



# Developer Guide to Building EachScape Blocks

February 2012

## Contents

Introduction	1
EachScape Architecture	2
Block Architecture	4
Creating Blocks in the Builder	7
Creating iOS Blocks	9
Creating Android Blocks	12

## Introduction

EachScape is a powerful application development system that enables app producers to take a building-block approach to developing mobile applications. It offers a drag and drop environment that can be used by app producers to create and manage high-end mobile applications without coding. EachScape enables its users to build custom applications with great design and functionality for iOS, Android, and HTML5. The applications can run on phones, tablets, and connected TVs that support those platforms.

At the simplest level, an EachScape block is code that plays by a certain set of rules, gathers and displays data, and presents and manages UI components. Blocks are components of the EachScape development environment, which is called the Builder. The Builder can be used to create applications for devices running iOS, Android, or HTML5. App producers create their mobile apps in the EachScape Builder using blocks. App producers literally assemble mobile applications largely by dragging and dropping blocks in the Builder (with a little bit of additional configuration in the same interface).

In order for app producers to create in EachScape, they need blocks to assemble. By offering your functionality as a block in EachScape, you can offer EachScape customers (app producers) the ability to easily incorporate your unique functionality into their mobile apps.

Blocks are code that implements a UI element based on certain rules. The code in a block can gather data from many sources, display and gather information from the end-user, react to system events such as tapping, pinching, and stretching, invoke services on the device or the Internet, and fire events that can be handled by scripted actions inside EachScape's application design environment. The code in a block intermediates between the device's native operating system and the EachScape development and runtime environment, providing instructions for the device to execute application functions and transfer feedback from device events to the application.

Some blocks were created by EachScape developers as part of the EachScape platform. Independent developers (like you) can also create blocks and offer them to EachScape customers through the Block Marketplace.

Once a block is added to the Block Marketplace, it generally becomes available to all EachScape customers, ranging from large enterprises, including media and consumer packaged goods companies, to small- and mid-size businesses. Blocks can be offered via license or sale, meaning that the developer can grant the right to use the block to EachScape customers (in other words, the block is available to customers who license that block from you). EachScape creates a business relationship with developers contributing blocks to the Builder and shares revenue with contributors.

Blocks are part of the EachScape environment, which creates a native version of the applications it generates. This means that each block is available only on each native platform you provide code for. To fully support the block on EachScape, you should write a version of it for each native platform.

This guide is intended to help developers understand the following concepts:

- How blocks fit into the cross platform web-based development environment that EachScape app producers use to create applications
- How to code blocks in iOS and Android development environments

EachScape offers a platform-independent way of creating applications. After reading this guide, most developers familiar with native mobile device operating systems will understand the skills needed and amount of work required to create a block.

This guide is broken into the following sections:

- **EachScape Architecture:** Explains what EachScape is and how it works
- **Block Architecture:** Provides a high-level overview of the process for developing a block and offering it in the Block Marketplace
- **How the Builder Uses Blocks:** Explains how the EachScape app development environment (called the Builder) uses blocks
- **How to Create Blocks:** Explains how to create blocks in the Builder
- **Creating iOS Blocks:** Provides guidance about writing and implementing blocks in the Apple iOS environment
- **Creating Android Blocks:** Provides guidance about writing and implementing blocks in the Google Android environment

## EachScape Architecture

In order to talk about EachScape, we need to define three types of users:

- **Block developers:** People who code blocks, which can be made available to app producers
- **App producers:** People who use blocks in EachScape to assemble mobile apps
- **App users:** People who use apps created by app producers on their devices

The EachScape architecture includes the following elements, all of which can be manipulated in the Builder:

- **View:** A single page of an app. A view fills the entire screen of the device, replacing any previously displayed views
- **Layer:** Each view can have multiple layers. The app user can see the layers one at a time, based on events she generates. Layers can be turned on or off by the app producer. A block can exist on either a view or on a layer
- **Event:** Two types of events must be distinguished in an EachScape application. **System events** come from user actions such as taps, pinches, swipes, and so on. The code in a block registers a handler that captures the system event and then does something to change the display or to fire an **EachScape event** to trigger some script action in the

