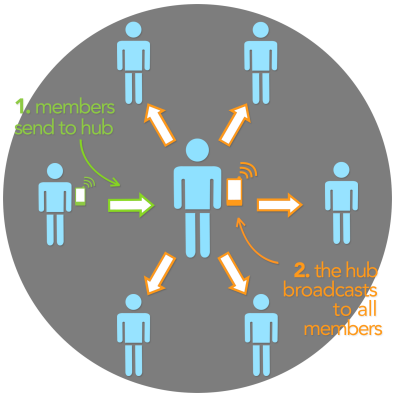
CHAPTER 11

Broadcast Hub

FrontlineSMS is a software tool used in developing countries to monitor elections, broadcast weather changes, and connect people who don't have access to the Web but do have phones and mobile connectivity. It is the brainchild of Ken Banks, a pioneer in using mobile technology to help people in need.

The software serves as a hub for SMS text communication within a group. People send in a special code to join the group, after which they receive broadcast messages from the hub. For places with no Internet access, the broadcast hub can serve as a vital connection to the outside world.

Figure 11-1.



In this chapter, you'll create a broadcast hub app that works similarly to FrontlineSMS but runs on an Android phone. Having the hub itself on a mobile device means that the administrator can be on the move, something that is especially important in controversial situations, such as election monitoring and healthcare negotiations.

In this chapter, you'll build a broadcast hub app for the fictitious FlashMob Dance Team (FMDT), a group that uses the hub to organize flash mob dances anywhere, anytime. People will register with the group by texting "joinFMDT" to the hub, and anyone who is registered can broadcast messages to everyone else in the group.

Your app will process received text messages in the following manner:

1. If the text message is sent from someone not yet in the broadcast list, the app responds with a text that invites the person to join the broadcast list.
2. If the text message with the special code "joinFMDT" is received, the app adds the sender to the broadcast list.
3. If the text message is sent from a number already in the broadcast list, the message is broadcast to all numbers in the list.

This app is more complicated than the No Text While Driving app in *Chapter 4*, but you'll build it one piece of functionality at a time, starting with the first auto-response

message that invites people to join. By the time you complete this, you'll have a pretty good idea of how to write apps utilizing SMS text as the user interface. Do you want to write a vote-by-text app such as those used on television talent shows, or the next great group texting app? You'll learn how here!

What You'll Learn

The tutorial covers the following App Inventor concepts, some of which you're likely familiar with by now:

- The `Texting` component for sending texts and processing received texts.
- List variables and dynamic data—in this case, to keep track of the list of phone numbers.
- The `for each` block to allow an app to repeat operations on a list of data. In this case, you'll use `for each` to broadcast messages to the list of phone numbers.
- The `TinyDB` component to store data persistently. This means that if you close the app and then relaunch it, the list of phone numbers will still be there.

Getting Started

You'll need a phone with SMS service to test or run this app. You'll also need to recruit some friends to send you texts in order to fully test the app.

Connect to the App Inventor website and start a new project. Name it "BroadcastHub", and also set the screen's title to "Broadcast Hub". Then, connect your device or emulator for live testing.

Designing the Components

Broadcast Hub facilitates communication between mobile phones. Those phones do not need to have the app installed, or even be smartphones; they'll communicate by text with your app. So, in this case, the user interface for your app is just for the group administrator.

The user interface for the administrator is simple: it displays the current *broadcast list*; that is, the list of phone numbers that have registered for the service, and all of the texts it receives and broadcasts.

To build the interface, add the components listed in *Table 11-1*.

Table 11-1. User interface components for Broadcast Hub

Component type	Palette group	What you'll name it	Purpose
`Label`	User Interface	`Label1`	This is the header "Registered Phone Numbers" above the list of phone numbers.
`Label`	User Interface	`BroadcastListLabel`	Display the phone numbers that are registered.
`Label`	User Interface	`Label2`	This is the header "Activity Log" above the log information.
`Label`	User Interface	`LogLabel`	Display a log of the texts received and broadcast.
`Texting`	Social	`Texting1`	Process the texts.
`TinyDB`	User Interface	`TinyDB1`	Store the list of registered phone numbers.

As you add the components, set the following properties:

1. Set the `Width` of each label to "Fill parent" so that it spans the phone horizontally.
2. Set the `FontSize` of the header labels (`Label1` and `Label2`) to 18 and check their FontBold boxes.
3. Set the `Height` of `BroadcastListLabel` and `LogLabel` to 200 pixels. They'll show multiple lines.
4. Set the `Text` property of `BroadcastListLabel` to "Broadcast List...".
5. Set the `Text` property of `LogLabel` to blank.
6. Set the `Text` property of `Label1` to "Registered Phone Numbers".
7. Set the `Text` property of `Label2` to "Activity Log".

Figure 11-1 shows the app layout in the Component Designer.

