)

To see how these verbs work, let's first take a look at the last section. When we called your phone, a robot answered with a "Hello World" message. We now know that TwiML powered that call, so let's take a look. Open <http://twimlets.com/message?Message=Hello+World> in your browser.

Notice the TwiML used for this application. This application uses the [Say](#) verb and says "Hello World" with a text-to-speech engine. For outgoing calls, we choose the TwiML URL at the time of the call. For incoming calls, we set a TwiML URL that is fetched each time someone calls into your Twilio number.

2.2.1 Twimlbin

Twiml can be hosted anywhere. It can be a static XML document or created dynamically by a web application. To make developing Twilio applications easier, you can host your TwiML on [Twimlbin](#). Applications with Twimlbin will autosave and let you know when there might be potential errors with your TwiML. While we love Twimlbin, we recommend that you host your Twilio production application on your own server.

Let's rebuild the "Hello World" greeting in [Twimlbin](#).

To create a new bin, go to the Twimlbin homepage and click "Create a new Twimlbin".

Now let's write our "Hello World" application with the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response><Say>Hello World</Say></Response>
```

Your final application should look like this.

Twimlbin

Twimlbin allows you to host Twilio Markup Language without a webserver, making it easy to get a Twilio phone number setup for Voice & SMS in no time.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Response>
3   <Say> Hello World. </Say>
4 </Response>
```

SAVED VALID TWIML

Put your valid TwiML (invalid XML will be shown in red) into the editor above. When Twilio requests this URL, we'll serve them that content directly rather than this formatted page.

Private:

<http://twimlbin.com/69f8c0b6>

Your **editable** Twimlbin - use this to edit your TwiML and as the URL when setting up a Twilio number.

Public:

<http://twimlbin.com/external/58a62abd62b3563c>

Your **public** Twimlbin URL - use this to share with others without allowing edits.

2.2.2 Configuring your phone number

Once you're done building your application, we'll want to configure your Twilio phone number. By configuring the Twilio phone number, whenever an incoming call is received on this number, Twilio will fetch the TwiML that is located at that URL.

First, we'll need to copy the "Public" URL of your Twimlbin.

Go to [your Twilio numbers page](#) and click on your Twilio phone number. Change the "Voice URL" field to your Twimlbin URL and "Save Changes"

The screenshot shows the Twilio console interface for configuring a phone number. It is divided into two main sections: 'Voice' and 'SMS'. Each section has a 'Voice Request URL' or 'SMS Request URL' field, a 'POST' button, and a description of the action. Below each section is a link for 'Optional Voice Settings' or 'Optional SMS Settings'. At the bottom, there is a red 'Save Changes' button and a 'Release Number' link.

Voice URLs ⬆️ ⬆️ ?

Voice Request URL POST ⬆️

Retrieve and execute the TwiML at this URL via the selected HTTP method when this number receives a phone call.

[Optional Voice Settings](#)

SMS URLs ⬆️ ⬆️ ?

SMS Request URL POST ⬆️

Retrieve and execute the TwiML at this URL via the selected HTTP method when this number receives an SMS.

[Optional SMS Settings](#)

Save Changes Release Number

Now give your Twilio number a call! You should hear a "Hello World" greeting.

2.2.3 Call Logs & SMS Logs

Now you're probably thinking:

- How long did that call last?
- How much did that call cost?
- What time was the call made?

All this information and more can be found in your [Call Logs](#).

Let's head over to your [Call Logs](#) by clicking on the Logs tab from your Account Dashboard.

You'll notice that the call duration is listed as 1. Call duration is rounded up to the nearest minute.

You SMS Messages logs can also be found under the header [SMS Messages](#)

Have any questions about your Logs? Ask your TA!

2.2.4 Debugging Errors

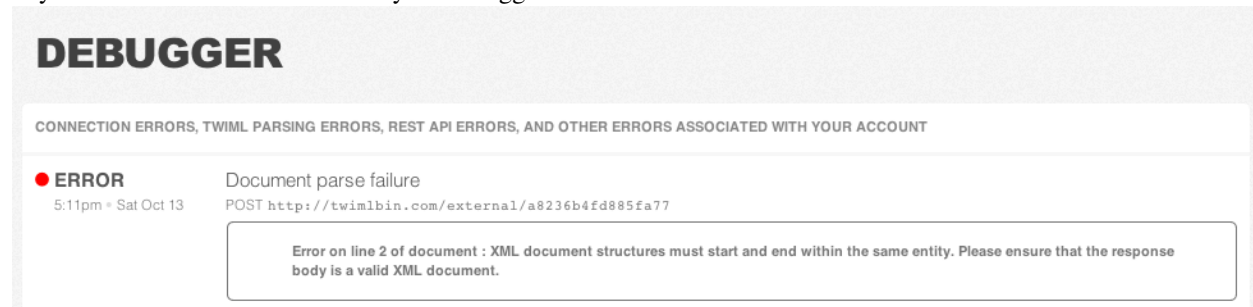
Nobody's perfect. Eventually, something will break and your application won't work. Instead of hoping this doesn't happen, let's make an error occur. Copy and paste the following TwiML into your Twimlbin.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
```

This TwiML is invalid. We open the Response and never close it.

Call your phone number. Do you hear application recorded message that says “We're sorry, an application error has occurred”?

Now let's find out why your application error has occurred. The first place we'll want to look is the [debugger](#). Navigate to your account dashboard and find your debugger.



Click on the error to see more detail.

The Debugger lets you know where in your application Twilio ran into an error. This page is broken down into three sections.

The [Request](#) section provides information on the data Twilio sent to your server.

The [Response](#) section lets you know how your server responded to Twilio. Twilio will always expect correctly formatted TwiML as a response. If your application tries to respond to Twilio with anything else, you will likely run into an error.

The Body section shows the content your application returned to Twilio. Here you'll see the invalid TwiML from your Twimlbin.

Find the error within the response your application sent to Twilio. What should it look like?

Hint: You may also click on the more information link at the top of the page.

2.2.5 Additional Information

- [TwiML: the Twilio Markup Language](#)
- [Twilio's Voice Request](#)
- [Your Voice Response](#)
- [Twilio's SMS Request](#)
- [Your SMS Response](#)
- [Debugging](#)

2.3 Building Static Applications

Congratulations! Now that you’ve built and debugged your first Twilio application, let’s start building Twilio applications with some of the other TwiML verbs. These applications can be hosted on [Twimlbin](#).

2.3.1 Call Forwarding

The [Dial](#) verb allows you to connect calls to other people. The following TwiML will forward any call received on your Twilio phone number to your personal phone number. Once you’ve wired up this TwiML to your number get a neighbor to test it out.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Please wait while we forward your call.</Say>
  <Dial>YOUR PHONE NUMBER</Dial>
</Response>
```

Twilio Call Legs

Find this call in your [Call Logs](#). You should notice two call records listed. What’s the difference between the two call records?

Direction For applications like Call Forwarding, your call will include two call legs. The inbound call leg noted under *Incoming* is the call made into Twilio. The outbound call leg noted under *Outgoing Dial* is the call made from Twilio with the [Dial](#) verb out to another phone number. Call Forwarding applications include both an inbound leg and an outbound leg.

Cost The cost of inbound and outbound calls are different. Inbound calls cost 1¢ per minute while outbound calls start at 2¢ per minute. The cost of outbound calls may also differ depending on the end destination of your outbound call. See the [Voice Pricing](#) page for more pricing information.

To While the “From” phone numbers are the same, the “To” phone number are different based on the phone number receiving your call.

2.3.2 Introducing Attributes

Each TwiML verb and noun has a set of attributes that allow you to modify its behavior. Let’s change our robot voice to a female voice by adding the [voice attribute](#) and setting it to “woman” like so, `<Say voice="woman">`. Additionally, let’s record our call by including the [record attribute](#) and mark that as “true”, `<Dial record="true">`.

