

Spreadsheets on GS - it's not steroids; just JavaScript


Spreadsheets are a tool that everybody is familiar with - because everybody needs spreadsheets for something. Google, among others, brought us spreadsheets in the browser (Google Docs) which took things up a notch in terms of utility, ubiquity and sharing. Now with a new service called Google Apps Script (or GS for short), we can do even more interesting and useful things with our spreadsheets that are worth examining. In this article we'll take a look at how you can take a "master" list, such as a list of all company employees, and break it down automatically into consumable sub-lists such as a list of employees by office location. Then we'll take that data and publish it to a series of webpages in Google Sites. We'll touch on permissioning since oftentimes access to the data may be based on role or authority. This same recipe could be useful in a thousand scenarios such as providing baseball team rosters from a master list of all the players in the league.

It's Déjà-vu all over again.

Writing applications using a single language (JavaScript) that can cross the boundary between client and server is a very enticing concept. Netscape first introduced the idea with their LiveWire server back in 1996. (It can take awhile for prescient ideas to gain popularity; and for those ideas to mature.)

People are taking advantage of the benefits:

- One language to learn, so less overhead to development
- Construct a DOM server-side and then modify it client-side using the same data structures, methods etc.
- Developing in the browser brings development closer to the



finished product.

Fortunately in the last several years, the amount of library code (e.g. <http://phpjs.org/>) and other aspects of the ecosystem (e.g. <http://jsfiddle.net/>) have grown tremendously to foster it's continued popularity and capability.

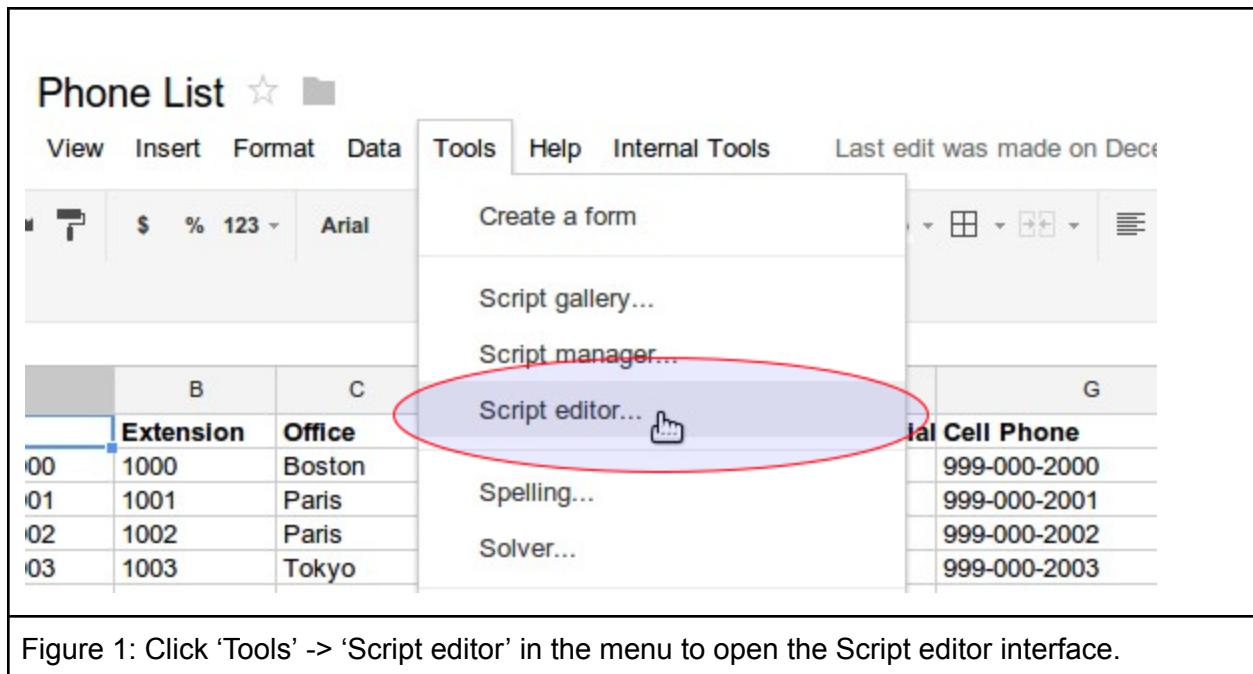
Google Apps Script is basically JavaScript run on the server. Google provides a complete development environment in the browser, with all the code versioning and hosting done for you. They also provide a library of functionality, with APIs for other Google services from email to documents and sites. For this tutorial, we'll assume that you have a Google Account and some familiarity with JavaScript. If you're a total beginner, see https://developers.google.com/apps-script/your_first_script.

