slack api (https://api.slack.com/)

Recommended reading

Sending messages using incoming webhooks

Incoming webhooks are a way to post messages from apps into Slack. Creating an incoming webhook gives you a unique URL to which you send a JSON (https://en.wikipedia.org/wiki/JSON) payload with the message text and some options. You can use all the usual formatting (/reference/surfaces/formatting) and layout blocks (/surfaces/messages#complex_layouts) with incoming webhooks to make the messages stand out.

If you're looking for the Help Center article on using webhooks with Workflow Builder, head over here (https://slack.com/help/articles/360041352714). Otherwise, read on!

> Webhooks do not work with organization-ready apps. Use our other messaging tools, including `chat.postMessage`, instead.

Getting started with incoming webhooks (https://api.slack.com/messaging/webhooks#getting_started)

We're going to walk through a 4-step process (if you've already done some of these things it'll be even easier) that will have you posting messages using incoming webhooks in just a few minutes:

1. Create a Slack app (if you don't have one already) (https://api.slack.com/messaging/webhooks#create-app)

Create your Slack app (/apps/new)

Pick a name, choose a workspace to associate your app with (bear in mind you'll probably be posting lots of test messages, so you may want to create a channel for sandbox use), then click **Create App**. If you've already created an app, you can use that one. Have a cookie for being prepared! 🍪

You'll be redirected to the settings page for your new app (if you're using an existing app, you can load its settings via your app's management dashboard (/apps)).

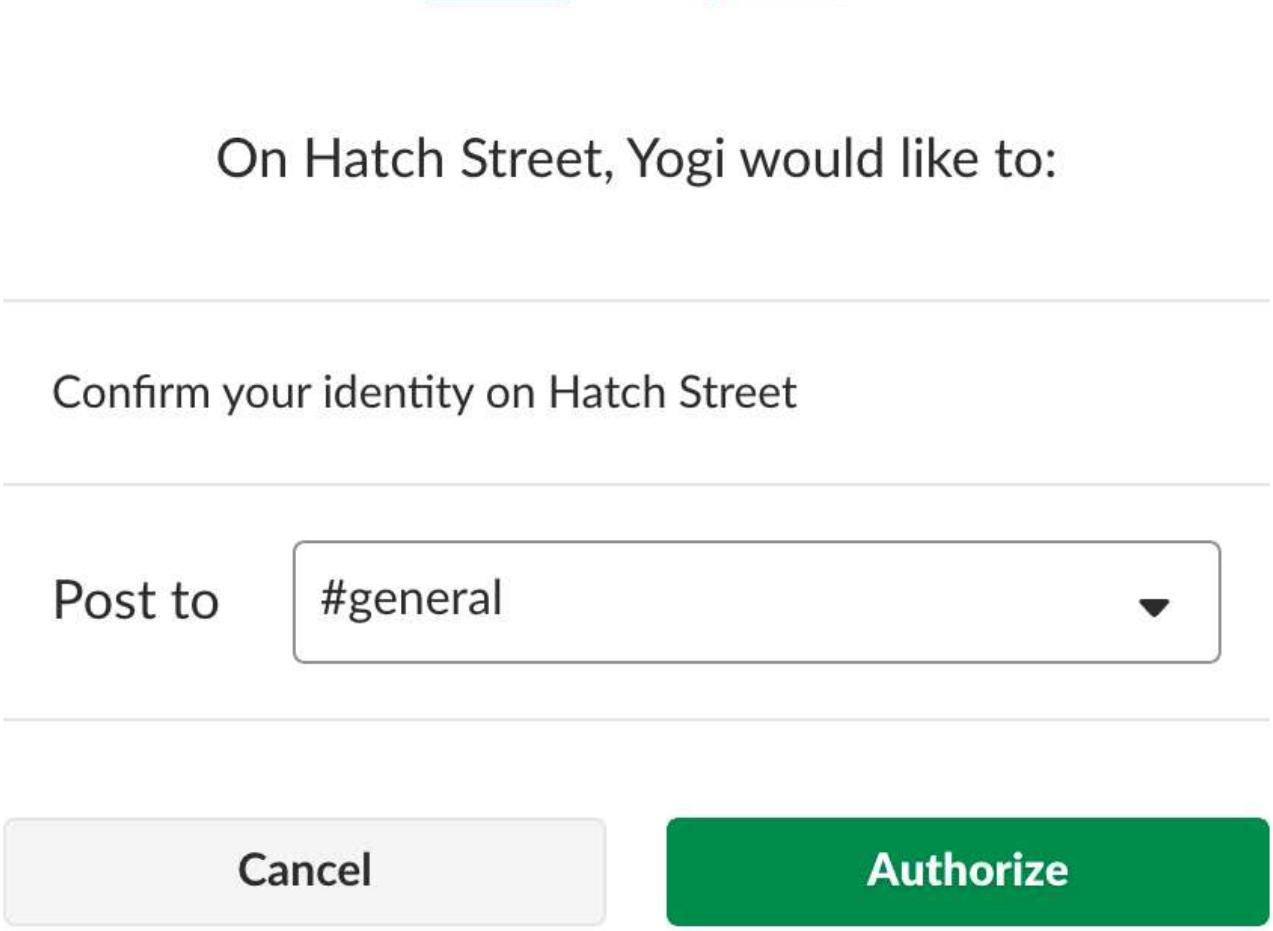
From here, select **Incoming Webhooks**, and toggle **Activate Incoming Webhooks** to on. If you already have this activated, well, you deserve another cookie! 🍪

