CA Release Automation
CDE Plug-in Creation
Best Practices Guide

Author :	Walter Guerrero
Version:	2.0
Filename:	CA-Release-Automation-CDE-Plug-in-Creation-Best-Practices-GuideV2.0.docx
Date:	12/7/2016

Table of Contents

Overview

Customers in their Continuous Delivery journey that is part of their Agile maturity have a new tool called Release Automation Continuous Delivery Edition, which permits customers the ability to build “releases”; and these releases can be part of a sprint, program increments, or milestones.

These releases are composed of multiple applications’ contents (which is what is going to be delivered at the end of the sprint or program increment). It becomes important as part of these releases that the different tools being used by the customer can be called directly by tasks that make up the RA CD Edition release.

Release Automation CDE provides the customers with the ability of creating additional plug-ins.

This document will take you thru the process that it will take for the creation of a RA CDE plug-in based off an offline JAVA project, which is built utilizing Maven as the build infrastructure.

The flow for the creation of RA CDE plug-in is as follows:

- Design the plug-in by selecting the proper Restful/SDK/client API calls to be used.
- Test the API calls independently.
- Setup the JAVA project utilizing Eclipse or IntelliJ.
- Convert the JAVA project in the IDE to a Maven project or any other build infrastructure.
- Implement the RA CDE plug-in JSON manifest
- Add the plug-in to the RA CDE implementation by importing the JSON manifest.
- Add an endpoint pointing to the desired remote component implementation.
- Add a task to a RA CDE release.

RA CDE Plug-in Framework

The RA CDE plug-in implementation follows the steps listed above, now we are going to start the detailed explanation of this process.

The steps below will help you understand how a RA CDE HTTP service based plug-in is developed and the steps that you need to take.

The plug-in code is based off an HTTP service that listens and processes a HTTP POST request from RA CDE. The HTTP service can execute any type of HTTP request at the remote endpoint.

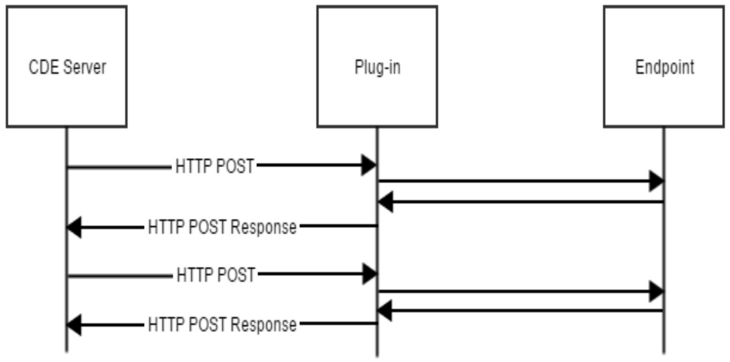
The setup of the plug-in is very flexible:

- You can use any programming language.
- Network API of the remote endpoint (HTTP, TCP, or other network protocol).

- Packaging.

All packaged plug-ins are JAVA based web applications that are package as Tomcat * WAR files, which can be deployed locally or remotely from the core product installation. You can also deploy plug-ins on a private or public cloud as needed. You could also host plug-ins using online services such as http://hook.io, especially for online or cloud based remote components.

The diagram below shows how the interface between RA CDE, plug-in, and remote component interact:



- The CDE Server makes an HTTP POST request to the plug-in when the invocation of a plug-in service occurs.
- The plug-in uses the information in the POST request to format the appropriate call to the remote component.
- The CDE Server makes repeated HTTP POST requests for the service until it receives a response.
- The plug-in sends an HTTP POST response with the service status to the CDE Server when the service is complete.

The RA CDE plug-in framework supports he following capabilities:

- Configuring endpoint connections to the remote component
- Creating automated tasks that instrument operations in the remote endpoint
- Importing application and environment models
- Importing content items [like defects, features, user stories]
- Performing connectivity tests to remote endpoints

The JSON scheme in the plug-in framework provides the following inputs:

externalTaskExecutionStatus

Provides a high-level status for the task. Set this input to one of the following values:

-
  - RUNNING
-
  - FAILED
-
  - FINISHED

executionContent

Provides a field in which to store the service execution content. The task execution context data might include a collection of key/value pairs of runtime information for this specific task execution. CA Release Automation Continuous Delivery Edition persistently stores this context as an opaque data object and delivers the object at the next invocation of this API for this specific task execution.

Type: Map **<**String,Sting>

Context Key Limit: 230 characters

Context Value Limit: 4000 characters

Notes:

-
  - If the response does not include the context element, CA Release Automation Continuous Delivery Edition keeps the context as is.
  - If the response includes an empty context element, CA Release Automation Continuous Delivery Edition clears the context for this task execution.

taskState

Provides a summary of the task status. This status appears on the UI next to the task status icon.
The status is returned at the **statusDescription** field of the Task REST API response and displayed on the UI.
Type: String

Limits: 255 characters

Example: Task Failed

detailedInfo

Provides a detailed summary of the task status. The status is returned at the **detailedStatusDescription** field of the Task REST API response. This detailed status appears when you click on the high-level status in the UI.

Type: String

Limits: 10,000 characters

Example: Endpoint value is incorrect. Provide valid endpoint details.

progress

Provides a progress indicator to display.

Type: Float

delayTillNextPoll

Specifies when CA Release Automation Continuous Delivery Edition initiates the next invocation of this API for this specific task execution (if at all).

Type: Long

Manifest Definition

The capabilities of the RA CDE custom plug-in are defined in a manifest JSON file, and the manifest file is broken down in the following sections:

- Basic plug-in information.
- Endpoint template information.
- Service information.
- Service parameter information.

Sample manifest.json file.

```
    "name": "Sample",
    "vendor": "CA Technologies",
    "uniqueId": "CA Technologies Sample",
    "description": "Sample Plug-in for CA Continuous Delivery Edition",
    "version": "1.0",
    "iconUrl": "CA.svg",
    "endpointTemplate": {
        "uniqueId": "Endpoint",
        "name": "Sample Endpoint",
        "description": "A Sample plug-in endpoint",
        "serviceType": "ENDPOINT",
        "endPointType": "Endpoint",
        "url": "Endpoint",
        "parameters": [{
            "uniqueId": "URL",
            "name": "URL",
            "displayName": "URL",
            "type": "string",
            "isOptional": false,
            "defaultValue": null,
            "description": "URL"
        }, {
            "uniqueId": "username",
            "name": "username",
            "displayName": "Username",
            "type": "string",
            "isOptional": false,
            "defaultValue": null,
            "description": "Username"
        }, {
            "uniqueId": "password",
            "name": "password",
            "displayName": "Password",
            "type": "password",
            "isOptional": false,
            "defaultValue": null,
            "description": "Password"
        }]
    },
    "services": [{
            "uniqueId": "Task1",
            "name": "Task1",
            "description": "Description of the first task",
            "serviceType": "TASK",
            "url": "tasks/task1",
            "parameters": [{
                "uniqueId": "parameter1",
                "name": "parameter1",
                "displayName": "Parameter1",
                "type": "string",
                "isOptional": false,
                "defaultValue": null,
                "description": "Description of the first parameter of the first task",
                "url": "tasks/task1/parameter1/values"
            }, {
                "uniqueId": "parameter2",
                "name": "parameter2",
                "displayName": "Parameter2",
                "type": "string",
                "isOptional": false,
                "defaultValue": "GET",
                "description": "Description of the second parameter of the first task",
                "possibleValues": [
                    "GET",
                    "POST",
                    "PUT",
                    "DELETE",
                    "PATCH",
                    "HEAD"
                ]
            }, {
                "uniqueId": "parameter3",
                "name": "parameter3",
                "displayName": "Parameter3",
                "type": "textarea",
                "isOptional": true,
                "defaultValue": null,
                "description": "Description of the third parameter of the first task",
                "url": "tasks/task1/parameter3/values",
                "dependencies": [
                    "parameter1"
                ]
            }]
        }, {
            "uniqueId": "Import Application Model",
            "name": "Import Application Model",
            "description": "Description of import application model",
            "serviceType": "APPLICATION",
            "url": "application-sources/application-source",
            "parameters": null
        }, {
            "uniqueId": "Import Content Items",
            "name": "Import Content Items",
            "description": "Description of import content items",
            "serviceType": "CONTENT",
            "url": "content-sources/content-source/content-items",
            "parameters": [{
                "uniqueId": "parameter1",
                "name": "Parameter1",
                "displayName": "Parameter1",
                "type": "string",
                "isOptional": false,
                "defaultValue": null,
                "description": "Description of the first parameter of the first import content items"
            }]
        }, {
            "name": "Connectivity Test",
            "description": "Connection Test",
            "serviceType": "CONNECTIVITY_TEST",
            "url": "connectivity-tests/connectivity-test",
            "uniqueId": "Connectivity Test"
        }


    ]
}
```