You can start off by making a script at <https://script.google.com>, but let's assume that you have an existing spreadsheet and want to add some functionality to it. We're going to start with a [sample spreadsheet](#) full of employees¹. Our imaginary company is a sales organization with field reps associated with offices in major locations throughout the globe. The spreadsheet has columns for

- Direct phone number
- Extension
- Office
- First Name
- Last Name
- Middle Initial
- Cell Phone

Open the spreadsheet and click Tools -> Script editor in the menu to open the Script editor. (See figure 1.)

¹ This anonymous sample data is reset every 24 hours.



Copy the Master

The first thing we want to do is make it so that the master list of employees is copied to another list designed to be a READ-ONLY reference so that casual phone lookups don't inadvertently change a record. Refer to Listing 1 where we create a JavaScript function named `copyPhoneList()`

Note: find the code in this article at <http://pastebin.com/U0C9pHUD>, nicely formatted with comments.

This is our main function - responsible for copying data from our master data set to another copy that is read-only (due to the sheet protection added to it).

Listing 1: Code.gs - copyPhoneList

```
function copyPhoneList() {
  var hasFirstName = false;
  var ss = SpreadsheetApp.getActiveSpreadsheet();

  var data = ss.getRangeByName("masterPhoneListData").getValues(); // the named range
  var range = ss.getRangeByName("masterPhoneListData");
  var numColumns = range.getEndColumn() - range.getColumn() + 1;
  //var numColumns = data[10].length; // how many columns are in each row
  var phoneList = new Array();
```

```

// then we'll iterate through our source data and push recordDatas over to the target for each
non-empty row
for (i=0; i<data.length; i++) { // we've already used a named data range to skip it's header
    hasFirstName = !isEmpty(data[i][4]);
    if (hasFirstName) { // if there is something in firstname, then get the record
        var record = new Array();
        for (j=0; j<numColumns; j++) {
            // we want to explicitly grab only columns a,b,c,d,e,g or 0,1,2,3,4,6
            switch(j) {
                case 0: // DID (Direct Inward Dial)
                case 1: // Ext
                case 2: // Office
                case 3: // First Name
                case 4: // Last Name
                case 6: // Cell Phone
                    record.push(data[i][j]);
                    break;

                default:
                    break;
            }
        }

        // now add each record to the data we're compiling
        phoneList.push(record);
    }
}

// now push it all over to the target sheet
var target = ss.getSheetByName("Read Only");
target.clearContents(); // empty out our target sheet, preserving the formatting

// first we'll create a heading row
var heading = ['Direct', 'Ext', 'Office', 'First Name', 'Last Name', 'Cell Phone'];
//target.appendRow(heading); // put the first heading row into our target sheet

var headersRange = target.getRange(1, 1, 1, heading.length);
headersRange.setValues([heading]);

```

```

// we'd like to have sorted data, by First Name, and if those are equal, then by Last Name
// JavaScript arrays can be sorted by using a callback function supplied as the argument to the
sort() method call
// http://www.sitepoint.com/sophisticated-sorting-in-javascript/
phoneList.sort(function(a, b) {
  if(a[3].toLowerCase() === b[3].toLowerCase()) {
    var x = a[4].toLowerCase(), y = b[4].toLowerCase();
    return x < y ? -1 : x > y ? 1 : 0;
  } else {
    var x = a[3].toLowerCase(), y = b[3].toLowerCase();
    return x < y ? -1 : x > y ? 1 : 0;
  }
});

// All done, put the data into place
target.getRange(2, 1, phoneList.length, phoneList[0].length).setValues(phoneList);
}

