Working with Databases

Figure 22-1.

Facebook has a database of every member's account information, friends list, and posts. Amazon has a database of just about everything you can buy. Google has a database of information about every page in the World Wide Web. Though not to such a scale, almost every nontrivial app you can create will interact with a database.

In most programming environments, building an app that communicates with a database is an advanced programming technique: you have to set up a server with database software such as Oracle or MySQL and then write code that interfaces with that database. In many universities, such database programming isn't taught until an upper-level software engineering or database course.

When it comes to databases, App Inventor does the heavy lifting for you (and lots of other useful things!). The language provides components that reduce database communication to simple store and get operations. You can create apps that store data directly on the Android device, and with some setup, you can create apps that share data with other devices and people by storing it in a centralized database on the Web.

The data stored in variables and component properties is short-term: if the user types some information in a form and then closes the app before that information has been stored in a database, the information will be gone when the app is reopened. To store information *persistently*, you must store it in a database. The information in databases is said to be *persistent* because even when you close the app and reopen it, the data is still available.

As an example, consider *Chapter 4*'s No Texting While Driving app, which sends an auto-response to incoming SMS text messages. The app has a default response that is sent, but it lets the user enter a custom message to be sent, instead. If the user changes the custom message to "I'm sleeping; stop bugging me" and then closes the app, the message should still be "I'm sleeping; stop bugging me," and not the original default, when the app is reopened. Thus, the custom message must be stored in a database, and every time the app is opened, that message must be retrieved from the database back into the app.

Storing Persistent Data in TinyDB

App Inventor provides two components to facilitate database activity: `TinyDB` and `TinyWebDB`. You use `TinyDB` to store persistent data directly on the Android device; this is useful for personal apps for which the user won't need to share data with another device or person, as in No Texting While Driving. On the other hand, you use `TinyWebDB` to store data in a web database that can be shared among devices. Being able to access data from a web database is essential for multiuser games and apps with which users can enter and share information (like the "MakeQuiz" app in *Chapter 10*).

The database components are similar, but `TinyDB` is a bit simpler, so we'll explore it first. With `TinyDB`, you don't need to set up the database at all; the data is stored in a database directly on the device and associated with your app.

You transfer data to long-term memory with the `TinyDB.StoreValue` block, as shown in *Figure 22-1*, which comes from the No Texting While Driving app.
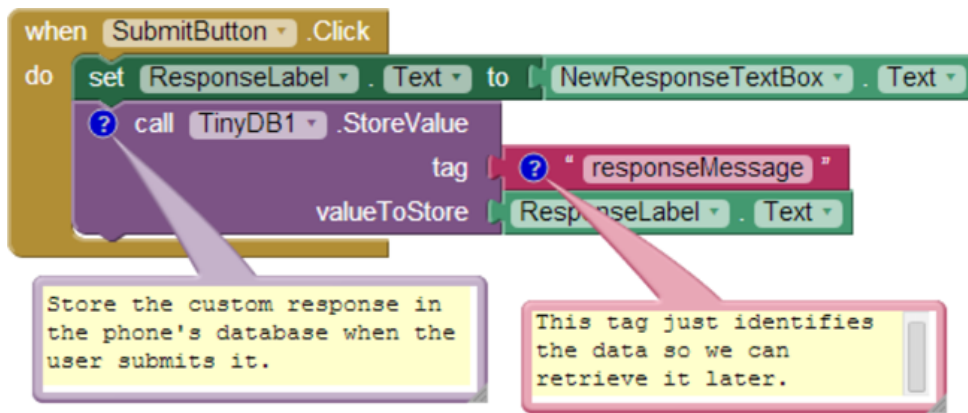
Figure 22-2. The TinyDB.StoreValue block stores data to the device's long-term memory

A tag-value scheme is used for database storage. In *Figure 22-1*, the data is tagged with the text "responseMessage." The value is some text that the user has typed in a text box for the new custom response—something like, "I'm sleeping; stop bugging me."

The `tag` parameter gives the data you're storing in the database a name, a way to reference the information. The `value` is the data itself. You can think of the tag as a key that you'll use later when you want to retrieve the data from the database.