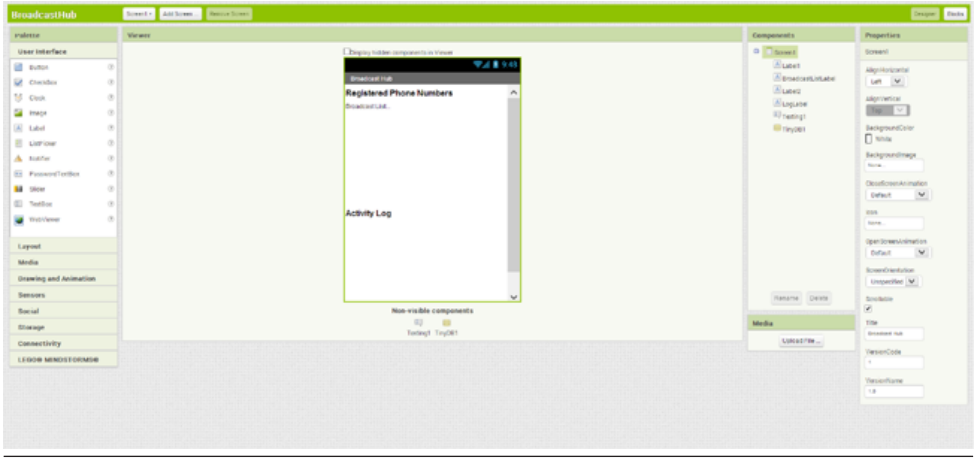


Figure 11-2. Broadcast Hub in the Components Designer

Adding Behaviors to the Components

The activity for Broadcast Hub is not triggered by the user typing information or clicking a button; rather, it's texts coming in from other phones. To process these texts and store the phone numbers that sent them in a list, you'll need the following behaviors:

- When the text message is sent from someone not already in the broadcast list, the app responds with a text that invites the sender to join.
- When the text message "joinFMDT" is received, register the sender as part of the broadcast list.
- When the text message is sent from a number already in the broadcast list, the message is broadcast to all numbers in the list.

RESPONDING TO INCOMING TEXTS

You'll start by creating the first behavior: when you receive a text, send a message back to the sender inviting her to register by texting "joinFMDT" back to you. You'll need the blocks listed in *Table 11-2*.

Table 11-2. Blocks for adding the functionality to invite people to the group via text

Block type	Drawer	Purpose
`Texting1.MessageReceived`	Texting1	Triggered when the phone receives a text.
`set Texting1.PhoneNumber to`	Texting1	Set the number for the return text.

Block type	Drawer	Purpose
`get number`	Drag from MessageReceived event handler	The argument of `MessageReceived`. This is the phone number of the sender.
`set Texting1.Message`	Texting1	Set the invite message to send.
`text` ("To join this broadcast list, text 'joinFMDT' to this number")	Text	The invite message.
`Texting1.SendMessage`	Texting1	Send it!

How the blocks work

If you completed the No Texting While Driving app in *Chapter 4*, these blocks should look familiar. `Texting1.MessageReceived` is triggered when the phone receives any text message. *Figure 11-2* shows how the blocks within the event handler set the `PhoneNumber` and `Message` of the `Texting1` component and then send the message.

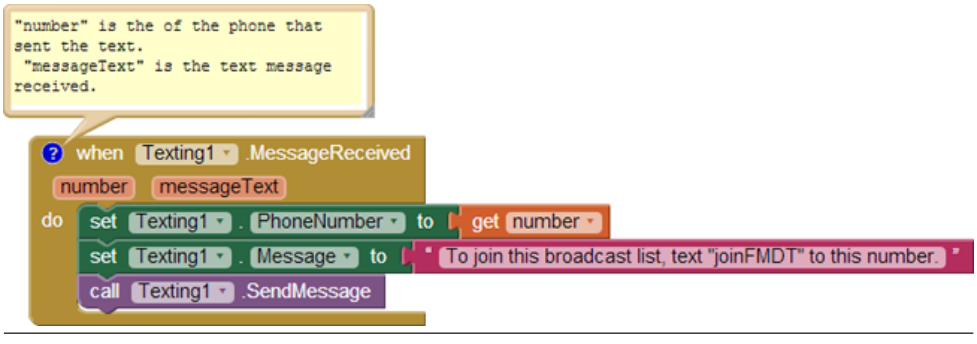


Figure 11-3. Sending the invite message back after receiving a text

Test your app *You'll need a second phone to test this behavior; you don't want to text yourself, because it could loop forever! If you don't have another phone, you can register with Google Voice or a similar service and send SMS texts from that service to your phone. From the second phone, send the text "hello" to the phone running the app. The second phone should then receive a text that invites it to join the group.*

ADDING NUMBERS TO THE BROADCAST LIST

It's time to create the blocks for the second behavior: when the text message "joinFMDT" is received, the sender is added to the broadcast list. First, you'll need to define a list variable, `BroadcastList`, to store the phone numbers that register. From

the Variables drawer, drag out an `initialize global` block and name it "BroadcastList". Initialize it to an empty list by using a `create a list` block from the Lists drawer, as shown in *Figure 11-3* (we'll add the functionality to build this list shortly).