```

I won't go through the code line-by-line since the comments indicate what's going on. We have access to all the normal JavaScript language functions, and use multiple functions which are built-in to the Google Spreadsheets API (See sidebar for a list of the base classes). For beginners our code may seem daunting, but it's really just an extension of examples that are available in the Google Apps Script website: <https://developers.google.com/apps-script/> Essentially, to copy the phone list, we select a range of data from a reference to our source spreadsheet. Then we loop through each row of data looking for rows where there is at least a FirstName. Each of those records becomes something we copy to a target list. We're going to leave out the column for Middle Initial because that data is unimportant to us. We can skip over those by simply enumerating the columns we do want. With the data in memory, ready to put in place in our target, we select another sheet by name and clear it's contents. We create an array to use as a header row in our new spreadsheet. Finally we use `getRange()` and `setValues` together to put the data in place.

Default Services in Google Apps Script

GS offers pre-written code libraries (<https://developers.google.com/apps-script/defaultservices>) covering wide areas of functionality including APIs to their other online services including: Base Services, Cache, Calendar, Charts, Contacts, Content, DocsList, Document, Domain, Finance, Gmail, Groups, Html, Jdbc, Language, Lock, Mail, Maps, Properties, Script, ScriptDb, Sites, Soap, Spreadsheet, Ui, UrlFetch, Utilities, and Xml

Once our function is ready, we can run it by selecting it in the top menu bar, and clicking the > play/run icon (Figure 2). If there are no syntax or other errors, we'll see the script execution complete.

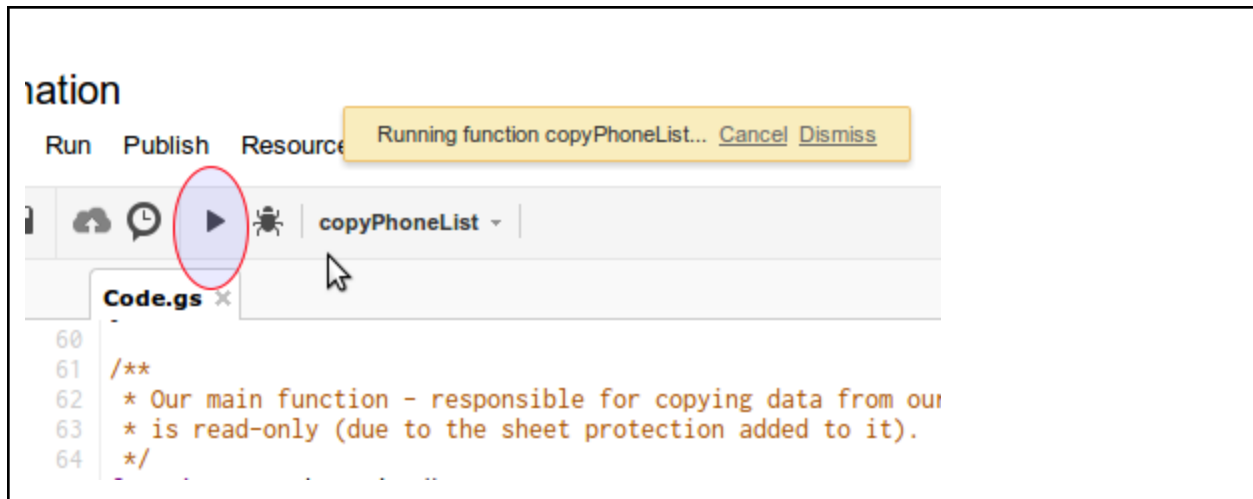


Figure 2: Select the function you wish to run (copyPhoneList) in the menu bar, and click the play icon to execute that piece of functionality.

Script Authorization

The first time you run a GS script, you will likely receive a notification that the script needs your authorization to be executed, as illustrated at

https://developers.google.com/apps-script/your_first_script

Permission granted

We want our copy to be usable as a reference, which means that it needs to be accessible to a lot of people, but we don't want them to make changes. In *Listing 2* we are using the built-in permissions infrastructure; we can restrict edit access to a few users. Note that we can use the `msgBox()` method of the Browser Services to pop a JavaScript `alert()`. Since this is server-side code we can't use a regular JavaScript `alert()` to see something on the client.