Likewise, you can think of an App Inventor `TinyDB` database as a table of tag-value pairs. After the `TinyDB1.StoreValue` in *Figure 22-1* is executed, the device's database will have the value listed in *Table 22-1*.

Table 22-1. The value stored in the databases

Tag	Value
responseMessage	I'm sleeping; stop bugging me

An app might store many tag-value pairs for the various data items that you want to be persistent. The tag is always text, whereas the value can be either a single piece of information (a text or number) or a list. Each tag has only one value; every time you store to a tag, it overwrites the existing value.

Retrieving Data from TinyDB

You retrieve data from the database by using the `TinyDB.GetValue` block. When you call `GetValue`, you request particular data by providing a tag. For the No Texting While Driving app, you can request the custom response by using the same tag as you used in the `StoreValue`, "responseMessage." The call to `GetValue` returns the data, so you must plug it into a variable.

Often, you'll retrieve data from the database when the app opens. App Inventor provides a special event handler, `Screen.Initialize`, which is triggered when the app launches. You need to be careful to consider the case when there is no data yet in the database (e.g., the first time app is launched). When you call `GetValue`, you specify a `valueIfTagNotThere` parameter. If there is no data, that value will be returned from the call.

The blocks in *Figure 22-2*, for the `Screen.Initialize` of No Texting While Driving app, are indicative of the way many apps load database data on initialization.

The blocks put the data returned from `GetValue` into the label `ResponseLabel`. If there is data already in the database, it is placed in `ResponseLabel`. If there is no data for the given tag, the `valueIfTagNotThere` value, "I'm driving right now, I'll text you later" in this case, is placed in `ResponseLabel`.

Figure 22-3. When the app launches, you'll often retrieve database information

Shared Data and TinyWebDB

The TinyDB component stores data in a database located directly on the Android device. This is appropriate for personal-use apps that don't need to share data among users. For instance, many people might install the No Texting While Driving app, but there's no need for the various people using the app to share their custom responses with others.

Of course, many apps do share data: think of Facebook, Twitter, and multiuser games. For such data-sharing apps, the database must reside on the Web, not the device, so that different app users can communicate with it and access its information.

TinyWebDB is the web counterpart to TinyDB. With it, you can write apps that store data on the Web, using a StoreValue/GetValue protocol similar to that of TinyDB.

By default, the TinyWebDB component stores data by using a web database set up by the App Inventor team and accessible at *http://appinvtinywebdb.appspot.com*. That website contains a database and serves (responds to) web requests for storing and retrieving data. The site also provides a human-readable web interface that a database administrator (you) can use to examine the data stored there.

This default database is for development only; it is limited in size and accessible to all App Inventor programmers. Because any App Inventor app can store data there, you have no assurance that another app won't overwrite your data!

If you're just exploring App Inventor or in early the stages of a project, the default web database is fine. But, if you're creating an app for real-world deployment, at some point you'll need to set up your own web database. Because we're just exploring right now, we'll use the default web database. Later in the chapter, you'll learn how to create your own web database and configure TinyWebDB to use that instead.

In this section, we'll build a voting app (depicted in *Figure 22-3*) to illustrate how TinyWebDB works. The app will have the following features:

- Users are prompted to enter their email address each time the app loads. That account name will be used to tag the user's vote in the database.

- Users can submit a new vote at any time. In this case, their old vote will be overwritten.

- Users can view the votes from everyone in the group.

- For the sake of simplicity, the issue being voted on is determined outside the app, such as in a classroom setting in which the teacher announces the issue and asks everyone to vote electronically. (Note that this example could be extended to allow users to prompt votes by posting issues to vote on from within the app.)