EachScape application. An EachScape event, which we just call events in this document, can be connected to script actions defined by the app producer

- **Script action:** Script actions are defined in the Builder. They can invoke different types of functionality, including calls to external services (such as “log onto Facebook”), displaying another view, or displaying or hiding a layer. A script action is selected from a menu in the Builder by an app producer

*Communication from a block is unidirectional. A block can invoke an event, and an event can invoke script actions. But a script action cannot make a block do anything. A script action can set a global variable, which, combined with another script action called after setting the variable, refreshes the block. A script action could make a layer disappear and make a new block appear, but it cannot directly affect a block. Blocks can be invoked only by system events.*

- **Master Block:** A Master Block defines a block’s basic characteristics. Blocks are instances of a Master Block

EachScape has a global namespace for variables, but does not deploy parameter passing. If your script action needs data or a variable from a block, the block must place that data in a global variable.

## Block Architecture

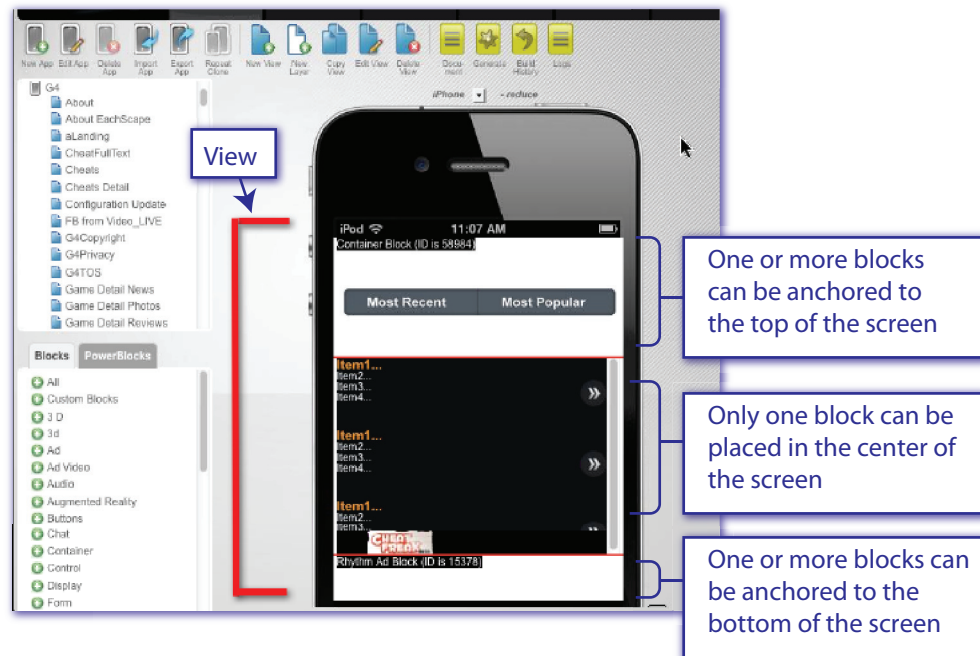
In this section, we explore the basic structure of a block and describe how it fits into the EachScape architecture. Blocks are developed in the Builder.



*The Builder*

## Block Functions

Blocks can appear in three areas of the screen. Blocks above the top red line are anchored to the top of the screen. Blocks below the bottom red line are anchored to the bottom of the screen. A single block can exist between the two red lines.