For our last call, the caller ID displayed was the phone number where the call originated from. Let’s change your caller ID using the [callerId attribute](#) so that it displays your Twilio phone number instead, `<Dial callerId="YOUR TWILIO PHONE NUMBER">`.

Earlier we verified your personal phone number so you may also use that as your caller ID. To use other phone numbers as your caller ID you can [verify those numbers](#) in your Twilio Account.

Test out the following code and see how the attributes have changed your application.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say voice="woman">Please wait while we forward your call</Say>
  <Dial record="true" callerId="YOUR TWILIO PHONE NUMBER">
    YOUR PHONE NUMBER
```

```
</Dial>
</Response>
```

Now try out some the [Say](#) verb and [Dial](#) verb attributes and see what they do.

2.3.3 Voice Mailbox

Recording audio is accomplished through the [Record](#) verb. The Record verb will play a beep and wait until a user presses # or hangs up. While the record attribute used earlier will record both sides of the conversation, the [Record](#) verb is used to record just the inbound call leg. Copy this TwiML into your bin and save. You can now leave messages on your number.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>After the beep, record your message</Say>
  <Record/>
</Response>
```

After you're done recording your message hang up. Twilio begins processing the recording right after your done. Head to your [recordings log](#) to listen to your message.

To delete your recording through the User Interface, click on the date of the recording. There's a "Delete Recording" link at the bottom right hand corner.

RECORDINGS

RECORDING SID	RE5fffd7f07e1ddefe119310623c8c155	Listen
CALL SID	CA[REDACTED]	Details
DATE	18:25:32 PDT 2012-07-23	
SOURCE		
DURATION	0 mins 36 sec	
		Delete Recording

Transcriptions of your recordings can also be made by Twilio with the [transcribe](#) attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>After the beep, record your message</Say>
  <Record transcribe="true"/>
</Response>
```

Rerecord your message. This time, record a longer message.

Head over to your [transcription log](#) to see your transcription and listen to your message.

2.3.4 Private Conference Line

Many times during project collaboration, you just need to get everyone on the same page. You can now have your own private conference line using the [Conference](#) noun and [Dial](#) verb. Put the following TwiML into your bin and save.

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

Now, you can give your Twilio number to a few people around you, have everyone call in, and start up a conversation.

2.3.5 One Song Music Hotline

To play an audio file back to the caller, use the [Play](#) verb. We can build a simple music hotline that just plays just one song to the caller.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>You are about to listen to Flight of Young Hearts by Mellotroniac.</Say>
  <Play>http://com.twilio.music.classical.s3.amazonaws.com/Mellotroniac_-_Flight_Of_Young_Hearts_Flut
</Response>
```

2.3.6 SMS Follow-Up

Using the [Sms](#) verb you can send SMS messages right after your call has ended.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>I have received your call.</Say>
  <Sms>Thank you for calling.</Sms>
</Response>
```

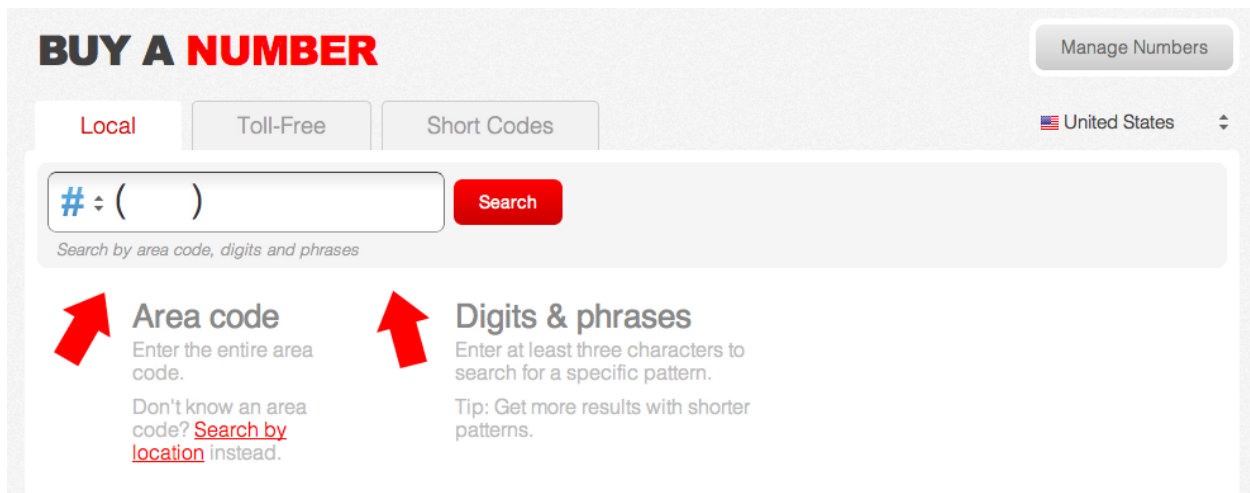
That was easy!

2.3.7 Swiss-Army Phone Number

Equipped with the knowledge of TwiML, you can now bend your Twilio phone number to your will. You've forwarded a call, recorded a message, and started a private conference line. Your phone is now yours to control.

But with this many applications, we definitely want more than one Twilio phone number. Let's purchase another number.

To purchase a Twilio phone number you will want to navigate to the [Numbers](#) tab. Click on the blue button titled [Buy a number](#).



You may search for the Twilio phone number you would like to purchase by area code, digits, and phrases as well as by the location.

2.4 Building Dynamic Applications

We’ve built awesome Twilio applications in the last two sections, but we’ve been limited to static TwiML. The true power of Twilio can only be unlocked by using a web application.

This section assumes you’ve completed the *Initial Setup* and have the Google App Engine SDK running locally on your computer.

2.4.1 Your first web application

The first part of this guide walks you through running a sample application. Before continuing, make sure your “Hello World” app is running and you have “Hello World” displayed in your browser. If you can’t remember how to run the sample app, refer back to *Initial Setup*.

Before we write our dynamic Twilio application, let’s first understand the “Hello World” example. Let’s go through the example line-by-line and see how it works. Inside our `main.py` file:

```
import webapp2
```

This line is the first part of our application. We use the `webapp2` Python module to create our web application. Before we can use it in our application, we must first import it.

```
class HelloWorld(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello World!')
```

Whenever a user makes a request to our application, this is the code that will be run. The output of the code gets displayed to the web browser. We will name this block of code “HelloWorld”. The name of a block of code is called the *RequestHandler*.

We are also making a request called `get`, which grabs the requested resource. This corresponds to the HTTP GET request. If you’d like to learn more about HTTP, the language browsers use to talk to servers, take a look at our *http* section.

```
app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)
```

In this part of our code, we create our application using the *webapp2* framework.

The web applications is a mapping of the URL we specify with the listed request handler. The above mapping says “Whenever someone visits the front page of my application (the / URL), process that request using the HelloWorld request handler”.

Your first task will be to change the message displayed in your browser. Open up `main.py` in your text editor and change the “Hello World” message on line 6 to “Hello TwilioCon”. Refresh the page to see your new message.

Congratulations! You’ve just created your first web application.

2.4.2 Responding with TwiML

A simple message is great, but we want to use our application to serve TwiML. How do we respond with TwiML instead of plain text? First, let’s change the message we respond with to valid TwiML.

```
import webapp2

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('<Response><Say>Hello TwilioCon</Say></Response>')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)
```

When someone requests the front page of our application, they will now get TwiML instead of HTML. If you refresh your page, nothing seems to have changed.

The problem is that while we’re sending back TwiML, the browser still thinks we’re sending it HTML. Since we never tell the browser that we are sending XML, which is the format of TwiML, it assumes that we are using HTML. To fix this problem we’ll include additional metadata via an HTTP header to tell the browser we’re sending valid TwiML.