Figure 22-4. A Voting app that stores votes to TinyWebDB

STORING DATA BY USING TINYWEBDB

The `TinyWebDB.StoreValue` block works in the same manner as `TinyDB.StoreValue`, except that the data is stored on the Web. For our voting sample, assume that the user can enter a vote in a text box named `VoteTextBox` and tap a button named `VoteButton` to submit the vote. To store the vote to the web database so that others can see it, we'll code the `VoteButton.Click` event handler like the example in *Figure 22-4*.
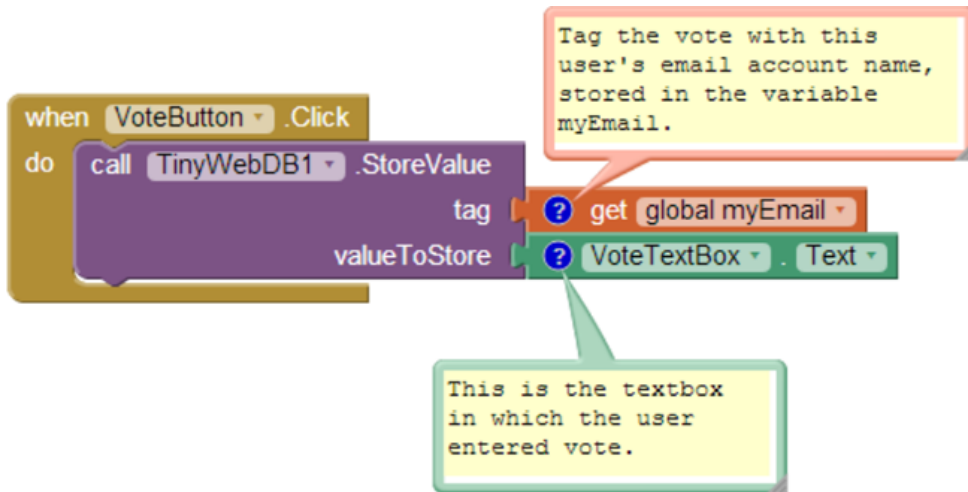
Figure 22-5. When the user enters a vote, it is stored on the web database

The tag used to identify the data is the user's email, which has previously been stored in the variable `myEmail` (you'll see this later). The value is whatever the user typed in VoteTextBox. So, if the user email was *joe@zmail.com* and his vote was "Pizza," the entry would be stored in the database as shown in *Table 22-2*.

Table 22-2. The tag and value for the vote are recorded in the database

tag	value
joe@zmail.com	Pizza

The `TinyWebDB.StoreValue` block sends the tag-value pair over the Web to the database server at *http://appinvtinywebdb.appspot.com*. As you test your app, you can go to that URL, click `getValue`, and enter a tag for which you've stored a value. The website will show you the current value for that tag.

REQUESTING AND PROCESSING DATA WITH TINYWEBDB

Retrieving data with `TinyWebDB` is more complicated than with `TinyDB`. With `TinyDB`, the `GetValue` operation immediately returns a value because your app is communicating with a database directly on the Android device. With `TinyWebDB`, the app is requesting data over the Web, which can take time, so Android requires a two-step scheme for handling it.

With `TinyWebDB`, a call to `GetValue` only *requests* the data; it should really be called "RequestValue" because it just makes the request to the web database and doesn't

actually get a value from it right away. To see this more clearly, check out the difference between the TinyDB.GetValue block and the TinyWebDB.GetValue block shown in *Figure 22-5*.

Figure 22-6. The TinyDB.GetValue and TinyDB.GotValue blocks

The TinyDB.GetValue block returns a value right away, and thus a plug appears on its left side so that the returned value can be placed into a variable or property. The TinyWebDB.GetValue block does not return a value immediately, so there is no plug on its left side.

Instead, when the web database fulfills the request and the data arrives back at the device, a TinyWebDB.GotValue event is triggered. So, you'll call TinyWebDB.GetValue in one place of your app, and then you'll program the TinyWebDB.GotValue event handler to specify how to handle the data when it actually arrives. An event handler such as TinyWebDB.GotValue is sometimes called a *callback procedure*, because some external entity (the web database) is in effect calling your app back after processing your request. It's similar to ordering at a busy coffee shop: you place your order and then wait for the barista to call your name to actually go pick up your drink. In the meantime, she's been taking orders from everyone else in line, too (and those people are all waiting for their names to be called, as well).

GETVALUE-GOTVALUE IN ACTION

For our sample app, we need to store and retrieve a list of the voters who have the app, as the app needs to show the votes of all users.

