CHAPTER 24

Communicating with Web APIs

Mobile technology and the ubiquitous nature of the Web have changed the world we live in. You can now sit in the park and do your banking, search Amazon.com to find reviews of the book you're reading, and check Twitter to see what people in every other park in the world are thinking about. Mobile phones have moved well past just calling and texting—now you have instant access to the world's data, too.

You can use your phone's browser to reach the Web, but often the small screen and limited speed of a mobile device can make this problematic. Custom apps, specially designed to pull in small chunks of particularly suitable information from the Web, can provide a more attractive alternative to the mobile browser.

In this chapter, we'll take a broader look at apps that source information from the Web. You'll start by creating an app that asks a website to generate a bar chart (image) of a game player's scores for display. Then we'll discuss how `TinyWebDB` *can be used to access any type of data (not just images) from the Web, and we'll provide a sample that accesses stock data from Yahoo! Finance. Finally, we'll discuss how you can create your own web information sources that can be used by App Inventor apps.*

Creativity is about remixing the world, combining (*mashing*) old ideas and content in interesting new ways. Eminem popularized the music *mashup* when he set his Slim Shady vocal over AC/DC and Vanilla Ice tracks. This kind of "sampling" is now common, and numerous artists—including Girl Talk and Negativland—focus primarily on creating new tracks from mashing old content.

The web and mobile world are no different: websites and apps remix content from various data sources, and most sites are now designed with such *interoperability* in mind. An illustrative example of a web mashup is Housing Maps (http://*www.housingmaps.com*), pictured in Figure 24-1, which takes apartment rental information from Craigslist (*http://www.craigslist.org*) and mashes it with the Google Maps API.

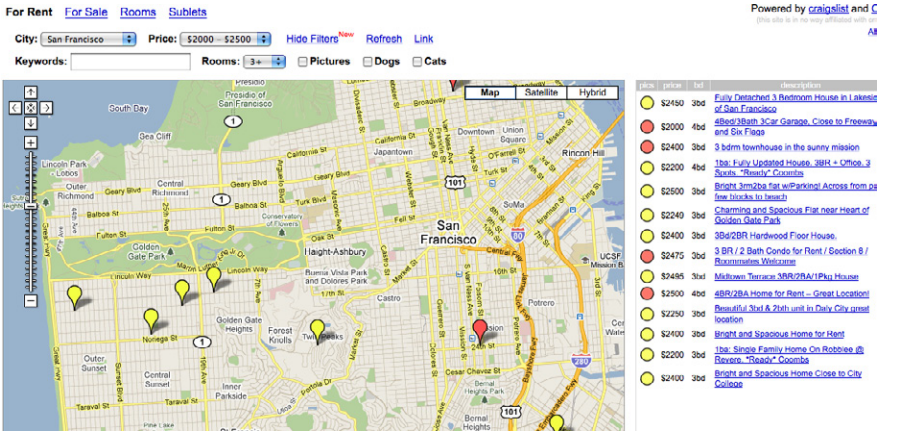

Figure 24-1. Housing Maps mashes information from Craigslist and Google Maps

Mashups like Housing Maps are possible because services like Google Maps provide both a website and a corresponding *web service API*. We humans visit *http://maps.google.com/* in a browser, but apps like Housing Maps communicate *machine to machine* with the Google Maps API. Mashups process the data, combine it with data from other sites (e.g., Craigslist), and then present it in new and interesting ways.

Just about every popular website now provides this alternative, machine-to-machine access. The program providing the data is called a *web service*, and the protocol for how a *client* app should communicate with the service is called an *application programmer interface*, or API. In practice, the term *API* is used to refer to the web service as well.

The Amazon Web Service (AWS) was one of the first web services, as Amazon realized that opening its data for use by third-party entities would eventually lead to more books being sold. When Facebook launched its API in 2007, many people raised their eyebrows. Facebook's data isn't book advertisements, so why should it let other apps "steal" that data and potentially draw many users away from the Facebook site (and its advertisements!). But its openness led Facebook toward becoming a *platform* instead of just a site—meaning that other programs, like FarmVille, could build on and tap into Facebook's functionality—and no one can argue with its success today. By the time Twitter launched in 2009, API access was an expectation, not a novelty, and Twitter acted accordingly. Now, as shown in Figure 24-2, most websites offer both an API and a human interface.