3. Create an incoming webhook (https://api.slack.com/messaging/webhooks#create_a_webhook)

Now that incoming webhooks are enabled, the settings page should refresh and some additional options will appear. One of those options is a very helpful button called **Add New Webhook to Workspace** — click it!

What this button does is trigger a shortcut version of the installation flow for Slack apps, one that is completely self-contained so that you don't have to actually build any code to generate an incoming webhook URL. We'll show how you can generate webhooks programmatically later, but for now, you'll see something like the following:

slack api (https://api.slack.com/)



Go ahead and pick a channel that the app will post to, then select **Authorize**. If you need to add the incoming webhook to a private channel, you must first be in that channel.

You'll be sent back to your app settings, where you should see a new entry under the **Webhook URLs for Your Workspace** section. Your webhook URL will look something like this:

```
https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

That URL is your shiny new incoming webhook, one that's specific to a single user and a single channel.

> If you're developing a GovSlack (/gov-slack) app for use by public sector customers, make your API calls to the `slack-gov.com` domain instead of the `slack.com` domain.

slack api (https://api.slack.com/)

Keep it secret, keep it safe. Your webhook URL contains a secret. Don't share it online, including via public version control repositories. **Slack actively searches out and revokes leaked secrets.**

4. Use your incoming webhook URL to post a message (https://api.slack.com/messaging/webhooks#posting_with_webhooks)

Later in this doc we'll explain how to make your messages more expressive or interactive, but for right now anything will do, so we're going to use that old standby — "Hello, world".

Make an HTTP POST request like this:

```
POST https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
Content-type: application/json
{
    "text": "Hello, world."
}
```

The URL that you're making the POST request to should be the same URL you generated in the previous step.

That's it! Go and check the channel your app was installed into, and you'll see that the "Hello, World" message has been posted by your app.

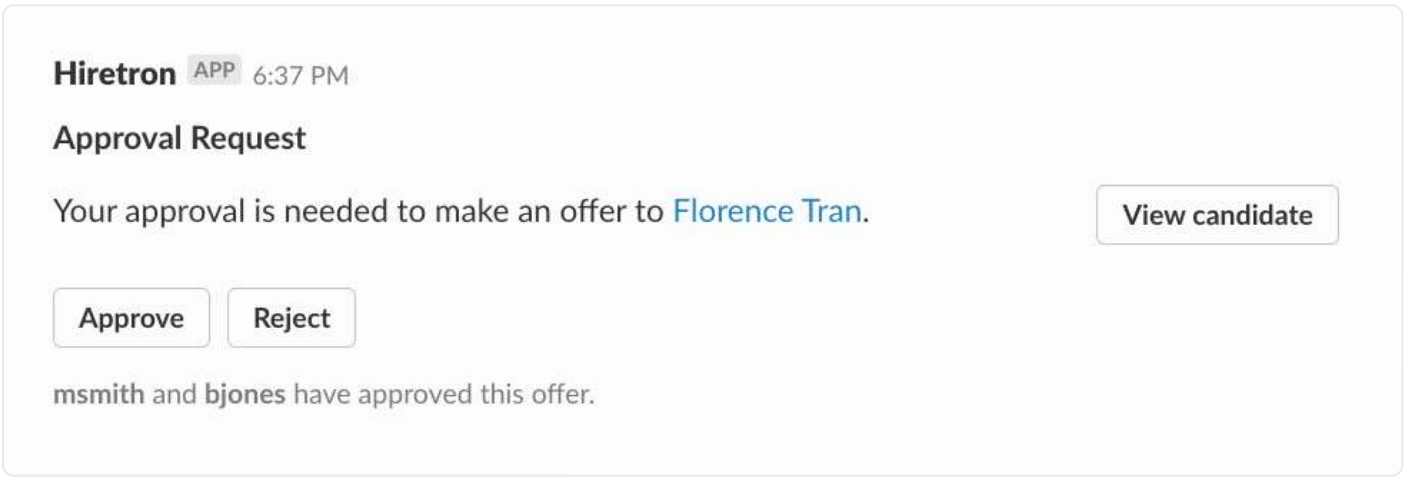
You can use this in a real Slack app without much change, just substitute your favorite HTTP Request library for cURL and structure all the requests in the exact same way. You'll also need to pay attention to some details we've outlined below when you're distributing your app.

Incoming webhooks do not allow you to delete a message after it's been posted. If you need a more complex chat flow including message deletion, call `chat.postMessage` (/methods/chat.postMessage).

Great work, you've set up incoming webhooks for your Slack app and made a successful test call, and you're ready to start making those messages more interesting and useful. We baked some extra cookies to celebrate! 🍪 🍪 🍪 🍪

~~om/messaging/webhooks#advanced_message_formatting)~~

Incoming webhooks conform to the same rules and functionality as any of our other messaging APIs. You can make your posted messages just a single line of text, or use interactive components (/messaging/interactivity).



The process of using all these extras and features is similar to the one explained above. The only difference is the JSON payload that you send to your webhook URL will contain other fields in addition to `text`. Here's a more advanced HTTP request example that you can use with the same webhook `url` that you used above:

```
POST https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
Content-type: application/json
{
    "text": "Danny Torrence left a 1 star review for your property.",
    "blocks": [
        {
                "type": "section",
                "text": {
                        "type": "mrkdwn",
                        "text": "Danny Torrence left the following review for your prop
                }
        },
        {
                "type": "section",
                "block_id": "section567",
                "text": {
                        "type": "mrkdwn",
                        "text": "<https://example.com|Overlook Hotel> \n :star: \n Door
                },
```

```
                        "alt_text": "Haunted hotel image"
                }
        },
        {
                "type": "section",
                "block_id": "section789",
                "fields": [
                        {
                                "type": "mrkdwn",
                                "text": "*Average Rating*\n1.0"
                        }
                ]
        }
    ]
}
```

This example uses Block Kit (/block-kit) visual components to make the message more expressive and useful.

We have some fantastic docs that explain how to use text formatting (/reference/surfaces/formatting) and Block Kit (/block-kit) to make your messages more interesting and interactive, so please dive into our overview of message composition (/surfaces/messages).

> You **cannot override** the default channel (chosen by the user who installed your app), username, or icon when you're using incoming webhooks to post messages. Instead, these values will always inherit from the associated Slack app configuration.

Posting your message as a reply in a thread (https://api.slack.com/messaging/webhooks#threads)

You can use an incoming webhook to make your message appear as a reply in a thread. You'll need to retrieve the message `ts` value, however, as it is not returned when sending a request to an incoming webhook. You'll use the `ts` value as the `thread_ts` field of the webhook request to generate the threaded reply.

You can retrieve the `ts` value in one of the following ways:

- Use the search.messages method to search for the message you sent based on the timestamp.
- Use the conversations.history method to search for the message from a set of messages.

Read about replying to messages (/messaging/sending#threading) and retrieving messages (/messaging/retrieving#individual_messages) for more details.

Generating incoming webhook URLs programmatically (https://api.slack.com/messaging/webhooks#incoming_webhooks_programmatic)

In the guide above, we demonstrated how to quickly generate a webhook URL through your app settings UI; however, when you're distributing your app (for use by non-collaborators), you'll need a way for it to generate those URLs on the fly.

Fortunately, incoming webhooks can be easily generated during the standard OAuth install flow (/docs/oauth#flow).

If you're going to distribute your app (/distribution#multi_workspace_apps), it's likely you're already planning to use the OAuth process anyway. Below we'll cover the adjustments you'll need to make to that process to enable incoming webhooks.

1. Change your scopes (https://api.slack.com/messaging/webhooks#adjust_scopes)

As part of the install process, your app defines a set of initial permission scopes (/docs/oauth-scopes) to request from a user. Whether you're using the Slack button (/docs/slack-button#button-widget) to provide a link for users to install your app or your own custom OAuth redirect (/docs/oauth#step_1_-_sending_users_to_authorize_and_or_install), there will be a `scope` parameter that sets this initial list of permissions.

To generate incoming webhook URLs, make sure you include the `incoming-webhook` permission (/scopes/incoming-webhook) in that `scope` list. When you do, users will see an additional permission on the Authorize screen that allows them to pick the channel where incoming webhooks will post to, as shown above.

Once a user installs your app and your app has completed the OAuth verification code exchange (/docs/oauth#step_3_-_exchanging_a_verification_code_for_an_access_token), you'll receive a JSON response like this:

```
{
    "ok": true,
    "access_token": "xoxp-XXXXXXXX-XXXXXXXX-XXXXX",
    "scope": "identify,bot,commands,incoming-webhook,chat:write:bot",
    "user_id": "XXXXXXXX",
    "team_name": "Your Workspace Name",
    "team_id": "XXXXXXXX",
    "incoming_webhook": {
        "channel": "#channel-it-will-post-to",
        "channel_id": "C05002EAE",
        "configuration_url": "https://workspacename.slack.com/services/BXXXXX",
        "url": "https://hooks.slack.com/TXXXXX/BXXXXX/XXXXXXXXXX"
    }
}
```

You can see that this OAuth response contains an `incoming_webhook` object, and right there in the `url` field is your brand new incoming webhook URL. You can now go ahead and use this URL to post a message, as demonstrated above. Here's a full explanation of all the fields in this `incoming_webhook` object:

Attribute	Type	Description
`channel`	String	The name of the channel the user selected as a destination for messages
`channel_id`	String	The ID of the same channel
`configuration_url`	String	A link to the page that configures your app within the workspace it was installed
`url`	String	The incoming webhook URL

Though in most cases you'll receive a "HTTP 200" response with a plain text `ok` indicating that your message posted successfully, it's best to prepare for scenarios where attempts to publish a message will fail.

Incoming webhooks may throw errors when receiving malformed requests, when utilized webhook URLs are no longer valid, or when something truly exceptional prevents your messages from making it through to channels and users.

Incoming webhooks return more expressive errors than our Web API, including more relevant HTTP status codes (like "HTTP 400 Bad Request", "HTTP 403 Forbidden", and "HTTP 404 Not Found"). These are described in our changelog: Changes to errors for incoming webhooks (/changelog/2016-05-17-changes-to-errors-for-incoming-webhooks).

Errors you may encounter include:

- `action_prohibited` usually means that an admin has placed some kind of restriction on this avenue of posting messages and that, at least for now, the request should not be attempted again.
- `channel_is_archived` indicates the specified channel has been archived and is no longer accepting new messages.
- `invalid_payload` typically indicates that received request is malformed — perhaps the JSON is structured incorrectly, or the message text is not properly escaped. The request should not be retried without correction.
- `invalid_token` means the token used was expired, invalid, or missing.
- `no_active_hooks` means the incoming webhook is disabled.
- `no_service` means the incoming webhook is either disabled, removed, or invalid.
- `no_service_id` means the `service_id` (`B00000000` in our examples above) was either invalid or missing.
- `no_team` means the Slack workspace was either missing or invalid.
- `no_text` means the `text` attribute was missing from the payload. Refer to the messages page for valid formatting details.
- `posting_to_general_channel_denied` is thrown when an incoming webhook attempts to post to the "#general" channel for a workspace where posting to that channel is 1) restricted and 2) the creator of the same incoming webhook is not authorized to post there.
- `team_disabled` means the Slack workspace is no longer active.
- `too_many_attachments` is thrown when an incoming webhook attempts to post a message with greater than 100 attachments. A message can have a maximum of 100 attachments associated with it.
- `user_not_found` and `channel_not_found` indicate that the user or channel being addressed either do not exist or are invalid. The request should not be retried without modification or until the indicated user or channel is set up.

slack api (https://api.slack.com/)

g/webhooks#workflows)

Build a workflow: Create a workflow that starts outside of Slack (https://slack.com/help/articles/360041352714-Build-a-workflow--Create-a-workflow-that-starts-outside-of-Slack)

Connect, simplify, and automate. Discover the power of apps and tools.

With webhooks (https://api.slack.com/messaging/webhooks#incoming_webhooks_programmatic), you can start a workflow (https://slack.com/help/articles/360035692513-Guide-to-Slack-Workflow-Builder) from outside of Slack, and use custom variables to post messages to Slack containing information from external services.

See also

- Removing Friction Between Feedback and the Customer (https://woobot.io/weve-got-somethin-ya/)
- Simple Kicker Status and Reservation System With Slack Integration
- Sending Connection Notifications to Slack from Google Compute Engine (https://cloud.google.com/community/tutorials/send-connect-notification-to-slack-from-google-compute-engine)
- Building Your First Serverless Slack App on GCP is so Easy (https://levelup.gitconnected.com/building-your-first-serverless-slack-app-on-gcp-is-so-easy-bb812bad5bd1)

slack api (https://api.slack.com/)

Cookie Preferences

Subscribe to our developer changelog (/changelog)

Connect with our platform community (https://slackcommunity.com/?utm_medium=referral&utm_source=apislack&utm_campaign=api_site_footer)

Need help? Contact developer support (/support)

How would you rate this page?