Communicating With Web APIs

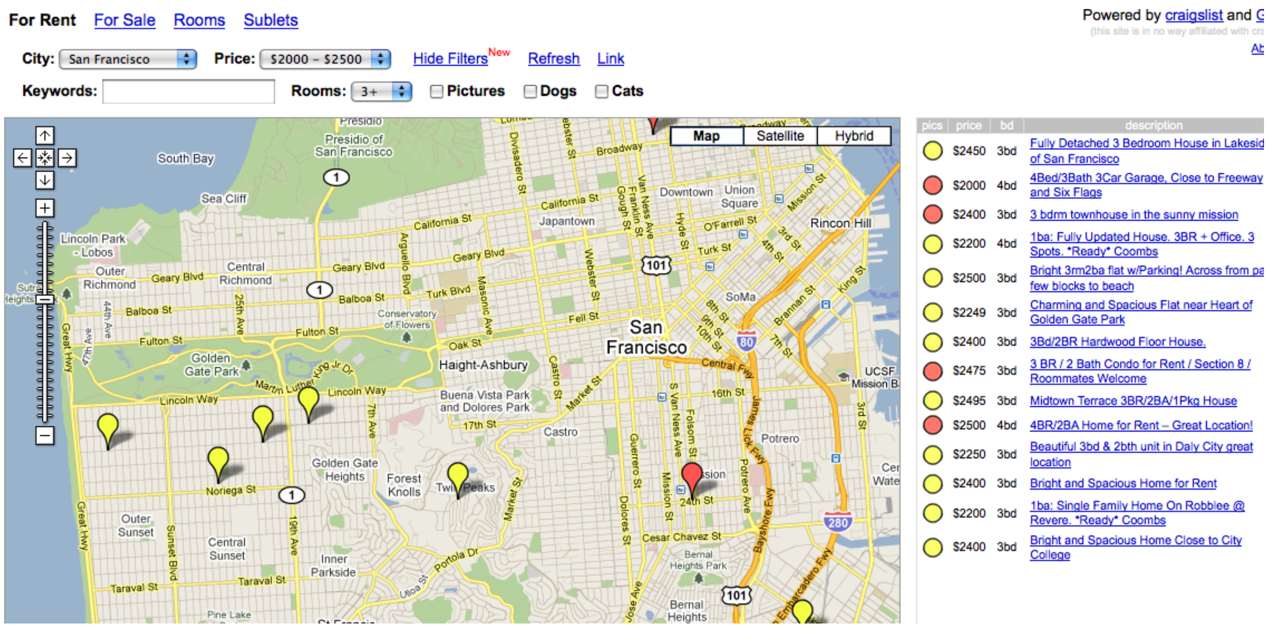
Mobile technology and the ubiquitous nature of the web have changed the world we live in. You can now sit in the park and do your banking, search Amazon to find reviews of the book you're reading, and check Twitter to see what people in every other park in the world are thinking about. Mobile phones 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 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.

The Amazon in the Bookstore app from chapter 13 provides a complete example of a web-enabled app. 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 web site to generate a bar chart (image) of the 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--as it does in the Amazon sample--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 Negativeland--focus primarily on creating new tracks from mashing old content.

The web and mobile world are no different: web sites 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 housingmaps.com, pictured in figure 24-1, which takes apartment rental information from craigslist.com and mashes it with the Google Maps API:



City: San Francisco Price: $2000 - $2500
Keywords: Rooms: 3+ Pictures Dogs Cats

pics	price	bd	description
🟡	$2450	3bd	Fully Detached 3 Bedroom House in Lakesid of San Francisco
🔴	$2000	4bd	4Bed/3Bath 3Car Garage, Close to Freeway and Six Flags
🔴	$2400	3bd	3 bdrm townhouse in the sunny mission
🟡	$2200	4bd	1ba: Fully Updated House. 3BR + Office. 3 Spots. *Ready* Coombs
🟡	$2500	3bd	Bright 3rm2ba flat w/Parking! Across from pa few blocks to beach
🟡	$2249	3bd	Charming and Spacious Flat near Heart of Golden Gate Park
🟡	$2400	3bd	3Bd/2BR Hardwood Floor House.
🔴	$2475	3bd	3 BR / 2 Bath Condo for Rent / Section 8 / Roommates Welcome
🟡	$2495	3bd	Midtown Terrace 3BR/2BA/1Pkg House
🔴	$2500	4bd	4BR/2BA Home for Rent – Great Location!
🟡	$2250	3bd	Beautiful 3bd & 2bth unit in Daly City great location
🟡	$2400	3bd	Bright and Spacious Home for Rent
🟡	$2200	3bd	1ba: Single Family Home On Robblee @ Revere. *Ready* Coombs
🟡	$2400	3bd	Bright and Spacious Home Close to City College

Figure 24-1. *Housing Maps mashes information from cragislist.com and maps.google.com.*

Mashups like Housing Maps are possible because services like maps.google.com provide both a web site and a corresponding *web service API* (or Application Programmer Interface) . We humans visit 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 web site 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 their data for use by third-party entities would eventually lead to more books being sold. When Facebook launched their API in 2007, many raised their eyebrows: Facebook's data isn't book advertisements, why should they let other apps "steal" it and potentially draw many away from the Facebook site (and its advertisements!. But the openness led Facebook towards becoming a *platform* instead of just a site, and no one can argue with their success. By the time Twitter launched in 2009, API access was an expectation, not a novelty, and Twitter followed suit.

So the web is one thing to us humans: a collection of sites to visit. To programmers, it is the world's largest and most diverse database of information. More machine-to-machine communication now occurs than human-machine communication on the web!