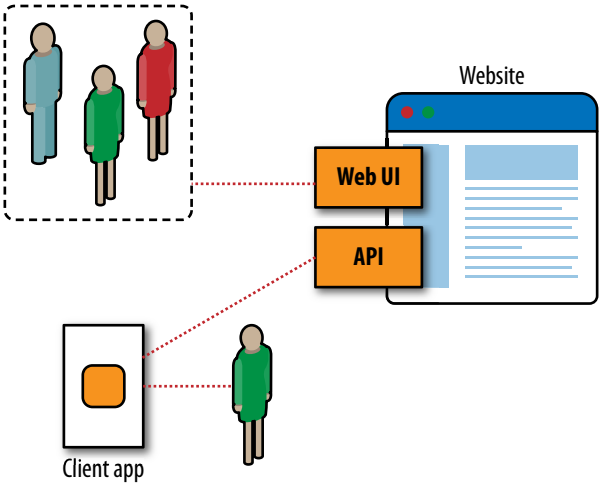

Figure 24-2. Most websites provide both a human interface and an API for client apps

So the Web is one thing to us average humans—a collection of sites to visit. To programmers, it is the world's largest and most diverse database of information. Machine-to-machine communication is now poised to outpace human–machine communication on the Web!

Talking to Web APIs That Generate Images

As we saw in Chapter 13 ("Amazon at the Bookstore"), most APIs accept requests in the form of a URL and return data (typically in standard formats like XML, or Extensible Markup Language; and JSON, JavaScript Object Notation). For these APIs, you use the `TinyWebDB` component to communicate, a topic we'll discuss in greater detail later in the chapter.

Some APIs, however, don't return data; they return a picture. In this section, we'll discuss how you can communicate with these image-generating APIs in order to extend App Inventor's user interface capabilities.

The Google Chart API is such a service. Your app can send it some data within a URL, and it will send back a chart that you can display in your app. The service creates many types of charts, including bar charts, pie charts, maps, and Venn diagrams. The Chart API is a great example of an interoperable web service whose purpose is to enhance the capabilities of other sites. Since App Inventor doesn't provide much in terms of visualization components, the ability to leverage a service like the Chart API is crucial.

The first thing to do is to understand the format of the URL you should send to the API. If you go to the Google Chart API site (*http://code.google.com/apis/chart*), you will see the overview shown in Figure 24-3.

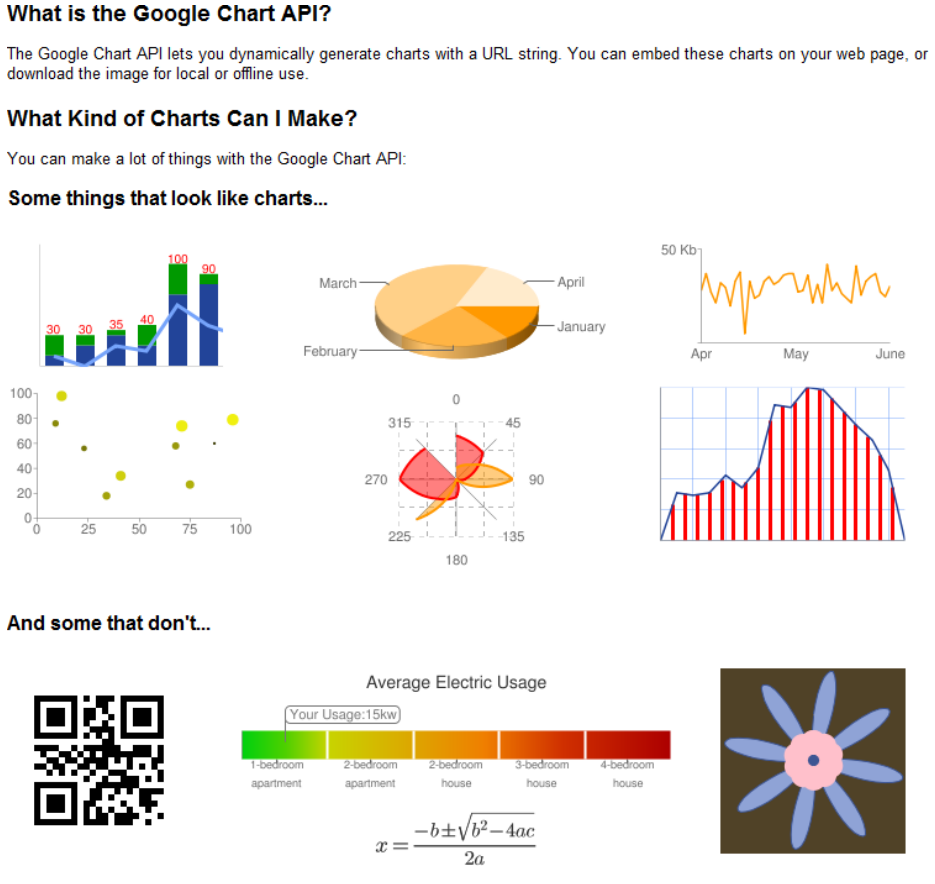

Figure 24-3. The Google Chart API generates numerous types of charts

The site includes complete documentation and a wizard to interactively create charts and explore how to build the URLs. The wizard is especially helpful, because you can use a form to specify the kind of chart you want and then examine the URL that the wizard generates to reverse-engineer what you want to send it for your specific data.

