Learning Google Apps Script

Learning Google Apps Script begins by covering the basics of the Google Application platform and goes on to empower you to automate most Google Applications.

You will learn the concepts of creating a menu, sending e-mails, building interactive webpages, and implementing all these techniques to develop an order processing workflow application.

You will be guided through all these tasks with plenty of screenshots and code snippets that will ensure your success in customizing and automating various Google Applications.

This guide is an invaluable tutorial for beginners who intend to develop the skills to automate and customize Google Applications.

Who this book is written for

If you are a beginner in the HTML, CSS, and JavaScript language, intend to build apps from scratch, and learn the techniques of customizing and automating Google Applications, then this book is ideal for you.

Learning Google Apps Script is perfect as your next step towards building interactive webpages and will provide you with further insights into the workings of various Google Applications.

What you will learn from this book

- Learn about the Google Apps Script platform and work with scripts to develop Google Apps
- Create custom menus and dialogs
- Parse and send e-mails
- Generate Google Calendar events
- Build Translator and RSS reader applications
- Develop interactive webpages
- Design interactive web forms
- Form a workflow application

Prices do not include local sales tax or VAT

Ramalingam Ganapathy

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Parsing and Sending E-mails'
- A synopsis of the book’s content
- More information on Learning Google Apps Script
Ramalingam Ganapathy is an independent computer software professional with more than 15 years of working experience of JavaScript and Google Apps Script. In 1985, he started his career as a digital electronic circuit designer and service engineer. Highly interested in reading technical books and building electronic projects, he is a detail-oriented and logical person. Since 2001, he has been freelancing with Elance and Upwork (formerly oDesk). He earned a good reputation on the Upwork portal, and most of his clients are satisfied.
Preface

Google Apps is a collection of applications, namely, Gmail, Calendar, Drive, Docs, Sheets, and Forms. You can customize or automate Google Apps using the scripting language JavaScript with Google's defined classes. Google implements Google Apps Script (GAS) based on JavaScript.

Almost all Google Apps provide one or more services. GAS services and APIs provide easy access to automate tasks across Google products and third-party services. You can use these service classes in your GAS code to customize or automate Google Apps.

This book introduces basic things first before moving to advanced concepts step by step with practical code and examples. By reading this book, you'll gather expertise in Google Apps Script. Happy reading!

What this book covers

Chapter 1, Introducing Google Apps Scripts, tells you about Google Apps and gives you an introduction to Apps Scripts, explains how to create a project, and introduces custom formulas.

Chapter 2, Creating Basic Elements, covers many types of dialog and how to create and display them, how to use the Logger class to log values, and how to debug your script.

Chapter 3, Parsing and Sending E-mails, talks about the ContactApp, MailApp, and GmailApp services. Using these services, you'll create many useful real-world applications, including an e-mail merger application.

Chapter 4, Creating Interactive Forms, deals with creating Forms dynamically by script, publishing the script as a web application, creating Forms using HtmlService, creating an e-voting application, and creating a ticket reservation application.
Chapter 5, Creating Google Calendar and Drive Applications, teaches the reader to create Calendar events and sync events from one Calendar to another Calendar. This chapter also teaches how to enable GAS advanced services.

Chapter 6, Creating Feed Reader and Translator Applications, is about learning and creating many useful applications, including RSS/Atom reader and language translator applications.

Chapter 7, Creating Interactive Webpages, tells how to create an RSS feed/publisher, a file uploading application, and a full-blown timesheet application using HtmlService.

Chapter 8, Building a Workflow Application, explains how to create a workflow application and proceeds create a useful real-world order processing application.

Chapter 9, More Tips and Tricks and Creating an Add-on, is all about using external libraries including OAuth2, and Apps Script add-ons.
In the previous chapter, you learned how to create basic GAS elements such as custom menu, dialog, and toast. You also learned how to debug your script codes. In this chapter, you will learn many real-world Gmail and Contacts applications including a mail merger application.

In this chapter, if you go through left and right square brackets inside code like `[[ value ]]`, then replace value with the actual value including the brackets.

For example, if the e-mail ID is `example@emample.com` and you go through `My email id [[emailid]] \n`, then replace it with `My email id example@example.com \n`.

