O'REILLY®

Slack Hacks

Tips & Tools for Team Collaboration

Edited by Troy Mott

Slack Hacks

Tips & Tools for Team Collaboration

John Adams, Guillaume Binet, Benoit Corne, Mike Street, and Aviv Laufer

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Slack Hacks
by Troy Mott

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Dawn Schanafelt
Production Editor: Melanie Yarbrough
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER

Interior Designers: Ron Bilodeau and Edie Freedman
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2016: First Edition

Revision History for the First Edition
YYYY-MM-DD: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491965054* for release details.

978-1-491-96505-4

[FILL IN]

Table of Contents

Preface

We're surrounded with a myriad of communication tools: email, Skype, Twitter, Facebook, and Snapchat, to name a few. Slack is a newer communication tool, borrowing IRC (Internet Relay Chat) concepts and transcending them with an intuitive interface and many other features that you will learn about in this book. Within this competitive environment, Slack has managed to attract nearly 3 million daily users in just three years, including some big companies like Dow Jones, Samsung, and The Wall Street Journal. One of the reasons behind Slack's exponential growth is easy customization; from organizing public and private channels, to searching and archiving messages, and even integrating bots and applications to boost productivity. These customizations have made Slack a true challenger to emails for team communication.

Who is this book for?

The hacks in this book are for developers, administrators, and Slack power users. You don't need to be a computer scientist to understand the hacks we're presenting in this book, but since most of them involving coding, having some programming skills is helpful.

What do you need to know prior to reading?

Familiarity with Slack's basic features is mandatory. We don't go through each feature in detail, since we assume that you know them. We implement the hacks using various programming languages and frameworks. If you are familiar with one language (C, C++, C#,

Haskell, Clojure, Groovy, PHP, etc.) you should be able to adapt the hacks to whichever language you prefer - Slack offers a great number of frameworks to communicate with the Slack API.

The code for the hacks in this book is available on https://github.com/slack-hacks.

What will this book provide?

This book starts with a brief introduction to Slack. It provides tips and hacks to use Slack efficiently, adapting it to your company and your community. The book then discusses Slack apps and integrations, covering both IFTTT and Zapier. Going deeper into Slack customization, the book shows how you can build your own applications, including team integrations and extensions, and Slack bots using different tools and frameworks.

Acknowledgements

Guillaume Binet: I wish to thank my wonderful wife Kuniko for being this incredible supporter for all those years and my visionary father Olivier who started it all by buying us a family computer in the early 80s.

Benoit Corne: Firstly, I would like to thank my wife and my children for their patience: they've helped me a lot by giving me some spare time to work on this book. I would like also to thank my coworkers at Ullink who encouraged me in this challenge, and especially Loic and Gregoire with whom I had worked with on my first Slack hack almost three years ago.

Mike Street: I would like to thank my wife for the support she gave me while I spent time on the book and for putting up with me talking about it all day every day. I would also like to thank my collegues at Liquid Light for thier encouragement in pushing me to be the best developer, author and productivity hipster I can be.

Technical Reviewers

Martin Ek is an avid follower of all things technology, with a special interest in web development and open source solutions. He is currently working on a Slack bot for a Norwegian startup.

Tony van Riet is the creator of lunchbot, a Slack bot that manages group lunch activity. Lunchbot keeps track of who buys lunch each day along with the cost of everybody's lunch, and maintains a running balance for each team member.

Magnus Skaalsveen is a skilled developer with extensive experience from a variety of techniques and tools for application and web development. He has worked extensively with front-end development, where he can work with everything from interaction design to implementation, particularly with HTML5, JavaScript and CSS3.

1

Introduction

Slack is much more than IRC on steroids. It contains emojis, gravatars, statistics and a powerful search engine. It is a platform where you can take advantage of public and private channels, which provide instant communication for effectively engaging with all types of groups and teams. You can also implement to-do lists and use notifications to help prioritize levels of communication so you can work smarter.

Slack is a great tool for increasing team communication and transparency in your business or organization. The real power of Slack comes from the ways in which you can extend it. Integrations in Slack allow you to add-in most any type of app or feature, whether that is Google Drive, Github or your favorite CRM platform. In order to get the most out of Slack, you need to learn about Webhooks, which allow you to push notifications and actions in and out of Slack.

To extend Slack even further, you can tap into the Slack APIs with almost any programming language and even create your own Slack bots that you can then use to carry out a myriad of tasks, such as having a bot get sandwich orders from your team or schedule a meeting that fits with everyone's schedules.

Once you adopt Slack and begin using it in these types of ways, you will discover how much more effective it is for you to work, communicate and even have fun via Slack. The hacks in this book will

not only provide many Slack solutions, they will also give you ideas for extending these solutions or creating your own.

Source code for the book

The source code for all hacks in this chapter can be found in this book's Github repo.

Let's start with the simplest way to add interesting features created by third parties: Slack Apps.

2

Slack Apps and Simple Webhooks

Apps are a great way to extend Slack to improve your workflow, monitor external services, and coordinate your team's efforts across devices, locations, and functional groups.

One such app is Google Calendar, which allows you to get reminders before an event starts, get summaries of the current day's and week's events, and get updates when an event is changed.

The real power of Slack comes from its ability to become your communication center. Slack apps can be bi-directional, not just receiving notifications from other services, but also sending commands to other services. With Slack you can also have a bot listen to your conversations and act according to this content.

You don't have to be a programmer in order to activate apps. Once you add them, you will wonder how you ever managed without them. Slack has ready-made apps with most of today's popular online services. The setup process for each app is a bit different, but once you learn how to integrate one service, the rest are easy.

Applications vs. integrations

A Slack App can be found at App Directory page, is packaged from a third party, and has a User Interface (UI).

An integration, on the other hand, is comprised of Custom Webhooks, bots and slash commands. In other words, it is a piece of software that integrates with Slack and can be installed from the Custom Integrations page.

Webhooks are a great way of creating your own simple integrations without having to integrate with the Slack API. (We cover the Slack API in Chapter 4.) Slack provides both Incoming and Outgoing Webhooks, so let's examine them both.

Incoming Webhooks

Incoming Webhooks let you push notifications into Slack from any service or application. Using Incoming Webhooks is a great way to make Slack act as a central spot where you can be notified of events coming from external systems.

For example, adding Incoming Webhooks to the beginning or end of a script allows a progress, success, or error report to be posted directly to Slack, so you don't have to constantly check manually.

Incoming Webhook scripts don't need to be hosted on a publicly accessible server or computer, but they do need to be hosted somewhere that has access to the Internet in order to post to Slack.

Outgoing Webhooks

Outgoing Webhooks allow you to trigger a script or action from within Slack using a predefined keyword or phrase.

These are handy if you want to allow anyone on your team (regardless of whether they have any coding experience) to initiate a process or script without having to leave Slack. For example, you can trigger a code to deploy or have a script that can reboot a server.

Unlike Incoming Webhooks, an Outgoing Webhook requires the process or script you want triggered to be accessible to Slack's servers. That's because there are several parameters passed in a payload when the Webhook is fired, which includes a security token for authentication.

The hacks in this chapter provide examples of some Slack apps. In addition to some simple Slack Apps, one hack requires you to create a new slash command, while another hack shows how to install a bot into your team.

> Slash commands are shortcuts for sending an HTTP request to a server. The user just types /_command name* and some arguments, and Slack returns the answer from the server.

Let's get started with some simple Apps.

> **Source code for the book**
>
> The source code for all hacks in this chapter can be found in this book's Github repo.

HACK 01 Keep track of Documents with Google Drive

> This app allows you to import Google Drive files by pasting a file's URL into Slack.

Go to the Google Drive app page and click the install button, then follow the on-screen instructions.

After you've done that, when a link is pasted into Slack, the message containing the link will be changed into a share message that points to the external reference. Doing this means:

- Other team members see the document's title.
- You can easily spot links to Google drive documents within the conversation (see Figure 2-1).
- The file will be added to Slack and the document's full text will be indexed for search (Slack will keep the preview of the document up-to-date if it gets updated on Google Drive).

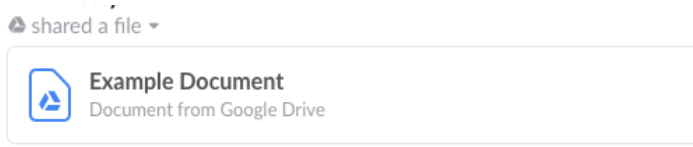


Figure 2-1. Google Drive Import

HACK 02 Add the Dropbox App

> The Dropbox app provides you with the same capabilities for Dropbox as the Google Drive app described above.

To install this app, go to the Dropbox app page and click the install button, and then follow the on-screen instructions.

From now on, when you paste a link to a Dropbox file in Slack, you will have a preview of that file, as shown in Figure 2-2.

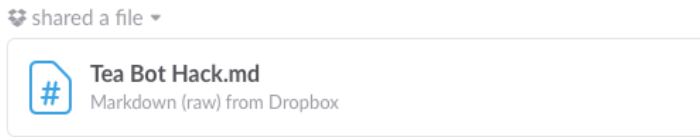


Figure 2-2. Dropbox Import

HACK 03 Easily Have Video Chats with Google Hangouts

> The Google Hangouts app allows you to enter the `/hangout` slash command in any channel. You will be given a link from which you can start the Hangout with a handy Slack control panel. From that panel, you can invite other Slack team members to the Hangout. To add this app, go to the Google Hangouts app page and click the install button, then follow the on-screen instructions.

This app is extremely useful if you have remote co-workers in your team and use Google Hangouts to have face-to-face conversations.

Being able to set up a Hangout and share the link without leaving Slack means it's easy to set up a quick call (see Figure 2-3).

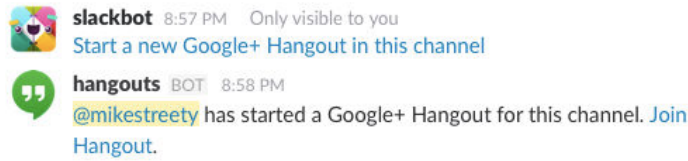


Figure 2-3. Google Hangouts

HACK 04 Start Skype Calls

If you prefer Skype over Google Hangouts, there is a Skype app available. Navigate to Skype app page and click the "Visit site to install" link. Click the "Add to Slack" button and select a channel. Now you can type `/skype` to start a video call (see Figure 2-4).

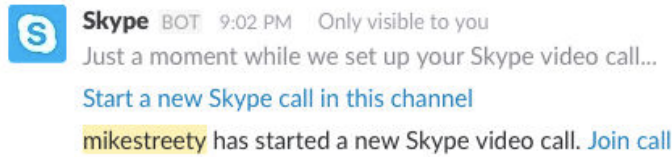


Figure 2-4. Skype

HACK 05 Add an Email App

This app works a bit different from the previous ones. It assigns an email address for the channel you choose. Visit the Email app page to install it. On the setup page that appears, choose the channel that you want the emails to be posted to. On the next page you will get an email address. All email sent to this address will appear in the selected channel (see Figure 2-5).

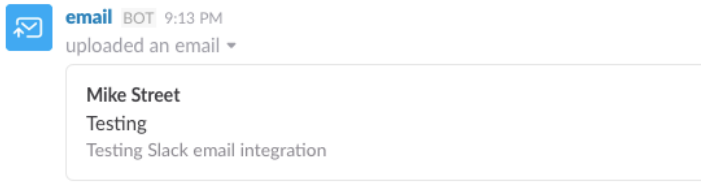


Figure 2-5. Email app

This app is great for subscribing to email newsletters because everyone in the channel can see the email without each user needing to subscribe.

HACK 06 Keep up-to-date with Github Activity

> This app posts activity on GitHub issues to a channel in Slack. Navigate to the GitHub app page and choose which channel the events will appear in, and which repositories to report on.

From now on you will have a channel with a stream of all the changes in your repositories so anyone can get a quick overview of what's going on. This app posts messages about commits, pull requests, comments, and issues. The messages include links to these events so you can click on them and go directly to Github. The GitHub channel can serve as the pulse for your team's development progress (see Figure 2-6).

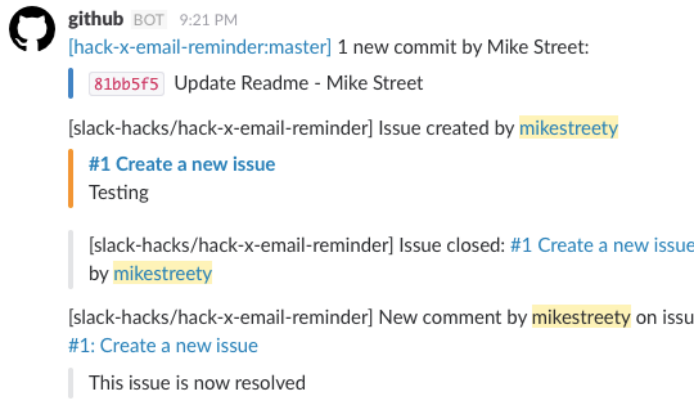


Figure 2-6. Github

HACK 07 Increase productivity with Kyber

> Kyber adds several productivity tools to Slack with one app. It provides project management tools, a team calendar, to-do lists and reminders, and other apps including IFTTT (If This Then That; see Chapter 3 for more info).

Navigate to Kyber Slack page and click the *Add to Slack* button, and then follow the on-screen instructions.

Once you've installed the app, go to the `#general` channel on Slack and type `/kyber` to get started. You will be presented with list of common Kyber commands (see Figure 2-7).

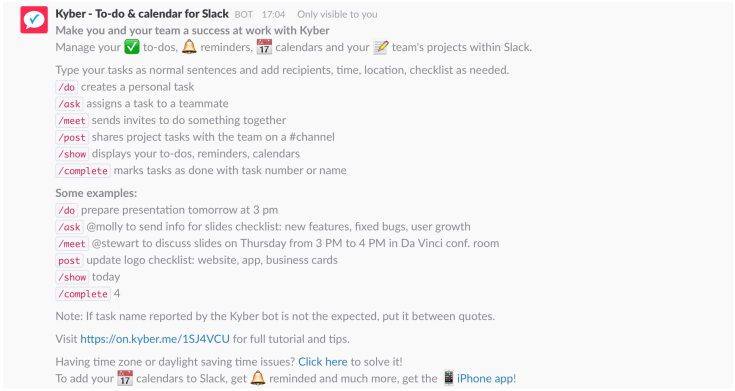


Figure 2-7. Common Kyber commands

In addition to the commands Kyber lists for you, the app comes with IFTTT integration, and here are just some of those actions that you can tap into:

- When a Github issue is assigned to you, add a to-do that specifies the repository, issue and link (recipe here).
- Add a to-do when a Trello card is assigned to you (recipe here).
- Log your work hours on your #calendar of choice (recipe here).
- Create a to-do with Amazon Alexa and add it to Slack (recipe here).

Chapter 3 covers IFTTT.

HACK 08 Send Quick Gifs with the Gif Keyboard

Gifs are the de-facto self expression tool on the Internet (a picture is worth a thousands words). The Gif Keyboard app will take your use of Gifs to the next level. It makes it easy to search a massive Gif collection, and you can also:

- Create custom GIFs by typing /gif "YOUR CAPTION HERE"
- Choose which GIF you want to send by typing /gifs

- Add personal GIFs of your company and coworkers

Navigate to the Gif Keyboard app page and click *Visit site to install* button. Then click the *Add to Slack* button, and follow the on-screen instructions.

Now you can select a Gif from a few options and not settle for the one that the app chooses for you. For example, you can issue `/gifs cat` and pick the one you wish to share from the various choices displayed (see Figure 2-8).

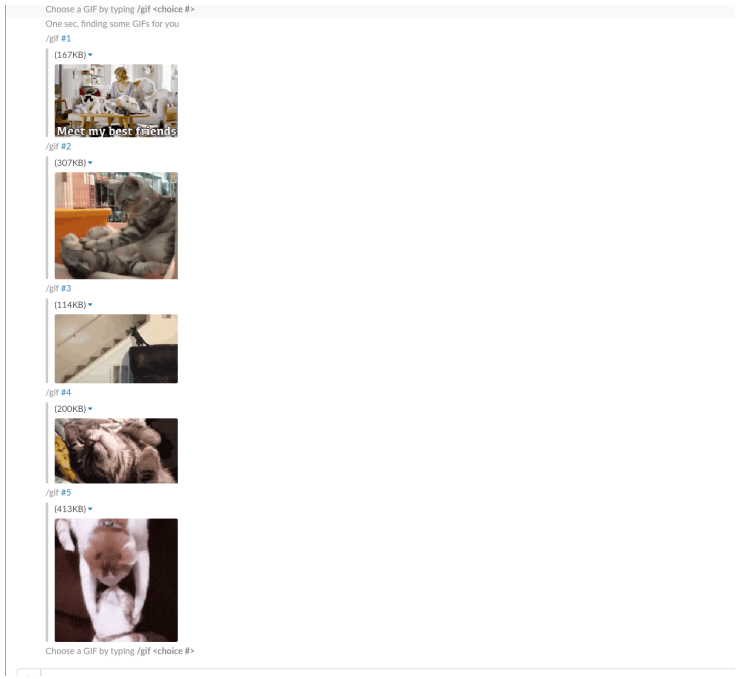


Figure 2-8. Cat Gifs

These choices are only visible to you, not your team members. To pick one, just type `/gif 2`, for example, and Gif number 2 will be selected and posted on the channel.

You can also add captions to your Gifs. For instance, Figure 2-9 shows the kind of thing you might see if you typed: `/gif cat "schrodinger's cat"`

Figure 2-9. Add Gif captions

You can combine the above and use `/gifs cat "schrodinger's cat"` to select an image and put a caption on it.

You can even save a Gif for a later by typing: `/gif save [alias]`

HACK 09 Make Scheduling Easy with Meekan

> Everybody hates scheduling meetings with multiple participants—finding a time when everyone is available is tough. The Meekan bot comes to your rescue. The bot scans the participant's calendars and comes up with one or more options that you can vote on.

To install it, navigate to the Meekan Scheduling Assistant page and click the *Visit site to install* button. On the next page click the *Install it now button* and then select the *Add to Slack* button. Finally, on the next page, click *Authorize*.

In Slack, you can now invite Meekan to a channel by typing `/invite meekan`. The next step is to connect your calendar to Meekan by typing `meekan connect me` or connect all the channel members' calendars by issuing `meekan connect @channel`. The bot will send you a direct message with a link for connecting your calendar, and after authorizing this you will be good to go. You can now enjoy the power of the bot. For example:

- Issue `meekan we want to have lunch next week`, and the robot will check everyone's calendar for a free lunch spot next week, and reply with a list of possible options.
- Set up a meeting with specific people (as opposed to everyone on the channel) by typing `meekan schedule lunch with @john and @amy`
- Ask Meekan to find an available room for your meeting: `meekan schedule a meeting next week, and please add a room`

This should give you a good idea of the types of useful apps that you can add to Slack.

HACK 10 Search for flights, car, or hotels

Kayak.com lets you find the best prices on flights, hotel rooms, and rental cars. They have developed a Slack app that lets you do this research from within Slack using a slash command along with natural language.

Installation

To add Kayak to your Slack team, head to the Kayak Slack App page and click the **Add to Slack** button. This takes you to an authorization page, which (once complete) will add a new *kayak* slash command that lets you perform travel-related queries.

Usage

One cool feature of the Kyak app is that you can write your queries in natural language. For example if you're looking for flights from San Francisco to Chicago on July 26th, just type this command:

```
/kayak flights from san francisco to chicago on July 26th
```

It will also work if you type this:

```
/kayak flights from san francisco to chicago on the 26th of July
```

You can use a relative time, like tomorrow, or next Friday:

```
/kayak flights from san francisco to chicago tomorrow
```

```
/kayak flights from san francisco to chicago next friday
```

You can even use a duration. If you have to plan a one-week trip, it's as easy as this:

```
/kayak flights from san francisco to chicago next friday for one week
```

You can replace `for one week` by `for 7 days` or `for 7 days`, too.

If you want to be very specific, you can:

```
/kayak direct flights in business class from san francisco to chicago
on the day after July 26th for three weeks for two adults and one child
```

The same kind of requests can be done for rental cars or hotels:

```
/kayak cars in miami tomorrow for the day
```

```
/kayak hotels around las vegas on next saturday for the weekend
```

For every request, a summary of four answers (with prices, schedules, and ratings) will be displayed with a link to Kayak's website to see the details, as shown in Figure 2-10.

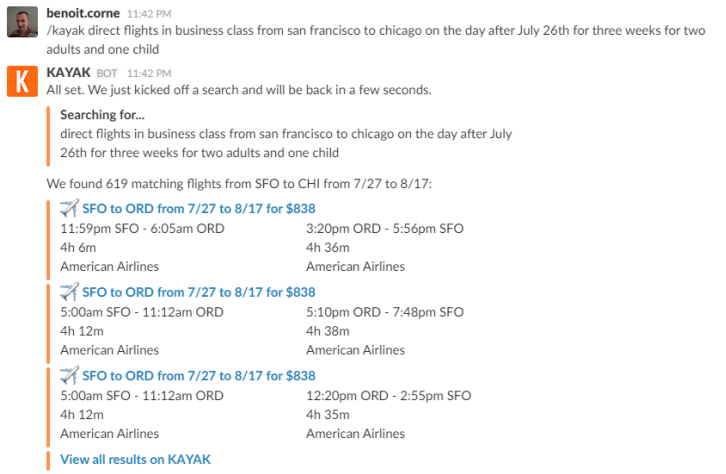


Figure 2-10. Example of kayak query

What if you don't want to share your results? The Kayak app's default behavior is to show its results to everyone in the channel where you run it. So if you wish to do a private search (or not bother everyone else), then just run this app in your own direct message channel. Why query in Slack? You may wonder why you should bother using Slack to do such queries? There are several good reasons: You can star the answers to retrieve them quickly You can

share them with other users You don't need to open a browser for the query—the summary gives you an idea of what to expect You won't see any ads in Slack By integrating Kayak with your Slack channel, you can easily have all sorts of travel information returned right in Slack.

HACK 11 Add Lytespark videoconferencing

> Slack is designed to be an asynchronous chat system, but you will often find yourself chatting with one or more users at once, and such conversations can take some time. What if you could switch from this mode to a videoconference in which you simply speak to chat and where you can share your screen?

Lytespark is a web based videoconferencing tool that has all of these features, and it can be integrated to Slack.

Head to the Lytepark Slack page and click "Add to Slack". Simply authorize the app, and then you can start uisng it.

The Lytespark extension adds two components:

- A slash command (/lytespark) for triggering a video chat (see Figure 2-11).

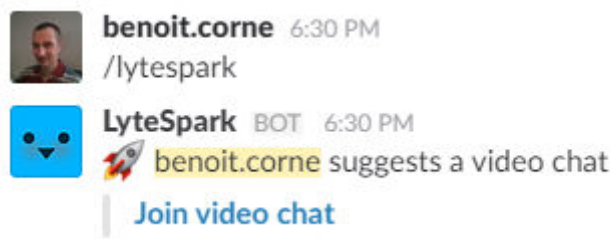


Figure 2-11. Lytespark slash command result

- A new bot that suggests when it might be a good time to open a video chat. To achieve this, it reads the channel activity and when there's a lot of discussion in a short amount of time the

bot gives a link to continue the discussion using video chat (see Figure 2-12).

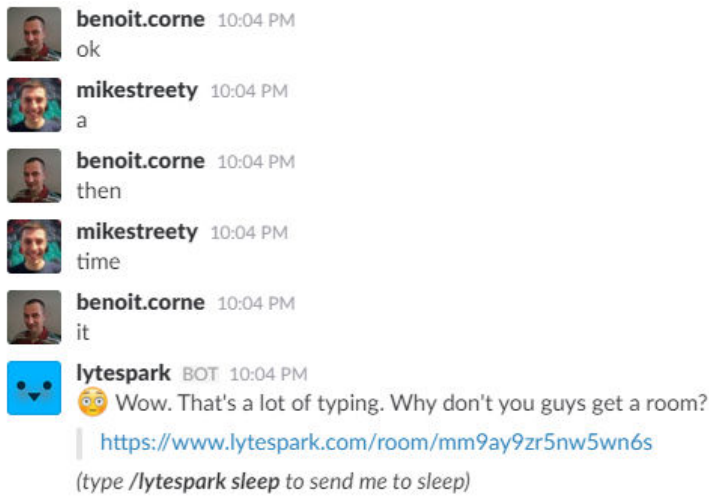


Figure 2-12. Lytespark bot triggering

Clicking the link opens a web page asking you for a name (no need to have an account) to enter the room (see Figure 2-13).

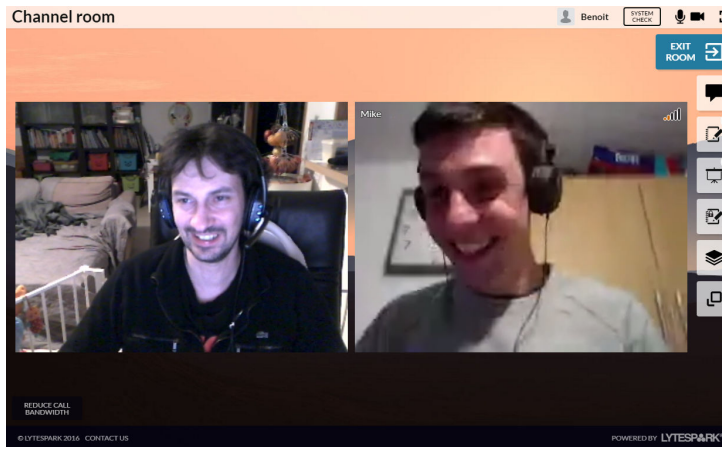


Figure 2-13. Lytespark chat room

Then you can resume your discussion, using the video chat. Screen sharing is available using a Google Chrome extension, plus you can take notes and even draw on a virtual board.

HACK 12 Add Stack Overflow search

StackOverflow is a programmer's best friend. By adding the following slash command you can access Stack Overflow from the comfort of your favorite Slack channel:

1. Click on the channel selector window (upper left corner in Slack).
2. Click **Configure Apps**.
3. Select the **Make a Custom Integration** button.
4. Click the **Slash Commands** button, and then enter these settings (the rest are default settings):
   - Command: `/overflow`
   - URL: `http://so.goel.io/overflow`
   - Method: `POST`
5. For the **Autocomplete help text**, turn on the "Show this command in the autocomplete list" checkbox. This will enable the following two options (type in the description and hint):
   - Description: `A programmer's best friend, now in Slack.`
   - Usage hint: `[search terms]`
6. Now, add the label:
   - Descriptive Label: `Search StackOverflow`
7. The final step is to select the **Save Integration** button.

The StackOverflow integration is now complete. Click in any channel and type `/overflow` with a space followed by whatever search term you are looking for on StackOverflow, such as `/overflow slack integration`. The top five results (if there are at least that many) will be returned in the channel for you to click on.

HACK 13 Have a Bot host your Standup meetings

> The Standup bot will let you run a virtual standup meeting within your Slack team by asking each team member what they did yesterday, what they plan to do tomorrow, and if anything is blocking them from carrying out their work.

The Standup bot will run the meeting. Once the standup is complete, a report will be generated and, if desired, emailed to team members. This bot comes in two flavors:

- A hosted paid version that can be found at the Standup bot page.
- An open source version that you can host yourself.

If you would like to run the open source version, there are detailed instructions about how to deploy it to Heroku on the project's Github page.

HACK 14 Send a Simple Message to Slack Using Incoming Webhooks

> In this hack you will learn how to post to a Slack channel using a PHP script. In the first part of this hack, we will post a simple, hard-coded message to a Slack channel whenever a script is called. In the second part, we will post a formatted message every time a Git commit is done on your repository.

Unlike Outgoing Webhooks, Incoming Webhooks do not need to be hosted externally - they just require an Internet connection. This allows you to host them on a local server (with your Git repository) and communicate with locally installed packages and sites.

Add Incoming Webhooks

The first step is to add Incoming Webhooks to your Slack team. To get started, navigate to the Incoming Webhooks App page and click install next to your team.

You will then be asked to select a channel to post to. (This setting can be overridden when sending the message.) The selection you make here determines the default channel if one isn't specified in your code. Pick one and click **Add Incoming Webhook**.

You will then be presented with a Webhook URL. Take note of this URL, since you will need it later. This URL is private and should not be posted anywhere public (as it allows anyone to post to any channel). If the URL becomes compromised, you can generate a new URL at the bottom of this page.

You can also customize the default icon and the name of your Incoming Webhook bot. (Like the default channel you selected, this can be overridden when posting the text.)

Run a PHP script

To begin, create a PHP file on a server that has Internet access. This can be on your computer if you have a local server running, or on a remote server accessed via the web.

To ensure that you have the file in an executable place, create a simple `Hello World` example and then trigger the file.

```
<?php

    echo 'Hello World';
```

We're running this through the command line, so to run the script, the command is:

```
$ php file.php
```

Post the simple message

Having a script return to you isn't of much help, since you want to post to Slack. Using cURL, you can `POST` to the Incoming Webhook URL to send a message. Replace `[WEBHOOK URL]` in the example below with the URL you were given when you set up the Webhook:

```
<?php

    // Your webhook URL
    $webhook = '[WEBHOOK URL]';

    // Encode the data
    $data = 'payload=' . json_encode(array(
        'text' => 'Hello World'
```

```
));

// PHP cURL POST request
$ch = curl_init($webhook);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);

echo $result;
```

Slack expects the data to be in `json` format, encoded with the key `payload`. This is what gets created at the beginning of the script. The `text` key is what is output into Slack. The last part of the script is the convoluted PHP cURL request required to `POST` data from a PHP script.

Wherever you run this script, it should return an `ok` if everything went well. (If it didn't, check your Webhook URL and make sure that your message isn't malformed.) In Slack, you should see your message posted to your designated channel (see Figure 2-14).

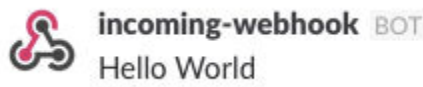


Figure 2-14. Incoming Webhook Success

Customize the Bot Name and Avatar

The next step in creating your Incoming Webhook bot is to customize its look and name (and change the channel if required). All customizations are passed into the payload array, parallel to the `text` key:

- `name` - The name needs to be plain text (emojis can be used but must be inserted directly in the script, rather than the `:cactus:` style that you can use inside Slack).
- `channel` - This is the channel the hook posts to. This can either be a public channel with the `#` (such as `#general`) or a private message with a user by using an `@` (such as `@user`).

- `icon_url` - Using the `icon_url` you can specify your bot's avatar, which needs to be a URL to a web hosted image. You can also use an `icon_emoji` instead.
- `icon_emoji` - This takes a standard emoji code to use as the icon for your messages, such as `:rocket:`. (If both the `icon_url` and `icon_emoji` are specified, the emoji takes precedence.)

Update the payload data to the following:

```
// Encode the data
$data = 'payload=' . json_encode(array(
    'username' => 'Incoming Bot',
    'channel' => '#general',
    'icon_emoji' => ':rocket:',
    'text' => 'A customised message'
));
```

This produces (in the `#general` channel) the following, as shown in Figure 2-15).

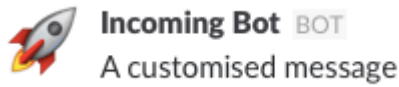


Figure 2-15. Customized Incoming Webhook Success

Add Message Attachments and Formatting

Slack Incoming Webhooks have the ability to have custom styled messages, featuring various data formatted with a bold title and a value. Called **Attachments**, these styled messages allow you to pass in several bits of information in one message while keeping the message readable.

You can have as many attachments as you wish (see Figure 2-16), but consider readability and your team's sanity before you go too overboard.

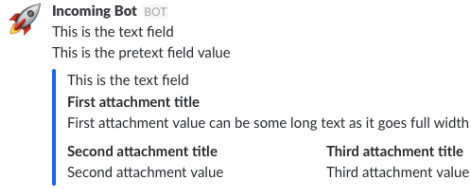


Figure 2-16. Webhook Attachments

Along with attachments, you can also pass additional message texts to precede the extra data. There is also a `fallback` value required for Slack clients that don't support attachments (like APIs).

To make attachments work, there are several extra fields you can pass in. Most of them are optional:

- `fallback` - This text should be a summary of the attachments and is used in the event that client doesn't support attachments. This is **required** but will not show in the default Slack clients.
- `pretext` - This text appears before the attachment. This is not required, but allows you to give an introduction to the extra data.
- `text` - This text appears *inside* the attachment (colored bar) but above any of the titled sections.
- `color` - This color appears on the left hand side of the attachment section of the message. This is useful if you want to add some visual context to your messages (such as a red border if something goes wrong, or green if something goes right).
- `fields` - This is the meat of the Webhook and contains the extra data. It is a multidimensional array, meaning it is an array of arrays. Each array inside the fields array is equal to an extra attachment. The attachments themselves have several fields you can use that are all optional:
  - — `title` - A bolded title for the attachment.
  - — `value` - An unbolded field for the attachment.
  - — `short` - A boolean value (True or false); by default this is false. This field allows the other fields to be half width and means you can have two short fields next to each other, side

by side in Slack. Using this, you can simulate a key/value pair reading left to right.

Here's a simple example of using the `fields` key to create several attachments for your Webhook:

```
$data = 'payload=' . json_encode(array(
    'username' => 'Incoming Bot',
    'icon_emoji' => ':rocket:',

    'fallback' => 'This is the fallback text',
    'pretext' => 'This is the pretext field value',
    'color' => '#0066ff',

    'fields' => array(
        array(
            'title' => 'First attachment title',
            'value' => 'First attachment value can be some long text \
                        as it goes full width'
        ),
        array(
            'title' => 'Second attachment title',
            'value' => 'Second attachment value',
            'short' => true
        ),
        array(
            'title' => 'Third attachment title',
            'value' => 'Third attachment value',
            'short' => true
        ),
        array(
            'title' => 'Fourth attachment title which can be long too',
            'value' => 'The value of the fourth attachment',
            'short' => false
        ),
        array(
            'title' => 'This is actually the fifth array title',
            'short' => true
        ),
        array(
            'value' => 'And the sixth array value',
            'short' => true
        )
    )
));
```

On the fourth attachment, you can see `short` has a value of `false`. Although this is the default, you can make your code more verbose by specifying this default value. The above code produces the example shown in Figure 2-17.

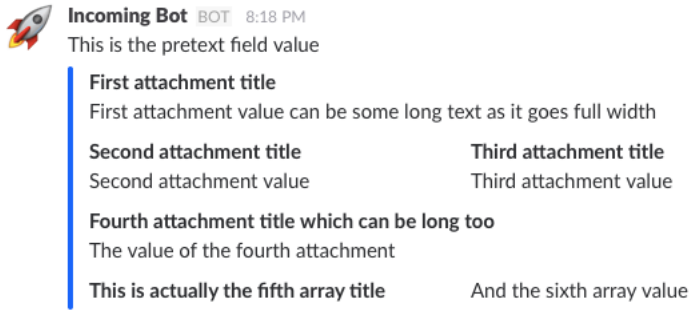


Figure 2-17. Example Webhook Attachments

You now know how to use an Incoming Webhook to post to a Slack channel with a PHP script.

HACK 15 Post a Message from Git

> Now that you can send messages to Slack, let's build up a `payload` programmatically. The payload we are going to build up includes a correctly formatted `$_POST` request to a specified URL provided by Slack.

We are going to use hooks within Git (the popular version control system). If you are unfamiliar with Git hooks, we recommend reading up on them before you get started. Essentially, they are scripts that can be triggered at certain points before or after a commit, push, branch checkout, and more. This allows you to extend and automate several Git tasks.

Using a Git hook, we are going to send a Slack message every time a commit is made on a certain repository. This might not be a practical real world example (as it would annoy everyone in the team), but it will get you used to Incoming Webhooks and will also expose you to Git hooks and their uses.

To build and test this script, you will want to create a new repository that you can commit to regularly without any repercussions, such as a test repo.

Make Your Git Hook

The first step is to create a simple Git hook to fire whenever you make a commit.

Hooks are stored inside the `.git` folder located at the root of your repository. Inside that folder is a folder called `hooks`. It contains some sample files. You can copy a sample file, rename it, and then customize it to make your own hook. For this hack, make sure you title your hook `post-commit` (with no extension). If, however, you decide to create your file from scratch, you must enable execution rights on the file by entering the following command on the command line:

```
chmod +x .git/hooks/post-commit
```

The `post-commit` hook will get triggered whenever you make a commit. Similar to what we did earlier, output a simple "Hello" whenever you commit to double check it's working:

```
#!/usr/bin/php
<?php

    echo "** Hello ** \n";
```

The `#!` at the top of the file is called a shebang and instructs the system on how it should process the file. The one we included tells the system that the following file is PHP.

The next step is outputting the word "Hello" followed by a new line. If you make a commit you should see this output in your terminal.

Now that we've verified that the post-commit hook runs once committed, we can trigger it manually without having to commit code every time. You'll need to do this via the command line:

```
./.git/hooks/post-commit
```

Get the last commit

In order to post our latest commit message to Slack, we first need to get it from the log. PHP can run commands using the built in `exec` function. Using the `-1` flag on `git log` we are able to retrieve *only* the last commit.

```
#!/usr/bin/php
<?php
```

```
exec('git log -1', $output, $src);

var_dump($output);
```

Running this should output your last commit in an array, with each line as a new string. This information is not very handy and it would be hard to extract anything useful from it to post to Slack.