Go ahead and play around with the website and the wizard and build some charts, and then take a look at the details of the URLs used to build them. For example, if you enter the following URL in a browser:

```
http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=300x225&chco=
A2C180&chtt=Vertical+bar+chart&chd=t:10,50,60,80,40,60,30
```

you'll get the chart shown in Figure 24-4.

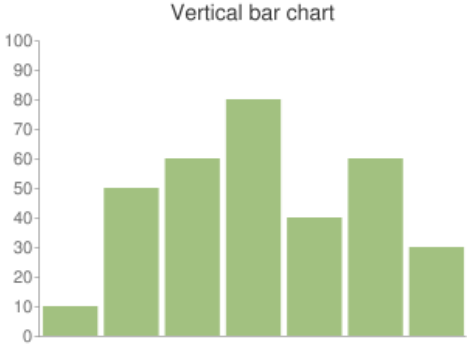

Figure 24-4. Google's Chart API generates this chart from the URL

To understand the rather complicated-looking URL specified previously, you need to understand how URLs work. In your browsing experience, you've probably noticed URLs with question marks (?) and ampersands (&). The ? character specifies that the first parameter of the URL request is coming. The & character then separates each succeeding parameter. Each parameter has a name, an equals sign, and a value. So the sample URL is calling the Chart API (*http://chart.apis.google.com/chart*) with the parameters listed in Table 24-1.

Table 24-1. The Chart API utilizes a URL with these parameters

Parameter	Value	Meaning
cht	bvg	The chart type is bar, vertical, grouped.
chxt	y	Show the numbers on the *y*-axis.
chbh	a	Width/spacing is automatic.
chs	300x225	The size of the chart in pixels.
chco	A2C180	The bar colors in hexadecimal notation.
chd	t:10,50,60,80,40,60,30	The data of the chart, with basic text format (t).
chtt	Vertical+bar+chart	The chart title; a + character indicates a space.

By modifying the parameters, you can generate various graphs. For more information on the types of graphs you can create, check out the API documentation at *http://code.google.com/apis/chart/index.html*.

Setting the Image.Picture Property to a Chart API

Now you know how to type the sample Chart API URL into a web browser to see the chart that is generated. To get a chart to appear in an app, you'll need to set the `Picture` property of an `Image` component to that same URL. To explore this, do the following:

1. Create a new app with a screen title of "Sample Chart App".
2. Add an `Image` component with a `Width` of "Fill parent" and `Height` of 300.
3. Set the `Image.Picture` property to the sample URL (*http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=300x225&chco=A2C180&chtt=Vertical+bar+chart&chd=t:10,50,60,80,40,60,30*). You can't set the property in the Component Designer, as it only allows you to upload a file. But you can set it in the Blocks Editor, as shown in Figure 24-5, so add a **Screen.Initialize** event handler and set the `Image.Picture` property there (note that you can't copy and paste on some machines, so you'll have to type out the full URL).

Figure 24-5. When the app starts, it sets the picture to a chart returned from the Chart API URL

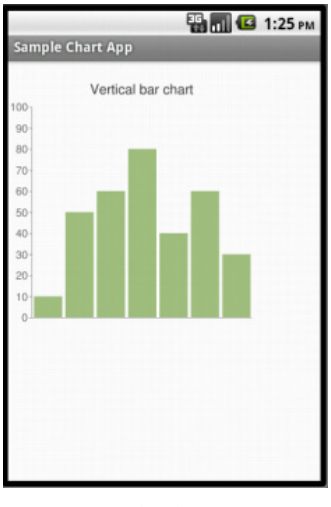

Figure 24-6. The chart in an app

You should see the image in Figure 24-6 on your phone or emulator.

Building a Chart API URL Dynamically

The preceding example shows how you can get a generated chart in your app, but it uses a URL with fixed data (10,50,60,80,40,60,30). Generally, you'll show *dynamic* data in your chart—that is, data stored in your variables. For example, in a game app, you might show the user's previous scores, which are stored in a variable `Scores`.