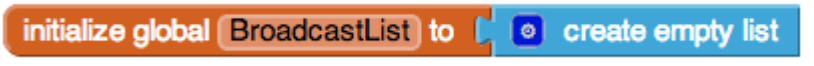


Figure 11-4. The BroadcastList variable for storing the list of registered numbers

Next, modify the `Texting1.MessageReceived` event handler so that it adds the sender's phone number to the `BroadcastList` if the message received is "joinFMDT." You'll need an `if else` block to check the message, and an `add item to list` block to add the new number to the list. The full set of blocks you'll need is listed in *Table 11-3*. After you add the number to the list, display the new list in the `BroadcastListLabel`.

Table 11-3. Blocks for checking a text message and adding the sender to the broadcast list

Block type	Drawer	Purpose
`if else`	Control	Depending on the message received, do different things.
=	Math	Determine whether `messageText` is equal to "joinFMDT."
`get messageText`	Drag out from MessageReceived event handler	Plug this into the = block.
`text` ("joinFMDT")	Text	Plug this into the = block.
`add items to list`	Lists	Add the sender's number to `BroadcastList`.
`get global BroadcastList`	Drag out from variable initialization block.	The list.
`get number`	Drag out from MessageReceived event handler	Plug this in as an item of `add items to list`.
`set BroadcastListLabel.Text to`	BroadcastListLabel	Display the new list.
`global BroadcastList`	Drag out from variable initialization block	Plug this in to set the `BroadcastListLabel.Text to` block.
`set Texting1.Message to`	Texting1	Prepare `Texting` to send a message back to the sender.
`text` ("Congrats, you...")	Text	Congratulate the sender for joining the group.

How the blocks work

The first row of blocks shown in *Figure 11-4* sets `Texting1.PhoneNumber` to the phone number of the message that was just received; we know we're going to respond to the sender, so this sets that up. The app then asks if the `messageText` was the special code, "joinFMDT." If so, the sender's phone number is added to the `BroadcastList`, and a congratulations message is sent. If the `messageText` is something other than "joinFMDT," the reply message repeats the invitation message. After the `if else` block, the reply message is sent (bottom row of the blocks).

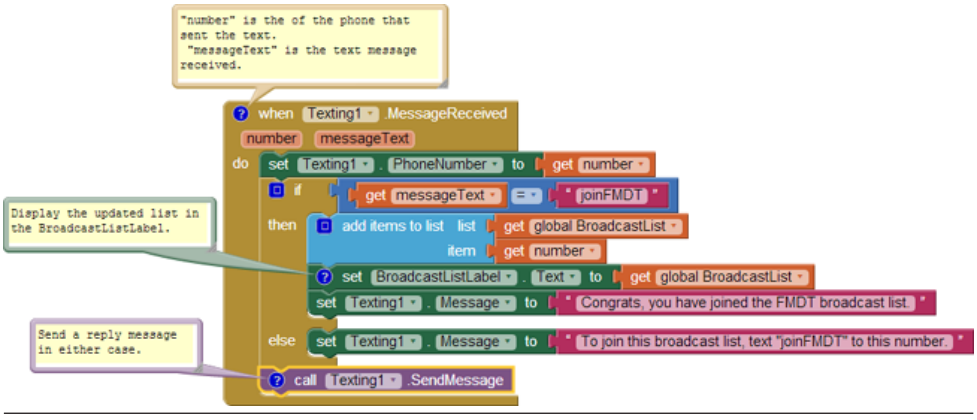


Figure 11-5. If the incoming message is "joinFMDT," add the sender to BroadcastList

Test your app *From a phone not running the app, send the text message "joinFMDT" to the phone running the app. You should see the phone number listed in the user interface under "Registered Phone Numbers." The "sending" phone should also receive the Congrats message as a text in reply. Try sending a message other than "joinFMDT," as well, to check if the invite message is still sent correctly.*

BROADCASTING THE MESSAGES

Next, you'll add the behavior so that the app broadcasts received messages to the numbers in `BroadcastList`, but only if the message arrives from a number already stored in that list. This additional complexity will require more control blocks, including another `if else` and a `for each`. You'll need an additional `if else` block to check if the number is in the list, and a `for each` block to broadcast the message to each number in the list. You'll also need to move the `if else` blocks from the previous

behavior and socket them into the `else` part of the new `if else`. All the additional blocks you'll need are listed in *Table 11-4*.