As a way of customizing the log output, Git allows you to pass the `--format` flag, enabling you to customize the output using placeholders. Take the following example:

```
git log -1 --format=%h%n%an%n%ae%n%ar%n%s%n%b
```

Although the format looks like gibberish, there is some logic too it. Each value is followed by a new line placeholder (`%n`). Removing those leaves us with the following:

- `%h` - abbreviated commit hash
- `%an` - author name
- `%ae` - author email
- `%ar` - author date, relative to now
- `%s` - subject (this is the subject of the commit message)
- `%b` - The commit body (If the body is spread over multiple lines, this will appear as multiple strings when `var_dump`ed.

This `git log` command can be added to our `post-commit` - add the following to your file:

```
// Get the last commit
exec('git log -1 --format=%h%n%an%n%ae%n%ar%n%s%n%b', $output);
```

Extract the Relevant Data

Now that we have specified the format of the commit message, we are able to know exactly what value the first five keys of the `$output` array will be.

Because the commit `body` can be as long as it wants, we are going to assign the known values and remove them from the array. This leaves `$output` with just the body message of the commit message. To do this, assign the values to human readable variables, and then `unset()` them from the `$ouput` array.

```
$hash = $output[0];
$name = $output[1];
$email = $output[2];
$time = $output[3];
$subject = $output[4];

// Remove the keys from the array
unset($output[0], $output[1], $output[2], $output[3], $output[4]);
```

The last variable we need is the commit body. This can be obtained by gluing the remaining `$output` array together with a new line between each value. You do this using the `implode()` PHP function.

```
// Glue the array back together as a string
$body = implode("\n", $output);
```

Make sure this is placed *after* the `unset()` function.

Build Your Message with Attachments

Using the code from the previous hack (#14), build up the correctly formatted `payload` to send off to Slack.

One last thing is that you can generate links in Slack Webhooks using a `<|>` syntax, allowing you to pass in `<http://www.google.com|Google>` (note the pipe character in the middle). This will produce a link to Google with the text "Google." Using this technique, you can create an email link to your author by passing in the two variables:

```
'<mailto:' . $email . '|' . $name . '>'
```

You now have the following code:

```
#!/usr/bin/php
<?php
    // Your webhook URL
    $webhook = '[WEBHOOK URL]';

    // Get the last commit
    exec('git log -1 --format=%h%n%an%n%ae%n%ar%n%s%n%b', $output);

    // Assign the variables we know
    $hash = $output[0];
    $name = $output[1];
    $email = $output[2];
    $time = $output[3];
    $subject = $output[4];

    // Remove the keys from the array
```

```
unset($output[0], $output[1], $output[2], $output[3], $output[4]);
// Glue the array back together as a string
$body = implode("\n", $output);

// Encode the data
$data = 'payload=' . json_encode(array(
    'username' => 'Git Bot',
    'icon_emoji' => ':robot_face:',

    'fallback' => 'The latest git commit',
    'pretext' => 'There has been a new commit',
    'color' => '#0066ff',

    'fields' => array(
        array(
            'value' => 'Commit Hash:',
            'short' => true
        ),
        array(
            'title' => '#' . $hash,
            'short' => true
        ),
        array(
            'title' => $hash
        ),
        array(
            'value' => '<mailto:' . $email . '|' . $name . '>',
            'short' => true
        ),
        array(
            'value' => $time,
            'short' => true
        ),
        array(
            'title' => $subject,
            'value' => $body
        )
    )
));

// PHP cURL POST request
$ch = curl_init($webhook);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);

echo $result;
```

Notice that for the commit hash, we have used two short fields to put the title and value next to each other. This gives the hash a title without taking up more vertical space.

The above code generates the result shown in Figure 2-18.

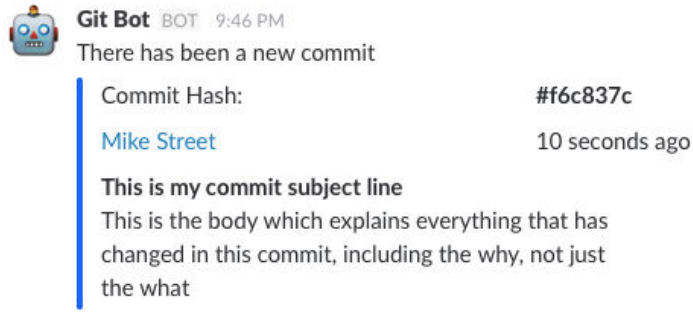


Figure 2-18. A Git Hook triggering an Incoming Webhook

HACK 16 Send a snippet of code from Jetbrain's IDE to Slack

Slack Storm is a free plugin available on most of Jetbrain's software (such as IntelliJ and PHPStorm) whose purpose is to let you quickly send a snippet of code to a Slack channel by just selecting the code in the IDE and right-clicking it (see Figure 2-19).

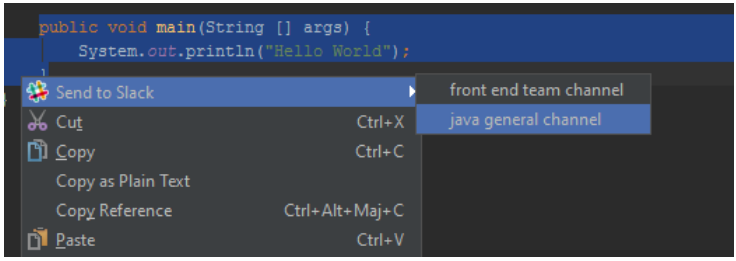


Figure 2-19. Slack Storm app in IntelliJ

The result on Slack is shown in Figure 2-20.

Figure 2-20. Slack Storm app in IntelliJ

Here are the simple steps for configuring Slack Storm:

- Install the plugin in your IDE (choose File > Settings > plugins, enter "Slack Storm" in the plugin name field, and click on "Search in repositories").
- Setup an Incoming Webhook on your Slack team, as outlined at the beginning of this chapter.
- In the IDE go to Tools > Slack Settings > Add Slack Channel, and then insert the channel details: A channel description, the Incoming Webhook you just created, the username the bot will use to post the snippet, and an emoji to use as the bot user's avatar.

3

Slack meets IFTTT and Zapier

While there is a huge selection of integrations available for Slack, sometimes the features you need to connect Slack to just aren't available. You can always write your own Webhook integration, but there are a couple of popular options for the non-developer, namely IFTTT (If This Then That) and Zapier.

While both of these platforms offer similar capabilities, IFTTT offers more integrations for personal use and IoT devices, while Zapier is geared more toward integrating business applications. Our recommendation is to check out both services, and use the one that best supports your needs.

> At this time, IFTTT is a completely free service, while Zapier offers a limited free plan as well as multiple paid plans.

The source code for all hacks in this chapter can be found in this book's Github repo.

IFTTT

IFTTT is a web-based service that allows you to chain together simple "recipes" that are trigged when one of your Internet-connected

devices or applications does something. Whether you want to hack Slack to display who's in the office by leveraging your Samsung SmartThings sensors, or you want to automatically post your daily Fitbit activity to your fitness channel, IFTTT can help you accomplish it.

The one major limitation to what IFTTT can achieve with Slack is that IFTTT can only *post* to Slack, and thus you cannot (without a little bit of hacking—see Hack #21) use it to trigger an action from within Slack.

Getting started with IFTTT

In order to get IFTTT integrated with Slack, you first need to create an account at *https://ifttt.com*. Once you've signed up, you will be presented with a screen that looks something like the one shown in Figure 3-1.

Figure 3-1. IFTTT Welcome Screen

Follow the onscreen tutorial to create your first recipe, and once you're done, follow along in Hack #17 to create your first Slack-integrated recipe.

HACK 01 Create an IFTTT Slack-integrated Weather recipe

On the IFTTT website, clicking the *My Recipes* menu item will present you with a screen that allows you to create custom recipes, as shown in Figure 3-2.

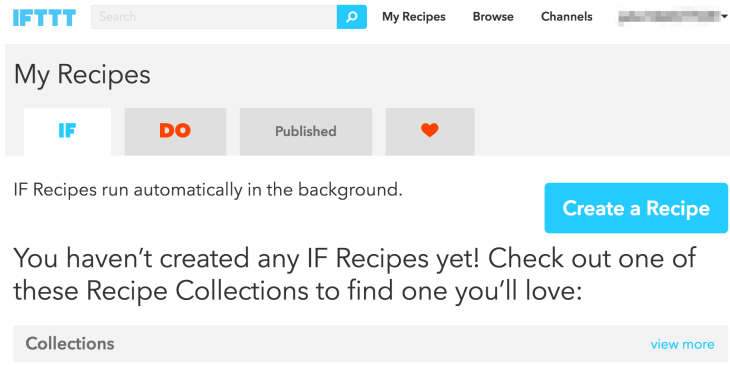


Figure 3-2. My First Slack Recipe

Go ahead and click on "Create a recipe," and you'll be presented with the "If this then that" screen. We're going to start with a simple integration to get weather information into Slack, and then we'll tie into some some more exciting applications.

Click on **this** and search for "weather." In the search results, click on "weather", and you'll see a screen asking you to connect with The Weather Channel. A screen similar to Figure 3-3 will be presented any time you connect to a new application. Clicking the button registers the integration into your IFTTT account.

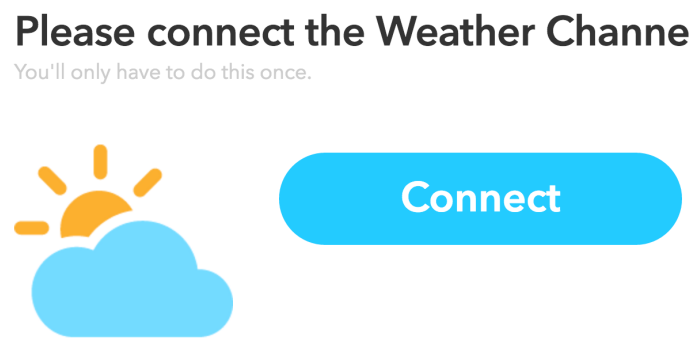


Figure 3-3. Connecting to The Weather Channel

Clicking **Connect** will take you to a screen where you can search for the location you want weather information to come from.

Once you've clicked "Connect," continue to the next step, and choose "Today's weather report." This will scroll you down to select a "time" for this event to be triggered. Select any time in the near future because you'll want to see this data flow into Slack soon. Continuing will then take you to a screen shown in Figure 3-4, which allows you to select a *that* action.

Figure 3-4. Select that

Click "that" and you'll be presented with the "search" screen. Type "slack" into the search box, and you'll see "Slack" as one of your results (see Figure 3-5).

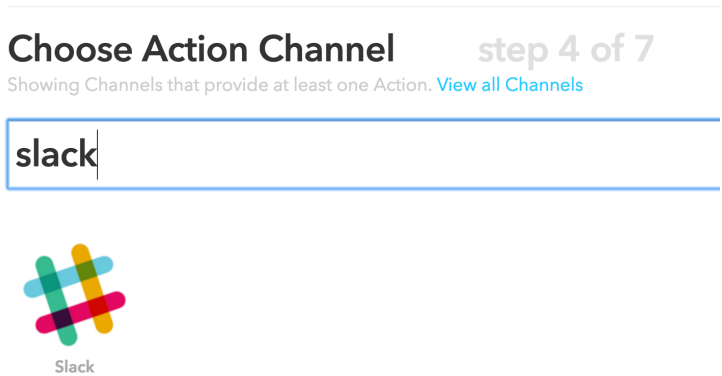


Figure 3-5. IFTTT Slack

Select Slack and connect it like you did with The Weather Channel. If you're currently signed into your team(s) in your browser, you'll see them listed. If not, sign into the team you want to integrate by selecting "sign into another team." Once you've selected your team, continue by authorizing IFTTT to post messages to your team (see Figure 3-6).

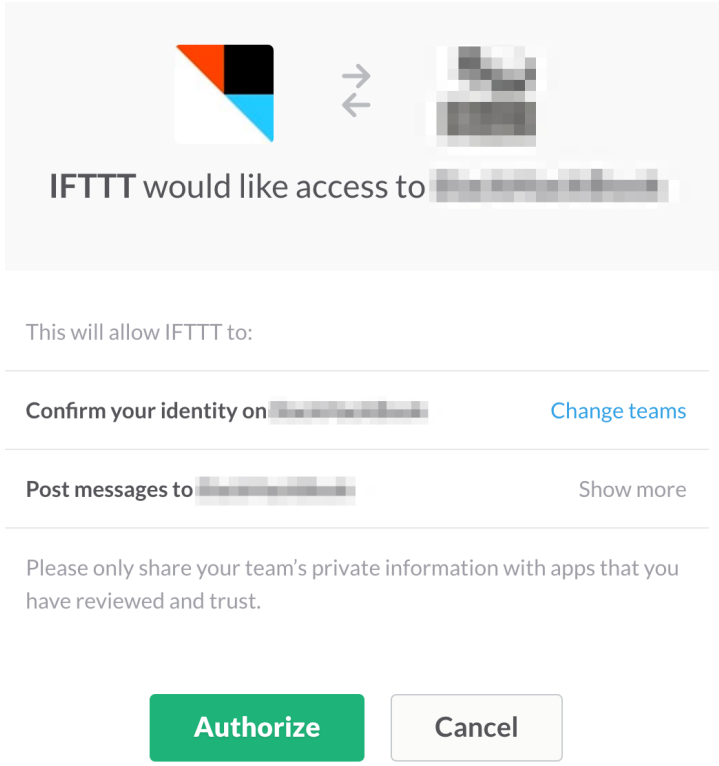


Figure 3-6. IFTTT Slack Authorization

After authorizing, continue on until you're able to "Choose an Action" for Slack. (As mentioned earlier, currently IFTTT only supports the *posting a message to a channel* action.) The list of actions available varies from channel to channel. They consist of actions IFTTT can perform. Clicking on the action will present you with the form shown in Figure 3-7, which you can customize to meet your needs.

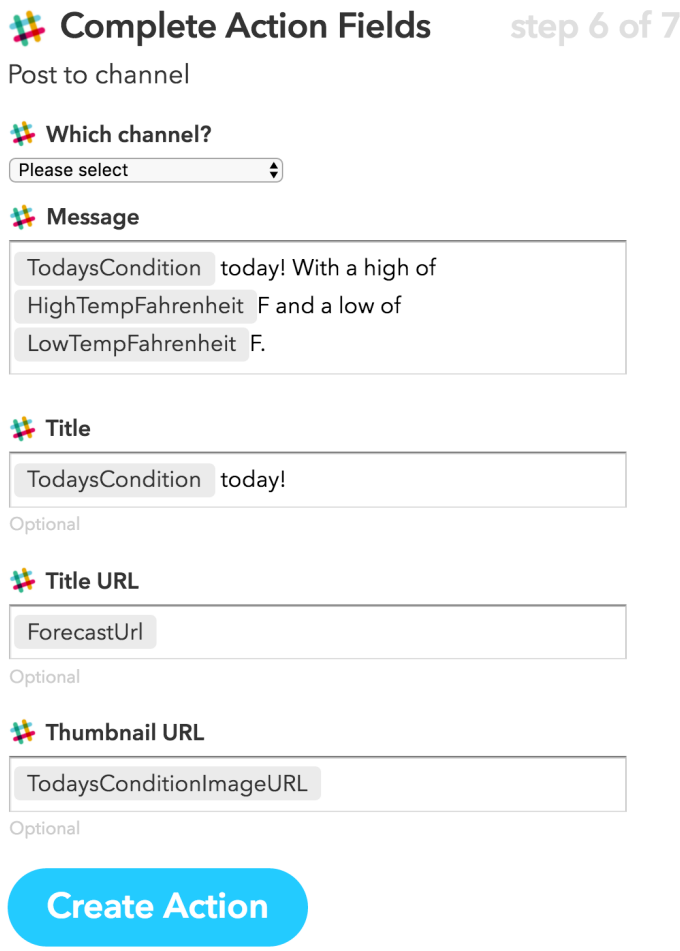


Figure 3-7. IFTTT Complete your action

The fields in this form allow you to customize what the output of the Slack messages look like. The field text consists of fixed text and variables. The variables have a gray background when you're just viewing the form, and switch to being encased in brackets {{ }} when you click into a field to edit. When you click into a fields, a flask icon (see Figure 3-8) will appear in the top right that allows you to view the available variables (a.k.a. ingredients).

Figure 3-8. This flask exposes all of the ingredients that are available to use in this action.

If you wish to change the text by adding more ingredients, click the flask to reveal a drop down where you can choose from available options (see Figure 3-9). These will change depending on the action you have chosen.

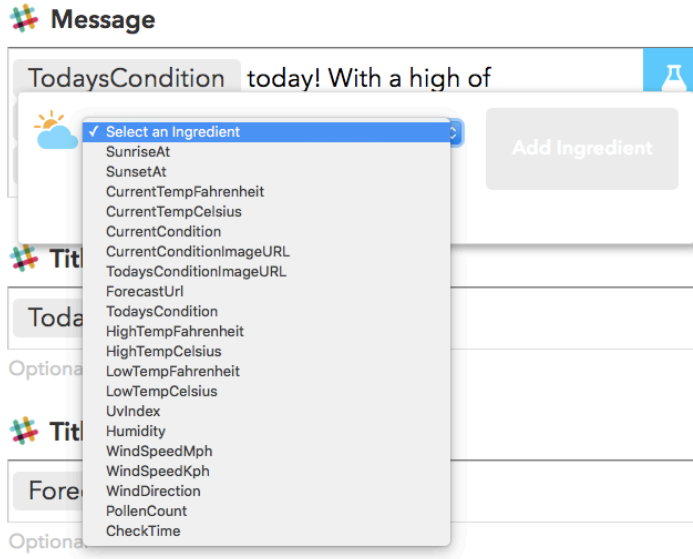


Figure 3-9. A list of available ingredients

Once you're happy with your message and text fields click "Create Action," name your recipe, then click "Create Recipe" and voila, you have a Slack app! Once the time you've set has passed (if you set a time, that is), you'll see a message that looks something like Figure 3-10 posted to your chosen Slack channel.

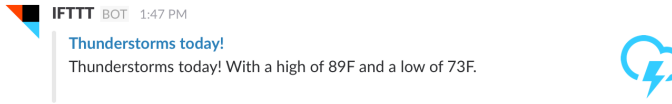


Figure 3-10. IFTTT Post To Channel

HACK 02 Monitor your Twitter followers with IFTTT

Twitter is a great social network, but as you get more and more followers, it's easy to miss notifications when somebody @mentions you or follows you. IFTTT and Slack can really help in this regard. By leveraging IFTTT, you can be notified when any of these events occur, and you can stream them into a #social channel.

Create a new recipe

To get started, head to www.ifttt.com, Click *My Recipes* and then click the *Create a recipe* button.

Select Twitter for your *this*

Click on **this** and search for Twitter. After typing a few letters, Twitter should be one of your choices (see Figure 3-11).

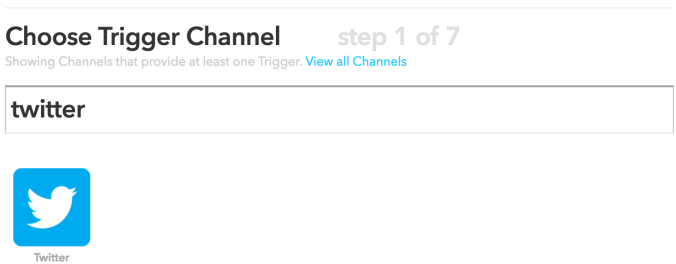


Figure 3-11. If Twitter…

Connect your Twitter account to IFTTT

If you've never used Twitter in this IFTTT account, you'll be prompted to connect. Go ahead and click the "Connect" button, and Twitter will prompt you to authorize IFTTT to access your account (after you've logged in). IFTTT asks for a lot of permissions so that it can be used as both the **this** and the **that** in IFTTT.

Once you've authorized access, continue to the next step in IFTTT.

Choosing a trigger

IFTTT's Twitter integration offers a lot of triggers. For this example, we'll choose "New follower" (see Figure 3-12). This means that any time somebody new follows your account on Twitter, this recipe will be initiated. Since this is a very simple trigger, you won't need to provide any additional settings. When prompted, click "Create Trigger".

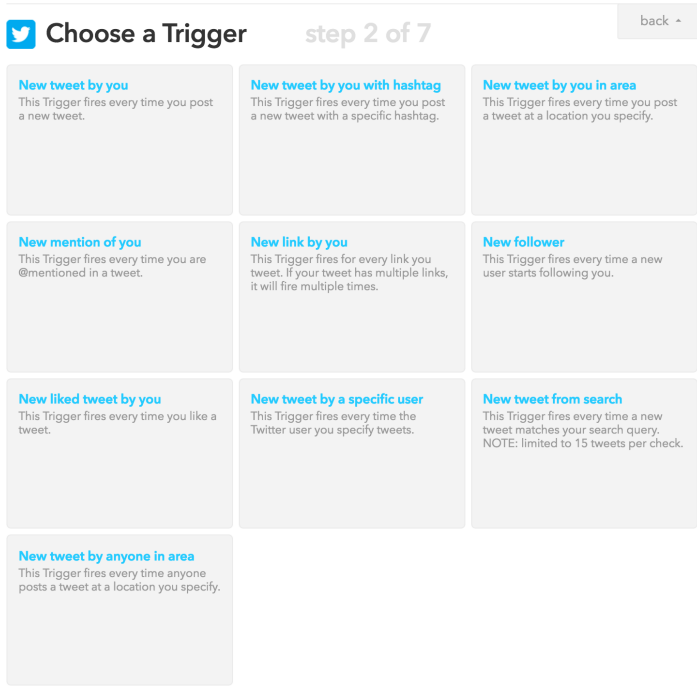


Figure 3-12. Choose a trigger

Selecting Slack for your *that*

Whenever you get a new follower, you want to have IFTTT post to a Slack channel of your choosing. Since you've already integrated Slack with IFTTT, this should be a breeze (see Figure 3-13).

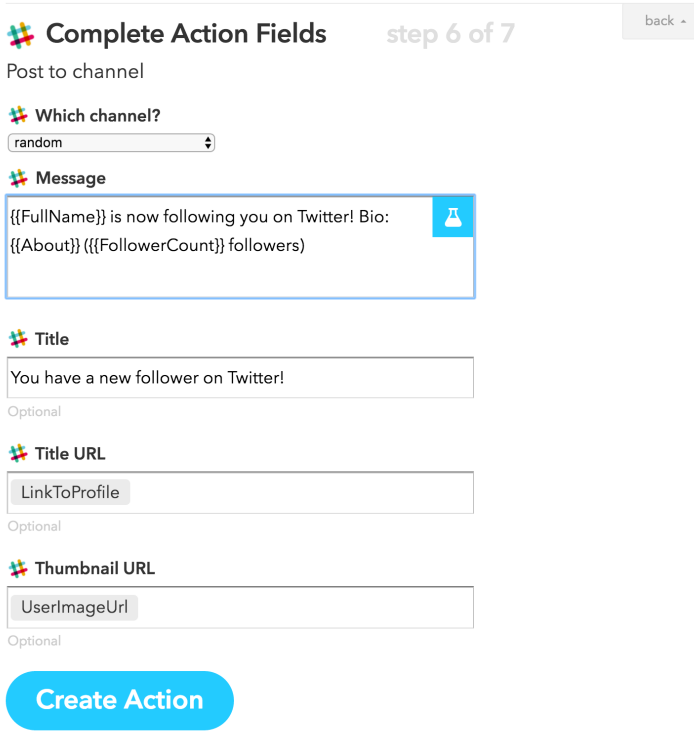


Figure 3-13. Complete the various action fields

You'll notice that there are a lot of different options On the Complete Action Fields screen. Clicking inside of any of the text boxes will present you with the flask icon you saw earlier in this chapter. Click it to see a list of all the "ingredients" that are available to use in this action.

Customize the message as you wish, adding/removing ingredients as you see fit, and then click Create Action. You will now receive a message in Slack whenever somebody new follows you on Twitter.

Triggering IFTTT from Slack

Since IFTTT doesn't allow you to trigger recipes based on what happens in Slack, we are limited by what we can easily achieve. Luckily, we have a trick up our sleeves that will allow us to get information out of Slack, and into another service (or even back into Slack). The trick involves using Maker (which allows you to connect IFTTT to any service that can make or receive web requests) with IFTTT; see Hack #21 for details. Some other great examples of what IFTTT can be used for include:

- Reminding the team about an upcoming meeting
- Posting a message when you enter or exit an area, such as "Alan has arrived at the New York office" (this requires the IFTTT Android or iPhone app)
- Posting an image of the day

The IFTTT website includes a bunch of prebuilt recipes that you can install. You should browse the existing recipes to get some ideas!

HACK 03 Tweet from inside of Slack with Zapier

Zapier is a powerful tool that serves a similar role as IFTTT, but it allows you to accomplish a lot more with Slack. One of the big advantages of using Zapier is that it allows you to get data out of Slack, not just put data into it. This means you can monitor public channels for certain words, and trigger an event to happen when it detects those words.

The free Zapier offering is quite limited at this time because it only allows you to use a single action in any given *Zap*. A Zap is what Zapier calls a recipe, but unlike IFTTT, Zapier allows you to filter, transform, and chain actions together, which makes it well worth the price to achieve great Slack solutions with little to no development work.

Getting started with Zapier

In order to get Zapier integrated with Slack, you first need to create a Zapier account by going to *https://zapier.com*. Once you've signed up, you will be presented with a screen that looks something like Figure 3-14.

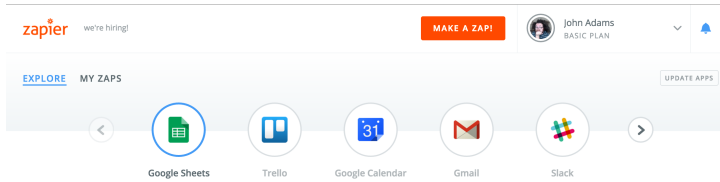


Figure 3-14. Zapier Welcome Screen

In this hack, we are going to create a Zap that allows you to tweet from inside of Slack. This requires either a paid version of Zapier or a trial of the Basic plan.

Create a new Zap

Go to *https://zapier.com/* and click "Make a Zap!" This will take you to a screen where you can select your "Trigger App."

Select Slack as your Trigger App

Underneath your name, you'll see that there are two prominent sections: "Trigger" and "Action." The trigger is the event that causes a Zap to happen, and an action is the result (or results, in more complex cases) of the aforementioned trigger.

You should see that "Choose App" is already selected for the trigger, and in the main section of the screen you will be presented with a search box and some "Popular Apps." If you see Slack, click it; if you don't see it, search for it by typing *Slack* into the search box (see Figure 3-15).

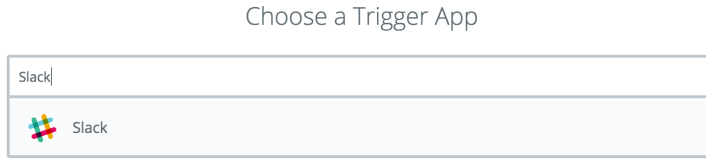


Figure 3-15. Slack as a Trigger

Select Your Trigger

The trigger is up to you, so feel free to adapt it to your needs. To make sure the Zap gets picked up anywhere in Slack, select "New Message Posted Anywhere." Saving and continuing will prompt you to choose an associated Slack account, or to associate a new Slack account if you haven't done so. Authorizing your account is pretty straightforward, and eventually you'll be presented with a screen asking you to post a "brand new message."

Post a Slack message that will be sent to Twitter

In Slack, head to a public channel (we recommend #random) and type "Twitter: My first Tweet from #slack!" Navigating back to your Zapier screen should show you a "Test successful" message.

Validate your message

Click "View your message" in Zapier in order to verify that the right message was received. The text of the message should be what you typed into Slack.

Create a filter

The purpose of this hack is to allow you (but not another team member) to type a message in a Slack channel that is automatically posted to your Twitter account. To do this you must create a filter in Zapier. Click on the + sign in the left hand navigation (above *action*), and then select the "Filter" button to create a filter (see Figure 3-16).

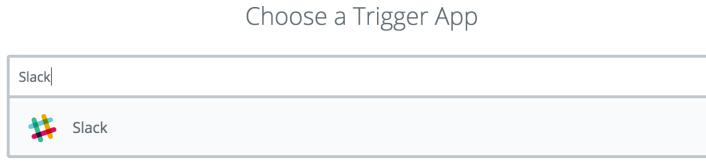


Figure 3-16. Create a filter

Select "Only continue if" and continue. Using the dropdown boxes, build up two filters with the following options (Replace [your username] with your actual username):

1. *Text / (Text) Starts with / Twitter:*
2. *User Name / (Text) Exactly matches / [your username]*

Once these filters are set up, you can test your filter on the Zapier website by clicking **continue**. You should see the filter results highlighted in green on the next page in Zapier.

Create an action: Code

For your action, search for "Code by Zapier." This will let you choose to write logic in either Python or JavaScript. Select the "Run Javascript" option.

Edit your Code template

First off, you need to create an Input that maps the Text "Twitter: My first Tweet from #slack!" into a variable called `data` (see Figure 3-17).

In the Code section, type the following, which removes the `Twitter:` portion of your Text and creates an output called *message*, which will be used in subsequent steps in this Zap:

```
output = {
  message: input.data.substring(9).trim()
}
```

Your page should now look like the one shown in Figure 3-17.

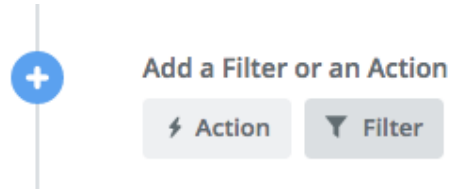


Figure 3-17. Coding with Zapier

Click **continue** and the next page of the zap should show the "My first Tweet from #slack!" message. Now let's add another action.

Create a Tweet action

Let's now add a second Action to your zap (remember, you need to have a paid Zapier membership—or at least a trial of one—in order to create multi-step Zaps).

In the zap window, select Twitter as your next action, choose "Create Tweet" as your Twitter action and, if you haven't already connected Zapier and your Twitter account, do that now.

Edit your Twitter template

Your next step in this zap is to "Set up Twitter Tweet." Go to "For your Message," in the zap window and select the Message output. Using the dropdown, select "Step1 Test." You can also choose to have Zapier shorten any URLs you type, which is a great feature. You then select the Continue button and this multi-step zap is complete. You can access the zap in the Zapier dashboard and be sure to turn on the zap.

To test this zap, go into one of your Slack channels and type "Twitter:" followed by a message. Whatever you type as your message (anything after Twitter:) will automatically be posted as a tweet in your Twitter account.

HACK 04 Create a screenshot of any URL

In this hack we are going to create a Zapier Zap that allows you to generate a screenshot of a website from inside of Slack. We're going to accomplish this by leveraging a service called CloudConvert (cloudcon-

vert.com). This service has a free tier, and is capable of so much more than what we're using it for, so feel free to play around with it.

Create a new Zap

Go to *https://zapier.com/* and click "Make a Zap!" This will take you to a screen where you can select your "Trigger App."

Select Slack as your Trigger App

There are a lot of choices for trigger apps, but we're interested in Slack. This integration provides a lot of flexibility, and will enable you to trigger Zaps without the need for slash commands.

Go ahead and type *Slack* into the search box, and then select it in the results.

Select Your Trigger

The trigger is up to you, so feel free to adapt it to your needs. For this example, select "New Message Posted Anywhere." Saving and continuing will prompt you to choose an associated Slack account, or to associate a new Slack account if you haven't already done so. If you haven't authorized your Slack account yet, be sure to do that.

Post your first message in Slack to generate a screenshot

In Slack, head to a public channel (we recommend #random) and type "screenshot: www.google.com". If you head back to Zapier, you should see a "Test successful" screen.

Validate your message

Make sure the message Zapier got was the correct one. You can validate this by expanding the "view your message" portion of the zap window.

Create a filter

Like in Hack #19, we want this action to be specific to you in Slack (meaning no other Slack users can use it), so we need to add a filter in Zapier. Click on the + sign in the sidebar to create a filter. The

filter will need to have the following two settings (Replace [your username] with your actual username):

1. *Text* / *(Text) Starts with* / *screenshot*:
2. *User Name* / *(Text) Exactly matches* / *[your username]*

Once this is set up you can test your filter on the Zapier website by clicking **continue**. You should see the filter results highlighted in green on the page that appears in the zap window.

Create an action: Code

The next step in the zap is the action, so you'll want to select "Code by Zapier." This will let you choose to write logic in either Python or JavaScript. For this example we will use JavaScript.

Edit your Code template

Proceed with the zap by creating an Input that maps the Text into a variable called *data*.

In the Code section of the zap window you need to type the following code snippet. This code removes the `screenshot:` portion of your message, and creates an output called *message* that will be used later in this zap:

```
output = {
  url: input.data.substring(11).trim().split('|')[0]
};
```

Continuing and testing your new action allows you to see the output of this step. The message that is output should be *http://www.google.com*. You'll notice that Slack has taken care of adding in the http:// portion of the URL, which will be useful later on when we run this hack, since CloudConvert will be accessing that URL.

Create a CloudConvert action

Next, you'll add a second Action to Slack (as you know, you need to have a paid Zapier membership in order to create multi-step Zaps).

Select CloudConvert as your next Zapier action, choose "Save Website" as your CloudConvert action, and provide CloudConvert access to Zapier. Of course, if you haven't set up a CloudConvert account, you will need to do so (it doesn't cost anything).

Edit your CloudConvert template

In the Edit Template field in the zap window select the "Step 3 URL" option in the dropdown as the Website URL option. For the Output Format you can choose whatever you'd like, but we will go wih JPG.

At the bottom of the screen you'll see an option for Save Output. Set this to yes, so that the JPGs are individually saved on CloudConvert.

Create a Slack Action

At this point in the zap we have a JPG in CloudConvert, but we need to get it inserted into the Slack channel via an action.

In the zap window, add a new Action, and pick Slack. You'll notice there are a lot of new options (since this isn't a trigger). Choose "Send Channel Message." This allows you to post the JPG from CloudConvert back into the Slack channel you initiated this zap from.

Edit your Slack Channel Message template

It is now time to Edit the Template of our message in the zap window. For the "Channel" option, choose "Use a Custom Value". This will let you specify a channel based on a previous step in the zap. For the "Custom Value for Channel Name" choose the "Step 1, Channel Name" from the dropdown. For the Message Text, you can mix text and values from previous steps, such as: "Generating a JPG of" followed by the "Step3:Url" option. For "Attach image by URL", choose "Step 4, OutputURL."

If you don't want to expand the URL in the Message Text, you can disable this behavior by changing Auto-expand links to No.

Select the "Testing this Step" in the zap window to have your message sent to Slack. The Image will automatically be attached to your post. Congratulations, you have completed this Hack! Now, in a Slack channel type "Screeshot:" followed by a URL, and an actual JPG of that website will be returned to Slack.

Prebuilt Zaps

Similar to IFTTT, Zapier has plenty of prebuilt zaps that you can install. These range from ones that let you get data from YouTube, MailChimp, Email, Google Drive, Sales Force (and plenty of other services) to ones that let you post data to Trello, Twitter, Google

Tasks, and Wrike, among others! Head over to the Slack Zapbook to get some idea of what Zapier can do.

HACK 05 Trigger IFTTT using Slack

At the beginning of this chapter we said that it wasn't possible to trigger an action from Slack using IFTTT without a little bit of hacking. Now, it's time to create a hack that allows Slack to trigger IFTTT.

Using the Maker trigger channel

This channel allows you to trigger an event each time a specific GET or POST HTTP request is done on a specific URI. Here's the URI format:

```
https://maker.ifttt.com/trigger/{event}/with/key/{secret_key}
```

You can pass up to three parameters, either using JSON content if you do a POST request, or as request params for a GET request. You can find all of this information on the Maker channel of IFTTT. The Maker channel allows you to connect IFTTT to any service that can make or receive web requests.

The first step of this hack is to get the secret key, which you do by navigating to *https://ifttt.com/maker* and clicking on *connect*. As long as you have an IFTTT account, you will be given a secret key.

Now that you have a key, you can proceed with the rest of the steps in this hack.

Linking the Maker trigger to a Slack event

On the Slack side, we use the Slack RTM API, with the goal of notifying a specific user or a channel about any Slack events.

For this hack we will build a small Slack bot written in Java by using the Simple Slack API library, which provides an easy way to get notified of events coming from the Slack RTM API (this library is described in detail in Chapter 4).