Listing 2: Code.gs - Setting permissions

```
/*
 * A function to add protection to a sheet so that it's not
 * inadvertently modified
 */
function protectSheet(sheetName) {
  if (typeof(sheetName) == "undefined") {
    var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheets()[0];
  } else {
    var sheet = SpreadsheetApp.getActiveSpreadsheet().getSheetByName(sheetName);
  }

  var permissions = sheet.getSheetProtection();
  var isProtected = permissions.isProtected();
  if (isProtected) {
    // Browser.msgBox('The sheet ' + sheetName + ' has been protected');
  } else {
    // Browser.msgBox('The permission check shows ' + permissions );
    permissions.addUser('editors@example.com');
    permissions.addUser('joe.boss@example.com');
    permissions.addUser('mary.manager@example.com');
    permissions.setProtected(true);
    sheet.setSheetProtection(permissions);
  }
}
```

Break it down

The next thing we want to do is break out our master list into several smaller spreadsheets according to what office the employee is affiliated with.

Note that when getting a reference to a spreadsheet, you can either get a reference to the active document or take an “external” approach by identifying the sheet with its unique id:

```
var ss = SpreadsheetApp.getActiveSpreadsheet();
var DATA_SPREADSHEET_ID = "0AqwQ0jPilu4IdGNqMzNXSy15ZWQ3c0ZFMkZja2RuS2c";
var ss = SpreadsheetApp.openById(DATA_SPREADSHEET_ID);
```

then to get the actual sheet in question, you can work with the ordinal index of the sheet you wish to retrieve (assuming that order does not change) or get the sheet by name:

```
var sheet = ss.getSheets()[0];  
var sheet = ss.getSheetByName("Read Only");
```

Listing 3: Code.gs - createOfficeSheets()

```
// This is where the data used will be retrieved from:  
var DATA_SPREADSHEET_ID = "0AqwQ0jPilu4ldGNqMzNXSy15ZWQ3c0ZFMkZja2RuS2c";  
  
function createOfficeSheets() {  
  var ss = SpreadsheetApp.getActiveSpreadsheet();  
  var sheet = ss.getSheets()[0];  
  
  var dataSs = SpreadsheetApp.openById(DATA_SPREADSHEET_ID);  
  // var dataSheet = dataSs.getSheets()[0];  
  var dataSheet = ss.getSheetByName("Read Only");  
  
  // Fetch all the data  
  var data = getRowsData(dataSheet);  
  
  // This is the data we want to display  
  var columnNames = ['Direct', 'Ext', 'Office', 'First Name', 'Last Name', 'Cell Phone'];  
  
  // Index data by office name  
  var dataByOffice = {};  
  var offices = [];  
  for (var i = 0; i < data.length; ++i) {  
    var rowData = data[i];  
    if (!dataByOffice[rowData.office]) {  
      dataByOffice[rowData.office] = [];  
      offices.push(rowData.office);  
    }  
    dataByOffice[rowData.office].push(rowData);  
  }  
  
  offices.sort();  
  var headerBackgroundColor = dataSheet.getRange(1, 1).getBackgroundColor();  
  for (var i = 0; i < offices.length; ++i) {  
    var sheet = ss.getSheetByName(offices[i]) ||
```



```

    ss.insertSheet(offices[i], ss.getSheets().length);
    sheet.clear();
    var headersRange = sheet.getRange(1, 1, 1, columnNames.length);
    headersRange.setValues([columnNames]);
    headersRange.setBackgroundColor(headerBackgroundColor);
    setRowsData(sheet, dataByOffice[offices[i]]);
  }
}

```

Trigger Happy

We've worked out our main functionality, and clicked the Run icon to test our code, but how do we tie this code to our spreadsheet so that it just works? GS provides some built-in events that you can attach code to. If defined, any function named `onEdit()` will run automatically when a user edits a spreadsheet. Similarly, `onOpen` will get triggered once per user session. With that in mind, we'll create our office sheets once per session, but we'll make the read-only copy on every edit. We'll also introduce a couple more features. The `onEdit` handler is obviously tied to an edit event, and so that carries with it some context. It makes it easy to add an automatic comment that records when a particular cell was last modified (answering such questions as "did I remember to update Joe's phone number?"). We also add in a `zebraStripe()` function that makes it easier to visually scan large tables.