```
import webapp2

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = "application/xml"
        self.response.write('<Response><Say>Hello TwilioCon</Say></Response>')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)
```

When you refresh the page, you should now see the entire TwiML response, (it may even be highlighted and formatted).

Using the Twilio Helper Library

Manually writing TwiML soon becomes very tiresome. If you miss a single ending tag, your entire application can break. Instead, we’ll use the `twilio-python` helper library to generate TwiML for us. This way we won’t have to

worry about messing up the syntax. To use the `twilio-python` helper library, we'll import it from Twilio with the code from `twilio import twiml` in line 2.

```
import webapp2
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = "application/xml"

        response = twiml.Response()
        response.say("Hello TwilioCon")
        self.response.write(str(response))

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)
```

When you refresh your page nothing should look different. The helper library code we just wrote is equivalent to the static TwiML we had before. Let's explain what the added code is actually doing.

```
response = twiml.Response()
```

Here we create a new `Response` object. Every Twilio application must begin with the `Response` TwiML. We'll add additional TwiML verbs and nest them within the `Response` TwiML.

```
response.say("Hello TwilioCon")
```

This methods adds a `Say` verb to the `Response` object. The other TwiML verbs `Play`, `Gather`, `Record`, and `Dial` may also be used as methods on `Response`.

```
self.response.write(str(response))
```

Here we turn our response code into a string using Python's built in `string` function. This will write a string to the response object and return a response to Twilio.

2.4.3 The Weather Channel

So far all our responses look the same. We're just returning static TwiML with the same message. Now let's build a dynamic application that interacts with your inputs. How about building "Weather Channel"! Instead of simply reading a message, we'll inform the caller of the current weather in his or her ZIP code.

To begin, we need data on the weather. Let's import the weather data from Yahoo! Weather API, from `util import current_weather`. Insert this code in line 2 right before we import our helper library.

Now that we have data on the weather, let's make sure our application can get the current weather of a particular ZIP code. Use the following code `weather = current_weather("94117")` so that we can get the weather from San Francisco's area code 94117. Let's also include the city so we can acknowledge to our callers where they are getting their weather from, `city = "San Francisco"`.

Finally, let's add our TwiML and so Twilio knows how to respond to the caller. In the end your application should look like this:

```
import webapp2
from util import current_weather
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):
```

```

def get(self):
    self.response.headers['Content-Type'] = "application/xml"

    weather = current_weather("94117")
    city = "San Francisco"

    response = twiml.Response()
    response.say("Hello from " + city)
    response.say("The current weather is " + weather)
    self.response.write(str(response))

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)

```

When you visit your localhost:8080 page. You'll see the following message.

```

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Hello from San Francisco</Say>
  <Say>The current weather is Partly Cloudy, 65 degrees</Say>
</Response>

```

Let's revisit our application. Right now our application only gets the weather for San Francisco for the ZIP code 94117.

Phone numbers contain a lot of data though. By using the phone number of your caller, Twilio can pass the ZIP code and city information to your application. Twilio labels this information "FromZip" and "FromCity". Instead of just getting the weather for San Francisco, let's make this application more relevant to your caller and use this data. Keep in mind, this isn't actually the ZIP code of the caller's live location though.

```

import webapp2
from util import current_weather
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = "application/xml"

        weather = current_weather(self.request.get("FromZip", "94117"))
        city = self.request.get("FromCity", "San Francisco")

        response = twiml.Response()
        response.say("Hello from " + city)
        response.say("The current weather is " + weather)
        self.response.write(str(response))

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)

```

In the case we can't find your ZIP code or city, your application will default back to providing the weather of San Francisco.

To test out the greeting, add the FromZip and FromCity parameter to your URL.

<http://localhost:8080/?FromZip=15601&FromCity=Greensburg>

You should now see the weather for Greensburg, PA show up in your TwiML response.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Hello from Greensburg</Say>
  <Say>The current weather is Cloudy, 59 degrees</Say>
</Response>
```

Whenever an HTTP request is sent to your application from Twilio, it includes data in query string and body of the request. The code we added when constructing the Say verb pulls that data from the HTTP request parameter.

```
self.request.get('FromZip')
```

Incoming Twilio Data

Adding this parameter to your URL mimics the request that Twilio will send to your server. All TwiML requests made by Twilio include additional information about the caller. Here is short list of some of the data that Twilio will send to your server with every call.

Parameter	Description
From	The phone number or client identifier of the party that initiated the call.
To	The phone number or client identifier of the called party.
CallStatus	A descriptive status for the call. The value is one of queued, ringing, in-progress, completed, busy, failed or no-answer
FromCity	The city of the caller.
FromState	The state or province of the caller.
FromZip	The postal code of the caller.
FromCountry	The country of the caller.

Phone numbers are formatted in E164 format (with a '+' and the country code, e.g. *+1617555121*).

For a complete list, check out [Twilio request parameters](#) on the Twilio Docs.

2.4.4 Handling Server Errors

Sometimes, errors will occur on the web application side of the code.

Internal Server Error

The server has either erred or is incapable of performing the requested operation.

```
Traceback (most recent call last):
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    rv = self.handle_exception(request, response, e)
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    rv = self.router.dispatch(request, response)
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    return route.handler_adapter(request, response)
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    return handler.dispatch()
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    return self.handle_exception(e, self.app.debug)
  File "/Applications/GoogleAppEngineLauncher.app/Contents/Resources/GoogleAppEngi
    return method(*args, **kwargs)
  File "/Users/twilio/cats/cats/kittens/main.py", line 6, in get
    self.response.write('Yes this is a contrived example.' + None)
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

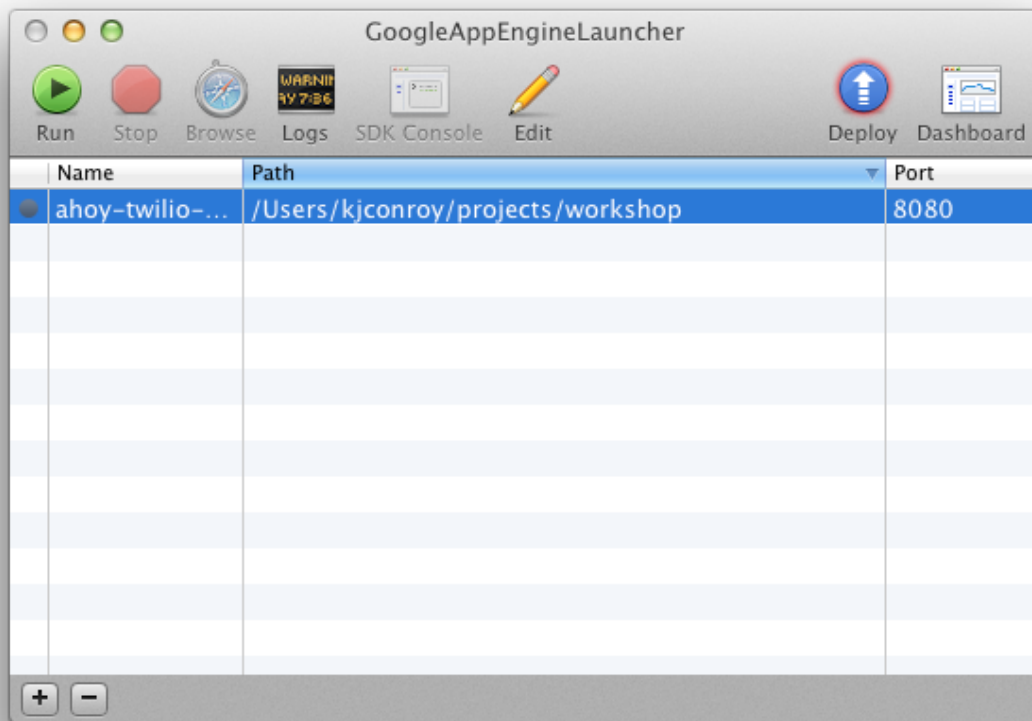
Don't panic if you see this. The stack trace will usually give you hints as to what error the application encountered, and where it occurred.

Some errors may also appear on the AppEngine logs. If the errors on the browser aren't too informative, try clicking on the "Logs" button on the AppEngine Launcher.

2.4.5 Deploy your Twilio application

We're now ready to hook up your brand new application to a Twilio number. To do this, we'll need to host your application live on the Internet, so that Twilio can access it!

Open the Google App Engine Launcher application, highlight your application, and hit the "Deploy" button. A window will pop up and show you the status of your deployment. It should take less than a minute to deploy.



Once it's deployed, copy the URL for your application, `http://<your-application-name>.appspot.com` and set it as the voice number for your Twilio phone number. Configuring Twilio numbers is covered in more detail in *Configuring your phone number*.

Note: Since we have only implemented the GET endpoint, be sure to configure your number to use the GET method instead of the default POST*

Now give it a call. You should hear your custom message. Hooray!

2.4.6 Gathering Digits From the Caller

Since not everyone's phone number is from the location they currently live, it may be helpful to add a feature to our app for checking the weather of any ZIP code. This allows us to interact with our application.

To achieve this, we're going to use a TwiML verb called `<Gather>`.

Let's begin by adding a `<Gather>` menu:

```
import webapp2
from util import current_weather
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):
```

```
def get(self):
    self.response.headers['Content-Type'] = "application/xml"
    city = self.request.get("FromCity", "San Francisco")

    response = twiml.Response()
    gather = response.gather(numDigits=1)
    gather.say("Press one for the weather in " + city)
    gather.say("Press two to get the weather for another zip code.")
    self.response.write(str(response))
```

The TwiML we've generated so far for the menu looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Gather numDigits="1">
    <Say>Press one for the weather</Say>
    <Say>Press two to get the weather for another zip code.</Say>
  </Gather>
</Response>
```

We now have a `<Gather>` verb which will allow the caller to type in exactly 1 digit (because of the attribute `numDigits`). Next, we need to hook it up to something that can process the input based on which digits were pressed. When we use `<Gather>`, these digits are passed to your application in the next callback using the parameter `Digits`.

Let's add some additional code to handle this callback.

```
import webapp2
from util import current_weather
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = "application/xml"
        city = self.request.get("FromCity", "San Francisco")

        response = twiml.Response()
        gather = response.gather(method="POST", numDigits=1)
        gather.say("Press one for the weather in " + city)
        gather.say("Press two to get the weather for another zip code.")
        self.response.write(str(response))

    def post(self):
        response = twiml.Response()

        weather = current_weather(self.request.get("FromZip", "94117"))

        digit_pressed = self.request.get("Digits")
        if digit_pressed == "1":
            response.say("The current weather is " + weather)
            response.redirect("/", method="GET")
        else:
            gather = response.gather(numDigits=5)
            gather.say("Please enter a 5 digit zip code.")
            self.response.write(str(response))

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
```

```
], debug=True)
```

First, we've specified that the action of the `<Gather>` should be an HTTP POST.

Next, we added some code to our `webapp2.RequestHandler` to respond to a POST request. Because our first `<Gather>` specifies a POST method and no action, the default action is the current URL (In this case, `/`). So, this code is what will get run after the first `<Gather>`.

We pull the value `digit_pressed` from `self.request.get("Digits")` which corresponds to what the caller pressed.

In the case that the `digit_pressed` was `"1"`, the behavior looks quite similar to our earlier example. We then redirect the user back to the beginning to the menu, so they can try again.

If the caller presses 2, we ask them for 5 more digits. We don't yet have the logic to process what to do with these 5 more digits, so nothing interesting will happen when they finish entering these digits.

Note: In this example, we use the `numDigits` attribute to know when the is done pressing digits. This works because we know we're looking for exactly one digit. If we didn't know this, we could use another attribute called `finishOnKey`.

Let's find out how to read the ZIP code.

```
import webapp2
from util import current_weather
from twilio import twiml

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = "application/xml"
        city = self.request.get("FromCity", "San Francisco")

        response = twiml.Response()
        gather = response.gather(method="POST", numDigits=1)
        gather.say("Press one for the weather in " + city)
        gather.say("Press two to get the weather for another zip code.")
        self.response.write(str(response))

    def post(self):
        response = twiml.Response()

        weather = current_weather(self.request.get("FromZip", "94117"))

        digit_pressed = self.request.get("Digits")
        if digit_pressed == "1":
            response.say("The current weather is " + weather)
            response.redirect("/", method="GET")
        else:
            gather = response.gather(action="/weather_for_zip", method="POST", numDigits=5)
            gather.say("Please enter a 5 digit zip code.")
            self.response.write(str(response))

class GetWeather(webapp2.RequestHandler):

    def post(self):
        response = twiml.Response()
```

```

        zipcode = self.request.get("Digits")
        weather = current_weather(zipcode)

        response.say("The current weather is " + weather)
        response.redirect("/", method="GET")

    self.response.write(str(response))

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/weather_for_zip', GetWeather),
], debug=True)

```

We've added a second `webapp2.RequestHandler` class. We also configure this handler to respond to the URL `/weather_for_zip` and changed the `<Handler>` to point to it.

When the caller has entered 5 digits, Twilio will do a POST request to `/weather_for_zip` with the digits pressed passed as the `Digits` argument. We use these digits to look up the weather, just as we did for the original app with the `FromZipCode` passed in by Twilio.

Now, deploy your application again, and try it out!

2.5 SMS Polling and Voting

One great use of SMS is to get large groups to vote on something easily. Nearly everyone in the audience will have a cell phone in their pocket - why not let them send in their votes over SMS?

We'll create a simple Twilio application to record and report votes via SMS.

2.5.1 Ballot Format

For this poll ballots don't need a format. Voters will simply text their choice to your Twilio number.

For example, to vote for Twilio, you'd text the following:

```
twilio
```

To see all the votes, we'll use the `twilio-python` helper library to fetch data from the Twilio REST API. Open `main.py` and add the following lines.

```

import webapp2
from twilio.rest import TwilioRestClient

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class HelloWorld(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
], debug=True)

```

We're creating a client to talk to the REST API. You'll need to replace the `ACCOUNT_SID` and `AUTH_TOKEN` with your account credentials. Your credentials are located at the top of your [account dashboard](#).

With the client created, we can now query the Twilio REST API for SMS messages.

```
import webapp2
from twilio.rest import TwilioRestClient

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        for msg in client.sms.messages.iter():
            self.response.write(msg.body + "<br>")

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)
```

To view your votes, open the Google App Engine Launcher. Click the **Run** button and then **Browse** button. In the URL bar, add /sms to the URL and reload.

This page will fail if you have multiple Twilio phone numbers. To fix this problem, we'll filter messages based on the **To** phone number. Replace **NUMBER** with one of your Twilio phone numbers. If you can't remember your number, you'll find them listed in the [Twilio account portal](#).

```
import webapp2
from twilio.rest import TwilioRestClient

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        for msg in client.sms.messages.iter(to="NUMBER"):
            self.response.write(msg.body + "<br>")

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)
```

Still, we're only seeing the contents of the messages.

2.5.2 Tallying Votes

In our election participants can only vote once, therefore each message should count for a single vote. We'll use a dictionary to keep track of votes.