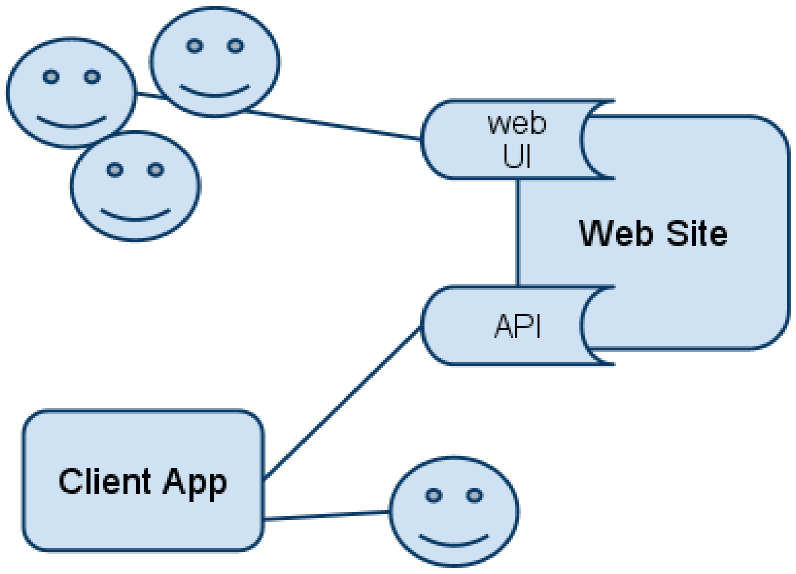


Figure 24-2. Most web sites provide both a human interface and an API for client apps.

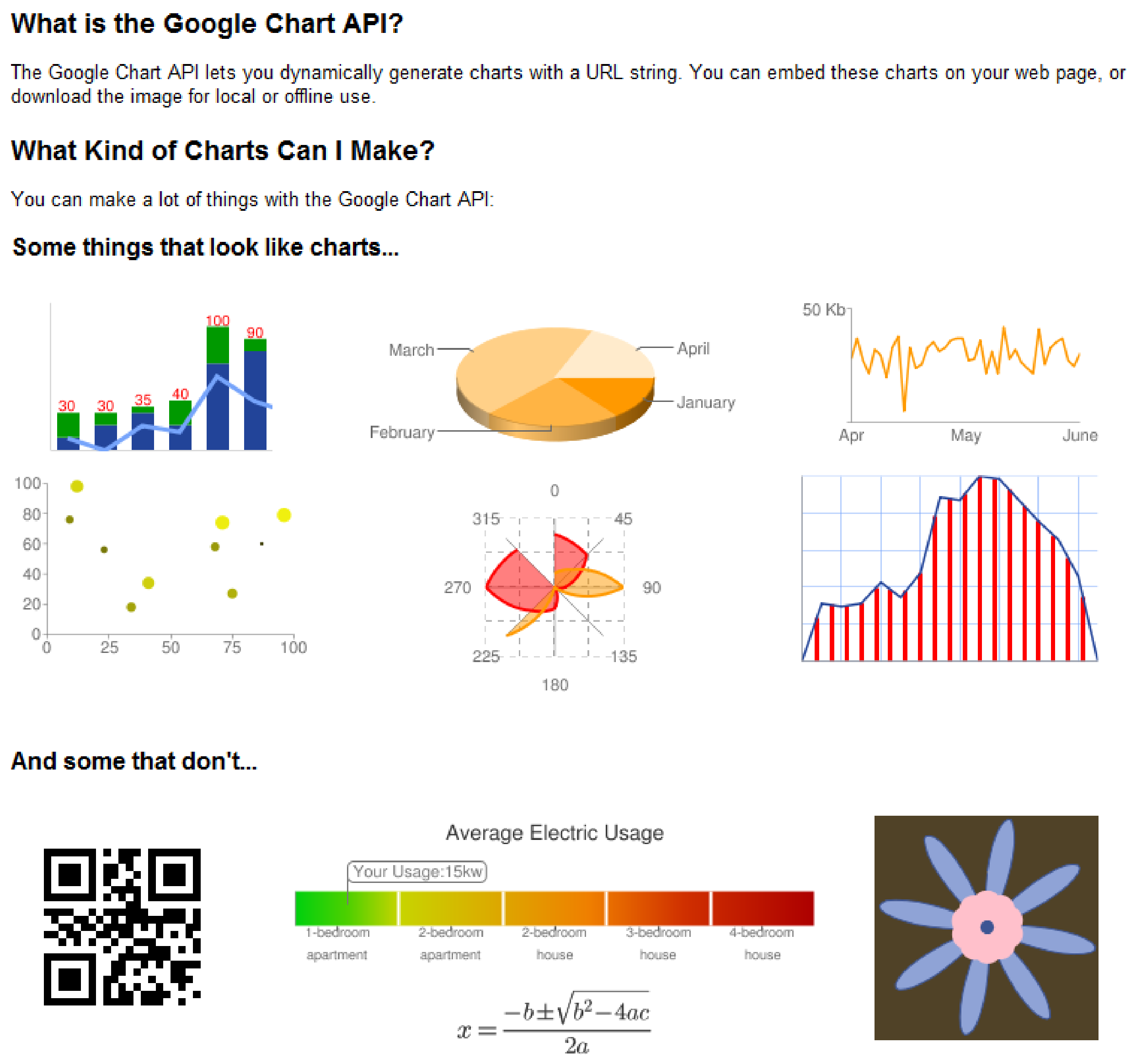
Talking to Web APIs that Generate Images

Most APIs accept requests in the form of a URL, and return data in standard formats like XML (Extensible Markup Language and JSON (Javascript Object Notation). For these APIs, you'll use the TinyWebDB component to communicate with them, and we'll discuss how to do that 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 the user interface capabilities of App Inventor.

