Extracted from:

Programming Your Home

Automate with Arduino, Android, and Your Computer

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

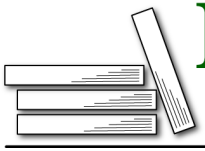
The Pragmatic Programmers

Programming Your Home

Automate with Arduino, Android, and Your Computer

Mike Riley

Edited by Jacquelyn Carter



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-690-6
Printed on acid-free paper.
Book version: P1.0—February 2012

7.4 Writing the Code for the Web Client

For the Web-enabled light switch, we will create a simple Ruby on Rails project to manage the user interface interaction first via a web browser. We won't spend a lot of time on the user interface, though, since that will ultimately be the job of the custom Android application we will create after the web interface is functionally tested.

Rails runs optimally on Mac or Linux computers, and it is already installed by default on Mac OS X 10.6. However, it is not the latest version. Because this project requires Rails 3.0 or higher, the instructions are not applicable to older versions of the framework. Follow the instructions on the Ruby on Rails website to get the latest Rails release running on your computer.

With the Rails web framework installed, create a new directory and switch to that directory before creating the new Rails project:

```
> mkdir ~/projects/ruby/rails/homeprojects/
> cd ~/projects/ruby/rails/homeprojects
> rails new x10switch

      create
      create  README
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      create  app/controllers/application_controller.rb
      create  app/helpers/application_helper.rb
      create  app/mailers
      create  app/models
      ...
      create  vendor/plugins
      create  vendor/plugins/.gitkeep
```

Next, change into the new `x10switch` directory and create a new controller called `command` with an action called `cmd()` to manage the interaction between the web interface and the Heyu terminal application.

```
> cd x10switch
    > rails generate controller Command cmd

          create  app/controllers/command_controller.rb
           route  get "command/cmd"
          invoke  erb
          create    app/views/command
          create    app/views/command/cmd.html.erb
```

```
      invoke  test_unit
      create    test/functional/command_controller_test.rb
      invoke  helper
      create    app/helpers/command_helper.rb
      invoke    test_unit
      create      test/unit/helpers/command_helper_test.rb
```

Then, locate the app/controllers/command_controller.rb file and check for the on and off parameters and execute the appropriate action:

```
class CommandController < ApplicationController
  def cmd
    @result = params[:cmd]

    if @result == "on"
      %x[/usr/local/bin/heyu on h3]
    end

    if @result == "off"
      %x[/usr/local/bin/heyu off h3]
    end
  end
end
```

The %x is a Ruby construct to execute an application with command-line arguments. Hence, %x[/usr/local/bin/heyu on h3] tells Heyu to send an on command code to the H3 house code X10 switch. Likewise, the %x[/usr/local/bin/heyu off h3] tells that same switch to turn off.

Next, edit the app/views/command/cmd.html.erb document and replace its placeholder contents with the following single line of embedded Ruby code to display the results of the On and Off request:

```
The light should now be <%= @result %>.
```

While we could go much further with this Rails application, dressing it up with a nice user-friendly interface accessed from the public/index.html file as well as providing more verbose output of the result of the action, I will leave that exercise for the aspiring reader. Since we will ultimately be controlling the switch from a native mobile client application, there's little incentive to invest time in whipping up a sparkly web UI when it will hardly ever be seen.

Finally, edit the config/routes.rb file and replace the get "command/cmd" with the following:

```
match "/command/:cmd", :to => 'command#cmd'
```

This instructs the Rails application on how to route incoming command requests to execute the on/off actions. Save your work and get ready to rumble!

If you're setting up a newer version of Rails (such as Rails 3.1) on a Linux system, you may also need to install a few package dependencies (or gems as they're known in Ruby parlance) in order for Rails to run. Just edit the Gemfile file that was generated in the x10switch directory and add the following:

```
gem 'execjs'
gem 'therubyracer'
```

Save the changes and then run this command:

```
> bundle install
```

This will download and install the extra files used by the Rails 3.1 JavaScript processing engine. With these two gems successfully installed, you're ready to run and test out the X10switch Rails application.

7.5 Testing Out the Web Client

With the X10 computer interface working and plugged into the serial port of the computer, fire up a development server of the Rails 3 code via this:

```
> cd ~/projects/ruby/rails/homprojects/x10switch
> rails s

=> Booting WEBrick
=> Rails 3.0.5 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2011-03-18 16:49:31] INFO  WEBrick 1.3.1
[2011-03-18 16:49:31] INFO  ruby 1.8.7 (2009-06-12) [universal-darwin10.0]
[2011-03-18 16:49:31] INFO  WEBrick::HTTPServer#start: pid=10313 port=3000
```

Open a web browser on the local machine and enter the following:

http://localhost:3000/command/on

If everything is coded correctly, you should see The light should now be on. in the browser window, as shown in Figure 25, *The browser should indicate the proper status of the light*, on page 8.