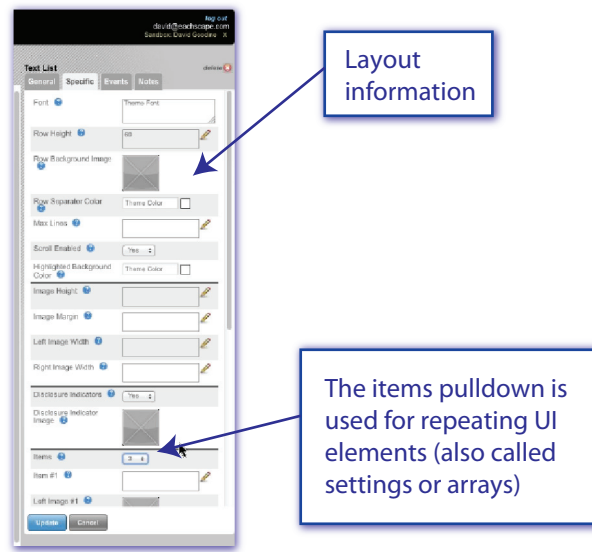
*A view in the Builder*

## Parts of a Block

Blocks are implemented based on a **model-view-controller** architecture.

The **model** is the data used in the block. The data comes from the General and Specific tabs in the Builder as well as from external sources. These tabs, located on the right side of the screen, provide generic parameters for every block in the Builder. The **General** tab covers attributes such as name, background image, fill color, and dimensions.

The **Specific** tab lets the app producer customize the look of the block based on the parameters specified by the block producer in the Master Block. It also lets you define repeating UI elements. The example block in this guide, [TextListBlock](#), has three repeating UI elements.



*The Specific tab*

You can also specify a data source using this tab by inserting the filename or the URL of a spreadsheet or other data resource.

The **controller** is the code that performs the work of the block, including displaying the block, handling system-level events, and prompting script actions.

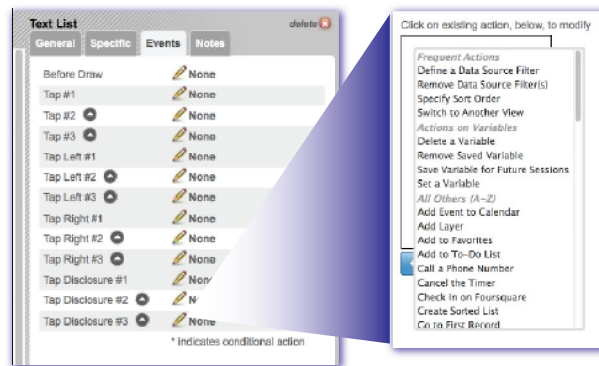
*To understand what the controller does, it is helpful to look ahead to the implementation of a block for a moment. The controller for a block in iOS is implemented through a method called `refreshView`.*

*When a block is loaded at runtime, a number of actions take place. The canvas is readied to accommodate the block, and data is pulled from external sources to serve the block. When the block is loaded and all of the data and the canvas are ready, then `refreshView` is called to build the view of the block. Handlers for user events, such as tap, pinch, and swipe, are registered automatically. The code embedded in the handlers obtains data for the block and allows it to react to events. The handlers can also fire events, which can invoke script actions.*

The **Events tab** in the Builder associates user interface events with script actions. Clicking on the small pencil icon launches the script action editor.

On the Events tab on the following screen, there are three repeated UI elements so that actions for each group of UI elements can be defined.