The following code snippet connects to a Slack team and has a listener listening to incoming messages:

```
public static void main(String[] args) throws Exception {
 //creating the session
 SlackSession session = SlackSessionFactory.createWebSocketSlackSession(TOKEN);
 //adding a message listener to the session
 session.addMessagePostedListener(SlackIFTTTTrigger::listenToIFTTTCommand);
 //connecting the session to the Slack team
 session.connect();
 //delegating all the event management to the session
 Thread.sleep(Long.MAX_VALUE);
}
```

This next code snippet then connects to the Slack RTM API and adds a listener that is notified for each posted message:

```
private static void listenToIFTTTCommand(SlackMessagePosted event,
                                         SlackSession session) {
 if (!event.getChannel().isDirect() || event.getSender().isBot()) {
  return;
 }
 if (!event.getMessageContent().trim().startsWith("!IFTTT")) {
  return;
 }
 String [] splitCommand = event.getMessageContent().trim().split(" ");
 if (splitCommand.length<2) {
  session.sendMessage(event.getChannel(),
                      "You have to provide an IFTTT event to trigger," +
                      " using the following syntax: !IFTTT {event_name})");
  return;
 }
 notifyIFTTT(event, session, splitCommand[1]);
}
```

The listener filters the event it receives to keep only messages, and the content is followed by an event name sent as a direct message to the Slack bot. If the event is not filtered out, then IFTTT is triggered, as you can see here:

```
private static void notifyIFTTT(SlackMessagePosted event,
                                SlackSession session, String iftttEvent) {
 HttpClient client = HttpClientBuilder.create().build();
 HttpGet get = new HttpGet("https://maker.ifttt.com/trigger/" +
                           iftttEvent + "/with/key/" + IFTTT_SECRET_KEY);
 try {
  HttpResponse response = client.execute(get);
  StatusLine statusLine = response.getStatusLine();
  if (statusLine.getStatusCode() >= 400) {
   session.sendMessage(event.getChannel(),
                       "An error occured while triggering IFTTT : " +
                       statusLine.getReasonPhrase());
  }
 } catch (IOException e) {
```

```
            session.sendMessage(event.getChannel(),
                                "An error occured while triggering IFTTT : " +
                                e.getMessage());
        }
    }
```

Using the code in this hack, we performed a GET HTTP call matching the required template, which then passed the IFTTT secret key and IFTTT event name as an argument. The end result is that you can use this code to trigger IFTTT remote-controlled software or objects with a simple Slack command. Just build the recipe with the appropriate Maker trigger and you will be set.

There are so many projects that take advantage of this IFTTT recipe, from pool temperature monitors to Darth Vader's command center. Go to *https://www.hackster.io/ifttt/projects* for a long list of examples.

Simple Slack API

Don't panic if you don't understand all of the implementation details about the Simple Slack API library (registering listener, events, sending messages and so on). The point of this hack was to work around the fact that IFTTT can't be triggered using Slack. In Chapter 4, you will learn a lot more about the library used to achieve this hack.

4

Develop Your Own App

In this chapter you will learn how to directly integrate with the Slack API by writing and deploying apps and slash commands for Slack.

The hacks described in this chapter cover specific applications, directly calling the Slack web API by manually performing HTTP calls to remote methods. You will become familiar with the API by seeing all of the interactions between the applications and the Slack server.

Slack API overview

Slack provides a powerful API for developers and admins to interact with. This API allows scripts to read and write to channels and conversations, and interact with users by means of bots and slash commands.

When the API is used, it transforms Slack from a normal chat client into a powerful central hub for your company. For example, scripts can provide you with regular updates on sales, or you can interact with your on-site live chat widget.

The Slack API is divided into two parts:

- *The Slack Web API*: This API is used to act on a Slack team, to post a message, to create a channel, or to browse a channel history.

- *The Slack RTM (Real Time Messaging) API*: This API is mostly used to get notified on what's happening in a Slack team.

We'll discuss both of these parts in detail in the following sections.

Curious why Slack has two APIs? Jump ahead to "Why two kinds of APIs?" for an explanation.

Authentication

Whichever part of the API you use (you might use both), you will have to perform authentication using a token. This token can be provided through an OAuth flow, or it can be tapped directly and given by Slack if it's a bot access token.

Slack Web API

This API is a set of methods that are called remotely using HTTP (GET or POST) calls. There's a method for every action you can perform on Slack: Sending a message, modifying it, subscribing to a channel, leaving a channel, browsing the channel list, and more. You will find a matching method in the Web API for all of these actions.

On each method call you have to provide the token you received for authentication in order to identify what is calling the method, and to see if it has the right to call the method. The response to the call will be in JSON format, and will give you some information about the way the command you sent was handled (whether it succeeded or failed). And, if you post a new message, the response will also provide you with the message's timestamp. This timestamp is useful since it serves as the message identifier. You will have to provide this timestamp if you want to later modify or delete the message.

For example, say we call the `chat.postMessage` method which uses the following arguments to post a message on a channel:

- *token*: your authentication token
- *channel*: #general
- *text*: hello from the API

This provides us with the following answer:

```
{
    "ok": true,
    "channel": "C1CFWD0PP",
```

```
        "ts": "1466844677.000002",
        "message": {
            "text": "hello from the API",
            "username": "testuser",
            "bot_id": "B1L8V9SU9",
            "type": "message",
            "subtype": "bot_message",
            "ts": "1466844677.000002"
        }
    }
```

You can see here the `ok` field is `true`, indicating the command was successful, along with a full description of the message that you sent. We won't describe every field, because the official documentation is quite thorough, but we will find the message timestamp stored in the `ts` field within `message`. As mentioned earlier, you need this timestamp to modify or delete the message.

If you make a mistake and send a message on a channel that doesn't exist, for instance, you'll get this result:

```
{
    "ok": false,
    "error": "channel_not_found"
}
```

All of the Web API's methods are described in the official Slack documentation.

Slack RTM API

The Slack Real Time Messaging (RTM) API is WebSocket-based. It lets you to receive events from Slack in real time, so you can send messages as a user. Slack Bots can also use this API to interact with users.

In order to use this API to listen to Slack events, you first need to make a call to *https://api.slack.com/methods/rtm.start*. This call returns a full description of the team as seen by the caller:

- Users in the team and their profiles (including avatar links)
- Public channels
- Private channels on which the third party application is a member of (other private channels won't be listed)

In addition to this full description, you'll be provided with a unique URL to open a secure web socket connection. This connection, once

established, allows for a bi-directional dialog between your application and the Slack servers. This dialog is in JSON format.

The ping command

The RTM API does not provide a wide set of message types to send over the websocket, but there's one you should know: *ping*. The purpose of this message is to ask Slack to send you a *pong* response. Sending this message at a regular rate allows you to detect disconnections and to take appropriate measures. To use *ping* send the following JSON message:

```
{
    "id": 1,
    "type": "ping"
}
```

Slack will answer you with this *pong* response:

```
{
    "id": 1,
    "type": "pong"
}
```

The `id` field allows you to match the pong, so you'll have to change it for each request to be able to pair each *ping* you send with a *pong* you receive some time later. The best way to do that is to start at 1 and to progressively increment the value.

Slack team events

The main reason to use the RTM API is to listen for team events through the web socket connection. There can be hundreds of events each second on big Slack teams. Events are just a way to indicate to a user that something happened on the team in such a manner that this user can react to the event. For example:

- When a new channel is created, the API user who receives the channel_created event can react by joining that channel.
- When a message is sent, the API user might want to respond to it depending on its content.

Here is an example of an event you can receive if a message is posted in a Slack channel that you are a member of:

```
{
    "type": "message",
```

```
    "channel": "C123456",
    "user": "U123456",
    "text": "Hello world",
    "ts": "1466844677.000002"
}
```

- *type*: This is the event identifier - in this example, it's a "message" event
- *channel*: The id of the channel the message was posted on
- *text*: The text content
- *ts*: The timestamp of the message that serves as a unique identifier within the channel it was posted in (so the channel's id and this timestamp are a unique key for a message in the Slack team)

The following event is sent whenever a new channel is created:

```
{
    "type": "channel_created",
    "channel": {
        "id": "C123456",
        "name": "newchannel",
        "created": 1462821547,
        "creator": "U123456"
    }
}
```

- *type*: As above, this is the event identifier
- *channel*: This is a JSON object containing details about the channel just created:
  - *id*: The new channel's id
  - *name*: The new channel's name
  - *created*: The new channel's creation time (This is a simple epoch timestamp and, unlike the info in a *ts* field, can't be used to identify channels)
  - *creator*: The user id of the user who created this new channel

There's a lot of different events you can receive through the RTM API. You can find a complete list here.

Why two kinds of APIs?

Slack made two APIs for a simple reason: they separated the APIs' duties.

The RTM API, using a websocket to keep a permanent connection, is a convenient way to quickly receive notifications. You don't have to poll a server at a regular rate to get notified, and the notifications are pushed through the websocket each time an event happens.

The web API, on the other hand, doesn't require a permanent connection, but it's well suited for performing some remote procedure calls and for acting on a server. A drawback of this kind of call is that it adds some latency due to the fact you have to establish a connection to the HTTP server for each call.

The fact that websockets are bi-directional doesn't make the web API useless: Slack could have provided a bigger RTM API with which you could call every command by sending JSON objects. But that would have been painful for people developing third party applications whose only purpose was to call methods on the Slack server without having to take care of the team events. That situation would have forced app developpers either to maintain a websocket connection, or to establish a new one each time they needed to call a remote method.

Slack engineers knew about these impediments because, they are using these APIs to build the official clients. If you have a look at the network activity of your Slack web page, you will easily see the calls to the Web API. Figure 4-1 shows an example of what happens when you create a new channel using an official Slack Client.

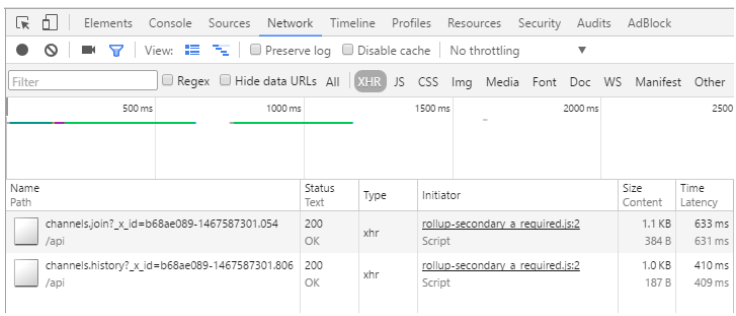


Figure 4-1. Slack's web API calls performed during the creation of a new channel using the official web client

Figure 4-1 shows two steps: * A call to `channels.join`: Despite its name (which is rather deceiving) this method creates a channel if the name of the channel you try to join doesn't exist. * A call to `channels.history`: The client is calling this method to fetch the message history of the newly created channel to retrieve the only message existing in a new channel, which is the initial message indicating that the creator joined the channel. The client needs this message to display the channel content.

You can't easily observe the message going through a websocket, but of course the official clients are using it. That's how they can be aware of new messages posted that they have to display, and how they know about channel topic changes, new channel members, new users, and everything that makes the client responsive to the changes to the team that should be displayed to the user.

An interesting point to note, is that you won't see any web API calls when you publish some messages. This is because the RTM API allows you to publish messages with no attachments, which is a quick way to bypass the latency of HTTP calls on messages that must be pushed as fast as possible (adding attachments to messages is not something users are doing frequently and so it is less latency sensitive). This behavior lightens the load on Slack's HTTP servers, since sending messages is the most common activity of a user. Feel free to have a closer look at the network traffic of the official client, because it contains a lot of useful info for understanding how Slack APIs can be used in your own bots or integrations.

API Language neutrality

You will see later in this book that the Slack APIs can be used by a wide variety of languages. The fact that the APIs are built upon HTTP, websockets and JSON structures makes them language neutral. Any programming language that allows you to do HTTP requests and a bit of JSON parsing/formatting can be used with the APIs. Taking a look at *https://api.slack.com/community* will give you an idea of the range of languages that can rely on these APIs (and the list is constantly growing).

Slack Bots

Bots are where Slack and its integrations really come alive. They allow you to build advanced functionality right into a chat applica-

tion, giving sometimes complex code an easy-to-use and familiar interface.

Bots appear in Slack as users in the team directory. They have the ability to read and write to conversations, they can be invited to channels and private groups, and you can interact with them via direct messages.

The most popular bots include Errbot and Hubot. These bots run independently of Slack and can be extended and customized to cater to the needs of a team.

For example, Errbot can organize tournaments for your team or generate graphs in conversations. Hubert can interact with Trello or return Google PageSpeed results. Chapter 6 contains hacks that use these bots.

Let's get started with some hacks that tap into the Slack APIs.

Source code for the book

The source code for all hacks in this chapter can be found in this book's Github repo.

HACK 01 Use Google Cloud Functions as a slash command endpoint

Slash commands in Slack are shortcuts that users can type to send an HTTP request to a server. The users doesn't need to know all of the details about which server to call and how to pass parameters, they just type `/commandName` with some arguments and Slack does the rest, returning the answer from the server. The Kayak integration presented in Chapter 3 (Hack #10) is a good example of how slash commands can be used in Slack.

In this hack you will learn how to host a slash command endpoint using Google Cloud Functions and integrate it with your Slack team. Google Cloud Functions is a feature available in the Google Cloud Platform, a cloud hosting solution offered by Google (*https://cloud.google.com/*). Using the Google Cloud Platform allows you to

integrate your hosted projects seamlessly with Google's API. This is done using Google Maps' features like generating a custom map image, or fetching a YouTube video stream. Google Cloud Functions are a very simple way to deploy a slack slash command endpoint on the web.

If you don't have yet a Google Cloud Platform account, you will have to create one for free using the URL given above. On the project you will create to host the endpoint, you will need to enable the *Google Cloud Billing API* (this is covered in the Quickstart step below).

Since Google Cloud Functions is still in alpha stage, you must perform the following steps to enable its usage on your project.

1. First, perform the five first steps of the Quickstart: *https://cloud.google.com/functions/quickstart*.
2. At the time of writing, cloud functions are pretty new, so you'll have to go to the API manager to activate the API.
3. Then use `gcloud init` to authenticate and select the project on which you want to deploy your function.
4. Functions at this point are in alpha, so you can add them to gcloud by typing: `gcloud components install alpha`. You will be prompted with the following output:

```
┌──────────────────────────────────────────────┐
│     These components will be installed.      │
├──────────────────────┬──────────┬────────────┤
│         Name         │  Version │   Size     │
├──────────────────────┼──────────┼────────────┤
│ gcloud Alpha Commands │ 2016.01.12 │ < 1 MiB │
└──────────────────────┴──────────┴────────────┘

For the latest full release notes, please visit:
  https://cloud.google.com/sdk/release_notes

Do you want to continue (Y/n)?
```

Once you've performed those steps, create a working directory:

```
$ mkdir ~/slack-function
$ cd ~/slack-function
```

And then create a bucket to store your code. The name of the bucket needs to be globally unique:

```
$ gsutil mb gs://[BUCKET_NAME]
```

Now that your function is ready, you can deploy it to be triggered via HTTP:

```
$ gcloud alpha functions deploy slackhello --bucket [BUCKET_NAME] \
            --trigger-http

Copying file:///tmp/tmpbr_Zen/fun.zip [Content-Type=application/zip]...
Uploading   ...ns/us-central1-slackhello-nojdajqvmusi.zip: 689 B/689 B
Waiting for operation to finish...|
```

Carefully note the given URL (it should be something like `https://[PROJECT].cloudfunctions.net/slackhello`). If you lose it, you can get it back querying the fucntion like this:

```
$ gcloud alpha function describe slackhello
```

Now, try your function to see if it is ready before trying it with Slack:

```
$ curl -X POST <HTTP_URL> --data '{"message":"Hello World!"}'
```

You are now ready to hook your function up to Slack! In your channel, click on the gear and then choose "Add an app or integration." Then in the search box type "command" and select Slash Commands (see Figure 4-2).

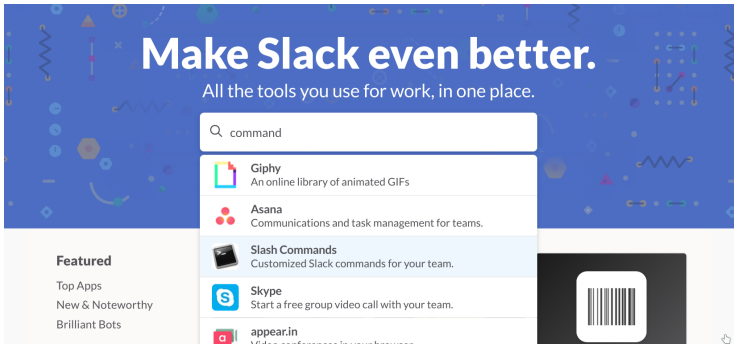


Figure 4-2. Select Slash Commands

Next, name your slash command so you can assign it to your Google Cloud function (see Figure 4-3).

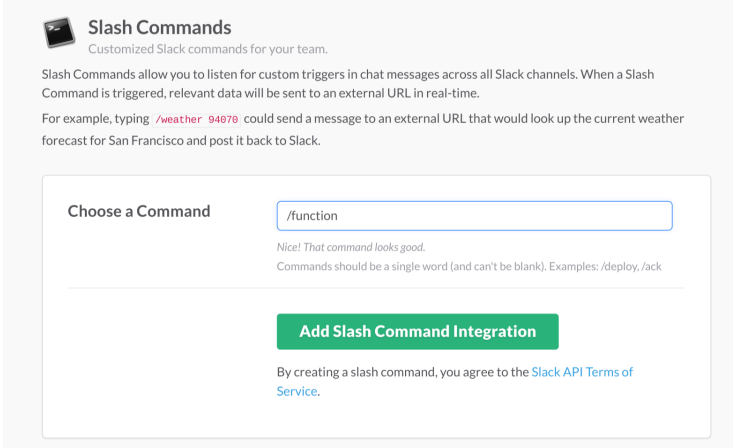


Figure 4-3. Name your command

Then hook your slash command up to the URL of your Cloud Function via the URL field of the slash command settings (see Figure 4-4).

Figure 4-4. Hook up function

Be sure to fill in the *Autocomplete help text* field in the settings to add your slash command in the autocomplete list.

HACK 02 Use an AWS Lambda to host a slash command endpoint

> Amazon Web Services' Lambda is a nice alternative to Google Cloud Functions. It hosts your code on Amazon's cloud so you can focus on your code and delegate the hosting duty to Amazon Web Services.

Using Lambda is a simple way to host a Slack slash command endpoint. Amazon has even provided a *blueprint* for doing so. (Blueprints are pre-built samples that Amazon makes available.)

In your AWS console, go to the Lambdas section and create a new Lambda from the *slack-echo-command-python* blueprint (see Figure 4-5).

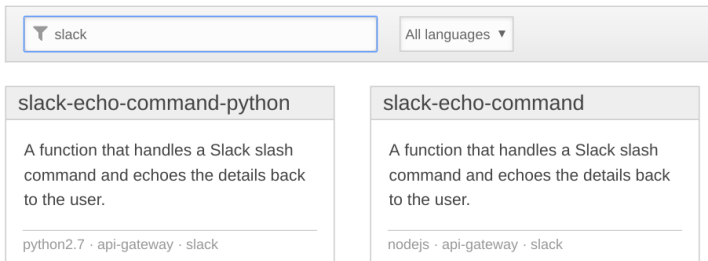


Figure 4-5. slack echo blueprint command

Then replace the template version with this simpler version:

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    user = event['user_name']
    command = event['command']
    channel = event['channel_name']
    command_text = event['text']
    return "%s invoked %s in %s with the following text: %s" % \
            (user, command, channel, command_text)
```

On the lambda endpoint, select "POST" as the method and set security to "Open", as shown in Figure 4-6.

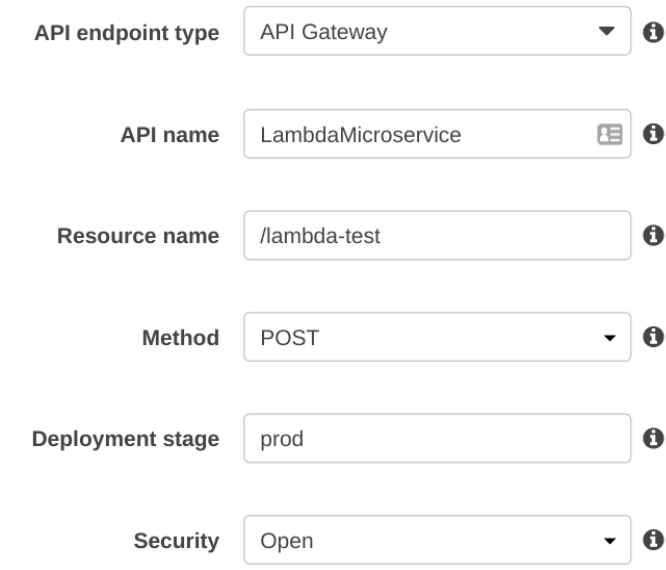


Figure 4-6. Lambda trigger configuration screen

Next, click on the API endpoint to go to the API gateway (see Figure 4-7).

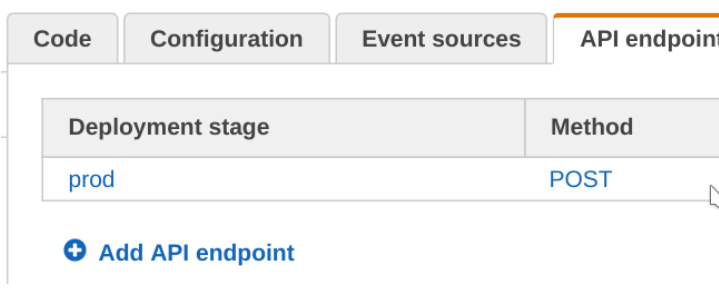


Figure 4-7. API gateway screen

Once the API gateway screen appears, click on the lambda, then the API endpoint, then POST to show the method execution (see Figure 4-8).

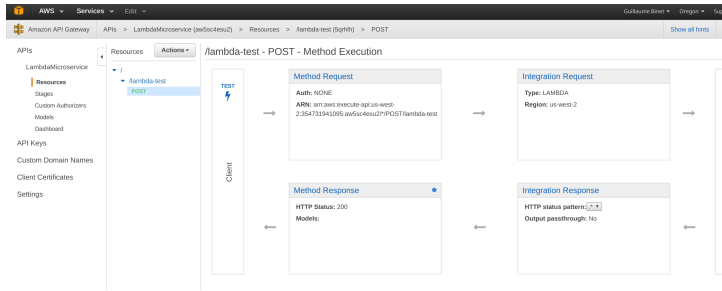


Figure 4-8. Method execution diagram screen

At the bottom of the method-execution diagram screen, click "add mapping template." On the screen that, appears set the content type to `application/x-www-form-urlencoded`, and copy the content available on this Github gist *https://gist.github.com/ryanray/668022ad2432e38493df* into the body.

This transformation is the magic behind the lambda code that transforms the POST data content encoded in `application/x-www-form-urlencoded` into JSON structured data, which is easier to manipulate.

Now that you've setup your endpoint on Amazon Web Services' Lambda, you can create a slash command in Slack using the process described in Hack #22, and then copy and paste the API endpoint into the URL field (see Figure 4-9).

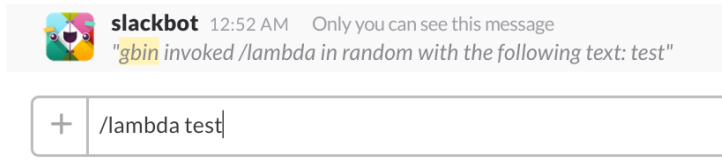


Figure 4-9. Testing the new slash command in Slack

HACK 03 Make a calculator with Google App Engine

Google App Engine is a platform-as-a-service (PAAS) that simplifies greatly the deployment and maintenance of a web based service (Like Heroku or Amazon Web Services). This is exactly what we need for a Slack bot.

In this hack, we will show you how to make a simple calculator with a Slack Slash command in Go.

Make a new application on App Engine

You'll need to open an account on the Google Cloud Platform, and start a project. You can use the GCP free trial: *https://cloud.google.com/free-trial/*.

Once you have created a project ID, write its name down, since it will be useful later. You can also see the project ID in the drop down menu on the top right of *https://console.developers.google.com*, as shown in Figure 4-10.

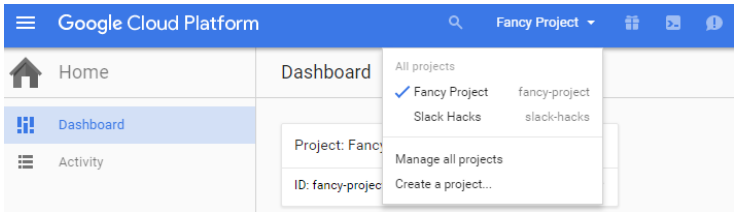


Figure 4-10. Project ID

Next you need to install the Go language App Engine SDK from *https://cloud.google.com/appengine/downloads*. You can check whether your SDK is correctly installed by typing `goapp version`.

Configure the slash command in Slack

Now let's configure a Slash command in Slack to handle our little calculator using techniques similar to the ones we used in Hacks #22 and #23. Enter the following in the Integrations Setting screen (Figure 4-11):

- Command: /calc.
- URL: *https://fizz-buzz-314.appspot.com/*
- Method: POST
- Token: This will be generated for you
- Customize Name: calculator
- Autocomplete help text: Select /feedback, turn on the "Show this command in the autocomplete list" checkbox.

- For the autocomplete description, we used “Calculate something” and our usage hint is “/calc 2+3*(5-7)”.

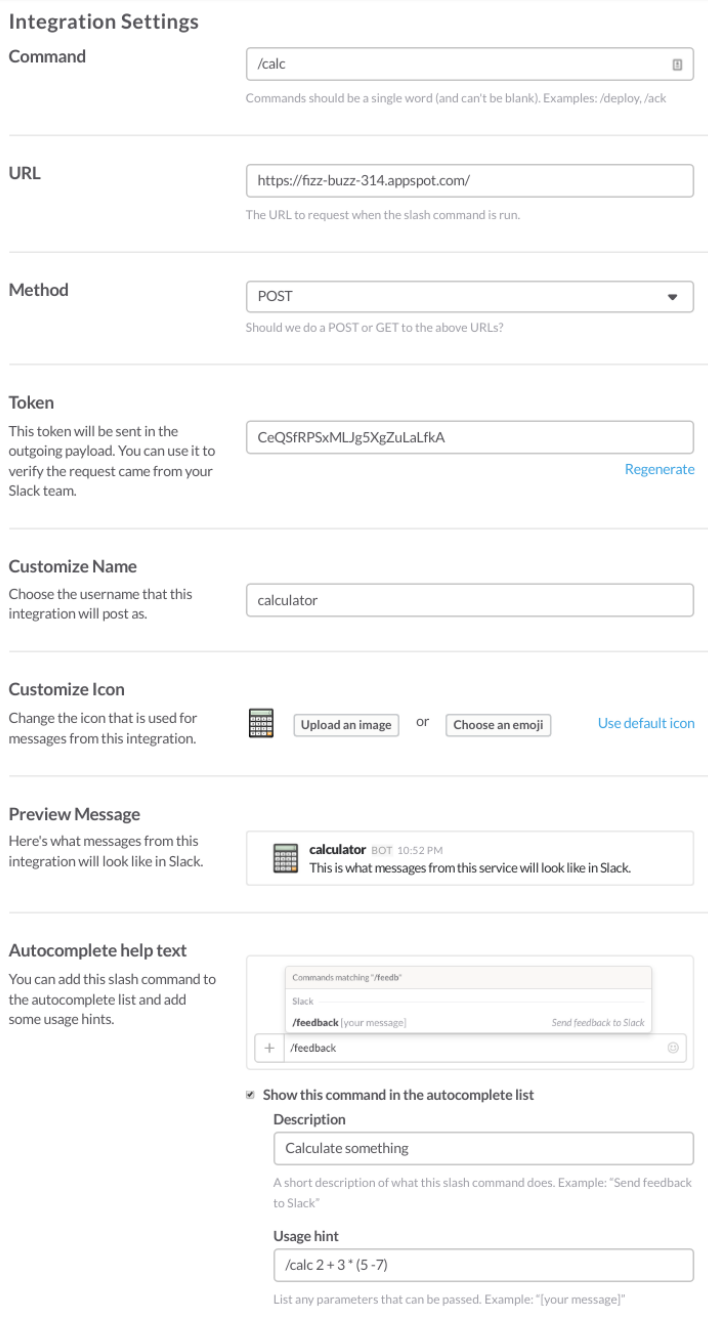


Figure 4-11. Configuration of the slack /calc command

Code and structure

You need to create the following files and directory structure:

```
├── app.yaml
└── calc
    └── cmd.go
```

The `app.yaml` file is an App Engine descriptor that sets up the runtime and serving parameters for your application. You can find the official documentation for this file here: *https://cloud.google.com/appengine/docs/go/config/appref*.

Enter the following code in app.yaml:

```
application: slashcalc
version: 1
runtime: go
api_version: go1

handlers:
- url: /.*
  script: _go_app
```

Next, we are going to define the logic of our command. The entry point is `handler`, which will be called every time an HTTP request lands on your application. The slash command will post a form with parameters like "text" that we will use to get the parameters the user gave to the command in the Slack chat box. The remaining code is pretty standard with some formatting and error handling.

HACK 04 **Security**

> In order to keep the example simple, it doesn't deal with any security. For example, we recommend that you match the security token you define in your Slack command from the `handler` function to be sure the request comes from Slack and not any third party on the Internet.

Here's the Go code of the calculator we will deploy on Google App Engine:

```
package calc

import (
```

```
        "fmt"
        "github.com/ryandao/go-calculator/lib"
        "net/http"
)

func init() {
        http.HandleFunc("/", handler)
}

const PAYLOAD string = `
{
  "response_type": "in_channel",
  "text": "= %s"
}`

// Strip trailing zeros from a float
func formatFloat(num float64) string {
        str := fmt.Sprintf("%.9f", num)
        truncate := len(str)

        for i := len(str) - 1; i >= 0; i-- {
                if str[i] == '0' {
                        truncate = i
                }
        }

        if truncate > 0 && str[truncate-1] == '.' {
                truncate--
        }

        return str[0:truncate]
}

func handler(w http.ResponseWriter, r *http.Request) {
        args := r.FormValue("text")
        lexer := lib.Lexer(args)
        interpreter := lib.Interpreter(&lexer)
        result, err := interpreter.Result()
        if err == nil {
                w.Header().Set("Content-Type", "application/json")
                fmt.Fprintf(w, PAYLOAD, formatFloat(result))
        } else {
                fmt.Fprintf(w, "ERROR %s [%s]", err, args)
        }
}
```

For this example we used a pre-built calculator parser library that you can find at *http://github.com/ryandao/go-calculator*. In order to install this library so App Engine can deploy it, you simply run the following command:

```
$ goapp get github.com/ryandao/go-calculator/lib
```

Testing locally

Before deploying your application, you can use App Engine's devappserver feature to test it locally.

To start a little local server with your code deployed on it, you can use `goapp serve` from the root of your application (where app.yaml is):

```
$ goapp serve
INFO     2016-06-07 06:08:40,109 devappserver2.py:769]
       Skipping SDK update check.
INFO     2016-06-07 06:08:40,173 api_server.py:205]
       Starting API server at: http://localhost:45511
INFO     2016-06-07 06:08:40,175 dispatcher.py:197]
       Starting module "default" running at: http://localhost:8080
INFO     2016-06-07 06:08:40,176 admin_server.py:116]
       Starting admin server at: http://localhost:8000
```

To test your application, give it an expression within the URL, such as `http://localhost:8080/?text=3*2` (see Figure 4-12).

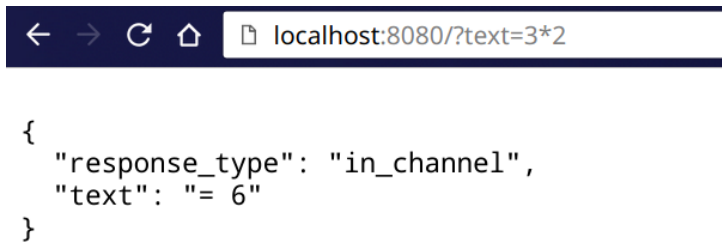


Figure 4-12. Testing the application locally

URL encoded format for + symbol

The text parameter needs to be URL encoded and + is a character used in the encoding grammar, so if you want to try out + you will need to use `%2B` instead. For example, for 2+3, use `http://localhost:8080/?text=3%2B2`.

Deploying on App Engine

This is where you'll need your project ID from the cloud console. We used `fizz-buzz-314` here, but you'll need to replace that with your actual ID. Use this command to deploy the application:

```
$ goapp deploy -application fizz-buzz-314 app.yaml
[...]
```

Once it is deployed, you can test the application independently from Slack with `https://fizz-buzz-314.appspot.com/?text=3*2`. Notice that the URL includes both `https` and the name of your project

You can then try your application in Slack (see Figure 4-13).

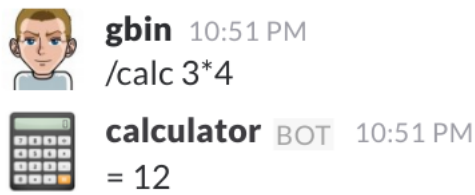


Figure 4-13. Result of the /calc command

You can use this example as a base for your own project or by analogy for any other language available on the App Engine (Java, PHP, Python).

HACK 05 Host a poker bot with Heroku

Heroku is an easy to use platform-as-a-service, and hosting a Slack bot on it is really simple. As an example, in this hack we will see how to use it to run the poker bot from *https://github.com/CharlieHess/slack-poker-bot*. This hack requires a Heroku Slack bot using Node.js.

Create a bot integration

First, create a bot user for your team via `https://TEAM.slack.com/services/new/bot`, where TEAM is the name of your Slack team.

Then fill in the API Token, Customize Name, and optionally upload a Customize Icon field on that web page (see Figure 4-14).

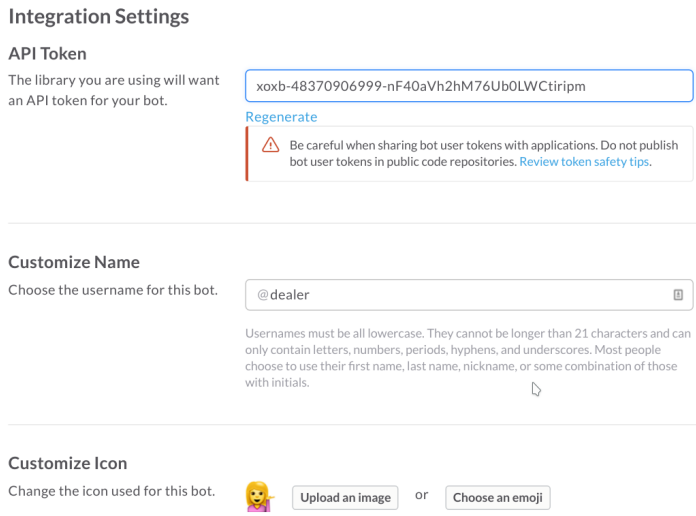


Figure 4-14. Configuring the custom Slack integration for our poker bot.

Install a local Heroku environment

Next, follow the toolbelt installation instructions at *https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up*. In a nutshell, if you want to install the toolbelt on your account on Linux do the following:

```
$ cd
$ wget https://s3.amazonaws.com/assets.heroku.com/\
        heroku-client/heroku-client.tgz
$ taz xzvf heroku-client.tgz
 [...]
$ mv heroku-client heroku
```

Then add the `~/heroku/bin` to your latexmath:[$PATH environment variable in your /.bashrc or /.zshrc by adding the line `export PATH=$]\{HOME}/heroku/bin:$PATH`.

You can then use the following command to login to your Heroku account (if you don't already have one, create one on heroku.com):

```
$ heroku login
heroku-cli: Installing CLI... 22.7MB/22.7MBB
```

```
Enter your Heroku credentials.
Email: gbin@domain.tld
Password (typing will be hidden):
Logged in as gbin@domain.tld
```

Deploy the project on Heroku

First, clone the code repository:

```
$ git clone https://github.com/CharlieHess/slack-poker-bot
$ cd slack-poker-bot
```

Then create an Heroku App from the root of this Git repository:

```
$ heroku create
Creating app... done, ⬢ mighty-tundra-99867
https://mighty-tundra-99867.herokuapp.com/ | \
              https://git.heroku.com/mighty-tundra-99867.git
```

If you run `git remote -v`, you'll see that `heroku create` effectively added a new remote to your local Git clone:

```
$ git remote -v
[...]
heroku  https://git.heroku.com/mighty-tundra-99867.git (fetch)
heroku  https://git.heroku.com/mighty-tundra-99867.git (push)
[...]
```

Then set your bot's API token:

```
$ heroku config:set SLACK_POKER_BOT_TOKEN=\
       xoxb-48370906999-nF40aVh2hM76Ub0LWCtiripm
Setting SLACK_POKER_BOT_TOKEN and restarting ⬢ \
       mighty-tundra-99867... done, v3
SLACK_POKER_BOT_TOKEN: xoxb-48370906999-nF40aVh2hM76Ub0LWCtiripm
```

You can now deploy your bot like so (note that this may take a while).