Table 11-4. Blocks for checking if the sender is in the group already

Block type	Drawer	Purpose
`if else`	Control	Depending on whether the sender is already in the list, do different things.
`is in list?`	Lists	Check to see if something is in a list.
`get global BroadcastList`	Drag out from variable initialization block	Plug this into the "list" socket of `is in list?`
`get number`	Drag out from MessageReceived event handler	Plug this into the "thing" socket of `is in list?`
`for each`	Control	Repeatedly send out a message to all members in the list.
`get global BroadcastList`	Drag out from variable initialization block	Plug this into the "list" socket of `for each`.
`set Texting1.Message to`	Texting1	Set the message.
`get messageText`	Drag out from the MessageReceived event	The message that was received and will be broadcast.
`set Texting1.PhoneNumber to`	Texting1	Set the phone number.
`get item`	Drag out from for each block	Hold the current item of the `BroadcastList`; it's a (phone) number.

How the blocks work

The app has become complex enough that it requires a *nested* `if else` block, which you can see in *Figure 11-5*. A nested `if else` block is one that is plugged into the socket of the `if` or `else` part of another, outer `if else`. In this case, the outer `if else` branch checks whether the phone number of the received message is already in the list. If it is, the message is relayed to everyone in the list. If the number is not in the list, the *nested* test is performed: the blocks check if the `messageText` is equal to "joinFMDT" and branch one of two ways based on the answer.

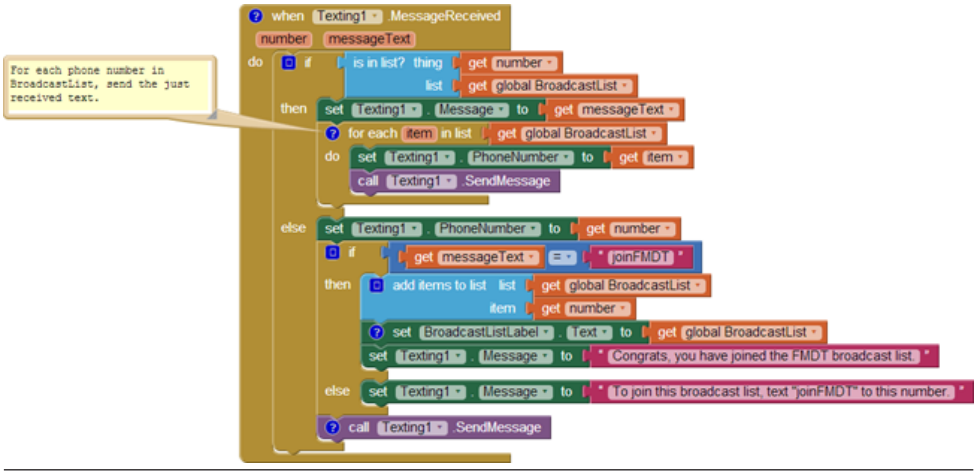


Figure 11-6. The blocks check if the sender is already in the group and broadcast the message if so

In general, `if` and `if else` blocks can be nested to arbitrary levels, giving you the power to program increasingly complex behaviors (see *Chapter 18* for more information on conditional blocks).

The message is broadcast by using a `for each` (within the outer `then` clause). The `for each` iterates through the `BroadcastList` and sends the message to each item. As the `for each` repeats, each succeeding phone number from the `BroadcastList` is stored in `item` (`item` is a variable placeholder for the current item being processed in the `for each`). The blocks within the `for each` set `Texting.PhoneNumber` to the current item and then send the message. For more information on how `for each` works, see *Chapter 20*.

Test your app *First, have two different phones register by texting "joinFMDT" to the phone running the app. Then, text another message from one of the phones. Both phones should receive the text (including the one that sent it).*

BEAUTIFYING THE LIST DISPLAY

The app can now broadcast messages, but the user interface for the app administrator needs some work. First, the list of phone numbers is displayed in an inelegant way. Specifically, when you place a list variable into a label, it displays the list with spaces between the items, fitting as much as possible on each line. So, the `BroadcastListLabel` might show the `BroadcastList` like this:

(+1415111-1111 +1415222-2222 +1415333-3333 +1415444-4444)

To improve this formatting, create a procedure named `displayBroadcastList` by using the blocks listed in *Table 11-5*. This procedure displays the list with each phone number on a separate line. Be sure to call the procedure from below the `add items to list` block so that the updated list is displayed.

Table 11-5. Blocks to clean up the display of phone numbers in your list

Block type	Drawer	Purpose
`to procedure` ("displayBroadcastList")	Procedures	Create the procedure (do not choose `to procedure result`).
`set BroadcastListLabel.Text to`	BroadcastListLabel	Display the list here.
`text` ("")	Text	Click `text` and then click Delete to create an empty text object.
`for each`	Control	Iterate through the numbers.
`get global BroadcastList`	Drag out from variable initialization block	Plug this into the "in list" socket of `for each`.
`set BroadcastListLabel.Text to`	BroadcastListLabel	Modify this with each of the numbers.
`join text`	Text	Build a text object from multiple parts.
`BroadcastListLabel.Text`	BroadcastListLabel	Add this to the label on each iteration of `for each`.
`text` ("\n")	Text	Add a newline character so that the next number is on the next line.
`get item`	Drag out from `for each` block.	The current number from the list.