To create such a dynamic chart, you must *build* the URL for the Chart API and load your variable data into it. In the sample URL, the data for the chart is fixed and specified in the parameter `chd` (`chd` stands for chart data):

```
chd=t:10,50,60,80,40,60,30
```

To build your scores chart dynamically, you'll start with the fixed part, `chd=t:`, and then step through the `Scores` list, concatenating each score to the text (along with a comma). Figure 24-7 shows a complete solution.

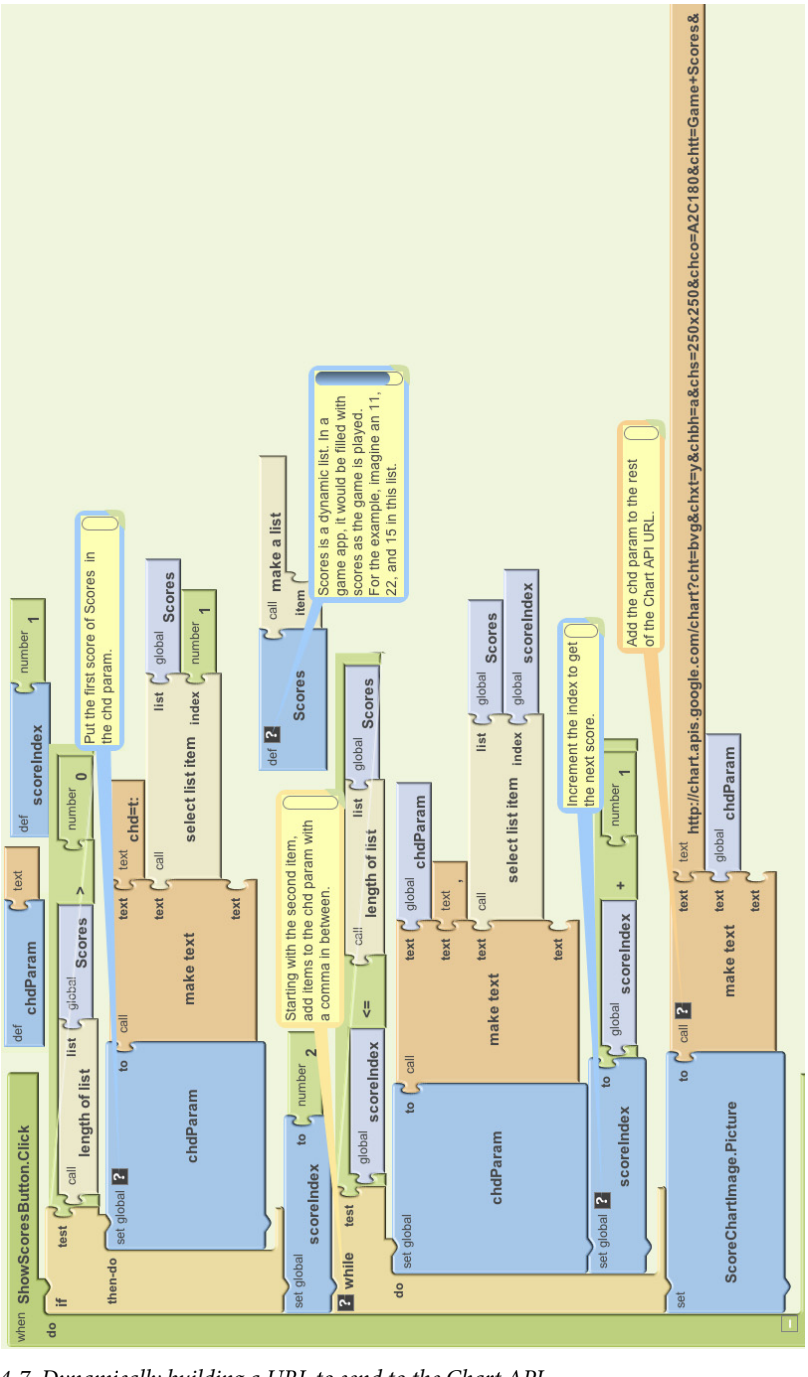

Figure 24-7. Dynamically building a URL to send to the Chart API

Let's examine the blocks more closely, because there's a lot going on in here, much of which we've covered in previous chapters. To understand such code, it's important to envision some real data. So let's assume the user has played three games in this app and that the variable `Scores` has three items: 11, 22, and 15.

The blocks in Figure 24-8 define a variable `chdParam` to store the part of the URL that will contain the `chd` data. The first row of blocks initializes the text of the `chdParam` from the list of `Scores`.