```
$ git push heroku master
Counting objects: 1211, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (465/465), done.
Writing objects: 100% (1211/1211), 5.43 MiB | 88.00 KiB/s, done.
Total 1211 (delta 733), reused 1211 (delta 733)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: ----> Node.js app detected
remote:
remote: ----> Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
```

```
remote:        NPM_CONFIG_PRODUCTION=true
remote:        NODE_ENV=production
remote:        NODE_MODULES_CACHE=true
[...]
remote: Verifying deploy.... done.
To https://git.heroku.com/mighty-tundra-99867.git
 * [new branch]      master -> master
```

Invite and use the bot

At the top right of your Slack window, click the gear icon and choose "Invite team members to join…" (see Figure 4-15).

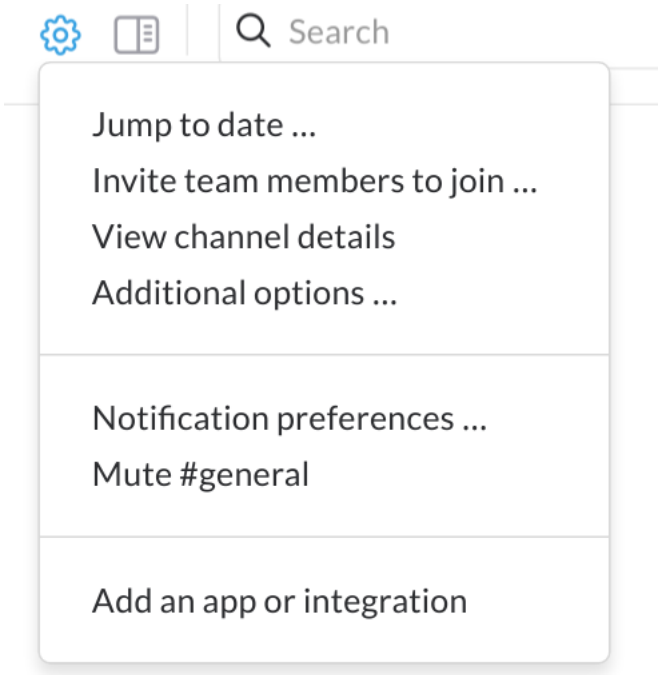


Figure 4-15. Invite the Heroku bot in your channel.

Select the newly created team member that should be online by then (see Figure 4-16).

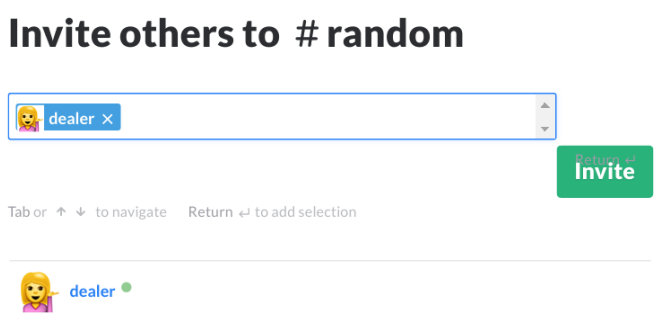


Figure 4-16. Invite the dealer

You can now ask the dealer (or whatever name you gave it) to deal by typing `@dealer: deal`; after that, all of the players wanting to play need to respond `yes`. Once they've done so, the game can get started, and the bot will message you in private about your hand (see Figure 4-17).

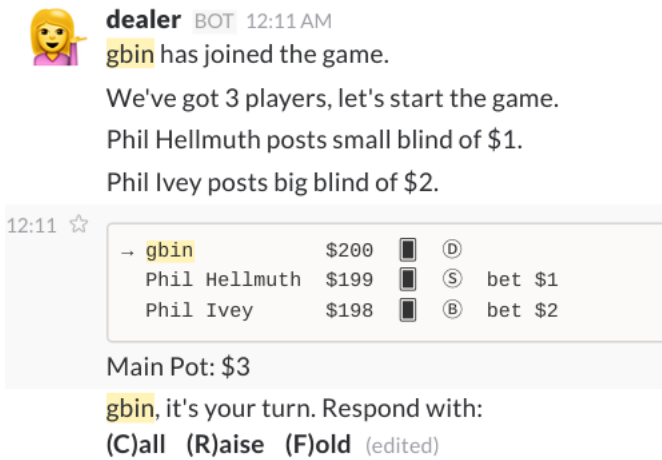


Figure 4-17. Heroku: a poker game

Later on in the game you will see something like Figure 4-18.

Figure 4-18. Later in the game

You may now have some ideas for creating and hosting your own games on Heroku!

HACK 06 Create a bot to randomly select someone for a task

In this hack, we'll build a PHP script which, when called, will pick a random person who is both a member of the Slack channel from which it was called, and and is currently online. The purpose of this particular script is to randomly select someone to make tea, but the same tool can be used to distribute other tasks around the office.

In this hack you'll learn the basic mechanism for triggering scripts and targeting specific members of a channel. Once you know how to do that, you can let your imagination run free and create all sorts of solutions.

HACK 07 **Hosting this hack**

> Consider hosting your PHP script on Google App Engine (simply follow the instructions in Hack #24 but use PHP instead of Go) or on Heroku (as described in Hack #25).

The steps for this script, which can be found on Github, are:

1. Trigger from a channel
2. Get the members in that channel
3. Filter out the offline members
4. Filter out any from the exclusion list
5. Select a random online team member for tea making

The first step is to get Slack to call your script and to return something. To do this, you need to add an outgoing webhook to your team.

Add an Outgoing Webhook for the !tea command

1. Go to the Outgoing Webhooks page and click *"install for your team"*.
2. Click *"Add Outgoing Webhooks Integration"*.
3. Fill in the fields as follows:
   - Channel: Any (or your dedicated Tea channel).
   - Trigger Word(s): !tea.
   - URL(s): The URL where your script will be (make sure it is a PHP file and publicly accessible).
   - Descriptive Label: Whatever you want.
   - Customize Name: Tea Bot.
   - Customize Icon: Click *"Choose an emoji"* and select the tea icon
4. Click Save.

It Started with Hello

Now that you've created the Outgoing Webhook, verify that the script is working. Outgoing Webhooks print whatever the script returns, as long as it is valid JSON that matches what is expected. The string to send to the Outgoing Webhook has to match the following JSON syntax in order to have the text successfully printed in the channel:

```
{
  "text": "text to print"
}
```

So, here's the PHP script for creating such a JSON string to print *Hello world*:

```
<?php

    header('Content-Type: application/json');
    echo json_encode(array(
        'text' => 'Hello world'
    ));
```

Using the trigger word we set up earlier (`!tea`), the above script will print `Hello World` in the channel that it was called from by your named bot.

> This script has to be called from a public channel—it doesn't work private channels or conversations.

Slack payload data

When using Outgoing Webhooks, Slack sends several bits of useful information to your page in the payload using some key/value pairs: * `channel_name`: the name of the channel from which the outgoing webhook was triggered * `channel_id`: the id of this same channel * `user_name`: the Slack username of the person who triggered the Outgoing Webhook * `user_id`: this same user's Slack ID * `text`: the text line which triggered the Outgoing Webhook (including the trigger word)

Along with the trigger word, you can pass additional text to your script. Returning to the tea example, you can ask for a specific tea

flavor by passing the flavor after the trigger word by running `!tea Earl Grey` (if you set **!tea** as your trigger word, that is).

This PHP script extracts the payload data passed after the keyword `!tea` for further use to build the bot response:

```
// Specify your trigger word
$trigger_word = '!tea';

// Remove our keyword and proceeding space from the
// text to extract name to exclude
$text = str_replace($trigger_word . ' ', '', $_POST['text']);
```

Using this script, you should now have all of the information that was passed after `!tea`. A simple example would be to echo what the user entered. Here's a complete example echoing what a user is passing after a keyword:

```
<?php
    // Specify your trigger word
    $trigger_word = '!echo';

    // Remove our keyword from the text to extract name to exclude
    $text = str_replace($trigger_word . ' ', '', $_POST['text']);


    // Return a random response
    header('Content-Type: application/json');
    echo json_encode(array(
        'text' => $text
    ));
```

In the example above, the user types:

> !echo hello everybody

And the bot replies

> hello everybody

> Text passed to the script can be used as a parameter to finely control what the script output will be. In the tea maker example, you can pass names to exclude from the next round of tea duty (very helpful for getting out of the next round).

From here, the possibilities of what your script can do are endless. Next up, we will connect to the Slack API, get channel and user data, and ultimately pick a tea maker!

Access Slack data using the API and Outgoing Webhooks

Now it is time to use the Slack PHP API to access channel and user data. Using the API we're going to connect to Slack, work out who is in a channel, who is online, and who should make the tea.

To do this, you can communicate directly with the API, but to make it even easier, let's use this Slack API PHP Wrapper. You'll need to download the script and place it in the same folder as your webhook script.

Before proceeding, you must generate an **oAuth token**. This allows Slack to verify that we are in fact the ones communicating with it, and that we're allowed to do that. If you are a member of multiple teams, then you'll need multiple oAuth tokens. To generate one, follow these steps:

1. Go to *https://api.slack.com/web*.
2. Scroll down to *"Generate test tokens"*.
3. If you have previously generated a token, there will be one near the end of the *"Test token generator"* section, with the team name on the left. If not, click the *"Create token"* button.
4. Make a note of the token.

What Channel are you on?

Slack sends a lot of information when triggering a script. One piece of this is the ID of the channel it was triggered from. This is stored in the `$_POST['channel_id]` key.

Here's how to connect to the Slack Web API, get the requested channel information and display it:

```
<?php

    // Replace AUTH TOKEN with your auth token
    $auth_token = 'AUTH TOKEN';
    // Include slack library from
    // https://github.com/10w042/slack-api
    include 'Slack.php';

    // Connect to Slack
    // Use authentication token found here: https://api.slack.com/
```

```
$Slack = new Slack($auth_token);

// Get the info for the channel requested from
$data = $Slack->call('channels.info', array(
    'channel' => $_POST['channel_id']
));

header('Content-Type: application/json');
echo json_encode(array(
    'text' => json_encode($data)
));
```

This code snippet connects to the Slack API using the library mentioned previously (using the `Slack()` class). Make sure your `include` path is correct.

The next part(following the definition of `$Slack` variable) gets the channel info using the `channel_id` found in the `$_POST` information. `$data` now contains all of the information about the current channel. This info is then passed back to Slack to post.

In JSON data returned by Slack when calling the `channels.info` method, you should notice channel information such as the `latest` event, the channel `purpose`, and the `created` timestamp. The key we're interested in is `members`, which lists the IDs for every member in the channel.

Picking an active user

To select a random user to prepare the tea, we need to select one among all the users that are active in the channel the command was sent on.

Through its Web API, Slack provides a method for getting information about a channel: `channels.info`. (You can see a detailed description of this method on the official Slack Web API documentation). Among this information is the channel members, so you can call it to get a list of the members' IDs. To convert IDs to names, use another Web API method that gives you information about a user from an ID: `users.info`. From this call we will be able to get the user name in order to determine who should be excluded from the tea making duty.

Finally, we will need to determine each user's presence by using the `users.getPresence` Slack API method and filter out the inactive users. The active member users will then be added to a new `$teaMak`

ers array. From this array, a random team member can then be picked to make the tea using the `pickOne` function whose purpose is to pick a random element from an array of elements.

Here's the script that performs all of these tasks:

```
// Get the info for the channel requested from
$data = $Slack->call('channels.info',
  array('channel' => $_POST['channel_id']));

$teaMakers = array();

// Loop through channel members
foreach ($data['channel']['members'] as $m) {
    // Get user data
    $userData = $Slack->call('users.info', array('user' => $m));
    // Check to see whether the user is online before adding
    // them to list of brewers
    $presence = $Slack->call('users.getPresence',
      array('user' => $m));
    $user = $userData['user'];

    // If there is an exclude, check to see if it matches
    // a user's real name (lowercase)
    // If it doesn't, add it to the $teaMakers array
    if($presence['presence'] == 'active')
        $teaMakers[] = $user;
}

function pickOne($array) {
    shuffle($array);
    return $array[rand(0, (count($array) - 1))];
}

$user = pickOne($teaMakers);

header('Content-Type: application/json');
echo json_encode(array(
    'text' => $user['id']
));
```

This script returns a user ID, which isn't very helpful to anyone. To fix that, instead of returning `$user['id']` in `json_encode`, you can return `$user['name']`. This tells you whose turn it is in human readable form.

The other option is to return the ID as a link to the user. This approach then, not only returns the username, but also notifies the user of their duty.

To build a link to a user in Slack, add a `<@` in front of the `$user['id']` and a `>` after it; for example:

```
header('Content-Type: application/json');
echo json_encode(array(
    'text' => '<@' . $user['name'] . '>'
));
```

Allow exclusions

The last step in building the tea bot is to allow exclusions to be passed in. Using the `$_POST['text']` key, you can exclude the trigger word and exclude the correct users.

Once you've isolated the exclusion text, you need to check to see whether the user matches the exclusion string; here's the code that does that:

```
if($presence['presence'] == 'active')
    if($exclude) {
        if(!(strpos(strtolower($user['real_name']),
          strtolower($exclude)) !== false))
            $teaMakers[] = $user;
    } else {
        $teaMakers[] = $user;
    }
```

If the exclusion string is present, but the user *doesn't* match the exclusion string, you still add them to the `teaMakers` array. You also convert everything to lower case when comparing, to remove any capital discrepancies.

Your script should now pick a random, online member to get up and put the kettle on. Remember, everyone can see what you type and input into the webhook, so you won't be able to get away with always excluding yourself!

Add a greeting to the tea bot

Now that the script is complete, you can add a nice greeting.

Being able to pick a random index from an array lets the bot make decisions; this will allow it to pick both the user and the response to use.

The following PHP script is the part of the tea bot that specifies an `array` at the top of the script and then, using a custom function, picks a random response:

```
<?php

    // Specify an array of responses
    $responses = array(
        'Hello!',
        'How are you?'
        'How is your day going?'
        'What do you call a chicken on roller skates?
        'Poultry in motion!'
        '
    );

    // Random picking function
    function pickOne($array) {
        shuffle($array);
        return $array[rand(0, (count($array) - 1))];
    }

    // Return a random response
    header('Content-Type: application/json');
    echo json_encode(array(
        'text' => pickOne($responses)
    ));
```

This code snippet specifies responses in a simple PHP array, which is used for our tea bot (see this book's Github repo for the bot's full source code).

Add some personality to the tea bot

An extra, optional step is to give your tea bot a little bit of personality. The following bit of code creates an array of random responses with a `{{USER}}` marker. Upon picking a victim, the tea bot uses this code to choose a response and substitute the marker with the selected marker.

First, create your array. For each entry, make sure the marker `{{USER}}` is present. Then assign it to the variable `$responses`, like so:

```
$responses = array(
        "It's about time {{USER}} put the kettle
          on - off you trot!",
        "Pop the kettle on {{USER}} - it's your turn to
          make a cuppa",
        "Who wants a drink? {{USER}} is heading to the kitchen
          to make one",
        "Coffee? Tea? Sugar? Peppermint Tea? Green Tea? Get
          your orders in as {{USER}} is making a round",
```

```
            "That's very nice of {{USER}} to make a round of tea!",
            "Mine is milk 2 sugars please {{USER}} - what about
              everyone else?",
            "The tea maker is... {{USER}}! Get brewing."
        );
```

From this array, we can use the `pickOne` function to choose a random response, replace the marker with the actual user, and return the response using `str_replace`.

`str_replace` is a built in PHP function that has three parameters: `str_replace($what_to_search_for, $what_to_replace_it_with, and $what_to_search_in)`. More details can be found on the PHP website.

Here's the script choosing a random user and substituting `{{USER}}` for the designated user (the tea bot will now be able to share random responses in Slack):

```
// Get a random user from the array
$user = pickOne($teaMakers);

// SEND OUT THE JSON!! Enjoy your brew
header('Content-Type: application/json');
echo json_encode(array(
    'text' => str_replace('{{USER}}', '<@' . $user['id'] .
      '>', pickOne($responses))
));
```

You have now successfully created your tea bot. Enjoy!

HACK 08 Create a Slash command for posting anonymous messages to a channel

In this hack you will learn how to implement a slash command in Golang and deploy it using Docker. When you activate the slash command `/anon Who am I ?`, a message will be posted to a pre-selected channel. The message will have a randomly selected avatar along with a quote. The avatar makes the user who sent the slash command anonymous since his message is posted without his name and avatar. This hack can be a source of fun if several people use this slash command, since multiple avatars can be created by multiple Slack users

to send random messages, as detailed in the Loading the avatars section below (see Figure 4-19).

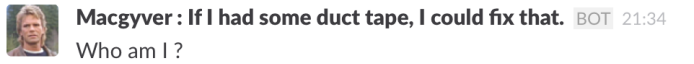


Figure 4-19. INSERT FIGCAPTION HERE

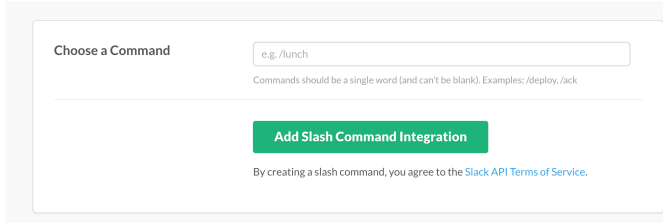
Figure 4-19 shows the results of an avatar and messsage triggered by an anonymous Slack user. This hack is purely for fun, although as you will see, there's a lot going on behind the scenes.

The source code for this slash command can be found on Github.

Add a Slash command

To create this slash command:

1. Go to the Slash Commands page and click Add Configuration.
2. Click *"Add Outgoing Webhooks Integration"*.
3. On the page that appears, fill in the command name, e.g. `/anon`, and then click *"Add Slash Command Integration"* (see Figure).



4. Leave the URL field empty.
5. Set the method to POST.
6. Write down the content of the Token field.
7. In the Description field write *"Send anonymous messages"*.

8. Click *"Save Integration"*.
9. Go to the Incoming webhooks page.
10. Click *"Add Configuration"* and copy the webhook URL (see Figure 4-20)

    +

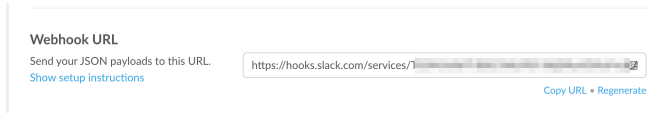


Figure 4-20. Setting up a webhook integration

11. Click *"Save Settings"*.

Setting up a server

In order for Slack to respond to your new slash command, you need to set up web servers that will accept the request from Slack after someone enters the command. In Go, each source file can define its own `init` function to set up whatever state is required. (Each file can actually have multiple `init` functions.) This means `init` is called after all of the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized. In our `init` function we will set the HTTP server port to 5000.

In the main function, we initialze a random number generator (we will use it later in order to select an avatar). We will get from the environment two variables:

- The channel that we would like to post to.
- The Slack token in order to validate requests.

The next step is to load the avatars from a JSON file. Once the loading is complete we will set up a handler function (readAnonymousMessage) for the `/` path, and then the HTTP server will start.

Here's the Go code that performs the steps we just described:

```
func main() {
    rand.Seed(time.Now().UnixNano())
    channel = os.Getenv("SLACK_CHANNEL_ID")
```

```
    token = os.Getenv(tokenConfig)
    getAvatars()
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        result := readAnonymousMessage(r)
        fmt.Fprintf(w, result)
    })
    http.ListenAndServe(fmt.Sprintf(":%d", port), nil)
}

func init() {
  //Init the web server port 5000
    flag.IntVar(&port, "port", 5000, "HTTP server port")
    flag.Parse()
}
```

Loading the Avatars

Avatars are loaded from a JSON file, `avatars.json`, which is located at the same directory as our binary. We have defined The JSON object structure for an avatar as follows:

- username - The user from which the message will be sent
- default_text - A text that appears next to the avatar's name
- icon_url - The user avatar url

Here's an example for a given avatar using this JSON structure:

```
[
  {
    "username": "Archer",
    "default_text": "Phrasing!",
    "icon_url": "http://i.imgur.com/CyIgnqi.png"
  }
]
```

The following code does two things:

1. It reads the JSON file.
2. It then populates three global arrays with the data from the file.

```
func getAvatars() []Avatar {
    raw, err := ioutil.ReadFile("./avatars.json")
    if err != nil {
        fmt.Println(err.Error())
        os.Exit(1)
    }

    var c []Avatar
```

```
        json.Unmarshal(raw, &c)
        for _, p := range c {
            avatars = append(avatars, p.Username)
            avatarIcon = append(avatarIcon, p.IconURL)
            avatarText = append(avatarText, p.DefaultText)

        }
        return c
    }
```

Getting a message from Slack

When users issue the `/anon` slash command, the Slack server will perform a POST request to our server. We will handle this request in the `readAnonymousMessage` function.

First, we will try to parse the data that we received. We will compare the token that we received to the one from the environment variable in order to make sure that the request is legit. The next validation is to make sure that there is an actual text to send. Finally, we will trim spaces and send an anonymous message.

Here's the code that performs these tasks:

```
    func readAnonymousMessage(r *http.Request) string {
      err := r.ParseForm()
      if err != nil {
          return string(err.Error())
      }
      if len(r.Form[keyToken]) == 0 || r.Form[keyToken][0] != token {
          return "Config error."
      }
      if len(r.Form[keyText]) == 0 {
          return "Slack bug; inform the team."
      }
      msg := strings.TrimSpace(r.Form[keyText][0])

      err = sendAnonymousMessage(msg)
      if err != nil {
          return "Failed to send message."
      }
      return fmt.Sprintf("Anonymously sent [%s] ", msg)
    }
```

Sending the anonymous message

To send the anonymous message, the code has to perform these steps:

1. Get the Slack webhook from the environment variable using `os.Getenv` function.
2. Generate a random avatar ID using the `rand.Intn` function.
3. Construct the message payload in JSON. The JSON structure for posting a message on a channel is available on the official Slack API documentation.

Here's the Go code that performs these steps:

```
func sendAnonymousMessage(message string) error {
    url := os.Getenv(webhookConfig)
    avatarID := rand.Intn(len(avatars))
    payload, err := json.Marshal(slackMsg{
        Text:      message,
        Channel:   channel,
        AsUser:    "False",
        IconURL:   avatarIcon[avatarID],
        LinkNames: "1",
        Username:  fmt.Sprintf("%s : %s", avatars[avatarID],
                               avatarText[avatarID]),
    })
    if err != nil {
        return err
    }
    _, err = http.Post(url, "application/json", bytes.NewBuffer(payload))
    return err
}
```

We are now ready to send the anonymous message via the webhook URL.

Deploying

Now, you can run the server inside a Docker container that can be deployed to your favorite cloud provider. A prebuilt binary Docker image can be found at Docker Hub. You can pull it by running the following command: `docker pull rounds/10m-annona`.

Here is the `Dockerfile` used to build this Docker image:

```
FROM rounds/10m-build-go
RUN apt-get update && \
  apt-get install  wget

RUN go get github.com/rounds/annona

EXPOSE 5000
```

```
RUN wget -O /root/go/bin/avatars.json \
    https://raw.githubusercontent.com/rounds/annona/master/avatars.json

CMD cd /root/go/bin/ && ./annona
```

The preceding code starts with a minimal base image. Then, we make sure that all packages are up to date and we use the standard go toolchain to fetch and build our annona binary with `go get`. After downloading the necessary avatars file using `wget`, we finally run the binary.

You now can run Docker and pass in the following three environment variables: - INCOMING_SLACK_WEBHOOK - SLACK_CHANNEL_ID - INCOMING_SLACK_TOKEN

Here's the command line to do so :

```
docker run -e INCOMING_SLACK_WEBHOOK='<your webhook URL>' \
           -e SLACK_CHANNEL_ID='<channel id where to post messages>' \
           -e INCOMING_SLACK_TOKEN='<Slack token obtained when cnfiguing the slas
           -p 5000\
           <your_image_name>
```

Make sure to map the Docker port (5000) to a port at the host with the `-p` flag to Docker.

After you deploy, make sure you go back to the URL at the slash command configuration page, and fill in your server address and port.

You have now seen some simple, custom-made applications you can integrate into Slack. In the next chapter, we cover how to integrate Slack into your business.

5

Slack for Teams

Slack is a great tool for improving communication and increasing transparency in your business, organization, club, or group of friends. Slack reduces internal emails, allowing you to convert your jumbled inbox into channels that contain all your crucial information, in a more organized and easy-to-find manner. In other words, Slack can help you separate important work messages from office chatter. It also provides quick communication via instant messages to teammates, which you can't do via email.

But Slack is much more than an email replacement; it can also be a central hub for all of your important services. It can be the one platform from which you access everything else in your daily workflow. With apps such as Trello, Google Calendar and so many others, it's a great place for your team to keep up-to-date with all aspects of your business.

Source code

Remember, the source code for all hacks in this chapter can be found in this book's Github repo.

This chapter provides a walkthrough of how you can integrate Slack into your business. There are also a couple of hacks that are relevant to using Slack in the workplace.

Before setting up and installing Slack for your team, make sure you know *why* you want to integrate it into your company or organization. Is it to reduce the volume of emails? Is it to increase communication between internal and remote workers? Is it to allow your team to see notifications from all of your services in one place? Whatever your reasoning, ensure you know exactly what problem you want to solve.

Once you've done that, ensure that everyone in the team is aware of what you're trying to accomplish. Doing this allows people to be invested in solving the problem and to understand why they are being introduced to this new platform.

You can't force a workflow on people who aren't invested. If Slack doesn't solve the problem you identified, you may have to look for an alternative solution to your problem.

Making Your Business Slack Ready

Getting a team invested in a new technology is always difficult. People tend to be divided into two camps: those that are eager to try new things and those that are skeptical about changing. Convincing the former is easy, but convincing the latter takes time.

If you've decided on Slack as your tool of choice (and a good choice it is) there are a few tips you can follow to make sure company rollout goes smoothly.

- **Get it configured correctly** - Take a few moments to go through the configuration options for your team. These can be accessed by clicking your team name in the top left of the Slack interface and then choosing the "Team Settings" option. For example, you might want to prevent some team members from adding integrations and apps, or you may wish to restrict the registration email to a specific domain.
- **Invite some beta testers** - Use your fellow team members (or close colleagues) to try out the service. Slack recommends designating a day for this purpose.
- **Prohibit email** - specify a trial period (for example a day or two) where all internal communications and questions must be sent via Slack, not email. This may seem a little over the top, but

it gets people used to using the platform. Once people understand how Slack works and what it can be used for, they will know when to use Slack and when to revert to email.

- **Designate admins** - Make sure the right people are admins. Pre-determine who should be in charge of advanced options and features before inviting everyone.
- **Roll it out gradually** - If your beta users are happy and embracing Slack, roll it out team by team - making sure you properly introduce the platform to them first - explaining how you expect them to use it and why they should. Just sending out an invite and expecting people to work it out themselves could upset people. Consider running a small presentation for each team to help them get used to Slack.
- **Make messages meaningful** - Slack doesn't just let you send simple messages - take a moment to read over the message formatting tips, and pass them on to other them members to help emphasize certain words or separate out code blocks. You can find these tips in the Slack Help Center.
- **Remember that Slack allows more than just words** - Think about installing some simple apps to help your team get their messages across. For example, Giphy allows Gifs to be sent with a simple keyword, and Growbot allows you to easily give praise to colleagues.

Once your company is using Slack, you're ready to start dividing the conversation and keeping chats on-topic using channels.

Channels

Slack uses channels to organize conversations and topics into separate rooms. Channel names and conversations can be seen by everyone on your organization's team but to be notified or join in the conversation you need to join the channel. This allows team members to be involved only in conversations they want or need to be in.

When creating a Slack group, the temptation is to make lots of channels for every eventuality you can think of. The problem with this approach is people can get confused and be unsure where to post conversations, and conversations can get diluted.

Start off small, with only two or three channels, and only create extra ones when they're requested and needed. For example, a

`#development` channel might be required for your development team, and if the `#general` channel gets overwhelmed with talks of last night's TV episodes, consider making an `#entertainment` channel.

If you are creating project- or team-specific channels, pick a naming convention and stick to it. For example, all project channels could start with `#prj-` and team channels could have a `#t-` prefix. This helps members identify the channel type, and also groups related channels together in the channel list.

Security

With an Instant Messenger at your fingertips, it's very easy to use it to share private information between colleagues, such as passwords or login details. But keep in mind that anyone can potentially see and access the information you share on Slack (this includes any apps or integrations installed for the team). So think twice before sharing sensitive information.

You can also take steps to improve the security of Slack itself. As the team owner, you can force your team to reset their passwords. Do this by visiting:

Team Settings → Authentication → Forced Password Reset.

Now we're ready to dive into some team hacks. The source code for all hacks in this chapter can be found in this book's Github repo.

HACK 01 Email a reminder of a conversation

> Although Slack can replace a lot of your need for email, there are some situations where you really need email, so here is a hack to help you integrate the two. For example, say someone outside of Slack asks you to perform a task, and you want to keep track of it in your inbox.

In this hack, you will create an HTML email that contains the last 10 Slack messages sent in a channel. Since this uses an Outgoing Webhook (see Chapter 2, *Slack Apps and Simple Webhooks*), it will only work in a public channel, and not a private conversation. This script will work by the user going to the relevant Slack channel and typing

`emailme` followed by a subject line such as `emailme Fix the web site bug`.

This task could be accomplished using a slash command instead, but Outgoing Webhooks were chosen for this hack to provide transparency—webhooks lets other Slack users to see that someone has requested an email transcript of the conversation.

Before you get started, create a `PHP` file on a web-accessible server called `email.php`. This is the file we are going to trigger and use to send the email.

Create the Hook

After you've created and saved your PHP file, the next step is to set up an Outgoing Webhook. The settings below are just a guide (most of these settings can be overridden in the script itself):

- Channel: **Any**
- Trigger Word(s): **!emailme**
- URL(s): **The URL where your script is hosted**
- Descriptive Label: Describe what the bot is doing
- Customize Name: *Email Bot *
- Customize Icon: Choose emoji and search for "email"; click the envelope emoji to select it.

Obtain an API Token

The script in this hack uses the Slack API to gather all of the information it needs. In order to interact with the API, you need to generate an oAuth token; here's how:

1. Go to *https://api.slack.com/web*.
2. Scroll down to *"Generate test tokens"*.
3. If you have previously generated a token it will be listed in the table on this page. If not, click the *"Create token"* button.
4. Make a note of the token.

Script set up

With this hack, we are going to include the Slack PHP Library. This makes it easier to interact with the API, and saves you from having to write your own cURL request.

Download the Slack API PHP file from the GitHub repository linked to above, and place it in the same directory as the `email.php` script you created earlier.

Next, open the `email.php` script and paste in the following code. This code includes the Slack API library and initializes a new instance of the Slack class used later in this script.

```
<?php
    // Include slack library from https://github.com/10w042/slack-api
    include 'Slack.php';

    // Initialise new instance
    $Slack = new Slack('[API TOKEN]');
```

When Slack triggers a Webhook, it sends a payload containing information about where the Webhook was triggered from. This includes the `channel_id` of the channel and the `user_id` of the user who triggered it.

Using this payload data, which is sent as `$_POST` data to the PHP script, we can query the Slack API. This allows you to get the email address of the user who triggered the Webhook and the channel's history. This will be used for the contents of the email.

In the code below, the channel history is loaded, and the user details are stored into a `$me` variable. Both the channel and user data are obtained by querying the Slack API. Lastly, the text that accompanied the `emailme` code (if there was any) is stored in a variable.

Place this code after your Slack API initialization:

```
// Get the last 11 messages of the channel conversation
$data = $Slack->call('channels.history', array(
    'channel' => $_POST['channel_id'],
    'count' => 11
));

// Get the user who requested the email
$me = $Slack->call('users.info', array(
    'user' => $_POST['user_id']
));
```

```
// Strip "emailme" from the message to get the subject line
// (if one was entered)
$text = str_replace('emailme', '', $_POST['text']);
```

The first variable, `$data`, is an array containing the last 11 messages in this channel, ranging from newest to oldest. This includes the one that triggered this Webhook. If you change this number, make sure you add an extra message to allow for the trigger message.

The `$me` variable contains all of the data about the user, including their name, image, and (most importantly) email address.

Create a messages array

Using the array of messages that we retrieved from the channel history, we can create a new array containing the messages along with the names and pictures of the users who wrote them. (Currently, the messages array in the `$data` from the Slack API contains only a user ID.)

Insert this code at the end of your PHP script:

```
// Create empty array
$messages = array();

// Loop through channel messages
foreach($data['messages'] as $d) {

    // Check  If the message has a user - this excludes bot
    // messages from the email
    if($d['user']) {
        // Get the user associated with the message
        $userData = $Slack->call('users.info', array('user' \
                    => $d['user']));

        // Add to the messages array the name & image of the user,
        // plus the actual message
        $messages[] = array(
            'user' => $userData['user']['real_name'],
            'image' => $userData['user']['profile']['image_24'],
            'message' => trim($d['text'])
        );
    }
}

// Remove the first item from the array (the trigger message)
array_shift($messages);
```

```
// Reverse the messages so oldest is first
$messages = array_reverse($messages);
```

The code above loops through the existing messages array and, if a `user` key is present in the message, adds it to the `$messages` array. If the `user` key is missing, this indicates it's a message from a bot or other Webhook, which you can ignore.

Lastly, the trigger message gets removed (we don't need it in the email) and the array gets reversed. Reversing the array means that, when we output the messages in the email, they will read from oldest to newest.

Specifying the Recipient and Subject

The first step in creating the email is to specify the recipient. To be sure that the name and email appears correctly in the email client's *To* field, specify a name and an email address in this format: `Full Name <email@address.com>`.

Because we have data about the current user stored in the `$me` variable, we can easily construct the recipient. The subject line can also be set, either by taking the user input text, or by setting a default one.

To accomplish these two things, add this code after the `$messages` array described in the previous section:

```
// Create recipient
$to  = $me['user']['profile']['real_name_normalized'] . ' <' . \
        $me['user']['profile']['email'] . '>';

// Set the subject line
$subject = ($text) ? $text : 'Email reminder from Slack';
```

Creating the Email Headers

The next step is to set the headers for the email. The following code tells the email client how to handle the incoming email so it can be processed as HTML. It also sets the sender of the message. Add this code to the end of your PHP file:

```
// Set the headers & sender of the email
$headers  = 'MIME-Version: 1.0' . "\r\n";
$headers .= 'Content-type: text/html; charset=iso-8859-1' . "\r\n";
$headers .= 'From: Slackbot <slack@yourdomain.com>' . "\r\n";
```

Specifying the Messages

Now that you've specified the email headers, you need to create variables for both the name of the user and a string of all the messages in a `<table>` element, which will be inserted into the HTML template.

Insert this code after the headers have been set:

```
// Set the name for the email
$name = $me['user']['profile']['first_name'];

// Create the messages table
$table = '<table width="100%" style="border-collapse: collapse">';

// Loop through all the messages
foreach($messages as $m) {
    $table .= '<tr><td style="border: 1px solid #ccc; padding: 5px; \
               width: 24px;"><img src="' . $m['image'] . '" \
               style="display: block"></td><td style="border: 1px \
               solid #ccc; padding: 5px;"><b>' . $m['user'] . '</b> \
               </td><td style="border: 1px solid #ccc; padding: 5px;"> \
               <i>' . $m['message'] . '</i></td><tr>';
}

// Close the table
$table .= '</table>';
```

HTML Email

To build the HTML email, we are going to use the PHP Heredoc syntax. This allows you to construct HTML and insert PHP variables without needing to escape them.

The full HTML is too long to include in this book, but you can download it from this book's Github repository. Here's an example of Heredoc in use:

```
$message = <<<HTML
Hello {$name}.<br><br>
You requested an email of the conversation.<br><br>
{$table}
HTML;
```

Send the email

We can now use PHP to send the email since we have the recipient, subject, message, and headers. Add the following to your PHP file:

```
// Send the email!
mail($to, $subject, $message, $headers);
```

Notify the User via Slackbot

The last thing this script does is notify the user through Slackbot that the email has been sent.

With the Slack API, if you post a message to a channel that has the same ID as a user ID, it will appear in the Slackbot private conversation. We are going to use this ability to private-message a user to send a message informing them the email has been sent.

Add this to the end of the PHP file you've been working on:

```
// Post a message in the Slackbot channel
$sendMessage = $Slack->call('chat.postMessage', array(
    'channel' => $me['user']['id'],
    'text' => 'Yo ' . $me['user']['profile']['first_name'] . ' \
                - I\'ve sent you that email you asked for',
    'username' => 'Emailbot',
    'icon_emoji' => ':email:'
));
```

The Slackbot message appears in Slack as shown in Figure 5-1.

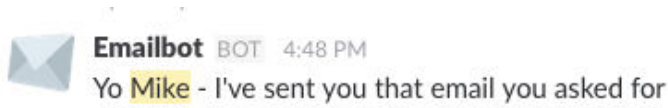


Figure 5-1. Email Bot

You now have an easy way to send recent Slack messages via email.

HACK 02 Celebrate unusual holidays

In this hack we will create a notification to inform you when there are unusual holidays. For example, did you know that the 20th of January is Penguin Awareness Day? Or that the 15th of September is Make A Hat Day?

Create an Incoming Webhook

This script uses Incoming Webhooks to post the message to Slack. Add an Incoming Webhook by visiting the Apps and Integrations app directory, searching for Incoming Webhooks and then clicking install. (See Chapter 2 for more information.) Finally, select a channel that you want the holiday messages to be posted to, and click **Add Incoming Webhook**.

Once installed, you will be shown a Webhook URL. Make a note of this URL, because you'll use it to post a message to your organization.

Collate your days

The next step is to build a list of days to shout about. This can be a personal collection of dates (birthdays and national holidays, say) or you can include unusual dates.

Create a PHP file called `dates.php` on a server. This is the file we'll be editing and adding the calendar to.

To build up a calendar, create 12 arrays inside an array, with numbers `01` → `12` as the keys, which will be the months. (For the sake of readability, make sure the numbers 1-9 are preceded with a `0`.) Inside each month, create a key for each day of that month. You can either do this yourself (using the example below) or you can simply download this template from Github.

Here's an example of what your finished arrays should look like:

```
$dates = array(
        // Jan
        '01' => array(
            '01' => 'New Years Day',
            '02' => '',
            '03' => '',
            '04' => '',
            '05' => '',
            ...
            '28' => '',
            '29' => '',
            '30' => '',
            '31' => ''
        ),
        // Feb
        '02' => array(
            '01' => '',
```

```
            '02' => '',
            '03' => '',
            '04' => '',
            // ... All the way up until december
            '25' => 'Christmas Day',
            '26' => '',
            '27' => '',
            '28' => ''
        )
);
```

Next, fill in the days you wish to celebrate. A pretty comprehensive collection of weird and wacky holidays can be found in this books's Github repository. (This list has been copied (and trimmed) from a very helpful website.) To keep the codebase clean, the dates have been saved in a separate file from the rest of the code and assigned to the `$dates` variable - this file will be included in our main PHP script in the next step.

The text input can include both Slack formatting (for example, surround the text in `*` to make the text bold) and emojis. The emojis can be either the text representation (`:cactus:`), or the actual symbol - both will be displayed correctly by Slack.

Now that you've created your list of dates, create a second PHP file titled `generate-day.php` in the same folder as `dates.php` and include your list of dates in it, like so:

```
<?php
    // Include the dates
    include './dates.php';
```

Get today's day

Now that you have a list of days, you need to retrieve info for the current date. To do that, create a new `DateTime()` object in your `generate-day` PHP file after the include. This will default to the current date if nothing is passed in.