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. As 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 this overview shown in figure 24-3:



What is the Google Chart API?

The Google Chart API lets you dynamically generate charts with a URL string. You can embed these charts on your web page, or download the image for local or offline use.

What Kind of Charts Can I Make?

You can make a lot of things with the Google Chart API:

Some things that look like charts...

And some that don't...

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.

So play around with the website and the wizard and build some charts, and take a look at the details of the URLs used to build them. For example, if the following URL is entered in a broswer:

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:

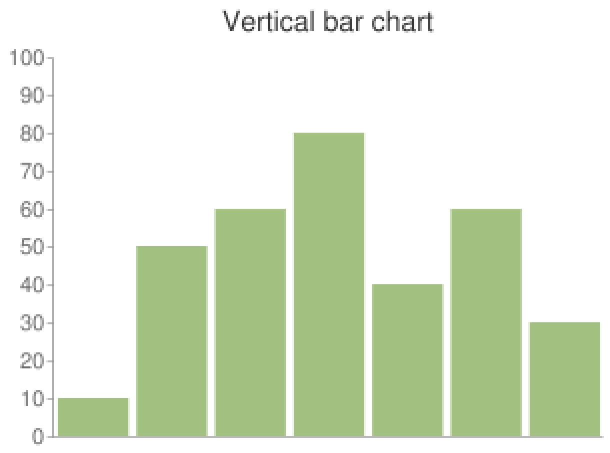


Figure 24-4. *If you enter the given URL into a browser, the chart above it will be drawn.*

To understand the rather complicated looking URL, 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 Charts API (http://chart.apis.google.com/chart) with the following parameters listed in table 24-1:

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

Parameter	**Value**	**Meaning**
cht	bvg	The chart type is bar, vertical,?
chxt	y	Show the numbers on the y axes
chbh	a	width/spacing is automatic
chs	300x225	size of the chart in pixels

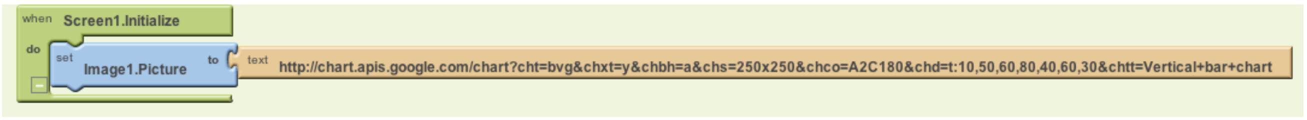
chco	A2C180	bar colors, in hexadecimal
chd	t:10,50,60,80,40,60,30	the data of the chart,with basic text fomat (t)
chtt	Vertical+bar+chart	chart title, + means space

By modifying the parameters you can generate various graphs. For more information, see the API documentation at http://code.google.com/apis/chart/index.html.

Setting the Image.Picture Property to a Chart API

You can type in 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 the URL. To explore this, do the following:

- Create a new App with Screen Title of "Sample Chart App"
- Add an Image component with Width of Fill Parent and Height of 300.
- 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-paste on some machines, so you'll have to type the URL):



```
when Screen1.Initialize
do  set Image1.Picture to  text  http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=250x250&chco=A2C180&chd=t:10,50,60,80,40,60,30&chtt=Vertical+bar+chart
```

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

You should see the image in figure 24-6 on your phone/emulator:

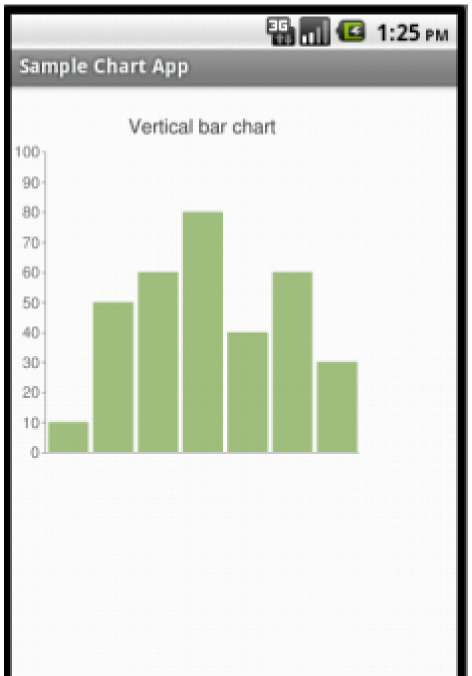


Figure 24.6. *The chart in an app*

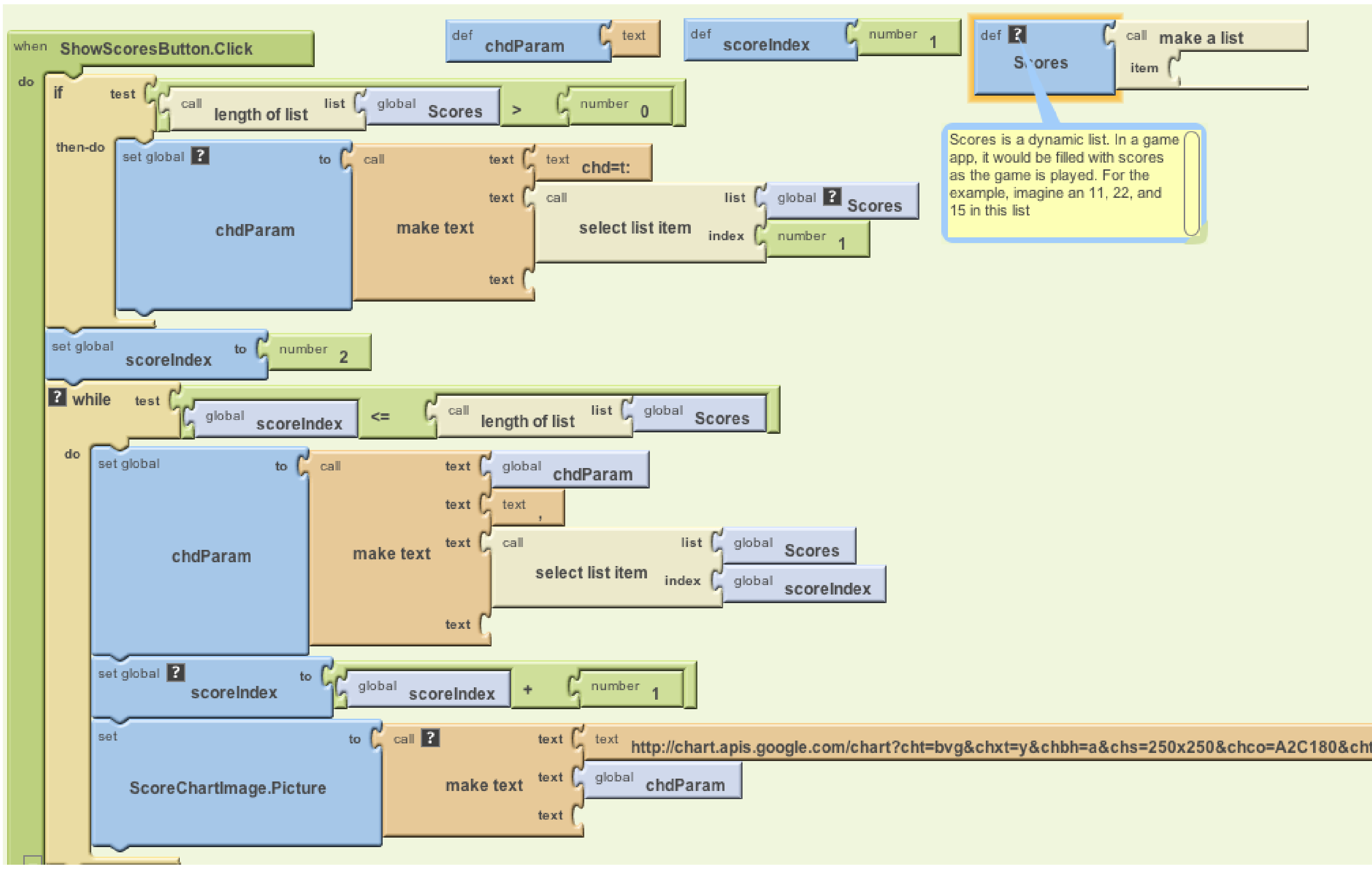
Building a Chart API URL Dynamically

The above sample 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:", then step through the **Scores** list concatenating each score to the text (along with a comma). Figure 24-7 shows a complete solution:


when ShowScoresButton.Click
do
if test call length of list list global Scores > number 0
then-do set global ? chdParam to call make text text text chd=t:
text call select list item list global ? Scores index number 1
text
set global scoreIndex to number 2
? while test global scoreIndex <= call length of list list global Scores
do set global chdParam to call make text text global chdParam text text ,
text call select list item list global Scores index global scoreIndex
text
set global ? scoreIndex to global scoreIndex + number 1
set ScoreChartImage.Picture to call ? make text text text http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=250x250&chco=A2C180&cht
text global chdParam
text
def chdParam text
def scoreIndex number 1
def ? Scores call make a list item

Scores is a dynamic list. In a game app, it would be filled with scores as the game is played. For the example, imagine an 11, 22, and 15 in this list

Figure 24.7. *Blocks for dynamically building a URL to send to the Chart API.*

Let's examine the blocks closely as there's a lot going on in here, much of which we've covered in previous chapters. In understanding 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 which will contain the chd data.The job of the first row of blocks is to initialize the text of the **chdParam** from the list of **Scores**:

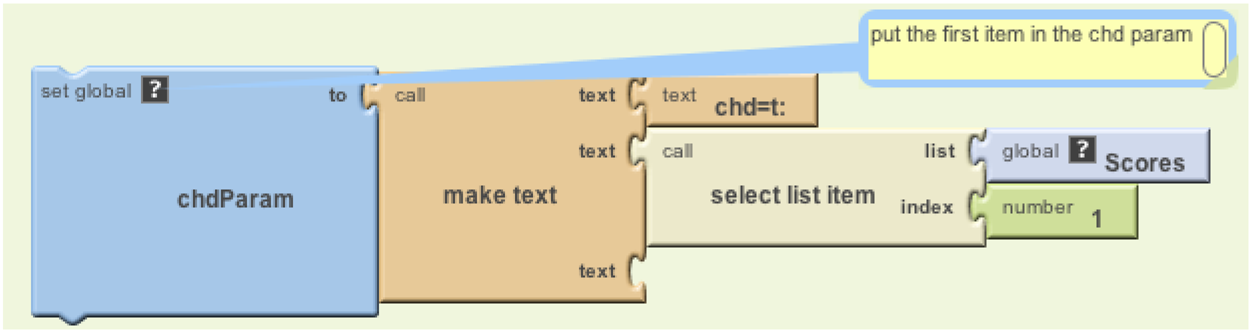
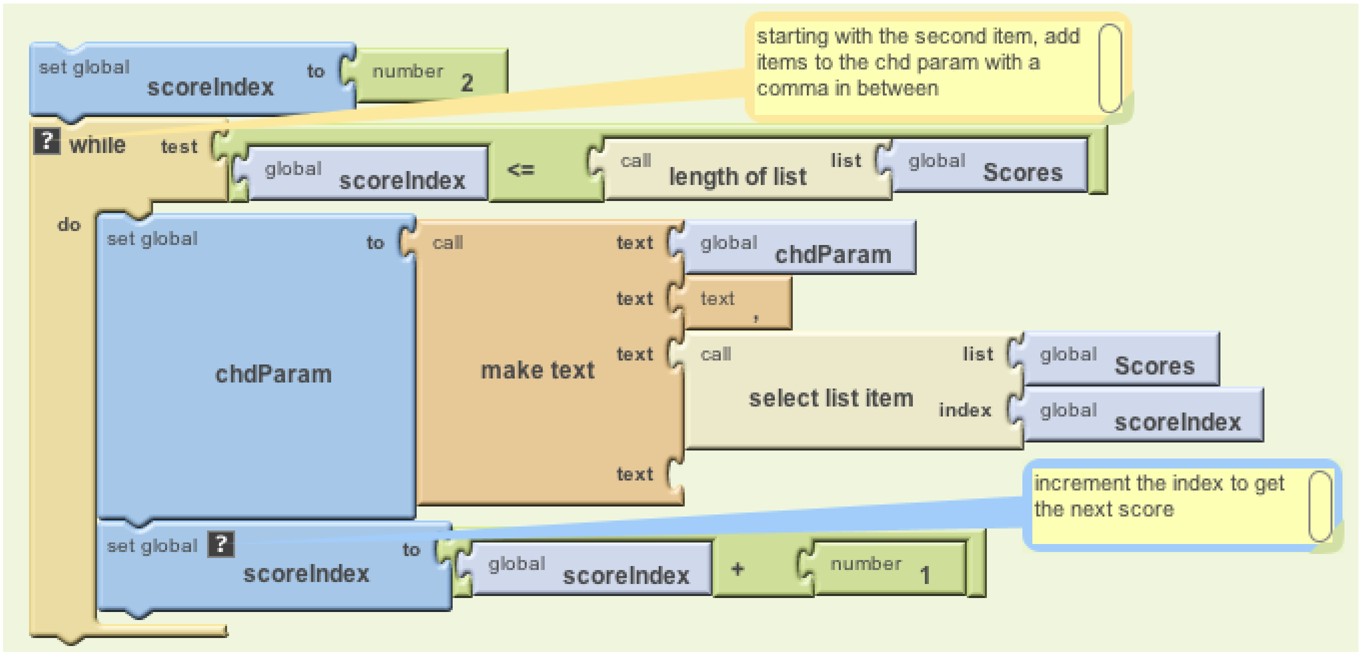

put the first item in the chd param

Figure 24-8. *Blocks for 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 job of the next set of blocks, shown in figure 24-9, is to add the rest of the scores to the **chdParam**:


starting with the second item, add items to the chd param with a comma in between

increment the index to get the next score

Figure 24-9. *Blocks for adding the successive scores to the **chdParam** variable.*

A **while** block is used in this sample instead of a **foreach** because foreach only allows you to do the same thing to each item. Here, we only want to insert commas before the 2nd item on (not the first). With **while**, we can put the first item in (Figure 7), then loop only starting from the second item on, always putting in a comma *before* the item. For more information on **while** and **foreach**, see Chapter 19.

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, the **chdParam** will contain “chd=t:11,22,15”. We have built the chd parameter dynamically! Check it compared to the chd parameter in the sample URL: does it look similar?

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

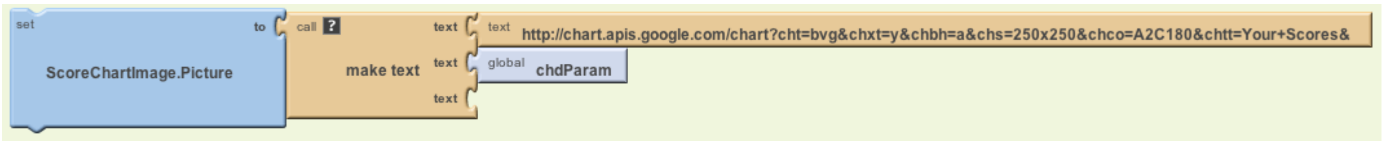

```
set ScoreChartImage.Picture to
  call make text
    text  text http://chart.apis.google.com/chart?cht=bvg&chxt=y&chbh=a&chs=250x250&chco=A2C180&chtt=Your+Scores&
    text  global chdParam
    text
```

Figure 24-10. *Set the picture to the full URL including the chd param just built.*

The blocks results in the `ScoreChartImage.Picture` property being set to the 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”, and the app appearing as in Figure 24-11.

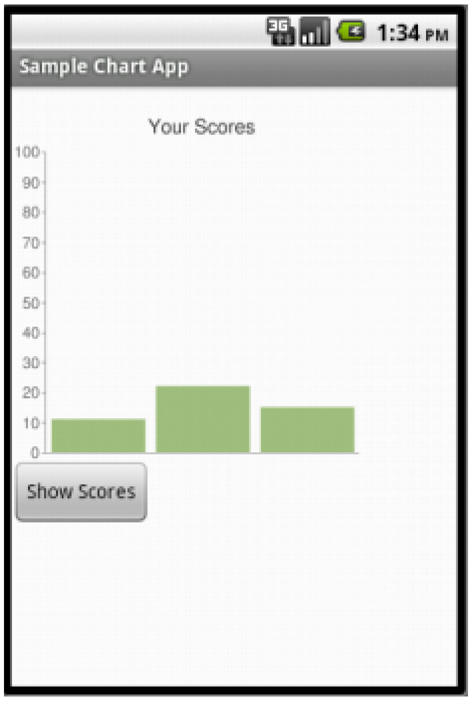


Figure 24-11. *The generated chart.*

You could add such a display to any game or any app by adding blocks similar to this sample.