WAR Specification

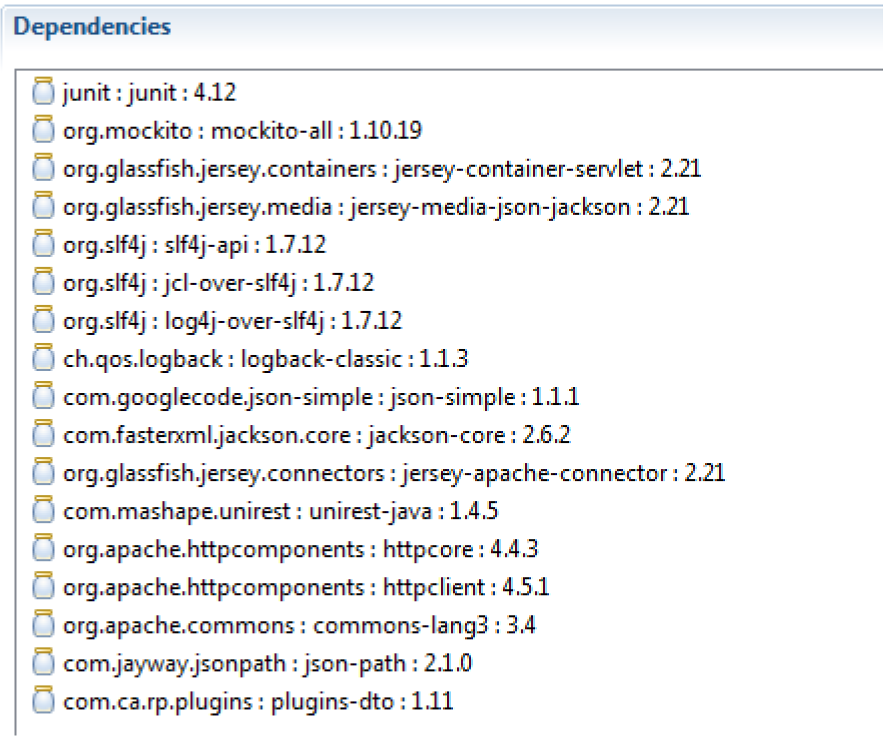
To provide flexibility in how a RA CDE plug-in is delivered, which can also be accomplished via Tomcat8 based JAVA WAR file. This WAR file will be created based off a JAVA dynamic web project for a Tomcat 8 installation.

You will use the methods defined at the beginning of this section, and you might need to pay close attention to the following condition. Where the plug-in might need to continue running, this is accomplished by setting the “delayTillNextPoll” to a given duration to poll again by the RA CDE plug-in.

The following are the pre-requisites needed to create this WAR file.

- JAVA JDK 1.8 (64-bit)
- Tomcat 8 (64-bit)
- Eclipse EE (64-bit)
- Maven 3.x (64-bit)
- Create a dynamic web project in Eclipse EE for Tomcat 8.0
- The following library is needed: “Plug-ins-dto-1.11-SNAPSHOT.jar”, which is part of the %TOMCAT_HOME%\webapps\cdd\lib

The following libraries need to be added via Maven dependencies:



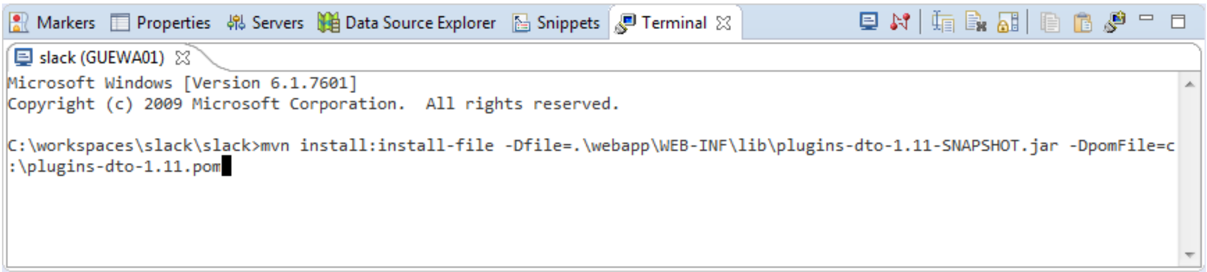
- junit : junit : 4.12
- org.mockito : mockito-all : 1.10.19
- org.glassfish.jersey.containers : jersey-container-servlet : 2.21
- org.glassfish.jersey.media : jersey-media-json-jackson : 2.21
- org.slf4j : slf4j-api : 1.7.12
- org.slf4j : jcl-over-slf4j : 1.7.12
- org.slf4j : log4j-over-slf4j : 1.7.12
- ch.qos.logback : logback-classic : 1.1.3
- com.googlecode.json-simple : json-simple : 1.1.1
- com.fasterxml.jackson.core : jackson-core : 2.6.2
- org.glassfish.jersey.connectors : jersey-apache-connector : 2.21
- com.mashape.unirest : unirest-java : 1.4.5
- org.apache.httpcomponents : httpcore : 4.4.3
- org.apache.httpcomponents : httpclient : 4.5.1
- org.apache.commons : commons-lang3 : 3.4
- com.jayway.jsonpath : json-path : 2.1.0
- com.ca.rp.plugins : plugins-dto : 1.11

To add the “Plug-ins-dto-1.11-SNAPSHOT.jar”, you need to do the following:

- Create the following POM file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ca.rp.plug-ins</groupId>
  <artifactId>plug-ins-dto</artifactId>
  <version>1.11</version>
</project>
```

Run the following Maven command as shown below:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\workspaces\slack\slack>mvn install:install-file -Dfile=.\webapp\WEB-INF\lib\plugins-dto-1.11-SNAPSHOT.jar -DpomFile=c
:\plugins-dto-1.11.pom
```

Best Practices

The following best practices will help you in creating a Release Automation CDE plug-in.