Instead of just printing the message body we'll print the message body and the number of votes it received.

```

import webapp2
from twilio.rest import TwilioRestClient
from collections import defaultdict

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        votes = defaultdict(int)

        for msg in client.sms.messages.iter(to="NUMBER"):
            votes[msg.body] += 1

        for vote, total in votes.items():
            self.response.write("{} {}<br>".format(total, vote))

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)

```

We can now see tallies. However, this code is very brittle. Votes for foo and Foo won't count for the same thing. Let's normalize the message bodies so that similar votes count for the same option.

```

import webapp2
from twilio.rest import TwilioRestClient
from collections import defaultdict

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        votes = defaultdict(int)

        for msg in client.sms.messages.iter(to="NUMBER"):
            votes[msg.body.upper().strip()] += 1

        for vote, total in votes.items():
            self.response.write("{} {}<br>".format(total, vote))

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)

```

2.5.3 Preventing Cheaters

Cheaters never prosper. And currently they don't get caught either. Any person can vote any number of times. We'll keep track of every number that's already voted, only allowing them a single vote. To do this, phone numbers will be added to a set and checked before each vote is tallied.

```
import webapp2
from twilio.rest import TwilioRestClient
from collections import defaultdict

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        votes = defaultdict(int)
        voted = set()

        for msg in client.sms.messages.iter(to="NUMBER"):
            if msg.from_ in voted:
                continue

            votes[msg.body.upper().strip()] += 1
            voted.add(msg.from_)

        for vote, total in votes.items():
            self.response.write("{} {}\\n".format(total, vote))

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)
```

2.5.4 Graphing the Results

No election is complete without graphs. Let's take the results from the previous section and make some pretty graphs. We'll use the [Google Graph API](#) due to its simplicity and price (free).

```
import urllib
import webapp2
from twilio.rest import TwilioRestClient
from collections import defaultdict

client = TwilioRestClient("ACCOUNT_SID", "AUTH_TOKEN")

class SmsVoting(webapp2.RequestHandler):

    def get(self):
        votes = defaultdict(int)
        voted = set()

        for msg in client.sms.messages.iter(to="NUMBER"):
```



```
        if msg.from_ in voted:
            continue

        votes[msg.body.upper().strip()] += 1
        voted.add(msg.from_)

    url = "https://chart.googleapis.com/chart"

    options = {
        "cht": "pc",
        "chs": "500x200",
        "chd": "t:" + ",".join(map(str, votes.values())),
        "chl": "|" .join(votes.keys()),
    }

    image = ''.format(url, urllib.urlencode(options))
    self.response.write(image)

class HelloWorld(webapp2.RequestHandler):

    def get(self):
        self.response.write('Hello World!')

app = webapp2.WSGIApplication([
    ('/', HelloWorld),
    ('/sms', SmsVoting),
], debug=True)
```

2.5.5 Existing Solutions

[Wedgies](#) is a very similar concept built on top of Twilio, but questions are limited to two answers. Great for simple surveys, but not for elections.

UNLOCKING THE BOX

3.1 Radio Call In

In this workshop we'll be designing a radio call in application using Twilio's `<Queue>` functionality. While we'll be using a radio show as our target, this style of queue management can be used for any phone number where many people may call at the same time.

There will be two numbers, one for calls coming in and one for the DJ. The DJ will call in to connect to the waiting callers one at a time. Once the DJ is finished with a caller, he or she will press # to move onto the next caller.

3.1.1 Prerequisites

The next sections assume a working knowledge of Twilio. You should be familiar with TwiML, configuring Twilio phone numbers, and the Twilio application model.

Also, we assume you are comfortable writing web applications. For reference, we'll be developing the application along the way using Python and Google App Engine.

Make sure you have a second Twilio number.

3.1.2 Using the Twilio Helper Libraries

Though this workshop will assume use of Python and the `twilio-python` helper library, Twilio offers helper libraries for a large set of languages. If you aren't using Python, download the [helper library](#) for your language of choice. You'll need the library in the next section.

For the `twilio-python` helper library, you may find the [Queue API Reference](#) helpful for this workshop.

3.1.3 Using `<Queue>`

We'll need two Twilio phone numbers to work with Queue - one for the DJ to dequeue calls from, and one for the listener to call into.

First, we'll enqueue calls via TwiML. In the example below, we enqueue calls into a queue named `radio-callin-queue`. Note that queues are created on `<Enqueue>` if they do not already exist.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>You are being enqueued now.</Say>
  <Enqueue>radio-callin-queue</Enqueue>
</Response>
```

Here is an example App Engine application that serves the above TwiML.

```
import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):

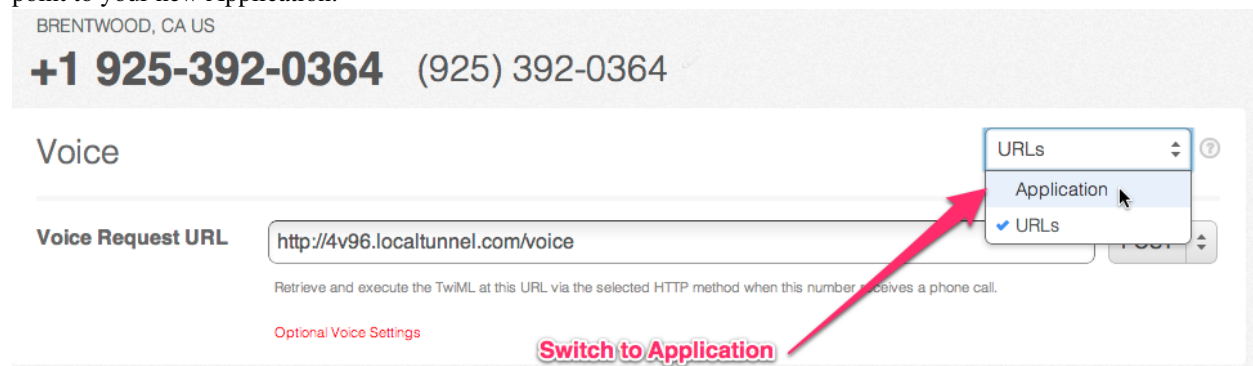
    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        resp = twiml.Response()
        resp.say("You are being enqueued now.")
        resp.enqueue("radio-callin-queue")
        self.response.write(str(resp))

app = webapp2.WSGIApplication([
    ('/twiml/enqueue', EnqueueHandler),
], debug=True)
```

We are going to use a TwiML Application to connect this TwiML with your listener queue number. We'll need to create an [Application](#) for the browser to call into. You can think of an Application kind of like a phone number; it's an entry point for incoming calls. Configure the Voice URL of your new Application to point to the TwiML above.

You want to connect your listener queue function to your Application. From the [Numbers](#) tab of your Dashboard, select the number you are going to use for your Listener Queue. Then from the dropdown select "Application", then point to your new Application.



Go ahead and try calling your number now. You should hear the Twilio default wait music.

We can spice up our TwiML endpoint by adding some custom wait music, using the `waitUrl` parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>You are being enqueued now.</Say>
  <Enqueue waitUrl="/twiml/wait" waitUrlMethod="GET">
    radio-callin-queue
  </Enqueue>
</Response>
```

Twilio will request `/twiml/wait` endpoint and process the TwiML there, which plays music. The `waitUrl` TwiML document only supports a subset of TwiML verbs, including `<Say>` and `<Play>`.

```
import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):
```

```

def get(self):
    self.response.headers['Content-Type'] = 'application/xml'

    resp = twiml.Response()
    resp.say("You are being enqueued now.")
    resp.enqueue("radio-callin-queue",
        waitUrl="/twiml/wait", waitUrlMethod="GET")
    self.response.write(str(resp))

app = webapp2.WSGIApplication([
    ('/twiml/enqueue', EnqueueHandler),
], debug=True)

```

The /twiml/wait endpoint will return TwiML that plays hold music for the queue.

```

<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Say>Please hold.</Say>
    <Play>http://com.twilio.sounds.music.s3.amazonaws.com/MARKOVICHAMP-Borghestral.mp3</Play>
    <Redirect/>
</Response>

```

We use the redirect verb at the bottom of the page so once the hold music has finished, Twilio will re-request the same URL and make sure the music doesn't stop.