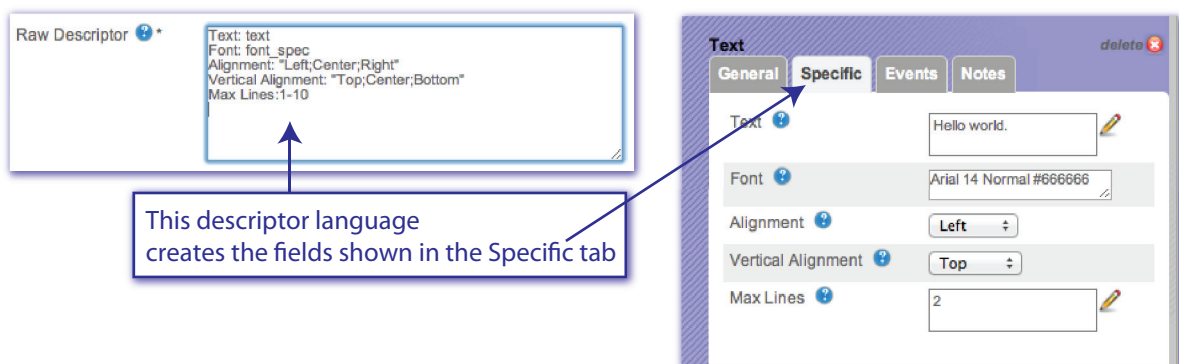


Clicking the pencil icon opens a drop-down list of actions

*Events tab leads to the script action editor*

## Creating Blocks in the Builder

So far, we've described the architecture of a block and how a block looks to the app producer. The Builder uses a descriptor language to define the structure of blocks. This language is entered in the **Master Block** section in the Builder, an area currently accessible only to EachScape developers. The descriptor language defines a new block from the Builder's perspective. The parameters under the General tab are the same for all blocks. The descriptor language entered in the Raw Descriptor field determines what appears under the Specific tab in the Builder.



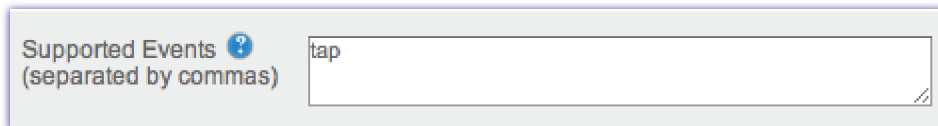
*Descriptor language is used to define what fields the app producer sees on the Specific tab*

The descriptor language sets specific parameters for your block, including text fonts, alignment, and number of lines. Default settings and item-level help for the Specific tab are also defined here.

If a block needs to set a variable, the field type variable name is entered in the Raw Descriptor field. For example, to set the variable `Save Username`, the entry in the Raw Descriptor

field would include `Save Username:variable name`. It is important to set variables before firing an event (otherwise, the event will not have any data to display).

Events that the block can fire are defined in the **Supported Events** field.



*Supported Events field*

**Remember:** *the Builder is a design time environment. It allows an app producer to specify how the app should look and how the blocks should be connected to other components.*

Once an app is defined in the Builder, the **Generate** interface uses the description of the blocks to create a meta-level description of the application as it was described in the Builder. This description is then sent to a **generator server** in the EachScape environment. The generator uses the metadata to assemble the right code for each platform. Code to implement each block must be part of the code base for each platform. The later sections of this guide explain how to write code for iOS and Android platforms.

When the generator has assembled the application for each platform, you will receive an email with a link to the EachScape server, from which the created app can be downloaded.

*Checklist for Creating a Block in the Builder:*

- *Understand what can be specified on the General tab*
- *Decide which parameters you want the app producer to set on the Specific tab*
- *Decide which events the block will handle. These events appear on the Events tab. From this point onward, the app producer will be able to add script actions to these events*
- *Use the Master Block interface to enter the descriptor language to describe the block and the fields that will appear on the Specific tab in the Builder*
- *Use the Master Block interface to define any global variables you need*
- *Establish communication with external data sources*

## Creating iOS Blocks

In the Apple iOS operating system, a block is implemented in two classes using the Model-View-Controller scheme discussed above:

- **Block (Model):** Acts as a collection of general and specific properties of an application
- **Controller (Controller):** The code that performs the work of the block, including displaying the block, handling system-level events, and prompting script actions

You may wonder why you don't have to implement a view class. That's because iOS provides view classes that work for 90% of the cases you'll encounter. In those other 10%, you may want to consider implementing a custom view class of your own.

Both the block and controller classes are implementations of larger, abstract classes. The Generate process links both the .h files and the .m files via their names to execute the block's instructions. This is why it's very important to correctly name all of the elements of your block, so that they will be picked up correctly by the Generate process.

## iOS To-do List

In this section, we describe how to enable block code in iOS.