You could also talk to other APIs that generate images. The key is that App Inventor provides a hook to the web through the Image component.

Talking to Web Data APIs

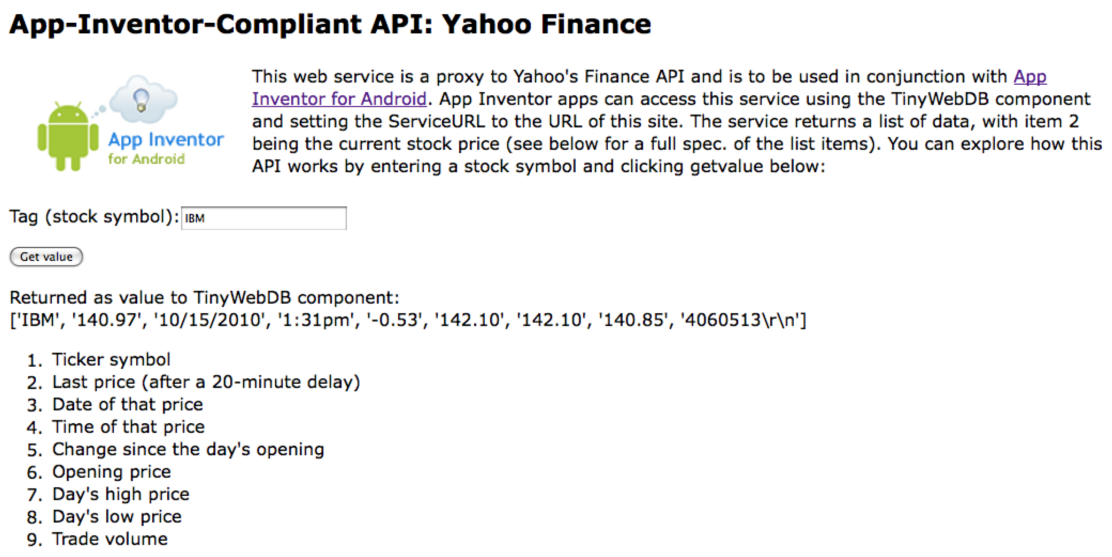
The Google Chart API is a web API that responds to requests by returning a picture. More commonly, APIs will return data that an app can process and use however it wants. The Amazon at the Bookstore app in Chapter 13, for instance, returns data in the form of a list of books with each book including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

To talk to an API from an App Inventor app, you don't need to build a URL, as with the Chart API example. Instead, you query the API much like you would a web database (Chapter 21): 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 have been created and more will soon follow. You can find some of these at the web site 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 to the service picutred in figure 24-12.


App-Inventor-Compliant API: Yahoo Finance

This web service is a proxy to Yahoo's Finance API and is to be used in conjunction with App Inventor for Android. App Inventor apps can access this service using the TinyWebDB component and setting the ServiceURL to the URL of this site. The service returns a list of data, with item 2 being the current stock price (see below for a full spec. of the list items). You can explore how this API works by entering a stock symbol and clicking getvalue below:

Tag (stock symbol): IBM

Returned as value to TinyWebDB component:
['IBM', '140.97', '10/15/2010', '1:31pm', '-0.53', '142.10', '142.10', '140.85', '4060513\r\n']

1. Ticker symbol
2. Last price (after a 20-minute delay)
3. Date of that price
4. Time of that price
5. Change since the day's opening
6. Opening price
7. Day's high price
8. Day's low price
9. Trade volume

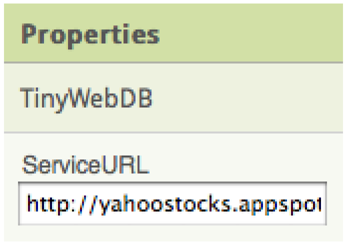
***Figure 24-12**. The web interface to the App-Inventor-Compliant Yahoo Finance API.*

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 on 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 above web service is to drag a TinyWebDB component into the Component Designer. There is only one property associated with TinyWebDB, its ServiceURL, pictured 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, you set this property to http://yahoostocks.appspot.com, the same URL you entered at the browser address bar above to see the web page interface.


Properties

TinyWebDB

ServiceURL

http://yahoostocks.appspot

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 yahoostocks.appspot.com web site:

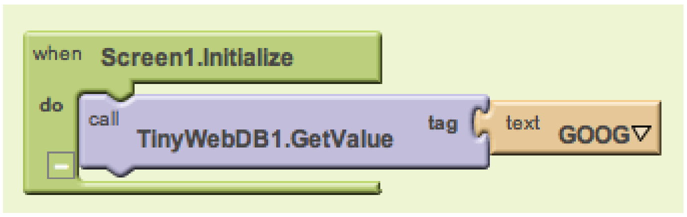

```
when Screen1.Initialize
do  call TinyWebDB1.GetValue tag  text GOOG
```

Figure 24-14. *Blocks to request stock information.*

As was discussed in the section on web databases in chapter X, the TinyWebDB communication is *asynchronous*: your app requests the data with **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 arrives back from the web service. From our examination of the human interface of 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 like in figure 24-15 :