```
// Get todays date
$today = new DateTime();
```

You can now query the `$today` object to get the current month and day with a leading zero to match the keys on the arrays.

```
// Get the current month with a leading zero
// echo $today->format('m')

// Get the current day with a leading zero
```

```
// echo $today->format('d')

// Get the special day!
$day = $dates[$today->format('m')][$today->format('d')];
```

Post to Slack

Before posting to Slack, let's ensure that there is something to post (you don't want to send an empty message). Add this to your `generate-day` PHP file:

```
// If today has a value
if(count($day) > 0) {
    // Post to Slack
}
```

The next step is to encode the data as JSON and assign it to the `payload` variable that Slack expects. We've chosen to title the bot `What "international" day is it today?` and have assigned it the `:rosette:` `icon_emoji` to add a little bit of celebration to the day. The text in the message is then simply the actual name of the day:

```
// Encode the data
$data = 'payload=' . json_encode(array(
    'username' => 'What "international" day is it today?',
    'icon_emoji' => ':rosette:',

    'text' => $day,
));
```

Finally, set the `$webhook_url` (to post to) and include the PHP cURL code.

```
// Your webhook URL
$webhook = '[WEBHOOK URL]';

// PHP cURL POST request
$ch = curl_init($webhook);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);

echo $result;
```

Your full `generate-day.php` file should look like the code below. Make sure this file is placed on a server that can run PHP scripts.

```php
<?php
    // Include the dates
    include './dates.php';

    // Get todays date
    $today = new DateTime();

    // Get the special day!
    $day = $dates[$today->format('m')][$today->format('d')];

    // If today has a value
    if(count($day) > 0) {

        // Encode the data
        $data = 'payload=' . json_encode(array(
            'username' => 'What "international" day is it today?',
            'icon_emoji' => ':rosette:',

            'text' => $day,
        ));

        // Your webhook URL
        $webhook = '[WEBHOOK URL]';

        // PHP cURL POST request
        $ch = curl_init($webhook);
        curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
        curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        $result = curl_exec($ch);
        curl_close($ch);

        echo $result;
    }
```

Try out the script by navigating to the URL of your script. The web page should say "ok" and your message should be posted to Slack.

Trigger the Script Daily

The last step of this hack is to trigger the script on a daily basis using a cronjob. This can be done either through a control panel (for example cPanel) if you have one, or through the command line. We explain both approaches in the following sections.

Control Panel

The following instructions are for setting up a cronjob using cPanel —other control panels may be slightly different.

Log into your cPanel and navigate to "Cron Jobs". You should be presented with a series of drop down options. If there is a "common settings" or "presets" option, select "Once a day" from it. This should pre-populate the fields.

If such an option isn't present, use the following settings instead:

- Minute: **0** (this runs at 0 minutes every hour triggered)
- Hour: **9** (this will trigger at 9am every morning)
- Day: * (run this every day)
- Month: * (run this every month)
- Weekday: **1-5** (Only run this Monday-Friday - you don't notifications at the weekend!)

In the `command` box, enter the following:

```
php /path/to/generate-day.php >/dev/null 2>&1
```

The `php` in this line of code tells the cronjob to run the script as PHP, and the path is the one to your PHP script. Lastly, the `>/dev/null 2>&1` tells the cronjob not to output or email any reports to the user.

Command Line

Cronjobs can also be set up on the server using the command line. If you don't have access to the command line for your sever or you are not comfortable doing this, ask someone who is familiar with setting up recurring tasks to help you, or use the control panel option instead.

The next step is to create the cronjob. Remember that the time this triggers will be based on the local server time. Check your server time by typing `date` on the command line.

To edit the cronjob file, type:

```
sudo crontab -e
```

This will open a file. Add the following line of code to the bottom of that file:

```
0 9 * * 1-5 php /path/to/generate-day.php >/dev/null 2>&1
```

These options and commands are very similar to that of the control panel version, and so should look familiar.

You'll know receive a little Slack message every day (when appropriate) that lets you know what you should be celebrating.

HACK 03 Add Hacker News Updates

> In this hack, you'll use an API to post news items to Slack. We'll use an API because, although Slack can handle RSS feeds, it doesn't provide control over the output, and some sites don't offer an RSS feed.

We're going to retrieve articles from Hacker News. The official Hacker News API requires a lot of requests to piece together the required information, so we will use the API provided on algolia.com.

The script we're going to create is written in PHP and runs a cron job every minute. It uses the provided API to get the latest story. If the story is dated after the last stored date, it will post to Slack and update the stored date to that of the most recent post.

This isn't a bulletproof method, because if two (or more) stories are posted within a minute of each other, this script will post only one story. But it should work well enough.

The API endpoint we are going to use is:

```
http://hn.algolia.com/api/v1/search_by_date?tags=story
```

Create an Incoming Webhook

This script uses Incoming Webhooks to post messages to Slack.

Add an Incoming Webhook by visiting the Apps and Integrations app directory, searching for Incoming Webhooks and then clicking install. Select a channel that you want the messages to be posted to and then click **Add Incoming Webhook**.

Once installed, you will be shown a Webhook URL. Make a note of the URL, since you'll use it to post messages.

Gather the data

The next step is to gather the data from the feed. The Hacker News API allows you to specify several parameters to limit the number of items you get, which in turn speeds up the request.

Create the PHP script on a server web-accessible called `feed.php`. This will host all of the code. At the beginning of the file, specify the URL of the feed.

```
<?php

    // Feed URL
    $feed = 'http://hn.algolia.com/api/v1/search_by_date?';
```

Next, create a `$lastDate` variable. This will start at `0`, but will be updated every time the script runs to store the date of the last item retrieved. You can then build a new request and get any new articles.

```
// Set a last collected date - this will be dynamically updated later
$lastDate = 0;
```

Now, build up the parameters for the API. PHP includes a function that can take an array and transform it into a URL encoded string: `http_build_query()`. (More information about this function can be found on the PHP Docs page). Add the following code to `feed.php`:

```
// Parameters
$params = array(
    // Only retrieve stories (not polls or comments)
    'tags' =>  'story',
    // Get any stories created after the lastDate integer
    'numericFilters' => 'created_at_i>' . $lastDate,
    // Only return 1 item
    'hitsPerPage' => 1
);

// Build the full path using http_build_query
$url = $feed . http_build_query($params);
```

Once you build the `$url` you can retrieve it using PHP's `file_get_contents()`. The feed will be in JSON format, which can easily be converted to a PHP object by wrapping `json_decode()` around the file request. To retrieve the feed and decipher it, add the following code to feed.php:

```
// Get the API data
$data = json_decode(file_get_contents($url));
```

Next, add the following code to check to see whether there are any new stories. If not, you will exit the script (there is no point in continuing).

```
// Check to see if there are any new stories
if(!count($data->hits))
    exit('No stories');
```

You want to post the latest story to Slack, so to make accessing all of the information easier, assign a new variable (`$story`) to the first item in the `hits` array in your PHP file:

```
// Isolate the story
$story = $data->hits[0];
```

Format the story's date

Before posting the data to Slack, you should format the date of the story, making it more human readable.

Currently, the date exists in a Unix timestamp format also known as ISO_8601. Using PHP, you can take the timestamp and convert it to something more easily understood by the the DateTime PHP class. Here is an example of how to do that:

```
// Format the date - pass an @ if using timestamp
$date = new DateTime('@' . $story->created_at_i);

echo $date->format('jS F Y g:ia');
// returns date in format of:
// 1st January 1970 12:01am
```

When using the `DateTime` class with a Unix timestamp, an `@` must be passed first, to allow the code to understand the input.

Build the payload for Slack

Now that we have a formatted date, we can proceed with formatting the payload data for Slack. In this block of code, we parse the Date using `DateTime()` and then build up an array of information ready to be posted to Slack. Add this code to your PHP file:

```
// Format the date - pass an @ if using timestamp
$date = new DateTime('@' . $story->created_at_i);

// Create fields array
$fields = array(
    array('title' => 'Title', 'value' => '<' . $story->url .
      '|' . $story->title . '>'),
```

```
        array('title' => 'Date', 'value' => $date->format('jS F Y g:ia'),
          'short' => true),
        array('title' => 'Author', 'value' => $story->author, 'short' => true)
    );

    // Conditional field if story_text is present (strip any HTML tags)
    if($story->story_text != null) {
        $fields[] = array('title' => 'Story text', 'value' => strip_tags($story->stor
    }
    // Conditional field if comment_text is present (strip any HTML tags)
    if($story->comment_text != null) {
        $fields[] = array('title' => 'Comment text', 'value' => strip_tags($story->co
    }

    // Encode the data
    $payload = 'payload=' . json_encode(array(
        // Username and nice icon
        'username' => 'Hacker News',
        'icon_emoji' => ':fax:',

        // Required fallback and some pretext
        'pretext' => 'A new story from Hacker News',
        'fallback' => 'New hack news story - ' . $story->url,

        // Hacker news orange
        'color' => '#ff6600',

        // Title as a link, date and author of the news story
        'fields' => $fields
    ));
```

Before building the payload, the `$fields` array is predefined and conditionally appends the `story_text` and `comment_text` if it is present. We've decide to attach the news as a field attachment because doing so gives you more control over the formatting and the color (in this example, it has been set to Hacker News orange).

The title of the story has been passed in as a link. This uses the Slack link format of `<URL|Link text>`. Note the vertical pipe `|` between the URL and text.

The username and icon is also set in the code; we went for the fax emoji to signify a new story.

Lastly, we need the PHP cURL request, so add this to the end of your file:

```
    // PHP cURL POST request
    $ch = curl_init($webhook);
    curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
```

```
curl_setopt($ch, CURLOPT_POSTFIELDS, $payload);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$result = curl_exec($ch);
curl_close($ch);
```

This produces the output shown in Figure 5-2.

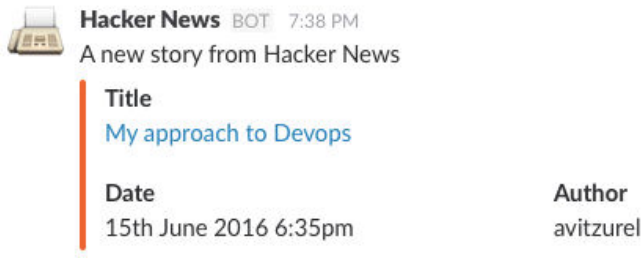


Figure 5-2. Hacker News Bot

Retrieving new items

As mentioned before, the feed URL can be passed an additional parameter in the form of a UNIX timestamp so that it retrieves stories from a specific date and time. If you use this parameter, it will improve the speed of the HTTP request, because it restricts the number of items the feed needs to return.

To achieve this, the script needs to create and use a temporary file to store the timestamp of the last story it processed. PHP includes the ability to read, write and create other files in the filesystem. To do this, it uses the following functions:

- `file_get_contents()` to retrieve the data from a file
- `file_put_contents()` to write data to a file.

Add this code to the very *top* of your feed.php file:

```
// Specify path to  Temp data file
$file = 'date.txt';

// Set a last collected date
$lastDate = file_get_contents($file);
if(!$lastDate) {
    $lastDate = 0;
}
```

Here, we specify the path to the temporary file. If this file doesn't exist, PHP returns `false`, in which case the `$lastDate` variable gets set to `0`. If the file is present with contents, the `$lastDate` gets assigned to the temporary file, which should be a UNIX timestamp.

At the very end of the PHP file, add the code below. It stores the timestamp of the processed story in the temporary file for the next time the script runs.

```
file_put_contents($file, $story->created_at_i);
```

The final feed.php script

Now that we've saved the last processed article and are constantly updating the temporary file, the webhook file is complete:

```
<?php

    // Your webhook URL
    $webhook = '[WEBHOOK URL]';

    // Feed URL
    $feed = 'http://hn.algolia.com/api/v1/search_by_date?';

    // Specify path to  Temp data file
    $file = 'date.txt';

    // Set a last collected date
    $lastDate = file_get_contents($file);
    if(!$lastDate) {
        $lastDate = 0;
    }

    // Parameters
    $params = array(
        // Only retrieve stories (not polls or comments)
        'tags' =>  'story',
        // Get any stories created after the lastDate integer
        'numericFilters' => 'created_at_i>' . $lastDate,
        // Only return 1 item
        'hitsPerPage' => 1
    );

    // Build the full path using http_build_query
    $url = $feed . http_build_query($params);

    // Get the API data
    $data = json_decode(file_get_contents($url));

    // Check to see if there are any new stories
```

```
if(!count($data->hits))
    exit('No stories');

// Isolate the story
$story = $data->hits[0];

// Format the date - pass an @ if using timestamp
$date = new DateTime('@' . $story->created_at_i);

// Create fields array
$fields = array(
    array('title' => 'Title', 'value' =>
      '<' . $story->url . '|' . $story->title . '>'),
    array('title' => 'Date', 'value' =>
      $date->format('jS F Y g:ia'), 'short' => true),
    array('title' => 'Author', 'value' =>
      $story->author, 'short' => true)
);

// Conditional field if story_text is present (strip any HTML tags)
if($story->story_text != null) {
    $fields[] = array('title' => 'Story text', 'value' => strip_tags($story->
}

// Conditional field if comment_text is present
if($story->comment_text != null) {
    $fields[] = array('title' => 'Comment text', 'value' => strip_tags($story
}

// Encode the data
$payload = 'payload=' . json_encode(array(
    // Username and nice icon
    'username' => 'Hacker News',
    'icon_emoji' => ':fax:',

    // Required fallback and some pretext
    'pretext' => 'A new story from Hacker News',
    'fallback' => 'New hack news story - ' . $story->url,

    // Hacker news orange
    'color' => '#ff6600',

    // Title as a link, date and author of the news story
    'fields' => $fields
));

// PHP cURL POST request
$ch = curl_init($webhook);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'POST');
curl_setopt($ch, CURLOPT_POSTFIELDS, $payload);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
```

```
$result = curl_exec($ch);
curl_close($ch);

file_put_contents($file, $story->created_at_i);

echo $result;
```

Trigger the Script every Minute

The final step is to trigger the script on a every minute using a cronjob. This can be done either through a control panel (for example cPanel) if you have one, or through the command line. We explain both methods below.

Control Panel

Log into your cPanel and navigate to "Cron Jobs". You should be presented with a series of drop down options. If there is a "common settings" or "presets" option, select "Once a minute" from it. This should pre-populate the fields.

If such an option isn't present, use the following settings instead:

- Minute: * (Every minute)
- Hour: * (Every Hour)
- Day: * (Every day)
- Month: * (Every month)
- Weekday: **1-5** (Every weekday)

In the `command` box, enter the following:

```
php /path/to/feed.php >/dev/null 2>&1
```

The `php` in this line of code informs the cronjob to run the script as PHP, and the path is the one to your PHP script. The `>/dev/null 2>&1` bit tells the cronjob not to output or email any reports to the user.

Command Line

Cronjobs can also be set up on the server using the command line. If you don't have access to the command line for your server or aren't comfortable doing this, find someone who is familiar with setting

up recurring tasks and ask for their help, or use the control panel option instead.

The first step is to create the cronjob. Remember that the time this triggers will be based on the local server time. (You can check your server time by tying `date` on the command line.)

To edit the cronjob file, type:

```
sudo crontab -e
```

This will open a file. Add the following line of code to the bottom of that file:

```
0 9 * * 1-5 php /path/to/generate-day.php >/dev/null 2>&1
```

These options and commands are very similar to that of the control panel version (described above), and so should look familiar.

You should now get the latest hacker news story every minute, every weekday!

HACK 04 Output your team's timezones

If you work with a team that's spread around the globe, you want to be sure that you don't try to contact someone in the middle of the night their time. This hack will show you how to create a handy slash command that outputs the current times and timezones of all the users in a given channel (see [Figure 05-03]).

Figure 5-3. Timezones Bot

To get started, create a PHP file called `timezones.php` on a web accessible URL. This script will be triggered by a slash command. In the same folder as this script, download the slack-api PHP wrapper.

This script is best for small teams or channels as it requires querying the Slack API several times.

Set up the slash command and auth token

Navigate to the Slack apps and integrations website and, using the search box at the top of the page, search for "Slash commands". Click the first result titled "Slash commands," and then click the "Add configuration" button on the left-hand side.

In the "Choose a Command" box, enter something appropriate, such as `/times`. On the following page, input the URL to your `timezones.php` script into the URL box. You can customise the look of your slash command using the "Customize Name" and "Customize Icon" fields using the settings below:

- Customize Name: Timezones
- Customise Icon: Click "Choose an Emoji" and search for the clock emoji.

Next, create an oAuth token, since you'll have to query users and channels to gather the information needed for this script.

Obtain an API Token

This script uses the Slack API to gather the info it needs. In order to interact with the API, you need to generate an oAuth token:

1. Go to *https://api.slack.com/web*.
2. Scroll down to *"Generate test tokens"*.
3. If you have previously generated a token, there will be one near the top, with the team name on the left. If not, click the *"Create token"* button.
4. Make a note of the token.

Script Set Up

There is a small amount of set-up we need to do for this hack. Since we are dealing with timezones, you need to make sure that the script

uses UTC as the base time, regardless of what the server is set to. To do that, add the following code to your `timezones` PHP script:

```
<?php

    // Your oAuth Token
    $token = '[AUTH TOKEN]';

    // Set the timezone independent of the server
    date_default_timezone_set("UTC");

    // Get the channel ID the script was triggered from
    $channelID = $_POST['channel_id'];

    // Include slack library from https://github.com/10w042/slack-api
    include 'Slack.php';

    // Create new Slack instance
    $Slack = new Slack($token);
```

Now that the Slack instance is initialized, we can query the Slack API to get a list of all of the members of the channel the script was triggered from using the payload Slack sends when triggering Webhooks:

```
// Get the cuurent channel and user
$channel = $Slack->call('channels.info', array('channel' => $channelID));
```

After you have the channel information, you need to get the current time as a Unix timestamp. You also need to create an empty array (you'll use it later). To accomplish both these things, add the following to your `timezones` file:

```
// Get the time right now
$now = time();

// Create an empty array
$times = array();
```

We used the `time()` function here instead of the `date()` function because, unlike the `date()` function, the `time()` function doesn't require any further parameters to get the timestamp. If you used the `date()` function instead, your preferred format would need to be passed as additional parameters.

Loop through the channel's users

Now that you have information about the channel, you can loop through each user on that channel and create a user instance based

on the user's ID. Once you have the user instance, you can then get the timezone and name of the user. To do that, append the following code to your PHP file:

```
// Loop through channel members
foreach($channel['channel']['members'] as $mid) {

    // Create member instance with ID
    $user = $Slack->call('users.info', array('user' => $mid));
}
```

Chances are (because you're reading this book) that you have at least one bot in your Slack team. We only really want to see timezones of *real* team members, so the code below excludes bots from the output of your script. Add this into the `foreach` loop above, after the `users.info` Slack API call.

```
// if the user is a bot, or doesn't have a timezone offset, skip them
if(($user['user']['is_bot'] == true) || \
  (!isset($user['user']['tz_offset']))) {
    continue;
}
```

This code checks whether the `is_bot` flag is set to true *or* the timezone offset is missing from the profile. If either (or both) of those conditions are true, then it fires `continue`. In the instance of a loop, the code will skip the remainder of the current code and move onto the next user.

Once you are happy that you have an actual user, there are a few variables to calculate.

We need to work out each user's time difference (in hours) compared to UTC, so we can output it at the end of the message. In the Slack API, the `tz_offset` is set in seconds, so dividing it by 60 twice gives us the figure we need. Add the following code after the `if` statement that's inside the `foreach` loop:

```
// Work out offset in hours from seconds
$userOffset = ($user['user']['tz_offset'] / 60) / 60;
```

Next, we want to get the user's name. If someone has filled out their profile correctly, the following code will grab their full name; if not, it will go with their username, so at least we have an identifying title. Add the following code after where the `$userOffset` variable is declared:

```
// Get the name of the user
$name = ($user['user']['real_name']) ? $user['user']['real_name'] : \
   $user['user']['name'];
```

In order to work out the current "local time" for the user, the following code adds the timezone offset seconds to the current timestamp. This provides a timestamp that you can format later. Add this code after where the `$name` variable is set:

```
// Add the timezeone offset to current time
$userTime = $now + $user['user']['tz_offset'];
```

The last variable we need to set is `$key`. We are appending all of the users to an array, and we want them listed in time order. By making the key of the array the same as the offset, we can ensure they will be in the correct order. Because a PHP can't have negative arrays, the following code adds 12 to every user. `-11` is the lowest the offset can be, so this ensures it will always be a positive number. Use the code below to set the `$key` variable—place it after the `$userTime` variable.

```
// Create an array key based on offset (so we can sort) and add 12
// (as it could be -11 at worst)
$key = $userOffset + 12;
```

Now that we have all of the variables we need, we can build the string for the user. Add this code inside the `foreach` loop:

```
// Append the details to the array as the key
$times[$key][] = '*' . $name . '*: ' .
date('h:i A', $userTime) . ' local time on ' . date('jS M', $userTime) .
' (UTC' . sprintf("%+d", $userOffset) . ' hours _' . \
  $user['user']['tz_label'] . '_)';
```

The string is being appended to the `$times` array, with the correct key. To allow for cases where two users are in the same timezone, we create a child array to hold them using the empty square brackets `[]` after the `[$key]`.

The string outputs the following text (with various formatting):

- `$name` - the name of the user.
- `date('h:i A', $userTime)` - the hour & minutes in a 12 hour clock (e.g. 12:01 AM).
- `date('jS M', $userTime)` - the day and month of the user's local time in the instance that any of the users are on a different day to the rest of the team (e.g. 1st Jan).

- `sprintf("%+d", $userOffset)` - This outputs the timezone offset with a + prepended if it is positive (e.g. +2 hours).
- `$user['user']['tz_label']` - This outputs the label for the user's timezone (e.g. Pacific Daylight Time).

A populated user string looks like this:

> *Joe Blogs:* 12:01 PM local time on 1st Jan (UTC-7 hours *Pacific Daylight Time*)

Pieced together, the entire user `foreach` loop now looks like this:

```
// Loop through chanel members
foreach($channel['channel']['members'] as $mid) {

    // Create member instance with ID
    $user = $Slack->call('users.info', array('user' => $mid));

    // If the user is a bot, or doesn't have a timezone offset, skip them
    if(($user['user']['is_bot'] == true) ||
       (!isset($user['user']['tz_offset']))) {
        continue;
    }

    // Work out offset in hours from seconds
    $userOffset = ($user['user']['tz_offset'] / 60) / 60;

    // Get the name of the user
    $name = ($user['user']['real_name']) ? $user['user']['real_name'] : \
        $user['user']['name'];

    // Add the timezeone offset to current time
    $userTime = $now + $user['user']['tz_offset'];

    // Create an array key based on offset (so we can sort) and add 12
    // (as it could be -11 at worst)
    $key = $userOffset + 12;

    // Append the details to the array as the key
    $times[$key][] = '*' . $name . '*: ' .
    date('h:i A', $userTime) . ' local time on ' . date('jS M', $userTime) .
    ' (UTC' . sprintf("%+d", $userOffset) . ' hours _' . \
      $user['user']['tz_label'] . '_)';

}
```

Processing the data

Now that we have a multidimensional array of `$times` with all of the users' timezones stored inside, we need to process the data to make it ready for Slack. To do that, add this code after the `foreach` loop:

```
// Sort array items by key
ksort($times);

// Flatten the array and implode it - separated by new lines
$text = implode("\n", call_user_func_array(
    'array_merge', $times
));
```

The first function sorts the array by keys (such as smallest to biggest). This means that people with a timezone of -10 (and a key of 2, since we added 12) come before people in timezone +3 (key of 15).

The next line helps flatten the array and convert it to a string. Currently, the `$times` array looks like this:

```
$times = array(
    2 => array(
        'Joe Blogs:...',
        'John Doe...'
    ),
    15 => array(
        'Jane Doe...'
    )
);
```

The keys were there purely for ordering purposes, so the array could be flattened. Calling the `call_user_func_array` function with `array_merge` does exactly that. After passing it through this series of functions, the array would become:

```
$times = array(
    'Joe Blogs:...',
    'John Doe…',
    'Jane Doe...'
);
```

As Slack expects a string, we can `impode()` that array, using the newline character `\n` as the "glue".

You can now trigger the script with the slash command you created (`/times` in this example), and get a list of all the local times for users in that channel.

Slack automatically notifies a user if their name is mentioned in a channel. This script mentions everyone's names so it is likely that they will get a notification when the script is called. Also, because the script is doing several API calls, the more people in the channel, the longer it will take to return results.

HACK 05 Connect Slack to Trello

Trello is a collaborative project-management tool. It helps teams and groups sort bugs and todos, based on the kanban board method. It incorporates the ideas of lists and cards. A card is often a task and is contained in a list (such as To-do or Done).

This hack will whos you how to create a new card on a specified Trello board using Outgoing Webhooks and the last message sent in a Slack channel. This same thing can be achieved using Zapier, but by using this custom hack instead of Zapier, you have control over the card's formatting, where it goes, and which board in Trello it goes on.

The process for this script will be:

1. User initializes the script by typing `add to trello <board name>`.
2. The script gathers the last message sent in that Slack channel.
3. The script works out what board the user mentioned. If the board doesn't exist or the script can't find it, it will use the default board.
4. A card is created in the first list of that board.
5. A message is returned with a link to the board.

For this hack you will need a Slack API key as well as a Trello key and token.

For the Trello credentials, we advise you to make a new autonomous user account and add it to all the boards. For example, create an account called Jeeves or Jarvis who can post to all the boards required.

In Slack, the bot results in the output shown in Figure 5-4.

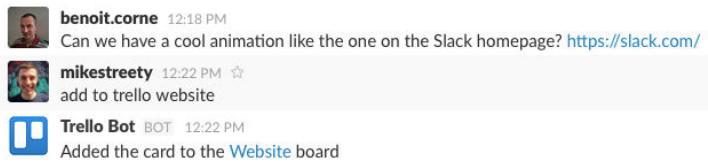


Figure 5-4. Trello Bot in Slack

And the resulting card in Trello looks like the one shown in Figure 5-5.

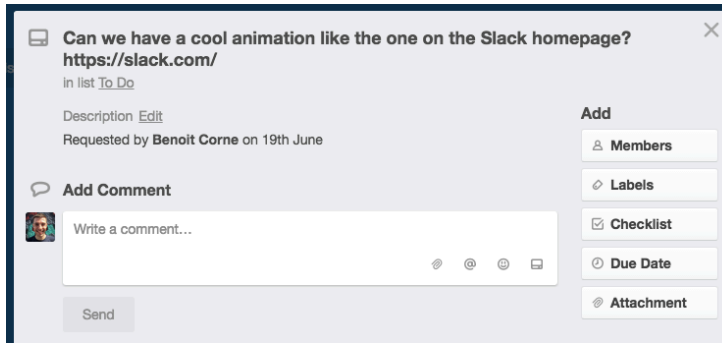


Figure 5-5. Trello Bot Card

Create the Outgoing Webhook

First you need to set up an Outgoing Webhook. The settings below are just a guide. Most of these settings can be overridden in the script itself.

- Channel: **Any**
- Trigger Word(s): **add to trello**
- URL(s): The URL where your script is hosted
- Descriptive Label: Describe what the bot does
- Customize Name: **Trello Bot**
- Customize Icon: Whatever you wish

Obtain a Slack API Token

This script use the Slack API to gather the information it needs. In order to interact with the API, you need to generate an oAuth token:

1. Go to *https://api.slack.com/web*.
2. Scroll down to *"Generate test tokens"*.
3. If you have previously generated a token, there will be one near the top, with the team name on the left. If not, click the *"Create token"* button.
4. Make a note of the token.

Obtain a Trello Key and Token

The Trello API allows a *logged in user* to get a key and token, which allows access to all of that user's boards and cards. To get a key and token:

1. Go to *https://trello.com/app-key*
2. Copy the developer key on that page
3. Click here to generate a token
4. When you're asked to allow the app use your account, click Allow
5. Copy the token that's displayed

Default Board ID

The last thing you need before you write your script is the ID of the default Trello board. This is the board that gets posted to if the script can't find the requested board or if not board is specified. You can get this info by navigating to the board and copying its ID from the URL. For example, you'd copy whatever appears in place of [ID] in the URL below - it will be some numbers and letters:

```
https://trello.com/b/[ID]/testing
```

Now that you have the required keys and tokens, let's start the script. On a server, create a PHP file called `trello.php`, and then enter the following code into it. This code specifies the Slack and Trello tokens, Trello key, and the default board ID:

```
<?php

    // Your Slack oAuth Token
    $slack_token = '[Slack auth token]';

    // Trello
    // https://trello.com/app-key
    $trello_key = '[Trello key]';
    $trello_token = '[Trello token]';
    $trello_default_board = '[Default board id]';

    // Trigger Word
    $trigger_phrase = 'add to trello';
```

We have specified the trigger word for the file. Make sure this matches what you entered when you created your Webhook so that you can remove the trigger word when you add the card to Trello.

Libraries

There are some PHP wrappers available that make using the Slack and Trello APIs easier.

The first one is the Slack PHP API wrapper. Download this script and place it next to your PHP script on the server.

The second is php-trello. Download this zip and place in a folder called `trello` in the same directory as your PHP script.

Include the libraries in your PHP file by adding the code below. The Trello wrapper requires the you to specify `namespace`.

```
// Include slack library
include 'Slack.php';

// Include Trello API helpers
include 'trello/src/Trello/OAuthSimple.php';
include 'trello/src/Trello/Trello.php';

// Set Trello namespace
use \Trello\Trello;
```

With these libraries loaded, you can initialize the `Trello` and `Slack` classes with the tokens you obtained earlier. To do so, insert the following code into `trello.php`:

```
// Create new Slack & Trello instance
$Slack = new Slack($slack_token);
$Trello = new Trello($trello_key, null, $trello_token);
```

Process the Slack channel data

You need to get the current Slack channel's history, to get both the message that triggered the script (as it contains the relevant board's name) and the message you wish to add to Trello as a card.

Because some of the messages posted in Slack aren't user messages, retrieve the last 5 message in a channel to ensure the board that gets added is the one requested. To do that, add this to `trello.php`:

```
// Get the last 5 messages of the channel conversation
$slack_data = $Slack->call('channels.history', array(
    'channel' => $_POST['channel_id'],
    'count' => 5
));
```

With the channel messages now available as a PHP variable, you need to remove the most recent one (this is the message that triggered the script). From there, we are going to remove the trigger phrase so we are left with the name of the board the user wishes to post to. Add the following to your PHP script after the Slack API call:

```
// Remove the last message - it will contain the trigger phrase
$slack_trigger_message = array_shift($slack_data['messages']);

// Remove trigger_phrase from message to get board name
$slack_board_name = trim(
    str_replace(
        strtolower($trigger_phrase),
        '',
        strtolower($slack_trigger_message['text'])
    )
);
```

For the final Slack message processing stage, we are going to loop through the remaining messages until we find one with a `type` equal to `message`. Once we do (it will generally be the first one), we are going to assign it to a variable and `break` out of the loop. The message also gets processed to remove `<` and `>` from any URL. Add the following code after where the `$slack_board_name` variable is defined:

```
// Find the next message in the channel - strip URL formatting
foreach($slack_data['messages'] as $message) {
    $message['text'] = preg_replace("/\<(http[^ ]+)>/", "\\1", $message['text']);
    if($message['type'] == 'message') {
        $trello_card = $message;
        break;
```

```
    }
}
```

Load the Trello board

In the next stage, you'll load the Trello board to post to it. To do this, you must use the API to load a message to an authenticated Trello user, get their open boards, and loop through them until you find one that matches. If there is no board specified (or matching), the default board will be loaded.

> The code that decides whether the board matches simply checks whether a string is contained within another string. For example, if you type `add to trello m`, the script will add the card to the first board it finds with an M in its name.

Add this code to your `trello` PHP script:

```
// Get the trello user
$trello_user = $Trello->members->get('me');

// Get all the boards the user has access to that are open
$trello_boards = $Trello->members->get(
    $trello_user->id . '/boards',
    array(
        'filter' => 'open'
    )
);

// Find the board that matches the message text
foreach ($trello_boards as $board) {
    if(strpos(
        strtolower($board->name),
        $slack_board_name
    ) !== false) {
        $trello_board = $board;
        break;
    }
}

// if there is no board - fall back to the default one
if(!$trello_board) {
    $trello_board = $Trello->boards->get($trello_default_board);
}
```

Pick the Trello list

Now that you have the Trello board selected, you want to load the lists contained within that board, and then select the first list on that board. (This will be the list you add the card to.) To accomplish these two things, add the following to the end of your script.