More importantly, Heyu should have executed the on command for the X10 device coded with the H3 house code. In other words, the light should have turned on. Turn the light off by submitting the off command:

http://localhost:3000/command/off

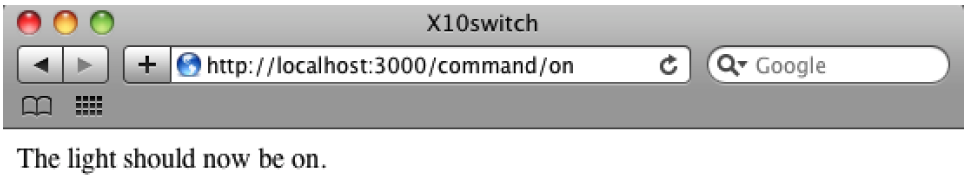


Figure 25—The browser should indicate the proper status of the light.

If the light turned off, congratulations! You have wired up and programmed everything correctly. When you're ready to expand the Rails application to handle even more commands, just add more `if @result ==` statements to the `CommandController` class containing the command you want Heyu to transmit. These commands could range from dimming lights to 30 percent, turning an appliance on for a specified duration, or managing a combination of power on/off events.

If you're interested in learning more about programming web applications using the Ruby on Rails framework, check out *Programming Ruby: The Pragmatic Programmer's Guide* [TFH09].

Now that the web application server is working, it's time to build a mobile client.

7.6 Writing the Code for the Android Client

You might be wondering why you should go through the trouble of building a native Android client when the web application we wrote can be accessed by the Android mobile web browser. Well, if all you wanted to do was toggle light switches on and off, then I would say you don't need a native client. The web interface works just fine and can be further enhanced using AJAX and slick HTML5/CSS3 user interface effects. But if you want to give a little more intelligence to the app, such as activating power switches based on your proximity to them or running an Android service that monitors inbound X10 events like motion detection and then sounds an alert on your phone to bring such events to your attention, a dynamic web page just won't do.

If you haven't already done so, download, install, and configure the Eclipse IDE, the latest Android SDK, and the ADK plug-in for Eclipse. Visit the Android SDK website for details on how to do so.[12]

12. http://developer.android.com/sdk

You will also need to create an Android Virtual Device (AVD) so that you can use it to test the client application in an Android emulator before sending the program to your Android device.[13] I suggest creating an AVD that targets Android 1.5 (API Level 3) so that it emulates the largest number of Android phones available.

Launch the Eclipse environment and select File→New→Android Project. Depending on the version of Eclipse you are running, this option might also be found on the File menu via New->Other->Android->Android Project. Call the project LightSwitch and select Build Target as Android 1.5. You can choose a higher Android version depending on what level of Android device you want to deploy the application to, but since the LightSwitch program will be sweet and simple, Android 1.5 should be adequate for this sample application.

In the Properties area, fill in the Application name as Light Switch and the Package name as com.mysampleapp.lightswitch, and check the Create Activity checkbox and enter LightSwitch. You can specify the Min SDK Version if you wish, but since we're developing for one of the more popular lowest-common-denominator versions of Android, we'll leave it blank for now. Before you continue, check to see if your New Android Project dialog box looks like the one shown in Figure 26, *Creating a new Android Project dialog box with completed parameters*, on page 10.

Android developers with good testing practices would then click the Next button in the New Android Project dialog box to set up a Test Project resource. However, in the interest of space and time, we'll go ahead and click the Finish button.

Once the Android Development Tools Eclipse plug-in generates the skeleton Light Switch application code, double-click the main.xml in the res/layout folder to open it into Android's simple form editor. Drag a ToggleButton control from the Form Widgets palette onto the main.xml graphical layout. Don't worry about perfectly aligning the control in the right spot for now. For this exercise, we're more interested in function over form.

Because this application won't require anything beyond the basic features found in the earlier Android operating system releases, change the Android version in the upper right corner drop-down box of the form editor to Android 1.5. Also, feel free to delete the default Hello world TextView element from the layout. When done, the layout should look similar to the screen shown in