You can use this Python snippet with AppEngine to output the TwiML above.

```

import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):

    def get(self):
        # Same as above

class WaitHandler(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        resp = twiml.Response()
        resp.say("Please hold.")
        resp.play("http://com.twilio.music.rock.s3.amazonaws.com/nickleus_-_"
            "original_guitar_song_200907251723.mp3")
        self.response.out.write(str(resp))

app = webapp2.WSGIApplication([
    ('/twiml/enqueue', EnqueueHandler),
    ('/twiml/wait', WaitHandler),
], debug=True)

```

Now your listener queue number should play hold music while callers are in the queue.

For the DJ dequeuing number, we use TwiML that bridges the current call to the queue. Note that <Dial>ing into a queue dequeues the front of the queue (the person who has been waiting longest), while the only way to get into a queue is by using the <Enqueue> verb.

```

<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Dial>

```

```
        <Queue>radio-callin-queue</Queue>
    </Dial>
</Response>
```

You will want to create a second Twilio Application for your DJ number, and configure that application's Voice URL to point to the TwiML above.

```
import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):

    def get(self):
        # Same as above

class WaitHandler(webapp2.RequestHandler):

    def get(self):
        # Same as above

class DequeueHandler(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        resp = twiml.Response()
        d = resp.dial()
        d.queue("radio-callin-queue")

        self.response.out.write(str(resp))

app = webapp2.WSGIApplication([
    ('/twiml/dequeue', DequeueHandler),
    ('/twiml/enqueue', EnqueueHandler),
    ('/twiml/wait', WaitHandler),
], debug=True)
```

Now, the DJ can call the DJ dequeuing number, and will automatically be connected to the first member on the queue.

By now, you may be wondering how to properly test this application. With two phone numbers, you need to fill up your queue with waiting callers. To help you fill your queue, we've created an application that will call a given number with fake callers, allowing you to easily simulate real users calling in. The application can be found at <http://queuetester.herokuapp.com/>.

3.1.4 Dynamic Queue Information

Twilio's Queue exposes dynamic information about the queue state that you can use to build rich applications. In this section, we'll move past static TwiML applications and start using the data Queue gives you to create dynamic TwiML through a web application.

We'll start by working on our hold music. Wouldn't it be cool if we could tell users where they were in the queue, how long they've been there, or even the average wait time for their queue? Twilio sends these parameters via POST data when invoking your application's waiting logic via HTTP.

Parameter	Description
QueuePosition	The current queue position for the enqueued call.
QueueSid	The SID of the Queue that the caller is in.
QueueTime	The time in seconds that the caller has been in the queue.
AvgQueueTime	The average amount of time the currently enqueued callers have been in the queue.
CurrentQueueSize	The current number of enqueued calls in this queue.

Using this information, we can inform our users what position they are in the queue and how long they can expect to wait before an answer.

Remember to change the `waitUrlMethod` from GET to POST now that we are using POST data for the `waitUrl`.

```
import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        resp = twiml.Response()
        resp.say("You are being enqueued now.")
        resp.enqueue("radio-callin-queue",
                    waitUrl="/twiml/wait", waitUrlMethod="POST")
        self.response.write(str(resp))

class WaitHandler(webapp2.RequestHandler):
    def post(self):
        response = twiml.Response()
        response.say("You are number %s in line." % self.request.get('QueuePosition'))
        response.say("You've been in line for %s seconds." % self.request.get('QueueTime'))
        response.say("The average wait time is currently %s seconds." % self.request.get('AvgQueueTime'))
        response.play("http://com.twilio.music.rock.s3.amazonaws.com/nickleus_-_original_guitar_song.mp3")
        self.response.out.write(str(response))

class DequeueHandler(webapp2.RequestHandler):

    def get(self):
        # Same as above

app = webapp2.WSGIApplication([
    ('/twiml/dequeue', DequeueHandler),
    ('/twiml/enqueue', EnqueueHandler),
    ('/twiml/wait', WaitHandler),
], debug=True)
```

You can also take advantage of similar information when a call is dequeued through the `action` parameter when enqueueing.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>You are being enqueued now.</Say>
  <Enqueue action="http://example.com/dequeue-logic">radio-callin-queue</Enqueue>
</Response>
```

Twilio will fetch the `action` URL and execute the TwiML received on the caller's end before he or she is bridged to the other call.

3.1.5 Handling Long Queue Times

We can use the `action` parameter to collect all sorts of useful metrics on the back end, or even issue hasty apologies for long queue wait times.

Let's try to implement some small features on our dequeue action call to let our users know we care. Using the `action URL parameters`, we can send an SMS apology if the wait time exceeded 30 seconds, or if their call was rejected from a full queue.

Here is some stub code that may help, if you are taking the Python / Google App Engine route.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>You are being enqueued now.</Say>
  <Enqueue action="http://example.com/dequeue-logic">radio-callin-queue</Enqueue>
</Response>
```

```
import webapp2
class DequeueLogic(webapp2.RequestHandler):
    def post(self):

        # ... FILL ME IN ...
        # res = self.request.get('QueueResult')

app = webapp2.WSGIApplication([('/dequeue-logic', DequeueLogic)], debug=True)
```

3.1.6 Closing Out the Queue

Unfortunately, all good things must come to an end. It's time for our radio show to close down until next time - but what about the people still on the waiting queue?

We can use `Queue` and `Member` REST API resources to programmatically look at all of our account's queues and active members on those queues.

Let's write a quick HTTP endpoint that will loop through our queue members, and dequeue each of them with a thank you message.

Then, we can iterate over its members and dequeue with some static thank you TwiML. Try it yourself! Hint: issuing an HTTP POST to a `Member` instance will dequeue that member.

```
import urllib
import webapp2

from twilio import TwilioRestClient

message_url = ('http://twimlets.com/message?' +
               urllib.quote_plus('Sorry, the queue is now closed.'))

class DequeueEveryone(webapp2.RequestHandler):
    def get(self):
        client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)
        # Should only have one queue, but let's make sure.
        for queue in client.queues.list():
            for member in queue.queue_members.list():
                queue_members.dequeue(message_url, member.sid)
```

Finally, we can delete the queue using a REST API call.


```
my_queue.delete()
```

3.1.7 Advanced Features

That is the end of the content for this tutorial. If you still have some time, try implementing some of these advanced features:

- Allowing the callers being dequeued to record a message for the DJs to listen to at the beginning of the next show.
- other features

3.2 Into the Browser

3.2.1 Using Twilio Client

Using Twilio Client, we can hook in to the full power of the Twilio API from your Web Browser. This includes the ability to make and receive phone calls, opening up the world of telephony to your dynamic web applications.

Let's try writing a web app that is capable of answering phone calls in a Twilio <Queue>. This way, our Radio DJ won't be required to use their personal phone when answering queues.

3.2.2 How Client Works

This is what an outbound Client call looks like:

1. Your server creates a [Capability Token](#) with the Application Sid you would like to use to handle outbound calls.
2. A user triggers a `connect()` action in the `twilio.js` Javascript library.
3. Twilio looks up the Application Sid for the Client, and retrieves the Voice URL for that application.
4. Twilio makes an HTTP request to the Voice URL and plays the TwiML it retrieves back to the user's browser.

To connect an inbound call to your Client browser, generate a [Capability Token](#) that allows incoming connections. Then return this TwiML in response to an inbound call:

```
<Response>
  <Dial>
    <Client>client-name</Client>
  </Dial>
</Response>
```

3.2.3 Creating an Application

We are going to reuse the application we created in the previous example. Set the Voice Request URL to a new endpoint for the TwiML we want to be executed when the DJ's browser connects (this should be the same URL as the DJ's dial-in number).