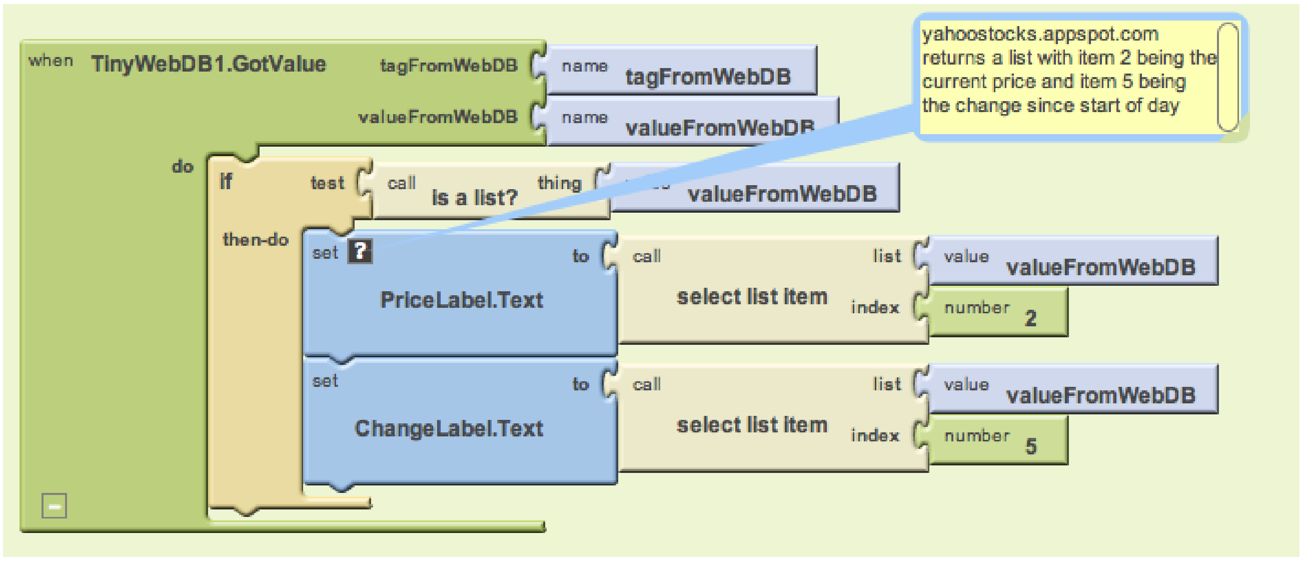


Figure 24-15. *When the data arrives from Yahoo, the* ***GotValue*** *event processes it.*

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 that data and shows it in the labels PriceLabel and ChangeLabel. Figure 24-16 provides a snapshot of the app in action:

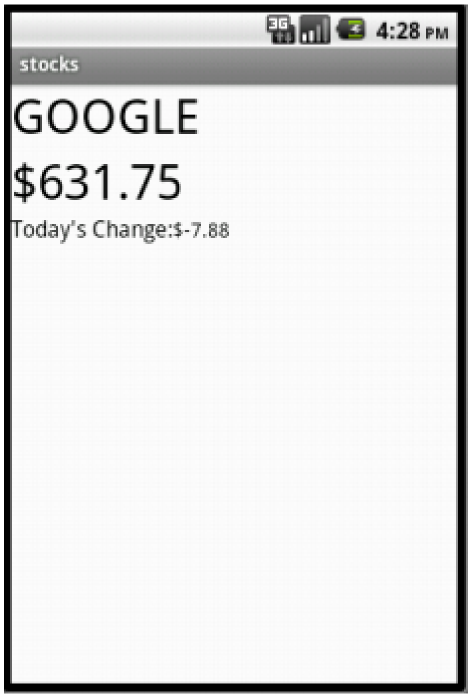


Figure 24-16. The Stocks App in action.

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 to 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 only talk to web services that follow TinyWebDB's expected protocol. 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, you also needed to setup 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 Computing Cloud to immediately deploy the service you create. Not only will these platforms host your web service, they'll let your service be hit by thousands of users 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 using 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, which you can customize.

The template code and instructions for deploying it on Google's App Engine servers is in Appendix 1. Simply put, you can download the template, modify a few key places in the code, then upload it to App Engine. Within minutes, you can have your own API that can be called using TinyWebDB in an App Inventor app.

Here's the particular code that you'll need to customize from the template:

```
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)
```

The function's name is get_value, and its indeed the code that is invoked when a your app calls an API with the **TinyWebDB.GetValue** function. The parameter *tag* of the function is the tag you send in the **GetValue** call.

The part in bold is the part you'll change. By default, it simply takes the tag sent in with the request and sends back 'hello tag'. 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.

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 make a call to another API (this is called "wrapping" a call--more specifically your **get_value** function will make the call to this other API).

Many APIs are complicated with hundreds of functions and complex user authorization schemes. Others, however, are quite simple. For many, you can even find sample code for them on the web. In fact, the Yahoo Stocks API that we used above was created in just this way.

Wrapping the Yahoo Finance API

The developer who wrote the Yahoo Stocks API for App Inventor did a quick web search, and found out that a URL in the form:

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

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

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

Using this knowledge, the template code we looked at earlier can be modified to call the Yahoo API:

```
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 callls 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, which 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 web sites and many mobile apps are not stand-alone entities; they rely on the interoperability of other sites to do their job. 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, such APIs can be accessed by App Inventor client programmers using the same **TinyWebDB.GetValue** scheme they use to access databases.

Appendix X. Template for an App-Inventor Compliant API
[need instructions for deploying this]