Planning

As in any development project, it is important that you plan how the RA CDE plug-in will interface and interact with the targeted tool, as well as availability of the plug-in's resources.

Portable Manifest

Keep the content of the manifest portable and agnostic to the exact location of the server container. A portable manifest lets the administrator relocate and deploy the plug-in on any server. Avoid hard coding specific server addresses and ports inside the plug-in manifest. The plug-in manifest should not be tightly coupled to its server container.

For example, all URL values in the sample plug-in manifest should use relative values such as tasks/task1.

Sensitive Information

Store all sensitive information, such as passwords, as endpoint parameters. Do not store sensitive information in task or content source parameters. The endpoint is a central location that plug-in tasks and content sources share. This best practice prevents the repeated entry of sensitive data. This practice also prevents the replication of sensitive data across release tasks and content sources.

Example: If a password value is updated for a remote component, you only have to update one location (endpoint details) in CA Release Automation Continuous Delivery Edition. This best practice also lets you implement different security levels for the endpoint creator and the task creator. The endpoint creator can hide the concrete password value from all task creators that use this endpoint.

Store the network connectivity details of remote components as endpoint parameters. Do not store network connectivity information in task or content source parameters. Use this best practice for central management and maintenance of all remote servers that plug-ins access over the system network.

When you use this best practice, the **ENDPOINTS** page lists all remote servers that the plug-ins access. You do not have to inspect all tasks to detect which network elements CA Release Automation Continuous Delivery Edition accesses.
Example: If an IP address of a remote component changes, you would only need to update one location (endpoint details) in CA Release Automation Continuous Delivery Edition.

Packaging

Ensure the manifest is part of the plug-in package. We do not recommend that you store the manifest separately from the rest of the plug-in.

Ensure the manifest resides at the root of the plug-in package.
If the base URL of the plug-in package is */cdd-slack-plug-in/* , the manifest resides directly under the root of the plug-in package: */cdd-slack-plug-in/manifest.json*

Naming and Versioning

Do not create multiple plug-ins with the same name.
Ensure that each unique ID is different. Using a distinct plug-in name helps to distinguish the plug-ins in the UI.

Ensure the unique ID of the plug-in is a string that includes the plug-in name and the vendor.
Example: CA Technologies Slack

After you have published the plug-in to the community, do not change the plug-in name and vendor in subsequent versions.

Update the plug-in version every time you modify and release the plug-in.

Tools

Prior to starting the development effort of the RA CDE plug-in, make sure that the targeted tool is accessible online, as well as making sure that the tool's server can interact with the GitHub implementation of Hook.io.

Using the proper version of the API calls

Some of the API calls can be used as version 1 (v1) or version 2 (v2), and there are other calls that will provide you with different results based on the version being utilized.

Using the correct parameter method

Given that some API calls use URL parameters and other calls are using JSON body parameters, it becomes very crucial that you become aware of the requirements for the API calls that you are going to be using. This is where the tools listed in this document can be of great help to you in determining how build the parameters require for a given API call.

Recommended Tools

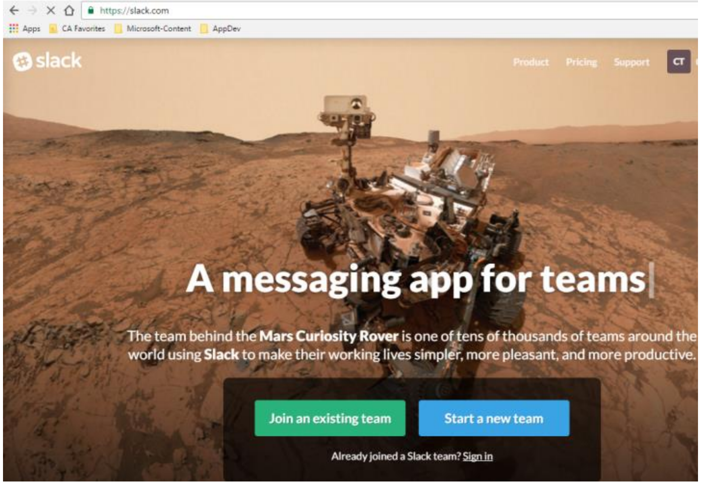
To fully test the different Slack incoming web hooks API calls, the following tools can be used:

- Postman by postman running as a standalone Google Chrome application or plug-in.
- REST Easy extension in FireFox.
- HttpRequester extension in FireFox.
- Curl, which is a command-line utility used extensively to test the different HTTP get/post calls.
- CA DevTest 9.x
- SOAPUI 5.2.x or greater.

RA CDE —SLACK Plug-in

As part of the process in how to create a RA CDE plug-in, we are going to utilize the SLACK messaging service.

Please visit the https://slack.com and sign up for the creation of a Slack team.

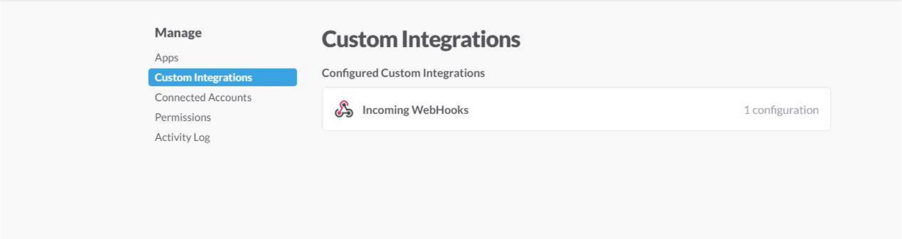


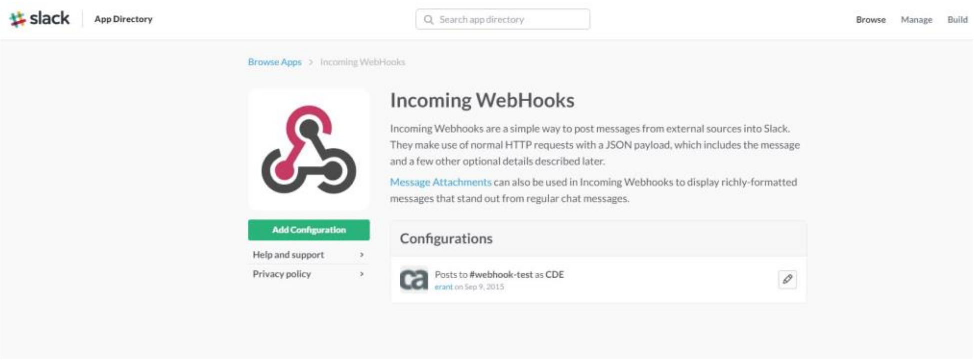
A messaging app for teams

The team behind the **Mars Curiosity Rover** is one of tens of thousands of teams around the world using **Slack** to make their working lives simpler, more pleasant, and more productive.