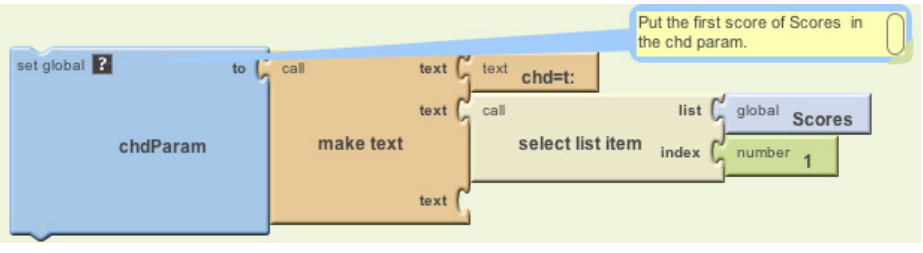

Figure 24-8. Beginning the chd parameter with "chd=t:" and the first score

After these blocks are performed, `chdParam` will contain `chd=t:11`, as 11 is the first value of the `Scores` list.

The next set of blocks, shown in Figure 24-9, adds the rest of the scores to the `chdParam`.

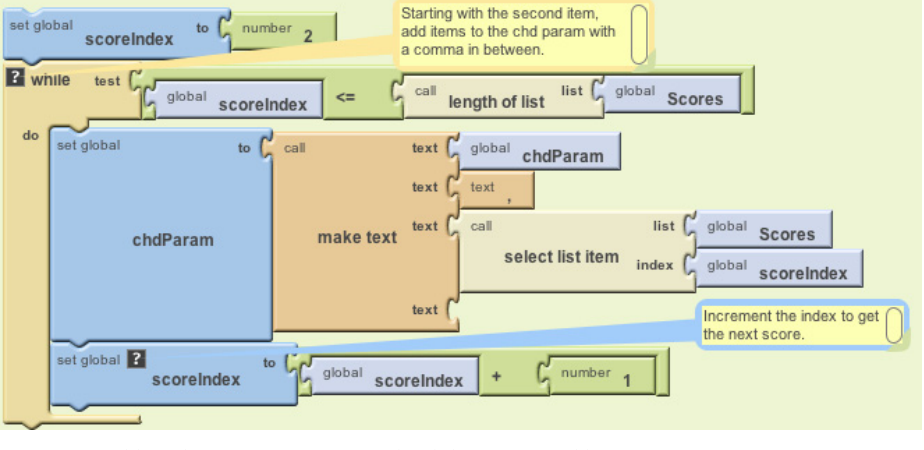

Figure 24-9. Adding the successive scores to the chdParam variable

We use a **while** block in this example instead of a **foreach** because **foreach** only allows you to do the same thing to each item. Here, we want to insert commas before the second item and any items that come after it (but not the first). With **while**, we can

put the first item in (Figure 24-8) and then loop starting from the second item, always inserting a comma *before* the item (make sure not to put a space afterward). For more information on **while** and **foreach**, see Chapter 20.

An index is used to keep track of where we are in the `Scores` list. On each iteration, **make text** adds a comma and the next item in `Scores`. After these blocks are performed, the `chdParam` will contain `chd=t:11,22,15`. We have built the `chd` parameter dynamically! (And we've also built it so that if more scores are added beyond these first three, it will still work.)

The blocks' last job is to concatenate the `chd` parameter with the rest of the Chart API URL, as shown in Figure 24-10.

Figure 24-10. Setting the picture to the full URL, including the chd parameter just built

The blocks set the `ScoreChartImage.Picture` property to this full URL: *http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=300x225&chco=A2C180&chtt=Game+Scores&chd=t:11,22,15*. Your users will see something similar to what is shown in Figure 24-11.

You could add such a display to any game or app by adding blocks similar to this example. You could also talk to other APIs that generate images and bring those into your app as well. The key is that App Inventor provides a useful connection to the Web through the `Image` component.

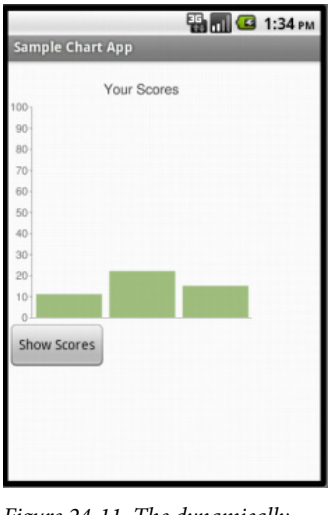

Figure 24-11. The dynamically generated chart

Talking to Web Data APIs

The Google Chart API is a web API that responds to requests by returning a picture. More commonly, APIs will return data that an app can process and use however it wants. The "Amazon at the Bookstore" app in Chapter 13, for instance, returns data in the form of a list of books, with each book including a title, current lowest price, and ISBN.

