

Run a Cisco Spark Integration locally

Duration: 15 minutes

Objectives

As introduced in previous labs, Cisco Spark Integrations and Bots concretize Cisco Spark's extensibility.

Cisco Spark Integrations let your applications request permission to invoke the Cisco Spark API on behalf of other users. The process used to request permissions is called "OAuth Grant Flow": it is documented in the [Integrations guide](#), and can be experimented in the Learning Lab [Understand the OAuth Grant flow for Cisco Spark Integrations](#).

In this lab, you will run an existing NodeJS code sample on your local machine, and register it as a Cisco Spark Integration to experiment Cisco Spark's OAuth scopes.

How to setup your own computer

Skip this section if you are using a machine provided by the event organizers.

In this learning lab, we will run a Cisco Spark Integration example. This requires a minimal NodeJS development environment on your local machine.

- NodeJS runtime: make sure you have both the "node" and "npm" runtimes installed locally on your machine. This lab has been tested with NodeJS v4.
- [optional] a NodeJS IDE: this lab does not require you to open any NodeJS file locally, but this is certainly something you'll want to do, and will be mandatory if you opt for the "to go further" instructions.
- "git" and "make" commands

Pre-requisites

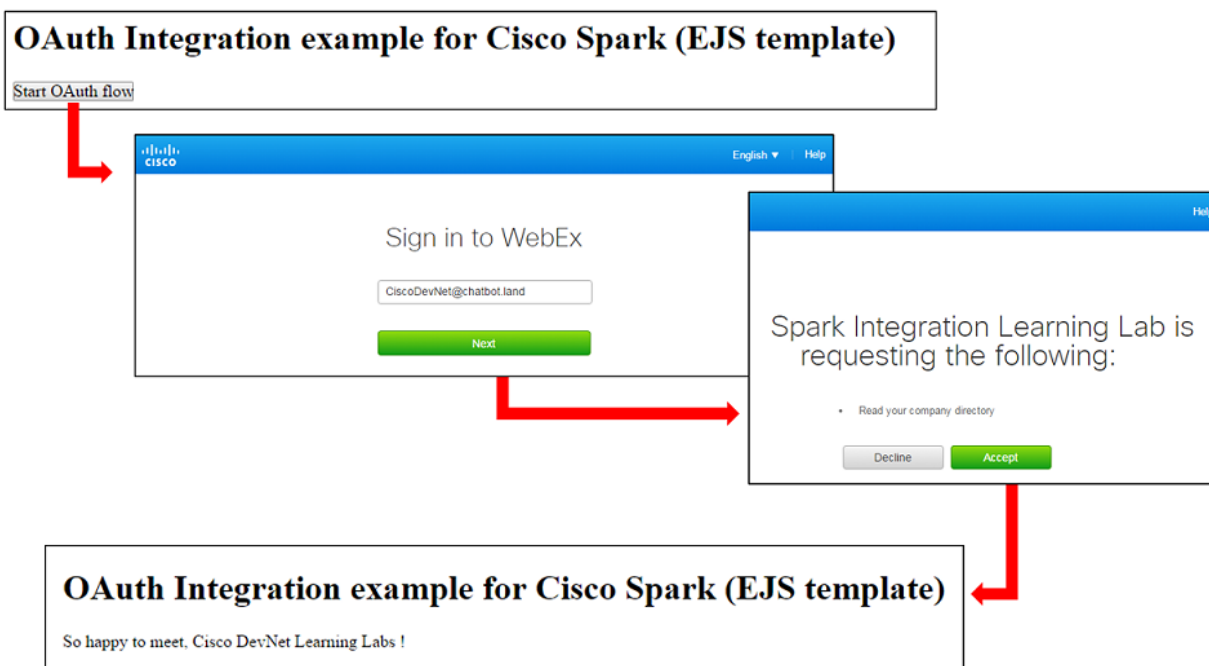
You will need a Cisco Spark user account to complete this lab. If you're not a Cisco Spark user yet, [click to sign up](#).