```
// Get all the lists in the board
$trello_lists = $Trello->boards->get(
    $trello_board->id . '/lists'
);

// Get the first list of the board
$trello_list = $trello_lists[0];
```

Create the Trello card

With the board and list selected, the last thing you need to do is to build the card and post it.

We want to add a bit of context to the card, including the name of the person who posted the note and the time they did so. Since we have the card title and details loaded in the `$trello_card` variable, we can load the user based on who posted it.

Because people aren't perfect, not everyone will have filled in their profile and has the `real_name` field populated. In the code below, we set the name to the `name` key, and override this if `real_name` exists. Insert this code at the bottom of `trello.php`:

```
// Get the Slack user who noted the bug
$slack_user = $Slack->call('users.info', array(
    'user' => $trello_card['user']
));

// Get the name of the user - if there is a real_name, use that
$slack_name = $slack_user['user']['name'];
if(count($slack_user['user']['real_name'])) {
    $slack_name = $slack_user['user']['real_name'];
}
```

Now you have all of the info you need to add the card to Trello. The list is loaded (with the ID we got before) and the card added. We set the title of the card (or `name`) to the same as the Slack message, and populated the description (`desc`) with the name and date of the request. To post the card to Trello, simply add this bit of code to trello.php:

```
// Post the card to trello
$Trello->lists->post($trello_list->id . '/cards', array(
    'name' => $trello_card['text'],
    'desc' => 'Requested by **' . $slack_name .
        '** on ' . date('jS F', $trello_card['ts'])
));
```

Report back to the user

The last thing you need to do is let the user know that the card was posted successfully. As a nice addition, the board the card was added to is linked in this return message, so the user(s) can jump straight to it. The link is created using Slack's `<link url|text>` syntax:

```
// Post back to board it was posted to
header('Content-Type: application/json');
echo json_encode(array(
    'text' => 'Added the card to the <' .
        $trello_board->shortUrl .
        '|' . $trello_board->name .
        '> board'
));
```

You're all set. You can now add a card to a Trello board by typing `add to trello` followed by the board name.

Although this script replicates some aspects of the Zapier app, it opens the gateway for further expansion. For example you can:

- Add custom labels to the card (to indicate that it's from Slack).
- Include a link to the message (to get chat context) by removing the `.` from the message timestamp and using the message name.
- Keep a key/value list of channels to boards so that, when the script is run in a certain channel, the resulting card always goes to a particular board.
- Set a board and use the text after the command to set the message title.
- Include the last 5 or so messages of the conversation in the Trello card.

6

Creating Chatbots

In this chapter you will learn to install a "full" chatbot. Unlike the chatbots covered earlier in this book, you can extend the functionality of full chatbots using an API that is native to the language you extend the bot in. (For instance, the Errbot chatbot that we use in this chapter is written in Python, so you can extend its functionality using Python.) Full chatbots are also software frameworks with many features allowing you to react to Slack events, mentions, custom presentations, and more.

Installing a full bot in Slack gives you access to a complete plugin ecosystem that has been built over time. Another advantage of using a full chatbot is that once you have set it up, you just need code to extend it; you don't have to deal with repeated deployment procedures like you do with apps and slash commands.

Chatbots also abstract you from the chat system that you are using. So for example, Errbot (one of the chatbot we will explore) has a plugin method called `send_card` that sends a native message attachment on Slack. If you connect your instance to Hipchat it will use Hipchat's native card rendering! This also applies for identity management, callbacks, a rendering language (markdown), etc. This means that you can publish and share your plugins and they should work out of the box for other users on other platforms.

Source code

The source code for all the hacks in this chapter can be found in this book's Github repo.

We will start with Errbot, an easy chatbot to extend in Python, and tour the capabilities of the bot with a series of examples you can use for your team. We will then discuss some other popular chatbots: Hubot and Lita, which are extensible in Coffeescript and Ruby, respectively. And in the last portion of this chapter, we cover another specialized Slack chatbot called Simple Slack API, which is open source software written in Java and freely available on Github. It allows Java (or other Java VM language) developers to quickly build a fully functional Slack bot running as a standalone application.

Let's get started.

HACK 01 Connect Errbot to Slack

Let's set up Errbot so you can take advantage of the hacks in this chapter that use it. As mentioned earlier, Errbot is a chatbot that is written and is extensible in Python. Its homepage and documentation is located at *http://errbot.io*.

Errbot is a daemon process that can connect to Slack and converse with you and your team. Since Errbot can install dependencies automatically, we recommend you install it into a Python *virtualenv* (we'll show you how). Once your local environment is ready, we will explore some cool features/hacks using the bot.

Some benefits of Errbot

Here are some of the benefits of using Errbot:

- It's managed by "chatops": for example you can install and configure plugins just by chatting with the bot.
- It's chat system agnostic: write your plugin once, and run it on every supported chat system.

- Its plugin repos are just Git repos.
- It has a very gentle learning curve.
- Its features include presence, mentions, cards, conversation flows, automation, and more.

Prerequisites

We recommend that you use the following to install Errbot:

- Linux, Mac OS, or Windows
- Python 3.4+
- virtualenv

HACK 02 **Windows and Python 2 compatibility**

> Errbot is compatible with Windows, but its support is limited (there is no daemon mode, so no background processing on Windows, and many public Errbot plugins are built to run on Linux).

Errbot is compatible with Python 2.7.x, but this support is going away, and Python 3 will give you a better experience anyway with Python 3 specific features like type hints.

Installing Errbot

First you need to create a virtualenv. We recommend that you use virtualenvwrapper (see the note below for installation details) or the standard virtualenv tool explained as an alternative below.

Once you have virtualenvwrapper installed, it is quite easy to make a new virtualenv:

```
$ mkvirtualenv errbotve
(errbotve) $
```

HACK 03 Installing virtualenvwrapper

> virtualenvwrapper makes it really easy to create virtualenvs. See *https://virtualenvwrapper.readthedocs.org/* for details.

As an alternative to virtualenvwrapper, you can use the standard virtualenv. With the commands below, we create a virtualenv in a hidden directory in your home directory called `.errbotve`:

```
$ sudo pip install virtualenv
$ virtualenv --python `which python3` $HOME/.errbotve

Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/gbin/.errbot-ve/bin/python3
Also creating executable in /home/gbin/.errbot-ve/bin/python
Installing setuptools, pip, wheel...done.
$ source $HOME/.errbotve/bin/activate
(errbotve) $
```

Once you have created a virtual environment, you can install Errbot on it. Pip, a tool used to install and manage Python software packages, will automatically fetch all of the basic dependencies for the bot:

```
(errbotve) $ pip install errbot[slack]
Collecting errbot
  Downloading errbot-4.1.3.tar.gz (191kB)
    100% |████████████████████████████████| 194kB 2.2MB/s
Collecting webtest (from errbot)
  Downloading WebTest-2.0.21.tar.gz (66kB)
    100% |████████████████████████████████| 71kB 6.5MB/s
Requirement already satisfied (use --upgrade to upgrade): \
    setuptools in ./.errbot-ve/lib/python3.5/site-packages (from errbot)
Collecting bottle (from errbot)
  Downloading bottle-0.12.9.tar.gz (69kB)
    100% |████████████████████████████████| 71kB 6.3MB/s

[...]


Successfully built errbot webtest bottle rocket-errbot yapsy markdown
    ansi pygments-markdown-lexer dnspython3 MarkupSafe cryptography
    cffi pycparser
Installing collected packages: six, WebOb, waitress, beautifulsoup4,
    [...]
```

```
Successfully installed MarkupSafe-0.23 Pygments-2.1.3 WebOb-1.6.1
    [...]
$
```

HACK 04 **Installing Slack support directly with Errbot**

> `pip install errbot[slack]` is a special syntax to make pip install the dependency required for Errbot to connect to Slack.

Now, create a root directory for your Errbot instance; this is where it will store its data, configs, and logs. Errbot will also automatically create a subdirectory for your plugins in development:

```
(errbotve) $ mkdir ~/errbot
(errbotve) $ cd ~/errbot
(errbotve) $ errbot --init
```

Try Errbot locally in a console

Now that you've got Errbot installed, you're ready to try it locally. To do so, run the following commands:

```
(errbotve) $ cd ~/errbot
(errbotve) $ errbot

[...]
22:37:15 DEBUG errbot.errBot *** frm = gbin
22:37:15 DEBUG errbot.errBot *** username = gbin
22:37:15 DEBUG errbot.errBot *** text =
22:37:15 DEBUG errbot.errBot Triggering callback_message on VersionChecker
22:37:15 DEBUG errbot.errBot Triggering callback_message on Flows
22:37:15 DEBUG errbot.errBot Triggering callback_message on Help
22:37:15 DEBUG errbot.errBot Triggering callback_message on Plugins
22:37:15 DEBUG errbot.errBot Triggering callback_message on Backup
22:37:15 DEBUG errbot.errBot Triggering callback_message on Utils
22:37:15 DEBUG errbot.errBot Triggering callback_message on Health
22:37:15 DEBUG errbot.errBot Triggering callback_message on ACLS
22:37:15 DEBUG errbot.errBot Triggering callback_message on ChatRoom

 >>>
```

Errbot presents you with a prompt `>>>`, and you can talk to it. For example, try the command `!about`:

```
>>> !about
```

```
[...]

This is Errbot version 4.1.3
• Visit http://errbot.io/ for more information about errbot in general.
• Visit http://errbot.io/en/latest/#user-guide for help with configuration,
  administration and plugin development.
 Errbot is built through the hard work and dedication of everyone who
contributes code, documentation and bug reports at
[...]
```

You can quit Errbot by pressing CTRL-C.

Using Errbot locally in a graphic mode (optional)

In addition to text mode (specified by the -T flag on the command line), Errbot also has a graphic mode (-G) that can display images, autocomplete commands, etc. This mode, like Text mode, is a development mode (available to all users that have access to the channel) that's useful for iterating quickly while you're developing plugins, because Errbot will not connect to any external chat service. In order to use Errbot's graphic mode, you need to install a dependency called pyside with `pip install pyside` (see Figure 6-1):

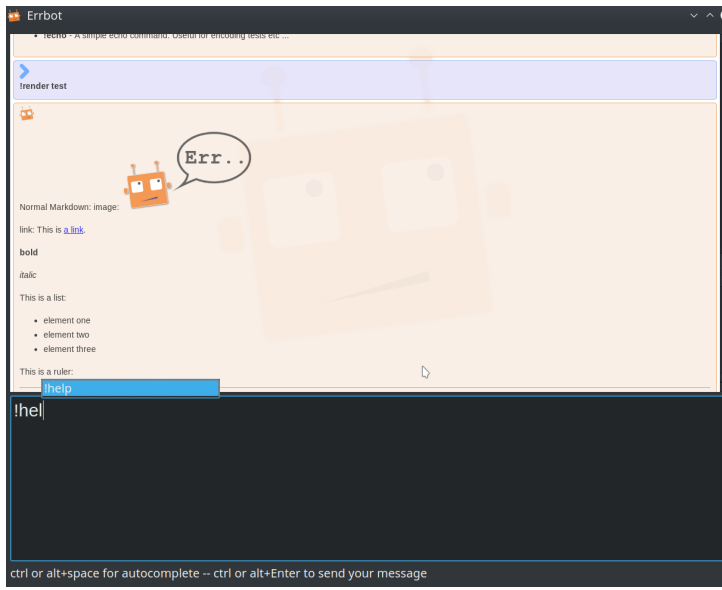


Figure 6-1. Errbot in graphic mode

Using Errbot in local text mode while keeping your settings in config.py You can always force back the Text or Graphic mode by starting Errbot with $ errbot -T or $ errbot -G to develop and test something quickly and locally without connecting to Slack.

Connect Errbot to Slack

In order to connect Errbot to Slack, first you'll need to create a bot user for your team using the form located at `https://TEAM.slack.com/services/new/bot` where TEAM is the name of your Slack team (see Figure 6-2).

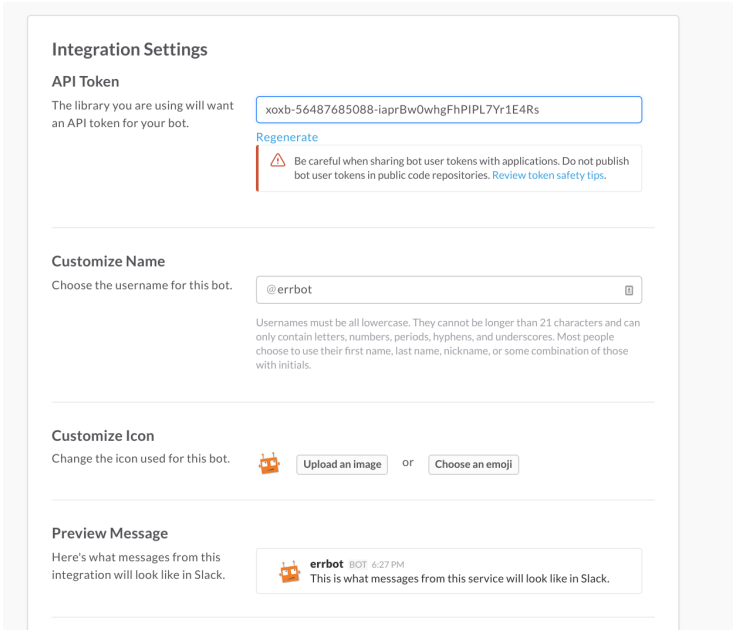


Figure 6-2. Slack bot configuration

Copy and paste the API Token entry from the Integration Settings (shown in Figure 6-2) into the `BOT_IDENTITY` section of your `config.py` file, and set the backend to `Slack`. Also, you'll need to set the `BOT_ADMIN` to a list of trusted users that can administer the bot.

Once you do all that, the content of `config.py` should look like something like this:

```
BACKEND = 'Slack'
BOT_IDENTITY = {
```

```
    'token': 'xoxb-56487685088-iaprBw0whgFhPIPL7Yr1E4Rs',
    }
BOT_ADMINS = ('gbin',)
```

HACK 05 **BACKEND value in config.py**

Setting BACKEND to `Slack` in config.py makes Errbot try to connect to Slack. Other possible values are the other supported chat systems like `hipchat`, `IRC`, etc.

Now invite the bot user to your chatroom using the channel's gear menu (see Figure 6-3).

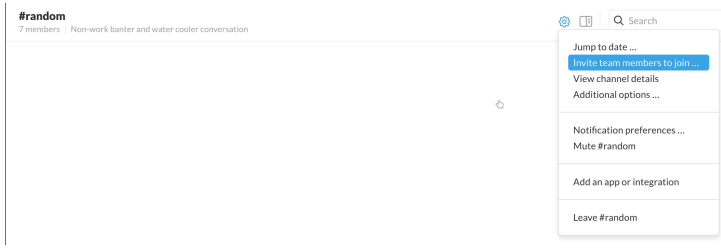


Figure 6-3. Slack bot invite 1

Then select the newly created bot user (see Figure 6-4):

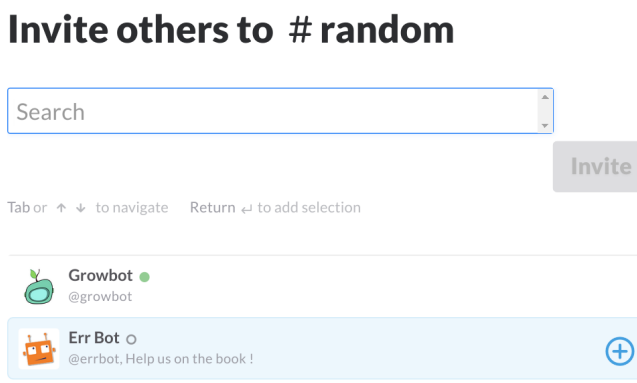
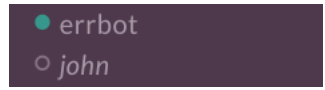


Figure 6-4. Slack bot invite 2

Now that Errbot is set up and invited to your Slack channel, you're ready to start Errbot with no parameter:

```
(errbotve) $ cd ~/errbot
(errbotve) $ errbot
[...]
```

The bot user's status-indicator cirlce should turn green in the chatroom you added Errbot to (see Figure), and you should now be able to "talk" to it. You can try commands like `!help`, `!whoami`, and `! echo cock-a-doodle-do !`.



Errbot's status indicator turns green (note the shaded circle).

Going beyond

Errbot has a lot of features for plugin designers. We will explore some of them in the following hacks, but here are some pointers from the documentation:

- Use *persistence* to store some data: *http://errbot.io/en/latest/user_guide/plugin_development/persistence.html*
- Make your plugin answer to events like *presence*: *http://errbot.io/en/latest/user_guide/plugin_development/presence.html#callbacks-for-presence-changes*
- Make your plugin answer to *webhooks*: *http://errbot.io/en/latest/user_guide/plugin_development/webhooks.html*
- Implement a conversational state or *flow*: *http://errbot.io/en/latest/user_guide/flow_development/index.html*
- Generate time-based events with *scheduling*: *http://errbot.io/en/latest/user_guide/plugin_development/scheduling.html*

Now that you have Errbot up and running, it's time to take advantage of what it can offer.

HACK 06 Organize tournaments with Errbot

In this simple hack, we'll show you how to use the err-tourney plugin to organize tournaments and maintain a ranking system. The ranking system is based on the *Elo system* (a method for ranking players), and is suitable for any one-on-one game like chess, go, table tennis, air hockey, pool, etc.

To get started, we will show you how to find and install a plugin on Errbot. As of this writing, Errbot has more than 300 public plugins available online, so let's examine how to search through the list.

Finding and installing err-tourney

If you setup Errbot with your Slack user as a `BOT_ADMIN` in the `config.py` file, you can administer the bot by sending it private messages.

For example, you can query the online plugin repository by sending the command `!repos search tourney` (see Figure 6-5).

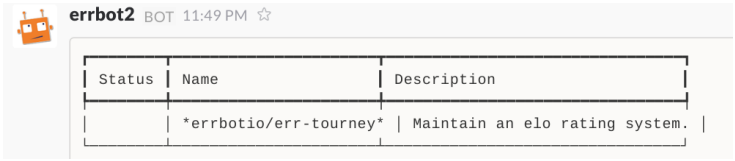


Figure 6-5. Searching the plugin repo.

Now that we have found the full name of the err-tourney plugin, we can install it simply by asking the bot to do so:

```
>>> !repos install errbotio/err-tourney
[...]
```

Checking out what commands the plugin exposed

You can use `!help`, or if you now the name of the plugin, `!help` plus the name of the plugin to see the commands available:

```
 >>> !help Tourney
Tourney
 Maintain an elo rating system.
• !elim cancel - Cancel the current direct elimination tournament
```

- `!elim start - Start a direct elimination tournament amongst the players`
- `!elo add - Add a player`
- `!elo match - record a match result`
- `!elo rankings - Printout the current elo rankings`
- `!elo remove - Remove a player`
- `!elo stats - Returns the current players statistics.`

Start using err-tourney

Now you can start using err-tourney to add players:

```
>>> !elo add tali
Player tali added

>>> !elo add stevo
Player stevo added

>>> !elo add davy
Player davy added

>>> !elo add pol
Player pol added

>>> !elo add gbin
Player gbin added
```

Once a result is in, you can start to record matches. Everyone starts with an elo ranking of 1500. After a match, the elo ranking of both involved players will swing depending on the results:

```
>>> !elo match pol davy pol
Game added pol won against davy.
       pol 1500 -> 1516
      davy 1500 -> 1484

>>> !elo match tali davy tali
Game added tali won against davy.
      tali 1500 -> 1515
      davy 1484 -> 1468
```

You can ask what the current statistics are:

```
>>> !elo stats
Player wins losses first/last
  tali    3     0  [2016-06-15 00:17:38/2016-06-15 00:19:00]
  gbin    2     1  [2016-06-15 00:18:12/2016-06-15 00:21:49]
   pol    1     1  [2016-06-15 00:15:06/2016-06-15 00:21:49]
 stevo    0     2  [2016-06-15 00:18:51/2016-06-15 00:19:00]
  davy    0     2  [2016-06-15 00:15:06/2016-06-15 00:17:38]
```

And who is the strongest:

```
>>> !elo rankings
01 -      tali [1544]
02 -      gbin [1518]
03 -       pol [1499]
04 -     stevo [1469]
05 -      davy [1468]
```

You have just learned seen how to install and use a plugin from a third party. Now let's see how to create your *own* plugin.

HACK 07 Generate ASCII art with Errbot

In this hack, we will create from scratch a very simple plugin for Errbot that can generate ASCII art. To do this, we will use Errbot's plugin-creation wizard.

First, go to your plugins directory (`~/errbot/plugins`), create a sub-folder for your plugin, and start the new-plugin wizard (the version prompts are left in this code snippet as default; you don't need to do anything with them):

```
$ cd ~/errbot/plugins
$ mkdir err-big
$ cd err-big
$ errbot --new-plugin
This wizard will create a new plugin for you in
            '/home/gbin/projects/err-plugins/err-big'.
What should the name of your new plugin be?
> Big
What may I use as a short (one-line) description of your plugin?
> This plugin will make Errbot display text as large ASCII art.
Which python version will your plugin work with? 2, 2+ or 3? I will
default to 3 if you leave this blank.
>
Which minimum version of errbot will your plugin work with? Leave
blank to support any version or input CURRENT to select the current
version (4.1.3)
>
Which maximum version of errbot will your plugin work with? Leave
blank to support any version or input CURRENT to select the current
version (4.1.3)
>
Success! You'll find your new plugin at
'/home/gbin/projects/err-plugins/err-big/big.plug'
(Don't forget to include a LICENSE file if you are going to publish
your plugin)
```

This will create big.plug, a plugin descriptor, and big.py, a plugin template containing a simple example to get you started. To show you a basic case (just a command), you can trim down the more comprehensive plugin the wizard created to the following from:

```
from errbot import BotPlugin, botcmd
from pyfiglet import Figlet

class Big(BotPlugin):
    """This plugin will make Errbot display text as large ASCII art."""
    @botcmd
    def big(self, msg, args):
        """ Generates a large ASCII art version of the text you enter."""
        return "```\n" + Figlet(font='slant').renderText(args) + "\n```"
```

Next, install the required dependency `pyfiglet`:

```
$ pip install pyfiglet
```

HACK 08 **Make Errbot automatically install plugin dependencies**

You can also add a `requirements.txt` file in the root of you plugin containing the list of you dependencies (one per line) and Errbot will install them when you execute the command `!repos instal`

Next, start Errbot in Text mode and it should respond to your new command:

```
$ errbot -T
[...]

>>> !big Slack Hacks !
[...]
   _____ __           __      __  __           __         __
  / ___// /___ ______/ /__   / / / /___ ______/ /______   / /
  \__ \/ / __ `/ ___/ //_/  / /_/ / __ `/ ___/ //_/ ___/  / /
 ___/ / / /_/ / /__/ ,<    / __  / /_/ / /__/ ,< (__  )  /_/
/____/_/\__,_/\___/_/|_|  /_/ /_/\__,_/\___/_/|_/____/  (_)
[...]
```

Now you can try it on Slack directly by restarting `errbot` without the `-T` parameter. Then, type `!big Some Text` in your Slack channel, and Errbot should respond with the ASCII version of the words you typed.

You have learned how to write a plugin and test it locally on Slack. You can now develop and iterate on a plugin for Errbot.

HACK 09 Send Facepalm memes with Errbot

In this hack we will create an Errbot plugin to display a random image (in our example, a meme) from the web to a Slack channel. The end result will look like the following (see Figure 6-6).

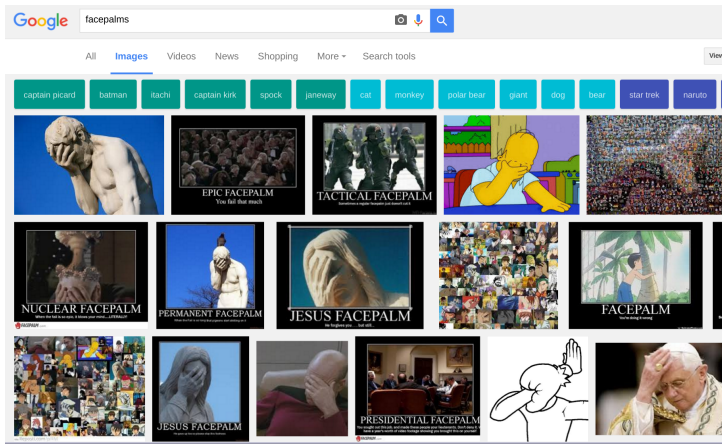


Figure 6-6. Displaying facepalms.

The base files to start with

This plugin will be structured like any other Errbot plugin.

In the `facepalms.plug` file, enter the following:

```
[Core]
Name = Facepalms
Module = facepalms.py

[Documentation]
Description = Display a random facepalm.
```

And in the base module `facepalms.py`, enter the following:

```
from errbot import BotPlugin, botcmd
```

```
class Facepalms(BotPlugin):
    pass
```

Gather some material for the plugin

You can find some funny facepalms on Google images, as shown in Figure 6-7.

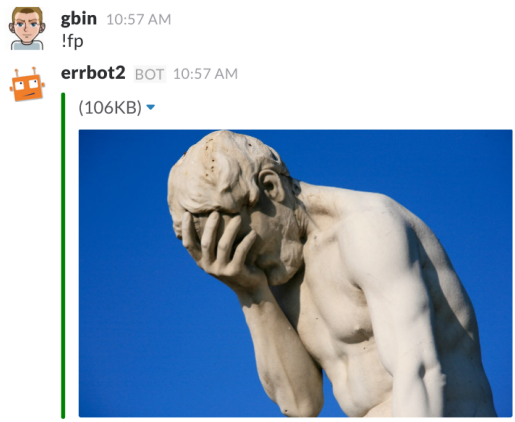


Figure 6-7. Finding facepalms.

Simply right-click/Ctrl-click on one and choose "View image" to copy its URL.

HACK 10 **Slack image formats**

> Be sure to only select images that are in formats supported by Slack: GIF, JPEG, PNG and BMP.

Add your selection as a constant at the top of the `facepalms.py` module:

```
from errbot import BotPlugin, botcmd

IMAGE_URLS = [
'https://upload.wikimedia.org/wikipedia/commons'
'/3/3b/Paris_Tuileries_Garden_Facepalm_statue.jpg',
'http://i.kinja-img.com/gawker-media/image/upload/'
't4pb8jdkc1c8bwcnck7i.jpg',
```

```
'https://qph.is.quoracdn.net/'
'main-thumb-t-475705-200-ygtxtfthoahjnxtxbndturgcxbnbyine.jpeg',
]

class Facepalms(BotPlugin):
    pass
```

Make Slack display an image

You can use the card feature (with `send_card`) to display the image as a Slack attachment:

```
import random

class Facepalms(BotPlugin):

    @botcmd
    def fp(self, msg, _):
        """Displays a random facepalm."""
        self.send_card(in_reply_to=msg, image=random.choice(IMAGE_URLS))
```

HACK 11 **A word on identities**

> `in_reply_to` is a shortcut that let's you avoid having to deal with public and private chatroom responses. You will have to use `self.build_identifier(str)` if you only have a textual representation of the room or the person you want to send the card to.

Now you can simply type `!fp` to trigger the plugin, which will display an image in response.

This hack has shown you how to use Slack attachments to display images. But Slack attachments can do much much more, so please refer to the send_card documentation to explore the possibilities.

HACK 12 Make Errbot self aware

This is a small trick you can play with Errbot. We've included it here mainly to show you some simple command parsing and how to build identifiers.

The goal is to talk in a one-on-one channel with Errbot, and use a command like the one shown in Figure 6-8 to make Errbot say something in a team channel.

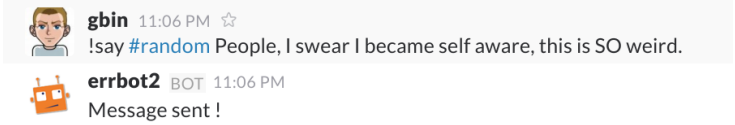


Figure 6-8. Say command.

The message shown in Figure 6-8, for example, will make the bot say the following in `#Random` (see Figure 6-9).

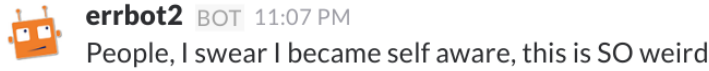


Figure 6-9. Say command result.

The code structure of the `say.py` module for this trick is pretty simple:

```
from errbot import BotPlugin, botcmd


class Startle(BotPlugin):

    @botcmd(split_args_with=' ')
    def say(self, _, args):
        """ Make Errbot say something to a room or a person.
            ie. !say #General I think I am self conscious. or
            !say @gbin Hello !
        """
        if len(args) < 2:
            return 'The format is !say #room something.'

        identifier = args[0]
        try:
            self.send(self.build_identifier(identifier),
                      ' '.join(args[1:]))
        except:
            return 'Cannot find room or person %s' % identifier
        return 'Message sent !'
```

`split_args_with` makes Errbot parse the input of the incoming message, instead of just sending it as a string in `args`. Errbot will

send it directly as a list of strings cut by the character you gave as a parameter.

Then, you can check whether `args` contains at least an identifier and a message.

We are assuming the first parameter is the identifier, so you need to convert this textual representation of the identifier into a real identifier with `build_identifier`. This will give you an object of type `Identifier` or `Room` that you can use in the send method.

HACK 13 No need to use self.send to reply

> When they want to reply to a message from Errbot, a lot of Errbot plugin designers use … `self.send(msg.frm, "my message")` which is overly complex and even buggy in some cases. Instead, you can simply use `return "my message"` and Errbot will reply to the command with the message returned as a string.

This hack has shown you how easy it is to translate text identifiers like @gbin to identifiers Errbot expects, which allows you to send messages.

Now let's see how to can persist (store) objects on disk.

HACK 14 Create polls with Errbot

> In this hack, we will create a plugin that serves as a virtual voting booth; in the process, we will explore how to use Errbot's persistence feature. The goal of this hack it to be able to define a poll, such as "Which restaurant should we go to for lunch?", let Errbot record all of the options, and finally tally the votes.

You can install a complete implementation of this plugin by sending the bot a private message: `!repos install err-poll`. We will walk you through all of it here so you can understand how to implement your own polls in Slack.

Datastructure

First, we need to determine how to persist the data for the plugin. We need to persist a list of polls and their current options, plus current tallies.

At the root of the persistence you have:

Entry	Type
polls	poll_name → PollEntry [dictionary]
current_poll	poll_name [str]

Create a file called `poll.py` that will the Python module for this plugin. With PollEntry holding the options, the current counts and who has already voted, the code in `poll.py` looks like this:

```
from typing import List, Mapping
from errbot.backends.base import Identifier

class PollEntry(object):
    """
    This is just a data object that can be pickled.
    """
    def __init__(self):
        self._options = {}
        self._has_voted = []

    @property
    def options(self) -> Mapping[str, int]:
        return self._options

    @property
    def has_voted(self) -> List[Identifier]:
        return self._has_voted
```

Storing objects with Errbot is very easy. The plugin itself, represented by `self` in Python, is a persistent dictionary. You can use the syntax `self['key'] = value` to store a value at a key.

Creating a poll

Now that you've created the plugin's basic structure, let's make the `!poll new` command. (We will ignore all of the error conditions to simplify the implementation.) Here is `poll.py` with this `!poll new` creation command:

```
class Poll(BotPlugin):
    # activate is called every time the plugin is loaded.
    def activate(self):
        super().activate()

        # initial setup
        # preallocate the 2 entry points to simplify the code below.
        if 'current_poll' not in self:
            self['current_poll'] = None
        if 'polls' not in self:
            self['polls'] = {}

    @botcmd
    def poll_new(self, _, title):
        with self.mutable('polls') as polls:   # see below.
            polls[title] = PollEntry()

        self['current_poll'] = title
        return 'Poll %s created.' % title
```

Here we use `with self.mutable('polls') as polls:` because the entries on self are *only persisted when they are written directly*. The `with` construct is equivalent to:

```
polls = self['polls']
polls[title] = PollEntry()
self['polls'] = polls   # This will persist 'polls'
```

So, now you should be able to test the command by running `!poll new restaurants` and it should create a new poll.

Adding an option

Now we need to add an option to the poll in `poll.py`:

```
    @botcmd
    def poll_option(self, _, option):
        current_poll = self['current_poll']
        with self.mutable('polls') as polls:
            poll = polls[current_poll]
            poll.options[option] = 0

            return '%s:\n%s' % (current_poll, str(poll))
```

This code retrieves the current poll, adds an option with 0 votes, and displays the current state of the poll on chat. You should now be able to define polls as shown in Figure 6-10.

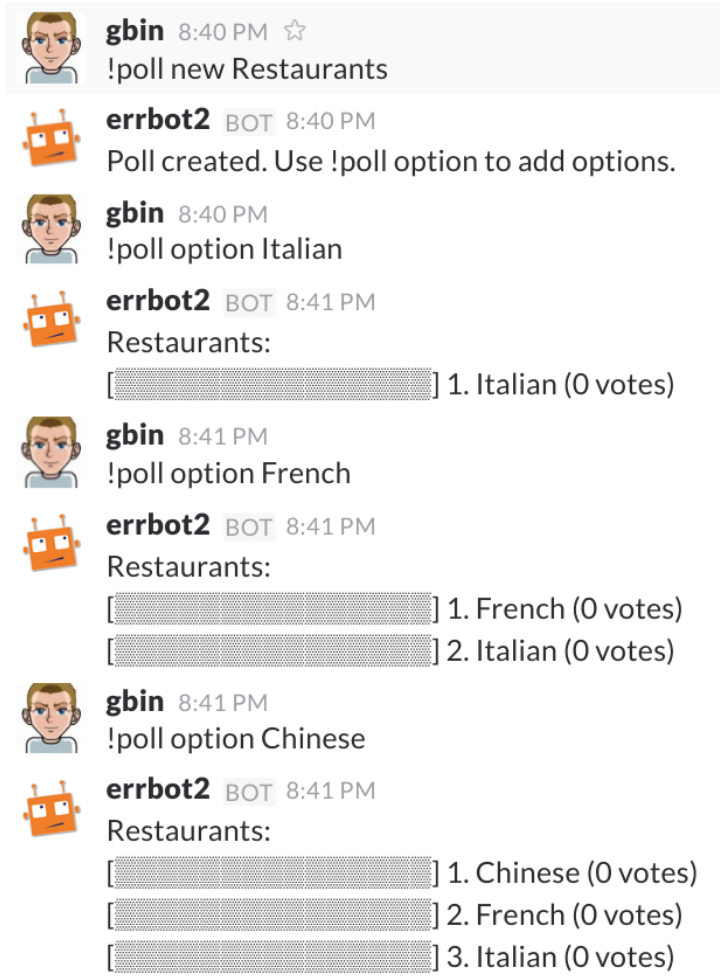


Figure 6-10. Defining a poll, made on the err-poll plugin with the bars counting the votes.

Enabling voting

Let's implement vote counting in `poll.py` with the `!vote` command:

```
    @botcmd
    def vote(self, msg, index):
        current_poll = self['current_poll']
        index = int(index)
        with self.mutable('polls') as polls:
            poll = polls[current_poll]
```

```
        # msg.frm is of type Identity and they are guaranteed to be
        # comparable.
        if msg.frm in poll.has_voted:
            return 'You have already voted.'

        # you can also persist Identity types
        poll.has_voted.append(msg.frm)
        # keys are in random orders, sorted helps to get a constant
        # one.
        option = sorted(poll.options.keys())[index - 1]
        poll.options[option] += 1
        return '%s:\n%s' % (current_poll, str(poll))
```

This code gets the entry index for the user who is talking to the bot (msg.frm), and then check whether the user has already voted. If not, we record his/her vote, the fact that s/he has voted, and display the current state of the votes.

Let's try recording some votes (see Figure 6-11).

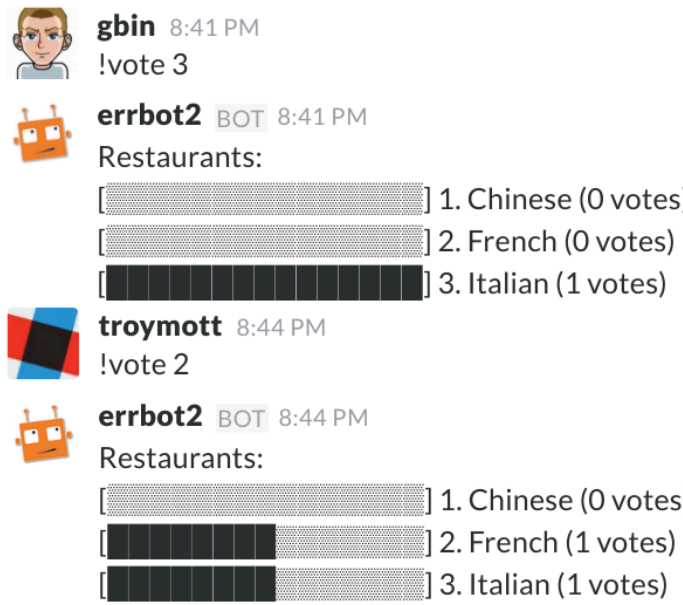


Figure 6-11. Voting.

Feel free to customize the code in this hack so you can craft your own polls in Slack.

HACK 15 Ambush a colleague with Errbot

The goal of this hack is to store a message that will be delivered to a Slack user as soon as she logs into Slack. The bot will contact her and send her a Slack attachment that includes your text and a red mark for dramatic effect.

Create the command

To get started, create a Python module in a new file called `ambush.py`. Next, create an `!ambush` command that simply records who the bot should contact and why. Here's code for the command in `ambush.py`:

```
from errbot import botcmd, BotPlugin


class Ambush(BotPlugin):

    @botcmd(split_args_with=' ')
    def ambush(self, msg, args):
        if len(args) < 2:
            return '!ambush @gbin emergency !! contact @growbot
              as soon as possible'
        idstr = args[0]
        if not self.build_identifier(idstr):
            return ('Cannot build an identifier'
                    'on this chat system with %s' % idstr)
        self[idstr] = (msg.frm, ' '.join(args[1:]))
        return 'Waiting for %s to show up...' % idstr
```

Assuming you've read the previous hacks (#35-#38), you should be familiar with `build_identifier` and persistence.

Wait for a user's presence status to change

Let's add a special callback method that will be called every time a user changes his or her presence (online, offline, away, etc.). Here is the implementation of `callback_presence` in `ambush.py`:

```
from errbot.backends.base import Presence, ONLINE

# this is conventional and defined in BotPlugin
def callback_presence(self, presence: Presence):
    # str(identifier) is the exact opposite of self.build_identifier
    idstr = str(presence.identifier)
```

```
        # test if it is the presence we are interested in.
        # status gives a string representing the "online",
        # "offline", "away" status of the user.
        if presence.status is ONLINE and idstr in self:
            # retrieve back the message
            frm, text = self[idstr]
            # red for the dramatic effect.
            self.send_card(to=presence.identifier, body=text, color='red')
            del self[idstr]    # it is done, cancel the alert.
```

In action

See here our plugin in action (see Figure 6-12).

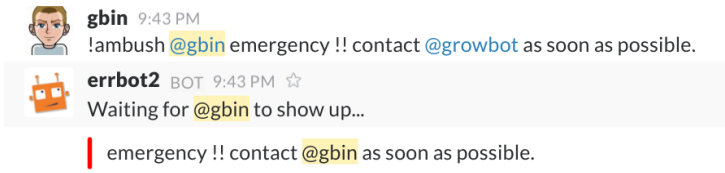


Figure 6-12. Testing !ambush.

This was a very simple example, but you can make a lot of variations on this hack depending on how your team and their Slack channels are organized.

HACK 16 Generate XKCD-style charts with Errbot

One of the main advantages of Errbot being writen in Python is its huge ecosystem of bindings and libraries (see Figure 6-13).

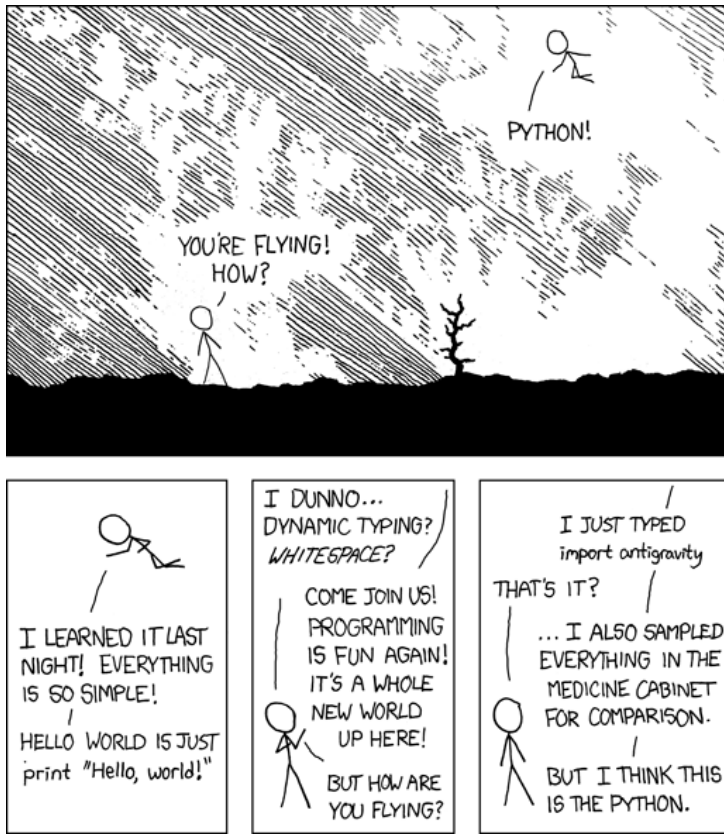


Figure 6-13. Obligatory XKCD reference.

In this hack, we will see how to use Matplotlib to generate fun charts like the one shown in Figure 6-14.

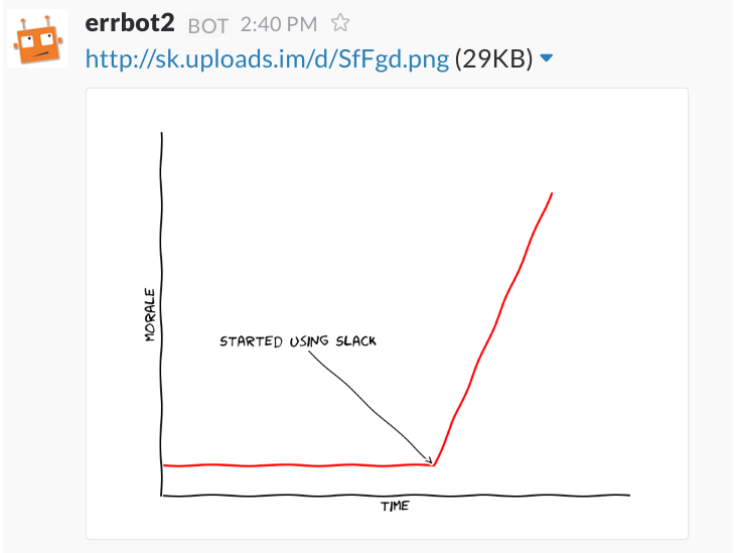


Figure 6-14. Example XKCD-style chart.

This is fun, but this hack can be easily adapted for more serious uses like infrastructure monitoring, scientific applications, and more.

> To install a complete version of this plugin, run the command `! repos install err-xkcdcharts`.

The base files to start with

The plugin we're going to create in this hack will be structured like any other plugin. We will need some dependencies in `requirements.txt`:

```
matplotlib
requests
```

`requests` is a popular Python library. It will be used to publish the image of the chart on the web so it can be displayed in Slack. `matplotlib` is a popular scientific charting library.

You can create the `xkcdcharts.plug` and `xkcdcharts.py` files manually, or you can generate them with the command `errbot --new-plugin`.

Let's begin by manually creating an ini `xkcdcharts.plug` file with this content:

```
[Core]
Name = XKCDCharts
Module = xkcdcharts.py

[Documentation]
Description = Draw XKCD looking charts in your chatroom.

