

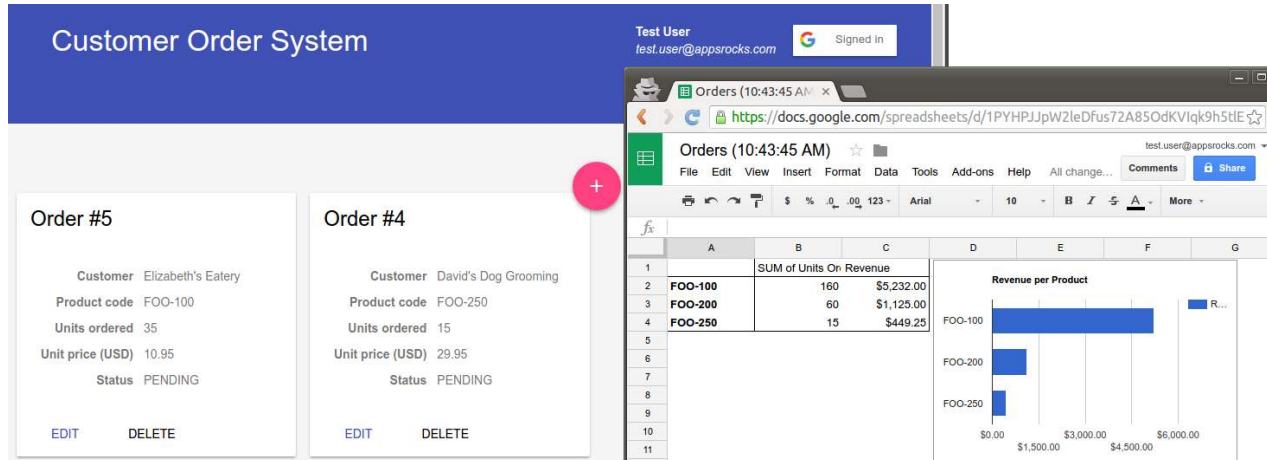
Google Sheets as a Reporting Tool: Sheets API

GSP236



Google Cloud Self-Paced Labs

Overview



In this lab you will learn how to use Google Sheets as a custom reporting tool for your users. You will modify an order tracking application to export database records to a spreadsheet and also build data visualizations with the Google Sheets API. You will build the sample application using Node.js and the Express web framework, but the same basic principles are applicable to any architecture.

What You'll Learn

In this lab, you will learn how to:

- Add Google Sign-in to an application.
- Install and configure the [Google APIs Client Library for Node.js](#).
- Create spreadsheets.
- Export database records to a spreadsheet.
- Create pivot tables and charts.

Prerequisites

This is an **advanced level** lab. Familiarity with Workspace APIs, Javascript, and the Google Cloud console is recommended but not required. If you're looking to skill up in these areas, be sure to check out the following labs:

- [Google Apps Script: Access Google Sheets, Maps & Code in 4 Lines of Code](#)
- [Deploy Node.js Express Application in App Engine](#)

Once you're ready, scroll down to get your lab environment set up.

Setup and Requirements

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.


Note: If you are using a Pixelbook, open an Incognito window to run this lab.


How to start your lab and sign in to the Google Cloud Console


1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.

[Open Google Console](#)

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. [Learn more.](#)


Username
google2727032_student@qwiklabs.n 

Password
k68CZxsxMZ 

GCP Project ID
qwiklabs-gcp-4fbfecac8667e457 

[New to labs? View our introductory video!](#)


- Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Sign in
Use your Google Account


[Forgot email?](#)


Tip: Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another**


Choose an account

 Your.Email@gmail.com

 google1381214_student@qwiklabs.net
Signed out

 **Use another account**

Account.

3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

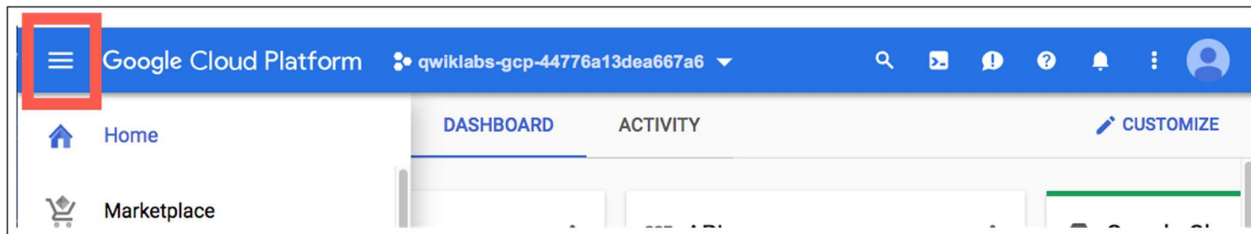
Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

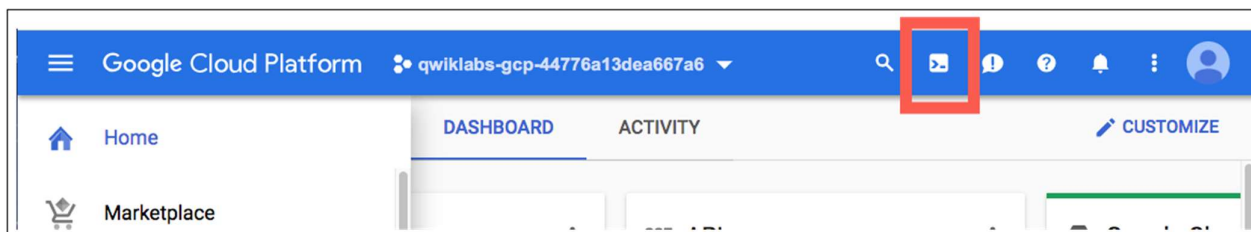
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.



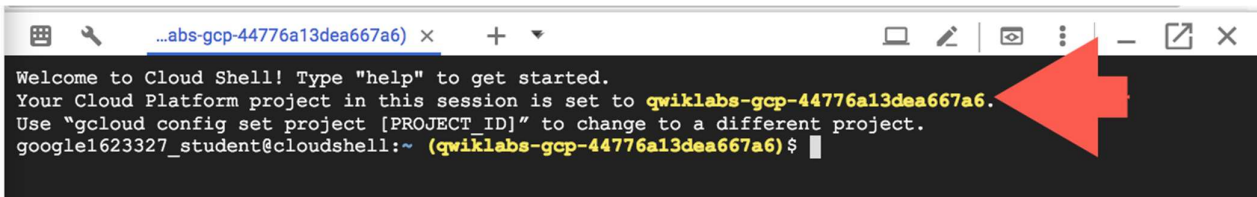
Click **Continue**.