The simplest scheme for retrieving list data is to request the data when the app launches, in the Screen.Initialize event, as shown in *Figure 22-6*. (In this example, we'll just call the database with the tag for "voterlist.")
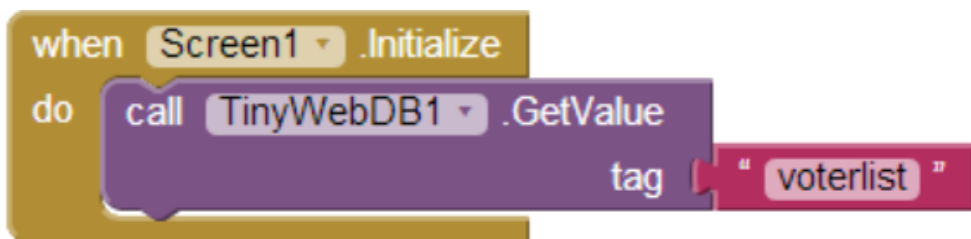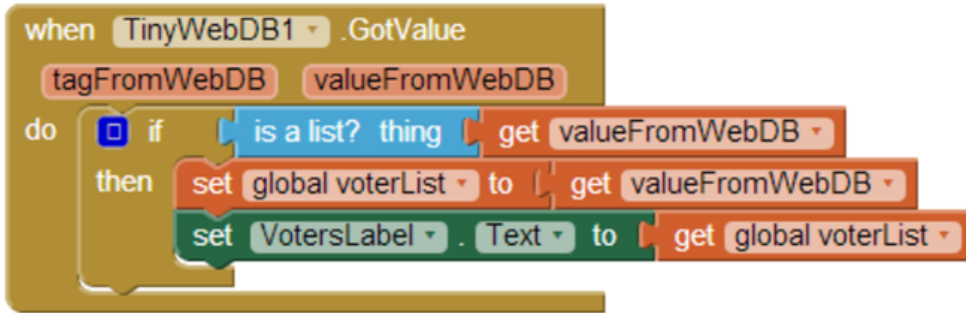

Figure 22-7. Requesting data in the Screen1.Initialize event

When the list of voters arrives from the web database, the `TinyWebDB1.GotValue` event handler is triggered. *Figure 22-7* shows some blocks for processing the returned list.

Figure 22-8. Using the GotValue event handler to process the returned list

The `valueFromWebDB` *argument* of `GotValue` holds the data returned from the database request. Event arguments such as `valueFromWebDB` have meaning only within the event handler that invokes them. They are considered *local* to the event handler, as you can't reference them in other event handlers.

Because arguments such as `valueFromWebDB` aren't globally accessible, if you need the information throughout your app, you need to transfer it to a global variable. In the example, `GotValue`'s primary job is to transfer the data returned in `valueFromWebDB` into the variable `voterList`, which you'll use in another event handler.

The `if` block in the event handler is also often used in conjunction with `GotValue`, the reason being that the database returns an empty text ("") in `valueFromWebDB` if there is no data for the requested tag. This empty return value occurs most commonly when it's the first time the app has been used. By asking if the `valueFromWebDB` is a list, you're making sure that there is some data actually returned. If the `valueFromWebDB` is the empty text (the `if` test is false), you don't put it into `voterList`.

A MORE COMPLEX GETVALUE/GOTVALUE EXAMPLE

The blocks in *Figure 22-7* are a good model for retrieving data in a fairly simplistic app. In our voting example, however, we need more complicated logic. Specifically:

- The app should prompt the user to type an email address when the program starts. We can use a `Notifier` component for this, which pops up a window. (You can find the `Notifier` in the "User Interface" palette in the Designer.) When the user types an email, we'll store it in a variable.

- Only after determining the user's email should you call `GetValue` to retrieve the voter list. Can you figure out why?