Another option we have is to expose functionality to the user in the form of custom spreadsheet menus. Actually, there is a whole UI Services section that we could use to build a custom interface, but that is for another day. For this demo, we'll just use `addMenu()` to allow the end user to update office sheets on demand (in addition to once per session).

Listing 4: Code.gs - Triggers

```

/*
 * Our main handler
 * The onEdit trigger is one of three built-in triggers
 * This function, if defined, is called automatically
 * @see https://developers.google.com/apps-script/understanding_triggers
 */
function onEdit(event) {
  var ss = event.source.getActiveSheet();
  var r = event.source.getActiveRange();

```

```

r.setComment("Last modified: " + (new Date()));
copyPhoneList();
zebraStripe("Read Only");

// createOfficeSheets(); // don't do this every edit
}

// The onOpen function is executed automatically every time a Spreadsheet is opened
function onOpen(e) {

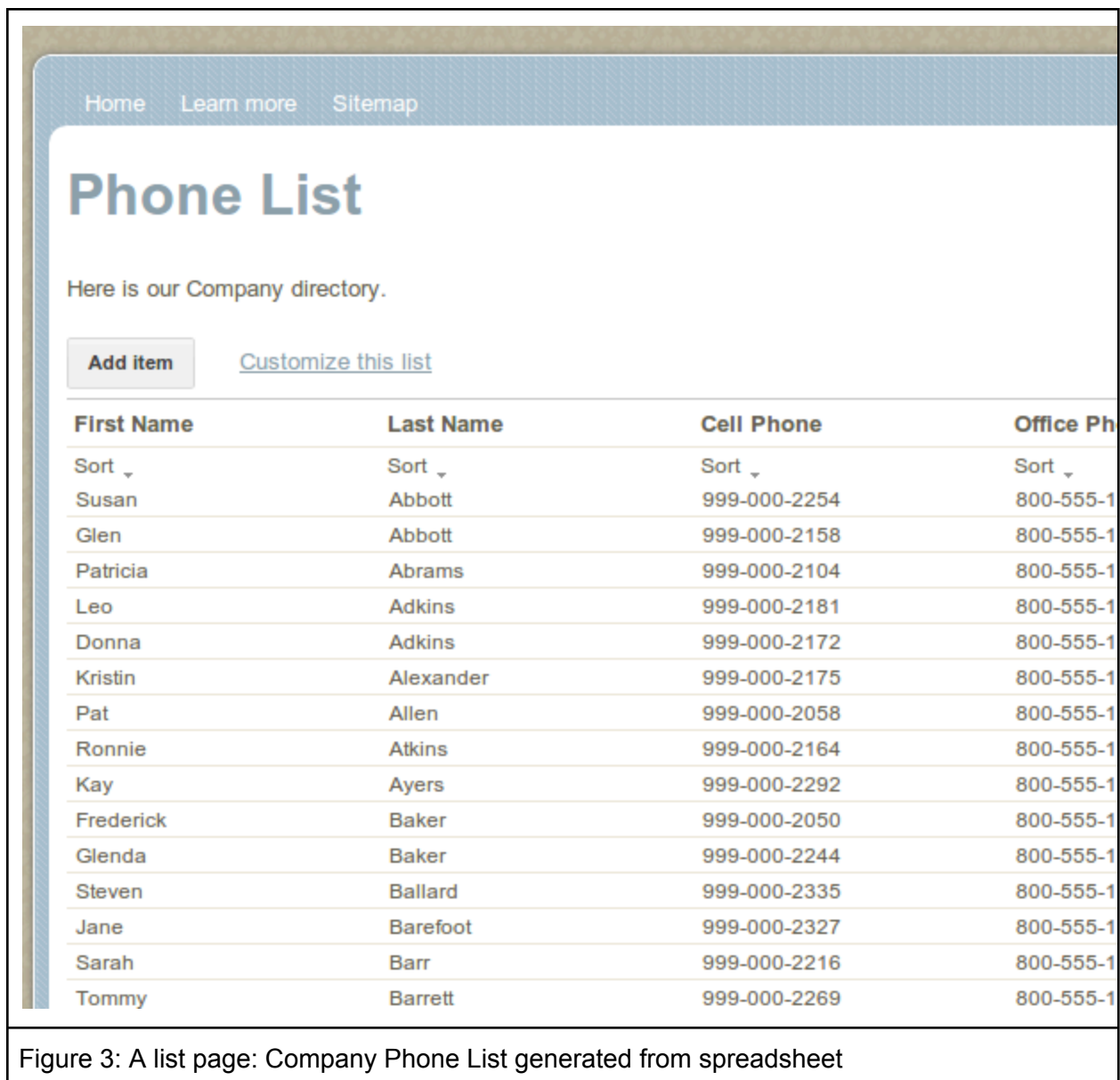
var ss = SpreadsheetApp.getActiveSpreadsheet();
var menuEntries = [];
menuEntries.push({name: "Generate Office Sheets", functionName: "createOfficeSheets"});
ss.addMenu("Internal Tools", menuEntries);
protectSheet('Read Only');
createOfficeSheets();
}