**Implement the block classes you will need.** Both of these classes are subclasses of abstract classes provided by EachScape, *ESBlock.m* and *ESBlock.h*. Both classes must be named properly in order to function, e.g., *ES<name>Block.h*. For example, to create a block named *TextList*, you would create class names *ESTextListBlock.h* and *ESTextListBlock.m*.

```
//
// ESTextListController.h
// EachScape
//
// Created by Mark Smith on 5/18/09.
// Copyright 2009 EachScape. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "ESBlockViewController.h"
#import "ESImageView.h"

@class ESTextListBlock; ← A class called ESTextListBlock

@interface ESTextListController : ESBlockViewController <UITableViewDelegate,
@private
    UITableView *_tableView;
    ESTextListCellConfig *_cellConfig;
}

@end
```

*ESTextListBlock* extends the abstract class *ESBlock*.

```
//
// ETextListBlock.h
// EachScape
//
// Created by Mark Smith on 5/18/09.
// Copyright 2009 EachScape. All rights reserved.
//

#import "ESBlock.h"
#import "ESFontSpec.h"
#import "ESSetting.h"

@interface ETextListBlock : ESBlock {
@private
}
```

ETextListBlock extends the abstract class ESBlock

**Implement the Controller classes (controller.h and controller.m).** Again, both classes must be named properly, e.g., *ES<name>controller.h*. For example, to control a block named *TextListBlock*, use *ETextListController.h* and *ETextListController.m* as class names. *ETextListController* extends the abstract class *ESBlockViewController*.

```
//
// ETextListController.h
// EachScape
//
// Created by Mark Smith on 5/18/09.
// Copyright 2009 EachScape. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "ESBlockViewController.h"
#import "ESImageView.h"

@class ETextListBlock;

@interface ETextListController : ESBlockViewController <UITableViewDelegate,
    UITableViewDataSource, ESImageViewDelegate> {
@private
    UITableView *_tableView;
    ETextListCellConfig *_cellConfig;
}

@end
```

ETextListController extends the abstract class ESBlockViewController

The controller class contains code that must be changed in order to properly link it to its intended companion block. By doing this, a new class is created. In this case, *Controller.h* links to *Block.h*.

**Implement *refreshView*.** The *refreshView* method is called once all of the data sources have been loaded into the block, and the view is ready to be drawn. *refreshView* is where the block does most of its work to construct and configure the components visible in the block. The code in the *refreshView* method draws the canvas, based on instructions from the blocks. System-level events such as pinching, swiping, or tapping can also trigger a call to *refreshView*.

```
#pragma mark Construction
```

```
- (void)dealloc {
    self.tableView = nil;
    self.cellConfig = nil;
```

```
    [super dealloc];
}
```

When `refreshView` is called, the block is redrawn

```
- (void)refreshView {
    [self.contentView removeAllSubviews];
```

**Decide whether you will use the settings object.** The settings object is an optional mechanism used when one of the things in the descriptor of a block is an array. A button row, for example, may have three buttons and you need to set the button for position one, two, and three in the array. A text list has multiple entries. The settings object handles situations like this. If you don't have a repeating user interface element, you don't need a settings object.

If you do need a settings object, in the *block.m* file, make the `settingClass` method return the setting class.

```
#pragma mark ESBlock
```

```
+ (NSString *)type {
    return @"textList";
}
```

The setting method must return the setting class

```
+ (Class)settingClass {
    return [ESTextListSetting class];
}
```

In the *block.h* file, the setting class must extend the **ESSetting** class. Then, in the *block.m* file, you create the implementation of the setting class.

```
@implementation ESTextListSetting
```

```
#pragma mark Public
```

```
- (NSString *)item{
    return [self expandKey:@"item"];
}
```