[Errbot]
min = 4.2.0
```

Then create a base module in `xkcdcharts.py` with this content:

```
import io

from errbot import BotPlugin, arg_botcmd  # Errbot base.
import requests   # used for image upload
import matplotlib
matplotlib.use('Agg')  # initializes matplotlib in "headless" mode
import matplotlib.pyplot as plt  # needs to be done after matplotlib.use
plt.xkcd()


class Charts(BotPlugin):
    pass
```

After running `pip install -r requirements.txt` we are ready to implement the heart of the plugin.

The first thing we need to do is to generate a .png in memory and upload it to an image provider. In our code (shown below), we used *http://uploads.im* for the sake of simplicity, but you can also use Google Cloud Storage or Amazon S3 for that. Add the save_chart method to the Charts class `xkcdcharts.py` like this:

```
# .. in class Charts ...
    def save_chart(self, plt):
        with io.BytesIO() as img:
            plt.savefig(img, format='png')
            img.seek(0, 0)
            req = requests.post('http://uploads.im/api',
                                files={'upload': ('chart.png', img)})
            res = req.json()
```

```
        url = res['data']['thumb_url'].replace('.im/t/', '.im/d/')
        return url
```

This method simply takes a matplotlib plot, renders it in memory in a file-like object `img`, and then uses the API of uploads.im (*http://uploads.im/api*) to store it on that site. The upload API call provides the final name of the file, and then we can construct a direct URL to the PNG, mimicking what the uploads.im frontend is doing.

As a parameter for our Errbot command, we will need to parse plenty of x,y pairs. Let's get that out of the way with a small `util` function:

```
def parse_tuple(t: str) -> (int, int):
    x, y = t.split(',')
    return int(x), int(y)
```

Now, we are ready for a first simple charting helper for an xy chart. This helper takes a list of coordinates and graphs them on a 120x120 chart:

```
from typing import Sequence
[...]
    # in class Charts
    def _xy(self, coords:Sequence[str]):
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        ax.spines['right'].set_color('none')
        ax.spines['top'].set_color('none')
        plt.xticks([])    # remove the ticks
        plt.yticks([])
        ax.set_xlim([0, 120])
        ax.set_ylim([0, 120])
        xs, ys = zip(*(parse_tuple(coord) for coord in coords))
        plt.plot(xs, ys, 'r')
        return self.save_chart(plt)
```

Here is a minimal version of a charting command using this helper:

```
    # still in class Charts
    @arg_botcmd('coords',
                metavar='coords',
                type=str,
                nargs='*',
                help='coordinates to plot a line in x1,y1 x2,y2'
                     'separated by space')
    def xy(self, _, coords=None):
        return self._xy(coords=coords)
```

This will define a command !xy that takes a series of coordinates like 12,23 14,56 (with no space after the comma), so you can make arbi-

trary charts. Let's try out our new command with a ramdom series (see [Figure 06-16]).

```
>>> !xy 0,10 30,40 50,30 70,70 90,50 100,100
```

Figure 6-15. Our first XKCD-style chart.

Now we can add the ability to include notes and labels to our graphs by improving the _xy method as shown below:

```
    def _xy(self, coords: Sequence[str]=None, note=None, xlabel=None, ylabel=None
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        ax.spines['right'].set_color('none')
        ax.spines['top'].set_color('none')
        plt.xticks([])
        plt.yticks([])
        if xlabel:
            plt.xlabel(xlabel)
        if ylabel:
            plt.ylabel(ylabel)
        ax.set_xlim([0, 120])
        ax.set_ylim([0, 120])
        if coords:
            xs, ys = zip(*(parse_tuple(coord) for coord in coords))
            plt.plot(xs, ys, 'r')
```

```
        if note:
            note_msg, xy, xy_txt = note
            plt.annotate(note_msg, xy=parse_tuple(xy), arrowprops=dict(arrowstyle
        return self.save_chart(plt)```
```

We can now add a command to graph a funny, hardcoded, upward trend with a label:

```
@arg_botcmd('note', type=str, nargs=1, help='Why it is going down ?')
@arg_botcmd('--xlabel', dest='xlabel', type=str, nargs=1, help='label for the
@arg_botcmd('--ylabel', dest='ylabel', type=str, nargs=1, help='label for the
def upchart(self, _, note=None, xlabel=None, ylabel=None):
    """
    Just a canned case of xy with those parameters:
       !upchart "your message" Message [--xlabel time] [--ylabel money]
       !xy 0,100 70,100 100,0 --note [your message] 70,100 15,50
    """
    return self._xy(coords=('0,10', '70,10', '100,100'),
                    note=(note[0], '70,10', '15,50'),
                    xlabel=xlabel[0] if xlabel else None,
                    ylabel=ylabel[0] if ylabel else None)
```

Now, if you send the command to the bot on Slack `!upchart "Started using Slack" --xlabel time --ylabel morale` you should get our initial example, shown in (see Figure 6-16).

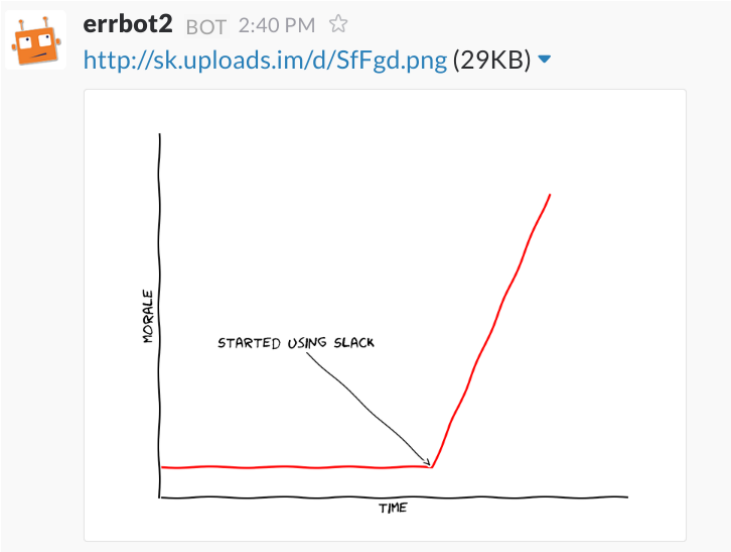


Figure 6-16. Example XKCD-style chart.

This hack has a lot of moving pieces that you can reuse in your plugins. It was all pretty easy to do because we had libraries and friendly APIs for us to use. But what if you don't have libraries and APIs? The next hack will show you how to scrape a website (extract information from it) in order to use it remotely via Errbot commands.

HACK 17 Test code snippets directly in chat with Errbot

In this hack, we will see how to use websites that don't have a public API. This is definitely not a recommended thing to do in normal circumstances, since things might break unexpectedly, but it can be useful for bringing tools that would have been inaccessible to your Slack channel otherwise (such as the Python oneliner shown in Figure 6-17). The goal is to use the service at *http://codepad.org* through Slack and Errbot.

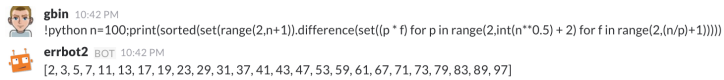


Figure 6-17. Example of Python's powerful oneliners.

Some analysis of the posting page

Go to *http://codepad.org* and right-click and select "Show source" (for Google Chrome) or simply download the page with the command `wget http://codepad.org`.

We are looking for what the page is posting to evaluate the user code entered on the page.

First, we'll need to locate the HTTP form in the page.

Here is an extract of the HTML source we are analyzing with some inline comments:

```
[...]
  <div class="editor" id="editor" >
  <!-- so it posts to the same page "http://codepad.org -->
  <form action="" method="post" id="editor">
  <table cellpadding="10" width="1%">
    <tr>
```

```
    <td colspan="2">
[...]
```

Then we locate in this same HTML source the various posted elements, such as a "lang" field with plenty of options:

```
[...]
  <tr>
    <td style="vertical-align: top">
    <span style="vertical-align:middle" class="label">Language:</span>
   <br/>

    <nobr>
    <label>
        <input style="vertical-align:middle" type="radio"
                  name="lang" value="C" checked="checked"/>
        <span style="vertical-align:middle" class="label">C</span>
    </label>
    </nobr>
     <br/>
    <nobr>
    <label>
        <input style="vertical-align:middle" type="radio" name="lang"
               value="C++"/>
            <span style="vertical-align:middle" class="label">C++</span>
    </label>
    </nobr><br/>
    <nobr>
    <label>
        <input style="vertical-align:middle" type="radio" name="lang"
               value="D"/>
        <span style="vertical-align:middle" class="label">D</span>
    </label>
   </nobr>
   <br/>

[...]
```

Still in this same HTML page, locate the code itself in `code`:

```
[...]
 <td style="vertical-align: middle">
   <textarea id="textarea" name="code" cols="80" rows="15" wrap="off">
   </textarea>
</td>
[...]
```

Then a little bit further in the same page, note the private flag in `private`:

```
[...]
 <input style="vertical-align:middle"
```

```
          type="checkbox" name="private" value="True"
        />
[...]
```

A little later in the same page, note the run flag in `run`:

```
[...]
 <input style="vertical-align:middle"
          type="checkbox" name="run" value="True"
            checked="True"
        />
[...]
```

And finally, later in the page, the expected submit button posting in `Submit`:

```
[...]
<input type="submit" name="submit" value="Submit"/>
[...]
```

Now we have all of the elements needed to emulate the post from Errbot.

Let's code the posting part

Now we can create `code.py`, the main module of the plugin, like in the previous hacks, and start to implement a function to post to the site with the help of the requests library:

```
import requests

def scrape_codepad(code, lang='Python', private=True, run=True):
    data = {'code': code,
            'lang': lang,
            'private': str(private),
            'run': str(run),
            'submit': 'Submit'}
    r = requests.post('http://codepad.org/', data=data)
    if not r.ok:
        return "Failed to contact codepad: %s" % r.content

    return "SUCCESS:\n\n%s" % r.content
```

You should be able to test this function easily in an interactive Python interpreter by running the following commands:

```
$ python
>>> import code
>>> scrape_codepad('print(2+3)')
SUCCESS:

[...]
```

Now we need to parse the resulting HTML.

Back to the drawing board

So first, let's make up some Python test code to try out in codepad.org:

```
for i in range(1, 11):
  print("line %s" % i)
```

Once you have submitted the form, let's try to find the result in the answer (see Figure 6-18).

Link: http://codepad.org/TOL30QuV [raw code | output | fork]

Python, pasted just now:

```
1  for i in range(1, 11):
2    print("line %s" % i)
```

Output:

```
 1  line 1
 2  line 2
 3  line 3
 4  line 4
 5  line 5
 6  line 6
 7  line 7
 8  line 8
 9  line 9
10  line 10
```

New paste:

Language: Python

```
for i in range(1, 11):
  print("line %s" % i)
```

Figure 6-18. Codepad result page.

Using the same technique as before, locate the interesting section of the HTML source we want to capture.

```
<a name="output">
<span class="heading">Output:</span> <-- anchor element to find first !-->
</a>
<div class="code"> <-- 1 -->
<table border="0" cellpadding="10" cellspacing="0">
<tr>
<td style="border-right: 1px solid #ccc;
    text-align: right; vertical-align: top">
```

```
<div class="highlight">
<pre><a style="" name="output-line-1">1</a>
<a style="" name="output-line-2">2</a>
<a style="" name="output-line-3">3</a>
<a style="" name="output-line-4">4</a>
<a style="" name="output-line-5">5</a>
<a style="" name="output-line-6">6</a>
<a style="" name="output-line-7">7</a>
<a style="" name="output-line-8">8</a>
<a style="" name="output-line-9">9</a>
<a style="" name="output-line-10">10</a>
</pre>
</div>
</td>
<td width="100%" style="vertical-align: top">
<div class="highlight">
<pre>
line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
</pre>
</div>
</td></tr></table>
</div>
```

We can see the 10 outputted lines and inline the `heading` span that we will anchor to in order to scrape the page.

We will use the library called Beautiful soup. This library has a very intuitive API to navigate through structures like this. More information about this library can be found at *https://www.crummy.com/software/BeautifulSoup/*.

So, as mentioned ealier, we first find the "anchor element" commented on the source with the `soup.find` method. This anchor element needs to be something that always appears with a specific marker, like the name `output`.

Next, we need to find the first `div` below (1). Then we can start to dive into the structure toward the first `td` of the table. You then find its sibling with `findNext('td')` and finally dive to the `pre` containing the response we are interested in capturing.

This logic looks like the following in the helper function `scrape_codepad`:

```
from bs4 import BeautifulSoup
# [...]
def scrape_codepad(code, lang='Python', private=True, run=True):
    # [...]
    # at the end of scrape_codepad
    soup = BeautifulSoup(r.content, 'lxml')
    output = soup.find('a', attrs={'name': 'output'})
    result = output.findNext('div').table.tr.td.findNext('td').div.pre.text
    return result.strip('\n ')
```

Once a local test works with simple expressions like `print(2+3)`, it is time to integrate with a real Errbot command. Doing so is trivial at that point:

```
from errbot import BotPlugin, botcmd

def scrape_codepad(code, lang='Python', private=True, run=True):
    [...]

class CodeBot(BotPlugin):

    @botcmd
    def python(self, _, args):
        """ Execute the python expression.
            ie. !python print(range(10))
        """
        return scrape_codepad(args)
```

You can try your plugin locally with the command `errbot -T`:

```
$ errbot -T
[,,,]
>>> !python print(3+2)
5
>>>
```

And finally on Slack with the command `errbot`.

This hack showed you how to extract a feature from an existing webpage and expose it as an Ernbot command on Slack. This concludes our series of hacks covering Errbot. Next, we'll use another bot called Hubot.

HACK 18 Connect Hubot to Slack

Hubot is a hugely popular chatbot originally written by the folks at Github. A lot of already-made plugins exist for it. In this hack, we will show you how to set it up.

Prerequisites

You will need for this hack:

1. node.js: visit *https://nodejs.org/* to download an installer for your system
2. update npm by executing the command `sudo npm install npm -g`

Setup

Once you have node.js correctly installed, you can install two helpers (yo and generator-hubot) with this command:

```
$ npm install yo generator-hubot
```

HACK 19 Installing npm packages for the user only or system wide.

By default npm installs the specified packages in the user's directory, but you can use <code>npm -g</code> for a system wide installation (-g for global).

Now that you have the helpers to generate a Hubot, you need to create a working directory for your Hubot instance.

```
$ mkdir myhubot
$ cd myhubot
$ yo hubot
```

Hubot will ask you a series of questions, and you can answer `slack` for the adapter.

You can remove the `hubot-redis-brain` entry from the file `external-scripts.json`. By doing so, you will be able to test Hubot without having to install a Redis database on your development machine.

You should be able to start and test Hubot locally by running the `hubot` command:

```
$ bin/hubot
[...]

myhubot>
```

Note that `myhubot` is the name of your Hubot instance, and this is also the prefix you use to send it commands, as shown in this example:

```
myhubot> myhubot help
Shell: myhubot adapter - Reply with the adapter
myhubot animate me <query> - The same thing as `image me`, except adds a
few parameters to try to return an animated GIF instead.
myhubot echo <text> - Reply back with <text>
myhubot help - Displays all of the help commands that Hubot knows about.
myhubot help <query> - Displays all help commands that match <query>.
myhubot image me <query> - The Original. Queries Google Images for
<query> and returns a random top result.
myhubot map me <query> - Returns a map view of the area returned by
`query`.
myhubot mustache me <url|query> - Adds a mustache to the specified URL
or query result.
myhubot ping - Reply with pong
myhubot pug bomb N - get N pugs
myhubot pug me - Receive a pug
myhubot the rules - Make sure hubot still knows the rules.
myhubot time - Reply with current time
myhubot translate me <phrase> - Searches for a translation for the
<phrase> and then prints that bad boy out.
myhubot translate me from <source> into <target> <phrase> - Translates
<phrase> from <source> into <target>. Both <source> and <target> are
optional
```

Feel free to try these preinstalled commands.

Connecting Hubot to Slack

To connect Hubot to Slack, we will use the preconfigured app on Slack. Go to the settings of your Slack channel and click on "Add an app or integration," as shown in Figure 6-19.

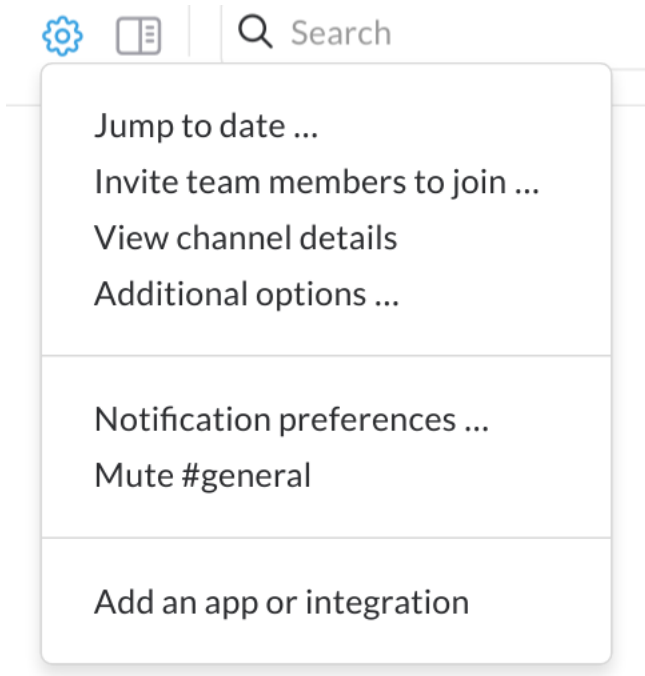


Figure 6-19. Add an app or integration

Search for the app called **Hubot** in the list, and install it for your team. This will start a small wizard where you will be able to choose a Slack username for your bot (see Figure 6-20).

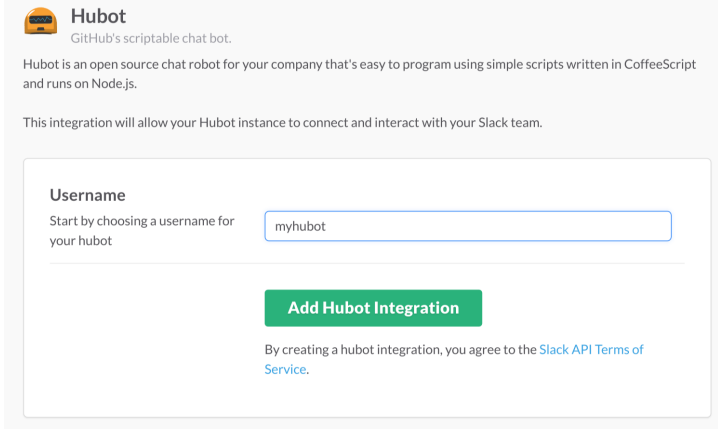


Figure 6-20. Choose hubot's name in the integration settings. The wizard will give you a Slack token.

In order to start and connect Hubot to Slack, you need to set an environment variable containing this Slack token, and ask Hubot to use the Slack adapter with the parameter `-a`:

```
$ HUBOT_SLACK_TOKEN=xoxb-48137346052-l3AKizOPnsunHka9qXFLtpHS \
  bin/hubot -a slack

[Fri Jun 03 2016 16:25:45 GMT-0700 (PDT)] INFO Connecting...
[Fri Jun 03 2016 16:25:46 GMT-0700 (PDT)] INFO Logged in as
            myhubot of SlackHackBook, but not yet connected
[Fri Jun 03 2016 16:25:46 GMT-0700 (PDT)] INFO Slack client now
            connected
[...]
```

Now invite Hubot into a channel from Slack (see Figure 6-21).

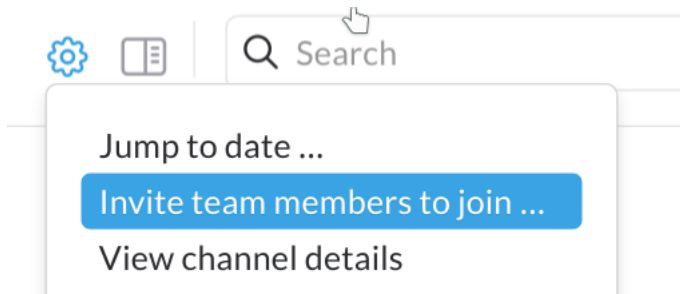


Figure 6-21. Invite Hubot to your team..

And invite Hubot into a team as well (see Figure 6-22).

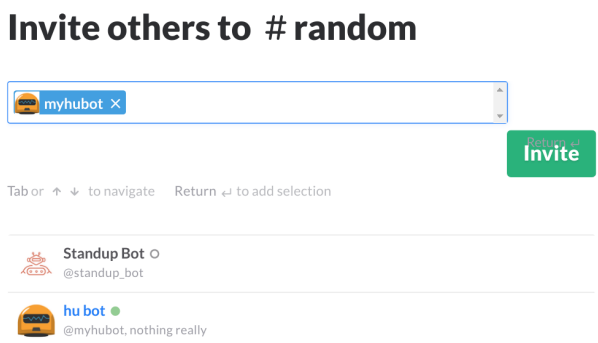


Figure 6-22. Invite Hubot to your team (cont'd).

Now that you have a Hubot instance installed, let's see how to implement a plugin for it.

HACK 20 Create a "Hello World" plugin on Hubot

This hack will get you started developing plugins for Hubot.

Implementing your first plugin for Hubot

By default, the scripts directory, created during the Hubot installation, is automatically loaded when Hubot starts up, and you can define a plugin there. Try creating the file `scripts/my.coffee` with this content:

```
module.exports = (robot) ->
  robot.respond /hi/i, (res) ->
    res.send 'Hello world!'
```

If you are not used to the Coffeescript syntax, the project's homepage has a good primer with a translated example in JavaScript: *http://coffeescript.org/*.

Another thing to keep in mind is the asynchronous nature of Node.js: it is programmed only by giving it functions (or callbacks). For example, the Hello World snippet above means, "export an

anonymous function taking a `robot` parameter that will call `.respond`, taking a pattern and another anonymous function, and taking `res` as a parameter in order to call `.send` on it." To know more about Node.js you can visit *http://nodejs.org*.

Testing your first plugin for Hubot

Once your plugin is defined, you can use it directly from the *shell adapter*, the command-line interface to Hubot:

```
$ bin/hubot
myhubot>myhubot hi
myhubot>hello world!
```

Now try the plugin on Slack by starting Hubot like you did at the end of Hack #42:

```
$ HUBOT_SLACK_TOKEN=xoxb-48137346052-l3AKizOPnsunHka9qXFLtpHS \
  bin/hubot -a slack
```

Once the bot is started, you'll be able to interact with it on Slack (see Figure 6-23).

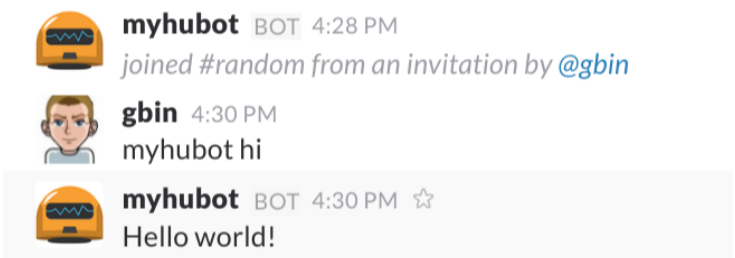


Figure 6-23. Talking to your newly installed Hubot instance on Slack.

Now that Hubot is installed and you know how to create plugins for it, you can implement more complex interactions with Hubot.

HACK 21 Connect Lita to Slack

Lita is a bot written by Jimmy Cuadra that is extensible in Ruby. Since a Ruby environment can be tricky to install and because Redis (a database) is a requirement for Lita, it is recommended that you use a virtual

machine to setup your development environment that will install all the dependencies correctly for you.

In this hack, we will show you how to set up a development environment using Virtualbox, which you will control from the command line with Vagrant (a command-line tool for Virtualbox).

Dependencies

First, you need Virtualbox from Oracle. You can download it and follow the installation instructions from *https://www.virtualbox.org/*. Then you will need Vagrant. You can follow the installation instructions at *https://www.vagrantup.com/*. Once you have Virtualbox and Vagrant installed, you're ready to install Lita.

Boot up your Lita virtual machine

Clone the Vagrant descriptor files with `git clone`:

```
$ git clone https://github.com/litaio/development-environment.git \
        lita-dev
```

Then start the virtual machine with `vagrant up`:

```
$ cd lita-dev
$ vagrant up
```

HACK 22 **Linux dependency on nfsd**

> On Linux, Vagrant will ask to have a running nfs daemon to be able to boot. For example, on Arch Linux, you'll need to install the `nfs-utils` package and run `systemctl start nfs-server.service`, checking the details for your specific distribution if you are missing the package.

Once the virtual machine is up, it will update a series of Docker images:

```
[...]
==> default: Status: Downloaded newer image for litaio/redis:latest
==> default: -- Image: litaio/ruby:latest
==> default: latest: Pulling from litaio/ruby
==> default: 7268d8f794c4: Already exists
==> default: a3ed95caeb02: Already exists
```

```
==> default: a3ed95caeb02: Already exists
==> default: 36315e88ddfc: Pulling fs layer
==> default: a3ed95caeb02:
==> default: Pulling fs layer
==> default: d884667190c7: Pulling fs layer
==> default: d884667190c7:
==> default: Waiting
==> default: a3ed95caeb02:
==> default: Download complete
==> default: 36315e88ddfc: Download complete
==> default: 36315e88ddfc:
==> default: Pull complete
==> default: 36315e88ddfc: Pull complete
==> default: a3ed95caeb02: Pull complete
==> default: a3ed95caeb02:
==> default: Pull complete
[...]
```

From here you should be able to SSH in your new environment with `vagrant ssh`:

```
$  vagrant ssh
CoreOS stable (1010.5.0)
core@lita-dev ~ $
```

Then you can enter the Lita development environment with `lita-dev`:

```
core@lita-dev ~ $ lita-dev
lita@5fac349fcb46:~/workspace$
```

Next, initialize a new Lita project with `lita new`. Doing so will create a base configuration for your bot:

```
lita@5fac349fcb46:~/workspace$ lita new
      create  lita
      create  lita/Gemfile
      create  lita/lita_config.rb
lita@5fac349fcb46:~/workspace$
```

Finish the installation by requesting to install all of the dependencies with `bundle`:

```
$  cd lita
$  bundle
```

You can now talk to the bot by using the `lita` command:

```
$  lita
Lita > Lita, help
Lita, help
Lita: help - Lists help information for terms and command the robot will
respond to.
```

```
Lita: help COMMAND - Lists help information for terms or commands that
begin with COMMAND.
Lita: info - Replies with the current version of Lita.
Lita: users find SEARCH_TERM - Find a Lita user by ID, name, or mention
name.
```

Now that you have Lita up and running, you're ready to connect it to Slack.

Connect Lita to Slack

In the Slack channel of your choice, click the Settings menu (the gear icon), choose "Add an app or integration," select "Brilliant bots," and then choose Lita. From there, the documentation in the wizard will help you with changing the adapter and adding your slack token.

Once you've followed those instructions, you can come back to your Lita development environment and prepare it so it can connect to Slack.

First you need to rerun `bundle` so the new dependency on `lita-slack` is taken into account.

Finally, fire up `lita` and it should connect to Slack:

```
lita@4ac79d5e8890:~/workspace/lita$ lita
fatal: Not a git repository (or any parent up to mount point
/home/lita/workspace)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
[2016-06-03 18:32:51 UTC] INFO: Connected to Slack.
```

Then invite Lita to your channel (see Figure 6-24).

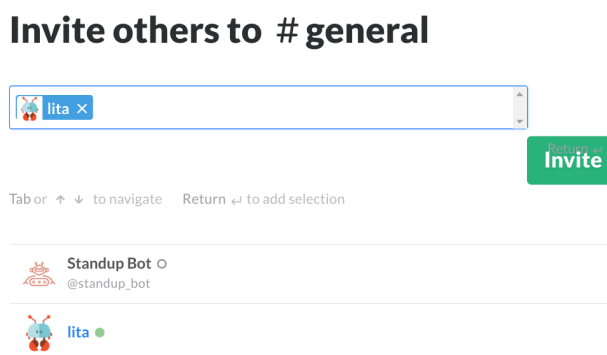


Figure 6-24. Inviting Lita to your channel

Now that you have Lita installed, let's create a *handler*, a type of plugin that adds a chat/command feature.

HACK 23 Create a "Hello World" handler on Lita

Lita has 3 types of plugins:

- **adapters** are pieces of glue code that connect Lita to a specific type of chat system (there is one for Slack, of course, but also for IRC, Hipchat, etc.)
- **handlers** add a chat/command feature.
- **extensions** are advanced plugins that expose transverse features that hook into Lita callbacks (i.e., extending Lita itself).

This hack will show you a minimal version of a handler.

Implementing your first handler

To begin, start a separate shell from the **host** machine in the Git repo you originally cloned from the lita installation. Note how the ~/workspace from the Lita docker container is mounted in the workspace directory in the Git repo. This allows you to use your favorite editor to develop on Lita.

You can come back to your workspace directory with `cd ~/workspace` on the development environment. We are going to ask Lita to scaffold the new handler we are going to develop. By "scaffold," we mean create the various files you'll need to make a minimal handler. This is done via the `lita handler` command:

```
$ lita handler slackhacks
[...]
```

(Lita will ask you a few Jenkins CI and coverage questions. Reply no to each of them.)

This command will create a `lita-slackhacks` Ruby gem that will hold your handler. It comes with to-dos in the gemspec you will need to fix before being able to build the gem. So the next step is to edit the `lita-hello.gemspec` file and set reasonable values like the ones shown here:

```
[...]
  spec.name          = "lita-slackhacks"
  spec.version       = "0.1.0"
  spec.authors       = ["Guillaume Binet"]
  spec.email         = ["gbin@generic.net"]
  spec.description   = "This is to say hello"
  spec.summary       = "This is a summary"
  spec.homepage      = "http://lita.io"
  spec.license       = "GPL"
  spec.metadata      = { "lita_plugin_type" => "handler" }
[...]
```

Now that the gem is specified, it should build and install correctly from the `~/workspace/lita-hello` directory in your **dev** shell:

```
$ bundle
[...]

Using lita-slackhacks 0.1.0 from source at `.`
Bundle complete! 6 Gemfile dependencies, 30 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

You'll need to make the Lita Gem load your handler gem, (rather than look for it on the Web), which you can do by forcing it to load from the disk in the `~/workspace/lita/Gemfile` file:

```
source "https://rubygems.org"

gem "lita"

gem "lita-slackhacks", path: "../lita-slackhacks"
```

This handler doesn't actually do anything yet, so it's time to add a command to your plugin. The scaffolding generated a class called `SlackHacks` that inherits from `Handler` in `lib/lita/handlers/slackhacks.rb`. You can see that it autoregisters at load time with Lita:

```
Lita.register_handler(self)
```

You'll need to add two things to this class to make the handler respond to a command. One is a simple method that gets a response object as parameter. The other is a registration for this method as a command for Lita.

Here is the full file with a simple Hello World response implemented.

```
module Lita
  module Handlers
    class Slackhacks < Handler
      route(/^hi\s+(?<name>.+)/, :hi)

      def hi(response)
        name = response.match_data['name']
        return response.reply('hello ' + name + ' !')
      end

      Lita.register_handler(self)
    end
  end
end
```

Testing our first handler on Lita

To test out your new handler, first you need to restart Lita. You can do that from your workspace by doing the following:

```
lita@4ac79d5e8890:~/workspace/lita$ lita
fatal: Not a git repository (or any parent up to mount point
/home/lita/workspace)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
Type "exit" or "quit" to end the session.
Lita > hi world
hi world
hello world !
Lita > lita, hi world
lita, hi world
hello world !
Lita >
```

You'll notice that Lita will answer either a normal message or when it has been summoned. You can restrict its responses to the latter by adding a `command: true` during the registration of the command.

```
route(/^hi\s+(?<name>.+)/, :hi, command: true)
```

```
Lita > hi toto
hi toto
Lita > lita, hi toto
lita, hi toto
hello toto !
Lita >
```

When you try it out on Slack, it should also respond to your little handler now (see Figure 6-25).

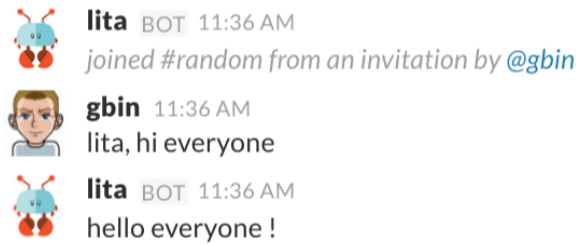


Figure 6-25. Lita working in the chatroom

From here, you can start implementing your own handlers pretty easily. Now let's see how to create a bot in Java.

HACK 24 Prep the Simple Slack API to build bots

The Simple Slack API is written in Java and is freely available on Github *https://github.com/Ullink/simple-slack-api*. It allows Java developers, or other Java VM language developers, to quickly build a fully functional Slack bot that runs as a standalone application. As you'll see, it can also be used to quickly develop some hacks.