**Creating Gmail Contacts by script**

You can create Gmail Contacts by script using the `createContact` method of the `ContactsApp` class. For example, if the name is Anika Sumi and the e-mail ID is `anika@example.com`, then the `ContactsApp.createContact("Anika", "Sumi", "anika@example.com")` code will create the expected contact.
To know more available methods of the ContactsApp class, in the code editor, type ContactsApp and . (a dot) next to it. Then, you can view all the available methods with parameter details in code hint as shown in the following screenshot:

You can see deprecated methods struck out in the preceding screenshot; you are advised not to use those methods.

**Accessing Sheet, cell, range, and offset**

A Google Sheet's spreadsheet has one or more Sheets or tabs in it. Sheets are indexed from left to right starting from 0. For example, the left-most Sheet is referred to by the index 0, the next one by 1, and so on. In GAS, we can refer to a Sheet by its index or by its name.

For example:

- The `getSheets()` method returns an array of Sheet objects. From the array, we can refer to an individual Sheet by its index.
- The `getSheetByName("Contacts")` function returns a Sheet object with the name Contacts.

In Google Sheets, column label starts from the letter A, and is counted in a programmatic point of view, from left to right starting with the number 1. For example, column A is 1, B is 2, and so on. Rows are identified by their respective label numbers. In GAS, we can reference a cell or a range of cells by A1 notation or by separate row and column numbers.
For example:

- The `getRange('D1:F10')` method returns a `Range` object referencing the cells from `D1` to `F10`
- The `getRange(1, 4, 10, 3)` method returns a `Range` object referencing the same range `D1:F10`

Offset is an indirect referencing method to refer to a cell/range from a base cell reference. An offset reference is determined by how many rows and columns it shifted from the base cell.

For example, if the base cell is `D1`, then the `offset(10, 3)` method returns the range `D1:F10`.

**Reading and writing the Sheet data**

Often you need to read and/or write data to/from the Sheet. Usually, use the `getValue` method to read a value from a cell and the `getValues` method to read values from a range. The `getValue` method returns a single value and the `getValues` method returns a 2-dimensional array of values. To write single value and 2-dimensional array of values, use `setValue` and `setValues` methods respectively.

**Building a Gmail Contact search application**

Now, we will create an application to search existing contacts. This application is able to search and list your Gmail Contacts in Sheets. Create a new Sheet and rename `Sheet1` to `Contacts` and set it up as shown in the following screenshot. Create a button and assign the function name `searchContacts` to it, as you learned in the previous chapter.
Create the `searchContacts` function as listed here:

```javascript
function searchContacts() {

  var SheetContacts = SpreadsheetApp.getActiveSpreadsheet().
    getSheetByName("Contacts");

  // Read input from cell A3
  var searchCriteria = SheetContacts.getRange("A3").getValue();

  // First 10 contacts.
  // [You can change this limit, but advised to keep small.]
  var numOfContacts = 10;

  // Clear existing sheet data
  SheetContacts.getRange(7,1,numOfContacts,4).clear();

  // Returns an array of contacts where
  // contacts name matches with search text.
  var contacts = ContactsApp.getContactsByName(searchCriteria);

  // Limit number of contacts.
  if(contacts.length > numOfContacts) contacts.length = numOfContacts;

  var cell = SheetContacts.getRange("A7");

  for(var i in contacts){
    var name = contacts[i].getFullName();
    var email = contacts[i].getEmails()[0];

    if(email) email = email.getAddress();
    else email = "";

    // For simplicity get the first phone number
    var phone = contacts[i].getPhones()[0];

    if (phone) phone = phone.getPhoneNumber();
    else phone = "";
  }
}
```

Here, `clear` is the `Range` object's method to clear everything including format and formula, in a range of cells. You can use the `clearContent` method of the `Sheet` object to clear the entire Sheet. Alternatively, you can use the `clearContent` method to clear content only.
// For simplicity get the first address
var address = contacts[i].getAddresses()[0];

if (address) address = address.getAddress();
else address = "";

// cell.offset(rowOffset, columnOffset)
cell.offset(i, 0).setValue(name);
cell.offset(i, 1).setValue(email);
cell.offset(i, 2).setValue(phone);
cell.offset(i, 3).setValue(address);
}
};

Do not copy paste the preceding code, but edit it yourself. By doing so, you'll be aware of available method signatures (method names and parameters) of classes such as SpreadsheetApp, ContactApp, and Contact with the help of the script editor's code hint feature.