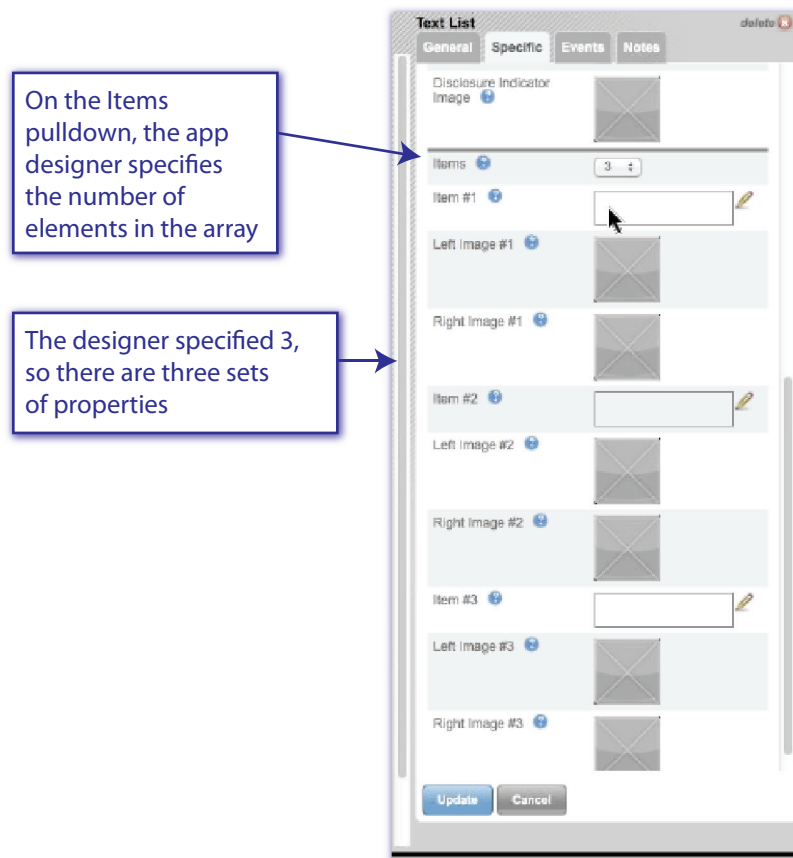
In the *block.m* file, you create the implementation of the setting class

```
- (NSString *)leftImage {
    return [self expandKey:@"leftImage"];
}
```

```
- (NSString *)rightImage {
    return [self expandKey:@"rightImage"];
}
```

```
@end
```

Note that no array is declared explicitly. The abstract classes look at the metadata and create an array based on the number of elements in the descriptor, as specified by the app producer on the Specific tab in the Builder.



*Specify number of elements in the array*

In the BlockController, you can get the settings array using `self.block.settings`.

**Decide how the block will use data.** Blocks often get information from one or more data sources. Information on how blocks get data is beyond the scope of this document.

**Decide what system-level events you will handle.** To enable the block to handle system-level events, the iOS code must register a recognizer.

## Creating Android Blocks

Creating Android blocks follows a similar pattern to creating iOS blocks. In Android, a block is implemented using two classes in the Model-View-Controller scheme. The two classes are:

- **Block:** This acts as a collection of general and specific properties of an application. In Android, the block file is called *Block.java*
- **Controller:** The code that performs the work of the block, including displaying the block, handling system-level events, and prompting script actions. In Android, the controller file is *BlockViewController.java*

Both of these classes are subclasses of larger, abstract classes provided by EachScape. The block and controller both override methods from the abstract classes while the controller implements the code.

The header file, *Block.java*, contains declarations while the file is the code itself. Both *Block.java* and *BlockViewController.java* link to names in the app's metadata that are passed to the Generate process to execute the block's instructions. This is why it's very important to correctly name all of the elements of your block so that they will not be misunderstood once the Generate process is underway.

## Android To-Do List

In this section, we describe how to write blocks in Android.

Suppose you would like to create a simple block that displays a text list. You would begin by extending the *Block.java* class. The naming convention is *<name>Block.java*. If you're creating a block called *TextListBlock*, the new class would be called *TextListBlock.java*. You extend *Block.java* by implementing two methods: *getType* and *getSettingClass* (optional).