```
#!/usr/bin/env python
###
### This web service is used in conjunction with App  Inventor for Android.
### It is a bare-bones web service API that just responds to any getValue(tag) with hello tag
### Author: Dave Wolber via template of Hal Abelson and input of Dean Sanvitale

import logging
```

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from google.appengine.ext.db import Key
from django.utils import simplejson as json
import os
from google.appengine.ext.webapp import template
import urllib

class GetValueHandler(webapp.RequestHandler):
  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 a text with multiple words
        if self.request.get('fmt') == "html":
          WriteToWeb(self,tag,value )
        else:
          WriteToPhone(self,tag,value)

  def post(self):
        tag = self.request.get('tag')
        self.get_value(tag)

  # there is a web ui for this as well, here is the get
  def get(self):
        # this did pump out the form
        template_values={}
        path = os.path.join(os.path.dirname(__file__),'index.html')
        self.response.out.write(template.render(path,template_values))
class MainPage(webapp.RequestHandler):
  def get(self):
        template_values = {}
        # render the page using the template engine
        path = os.path.join(os.path.dirname(__file__),'index.html')
        self.response.out.write(template.render(path,template_values))

#### Utilty procedures for generating the output
def WriteToPhone(handler,tag,value):
        handler.response.headers['Content-Type'] = 'application/jsonrequest'
        json.dump(["VALUE", tag, value], handler.response.out)

def WriteToWeb(handler, tag,value):
        template_values={"result":  value}
        path = os.path.join(os.path.dirname(__file__),'index.html')
```

```
          handler.response.out.write(template.render(path,template_values))

### Assign the classes to the URL's
application =   \
   webapp.WSGIApplication([('/', MainPage),
                    ('/getvalue', GetValueHandler)
                    ],
                    debug=True)
def main():
  run_wsgi_app(application)

if __name__ == '__main__':
  main()
```

*****CUTS BELOW HERE*****

You’ll learn how to use TinyWebDB to communicate with special, App-Inventor-Compliant web information sources, and you’ll learn how to create your own web sources that live in the “cloud”.

This process has become much easier with the advent of free “cloud computing” tools like Google’s App Engine and Amazon’s Elastic Computing Cloud. It requires some traditional programming with a language like Python, Java, or PHP, but with the templates we provide, and cloud computing, you can learn to create App-Inventor-compliant web sources.

Whereas HTML is the language used when a server sends information to a browser for human consumption, XML is generally used when a server sends information to a program on another server. Both HTML and XML include data within them (e.g., book information), but HTML also has presentation information-- specifications for how the information should be displayed-- while XML does not. [examples of HTML/XML]

***
In traditional programming, talking to web information sources requires parsing either JSON or XML code. A service might return XML data such as:

```
<Book
        <title>Outliers</title>
        <author>Malcolm Gladwell</author>
</Book>
```

Your code has to "parse" the XML to get it into an easy-to-use variable form.

App Inventor eases the task of creating web-enabled apps through its TinyWebDB component. In creating TinyWebDB, the Google engineers did the heavy-lifting of parsing data. The application programmer-- the beneficiary-- can talk to web services through a simple tag-value protocol. You call a TinyWebDB.GetValue function with a tag, and the component does the parsing work to return you data in the form of simple text or a list.

In the past, creating a web service was a difficult task requiring sophisticated knowledge of databases, web servers, and web service XML prototocols. Fortunately, cloud computing tools like App Engine have made this task significantly easier. Within minutes and with no more resources than access to a browser you can set up a web service and database that run on Google's servers, free of charge! Furthermore, using the templates we provide, you can create web services that can 1) access information from other web information sources, and 2) talk to an App Inventor "client" app.

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 to find the cost of the book you're reading, and check Twitter to see what people in every other park in the world are thinking about. A mobile phone lets you communicate with people, but it also gives you instant access to the world's data.

You can use your phone's browser to reach the web, but often the small screen and limited speed of a mobile device 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.

Some assume that, because App Inventor is a visual language, it lacks the depth to build such complex, web-enabled apps, that such apps require hard-core Java SDK programming.

But the same TinyWebDB component that can be used to store data in a web database can also be used to access information sources on the web.

TinyWebDB serves as a perfect facilitator between App Inventor app programmers, who can request web data with a simple GetValue(tag) request, and back-end programmers, who can program web services for use by App Inventor client programmers.

There are now a number of App-Inventor-compliant web information sources available to App Inventor programmers. These can be found at various sites on the web. The first author's site, appinventorapi.com, has a number of them, including services for:

- Amazon book and pricing information
- Stock information from Yahoo Finance
- RSS feeds (posts from any blog)
- Weather data from yyy

The first part of this chapter describes methods for creating apps that "consume" information from these services. Later in the chapter, you'll be introduced to creating your own App-Inventor-compliant web sources using some pre-existing templates, the Python language, and Google's App Engine.

Often the client app requests the data using a fairly common looking URL, with the data returned in XML or JSON (Java Script Object Notation) format.

The protocol is simple: the service must handle the following request:

```
getValue?tag=<tag>
```

and that request should result in the return of a JSON (JavaScript Object Notation) object holding either a list or string (text). Your App Inventor app doesn't need to deal with the JSON; TinyWebDB will convert it into a simple list or text object.

This is especially nice because, for many APIs, you can find sample Python code for calling them on the web. Thus with some exploration and some cut-and-paste, you can create App-Inventor-Compliant web services without being an expert Python or web programmer.

With App Inventor you can build games, quizzes, and other stand-alone apps, but soon enought issue of web access comes up. 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 twitters?

To old-timers, the fact that the students expect the answer to be 'yes' is quite audacious. But such is the ubiquitousness of the web today, and one must also take into account the empowerment that a few weeks with App Inventor provides.

Building client apps that talk to pre-existing App Inventor services is quite reasonable for such beginners, though understanding an API specification given on a web site and applying it within the realm of the TinyWebDB component is a conceptual leap.

Actually writing web services is a bigger hurdle. Some can certainly do it, like my student Paul Cafardo... who took a Python course after his App Inventor course and by the end created an App Inventor compliant service for the eventful API. However, I don't expect too many beginners to leap over the TinyWebDB bridge. You'll certainly see bridge jumpers going the other way-- experienced programmers creating web services and App Inventor clients. I think you'll also see on-line collaborations, perhaps at sites like appinventoapi.com, where desginers/App Inventor programmers ask for particular services and partner with programmers who can easily provide such code.