To talk to an API from an App Inventor app, you don't need to build a URL, as we did with the Chart API example. Instead, you query the API much like you would a web database (Chapter 22): just send your request as the tag to the **TinyWebDB.GetValue** block. The `TinyWebDB` component takes care of actually generating the URL that you send to the API.

`TinyWebDB` does not provide access to all APIs, even those that return a standard data format such as RSS. `TinyWebDB` can only talk to web services for which an App Inventor "wrapper" service, with a particular protocol, has been created. Fortunately, a number of these services have been created already, and more will soon follow. You can find some of these at *http://appinventorapi.com*.

Exploring the Web Interface of an API

In this section, you'll learn how to use `TinyWebDB` to bring in stock price data from the App Inventor–compliant API at *http://yahoostocks.appspot.com.* If you go to the site, you'll see the web (human) interface of the service pictured in Figure 24-12.

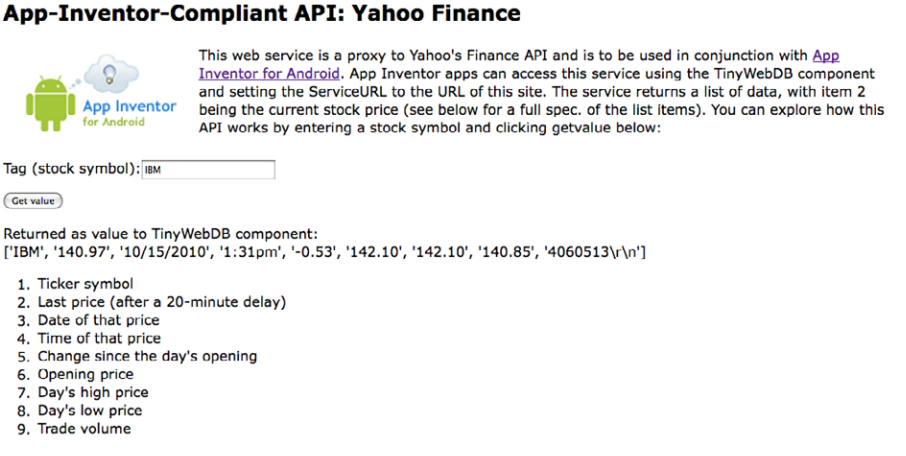

Figure 24-12. The web interface of the App Inventor–compliant Yahoo! Finance API

Try entering "IBM" or some other stock symbol into the Tag input box. The web page returns current stock information as a list, with each item representing a different piece of information, as described in the numerical listing further down the page.

Note that this web interface isn't meant as a new or interesting way to find stock information; its sole purpose is to allow *programmers* to explore the API for communicating with the underlying machine-to-machine web service.

Accessing the API Through TinyWebDB

The first step in creating an app that talks to the preceding web service is to drag a `TinyWebDB` component into the Component Designer. There is only one property associated with `TinyWebDB`, its `ServiceURL`, shown in Figure 24-13. By default, it is set to a default web database, *http://appinvtinywebdb.appspot.com*. Since we want to instead access the Yahoo! Stocks API, set this property to *http://yahoostocks.appspot.com*, the same URL you entered at the browser address bar earlier to see the web page interface.

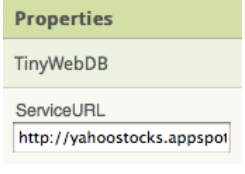

Figure 24-13. The ServiceURL is set to http://yahoostocks.appspot.com

The next step is to make a **TinyWebDB.GetValue** call to request data from the site. You might do this in response to the user entering a stock symbol and clicking a Submit button in your app's UI, or you might do it in the **Screen.Initialize** event to bring in information about a particular stock right when the app is opened. In any case, when you call **GetValue**, you should set the tag to a stock symbol, as illustrated in Figure 24-14, just as you did at the *http://yahoostocks.appspot.com* website.

Figure 24-14. Requesting stock information

As we covered in Chapter 10's MakeQuiz app and in Chapter 22's discussion of databases, the `TinyWebDB` communication is *asynchronous*: your app requests the data with **TinyWebDB.GetValue** and then goes about its business. You must provide a separate event handler, **TinyWebDB.GotValue**, to program the steps the app should take when the data actually comes back from the web service. From our examination of the human interface of *http://yahoostocks.appspot.com*, we learned that the data returned from **GetValue** is a list, with particular list items representing different data about the stock (e.g., item 2 is the latest price).

A client app can use some or all of the data the service provides. For example, if you just wanted to display the current stock price and its change since the day's opening, you might configure blocks as shown in Figure 24-15.