After you have created a Slack team, you will need to create a new incoming web hook integration, by going to the following link

https://[YOURS-SLACK-TEAM].slack.com/apps/manage/custom-integrations



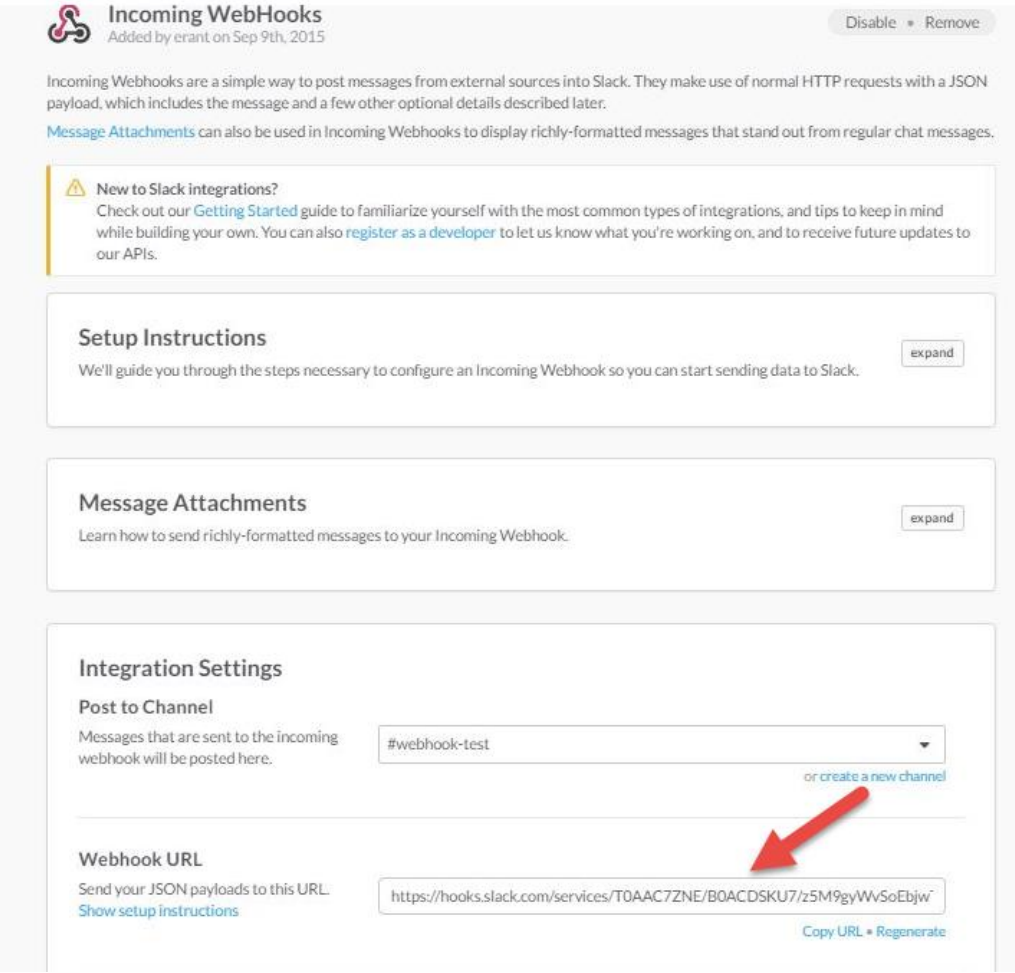


Incoming Webhooks are a simple way to post messages from external sources into Slack. They make use of normal HTTP requests with a JSON payload, which includes the message and a few other optional details described later.

Message Attachments can also be used in Incoming Webhooks to display richly-formatted messages that stand out from regular chat messages.

Posts to **#webhook-test** as **CDE**
erant on Sep 9, 2015

As per the figure below, you will create the desired web hook, which will default to a specific channel where to the messages will post, as well as the Webhook URL as pointed by the red arrow. This is the information that you will need to complete the process.



Incoming WebHooks

Added by erant on Sep 9th, 2015

Incoming Webhooks are a simple way to post messages from external sources into Slack. They make use of normal HTTP requests with a JSON payload, which includes the message and a few other optional details described later.

Message Attachments can also be used in Incoming Webhooks to display richly-formatted messages that stand out from regular chat messages.

New to Slack integrations?

Check out our Getting Started guide to familiarize yourself with the most common types of integrations, and tips to keep in mind while building your own. You can also register as a developer to let us know what you're working on, and to receive future updates to our APIs.

Setup Instructions

We'll guide you through the steps necessary to configure an Incoming Webhook so you can start sending data to Slack.

Message Attachments

Learn how to send richly-formatted messages to your Incoming Webhook.

Integration Settings

Post to Channel

Messages that are sent to the incoming webhook will be posted here.

#webhook-test

or create a new channel

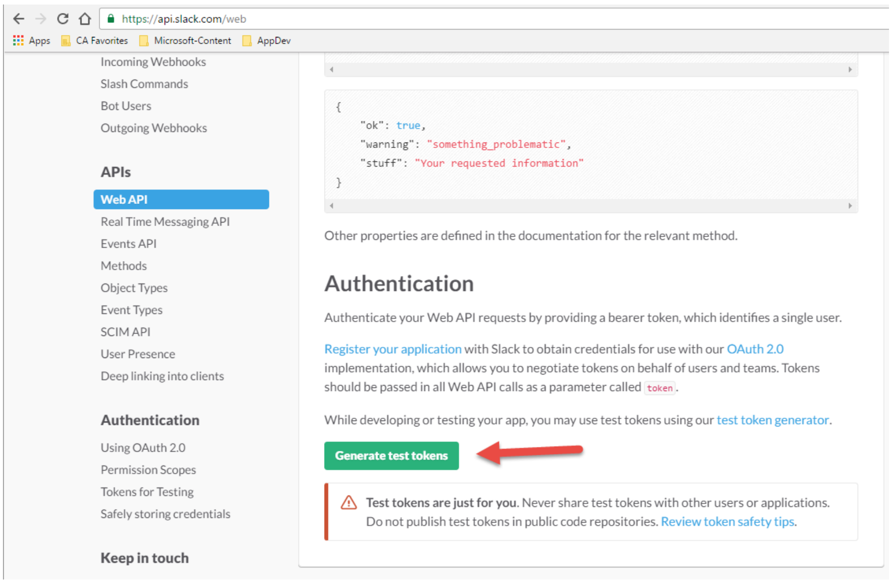
Webhook URL

Send your JSON payloads to this URL.

Show setup instructions

https://hooks.slack.com/services/T0AAC7ZNE/B0ACDSKU7/z5M9gyWvSoEbjw

Now you need to generate a test token as shown below:



```
{
    "ok": true,
    "warning": "something_problematic",
    "stuff": "Your requested information"
}
```

Other properties are defined in the documentation for the relevant method.

Authentication

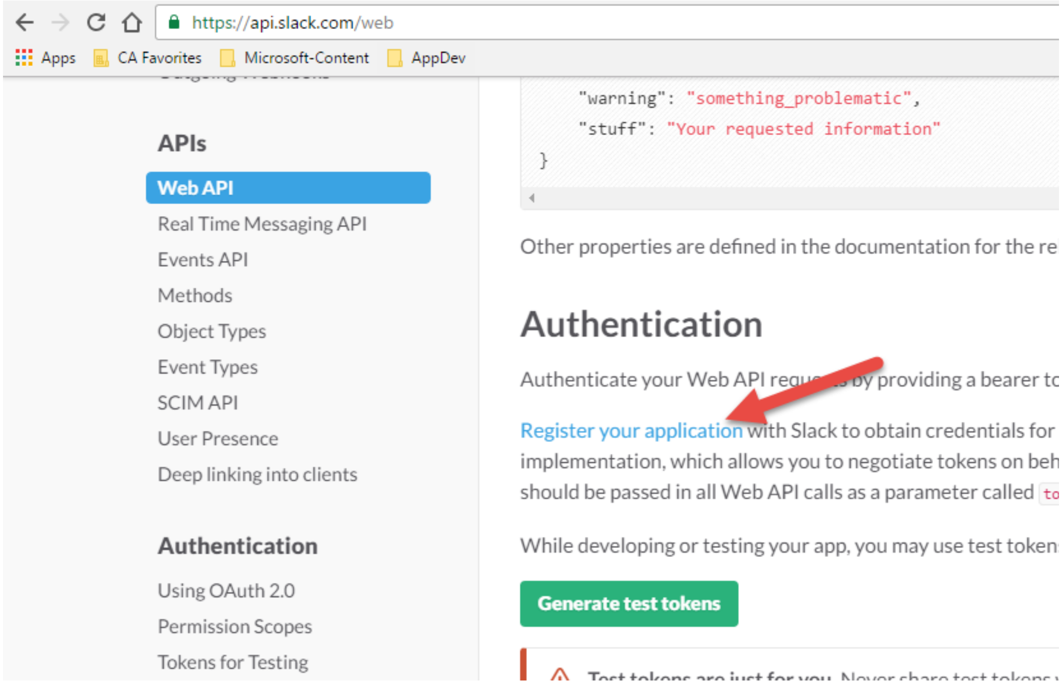
Authenticate your Web API requests by providing a bearer token, which identifies a single user.