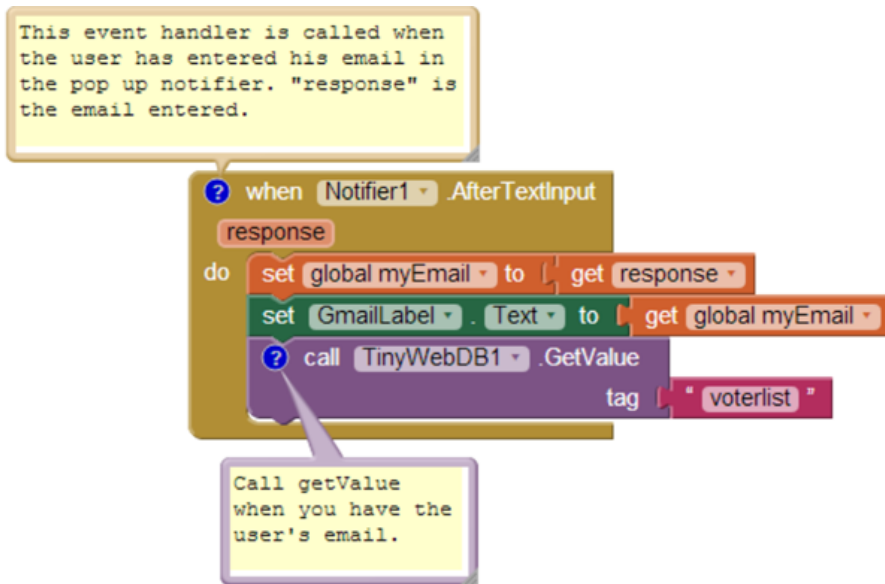*Figure 22-8* shows the blocks for this more complicated scheme for requesting the database data.

Figure 22-9. In this more complex scheme, GetValue is called after getting the user's email instead of in Screen.Initialize

Upon startup (`Screen1.Initialize`), a `Notifier` component prompts the user to type an email address. When the user does so, and the `Notifier.AfterTextInput` event handler is triggered, the entry is put into a variable and label, and then `GetValue` is called to get the list of voters. Note that `GetValue` isn't called directly in `Screen.Initialize`, because we need the user's email address to be set first.

So, with these blocks, when the app initializes, it prompts the user to type an email address and then calls `GetValue` with a tag of "voterlist." When the list arrives from the Web, `GotValue` is triggered. Here's what should happen:

- `GotValue` should check if the data that arrives is non-empty (someone has used the app and initiated the voter list). If there is data (a voter list), `GotValue` should check if our particular user's email address is already in the voter list. If it's not, it should be added to the list, and the updated list should be stored back to the database.

- If there isn't yet a voter list in the database, we should create one with the user's email address as the only item.

Figure 22-9 shows the blocks for this behavior.

The blocks first ask if a non-empty voter list came back from the database by calling is a list? If so, the data is put into the variable voterList. Remember, voterList will have email addresses for everyone who has used this app. However, we don't know if this particular user is in the list yet, so we need to check. If the user is not yet in the list, the user's email address is added with add item to list, and the updated list is stored to the web database.
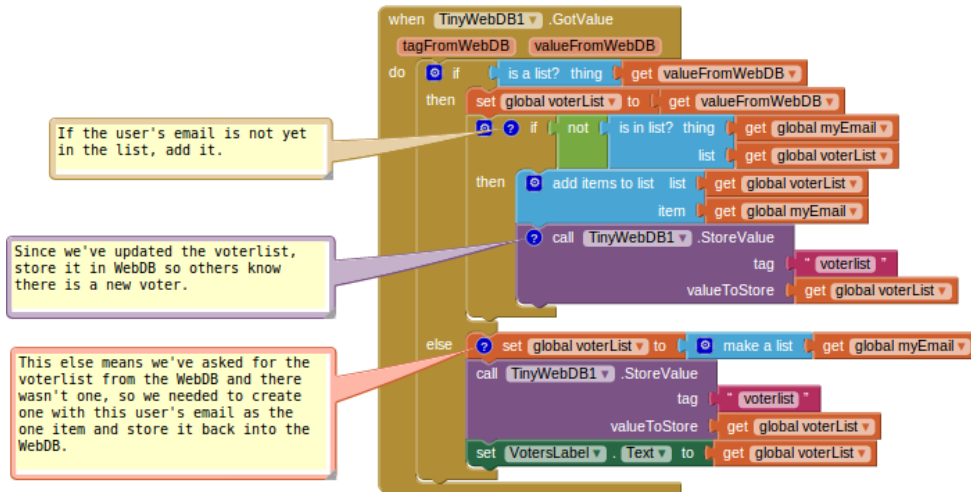
Figure 22-10. Using the GotValue blocks to process the data returned from the database and perform different actions based on what is returned

The else of the if else block is invoked if a list wasn't returned from the web database; this happens if nobody has used the app yet. In this case, a new voterList is created with the current user's email address as the first item. This one-item voter list is then stored to the web database (with the hope that others will join, as well!).