Cloud Shell

Google Cloud Shell provides you with command-line access to your cloud resources directly from your browser. You can easily manage your projects and resources without having to install the Google Cloud SDK or other tools on your system. [Learn more.](#)

Continue

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



```
...abs-gcp-44776a13dea667a6) x + ▾  
Welcome to Cloud Shell! Type "help" to get started.  
Your Cloud Platform project in this session is set to qwiklabs-gcp-44776a13dea667a6.  
Use "gcloud config set project [PROJECT_ID]" to change to a different project.  
google1623327_student@cloudshell:~ (qwiklabs-gcp-44776a13dea667a6) $
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:  
- <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:  
- google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]  
project = <project ID>
```

(Example output)

```
[core]  
project = quiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the [gcloud command-line tool overview](#).

Code editors

In this lab you'll view and edit files. You can use the shell editors that are installed on Cloud Shell, such as `nano` or `vim` or use the Cloud Shell code editor. This lab uses the Cloud Shell code editor.

Get the Sample Code

In Cloud Shell run the following command to clone the GitHub repository:

```
git clone https://github.com/googlecode labs/sheets-api.git
```

You'll be working off the copy located in the `start` directory, but the GitHub repository contains a set of directories representing each step along the process, in case you need to reference a working version.

Enable the Sheets API

Enable the Sheets API in the Cloud Console.

From the **Navigation Menu** and select **APIs & Services > Library**.

In the search bar, search for the "Google Sheets". Click on the **Google Sheets API** tile and click **Enable**.

Run the Sample App

Now that you have your files downloaded and the Sheets API enabled, get the sample order tracking application up and running. Follow the instructions below to install and start the Node.js/Express web application.

1. In Cloud Shell, navigate to the lab's `start` directory:

```
cd sheets-api/start/
```

2. Enter the following commands to install the Node.js dependencies:

```
npm install
```

3. If you get an error message like the following:

```
npm ERR! cb() never called!

npm ERR! This is an error with npm itself. Please report this error at:
npm ERR! <https://npm.community>

npm ERR! A complete log of this run can be found in:
npm ERR! /home/student_02_aac0fb47feb6/.npm/_logs/2020-03-30T17_10_11_099Z-
debug.log
```

Rerun the `npm install` command.

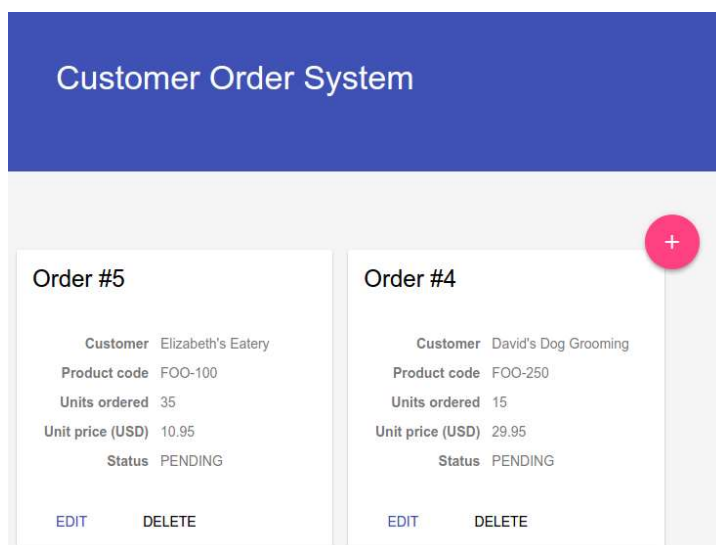
4. Enter the following command to start the server:

```
npm start
```

The server has started when you see the output:

```
Listening on port 8080.
```

5. In the Cloud Shell top menu, click **Web preview > Preview on port 8080**. The **Customer Order System** page opens in a new tab.



The application provides the ability to create, update, and delete a set of simple order records. Included is a SQLite database with some sample data, but feel free to add, update, and delete orders as you progress through the lab.

Take a moment to familiarize yourself with the code, and refer to the table below for a general overview of the application's structure:

app.js	Configures the Express web application framework.
config.json	A configuration file, containing the database connection information.
db.sqlite	A SQLite database used to store the order records.
models/	Contains the code that defines and loads the database models. This application uses the Sequelize ORM library for reading and writing to the database.
node_modules/	Contains the project's dependencies, as installed by npm.
package.json	Defines the Node.js application and its dependencies.
public/	Contains the client side JavaScript and CSS files used by the application.
routes.js	Defines the URL endpoints the application supports and how to handle them.
server.js	The entry point into the application, which configures the environment and starts the server.
views/	Contains the HTML templates to be rendered, written using the Handlebars format. The Material Design Lite (MDL) library has been used for layout and visual appeal.

You will be modifying the base application in the `start` directory, but if you have trouble with a certain step you can switch to that step's directory to see the final result.

Automatic restarts

This application uses [nodemon](#) to automatically reload the application whenever you change a source file. This means that you shouldn't need to stop and restart the server after each step.

Create a Client ID

1. From the Cloud Console, open the **Navigation menu**, select **APIs & Services > Credentials**.
2. Click on the **OAuth Consent Screen** in the left menu. For the **User Type** field, select **Internal** and click **Create**.
3. For the **App name** property, type in "Google Sheets API Quickstart".
4. For **User support email**, select your student username from the dropdown menu.
5. For **Developer contact information**, enter your student username. You can get it from the connection detail panel.
6. Click **Save and Continue**.
7. From the left-hand menu click on **Credentials**. Click on the **Create credentials** dropdown and select **OAuth client ID**.
8. Select the application type as **Web application** and enter the **Name** as Google Sheets API Quickstart.
9. In the text box **Authorized JavaScript origins**, Click **ADD URL** and enter the *entire* URL of the **Customer Order System**.

Example
URL:



10. Then click **Create**:
11. **Copy** the resulting **client ID**, as you will need it in the next step and click **Ok**.

OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services



OAuth access is restricted to users within your organization unless the [OAuth consent screen](#) is published and verified.

Your Client ID

388391214415-jd91mpsvaanq1p6ko1ku3k2uo7es4d0o.apps.gc



Your Client Secret

ThRPFwyWq995ixK2YmQcV0Kh



OK

Add Google Sign-in

Before we can start exporting data to Google Sheets, we need the user to sign in to your application with their Google account and authorize access to their spreadsheets. To do this, we'll be using the [Google Sign-in for Websites](#), a simple JavaScript library you can add to an existing web app.

In Cloud Shell, launch the Cloud Shell code editor by clicking the **Open Editor** icon:



Open Editor

Navigate to `sheets-api/start` and view the `start` directory structure. In the following sections, file paths are relative to the `start` directory.

Open the file `views/layout.handlebars`, which defines the layout for each page, underneath the section that says `<!-- TODO: Add Google Sign-in library -->` add the following:

```
<meta name="google-signin-scope"
      content="https://www.googleapis.com/auth/spreadsheets">
<meta name="google-signin-client_id" content="{YOUR CLIENT ID}">
<script src="https://apis.google.com/js/platform.js" async defer></script>
```

Replace the placeholder `{YOUR CLIENT ID}` with the **OAuth2 client ID** you created in the previous step.

This code sets the OAuth2 client ID to use the scope to request the Google Sign-in library. In this case, we are requesting the scope [spreadsheets](#), since the application needs read and write access to the user's spreadsheets.

Next we need to drop in the code that renders the sign-in button and displays the signed-in user's information. Add the following code to `views/layout.handlebars`, just below `<!-- TODO: Add Google Sign-in button -->`:

```
<div id="profile" style="margin: 0 20px;">
  <b class="name"></b><br/>
  <i class="email"></i>
</div>
<div class="g-signin2" data-onsuccess="onSignIn"></div>
```

Save the file by selecting **File > Save**.

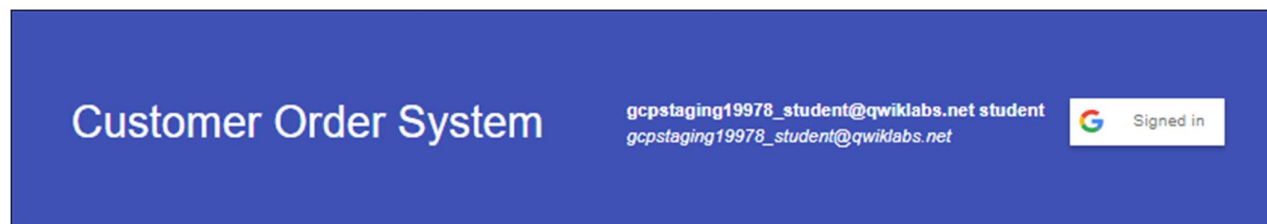
Finally, we need to add a little bit of client-side JavaScript to populate the profile section once sign-in is complete. Add the following to `public/script.js`:

```
function onSignIn(user) {
  var profile = user.getBasicProfile();
  $('#profile .name').text(profile.getName());
  $('#profile .email').text(profile.getEmail());
}
```

Save the file.

Reload the application in your browser, click **Sign in**, and authorize access to your Google account.

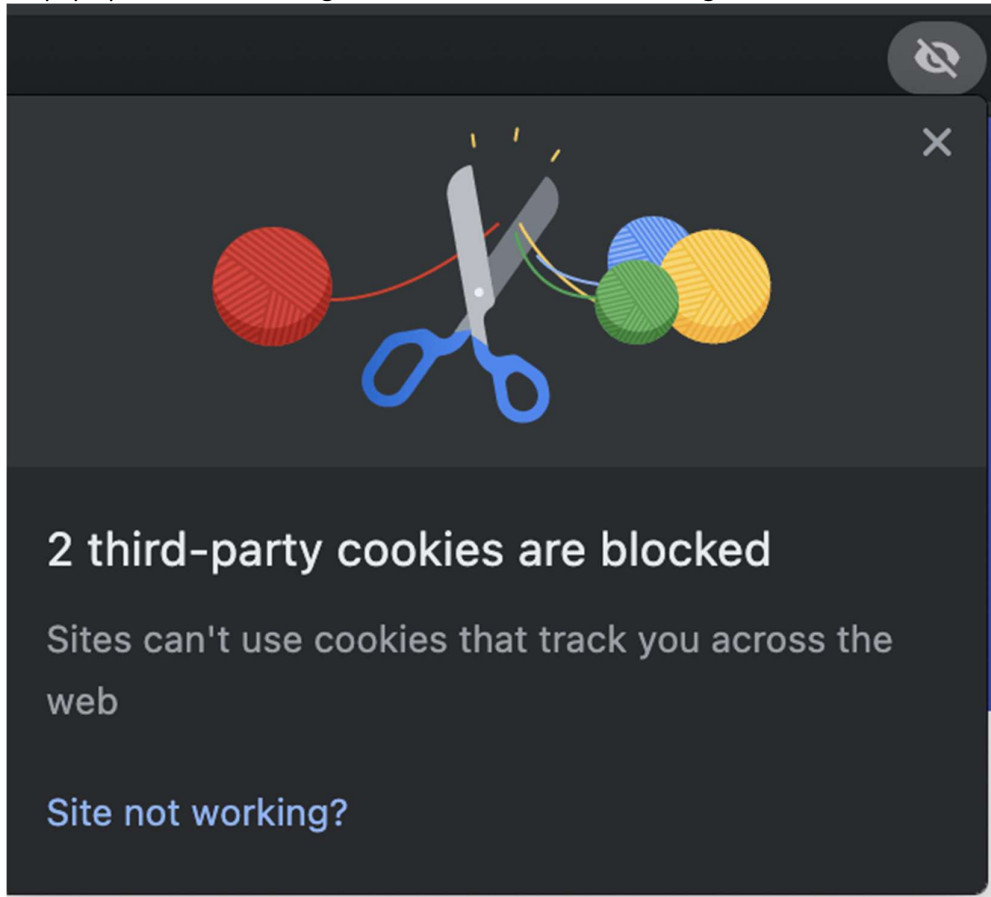
Your Qwiklabs name and email address should be displayed in the application's header.



Using Google Sign-in

For simplicity, the sample app in this lab doesn't have a user or login system, and Google Sign-in is only used to obtain the authorization needed to make requests to the Google Sheets API. In a real application, you should use Google sign-in as a way to onboard and login users as well. The documentation on how to [Authenticate with a backend server](#) goes into more detail.

If running on an Incognito page, you may need to enable cookies on the page in order to see that you've successfully signed it. You can do this by clicking on the eye icon in the URL of your browser. Choosing "Site not working?" from the pop up. And then, clicking **Allow Cookies** from the following window.



Add Spreadsheet Controls

We want to keep track of the spreadsheets our application has created, so that as the data in our application changes the spreadsheets can be updated. In order to do that we need to create a table in our database to store information about the spreadsheets and add some controls to our UI.

In the code editor, click the `models` directory, then select **File > New File** to create a new file in the `models` directory. Name the new file `spreadsheets.js` and add the following code:

```
"use strict";

module.exports = function(sequelize, DataTypes) {
  var Spreadsheet = sequelize.define('Spreadsheet', {
    id: {type: DataTypes.STRING, allowNull: false, primaryKey: true},
    sheetId: {type: DataTypes.INTEGER, allowNull: false},
    name: {type: DataTypes.STRING, allowNull: false}
  });

  return Spreadsheet;
};
```

Save the file.

This uses the [Sequelize ORM](#) to define a new table that stores the ID, sheet ID, and name of the spreadsheets we create.

Next, we need to fetch all of the spreadsheets we've stored when the index page loads, so we can display them in a list. In `routes.js`, replace the code for the `"/"` route with the following:

```
router.get('/', function(req, res, next) {
  var options = {
    order: [['createdAt', 'DESC']],
    raw: true
  };
  Sequelize.Promise.all([
    models.Order.findAll(options),
    models.Spreadsheet.findAll(options)
  ]).then(function(results) {
    res.render('index', {
      orders: results[0],
      spreadsheets: results[1]
    });
  });
});
```

Save the file.

Next, we need to display the list of spreadsheets in the template. Add the following code to the end of `views/index.handlebars`, within the existing `<div class="mdl-grid">`:

```

<section id="spreadsheets"
  class="mdl-cell mdl-cell--4-col relative">
  <div class="mdl-list">
    <div class="mdl-list__item">
      <span class="mdl-list__item-primary-content mdl-layout-title">
        Spreadsheets</span>
      <span class="mdl-list__item-secondary-action">
        <button class="mdl-button mdl-js-button mdl-button--raised
          mdl-js-ripple-effect mdl-button--colored"
            rel="create" type="button">Create</button>
      </span>
    </div>
    {{#each spreadsheets}}
    <div class="mdl-list__item">
      <a class="mdl-list__item-primary-content"
        href="https://docs.google.com/spreadsheets/d/{{id}}/edit"
        target="_blank">{{name}}</a>
      <span class="mdl-list__item-secondary-action">
        <button class="mdl-button mdl-js-button mdl-button--raised
          mdl-js-ripple-effect"
            rel="sync" data-spreadsheetid="{{id}}"
            type="button">Sync</button>
      </span>
    </div>
    {{/each}}
  </div>
</section>

```

Save the file.

Finally, we need to wire up the create and sync spreadsheet buttons. Add the following code to `public/script.js`:

```

$(function() {
  $('button[rel="create"]').click(function() {
    makeRequest('POST', '/spreadsheets', function(err, spreadsheet) {
      if (err) return showError(err);
      window.location.reload();
    });
  });
  $('button[rel="sync"]').click(function() {
    var spreadsheetId = $(this).data('spreadsheetid');
    var url = '/spreadsheets/' + spreadsheetId + '/sync';
    makeRequest('POST', url, function(err) {
      if (err) return showError(err);
      showMessage('Sync complete.');
```

```
},
error: function(response) {
  setSpinnerActive(false);
  return callback(new Error(response.responseJSON.message));
}
});
}
```

Save the file.

Storing credentials

In this sample application, we're simply passing the OAuth2 access token to the backend with each button click. In a real application consider storing credentials in the authenticated user's session for easier retrieval in the backend.

Reload the application in your browser and you should see the new spreadsheets section on the right-hand side or bottom of the screen.

The screenshot shows a web application titled "Customer Order System". In the top right corner, it displays "Test User" with the email "test.user@appsrocks.com" and a "Signed in" status with a Google logo. The main content area features two order cards. The first card, "Order #5", lists details for "Elizabeth's Eatery": Product code FOO-100, Units ordered 35, Unit price (USD) 10.95, and Status PENDING. It has "EDIT" and "DELETE" buttons. The second card, "Order #4", lists details for "David's Dog Grooming": Product code FOO-250, Units ordered 15, Unit price (USD) 29.95, and Status PENDING. It also has "EDIT" and "DELETE" buttons. A red circular button with a white "+" sign is positioned between the two order cards. To the right of the order cards, there is a "Spreadsheets" section with a blue "CREATE" button.

Note: depending on where you inserted your elements, the new spreadsheets section might be on the left-hand side of the page.

Since the database is empty there are no spreadsheets to show, and the create button won't do anything just yet.

Creating Spreadsheets

The Google Sheets API provides the ability to create and update spreadsheets. To start using it install the [Google APIs Node.js client library](#) and the companion auth library. Click **Open Terminal** and then in Cloud Shell, click the plus sign to open a second command line interface.



Change your current working directory:

```
cd sheets-api/start
```

Run the following commands in your console:

```
npm install googleapis@26.* --save
npm install google-auth-library@1.* --save
```

Next we'll create a helper class that will use the libraries to create and update our spreadsheets. Back in code editor, create a file called `sheets.js` in the `sheets-api/start` directory of the application with the following code:

```
var {google} = require('googleapis');
var {OAuth2Client} = require('google-auth-library');
var util = require('util');

var SheetsHelper = function(accessToken) {
  var auth = new OAuth2Client();
  auth.credentials = {
    access_token: accessToken
  };
  this.service = google.sheets({version: 'v4', auth: auth});
};

module.exports = SheetsHelper;
```

Given an OAuth2 access token, this class creates the credentials and initializes the Sheets API client.

Next we'll add a method for creating a spreadsheet. Add the following to the end of `sheets.js`:

[sheets.js](#)