Register your application with Slack to obtain credentials for use with our OAuth 2.0 implementation, which allows you to negotiate tokens on behalf of users and teams. Tokens should be passed in all Web API calls as a parameter called `token`.

While developing or testing your app, you may use test tokens using our test token generator.

⚠ **Test tokens are just for you.** Never share test tokens with other users or applications. Do not publish test tokens in public code repositories. Review token safety tips.

After the test token has been generated, you will need to make sure that the application is registered, by clicking on the "Register your application" link as shown below.



```
    "warning": "something_problematic",
    "stuff": "Your requested information"
}
```

Other properties are defined in the documentation for the re

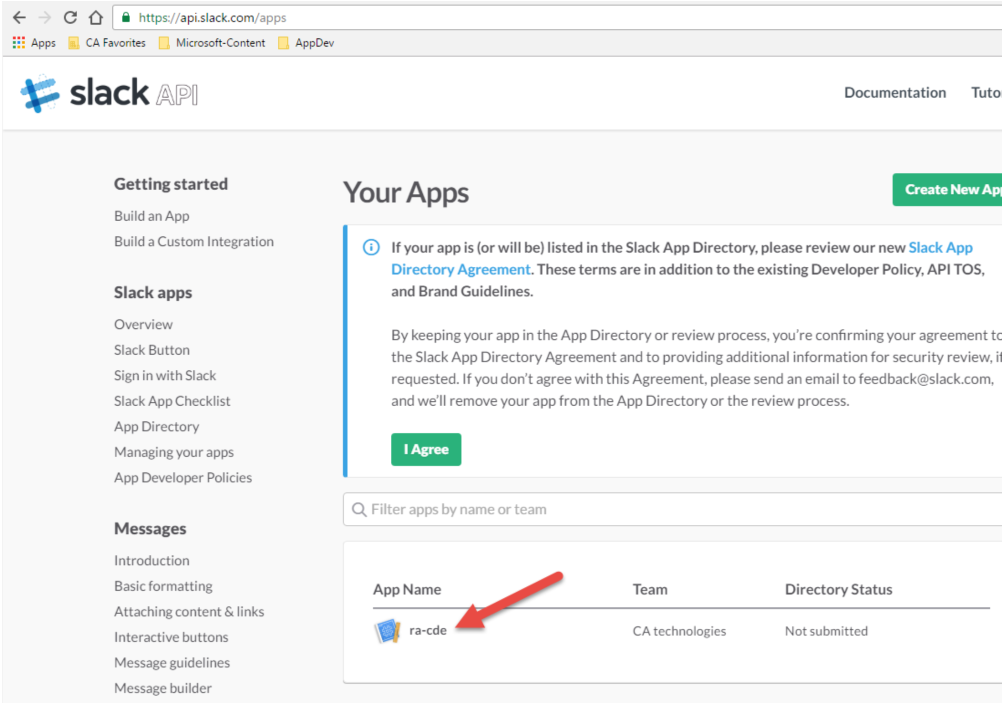
Authentication

Authenticate your Web API requests by providing a bearer tc

Register your application with Slack to obtain credentials for implementation, which allows you to negotiate tokens on beh should be passed in all Web API calls as a parameter called `to`

While developing or testing your app, you may use test token

Your application should be already listed, click on it, to see the client id and secret.



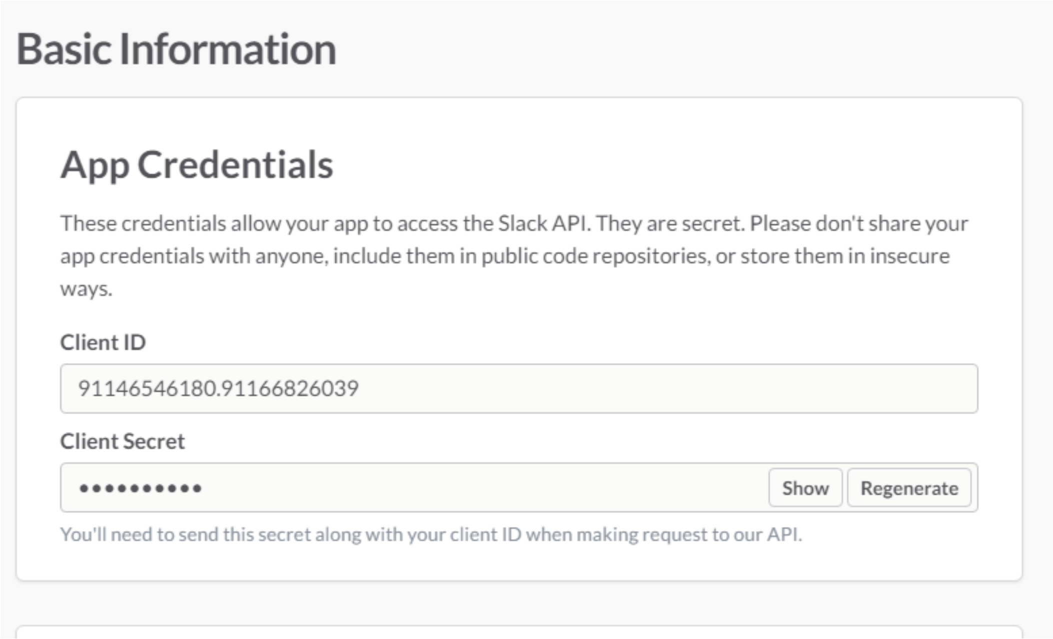
Your Apps

If your app is (or will be) listed in the Slack App Directory, please review our new Slack App Directory Agreement. These terms are in addition to the existing Developer Policy, API TOS, and Brand Guidelines.

By keeping your app in the App Directory or review process, you're confirming your agreement tc the Slack App Directory Agreement and to providing additional information for security review, if requested. If you don't agree with this Agreement, please send an email to feedback@slack.com, and we'll remove your app from the App Directory or the review process.

App Name	Team	Directory Status
ra-cde	CA technologies	Not submitted

Write down the client ID and display the secret as well. You are going to be utilizing this information for the setup of the Slack RA CDE plug-in.



Basic Information

App Credentials

These credentials allow your app to access the Slack API. They are secret. Please don't share your app credentials with anyone, include them in public code repositories, or store them in insecure ways.