If you're using your own computer, make sure it meets the requirements listed above.

Step 1: Clone the Cisco Spark Integration example

For the sake of this lab, we will leverage a [ready-to-run Cisco Spark Integration](#).

This integration example consists in a [NodeJS server](#) that serves an HTML page which initiates a Cisco Spark OAuth Grant flow, and drives the end-user to a consent form. This integration also listens on a “redirect URL” that is invoked by Cisco Spark as the end-user accepts or declines to grant permissions.



Run the commands below to clone the [github project](#), get a local copy of the provided example, install the project dependencies and launch your integration.

Make sure your local environment meet all the pre-requisities. The git, NodeJS and npm commands are required to copy and launch the example on your local machine. If your environment does not provide the "make" command, you can paste the run section of the Makefile as an alternative.

```
> git clone https://github.com/CiscoDevNet/spark-integration-sample
> cd spark-integration-sample
> npm install
> make
oauth Cisco Spark OAuth Integration started on port: 8080 +0ms
```

Check your integration's home page is now live at <http://localhost:8080>.

Click on the "Start OAuth flow" button, and go through the OAuth Grant flow as described on the picture above.

At the end of the flow, check your Cisco Spark full name is successfully displayed.


Step 2: Register your Cisco Spark Integration

In this step, we will register our integration on Cisco Spark.

Go to the Cisco Spark Developer portal and access the [“New integration”](#) page.

Fill in the form with a name, a description, an email and an icon for your integration. As the icon size must be 512x512 or larger and publicly accessible, you can leveraging this [DevNet logo png](#).

In the redirect URI(s) field, specify <http://localhost:8080/oauth>.

Name	<input type="text" value="Spark Integration Learning Lab"/>
Description	<div><div>An example of Spark integration that initiates the OAuth flow from <code>http://localhost:8080/</code> and redirects to <code>http://localhost:8080/oauth</code></div><div>Moreover, the integration declares all OAuth scopes, so that we can specify mandatory scopes at runtime.</div></div>
Support Email	<input type="text" value="stsfartz@cisco.com"/>
App Icon	<div><input type="text" value="https://cdn-images-1.medium.com/max/800/1*L9U_KTqEpF-IJx"/></div>
Redirect URI(s)	<div><input type="text" value="http://localhost:8080/oauth"/>Add URI</div>

The last section of the form concerns the OAuth scopes of your integration. These scopes correspond to the set of authorizations that the end-user will be asked to grant. For the sake of this lab, we will refine the scope [at runtime](#).

Check the scopes listed below, and click “Save changes”.

Scopes

Scopes define the level of access that your integration requires. [Learn more](#)

- ☒ `spark:people_read` Read your users' company directory
- ☒ `spark:rooms_read` List the titles of rooms that your user's are in
- ☒ `spark:rooms_write` Manage rooms on your users' behalf
- ☒ `spark:memberships_read` List people in the rooms your user's are in
- ☒ `spark:memberships_write` Invite people to rooms on your users' behalf
- ☒ `spark:messages_read` Read the content of rooms that your user's are in
- ☒ `spark:messages_write` Post and delete messages on your users' behalf
- ☒ `spark:teams_read` List the teams your users are in
- ☒ `spark:teams_write` Create teams on your users' behalf
- ☒ `spark:team_memberships_read` List the people in the teams your user's belong to
- ☒ `spark:team_memberships_write` Add people to teams on your users' behalf

Cisco Spark automatically generates a unique Client ID and Secret for our Cisco Spark Integration.

Leave this page open, or keep these values in a safe place as we will use them in the next step.

Note that you don't need to copy the OAuth Authorization URL since our code sample does [dynamically generate](#) this URL from the Client Id, Client Secret, Redirection URL, State and OAuth Scopes.

Step 3: Configure your integration

We will now configure our integration to use the Auth client and secret we just created.

Go to the integration code folder on your local machine, look for a file named [Makefile](#), and open it in a Text editor.