3.2.4 Generating a Token

Since Twilio Client applications are being run on the browser, we need to grant the end user temporary privileges on our Twilio Account. This is the job of [Capability Tokens](#). Capability Tokens allow us to lock down access to what we want the end user's session to be able to do. For our needs we only need to add access to making an outgoing connection to our new Application.

Here is the function we'll use for generating the Capability Token.

```
from twilio.util import TwilioCapability

def generate_token(account_sid, auth_token, application_sid):
    capability = TwilioCapability(account_sid, auth_token)
    # Allow access to the Call-in ApplicationSid we created
    capability.allow_client_outgoing(application_sid)
    return capability.generate()
```

3.2.5 Answering Queues in the Browser

The first thing we'll need to build is a web interface. Let's start by adding a new AppEngine RequestHandler into main.py.

We have included some helper functions for generating [Capability Tokens](#) and rendering templates on AppEngine. Those are imported from util.py.

```
from util import generate_token, render_template

class IndexPage(webapp2.RequestHandler):

    def get(self):
        params = {
            "token": generate_token(ACCOUNT_SID, AUTH_TOKEN, APP_SID)
        }
        self.response.out.write(render_template("index.html", params))
```

Here is the index.html file we are rendering.

```
<!DOCTYPE html>
<html>
<head>
  <title>Radio Call-In Dashboard</title>
  <link rel="shortcut icon" href="//static0.twilio.com/packages/favicons/img/Twilio_64.png" />
  <link rel="apple-touch-icon" href="//static1.twilio.com/packages/favicons/img/Twilio_57.png" />
  <link rel="apple-touch-icon" sizes="72x72" href="//static0.twilio.com/packages/favicons/img/Twilio_57.png" />
  <link rel="apple-touch-icon" sizes="114x114" href="//static0.twilio.com/packages/favicons/img/Twilio_114.png" />
  <script type="text/javascript"
    src="http://static.twilio.com/libs/twiliojs/1.0/twilio.min.js"></script>
  <script type="text/javascript"
    src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
  </script>
  <link href="http://static0.twilio.com/packages/quickstart/client.css"
    type="text/css" rel="stylesheet" />
  <script type="text/javascript">

    Twilio.Device.setup("{{ token }}");

    Twilio.Device.ready(function (device) {
      $("#log").text("Ready");
```

```

});

Twilio.Device.error(function (error) {
  $("#log").text("Error: " + error.message);
});

Twilio.Device.connect(function (conn) {
  $("#log").text("Successfully established call");
});

function call() {
  Twilio.Device.connect();
}
</script>
</head>
<body>
  <button class="call" onclick="call();">
    Connect to Queue
  </button>

  <div id="log">Loading pigeons...</div>
</body>
</html>

```

There are two important lines in the Javascript that make this work:

```
Twilio.Device.setup("{ token }");
```

The above line of code calls `Twilio.Device.setup` and uses our templating engine to pass in a valid Capability Token. When `setup` finishes, the function passed into `Twilio.Device.ready` will fire to let the browser know that Twilio has initialized access to the microphone, speakers, and we've started listening for incoming calls (if applicable).

```
function call() {
  Twilio.Device.connect();
}
```

This code defines a new function called `call` that just wraps `Twilio.Device.connect`, which initiates an outgoing call to the Application we created earlier. In this case, calling `call()` should execute the TwiML below. We've made a small change so that the DJ can press "*" to end the current call, and dial the next person in the queue.

```

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial hangupOnStar="true">
    <Queue>radio-callin-queue</Queue>
  </Dial>
  <Redirect method="GET"></Redirect>
</Response>

```

Change your application's Voice URL so it serves this TwiML when dialed.

3.2.6 Getting the Next Caller From the <Queue>

We want to make it easy to hangup the current call and move to the next one by pressing the "*" key on the phone. Twilio Client has a feature for sending DTMF tones (the tone when you press "*" on your phone) programmatically.

First, we need to hold on to the response of `Twilio.Device.connect()` so let's add a global variable called `connection` and have every `call()` command set it. Replace the existing `call` function with something like

this:

```
var connection = null;
function call() {
    connection = Twilio.Device.connect();
}
```

Now, we can add a new function, called `next()`:

```
function next() {
    if (connection) {
        connection.sendDigits("*");
    }
}
```

Because we added a *hangupOnStar* attribute to our TwiML, sending a “*” symbol via DTMF tone will hang up on the current caller, and connect the browser to the next caller.

Now we just need to add another button to trigger the hangup.

```
<button class="hangup" onclick="next();" >
    Next Caller
</button>
```

3.2.7 Adding UI To Display the Queue

Let’s add a feature where we can see a visualization of the queue. We’ll add a new queue status endpoint, which will return the current queue status as JSON.

```
import json
from twilio.rest import TwilioRestClient

class QueueStatusPage(webapp2.RequestHandler):

    def get(self):
        client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)

        for queue in client.queues.list():
            if queue.friendly_name == 'radio-callin-queue':
                self.response.out.write(json.dumps({
                    'current_size': queue.current_size,
                    'average_wait_time': queue.average_wait_time,
                }))
                return

        self.abort(404)
```

Add this `QueueStatusPage` into the `WSGIApplication`’s routing map as `/queue-status`. Now we need some HTML for the status, and Javascript to poll the state of the queue and update the UI.

Add this HTML:

```
<div style="width: 500px; font-family: sans-serif; text-align: left; margin: 0 auto;">
    <h2>Queue Status</h2>
    <ul>
        <li>Current size: <span id="current-size">0</span>
        <li>Average wait time: <span id="average-wait-time">0</span>
    </ul>
```

```
<a href="javascript:void(0)" onclick="getQueueStatistics();">Refresh</a>
</div>
```

And this Javascript function to fetch the latest queue status and insert it into the page.

```
var getQueueStatistics = function() {
  $.getJSON("/queue-status", function(result) {
    $("#current-size").text(result.current_size);
    $("#average-wait-time").text(result.average_wait_time);
  });
};

// run the queue fetcher once on page load
$(function() {
  getQueueStatistics();
});
```

3.2.8 Advanced Features

That is the end of the content for this tutorial. If you still have some time, try implementing some of these advanced features:

- Add a chart showing the wait time of each queue participant.
- Allow users to call in to the DJ hotline using their browser.
- Add a “whisper” URL to play instructions to the DJ before her call connects.

3.3 Extending Radio Call In

Let’s add some additional features to our current application. Any of these additions are also suitable for call centers.

3.3.1 SMS Feedback

We’ve set up a basic call-in line. Let’s say we want to keep the dialogue going with our callers a little more. We are going to send an SMS to callers after they get off the line.

Sending an SMS From a Call

We want to send an SMS after the DJ call is complete. Let’s modify our customer call in TwiML slightly to request a new endpoint after the call connects.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Enqueue action="/sms" waitUrl="/wait-loop">radio-callin-queue</Enqueue>
</Response>
```

After the DJ disconnects from the call, Twilio will make a POST request to the `/sms` endpoint. Let’s set up that endpoint to send an SMS to the caller.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Sms>Thanks for calling the DJ hotline! Do you have any feedback for us?</Sms>
</Response>
```

This will send a text asking for feedback after the call completes.

You can retrieve users responses from the Twilio API. Add a server side endpoint to fetch your 50 most recent inbound text messages.

```
import json
from twilio import TwilioRestClient

class RetrieveSMSEndpoint(webapp2.RequestHandler):

    def get(self):
        client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)
        # Note direction=inbound filter, or you will show all of your
        # outbound messages in this list as well.

        params = {
            'msgs': client.sms.messages.list(direction='inbound'),
        }
        self.response.out.write(render_template('messages.html', params))
```

Add this RetrieveSMSEndpoint into the WSGIApplication's routing map as /inbound-sms. The messages.html template prints out the feedback you've received.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Radio Call-In Dashboard Feedback</title>
    <link href="http://static0.twilio.com/packages/quickstart/client.css"
          type="text/css" rel="stylesheet" />
  </head>
  <body>
    <h1>Feedback</h1>
    <ul>
      {% for msg in msgs %}
      <li>{{ msg.body }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Now, you'll be able to see when customers send you feedback on your call-in line.