Client ID

91146546180.91166826039

Client Secret

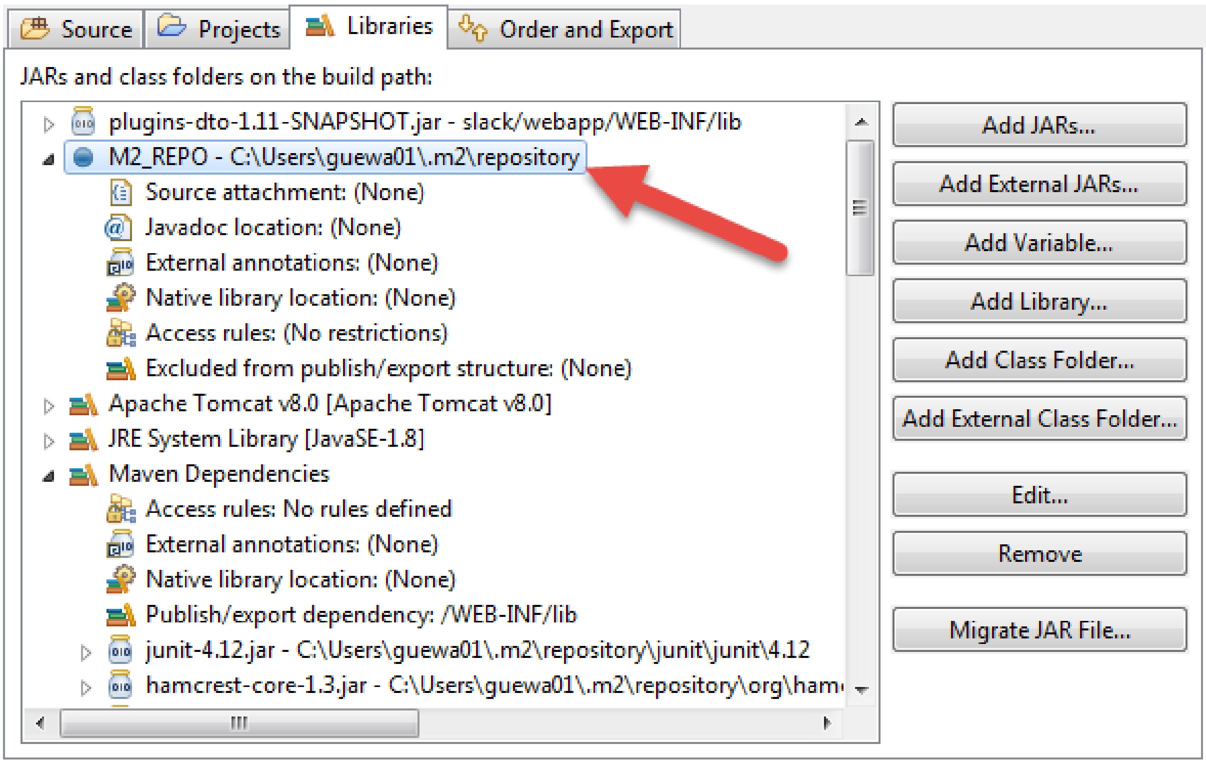
••••••••••

You'll need to send this secret along with your client ID when making request to our API.

Slack Eclipse Project

As part this document, we have included an Eclipse dynamic web project that utilizes Maven as is software project management for the additional libraries that will be needed.

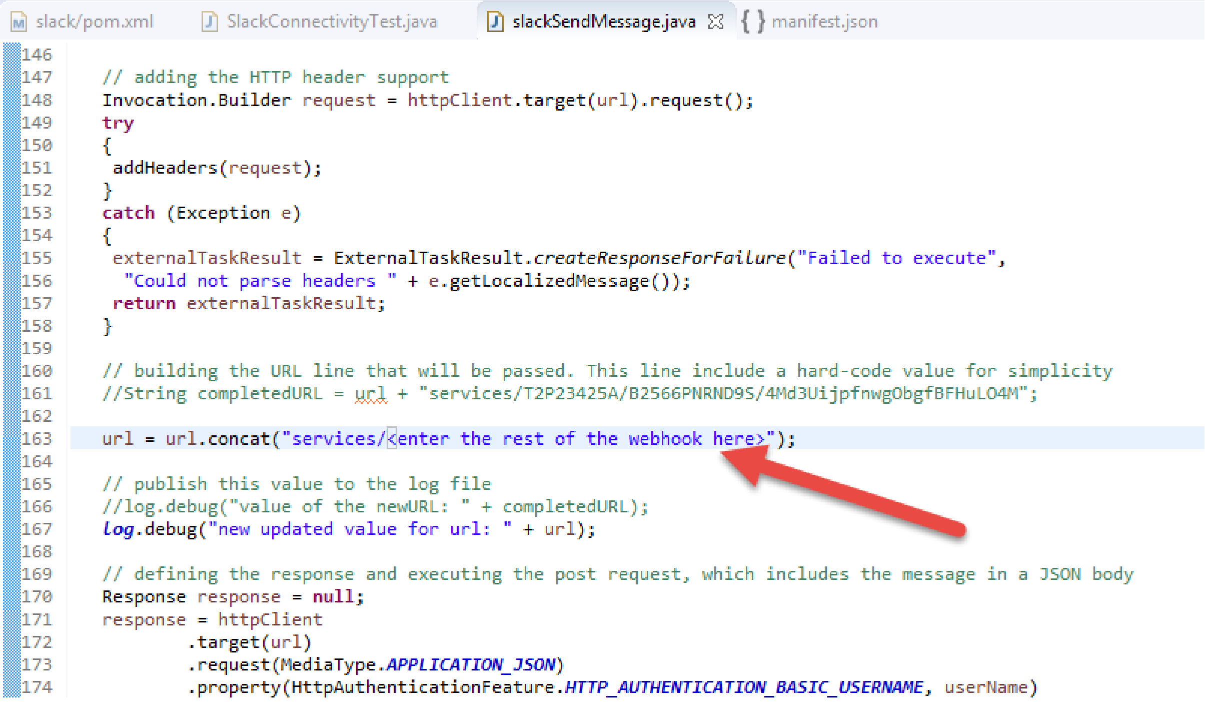
It is assumed that Maven has been installed and configured correctly. Expand the slack project that is part of this document, make sure that you fix the Maven repository path as shown below.



JARs and class folders on the build path:

- plugins-dto-1.11-SNAPSHOT.jar - slack/webapp/WEB-INF/lib
- M2_REPO - C:\Users\guewa01\.m2\repository
  - Source attachment: (None)
  - Javadoc location: (None)
  - External annotations: (None)
  - Native library location: (None)
  - Access rules: (No restrictions)
  - Excluded from publish/export structure: (None)
- Apache Tomcat v8.0 [Apache Tomcat v8.0]
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
  - Access rules: No rules defined
  - External annotations: (None)
  - Native library location: (None)
  - Publish/export dependency: /WEB-INF/lib
  - junit-4.12.jar - C:\Users\guewa01\.m2\repository\junit\junit\4.12
  - hamcrest-core-1.3.jar - C:\Users\guewa01\.m2\repository\org\ham

Now open the “slackSendMessage.java”, and enter the web hooks URL that you created before, please copy the values after the “services/”.



```

// adding the HTTP header support
Invocation.Builder request = httpClient.target(url).request();
try
{
 addHeaders(request);
}
catch (Exception e)
{
 externalTaskResult = ExternalTaskResult.createResponseForFailure("Failed to execute",
  "Could not parse headers " + e.getLocalizedMessage());
 return externalTaskResult;
}

// building the URL line that will be passed. This line include a hard-code value for simplicity
//String completedURL = url + "services/T2P23425A/B2566PNRND9S/4Md3UijpfnwgObgfBFHuLO4M";

url = url.concat("services/<enter the rest of the webhook here>");

// publish this value to the log file
//log.debug("value of the newURL: " + completedURL);
log.debug("new updated value for url: " + url);

// defining the response and executing the post request, which includes the message in a JSON body
Response response = null;
response = httpClient
        .target(url)
        .request(MediaType.APPLICATION_JSON)
        .property(HttpAuthenticationFeature.HTTP_AUTHENTICATION_BASIC_USERNAME, userName)
```

After you have accomplished the above steps, you can perform a build. Highlight the "slack" project name, select the "Run as → Maven clean", then perform a "Run as→Maven install".