```
SheetsHelper.prototype.createSpreadsheet = function(title, callback) {
  var self = this;
  var request = {
    resource: {
      properties: {
        title: title
      },
      sheets: [
        {
          properties: {
            title: 'Data',
            gridProperties: {
```

```

        columnCount: 6,
        frozenRowCount: 1
      }
    },
    // TODO: Add more sheets.
  ]
}
};
self.service.spreadsheets.create(request, function(err, response) {
  if (err) {
    return callback(err);
  }
  var spreadsheet = response.data;
  // TODO: Add header rows.
  return callback(null, spreadsheet);
});
};

```

Save the file.

This method defines a simple [Spreadsheet](#) object and calls the [spreadsheets.create](#) method to create it on the server.

Finally, we need to add a new route to our application that takes the request from the spreadsheet controls, calls the helper to create the spreadsheet, and then saves a record in the database. Add the following code to the end of `routes.js`:

[routes.js](#)

```

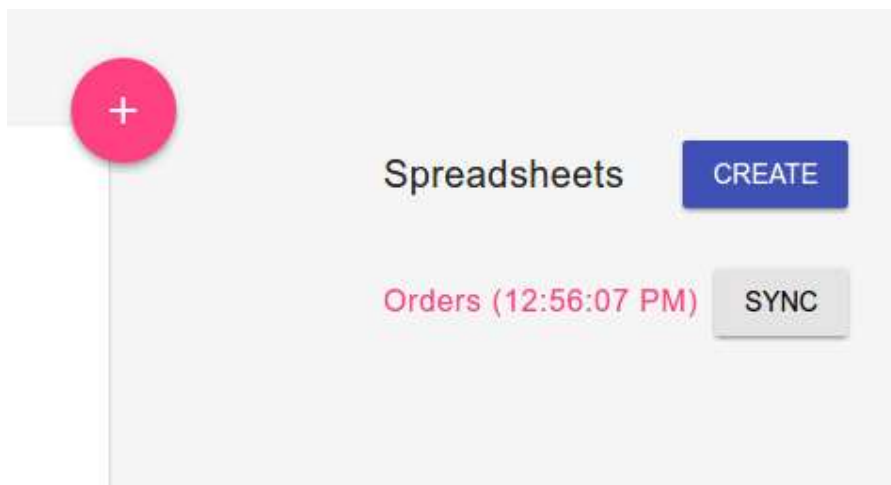
var SheetsHelper = require('./sheets');

router.post('/spreadsheets', function(req, res, next) {
  var auth = req.get('Authorization');
  if (!auth) {
    return next(Error('Authorization required.'));
  }
  var accessToken = auth.split(' ')[1];
  var helper = new SheetsHelper(accessToken);
  var title = 'Orders (' + new Date().toLocaleTimeString() + ')';
  helper.createSpreadsheet(title, function(err, spreadsheet) {
    if (err) {
      return next(err);
    }
    var model = {
      id: spreadsheet.spreadsheetId,
      sheetId: spreadsheet.sheets[0].properties.sheetId,
      name: spreadsheet.properties.title
    };
    models.Spreadsheet.create(model).then(function() {
      return res.json(model);
    });
  });
});

```

Save the file.

Reload the application in your browser and click **Create**. The following should appear:



A new spreadsheet is created and displayed in the list. Click on the spreadsheet's name to open it, and you'll see that it has one sheet called Data which is currently blank.

A screenshot of the Google Sheets application window for a spreadsheet titled 'Orders (12:56:07 PM)'. The window has a green header bar with a spreadsheet icon on the left and the title 'Orders (12:56:07 PM)' in the center, followed by a star icon and a folder icon. On the right of the header bar, the email 'test.user@appsrocks.com' is shown with a dropdown arrow. Below the header bar is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Format', 'Data', 'Tools', 'Add-ons', and 'Help'. To the right of the menu bar are two buttons: 'Comments' and 'Share'. Below the menu bar is a toolbar with various icons for formatting and editing. The main area is a grid with columns labeled A through F and rows numbered 1 through 9. The cell at the intersection of column A and row 1 is selected, indicated by a blue border. The grid is currently blank.

Add Header Row

Now that we are creating spreadsheets, let's start formatting them nicely, starting with a header row. We'll have the application add this header row right after it creates the spreadsheet.

In the Cloud Shell code editor, in `sheets.js`, **replace** `return callback(null, spreadsheet);` in the method `createSpreadsheet` with the following:

[sheets.js](#)

```
var dataSheetId = spreadsheet.sheets[0].properties.sheetId;
var requests = [
  buildHeaderRowRequest(dataSheetId),
];
// TODO: Add pivot table and chart.
var request = {
  spreadsheetId: spreadsheet.spreadsheetId,
  resource: {
    requests: requests
  }
};
self.service.spreadsheets.batchUpdate(request, function(err, response) {
  if (err) {
    return callback(err);
  }
  return callback(null, spreadsheet);
});
```

This code uses the Sheets API's [spreadsheets.batchUpdate](#) method, which is used for nearly every type of manipulation to a spreadsheet. The method takes an array of [Request](#) objects as input, each of which contains the specific type of request (operation) to perform on the spreadsheet. In this case, we're only passing a single request to format the header row, which we'll define in a minute.

Next we'll need to define the column headers. Add the following code to the end of `sheets.js`:

[sheets.js](#)

```
var COLUMNS = [
  { field: 'id', header: 'ID' },
  { field: 'customerName', header: 'Customer Name' },
  { field: 'productCode', header: 'Product Code' },
  { field: 'unitsOrdered', header: 'Units Ordered' },
  { field: 'unitPrice', header: 'Unit Price' },
  { field: 'status', header: 'Status' }
];
```

This code also defines the corresponding fields in the Order object (AKA database columns) which we'll use later on.

Finally let's define the `buildHeaderRowRequest` method we referenced earlier. In the same file add the following:

[sheets.js](#)

```
function buildHeaderRowRequest(sheetId) {
  var cells = COLUMNS.map(function(column) {
    return {
```