Requesting Data with Various Tags

The voting app thus far manages a list of an app's users. Each person can see the email addresses of all the other users, but we haven't yet created blocks for retrieving and displaying each user's vote.

Recall that the VoteButton.Click event submited a vote with a tag-value pair of the form "email: vote." If two people had used the app and voted, the pertinent database entries would look something like *Table 22-3*.

Table 22-3. The tag-value pairs stored in the database

tag	value
voterlist	[*bill@zmail.com, joe@zmail.com*]
bill@zmail.com	Hot dogs
joe@zmail.com	Pizza

When the user clicks on the ViewVotes button, the app should retrieve all votes from the database and display them. Suppose that the voter list has already been retrieved into the variable voterList; we can use a for each to request the vote of each person in the list, as shown in *Figure 22-10*.
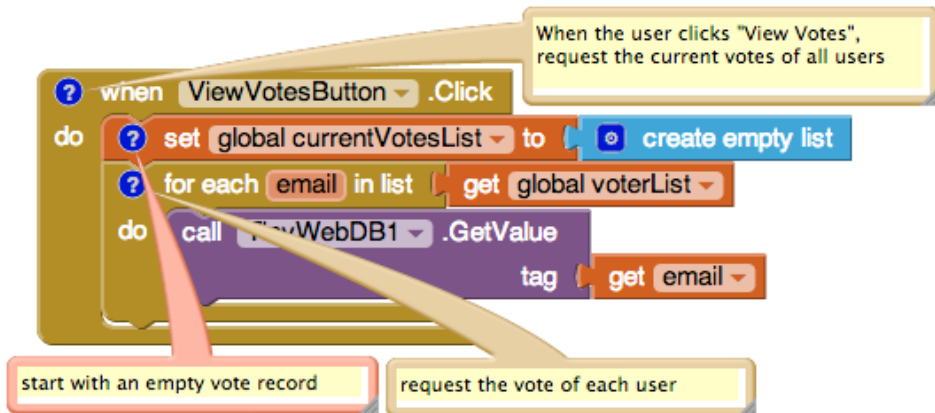
Figure 22-11. Using a for each block to request the vote of each person in the list

Here we initialize a variable, currentVotesList, to an empty list, because our goal is to add the up-to-date votes from the database into this list. We then use for each to call TinyWebDB1.GetValue for every email address in the list, sending the current item of the for each, renamed "email," as the tag in the request. Note that the votes won't actually be added to currentVotesList until they arrive via a series of GotValue events.

Now that we want to display the votes in our app, things get a bit more complicated yet again. With the requests from ViewVotesButton, TinyWebDB.GotValue will now be returning data related to all the email tags, as well as the "voterlist" tag used to retrieve the list of user email addresses. When your app requests more than one item from the database with different tags, you need to code TinyWebDB.GotValue to handle all possible requests. (You might think that you could try to code multiple

`GotValue` event handlers, one for each database request—can you figure out why this won't work?)

To handle this complexity, the `GotValue` event handler has a `tagFromWebDB` argument that informs you as to which request has just arrived. In this case, if the tag is "voterlist," we should continue to process the request as we did previously. If the tag is something else, we can assume it's the email of someone in the user list, stemming from the requests triggered in the `ViewVotesButton.Click` event handler. When those requests come in, we want to add the incoming data—the voter and vote—to the `currentVotesList` so that we can display it to the user.