The WAR file can be found in the ${project_loc}\slack\target\cdd-slack-plug-in-1.0.0.war.

Adding Plug-in to RA CDE

Now you will need to define the following in Release Automation CDE:

- Register the plug-in
- Add the Slack endpoint based off the registered plug-in
- Add a task to a given release's phase

The above steps are accomplished by following the steps below:

Register the plug-in with Release Automation CDE, in this scenario that plug-in will be Bamboo as described in the prior sections. To get to plug-ins, select Administration→Plug-ins in RA CDE.

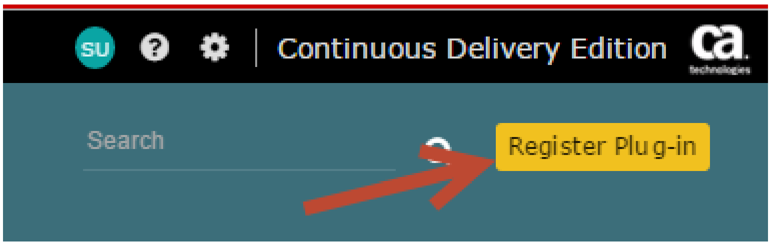
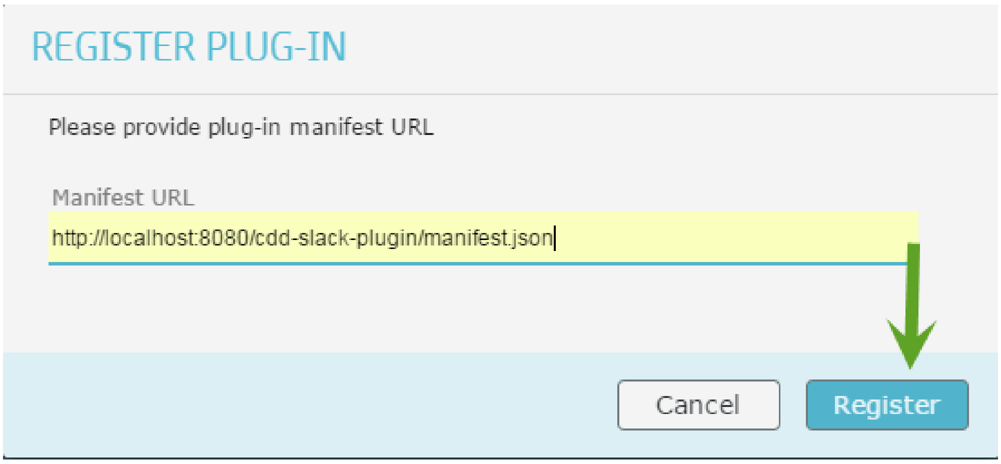


Figure 13: Registering the plug-in

Now we are going to register the Bamboo plug-in, you will take the complete URL for the manifest file that describes how the plug-in will interface with RA CDE, click the "Register" button to complete the plug-in registration.

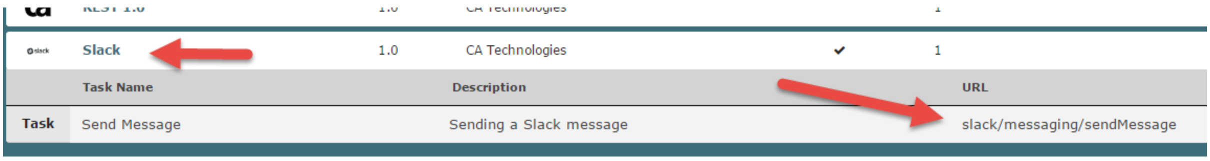


REGISTER PLUG-IN

Please provide plug-in manifest URL

Manifest URL: http://localhost:8080/cdd-slack-plugin/manifest.json

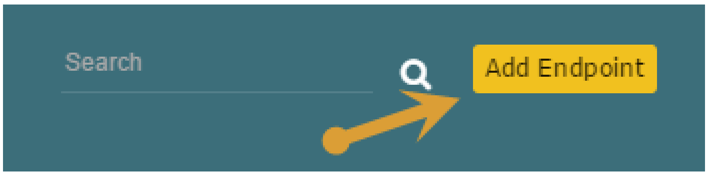
Once the Slack plug-in has registered successfully, you will see it listed as one of the available plug-ins and its available services.



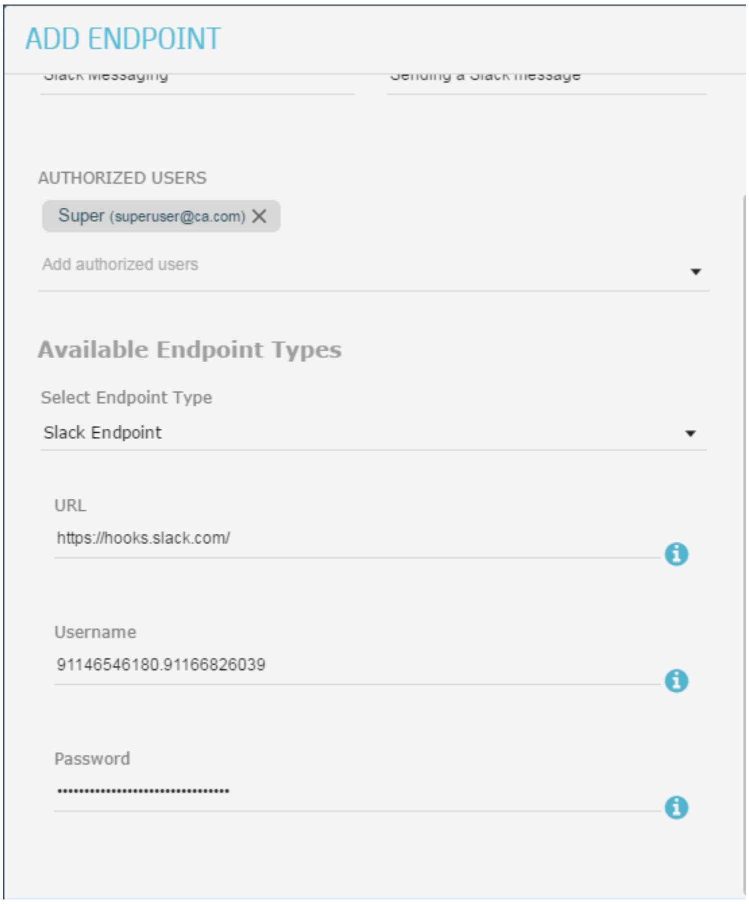
Slack	1.0	CA Technologies	✓	1

	Task Name	Description	URL
Task	Send Message	Sending a Slack message	slack/messaging/sendMessage

After the Slack plug-in has been registered, we move to the Administration→Endpoints to add an endpoint for the tasks as defined as part of a release's phases can be executed. Click the "Add Endpoint" to start the process.