```

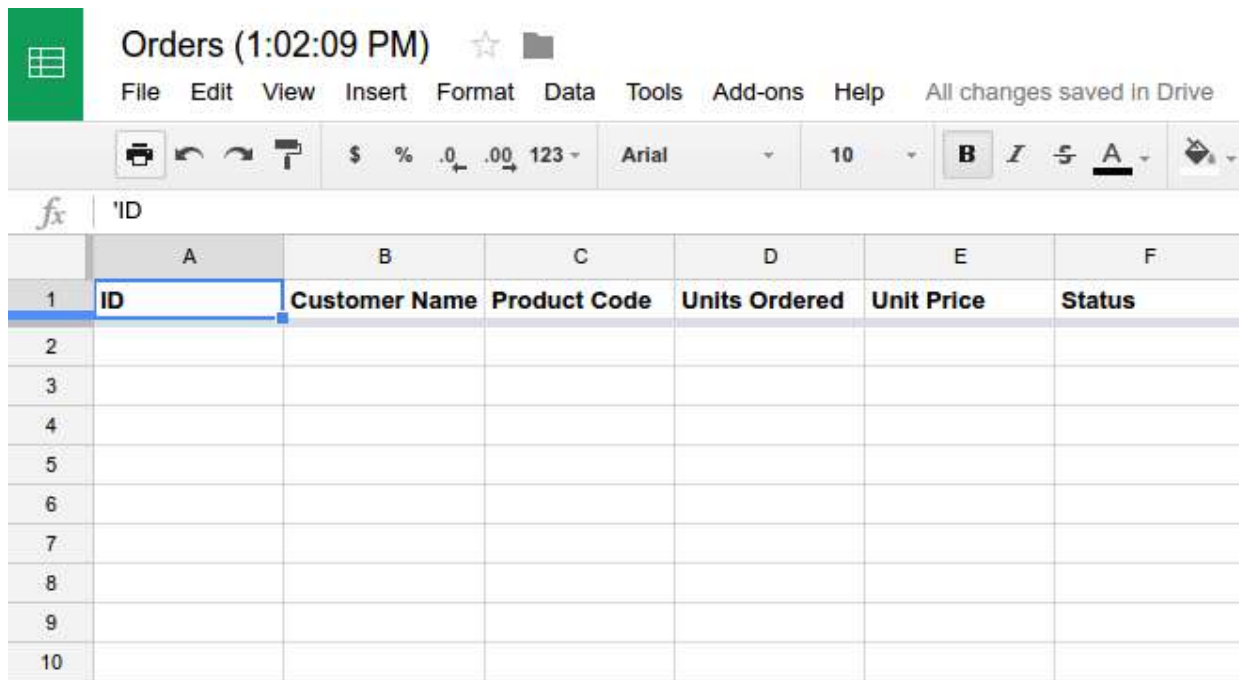
    userEnteredValue: {
      stringValue: column.header
    },
    userEnteredFormat: {
      textFormat: {
        bold: true
      }
    }
  }
});
return {
  updateCells: {
    start: {
      sheetId: sheetId,
      rowIndex: 0,
      columnIndex: 0
    },
    rows: [
      {
        values: cells
      }
    ],
    fields: 'userEnteredValue,userEnteredFormat.textFormat.bold'
  }
};
}

```

Save the file.

This code loops over each column and creates a [CellData](#) object for each one, which has the column's title as the value and which sets the formatting to bold. All of these cells are assembled together into an [UpdateCells](#) request and returned. The fields parameter is required and specifies exactly which fields of the CellData object to look at when applying the changes.

Reload the application in your browser and click **Create**. The resulting spreadsheet should include a header row with a column for each field we defined:



	A	B	C	D	E	F
1	ID	Customer Name	Product Code	Units Ordered	Unit Price	Status
2						
3						
4						
5						
6						
7						
8						
9						
10						

Sync Data to the Spreadsheet

Of course, all of this creating and formatting a spreadsheet is meaningless if you don't put any actual data into it.

Let's add a new route to `routes.js` that will kick off a sync:

[routes.js](#)

```
router.post('/spreadsheets/:id/sync', function(req, res, next) {
  var auth = req.get('Authorization');
  if (!auth) {
    return next(Error('Authorization required.'));
  }
  var accessToken = auth.split(' ')[1];
  var helper = new SheetsHelper(accessToken);
  Sequelize.Promise.all([
    models.Spreadsheet.findByIdPk(req.params.id),
    models.Order.findAll()
  ]).then(function(results) {
    var spreadsheet = results[0];
    var orders = results[1];
    helper.sync(spreadsheet.id, spreadsheet.sheetId, orders, function(err) {
      if (err) {
        return next(err);
      }
      return res.json(orders.length);
    });
  });
});
```

Save the file.

Like the previous route for creating spreadsheets, this one checks for authorization, loads models from the database, and then passes the information to the `SheetsHelper` which will transform the records to cells and make the API requests. Add the following code to `sheets.js` to do just that:

[sheets.js](#)

```
SheetsHelper.prototype.sync = function(spreadsheetId, sheetId, orders, callback) {
  var requests = [];
  // Resize the sheet.
  requests.push({
    updateSheetProperties: {
      properties: {
        sheetId: sheetId,
        gridProperties: {
          rowCount: orders.length + 1,
          columnCount: COLUMNS.length
        }
      }
    },
    fields: 'gridProperties(rowCount,columnCount)'
  });
  // Set the cell values.
  requests.push({
    updateCells: {
      start: {
        sheetId: sheetId,
        rowIndex: 1,
```

```

        columnIndex: 0
      },
      rows: buildRowsForOrders(orders),
      fields: '*'
    }
  });
  // Send the batchUpdate request.
  var request = {
    spreadsheetId: spreadsheetId,
    resource: {
      requests: requests
    }
  };
  this.service.spreadsheets.batchUpdate(request, function(err) {
    if (err) {
      return callback(err);
    }
    return callback();
  });
};

```

Here again we're using the `batchUpdate` method, this time passing in two requests. The first is an [UpdateSheetPropertiesRequest](#) which resizes the sheet to ensure there are enough rows and columns to fit the data it's about to write. The next is another [UpdateCells](#) request, which sets the cell values and formatting. The `buildRowsForOrders` function is where we convert the `Order` objects into cells. Add the following code to the same file:

[sheets.js](#)