How the blocks work

The `for each` in `displayBroadcastList` successively adds a phone number to the end of the label, as shown in *Figure 11-6*, placing a newline character (\n) between each item in order to display each number on a new line.

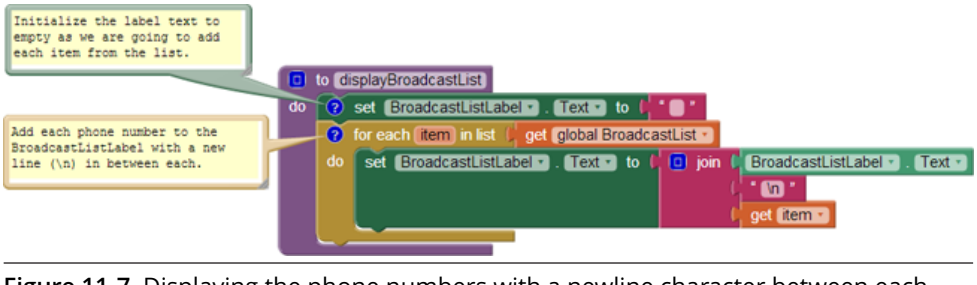


Figure 11-7. Displaying the phone numbers with a newline character between each

Of course, this `displayBroadcastList` procedure will not do anything unless you call it. Place a call to it in the `Texting1.MessageReceived` event handler, right below the call to `add item to list`. The call should replace the blocks that simply set the `BroadcastListLabel.Text` to `BroadcastList`. You can find the `call displayBroadcastList` block in the Procedures drawer.

Figure 11-7 shows how the relevant blocks within the `Texting1.MessageReceived` event handler should look.

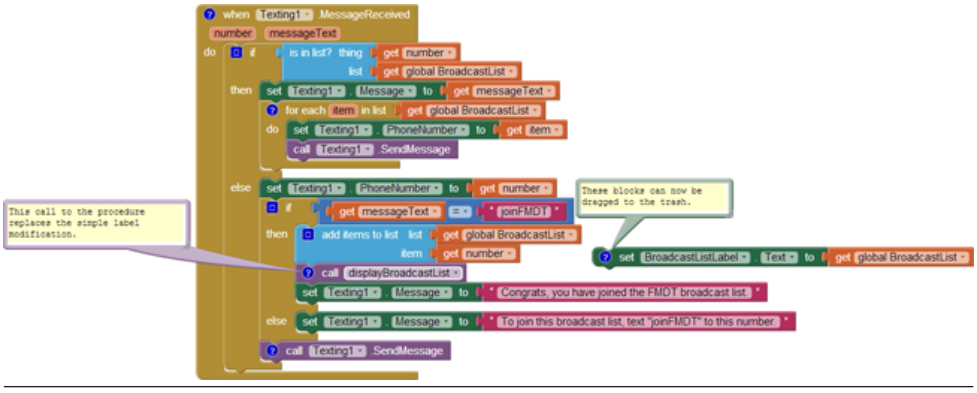


Figure 11-8. Calling the displayBroadcastList procedure

For more information on using `for each` to display a list, see *Chapter 20*. For more information about creating and calling procedures, see *Chapter 21*.

Test your app *Restart the app to clear the list and then have at least two different phones register (again). Do the phone numbers appear on separate lines?*

LOGGING THE BROADCASTED TEXTS

When a text is received and broadcast to the other phones, the app should log that occurrence so that the administrator can monitor the activity. In the Component Designer, you added the label `LogLabel` to the user interface for this purpose. Now, you'll code some blocks that change `LogLabel` each time a new text arrives.

You need to build a text that says something like "message from +1415111-2222 was broadcast." The number +1415111-2222 is not fixed data; instead, it is the value of the argument `number` that comes with the `MessageReceived` event. So to build the text, you'll concatenate the first part, "message from," with a `get number` block and finally with the last part of the message, the text "broadcast."

As you've done in previous chapters, use `join` to concatenate the parts by using the blocks listed in *Table 11-6*.

Table 11-6. Blocks to build your log of broadcasted messages

Block type	Drawer	Purpose
`set LogLabel.Text to`	LogLabel	Display the log here.
`join`	Text	Build a text object out of multiple parts.
`text` ("message from")	Text	This is the report message.
`get number`	Drag out from MessageReceived event handler	The sender's phone number.
`text` ("broadcast\n")	Text	Add the last part of "message from 111–2222 broadcast" and include newline.
`LogLabel.Text`	LogLabel	Add a new log to the previous ones.

How the blocks work

After broadcasting the received message to all of the numbers in `BroadcastList`, the app now modifies the `LogLabel` to add a report of the just-broadcasted text, as shown in *Figure 11-8*. Note that the message is added to the beginning of the list instead of the end. This way, the most recent message sent to the group shows up at the top.