13. http://developer.android.com/guide/developing/devices/managing-avds.html

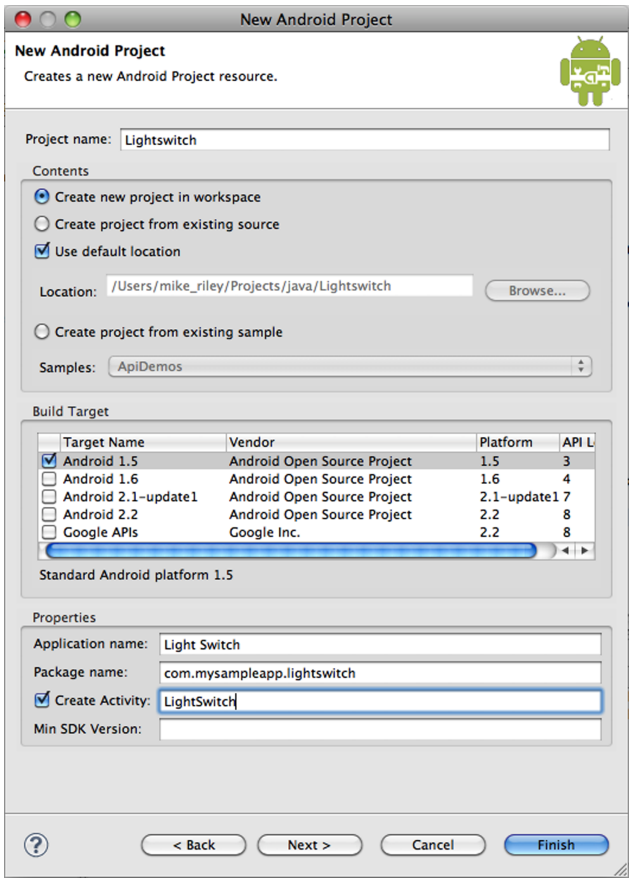


Figure 26—Creating a new Android Project dialog box with completed parameters

Figure 27, *The graphical form layout of the Light Switch application*, on page 11. Save the main.xml file.

Expand the src→com.mysampleapp.lightswitch tree and double-click the LightSwitch.java file. Because we will be using the ToggleSwitch widget, the first thing we need to import is the android.widget.ToggleButton class.

Next, add the java.net.URL and java.io.InputStream libraries, since we'll be creating URL objects to pass to Java InputStream object. The import statement section of the LightSwitch.java file should now look like this:

```
package com.mysampleapp.lightswitch;
import android.app.Activity;
import android.os.Bundle;
import android.widget.ToggleButton;
import android.view.View;
```

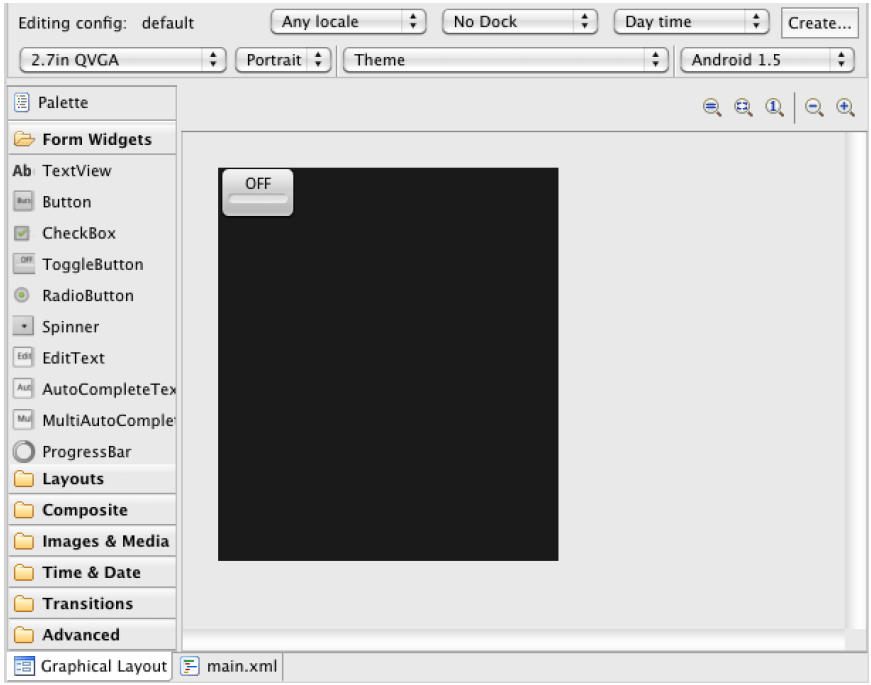


Figure 27—The graphical form layout of the Light Switch application

```
import java.net.URL;
import java.io.InputStream;
```

Now we have to make the LightSwitch aware of the ToggleSwitch by finding it by ID in the LightSwitch class's OnCreate event and adding an event listener to monitor when the switch is toggled on and off:

```
public class LightSwitch extends Activity {
  /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      final String my_server_ip_address_and_port_number =
        "192.168.1.100:3344";
      final ToggleButton toggleButton =
            (ToggleButton) findViewById(R.id.toggleButton1);
      toggleButton.setOnClickListener(new View.OnClickListener()
      {
        public void onClick(View v) {
          if (toggleButton.isChecked()) {
            try {
                final InputStream is = new URL("http://"+
```

```
my_server_ip_address_and_port_number +"/command/on").openStream();
                }
                catch (Exception e) {
                }
          } else {
            try {
                final InputStream is = new URL("http://"+
my_server_ip_address_and_port_number +"/command/off").openStream();
                }
                catch (Exception e) {
                }
            }
          }
      });
    }
}
```

Be sure to set the my_server_ip_address_and_port_number string in the example above to the IP address and port that you plan to use to run the Rails application server we wrote in Section 7.4, *Writing the Code for the Web Client*, on page 5. And that's it! Go ahead and run the application in the Android emulator to make sure it compiles and shows up on the screen correctly.