```

Our spreadsheets will be accessible according to however we've shared them. But we can take it a step further by publishing these spreadsheets in web pages that we build in Google Sites. First create a site at <https://sites.google.com> Then create a new top-level page, and make it type:list. In the definition of that page, we'll choose the basic fields that we want to publish from our spreadsheet.

As an example, I've created a site named `examplegsr` and I created a page named 'phone-list' which can be viewed at <https://sites.google.com/site/examplegsr/phone-list>. Note; 'name' is not the title, it's the URL of the page.

With the code in *Listing 5*, I can publish this spreadsheet data as list elements in a page on my site. See figure 3.



Listing 5: Code.gs - pushToSite

```
function pushToSite() {
  var phoneList = new Array();
  // var ss = SpreadsheetApp.getActiveSheet().getSheetByName('Read Only');
  var ss = SpreadsheetApp.getActiveSpreadsheet().getSheetByName('Read Only');

  var site = SitesApp.getSiteByUrl("https://sites.google.com/site/examplegsr/");
  // Or, if you are a Google Apps user:
  // var site = SitesApp.getSite("YOUR_DOMAIN", "SITE_NAME")
}
```

```

var page = site.getChildByName("phone-list");

var data =
SpreadsheetApp.getActiveSpreadsheet().getRangeByName("masterPhoneListData").getValues
();
var range =
SpreadsheetApp.getActiveSpreadsheet().getRangeByName("masterPhoneListData");
var numColumns = range.getEndColumn() - range.getColumn() + 1;

// then we'll iterate through our source data and push recordDatas over to the target for each
non-empty row
for (i=0; i<data.length; i++) { // we've already used a named data range to skip it's header
  hasFirstName = !isCellEmpty(data[i][3]);
  if (hasFirstName) { // if there is something in firstname, then get the record
    var record = new Array();
    record.push(data[i][3]); // First Name
    record.push(data[i][4]); // Last Name
    record.push(data[i][6]); // Cell Phone
    record.push(data[i][0]); // Direct
    record.push(data[i][2]); // Office
    // now add each record to the data we're compiling
    // phoneList.push(record);
    page.addListItem(record);

  }
}
}

```

Further info

See the home for GS <https://developers.google.com/apps-script/> and the various articles they have curated at

<https://developers.google.com/apps-script/articles>.

Code: <http://pastebin.com/U0C9pHUD>

Sample Spreadsheet Data:

<https://docs.google.com/spreadsheet/ccc?key=0AqwQ0jPilu4ldGNqMzNXSy15ZWQ3c0ZFMkZja2RuS2c#gid=0>

Sample Site: <https://sites.google.com/site/examplegsr/phone-list>

- Gregory Scott Rundlett

A longtime Free Software advocate participating in the Boston Linux User Group and Greater New Hampshire Linux User Group. He lives with his two sons in Salisbury, MA.