Modify CLIENT_ID and CLIENT_SECRET with the values generated in previous step.

OAuth Settings

Learn more about authentication in the
[Apps & OAuth Guide](#)

Client ID

C9901101c66249d7e6b7cb174941a400e2e01f7d80d0b1f08b11665bad5cbb66

Copy

Client Secret

aaa8f0304a9b49a1654b74a14faf7b939481341ab09c9e47bab9d7c1e54e62a7

Copy

OAuth Authorization URL

You can use the URL below to initiate an OAuth permission request for this app. It is configured with your redirect URI and app scopes. Be sure to update the state parameter.

```
https://api.ciscospark.com/v1/authorize?client_id=C9901101c66249d7e6b7cb174941a400e2e01f7d80d0b1f08b11665bad5cbb66&response_type=code&redirect_uri=http%3A%2F%2Flocalhost%3A8080%2Foauth&scope=spark%3Amessages_write%20spark%3Arooms_read%20spark%3Ateams_read%20spark%3Amemberships_read%20spark%3Amessages_read%20spark%3Arooms_write%20spark%3Apeople_read%20spark%3Akms%20spark%3Amemberships_write%20spark%3Ateams_write%20spark%3Ateam_memberships_read%20spark%3Ateam_memberships_write&state=set_state_here
```

Save the file, and restart your integration.

Open your integration home page at <http://localhost:8080>.

Go through the OAuth Grant Flow again, and check it now displays the name and image of your integration.

Step 4: Understanding the OAuth Grant flow and OAuth scopes

We will now go through our integration source code, and drill into the various steps involved to be granted an access token acting on behalf of Cisco Spark users.

Before reading the code explanations below, we recommend you go through the [authentication principles for Integrations](#) on "Spark for Developers".

The core logic of your integration takes place in file named [“server.js”](#)

[Lines 26 and below](#): this is where our integration reads its configuration. This can be modified at runtime by setting various environment variables, or directly by updating the source code. Note that a default configuration is provided, it refers to a pre-registered Cisco Spark integration running on localhost.

[Lines 40 and below](#): our goal here to retrieve an authorization code. To do so, the integration registers a default route to serve an EJS template “initiate.ejs”. The EJS template give us the flexibility to inject the OAuth flow redirect URL. This URL is computed from the configuration above, and invoked by the Start flow button. Moreover, a File server is mounting the “www” directory at the service Root URI, in order to serve static files (html, css...).

[Lines 69 and below](#): this is where our integration is invoked back as the Cisco Spark end-user grants or declines the permission to access his data. This section of code checks the query parameters sent by Cisco Spark. If everything looks good, we issue a new request to Cisco Spark with the authorization code we just received, at [lines 113 and below](#). The Cisco Spark API responds with a JSON payload that contains an access token and a refresh token as well as expiration dates.

[Lines 170 and below](#): the OAuth grant flow is now completed: our code can use the access token to invoke the Cisco Spark API, with the authorizations granted by the end-user.

We now know enough to tweak the code example for our own needs.

Modify your Cisco Spark Integration

Our NodeJS code currently asks permission for the “**spark:people_read**” scope, in order to be authorized to access the Cisco Spark end-user's full name.

Open file [server.js](#).

Change the requested scopes to “spark:rooms_details” at line 30.

Restart your integration, and take the OAuth flow starting from your integration home page at <http://localhost:8080>.

Note that the OAuth Grant Flow now asks for the new scope.

Once you have granted access to your integration, check an error page is displayed: the call to the Cisco Spark API now returns a 403(Forbidden) status code as our access token does not contain the people-read scope!

To go further

What about making your integration now list the rooms that the Cisco Spark end-user is a member of. Here are a few tips if you opt to implement this use case:

- change the OAuth permission with the [“spark:rooms read” scope](#),
- switch the EJS template to the [rooms-list.ejs](#) template provided.
- And load the user’s list of rooms by calling the [Spark API List Rooms](#) resource.