Advanced Features

Try implementing some of these advanced features:

- Add a server-side endpoint to reply to users directly from your dashboard.
- Send yourself an email whenever someone new writes in.
- Add paging to your SMS list - eg a "More" button which will fetch the next 50 SMS messages from your list.
- Add a way to hide SMS messages you've seen/replied to already.

3.3.2 Call Recording

To help our DJ improve his manners on the phone, let's record all of the incoming calls to our Queue.

Many states have laws about letting the party know that they are being recorded. So let's add a short message before the call, letting the calling party know that we are recording the call.

We need to alter the TwiML that plays on the DJ's side to [add a url attribute](#). This URL will hold TwiML telling the caller they are about to be recorded. We're also going to add `record="True"` to the Dial verb, to record the call.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Dial record="true" hangupOnStar="true">
    <Queue url="/record-message">radio-callin-queue</Queue>
  </Dial>
  <Redirect></Redirect>
</Response>
```

Then at the `/record-message` route, place the following TwiML:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>This call is being recorded.</Say>
</Response>
```

That's it! Now all of your incoming calls will be recorded. To listen to the recorded calls, go to the [Recordings page](#) of your Twilio Dashboard.

Advanced Features

Try implementing some of these advanced features:

- Listen to your recordings straight from your Dashboard.
- Export your recordings to a private S3 folder, and delete the Twilio copies of the recording.
- Set up monitoring of your application, so that you receive notifications if your application becomes unreachable.
- Set up HTTP Authentication so that Twilio must authenticate before reading TwiML from your server, and other browsers will receive a 401 Forbidden message.

3.3.3 Routing Calls to Voicemail

Phone numbers are available all of the time, but your agents might not be. We don't want to put people on hold if there's no human available.

Instead, if our call-in hotline is overwhelmed, we'll give people the option to leave a voicemail if there are currently long hold times for a real agent.

Let's take a look at what that logic would look like.

```
import webapp2
from twilio import twiml
from twilio.rest import TwilioRestClient

class EnqueueHandler(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)

        resp = twiml.Response()
```

```
for queue in client.queues.list():
    if queue.friendly_name == 'radio-callin-queue':

        if queue.average_wait_time < 5: # Wait is less than five minutes
            continue

        g = resp.gather(num_digits=1, timeout=10, action="/voicemail")
        g.say("The wait is {} minutes.".format(queue.average_wait_time))
        g.say("Press 1 to leave a voicemail, "
              "or stay on the line for an agent")

    resp.enqueue("radio-callin-queue",
                 waitUrl="/twiml/wait", waitMethod="GET")
    self.response.write(str(resp))

class WaitHandler(webapp2.RequestHandler):
    # Rest of the file is the same as the previous pages...
```

If the average queue wait time is higher than some threshold, we listen for a key input, and redirect people to voicemail if they press a key. If the user does not press a key, they'll just fall through to the <Enqueue> verb at the bottom of the handler.

Otherwise, the queue wait times are short, so we place people in the call queue.

We need to add a new handler for our voicemail endpoint. Set up the following route to listen at /voicemail.

```
import webapp2
from twilio import twiml

class EnqueueHandler(webapp2.RequestHandler):
    # Same as above..

class VoicemailHandler(webapp2.RequestHandler):

    def get(self):
        """ GET /voicemail """
        self.response.headers['Content-Type'] = 'application/xml'

        resp = twiml.Response()
        resp.say("Please leave a message after the tone.")
        resp.record()
        self.response.write(str(resp))

class WaitHandler(webapp2.RequestHandler):
    # Rest of the file is the same as the previous pages...
```

Try it out! You should be able to retrieve these recordings from your [Twilio Dashboard](#).

It's Closing Time

If the queue is closed, we redirect straight to voicemail.

```
import webapp2
from twilio import twiml
from twilio.rest import TwilioRestClient
```



```

from datetime import datetime

def queue_closed(opening_hour=9, closing_hour=17):
    now = datetime.now()
    # Check if current time is before opening or after closing
    return now.hour < opening_hour or now.hour > closing_hour

class EnqueueHandler(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'application/xml'

        client = TwilioRestClient(ACCOUNT_SID, AUTH_TOKEN)

        resp = twiml.Response()

        for queue in client.queues.list():
            if queue.friendly_name == 'radio-callin-queue':

                if queue.average_wait_time < 5: # Wait is less than five minutes
                    continue

                g = resp.gather(num_digits=1, timeout=10, action="/voicemail")
                g.say("The wait is {} minutes.".format(queue.average_wait_time))
                g.say("Press 1 to leave a voicemail, "
                    "or stay on the line for an agent")

            if queue_closed():
                resp.say("The queue is currently closed. Please stay "
                    "on the line to leave a voicemail.")
                resp.redirect('/voicemail')
            else:
                resp.enqueue("radio-callin-queue",
                    waitUrl="/twiml/wait", waitMethod="GET")

        self.response.write(str(resp))

class WaitHandler(webapp2.RequestHandler):
    # Rest of the file is the same as the previous pages...

```

Our users will now hear a helpful message when the queue is closed, instead of waiting around for an agent that will never pick up

Advanced Features

That's the end of the content for this tutorial. If you still have some time, try implementing some of these advanced features:

- Send an email to yourself when someone leaves a new recording.
- Write a unit test to check the logic in your controller.

3.4 Best Practices for Developing On Twilio

Here's a summary of a number of best practices we've found that work well in the development of your Twilio applications.

3.4.1 Using TwiML

This section covers pro tips related to handling inbound Twilio requests and responding to them with [TwiML](#).

- Use a [helper library](#) to generate your TwiML and avoid simple typo errors.
- Make use of [Fallback URLs](#) for phone numbers and TwiML apps to prevent your users from hearing the dreaded “We’re sorry - an application error has occurred.”
- If you’re looking to maintain any state around calls, be sure to leverage [Status Callback URLs](#) to get asynchronous notifications of completed calls.

3.4.2 Using the REST API

Here we cover a few helpful tips for using the [REST API](#) for outbound Twilio requests like placing calls or sending text messages.

- Make use of the exceptions available in your helper library to prevent malformed phone numbers or missing permissions from causing fatal errors. Try/catching all outbound REST requests against these exceptions give you an easy way to handle errors gracefully and log for debugging later.
- If making a large number of requests - like when sending text messages to a group of contacts - make use of a task queue like [Celery](#) to send asynchronously.
- Use the new [Usage API](#) to reduce API calls for summary statistics on Twilio usage.

3.4.3 Using Twilio Client

[Twilio Client](#) for JavaScript, iOS and Android are excellent ways to stay in touch with your users.

- Be sure to include a visual cue for first time users to click “Allow” in the pop-up permissions dialog.
- Set your [token expiration](#) to a value that makes sense for your use case. By default, this is an hour.
- Use the [parameters](#) property to surface important details to your users like who is calling and what is dialed.

3.4.4 Security

Security is critical to telephony applications - here’s some tips on using Twilio safely.

- Never bundle your AccountSid and AuthToken in a client-side application, even if it is compiled.
- Always generate Twilio Client capability tokens server-side.
- Use [Digest Authentication](#) and SSL in concert for your TwiML URLs to make Twilio authenticate with your web server.
- Use [Request Validation](#) to further confirm that requests are legitimately coming from Twilio.

3.4.5 Testing

Testing your apps before you go into production is always wise. Here are a few tips to make sure your tests work well.

- If mocking the Twilio REST Client, be sure mock the *resource* instead of the client itself for best effect.
- Simulate Twilio in your test web client by matching the parameters found in a [Twilio request](#).
- Test early, test often. It’s good for you!