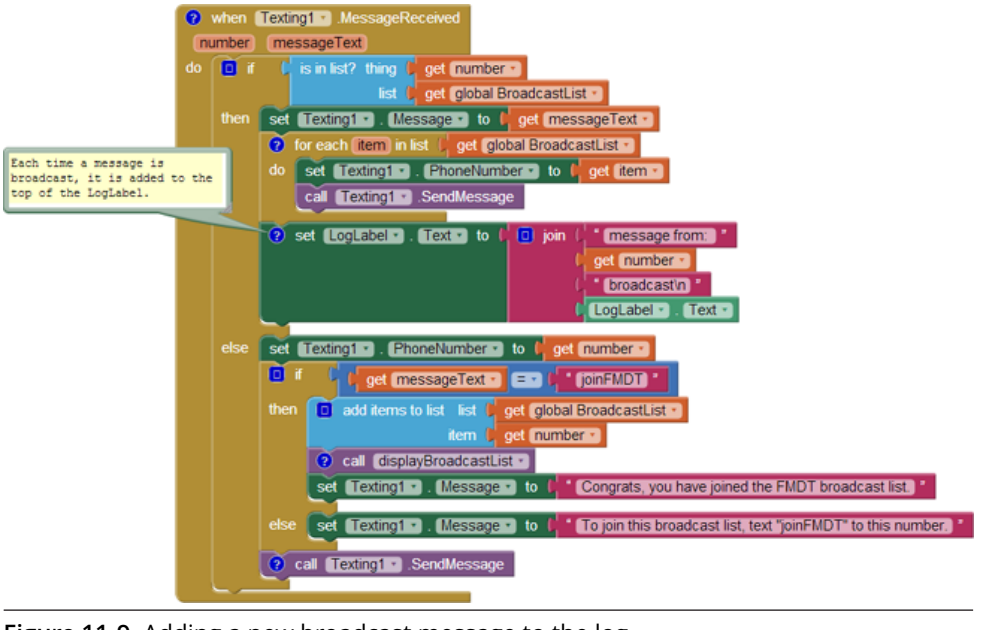


Figure 11-9. Adding a new broadcast message to the log

The `join` block creates new entries of the form:

```
message from: 111-2222 broadcast
```

Each time a text is broadcast, the log entry is *prepended* (added to the front) to the `LogLabel.Text` so that the most recent entries will appear on top. The way you organize the `join` block determines the ordering of the entries. In this case, the new message is added with the top three sockets of `join`, and `LogLabel.Text`—which holds the existing entries—is plugged into the last socket.

The "\n" in the text "broadcast\n" is the newline character that causes each log entry to display on a separate line:

```
message from: 1112222 broadcast
message from: 555-6666 broadcast
```

For more information about using `for each` to display a list, see *Chapter 20*.

STORING THE BROADCASTLIST IN A DATABASE

Your app sort of works, but if you've completed some of the earlier tutorials, you've probably guessed that there's a problem: if the administrator closes the app and relaunches it, the broadcast list will be lost and everyone will have to register again. To fix this, you'll use the `TinyDB` component to store and retrieve the `BroadcastList` to and from a database.

You'll use a similar scheme to that which you used in the MakeQuiz app (*Chapter 10*):

- Store the list to the database each time a new item is added.
- When the app launches, load the list from the database into a variable.

Start by coding the blocks listed in *Table 11-7* to store the list in the database. With the `TinyDB` component, a tag is used to identify the data and distinguish it from other data stored in the database. In this case, you can tag the data as "broadcastList." You'll add the blocks in the `Texting1.MessageReceived` event, under the `add items to list` block.

Table 11-7. Blocks to store the list with TinyDB

Block type	Drawer	Purpose
`TinyDB1.StoreValue`	TinyDB1	Store the data in the database.
`text` ("broadcastList")	Text	Plug this into the "tag" slot of `StoreValue`.
`get global BroadcastList`	Drag out from variable initialization block	Plug this into the "value" slot of `StoreValue`.

How the blocks work

When a "joinFMDT" text comes in and the new member's phone number is added to the list, `TinyDB1.StoreValue` is called to store the `BroadcastList` to the database. The tag (a text object named `broadcastList`) is used so that you can later retrieve the data. *Figure 11-9* illustrates that the value that is called by `StoreValue` is the variable `BroadcastList`.