Enter the necessary information as shown below, you will have to select the correct endpoint type. Click the "Add" button to complete adding the Slack endpoint.



ADD ENDPOINT

AUTHORIZED USERS

Super (superuser@ca.com)

Add authorized users

Available Endpoint Types

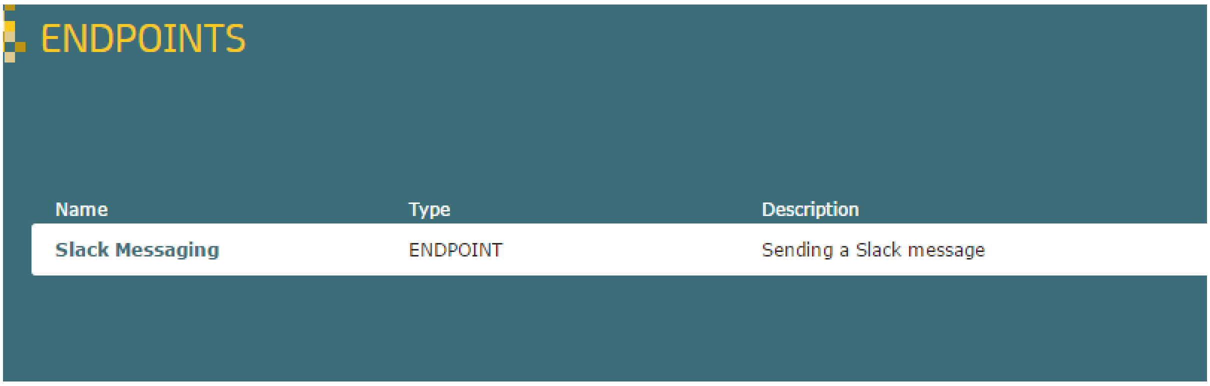
Select Endpoint Type: Slack Endpoint

URL: https://hooks.slack.com/

Username: 91146546180.91166826039

Password: ••••••••••••••••••••••••••••••••

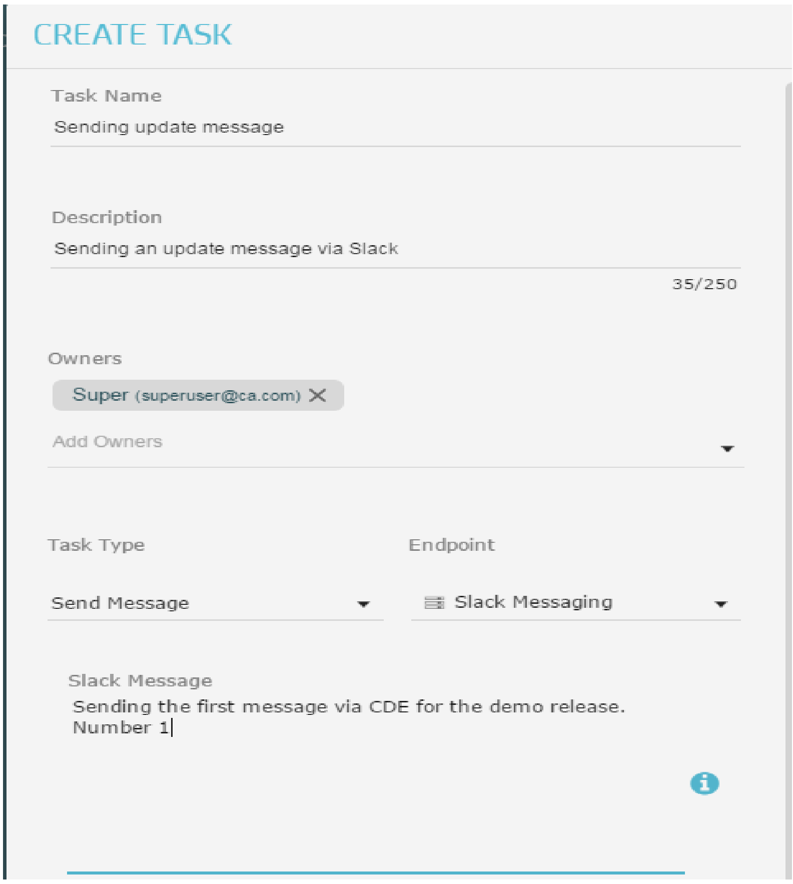
Please make sure that you enter the URL value as shown above, and use the client ID and secret that you wrote down in the Slack section. You will see the endpoint defined.



ENDPOINTS

Name	Type	Description
Slack Messaging	ENDPOINT	Sending a Slack message

Now we need to go to a release, which already has a phase defined and select "Create a task…" option in one of the phases and add entries as shown below, once you have entered the necessary information, click on the "Create" button.



CREATE TASK

Task Name: Sending update message

Description: Sending an update message via Slack

35/250

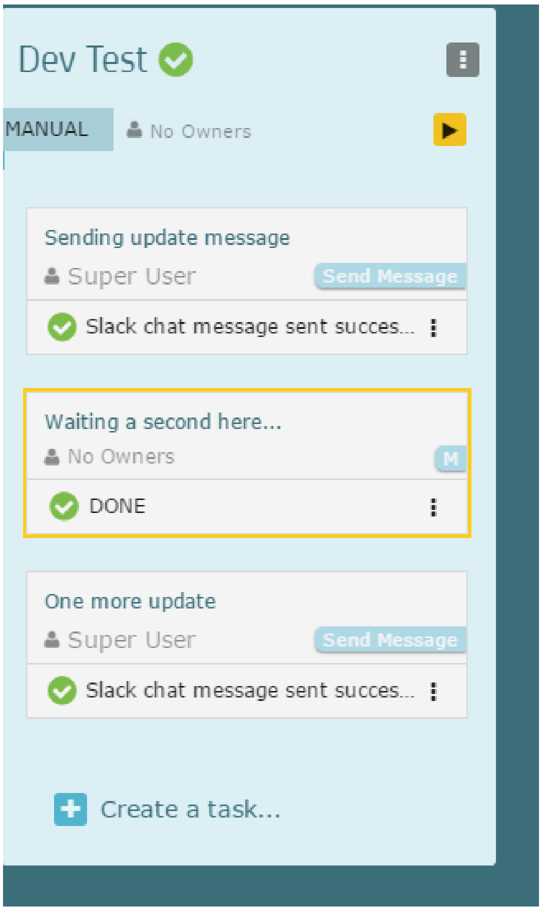
Owners: Super (superuser@ca.com)

Task Type: Send Message

Endpoint: Slack Messaging

Slack Message: Sending the first message via CDE for the demo release. Number 1

The newly created Slack task will be shown in the phase.



Copyright Notice

Useful Links

https://slack.com

https://api.slack.com

https://docops.ca.com/ca-release-automation-continuous-delivery-edition/6-2/en/integrations/develop-custom-plug-ins

https://docops.ca.com/ca-release-automation-continuous-delivery-edition/6-2/en/integrations/develop-custom-plug-ins/develop-custom-plug-in-http-service