```

function buildRowsForOrders(orders) {
  return orders.map(function(order) {
    var cells = COLUMNS.map(function(column) {
      switch (column.field) {
        case 'unitsOrdered':
          return {
            userEnteredValue: {
              numberValue: order.unitsOrdered
            },
            userEnteredFormat: {
              numberFormat: {
                type: 'NUMBER',
                pattern: '#,##0'
              }
            }
          };
          break;
        case 'unitPrice':
          return {
            userEnteredValue: {
              numberValue: order.unitPrice
            },
            userEnteredFormat: {
              numberFormat: {
                type: 'CURRENCY',
                pattern: '"$"#,##0.00'
              }
            }
          };
          break;
        case 'status':
          return {
            userEnteredValue: {
              stringValue: order.status
            },
            dataValidation: {

```

```

        condition: {
          type: 'ONE_OF_LIST',
          values: [
            { userEnteredValue: 'PENDING' },
            { userEnteredValue: 'SHIPPED' },
            { userEnteredValue: 'DELIVERED' }
          ]
        },
        strict: true,
        showCustomUi: true
      }
    };
    break;
  default:
    return {
      userEnteredValue: {
        stringValue: order[column.field].toString()
      }
    };
  }
});
return {
  values: cells
};
});
}

```

Save the file

The `unitsOrdered` and `unitPrice` fields set a numeric value as well as number format to ensure the values are displayed correctly. Additionally, the `status` field has a data validation set in order to display a dropdown of the allowed status values. Although not particularly useful in this lab, adding data validation to the spreadsheet can be useful if you want to allow users to edit the rows and later impact them back into your application.

Reload the application in your browser and click the Sync button next to the spreadsheet link. The spreadsheet should now contain all your order data. Add a new order and click Sync again to see the changes.

	A	B	C	D	E	F
1	ID	Customer Name	Product Code	Units Ordered	Unit Price	Status
2	1	Alice's Antiques	FOO-100	25	\$12.50	DELIVERED ▾
3	2	Bob's Brewery	FOO-200	60	\$18.75	SHIPPED ▾
4	3	Carol's Car Wash	FOO-100	100	\$9.25	SHIPPED ▾
5	4	David's Dog Groo	FOO-250	15	\$29.95	PENDING ▾
6	5	Elizabeth's Eatery	FOO-100	35	\$10.95	PENDING ▾

Add a Pivot Table and Chart

Your application now exports to Google Sheets, but honestly, you could have achieved a similar result by exporting CSVs and manually importing them into Google Sheets. What separates this API-based approach from CSV is the ability to add complex features to spreadsheets, such as pivot tables and charts. This allows you to leverage Google Sheets as a dashboard to your data that users can easily customize and extended.

To get started, add a new sheet to our spreadsheets to contain the pivot table and chart. It's best to keep the sheets of raw data separate from the aggregations and visualizations, so that your syncing code can just focus on the data. In `sheets.js`, add the following code to the array of sheets being created in `SheetsHelper's createSpreadsheet` method:

[sheets.js](#)

```
{
  properties: {
    title: 'Pivot',
    gridProperties: {
      hideGridlines: true
    }
  }
}
```

Later on in `createSpreadsheet` method, we'll need to capture the ID of the "Pivot" sheet and use it to build some new requests. Add the following code after `var requests = \[...\]`:

[sheets.js](#)

```
var pivotSheetId = spreadsheet.sheets[1].properties.sheetId;
requests = requests.concat([
  buildPivotTableRequest(dataSheetId, pivotSheetId),
  buildFormatPivotTableRequest(pivotSheetId),
  buildAddChartRequest(pivotSheetId)
]);
```

Finally, add the following functions to the file, which create requests for building the pivot table, formatting the results, and adding the chart:

[sheets.js](#)

```
function buildPivotTableRequest(sourceSheetId, targetSheetId) {
  return {
    updateCells: {
      start: { sheetId: targetSheetId, rowIndex: 0, columnIndex: 0 },
      rows: [
        {
          values: [
            {
              pivotTable: {
                source: {
                  sheetId: sourceSheetId,
                  startRowIndex: 0,
                  startColumnIndex: 0,
                  endColumnIndex: COLUMNS.length
                },
                rows: [
                  {
                    sourceColumnOffset: getColumnForField('productCode').index,
```

```

        showTotals: false,
        sortOrder: 'ASCENDING'
    }
],
values: [
    {
        summarizeFunction: 'SUM',
        sourceColumnOffset: getColumnForField('unitsOrdered').index
    },
    {
        summarizeFunction: 'SUM',
        name: 'Revenue',
        formula: util.format("='%s' * '%s'",
            getColumnForField('unitsOrdered').header,
            getColumnForField('unitPrice').header)
    }
]
}
}
]
}
],
fields: '*'
}
};
}

```

```

function buildFormatPivotTableRequest(sheetId) {
    return {
        repeatCell: {
            range: { sheetId: sheetId, startRowIndex: 1, startColumnIndex: 2 },
            cell: {
                userEnteredFormat: {
                    numberFormat: { type: 'CURRENCY', pattern: '"$"#,##0.00' }
                }
            },
            fields: 'userEnteredFormat.numberFormat'
        }
    };
}

```

```

function buildAddChartRequest(sheetId) {
    return {
        addChart: {
            chart: {
                spec: {
                    title: 'Revenue per Product',
                    basicChart: {
                        chartType: 'BAR',
                        legendPosition: 'RIGHT_LEGEND',
                        domains: [
                            // Show a bar for each product code in the pivot table.
                            {
                                domain: { sourceRange: { sources: [{
                                    sheetId: sheetId,
                                    startRowIndex: 0,
                                    startColumnIndex: 0,
                                    endColumnIndex: 1
                                }]}
                            }
                        ]
                    },
                    series: [
                        // Set that bar's length based on the total revenue.
                        {
                            series: { sourceRange: { sources: [{
                                sheetId: sheetId,
                                startRowIndex: 0,

```

```

        startColumnIndex: 2,
        endColumnIndex: 3
      ]]]]
    }
  ]
}
},
position: {
  overlayPosition: {
    anchorCell: { sheetId: sheetId, rowIndex: 0, columnIndex: 3 },
    widthPixels: 600,
    heightPixels: 400
  }
}
}
}
};
}