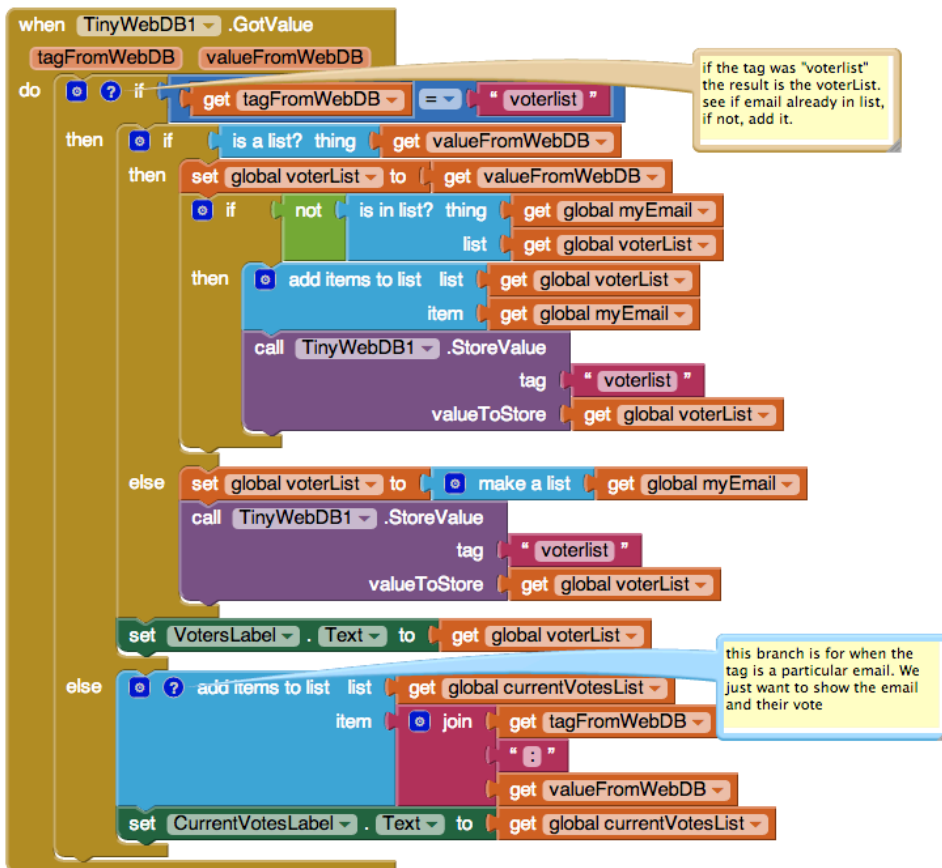*Figure 22-11* shows the entire `TinyWebDB.GotValue` event handler.

Figure 22-12. The TinyWebDB.GotValue event handler

Setting Up a Web Database

As we mentioned earlier in the chapter, the default web database at *http:// appinvtinywebdb.appspot.com* is intended for prototyping and testing purposes only. Before you deploy an app with real users, you need to create a database specifically for your app.

You can create a web database by using the instructions at *http:// appinventorapi.com/create-a-web-database-python-2-7*. This site was set up by one of the authors (Wolber) and contains sample code and instructions for setting up App Inventor web databases and APIs. The instructions point you to some code that you can download and use with only a minor modification to a configuration file. The code download is the same as that used for the default web database set up by App Inventor. It runs on Google's App Engine, a cloud-computing service that will host your web database on Google's servers for free (well, at least until the site receives a certain number of hits). By following the instructions, you can have your own private web database that is compliant with App Inventor's protocols up and running within minutes and begin creating web-enabled mobile apps that use it.

When you create and deploy your own custom web database, the App Engine tool provides you with a URL where your server resides. You can direct your app to use your custom database server instead of the default *http:// appinvtinywebdb.appspot.com*, by changing the `ServiceURL` property in the `TinyWebDB` component. After that property is changed, all calls to `TinyWebDB.StoreValue` and `TinyWebDB.GetValue` will interface with the new web service.

Summary

App Inventor makes it easy to store data persistently through its `TinyDB` and `TinyWebDB` components. Data is always stored as a tag-value pair, with the tag identifying the data for later retrieval. Use `TinyDB` when it is appropriate to store data directly on the device. When data needs to be shared across phones (e.g., for a multiplayer game or a voting app), you'll need to use `TinyWebDB`, instead. `TinyWebDB` is more complicated because you need to set up a callback procedure (the `GotValue` event handler) as well as a web database service.