After you have edited and saved code without error, turn to the spreadsheet window. If you enter a search term in the A3 cell (search box) and click on Search, then the first 10 contacts will be listed as shown in the following screenshot (the listed contacts details vary as per your Gmail username and contacts):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Search Gmail Contacts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Name</td>
<td>Email</td>
<td>Phone</td>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ramalingam Ganapathy</td>
<td>**********@gmail.com</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Reshna Raman</td>
<td>**********@pa0ckp0ub.com</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Adam Ramshaw</td>
<td>**********@gmail.com</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What if you want to update the listed contacts by the `searchContacts` function? For example, you may want to update the phone number and/or address of a contact. To update contact fields, we will create another function called `updateContacts`. Before creating that, in the Contacts Sheet, add a button next to Search named Update and assign function name `updateContacts` as shown in the following screenshot:

```
function updateContacts(){
  var SheetContacts = SpreadsheetApp.getActiveSpreadsheet()
    .getSheetByName("Contacts");

  var cell = SheetContacts.getRange("A7");
  var numOfContacts = 10;

  for(var i = 0; i < numOfContacts; i++){
    var email = cell.offset(0, 1).getValue();
    // Skip if email field is null
    if(!email) continue;

    var contact = ContactsApp.getContact(email);
    // Skip if contact is null or undefined
    if(!contact) continue;

    var name = cell.offset(i, 0).getValue();
  }
```

Update those field values you would like to update. Now create the function listed here:
// Skip if name field is null
if(!name) continue;
contact.setFullName(name);

var phone = cell.offset(i, 2).getValue().toString();

// Returns phone numbers as an array
var contPhone = contact.getPhones(ContactsApp.Field.MAIN_PHONE)[0];

// Update main phone number if exist otherwise add.
if(phone){
  if(contPhone){
    contPhone.setPhoneNumber(phone);
  } else {
    contact.addPhone(ContactsApp.Field.MAIN_PHONE, phone);
  }
}

var address = cell.offset(i, 3).getValue().toString();

// Returns address as an array
var contAddress = contact.getAddresses(ContactsApp.Field.HOME_ADDRESS)[0];

// Update home address if exist otherwise add.
if(address){
  if(contAddress) {
    contAddress.setAddress(address);
  } else {
    contact.addAddress(ContactsApp.Field.HOME_ADDRESS, address);
  }
}
The preceding function retrieves contacts by the given e-mail ID; and, for each contact, it also retrieves field values and updates/adds those field values with the Sheet values. This function can update/add full name, phone, and address fields but not the e-mail ID.

**Building the Gmail parser application**

The `parseEmail` function is able to check 10 latest inbox threads, extract the `from` field and body text from unread messages, and put the gathered data in the left-most tab of the Sheet. Create the `parseEmail` function as listed here:

```javascript
/**
 * Gets content of latest unread message in Gmail inbox
 * and puts gathered data in left most tab of Sheets.
 *
 * function parseEmail(){

    // Left most sheet/tab
    var emailSheet = SpreadsheetApp.getActiveSpreadsheet()
    .getSheets()[0];

    // Clear the entire sheet.
    emailSheet.clear();

    // Checks maximum 10 threads
    var thread = GmailApp.getInboxThreads(0,10);

    var row = 1;

    for(var thrd in thread){
        var messages = thread[thrd].getMessages();

        for (var msg in messages) {
            var message = messages[msg];

            if(message && message.isUnread())
                emailSheet.getRange(row,1).setValue(message.getFrom());

                emailSheet.getRange(row++,2)
                .setValue(message.getPlainBody());

            }
        }
    }

You can use RegExp to extract only the required data from the message body text.
Properties service

GAS provides the properties service to store and/or to retrieve project-related data. The data organized as key/value pairs, can be set manually or by script codes. The following screenshot shows how you can set properties manually. To see this dialog, click on the File menu and select Project properties:

You can use manually created project properties in script codes, but the properties created by code sometimes may not be visible in the Project properties dialog. You can create, update, or delete project properties in codes.

In the next task, we are going to use project properties.

Downloading Gmail attachments to Drive

The `saveEmailAttachmentsToDrive` function can download Gmail attachments to Drive. In this function `PropertiesService` is used to avoid repeated downloading of the same attachment. The `createFolder_` function is used to create folders, if not already exist, with the given name in Drive.
If any function name is appended with _ then it will not be listed under the Run menu. You cannot run these functions directly, but they can be called from the other functions. These are called private functions.

You can create the createFolder_ function in the same script file along with the saveEmailAttachmentsToDrive function or in a separate script file such as Library.gs:

```javascript
/**
 * Checks latest 100 inbox threads,
 * saves attachments in 'Gmail attachments' folder,
 */
function saveEmailAttachmentsToDrive(){

    // Create 'Gmail Attachments' folder if not exists.
    createFolder_('Gmail attachments');

    // Get inbox threads starting from the latest one to 100.
    var threads = GmailApp.getInboxThreads(0, 100);

    var messages = GmailApp.getMessagesForThreads(threads);

    var folderID = PropertiesService.getUserProperties().getProperty("FOLDER");

    var file, folder = DriveApp.getFolderById(folderID);

    for (var i = 0 ; i < messages.length; i++) {
        for (var j = 0; j < messages[i].length; j++) {
            if(!messages[i][j].isUnread()){
                var msgId = messages[i][j].getId();

                // Assign '' if MSG_ID is undefined.
                var oldMsgId = PropertiesService.getUserProperties().getProperty('MSG_ID') || '';

                if(msgId > oldMsgId){
                    var attachments = messages[i][j].getAttachments();
                }
            }
        }
    }
}
```
for (var k = 0; k < attachments.length; k++) {
    PropertiesService.getUserProperties()
        .setProperty('MSG_ID', messages[i][j].getId());

    try {
        file = folder.createFile(attachments[k]);
        Utilities.sleep(1000);// Wait before next iteration.
    } catch (e) {
        Logger.log(e);
    }
}
else return;
};

The preceding function calls the following `createFolder_` function with the folder name as an argument. The function `createFolder_` looks for the given folder, creates if it does not exist, and returns its unique ID:

```
function createFolder_(name) {
    var folder, folderID, found = false;

    /*
    * Returns collection of all user folders as an iterator.
    * That means it do not return all folder names at once,
    * but you should get them one by one.
    */
    var folders = DriveApp.getFolders();

    while (folders.hasNext()) {
        folder = folders.next();
        if (folder.getName() == name) {
            folderID = folder.getId();
            found = true;
            break;
        }
    }
}
```
Parsing and Sending E-mails

```javascript
if (!found) {
    folder = DriveApp.createFolder(name);
    folderID = folder.getId();
}

PropertiesService.getUserProperties()
    .setProperty("FOLDER", folderID);

return folderID;
}
```

In the preceding function the `getFolders` method is an iterator method. An iterator does not return all the data in one go, but only the current data. To get successive data, you should call `next` method repeatedly until `hasNext` became `false`.

Sending e-mails using the MailApp service

The `sendEmail` function is able to send e-mails with prefixed messages. Remember to replace e-mail ID and message text. This service is mainly used to send e-mails with limited methods (only `sendEmail` and `getRemainingDailyQuota`), and it cannot access your Gmail account. You can use the `GmailApp` class for more methods:

```javascript
function sendEmail()
{
    var to = "[[receiver email id]]";
    var message = "[[message]]\n";

    MailApp.sendEmail(to, "Chapter 3", message);
}
```

Sending an e-mail notification on Form submission

Imagine if you created a Form and presented it to many users. It would be tedious to open the response Sheet every time to verify whether any user has submitted the Form or not. The problem would be worse if you created many Forms and sent them to many users. It will be helpful receiving a notification e-mail whenever there is a Form submission.
For this task, create a Form with three fields as shown in the following screenshot:

Submit the test data from a live form. Your submitted data will be saved in a response Sheet named something like Form Responses 1. The column headers will be as per your Form fields as shown in the following screenshot. Data may vary as per your input.

In the script file, you need to make the following changes:

1. Enter the sendEmail function mentioned from the following code.
2. Replace the receiver's e-mail ID. If you run this function, then it will send an e-mail with the last submitted data (bottom-most row) in the response Sheet.
3. Check the Sheet's actual name and the name used in the code; they should be exactly the same. If you are not sure, right-click on the Sheet name and select Rename...
Parsing and Sending E-mails

4. Copy the Sheet name from the Rename dialog and paste it in the following code:

```javascript
function sendEmail()
{
  var sheet = SpreadsheetApp.getActiveSpreadsheet()
    .getSheetByName("Form Responses 1");

  var lastRow = sheet.getLastRow();
  var lastCol = sheet.getLastColumn();
  var data = sheet.getRange(lastRow,1,1,lastCol)
    .getValues()[0];

  var to = "[[ receiver email id]]";
  var message = "Name: " + data[1] + "\n";

  message += "Phone: " + data[2] + "\n";
  message += "Question: " + data[3] + "\n";

  // MailApp.sendEmail(recipient, subject, body);
  MailApp.sendEmail(to, "Chapter 3", message);
}
```

You created a Form and a function to send response data to an e-mail ID. Creating a trigger so as to run the sendEmail function as soon as a Form is submitted will complete this task.

Creating triggers manually

To create a trigger, in the code editor click on Resources and select Current project's triggers then the Current project's triggers dialog will open. Already created triggers will be listed in this dialog, otherwise a link to create a new trigger will appear. Click on the No triggers set up. Click here to add one now link. Select the options from the dropdowns as shown in the following screenshot:
Under the **Run** heading, select the `sendEmail` function for which you want to create the trigger. Select **From spreadsheet** and **On form submit** under the **Events** heading as shown in the preceding screenshot.

If a Form user submits data to the spreadsheet, the trigger will run the `sendEmail` function.

For more info on triggers, please go to [https://developers.google.com/apps-script/guides/triggers/](https://developers.google.com/apps-script/guides/triggers/).

### Creating and deleting triggers by script

You can create or delete triggers programmatically as shown in the following sample code:

```javascript
/**
  * Deletes all the triggers.
  *
  */
function deleteTriggers(){
  var triggers = ScriptApp.getProjectTriggers();

  triggers.forEach(function(trigger){
    try{
      ScriptApp.deleteTrigger(trigger);
    } catch(e) {
      throw e.message;
    }\
  
    Utilities.sleep(1000);
  });
}

function createTrigger(){
  var ss = SpreadsheetApp.getActiveSpreadsheet();

  // Create new trigger
  ScriptApp.newTrigger("sendEmail")
  .forSpreadsheet(ss).onFormSubmit().create();
}
```
In the `deleteTriggers` function, the `Utilities service's sleep` method is used to pause the script temporarily for the specified milliseconds. Otherwise, you may experience the `Too many service invocation...` error.

### Forwarding e-mails if the specific keyword is found in the message body

The `forwardEmails` function is able to forward e-mail messages, if a specific keyword is found in the body text to a prefixed e-mail ID. Be cautious about the number of iterations of the `for` loop while testing your code so that you can avoid lot of messages forwarded in error:

```javascript
/**
 * 1. Checks all unread inbox threads and messages.
 * 2. If specific keyword found then forwards it to another recipient.
 * 3. Marks that message as Read.
 */
function forwardEmails() {
  var recipient = "[forward email id]";
  /*
   * Use keywords separated by '|'.
   * For example: "purchase | invoice"
   */
  var words = "keywords list";
  var regExp = new RegExp(words, 'g');

  var len = GmailApp.getInboxUnreadCount();

  for (var i = 0; i < len; i++) {
    // get 'i'th thread in inbox
    var thread = GmailApp.getInboxThreads(i, 1)[0];

    // get all messages in 'i'th thread
    var messages = thread.getMessages();
  }
}
```
var msgLen = messages.length;
var isAllMarkedRead = true;

// iterate over each message
// CAUTION: limit loop iterations for initial testing.
for (var j = 0; j < 5 /* msgLen */; j++) {
  var message = messages[j];

  if(message.isUnread()){
    var bodyText = message.getPlainBody();
    var test = regExp.exec(bodyText);
    message.forward(recipient);
    isAllMarkedRead = false;
    message.markRead();
  }
}

if(isAllMarkedRead) len++;
Utilities.sleep(1000);
};

function sendEmailWithAttachments(){
  var file = SpreadsheetApp.getActiveSpreadsheet()
    .getAs(MimeType.PDF);

  // MailApp.sendEmail(recipient, subject, body, options)
  MailApp.sendEmail(
    "[[ Recipient email id ]]",
    "Chapter 3",
    ",
    
    { attachments: [file],
      name: 'Chapter 3 test attachment'
    }
  );
}
Embedding inline images in an e-mail message

To embed images such as a logo in your e-mail message, you may use HTML code instead of plain text. Upload your image to Google Drive, retrieve, and use that file ID in code:

```javascript
function sendEmail(){
  var sheet = SpreadsheetApp.getActiveSpreadsheet()
    .getSheetByName("Form Responses 1");

  var lastRow = sheet.getLastRow();
  var lastCol = sheet.getLastColumn();
  var data = sheet.getRange(lastRow,1,1,lastCol).getValues()[0];

  var image = DriveApp.getFileById("[[image file's id in Drive ]]").getBlob();

  var to = "[[Recipient email id ]]";
  var message = '<img src="cid:logo" /></p';

  message += "<p>Name: " + data[1] + "</p>"
  message += "<p>Phone: " + data[2] + "</p>"
  message += "<p>Question: " + data[3] + "</p>"

  MailApp.sendEmail(
    to,
    "Chapter 3 inline image example",
    "",
    {
      inlineImages:{ logo:image },
      htmlBody:message
    }
  );
}
```

Building an e-mail merger application

Sending personalized e-mails to hundreds of recipients at a time might be a time consuming task. Composing the draft and entering the subject and recipient's e-mail ID for each message might be tedious too. Using this mail merger application, you can send the same kind of information to all recipients, but customized to some extent. For example, greeting an individual.
The first step is creating a draft in your Gmail as shown in the following screenshot. The draft is used as a template. You can use any special character to enclose the text to be replaced. In the draft, the code shown in the following screenshot uses left (<>) and right (>>) angled brackets to replace the first name with the **First Name** column data in an **EmailList** Sheet. You can include any other placeholder or field as per your requirement. Set up the draft, but don't send it now:

Create a Sheet with the name as **EmailList** in a new Sheet or existing Sheet. Create the column headers as shown here:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>Email Ids</td>
<td>Subject</td>
<td>Date/Time</td>
<td></td>
</tr>
<tr>
<td>Ramalingam</td>
<td><a href="mailto:example@example.com">example@example.com</a></td>
<td>Test</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Create functions as shown in the following code, in the script editor. Replace the draft and sender name with actual values. Set \texttt{maxEmails} (this code uses 50) by considering your daily e-mail sending quota:

```javascript
// Returns your draft text.
function getDraftBody(draftName){
    var drafts = GmailApp.getDraftMessages();
    for(var i in drafts)
        if( drafts[i].getSubject() == draftName )
            return drafts[i].getPlainBody();
}

function sendEmails(){
    // EmailList sheet column numbers, 0 based.
    const FIRST_NAME_COL = 0;
    const EMAIL_IDS_COL = 1;
    const SUB_COL = 2;
    const DATE_COL = 3;
    const maxEmails = 50;
    var draftName = "Chapter 3"; // Draft's subject name
    var draftBody = getDraftBody(draftName);
    var quotaLeft = MailApp.getRemainingDailyQuota();
    var ss = SpreadsheetApp.getActive();
    var sheet = ss.getSheetByName("EmailList");
    // Gets all sheet data as a 2-dimensional array.
    var data = sheet.getDataRange().getValues();
    var header = data.shift();
    for(var i=0,count=0; count < maxEmails && count < quotaLeft && i < data.length; ++i){
        var firstName = data[i][FIRST_NAME_COL];
        var recipient = data[i][EMAIL_IDS_COL];
        var subject = data[i][SUB_COL];
        var htmlBody = draftBody.replace("<<FirstName>>", firstName);
        if(recipient){
            GmailApp.sendEmail(recipient,
        }
```
subject,
"
{
    name:"[[ Sender Name ]]",
    htmlBody:htmlBody
}

};

data[i][DATE_COL] = new Date();

++count;
}
};

// Inserts header at top of the array.
data.unshift(header);

// Stores values of array in sheet.
sheet.getRange(1, 1, data.length, header.length)
    .setValues(data);
}

Populate data in the EmailList Sheet. To send e-mails, run the sendEmails function. The <<FirstName>> field in your draft will be replaced as per your First Name column data in the EmailList Sheet. That's it!

Congratulations! You have created a working e-mail merger application.

Summary
In this chapter, you learned about ContactsApp, MailApp, and GmailApp classes and their methods. Using these classes, you created many useful real-world applications including an e-mail merger application. In the next chapter, you will learn how to create Forms programmatically using FormApp and HtmlService classes. Also you will learn about doGet and doPost simple trigger functions.
Where to buy this book
You can buy Learning Google Apps Script from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.