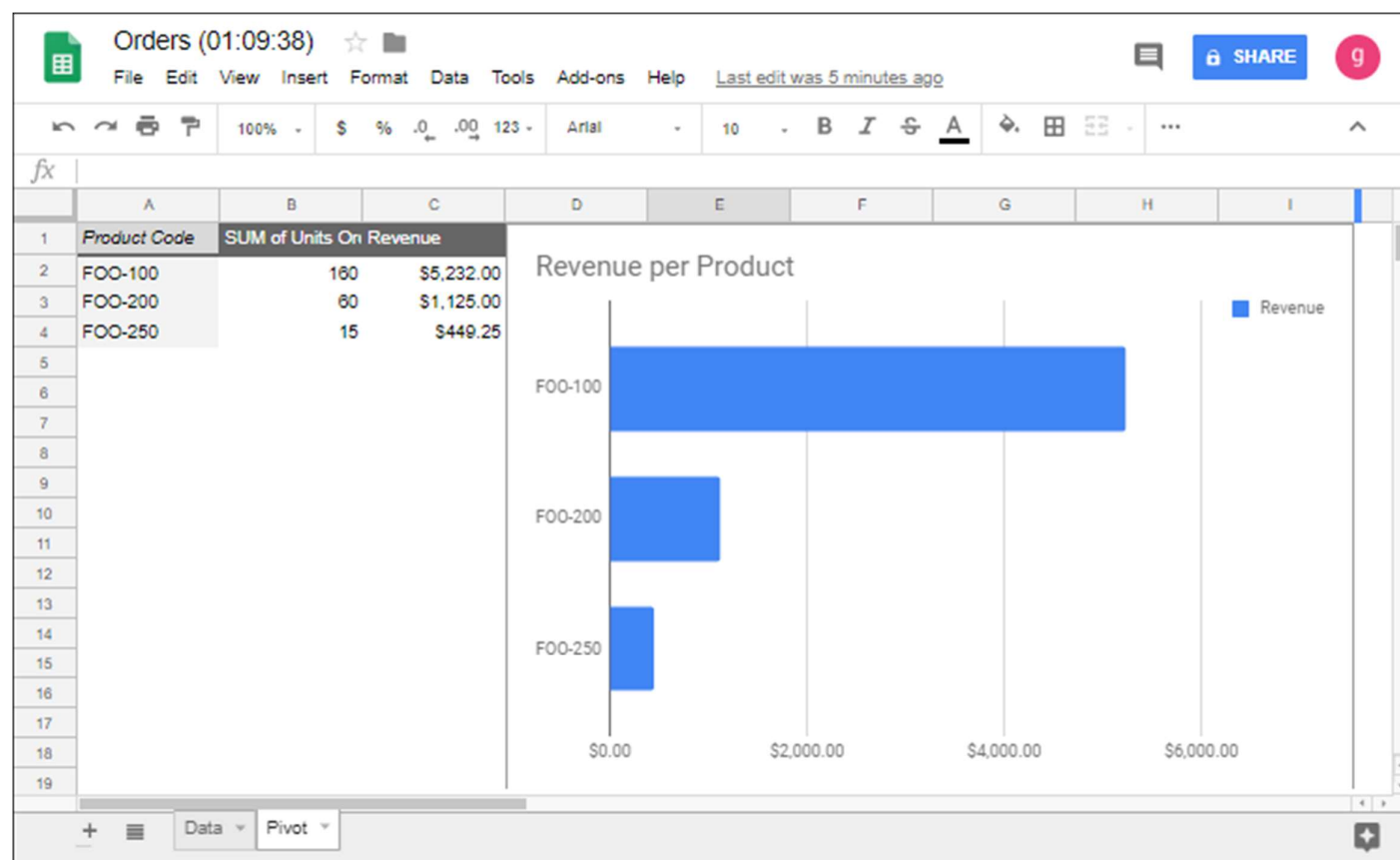
function getColumnForField(field) {
  return COLUMNS.reduce(function(result, column, i) {
    if (column.field == field) {
      column.index = i;
      return column;
    }
  });
  return result;
});
}

```

Save the file.

Reload the application in your browser and click the Create button again. The resulting spreadsheet should contain a **Data** sheet and a **Pivot** sheet. The Pivot sheet contains an empty pivot table and chart.

Click **Sync** to add data to the spreadsheet, and watch the pivot table and chart come to life with real data.



Congratulations!

You've successfully modified an application to export data to Google Sheets. Your users can now build custom reports and dashboards over your data without the need for additional code, while being kept perfectly in sync as the data changes.

Finish Your Quest



This self-paced lab is part of the [Workspace Integrations](#) Quest. A Quest is a series of related labs that form a learning path. Completing this Quest earns you the badge above, to recognize your achievement. You can make your badge (or badges) public and link to them in your online resume or social media account. [Enroll in this Quest](#) and get immediate completion credit if you've taken this lab. [See other available Qwiklabs Quests](#).

Take your next lab

If you haven't already, be sure to check out the other Qwiklabs on Google apps and tools, like:

- [Google Apps Script: Access Google Sheets, Maps & Gmail in 4 Lines of Code](#)

Learn more

- Read the Google Sheets API [developer documentation](#).
- Post questions and find answers on Stackoverflow under the [google-sheets-api](#) tag.
- [Google Workspace Learning Center](#)

Think about possible improvements

Here are some additional ideas for making an even more compelling integration:

- Automatic sync Rather than have the user click a button to sync, update the spreadsheets automatically whenever orders are changed. You'd need to request [offline access](#) from your users, and keep track of which user owns a given spreadsheet.
- Two-way sync Allow users to modify order details in the spreadsheet and push those changes back into the application. You could use [Google Drive push notifications](#) to detect when the spreadsheet has been updated and the Sheets API's [spreadsheets.values.get](#) to pull in the updated data.

Google Cloud Training & Certification

...helps you make the most of Google Cloud technologies. [Our classes](#) include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual options to suit your busy schedule. [Certifications](#) help you validate and prove your skill and expertise in Google Cloud technologies.

Manual Last Updated February 16, 2021

Lab Last Tested December 10, 2020

Copyright 2021 Google LLC All rights reserved. Google and the Google logo are trademarks of Google LLC. All other company and product names may be trademarks of the respective companies with which they are associated.