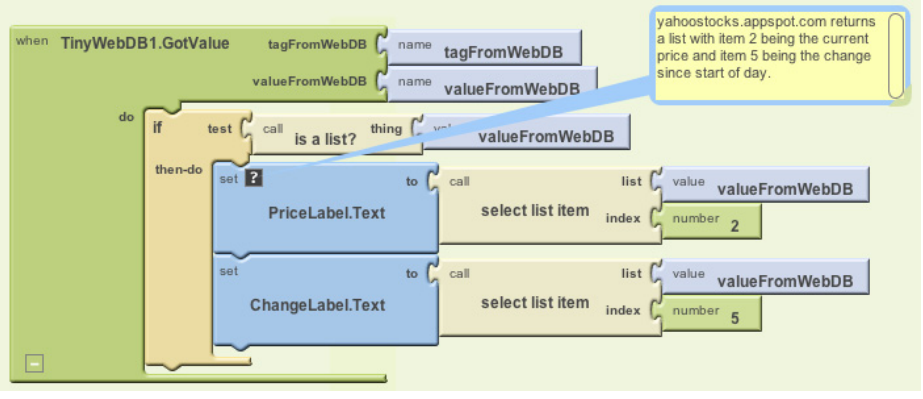

Figure 24-15. Using the GotValue event to process the data that arrives from Yahoo!

If you check the API specification at *http://yahoostocks.appspot.com*, you'll see that the second item in the returned list is indeed the current price, and the fifth item is the change since stocks began trading that day. This app simply extracts those items from what is returned by the API, and shows them in the labels `PriceLabel` and `ChangeLabel`. Figure 24-16 provides a snapshot of the app in action.

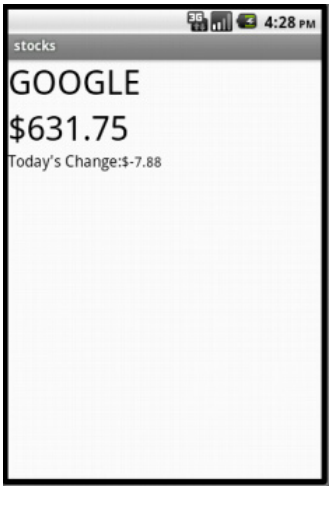

Figure 24-16. The Stocks App in action

Creating Your Own App Inventor–Compliant APIs

`TinyWebDB` is the bridge from an App Inventor app to the Web. It lets App Inventor programmers talk to web services with the simple tag-value protocol inherent in the `GetValue` function. You send a particular tag as the parameter, and a list or text object is returned as the value. In this way, the App Inventor programmer is shielded from the difficult programming required to *parse* (understand and extract data from) standard data formats like XML or JSON.

The tradeoff is that App Inventor apps can talk only to web services that follow `TinyWebDB`'s expected protocol—it expects data to be returned in a very specific way, and the API has to provide its data accordingly. App Inventor doesn't have a component for accessing an arbitrary web service that returns standard data formats such as XML or JSON. If there isn't an App Inventor–compliant API already available, someone with the ability to write a web program must create it.

In the past, building APIs was difficult because you not only needed to understand the programming and web protocols, but you also needed to set up a server to host your web service, and a database to store the data. Now it's much easier, as you can leverage cloud-computing tools like Google's App Engine and Amazon's Elastic Compute Cloud to immediately deploy the service you create. These platforms will not only host your web service, but they'll also let thousands of users access it before charging you a single dime. As you can imagine, these sites are a great boon to innovation.

Customizing Template Code

Writing your own API may seem daunting, but the good news is that you don't need to start from scratch. You can leverage some provided template code that makes it especially easy to create App Inventor–compliant APIs. The code is written in the Python programming language and uses Google's App Engine. The template provides boilerplate code for getting the data into the form that App Inventor needs, and a function, `get_value`, that you can customize.

You can download the template code and instructions for deploying it on Google's App Engine servers at *http://appinventorapi.com/using-tinywebdb-to-talk-to-an-api/*. You might notice that the link takes you to the same *appinventorapi.com* site that was used in Chapter 21 to create a custom web database. Building an API is similar, only instead of just storing and retrieving data, you'll call some other service to access the data you need.

To create your own web API, you'll download the template, modify a few key places in the code, and then upload it to App Engine. Within minutes, you will have your own API that can be called using `TinyWebDB` in an App Inventor app.

Here's the particular code from the template that you'll need to customize (don't worry about the text that comes after the # symbol; like the comments in App Inventor, it just describes what the code following it is doing):

```
def get_value(self, tag):
    #For this simple example, we just return hello:tag, where tag is sent in by client
    value="hello:"+tag
    value = "\""+value+"\""   # add quotes if the value is has multiple words
    if self.request.get('fmt') == "html":
         WriteToWeb(self,tag,value )
    else:
         WriteToPhone(self,tag,value)
```

This code is for a *function* (same as a *procedure* in App Inventor) called `get_value`, and it's indeed the code that is invoked when your app calls an API with the **TinyWebDB.GetValue** function. `tag` is a *parameter* of the function and corresponds to the tag you send in the **GetValue** call.