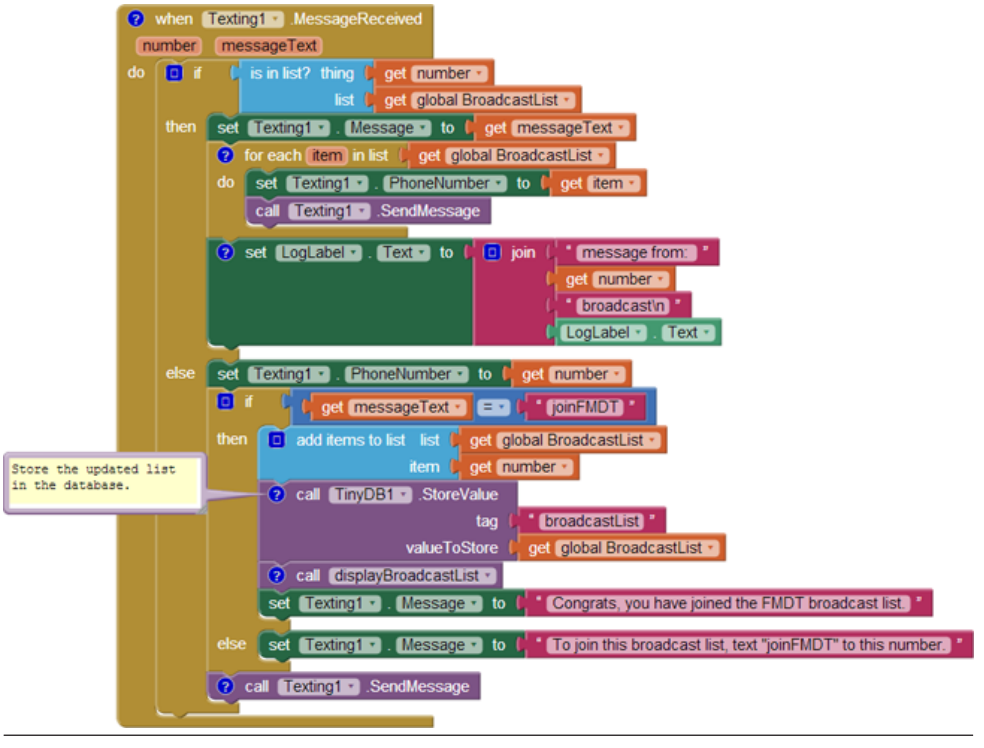


Figure 11-10. Calling TinyDB to store the BroadcastList

LOADING THE BROADCASTLIST FROM A DATABASE

Add the blocks listed in *Table 11-8* for loading the list back in each time the app launches.

Table 11-8. Blocks to load the broadcast list back into the app when it launches

Block type	Drawer	Purpose
`Screen1.Initialize`	Screen1	Triggered when the app launches.
`TinyDB1.GetValue`	TinyDB1	Request the data from the database.
`text` ("broadcastList")	Text	Plug this into the "tag" socket of **`GetValue`**.

Block type	Drawer	Purpose
`call displayBroadcastList`	Procedures	After loading data, display it.

When the app begins, the `Screen1.Initialize` event is triggered, so your blocks will go in that event handler.

How the blocks work

When the app begins, the `Screen1.Initialize` event is triggered. The blocks shown in *Figure 11-10* request the data from the database with `TinyDB1.GetValue`.

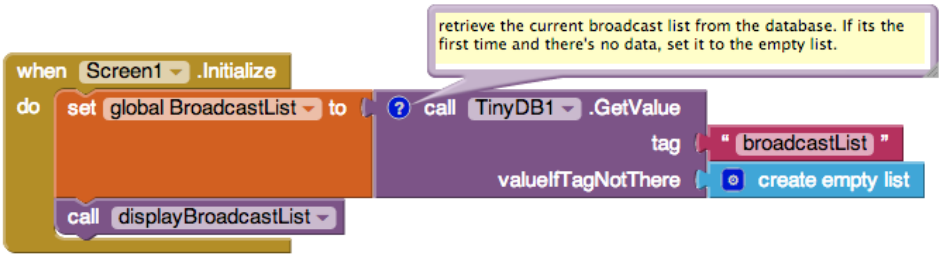


Figure 11-11. Loading the BroadcastList from the database

You call `TinyDB.GetValue` by using the same tag you used to store the list (`broadcastList`). In the general case, the previously stored list of phone numbers will be returned and placed in the variable `BroadcastList`. But `TinyDB.GetValue` provides a socket, `valueIfTagNotThere`, for specifying what the block should return if there is not yet data in the database for that tag, as will happen the first time this app is run. In this case, an empty list is returned.

Test your app *You can use live testing for apps that modify the database, but do it carefully. In this case, text the app with another phone to add numbers to the BroadcastList, and then restart the app. You can restart in live testing mode by switching to the designer and modifying some property, even something such as changing the font of a label. Note that to fully test database apps you need to package and truly download the app to a phone (choose "Build > App (save apk to my computer"). After you've downloaded your app, use your other phones to send a text to join the group and then close the app. If the numbers are still listed when you relaunch the app, the database part is working.*

The Complete App: Broadcast Hub

Figure 11-11 illustrates the blocks in the completed Broadcast Hub app.