Simple Slack API internals

Here's how all Slack bots made with the Simple Slack API work:

1. The bot establishes a websocket connection using the Slack RTM API.
2. Once it's connected, the bot receives various information about the team such as users, channels, etc.
3. Once all of this information has been received, the library keeps a websocket thread running that unmarshalls (deserializes) the Slack events into Java objects. When an event arrives, the library dispatches them to all registered listeners.
4. A heartbeat function is included in the library and is set to automatically monitor the connection status by sending Slack ping requests at regular intervals.

Here's an example of a JSON event sent over the websocket connection:

```
{
  "type": "message",
  "user": "U042PASSS",
  "text": "Hello world",
  "bot_id": "B023PAXME",
  "user_team": "T055KFDH",
  "team": "T055KFDH",
  "user_profile": {
    "avatar_hash": "03f50aa1ce70",
    "image_72": "https:\/\/avatars.slack-edge.com\/2015-03-18\/4aze.jpg",
    "first_name": "Rob",
    "real_name": "Rob Ot",
    "name": "rob_ot"
  },
  "channel": "D023PFEF",
  "ts": "1465298109.000035"
}
```

A bot built from Simple Slack API can react to these events using a set of listeners, and by using the Slack Web API RPC methods to act on the channels: sending a message, creating a channel, pinning a reaction emoji, etc.

Simple to use

By handling the websocket connection, the JSON stream, and API mappings, the Simple Slack API lets you focus on the business logic you want to implement on top of Slack. You shouldn't have to bother about the connection status, or parsing JSON (unless you plan to code some tricky bots), or mastering the HTTP protocol (knowing

about this is helpful but not mandatory). The core class of the API is *SlackSession*, which lets you register listeners or send commands just by calling methods.

Initiating a Slack session

The first thing you need to do is to instruct Simple Slack API to connect to Slack by opening a session. Once opened, this session will be the entry point that you can use to interact programmatically with the Slack team. Here's how to open a session:

```
SlackSession session = SlackSessionFactory.
                       createWebSocketSlackSession("bot-token-here");
session.connect();
```

That's it. With these two lines of code, your bot is connected to a Slack team.

Accessing the team's users and channels

Once the session is opened, you can access the team user directory or browse the public channel list from the newly created session.

Here's how to retrieve a channel:

```
// searching by channel name
SlackChannel channel1 = session.findChannelByName("cool-channel");
// searching by channel id
SlackChannel channel2 = session.findChannelById("C0123456");
// Fetching all the channel list
Collection<SlackChannel> channels = session.getChannels();
```

And here's how to retrieve a user:

```
// searching by user name
SlackUser user1 session.findUserByUserName("john.doe");
// searching by email
SlackUser user2 session.findUserByEmail("john.doe@acme.com");
// searching by user id
SlackUser user3 session.findUserById("U012345");
// Fetching all the user list
Collection<SlackUser> users = getUsers();
// Knowing the user the bot is using to connect to Slack
SlackPersona botId = session.sessionPersona();
```

Ids

In the code used in this hack, you can see some methods for retrieving users or channels if you know their ID. The ID is a unique internal identifier created and managed by Slack.

Interacting with the team

Now that you have an open session and you know how to get the team's channels and users, you can interact with those objects. All of these method use the Slack Web API described here *https://api.slack.com/methods*. Virtually all of the Slack Web API methods are accessible through this library.

Here's how to join a channel:

```
session.joinChannel("slack-hacks");
```

And how to post a message in a channel:

```
SlackChannel channel = session.findChannelById("C0123456");
session.sendMessage(channel, "Hello world");
```

And how to invite a user to a channel:

```
SlackChannel channel = session.findChannelById("C0123456");
SlackUser user = session.findUserByUserName("john.doe");
session.inviteToChannel(channel, user);
```

A bot connected to a Slack client

To get a better understanding of Simple Slack API, think of the Slack bot you are developing as being connected like a standard user using a Slack client. The commands you are sending would behave in the same way if you were using the client feature. For example, joining a channel that doesn't exists creates it, yet posting a message in a channel you have not joined will fail.

Listening to team events

You can make the bot aware of what is going on in the channel. There are dozens of different events to listen to, so let's examine the most interesting ones and see how to easily react on these events.

Here's how to make your bot listen to a posted message:

```
session.addMessagePostedListener(new SlackMessagePostedListener() {
  @Override
  public void onEvent(SlackMessagePosted event, SlackSession session) {
   // put code here that you want to execute on such event
  }
});
```

And here's how you make the bot listen with a lambda expression:

```
session.addMessagePostedListener((_event, _session) -> {
   // put code here that you want to execute on such event
});
```

Your listener will get all of the message events as its first parameter regardless of which channel the message was posted on or which user posted it (including the listening bot itself). So, generally the first thing the listener code will do is to filter out events based on the attributes of channels/users. Those attributes are available on the event object itself.

Listening to channel creation

The example below combines all the things you've seen so far to interact with the team and the channel-creation event listening mechanism. This code shows how to have your bot join each newly created channel (except those it has created itself):

```
session.addchannelCreatedListener((_event, _session) -> {
  if (_event.getCreator().getId().equals(
      _session.sessionPersona().getId())) {
    _session.joinChannel(_event.getSlackChannel().getName());
  }
});
```

Now that you have been introduced to the Simple Slack API, let's put it to use in the next hack.

HACK 25 Build a JSON pretty printer bot

> In this hack you'll learn how to build your first Slack bot using the Simple Slack API library. This bot will *prettify* (reformat nicely) any JSON file that is sent to it. This is a convenient tool to have when you are developing tools around the JSON based protocol, like the Slack RTM API.

In this step-by-step example, we will assume you know how to code in Java and that you have JDK 8 already installed. We will use Gradle (*http://gradle.org/*) as our build tool to put everything together.

Setup

Before you can start building your bot, you need to do some setup:

1. Download the latest distribution of Gradle from *http://gradle.org/gradle-download/*

2. Unzip the distribution into a directory of your choice
3. Create a GRADLE_HOME environment variable
4. Add GRADLE_HOME/bin to your PATH environment variable

After you've performed those seup steps, you should be able to run `gradle` from the command line. Now that you have Gradle set up, you're ready to perform the remaining setup steps:

1. Create a directory where you want to put your bot source files
2. Create a `build.gradle` file with the following content:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.ullink.slack:simpleslackapi:0.6.0'
    compile 'org.apache.httpcomponents:httpclient:4.4'
    compile 'com.cedarsoftware:json-io:4.4.0'
}
```

1. Create the following directory: `src/main/java/com/oreilly/slackhacks`
2. In this newly created directory, create the following file: `JSONPrettyPrinter.java`

   + build.gradle

   + If you are not familiar with Gradle, build.gradle is used to describe the project's structure, its dependencies, and how to retrieve them. So in this hack, you will use the last version of simpleslackapi and json-io, an open source library (*https://github.com/jdereg/json-io*), which provide a simple tool to prettify JSON.

You are now ready to write the bot code. To keep things as simple as possible, everything in his hack will be written in the main method in a single class.

Bot skeleton

We have a clear vision of what the bot is supposed to do (prettify JSON files), so let's try to define a skeleton.

As you've seen, it's quite easy to create a Slack session, add a listener, and connect to a Slack team. So, let's create a skeleton for the bot:

```
public class JSONPrettyPrinter {

private static final String TOKEN = "insert_here_your_bot_token";

public static void main(String [] args) {

//create a session using the token
SlackSession session = SlackSessionFactory.createWebSocketSlackSession(TOKEN);

//add to the session a listener to get informed of all
//message posted that the bot could be aware of
session.addMessagePostedListener(JSONPrettyPrinter::processEvent);

//connect to the Slack team
session.connect();

//make the main thread wait and delegate the event processing
//to the session, dispatching them to the listener
Thread.sleep(Long.MAX_VALUE);
}

}
```

So far, there's nothing new here except the call to the `processEvent` static method. Let's have a closer look at what this method should do.

Processing the event

When the bot receives a *SlackMessagePosted* event, the logic that handles the event is easy to implement:

- Does the event need to be processed? If not, then don't process it
- If the event needs to be processed:
  - — download the file attached to the message
  - — format the JSON
  - — send back a new file containing the formatted JSON
- If an issue occurs during the processing (for example, the bot is unable to download the file, or the file doesn't contain JSON formatted text) then notify the user of this fact.

In Java, this is expressed as follows:

```
private static void processEvent(SlackMessagePosted event,
                                 SlackSession session) {
  // does the event needs to be processed?
  if (!isFileSentFromAnotherUser(event, session)) {
            return;
  }

  //trying to pretty print the file
  try {
    sendBackPrettifiedFile(event, session, formatJson(downloadFile(event)));
  } catch (Exception e) {
    failOnException(event, session, e);
  }
}
```

Determining whether the event needs to be processed

The `isFileSentFromAnotherUser` method serves as a filter that discards every event the bot doesn't have to process. Here are the events the bot should ignore:

- messages without a file attached to them
- messages sent on a non-direct channel
- messages sent by the bot itself

Here is the implementation of this using Java:

```
private static boolean isFileSentFromAnotherUser(SlackMessagePosted event,
                                                 SlackSession session) {
  //if the event is not on a direct channel
  if (!event.getChannel().isDirect())
    return false;

  //if the event was triggered by the bot
  if (event.getSender().getId().equals(
      session.sessionPersona().getId()))
    return false;

  //if the event doesn't contain a file
  if (event.getSlackFile() == null)
    return false;
  //otherwise
  return true;
}
```

Download the file attached to the downloadFile

Whenever a file is referenced in a Slack event, the URL to download it is provided. For security reasons, this URL is not accessible to non-authorized users. In order to download a private Slack file using an HTTP GET request, a bot has to provide its token key in a header of the GET request using this format: `Authorization: Bearer xxxxxx`, where *xxxxxx* is the bot's token.

In this hack, we will use the Apache HttpClient library (*https://hc.apache.org/httpcomponents-client-ga/index.html*) to perform the HTTP GET call with the correct header. Let's see how to download a Slack file in Java:

```
private static String downloadFile(SlackMessagePosted event)
throws IOException {
  //creating a simple http client
  HttpClient client = HttpClientBuilder.create().build();
  // defining a get HTTP request
  HttpGet get = new HttpGet(event.getSlackFile().getUrlPrivate());
  // adding the Authorization header with the bot token
  get.setHeader("Authorization", "Bearer " + TOKEN);
  // send the get request
  HttpResponse response =  client.execute(get);
  // initiate a reader to read the response
  BufferedReader buffer = new BufferedReader(
    new InputStreamReader(response.getEntity().getContent()));
  // collect all the file lines and return the content
  return buffer.lines().collect(Collectors.joining("\n"));
}
```

Format the JSON

As mentioned at the beginning of this hack, we're going to use JSON-IO to format the JSON data, thanks to its `JsonWriter` class, which is straightforward:

```
private static String formatJson(String jsonString) {
  return JsonWriter.formatJson(jsonString);
}
```

Send back a new, formatted JSON file

Here we will use the sendFile method provided by the SlackSession class, giving the name of the file and using the prettified JSON string as data:

```
private static void sendBackPrettifiedFile(SlackMessagePosted event,
                                           SlackSession session,
                                           String formattedJson) {
  //sending a file to the same direct channel, using the String byte
  //array and a new filename: the original filename prefixed with pretty_ as
  session.sendFile(event.getChannel(),formattedJson.getBytes(),
                   "pretty_"+event.getSlackFile().getName());
}
```

Handling processor errors

If the file can't be downloaded or it isn't a JSON file, an exception will be thrown, and the bot has to notify the user it wasn't able to process the file. Sending a message is the simplest way to do that:

```
private static void failOnException(SlackMessagePosted event,
                                    SlackSession session, Exception e) {
  //we're responding on the direct channel uing a simple message
  // that an issue occured during the file processing
  session.sendMessage(event.getChannel(),
          "Sorry I was unable to process your file : " + e.getMessage());
}
```

And you're done. Putting it all together will take less than 100 lines of Java code. You just have to build a JAR with all of the dependencies with the standard tool (*https://docs.oracle.com/javase/tutorial/deployment/jar/build.html*) or your favorite IDE.

Let's look at how this bot behaves. We took the JSON extract we presented during the introduction of this library and removed all of the formatting. The result is shown in Figure 6-26.

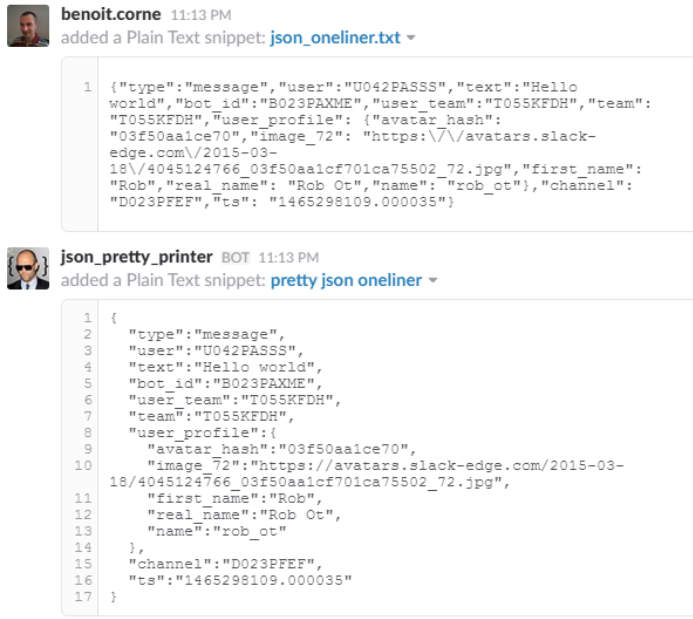


Figure 6-26. Pretty printing json with the Slack bot

This hack showed you how to build a bot to prettify any JSON posted to it. Let's now see a hack that dispatches customer support requests through Slack.

HACK 26 Dispatch customer support through Slack with Smooch

> Thanks to its channel features, Slack is a good way to centralize the communication within your organization. But you can't invite all of your customers to your Slack team. In most cases customer support is done through another medium (email, forum, instant messaging).

Smooch provides a service that lets you link your customer support tool to a Slack team channel. This is a nice integration, but the channel can be very busy and your customer service department can be quickly mix up messages trying to answer them. A solution to this issue is to add to this Smooch integration a Slack bot that dispatches

requests. In this hack, we'll show you how to integrate Smooch with Slack and add the dispatch bot.

Installing Smooch

You first have to create an account on the smooch website: *https://smooch.io/*.

Here are the steps you'll need to follow during this creation: 1. Once your email is confirmed, you will need to configure your smooch account by answering a few questions asked by a robot. 2. Select the way your customers are reaching your customer support. 3. Select the solution the customer support team will use to receive customer requests.

In this hack, emails are used as the method for reaching customer support. Smooch will create an email address that forwards the messages sent to it to your Slack channel. Alternatively, you can use your own support email by enabling the email forwarding in the smooch configuration screen (see Figure 6-27).

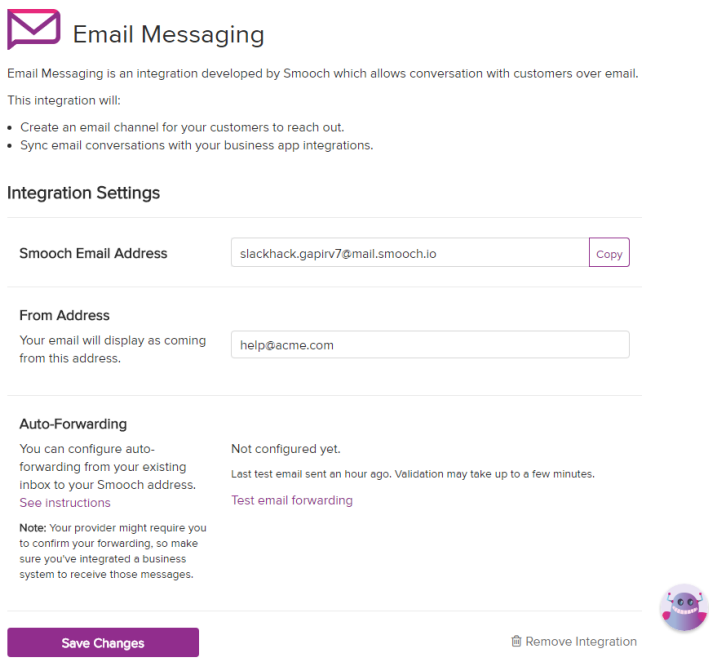


Figure 6-27. Smooch's email settings

By default, on Slack, the messages are sent to the *#general* channel, but you can choose a different channel in Smooch's Slack settings (see Figure 6-28).

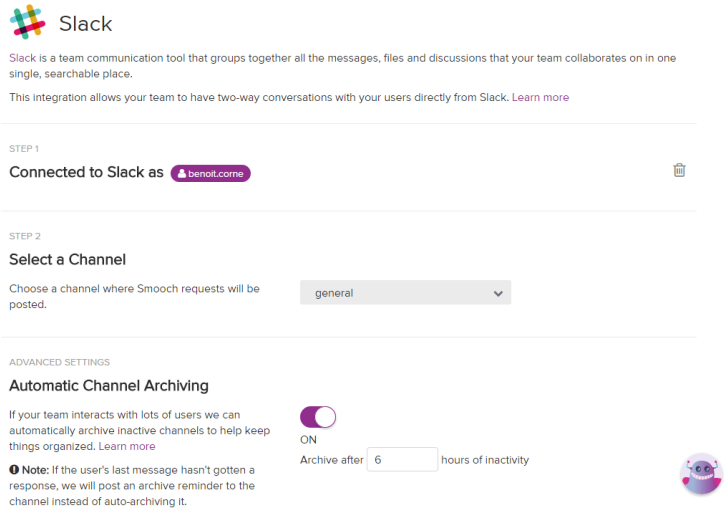


Figure 6-28. Smooch's Slack settings

Standard Smooch use case

Now that Smooch is setup, let's see what happens when you receive a customer support request. Imagine a customer sends an email to customer support:

```
to: help@acme.com
Subject : need help
Content :
hey, I need help
```

On Slack, the customer support team receives a message notifying them that the customer sent an email and that a dedicated channel has been created to deal with it (see Figure 6-29).

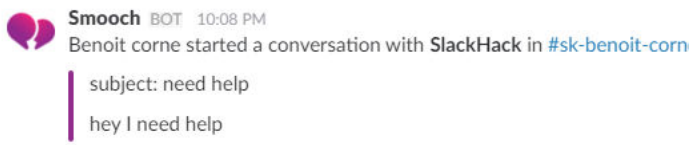


Figure 6-29. Customer support receives a message from smooch

As you can see in Figure 6-30, Smooch creates a channel for the answer.

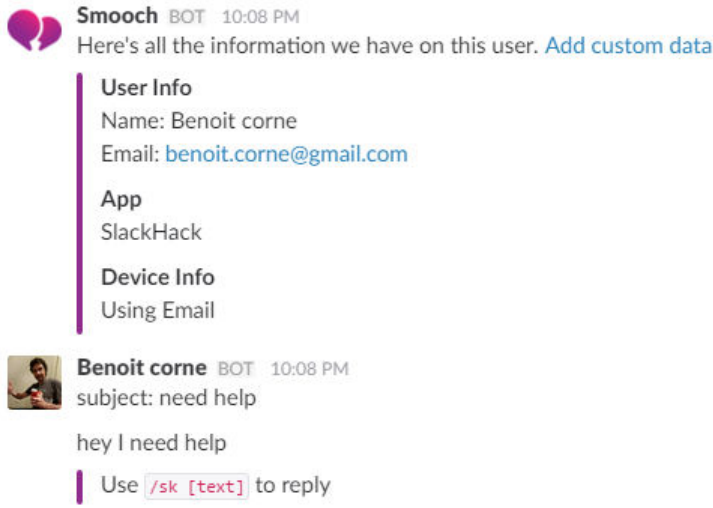


Figure 6-30. Smooch creates a dedicated channel to answer the customer

A customer support representative can then answer the customer's request using the /sk command:

```
/sk Hello Sir, could you please elaborate? Best regards, John Smith
```

And the customer instantly receives the following email:

```
from: help@acme.com
Subject : Re: need help
Content :
Hello Sir, could you please elaborate? Best regards, John Smith
```

Leveraging Slack's history feature

The Smooch Slack integration will create a channel for each new user and will auto archive the channel after a specific duration of inactivity (6 hours by default). If the same user sends a new request a week later, the archived channel will be unarchived and all the history from past requests will be kept.

Creating the customer support dispatcher

Now that you have Smooch up and running, we're going to create a Slack bot dispatcher in order to distribute the incoming customer support requests. The dispatcher will report who should take care of the incoming customer support requests and send a message to the customer support team member notifying him of the request he or she has to handle. This reduces the risk of having a question go unanswered or having more than one customer service rep respond.

The Slack bot will be developed in the Java language using the Simple Slack API library. We won't detail the Java environment setup procedure since everything will fit into a single class.

Slack bot skeleton

Here is the bot's skeleton:

```
private static final String TOKEN = "insert your token here";

public static void main(String [] args) throws Exception {
 //creating the session
 SlackSession session = SlackSessionFactory.createWebSocketSlackSession(TOKEN);
 //adding a message listener to the session
 session.addMessagePostedListener(
                CustomerSupportDispatcher::processMessagePostedEvent);
 //connecting the session to the Slack team
 session.connect();
 //delegating all the event management to the session
 Thread.sleep(Long.MAX_VALUE);
}
```

As in the previous hack, we reproduce the same main structure as the JSON pretty printer, and all of the interesting code is in the message processor: the `processMessagePostedEvent` method.

Message processor structure

Here's how the message processor is structured:

```
private static void processMessagePostedEvent(SlackMessagePosted event,
                                              SlackSession session) {
//filtering all the messages the bot doesn't care about
if (isNotASmoochMessage(event)) {
 return;
}
//notifying the CS member
notifyCustomerSupportMember(event,
                            selectCustomerSupportMember(event,session),
```

```
                                    session);
    }
```

The processor is divided into three steps:

1. Filter out the messages not coming from Smooch
2. Select a customer support representative
3. Notify the rep that he has an incoming request to process

Message filtering

In this method, some validation on the event is done to discard events we don't want to process; verifying the sender's name and message type should be sufficient:

```
private static boolean isNotASmoochMessage(SlackMessagePosted event) {
 //if the event is not a bot message
 if (event.getMessageSubType() != BOT_MESSAGE) {
  return true;
 }
 //if the sender is not named Smooch
 if (!"Smooch".equals(event.getSender().getUserName())) {
  return true;
 }
 return false;
}
```

Selecting a customer support rep

For this step we've chosen to keep things simple, so the selection will be made randomly among all of the users in the support channel that are not bots and are currently active:

```
private static SlackUser \
     selectCustomerSupportMember(SlackMessagePosted event,
                                 SlackSession session) {
 SlackChannel channel  = event.getChannel();
 Collection<SlackUser> users = channel.getMembers();
 List<SlackUser> selectableUsers = users.stream()
       .filter(user -> !user.isBot() && \
        session.getPresence(user) == ACTIVE)
       .collect(Collectors.toList());
 return selectableUsers.get((int)Math.random()*selectableUsers.size());
}
```

Notifying the rep of an incoming request

In this step, we first get the channel reference that will be used to send messages. Smooch writes this reference in the attachment it posts, so we need to parse the underlying event JSON to get the attachment details. (We won't go over the details of the JSON parsing—you can check it out in this hack's source code on GitHub.)

After this parsing, the Slack bot will send a message to the support channel to notify the other reps that a team member has been selected to handle the issue. It also sends a direct message to the selected team member to inform him/her that he/she has to take care of this request. This code shows the flow that is implemented from the notification to the customer support member:

```
private static void notifyCustomerSupportMember(SlackMessagePosted event,
                                                 SlackUser nominee,
                                                 SlackSession session) {
 String channelReference = getCreatedChannelReference(event);
 session.sendMessage(event.getChannel(), "<@" + nominee.getId() + ">:" +
                     " will handle the issue in " + channelReference);
 session.sendMessageToUser(nominee,
                     " Could you please handle the issue in " +
                     channelReference,null);
}
```

This hack showed you how to implement a more complex interaction between your Slack bot and your users. This is a good base to build upon; for example, by adding a persistence layer, you can extend this bot to store interesting statistics like how many requests a support assignee has been asked to handle, or the most common keywords in those customer requests.

Now let's see how to make a training bot.

HACK 27 Create a bot to check your multiplication skills

You would probably like to train your brain, so why not have a bot to ask you the results on multiplication tables, measuring the total time you need to give the correct answers on ten multiplication sets? We will produce such a Slack bot in this hack. We'll do this using the Simple Slack API framework and Java.

Slack bot skeleton

This hack uses the same bot skeleton you saw in hack #48. We're going to create a session, register a message post listener, and connect the session to the Slack server.

Message processor structure

Here is the event handler:

```
private static void processMessagePostedEvent(SlackMessagePosted event,
                                              SlackSession session) {
 handleTestTrigger(event, session);
 handleAnswer(event, session);
}
```

The processor has to handle two kinds of messages:

- detecting a keyword to launch the test
- detecting an answer to a running test

The following sections explain how to do so.

Handling test launch requests

To launch a test, users have to send a *!times* command to the bot through a direct message. The bot will then check whether the user is currently running a test, and if not, will store the test variables in a GameSession instance (keeping track of time spent to answer, and some other values). Then it starts the test. Here's the code that performs all of these steps:

```
private static void handleTestTrigger(SlackMessagePosted event,
                                      SlackSession session) {
 //trigger is accepted only on direct message to the bot
 if (!event.getChannel().isDirect()) {
  return;
 }
 //looking for !times command
 if ("!times".equals(event.getMessageContent().trim())) {
  // check whether a game is already running with this user,
  // and if so, ignore this command
  if (gameSessions.containsKey(event.getSender().getId())) {
   return;
  }
  GameSession gameSession = prepareGameSession(event, session);
  gameSession.timer.start();
  sendTimes(gameSession, session);
 }
}
```

Handling test answers

Answers are numbers only and are sent on the direct channel between the user and the Slack bot. If the user hasn't started a test and sends the bot a numeric message, the bot should ignore that message. If a number is given during a test, then the bot will react according to whether the answer is true or false.

Here how these guidelines translate into code:

```
private static void handleAnswer(SlackMessagePosted event,
                                 SlackSession session) {
 //no test launched for this user
 GameSession gameSession = gameSessions.get(event.getSender().getId());
 if (gameSession == null) {
  return;
 }
 //an answer should be given on a direct channel
 if (!event.getChannel().isDirect()) {
  return;
 }
 //an answer is a number
 String answerValue = event.getMessageContent().trim();
 try {
  int resultGiven = Integer.parseInt(answerValue);
  if (resultGiven == gameSession.goodResult) {
   //correct answer
   goodAnswer(event, session, gameSession);
  } else {
   wrongAnswer(event, session);
  }
 } catch (NumberFormatException e) {
  //ignore the result
  return;
 }
}
```

Sending a test question

When the test starts or when a user gives a correct answer, the bot randomly selects two numbers between 1 and 10 and sends the multiplication problem to the user as a message:

```
private static void sendTimes(GameSession gameSession,
                              SlackSession session) {
 //select two values to multiply
 int a = 1 + (int) (Math.random() * 10);
 int b = 1 + (int) (Math.random() * 10);
 gameSession.goodResult = a * b;
 gameSession.questionTimestamp = session.sendMessageToUser(
```

```
            gameSession.user, a + " x " + b, null).getReply().getTimestamp();
    }
```

Correct answer behavior

The following code tells the bot to indicate a correct answer by pinning a checkmark emoji on it. The bot then computes the time spent to answer and moves to the next test step (either another multiplication problem or the end of the test):

```
private static void goodAnswer(SlackMessagePosted event, SlackSession session,
                               GameSession gameSession) {
 session.addReactionToMessage(event.getChannel(), event.getTimeStamp(),
                              "white_check_mark");
 computeTime(event, gameSession);
 nextTestStep(session, gameSession);
}
```

Wrong answer behavior

As with a correct answer, the bot pins an emoji to the user's response, but this time a red *X* is used to show the user the answer was wrong:

```
private static void wrongAnswer(SlackMessagePosted event,
                                SlackSession session) {
 session.addReactionToMessage(event.getChannel(), event.getTimeStamp(), "x");
}
```

Ending the test

At the end of the test, the Slack bot displays the total time spent to give the ten answers, and removes the game session to allow the user to start a new test.

```
private static void showTestResults(GameSession gameSession,
                                    SlackSession session) {
 session.sendMessageToUser(gameSession.user, "You took " +
    gameSession.cumulativeTime + " seconds to complete the test", null);
 gameSessions.remove(gameSession.user.getId());
}
```

In this hack, we saw how to interact with users and maintain a state machine to keep track of the progress of a game. Now, let's next see how we can use the Simple Slack API to run a "quiz" on a channel.

HACK 28 Create a bot to run a quiz on a channel

> Your team will likely have a Slack channel dedicated to entertainment, fun or challenges. Why not setup a bot to ask the questions? It could be an impartial referee and it could enforce a time constraint on the responses. This hack—which will be developed in Java—shows you a simple way to develop such a Slack bot using the Simple Slack API framework. The bot will send questions and propose four possibles answers.

Any user on the channel can give one answer to each question. The first one to give the right answer is awarded a point. If the right answer is not given within 10 seconds, the bot asks a new question. By the end of the question session, the bot displays a scoreboard for all of the users who have answered at least one question.

Slack bot skeleton

This bot uses the same pattern as the previous bot (Hack #49), using a similar framework. There's one additional step in this hack—loading the quiz questions—but that is not really related to Slack. It is just a matter of retrieving question definitions from a JSON file (see this book's Github repository for an example).

The main entry point looks like this:

```
public static void main(String[] args) throws Exception {
 //loading allQuestions
 allQuestions = loadQuestions();
 //creating the session
 SlackSession session = SlackSessionFactory.createWebSocketSlackSession(TOKEN);
 //adding a message listener to the session
 session.addMessagePostedListener(QuizzBot::processMessagePostedEvent);
 //connecting the session to the Slack team
 session.connect();
 //delegating all the event management to the session
 Thread.sleep(Long.MAX_VALUE);
}
```

Message processor structure

The processor has to handle two kinds of messages: requests for a quiz and answers during a quiz. Here's the code that lets it do so:

```
private static void processMessagePostedEvent(SlackMessagePosted event,
                                              SlackSession session) {
 handleQuizzRequest(event, session);
 handleAnswer(event, session);
}
```

Handling quiz requests

Quiz requests are triggered using *!quizz*. The event handler below will first check whether the message contains this keyword. If the keyword is found, then the bot has to check whether it is already running a quiz. If so, it asks the user who requested a quiz to wait (the bot can handle only one quiz at a time). If no quiz is currently run, then the bot prepares the quiz data (randomly choosing questions and preparing the scoreboard) to send the first question.

```
private static void handleQuizzRequest(SlackMessagePosted event,
  SlackSession session) {
 //looking for !quizz command
 if ("!quizz".equals(event.getMessageContent().trim())) {
  // check if a quiz is currently in progress on a channel
  if (quizzChannel != null) {
   session.sendMessage(event.getChannel(), "I'm sorry " + event.getSender().getRe
                       ", I'm currently running a quiz, please
                        wait a few minutes");
   return;
  }
  prepareQuizzData(event);
  sendNewQuestion(session);
 }
}
```

Handling quiz answers

Quiz answers are digits between 1 and 4; any other values are ignored. (If no quiz is running, the bot will ignore *all* messages.) Whatever the answer given, the respondent's user ID is stored in order to avoid having one user give two answers. Depending on whether the answer is right or wrong, the bot will either give a point to the user and trigger the next question (or, if that was the last question, display scoreboard) or indicate to the user that it was a wrong answer. Here's the code that handles all of this:

```
private static void handleAnswer(SlackMessagePosted event,
  SlackSession session) {
 //no quiz launched
 if (quizzChannel == null) {
  return;
```

```
    }
    //an answer should be given on the quiz channel only
    if (!event.getChannel().getId().equals(quizzChannel.getId())) {
     return;
    }
    //an answer is a single digit from 1 to 4
    String answerValue = event.getMessageContent().trim();
    if (!ACCEPTED_ANSWERS.contains(answerValue)) {
     //ignore answer
     return;
    }
    int currentCounter = questionCounter;
    synchronized (lock) {
     if (questionCounter != currentCounter) {
      // the question has timed out and the next one was sent by the bot
      return;
     }
     //A user can answer only once per question
     if (answeredUser.contains(event.getSender().getId())) {
      session.sendMessage(quizzChannel, "I'm sorry " +
        event.getSender().getRealName() +
                              ", you can only give one answer per question");
      return;
     }
     //This user has now given an answer
     answeredUser.add(event.getSender().getId());
     //Check value
     if (Integer.parseInt(answerValue) == expectedAnswer) {
      //good answer
      goodAnswer(event, session);
     } else {
      wrongAnswer(event, session);
     }
    }
   }
```

Sending a new question

When the quiz starts or when a player gives a correct answer, the bot sends a new question. To achieve that, the bot picks from its randomized question list and prepares a Slack attachment to nicely format the question. Then it sends the question on Slack and starts a timer to wait up to 10 seconds; if no correct answer is given by the time the 10 seconds run out, the bot sends a new question. Here's the code that does all of this:

```
   private static void sendNewQuestion(SlackSession session) {
    //resetting the users who have answered to this question
    answeredUser = new HashSet<>();
    Question currentQuestion = shuffledQuestions.get(questionCounter);
```

```
 expectedAnswer = currentQuestion.expectedAnswer;
 SlackAttachment attachment = new SlackAttachment(currentQuestion.
   question,
                                                    currentQuestion.
                                                      question, "", "");
 attachment.addField("1", currentQuestion.answer1, true);
 attachment.addField("2", currentQuestion.answer2, true);
 attachment.addField("3", currentQuestion.answer3, true);
 attachment.addField("4", currentQuestion.answer4, true);
 session.sendMessage(quizzChannel, "", attachment);
 timer = buildTimer(session);
 timer.start();
}
```

Handling a correct answer

When a player chooses the right answer, the timer is interrupted (since the bot was waiting for a correct answer to be given) and a point is given to the player. Next, the bot tells the user he gave the right answer and triggers the next step: either posting either a new question or (if that was the last question) the final results of the quiz. Here's the code that handles that:

```
private static void goodAnswer(SlackMessagePosted event, SlackSession
  session) {
 timer.interrupt();
 increaseScore(event);
 session.sendMessage(quizzChannel, "Good answer " +
   event.getSender().getRealName());
 nextQuizzStep(session);
}
```

Handling a wrong answer

When someone makes an incorrect guess, that player is registered in the scoreboard, since he has participated; even if the answer is wrong his score will be displayed at the end of the quiz. Then a message is sent to that user to tell him that his answer was wrong. Here's the code that accomplishes these two steps:

```
private static void wrongAnswer(SlackMessagePosted event, SlackSession
  session) {
 registerPlayerInScoreBoard(event);
  session.sendMessage(quizzChannel, "I'm sorry, you're wrong " +
                                    event.getSender().getRealName());
}
```

Displaying the quiz result

At the end of the quiz, the result is displayed using an attachment. The score is stored in a map associating the player's user id to his score, so the bot adds a field for each participating player. The bot then sends the attachment with an empty message, and finally resets questions and the quizzChannel variable so a new quiz can be requested:

```
private static void showResults(SlackSession session) {
 SlackAttachment attachment = new SlackAttachment("Final score",
                                                  "Final score", "", "");
 for (Map.Entry<String, Integer> entry : score.entrySet()) {
  attachment.addField(session.findUserById(entry.getKey()).getRealName(),
                      entry.getValue().toString(), true);
 }
 session.sendMessage(quizzChannel, "", attachment);
 shuffledQuestions = null;
 quizzChannel = null;
}
```

In all of the Java Slack bots presented in this chapter, you've seen how to send messages, read files, create files, and pin reaction emojis. This is a good start to give you some ideas to implement your own hacks.

Over the course of the more than 50 hacks in this book, we have detailed the many ways that Slack can benefit you and your company or community. Through integrations, extensions, and bots, we have pointed you in the right direction so you can take advantage of your own Slack implementation, and we have provided you with tools and ideas to create your own custom Slack solutions.

Index

About the Authors

Guillaume Binet is the creator of Errbot (http://errbot.io), and is currently working at Google on the Cloud signals team. In his spare time, you can find him car racing, scuba diving or tinkering on his latest electronic project.

Mike Street has been a professional front-end developer since 2010, but was building websites long before that. He spends his time making the web a better place, one website at a time. He specializes in SCSS, Gulp, JavaScript and ranting on the Internet (http://www.mikestreety.co.uk).

Benoit Corne is a software engineer and is the creator of simple-slack-api (https://github.com/Ullink/simple-slack-api), a framework designed to quickly build Slack bots in Java. He enjoys roleplaying games, math challenges, and spending time with his wife and three sons.

John Adams is a polyglot developer from St. Petersburg, FL. He currently serves as Chief Architect for the e-commerce startup Bringhub, happily leading his team of developers in writing Golang, Angular, and React applications. When he is not working, John enjoys spending time with his wonderful wife and three boys.

Aviv Laufer is a passionate developer, designer, and manager of software with over 20 years of extensive experience, managing teams from 20 to over 200 people. He enjoys spending time with his family and ultra-running.

Colophon

The animal on the cover of *FILL IN TITLE* is *FILL IN DESCRIPTION*.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *FILL IN CREDITS*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.