The bolded code is the part you'll change. By default, it simply takes the tag sent in with the request and sends back "hello tag." (In other words, if you call this code with the tag "joe," it returns "hello joe"). It does this by setting the variable `value`, which is then sent to the function `WriteToWeb` if the request came from the Web, or `WriteToPhone` if the request came from a phone.

Note. *Even if you've never looked at Python or other programming code, you may find the sample above somewhat readable from your experience with App Inventor. The "def get_value..." line defines a procedure, the "value=..." lines are setting the variable "value" to something, and the "if.. " statements should look familiar. The fundamental concepts are the same, its just text instead of blocks.*

To customize the template, you replace the bold code with any computation you want, as long as that code places something in the variable `value`. Often, your API will make a call to another API (this is called "wrapping" a call—more specifically, your `get_value` function will make the call to some other API).

Many APIs are complicated, with hundreds of functions and complex user authorization schemes. Others, however, are quite simple, and you can even find sample code for accessing them on the Web, as you'll see in the next section.

Wrapping the Yahoo! Finance API

The Yahoo! Stocks API for App Inventor used in this chapter was created by modifying the template code above with code found through a simple web search. As the goal was wrapping the Yahoo! Stocks API for use by App Inventor, the developer (Wolber) did a web search for "Python Yahoo Stocks API". From the site http://www.gummy-stuff.org/Yahoo-data.htm, he found that a URL in the form:

http://download.finance.yahoo.com/d/quotes.csv?f=sl1d1t1c1ohgv&e=.cs v&s=IBM

would return a text file with a single comma-separated string of data. The preceding URL returns this text string:

```
"IBM",140.85,"10/15/2010","3:00pm",-0.65,142.10,142.10,140.60,4974553
```

He then found some Python code for accessing the Yahoo! Stocks API at http://www.goldb.org/ystockquote.html. With some quick cutting and pasting and a bit of editing, the App Inventor wrapper API was created by modifying the template in the following manner:

```
def get_value(self, tag):
    # Need to generate a string or list and send it to WriteToPhone/ WriteToWeb
    # Multi-word strings should have quotes in front and back
    # e.g.,
    #    value = "\""+value+"\""
    # call the Yahoo Finance API and get a handle to the file that is returned
    quoteFile=urllib.urlopen("http://download.finance.yahoo.com/d/quotes.csv?f=
    sl1d1t1c1ohgv&e=.csv&s="+tag)
    line = quoteFile.readline()  # there's only one line
    splitlist = line.split(",")  # split the data into a list
    # the data has quotes around the items, so eliminate them
    i=0
    while i<len(splitlist):
        item=splitlist[i]
        splitlist[i]=item.strip('"') # remove " around strings
        i=i+1
    value=splitlist
    if self.request.get('fmt') == "html":
         WriteToWeb(self,tag,value )
    else:
         WriteToPhone(self,tag,value)
```

The bolded code calls the Yahoo! API within the `urllib.urlopen` function call (this is one way to call APIs from the Python language). The URL has a parameter, `f`, that specifies the type of stock data you want (this parameter is something like the cryptic parameters required by the Google Chart API). The data returned from Yahoo! is then put into the variable `line`. The rest of the code splits up the items into a list, removes the quotation marks around each item, and sends the result to the requester (either the web interface or an App Inventor app).

Summary

Most websites and many mobile apps are not standalone entities; they rely on the interoperability of other sites to do their jobs. With App Inventor, you can build games, quizzes, and other standalone apps, but soon enough, you'll encounter issues related to web access. Can I write an app that tells me when the next bus will arrive at my usual stop? Can I write an app that texts a special subset of my Facebook friends? Can I write an app that sends tweets? App Inventor provides two hooks to the Web: (1) you can set the `Image.Picture` property to a URL to bring in a (generated) image, and (2) you can use `TinyWebDB` to access data in a specially designed web API.

App Inventor does not provide arbitrary access to APIs. Instead, the system relies on programmers to create "wrapper" APIs that follow a particular protocol. Once created, these APIs are available to App Inventor app programmers using the same **TinyWebDB.GetValue** scheme they use to access databases. Actually writing APIs is certainly a bigger hurdle than writing apps in App Inventor, but if you're interested in learning how, be sure to check out some Python books and courses (O'Reilly has a few of those!), and you'll be on your way.