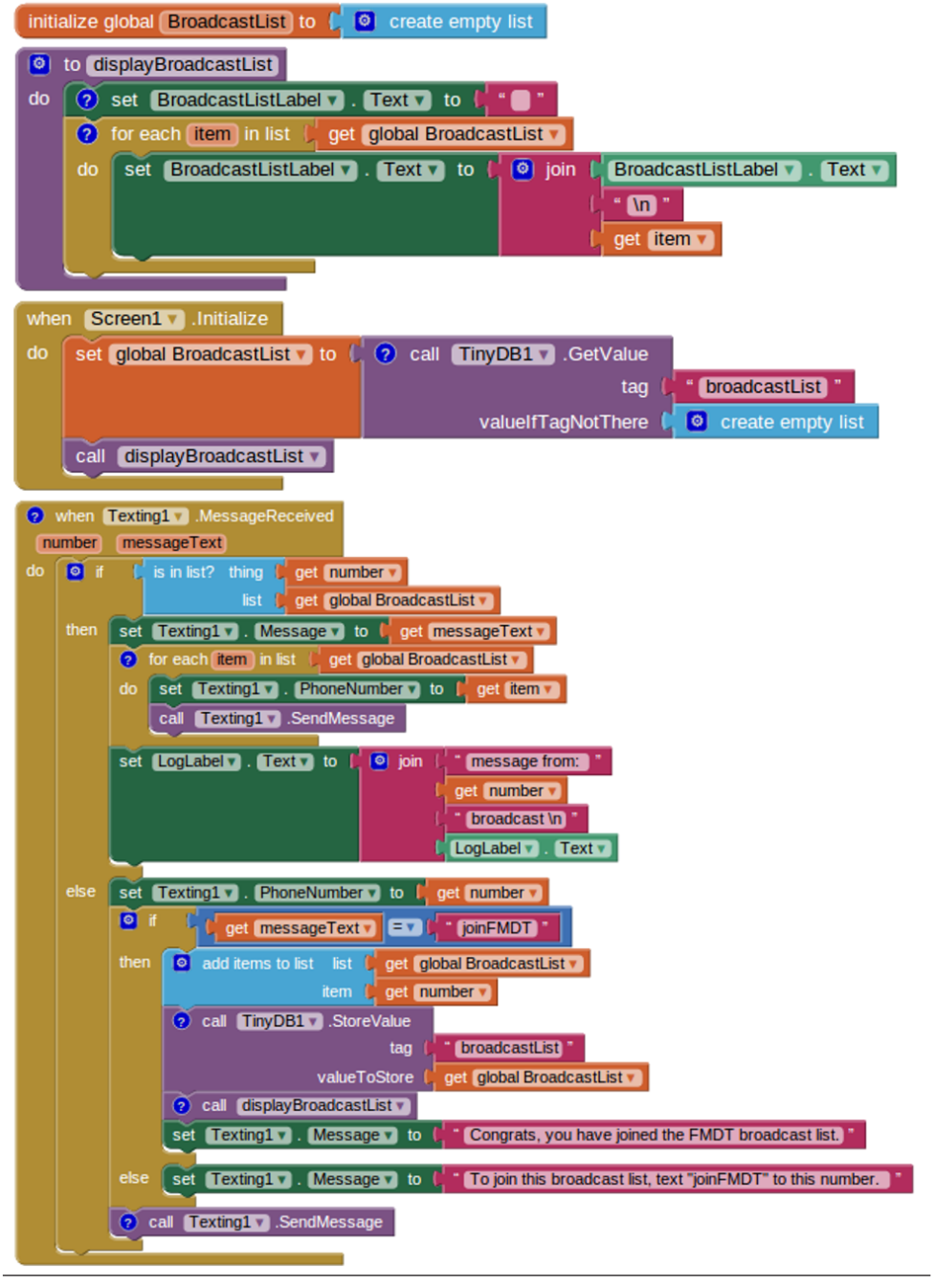


Figure 11-12. The complete Broadcast Hub app

Variations

After you've celebrated building such a complex app, you might want to explore some of the following variations:

- The app broadcasts each message to everyone, including the phone that sent the message. Modify this so that the message is broadcast to everyone but the sender.
- Allow client phones to remove themselves from the list by texting "quit" to the app. You'll need a `remove from list` block.
- Give the hub administrator the ability to add and remove numbers from the broadcast list through the user interface.
- Let the hub administrator specify numbers that should not be allowed into the list.
- Customize the app so that anyone can join to receive messages, but only the administrator can broadcast messages.
- Customize the app so that anyone can join to receive messages, but only a fixed list of phone numbers can broadcast messages to the group.

Summary

Here are some of the concepts we covered in this tutorial:

- Apps can react to events that are not initiated by the app user, such as a text being received. This means that you can build apps in which your "users" are on a different phone.
- You can use nested `if else` and `for each` blocks to code complex behaviors. For more information on conditionals and `for each` iteration, see *Chapter 18* and *Chapter 20*, respectively.
- You can use the `join` block to build a text object out of multiple parts.
- You can use `TinyDB` to store and retrieve data from a database. A general scheme is to call `StoreValue` to update the database whenever the data changes and call `GetValue` to retrieve the database data when the app starts.