**Implement *getType*.** This method establishes the type of block you want to create. In this case, since we are creating a text list block, we would change "*<blockName>*" to return *Block.TEXT\_LIST*, a static String with a value of "textList", which is the block's name in the Builder, to be used as a reference.

```
public class TextListBlock extends Block {  
    @Override  
    public String getType() {  
        return Block.TEXT_LIST;  
    }  
}
```

TextListBlock extends Block

getType() method returns Block.TEXT\_LIST

**Create the setting class, if desired.** A block has the option of having a collection of UI components displayed multiple times with different values for each collection. The Setting class contains the information for each collection. These collections of UI components can also have their own individual events for the collection as a whole or for the individual UI components.

The number of collections can be set on the Specific tab in the Builder, if settings are enabled. (Settings are not required for a block.)

You retrieve settings from the Specific tab in the Builder using code like this:

```
mSettingsList = block.getSettings();
```

**Implement *getSettingClass*.** The naming convention is similar to that of other classes, e.g., *<name>Setting.java*. To create a text block setting class, you would use *TextListSetting.java*. Code without a setting class will return null for *getSettingClass*; code with a setting class will return the setting class's name (in this case, *TextListSetting.java*).

**Extend the `BlockViewController` Class.** Now you will need to implement the `Controller` class for your text object. The format is `<name>ViewController.java`. In this example, we create a class called `TextViewController.java`.

```
public class TextListController extends BlockViewController {
    private ListView mListView;
    private List<Setting> mSettingsList;
    private AppViewControllerActivity mContext;
    private FontSpec mFontSpec;
```

TextListController  
extends  
BlockViewController

**Implement `getViewLayout`.** `getViewLayout` establishes the UI view by returning the layout components that will be drawn onscreen, such as scroll bars, text boxes, and labels. Override `getViewLayout` to set the UI view you want. Additionally, all handlers for system-level events are registered in `getViewLayout`.

Use `getViewLayout` to set the UI view

```
@Override
public View getViewLayout(AppViewControllerActivity context) {
    mContext = context;
    mListView = new ListView(context);
    mListView.setAdapter(new TextListAdapter());
    mListView.setOnItemClickListener(new OnItemClickListener() {
        @SuppressWarnings("rawtypes")
        public void onItemClick(AdapterView parent, View v, int position, long id) {
            //Handle the setting event
            mSettingsList.get(position).handleEvent(mContext, EventHandler.TAP, v);
        }
    });
};
```

**Implement `setLayoutBackground`.** This sub-method reads the information about the background the app producer sets in the General tab in the Builder.


```
setLayoutBackground(context, mListView, block);
return mListView;
}
```

setLayoutBackGround reads the  
information that the app designer  
set in the General tab (here, mListView)

**Implement `refreshView`.** The `refreshView` method is used to refresh the content in the layout with the latest information.



```
}  
  
@Override  
public void refreshView() {  
    mSettingsList = block.getSettings();  
    if(mListView != null) {  
        TextListAdapter adapter = (TextListAdapter)mListView.getAdapter();  
        adapter.notifyDataSetChanged();  
        mListView.setSelectionAfterHeaderView();  
    }  
}
```



When refreshView is called,  
the block is redrawn


**Set up script actions.** Script actions can be associated with the collection of UI components for a setting or with an event associated to an individual UI component within the collection.

## Android Blocks and Dimension Scaling

Android devices are made by many manufacturers, and they have varying screen sizes even within their product lines. You may wish to set a standard height for your graphical items on all screens, or, you can control the dimensions of the screen display on the application through the block. For example, you can set an absolute setting for the height of a row by specifying that `getRowHeight` return 44 pixels.

There's another option, however. The `getScaledDimension` method converts absolute pixel dimensions into the dimensions and resolution needed for devices that are larger or smaller than the original design. It allows the display to resize automatically to fit the screen of the device on which the application is running:

```
public int getRowHeight() {  
    int rowHeight = Util.zeroIfNotValid(expandKey("rowHeight"));  
  
    if(rowHeight == 0)  
        return 0;  
  
    return AppResource.getInstance().getScaledDimension(rowHeight);  
}
```



`getScaledDimension`  
allows the app to scale  
appropriately for the device