By: Robin Lord

September 20th, 2017

So You Want to Build a Chat Bot – Here's How (Complete with Code!)

Tools | Technical SEO

26 28

The author's views are entirely his or her own (excluding the unlikely event of hypnosis) and may not always reflect the views of Moz.

You're busy and (depending on effective keyword targeting) you've come here looking for something to shave months off the process of learning to produce your own chat bot. If you're convinced you need this and just want the how-to, skip to "What my bot does." If you want the background on *why* you should be building for platforms like Google Home, Alexa, and Facebook Messenger, read on.

Why should I read this?

Do you remember when it wasn't necessary to have a website? When most boards would scoff at the value of running a Facebook page? Now Gartner is telling us that customers will manage 85% of their relationship with brands without interacting with a human by 2020 and publications like Forbes are saying that chat bots are the cause.

The situation now is the same as every time a new platform develops: if you don't have something your customers can access, you're giving that medium to your competition. At the moment, an automated presence on Google Home or Slack may not be central to your strategy, but those who claim ground now could dominate it in the future.

The problem is *time*. Sure, it'd be ideal to be everywhere all the time, to have your brand active on every platform. But it would also be ideal to catch at least four hours sleep a night or stop covering our keyboards with three-day-old chili con carne as we eat a hasty lunch in between building two of the Next Big Things. This is where you're fortunate in two ways;

1. When we develop chat applications, we don't have to worry about things like a beautiful user interface because it's all speech or text. That's not to say you don't need to worry about user experience, as there are rules (and an art) to designing a good conversational back-and-forth. Amazon is actually offering some hefty prizes for outstanding examples.
2. I've spent the last six months working through the steps from complete ignorance to creating a distributable chat bot and **I'm giving you all my workings.** In this post I break down each of the levels of complexity, from no-code back-and-forth to managing user credentials and sessions the stretch over days or months. I'm also including full code that you can adapt and pull apart as needed. I've commented each portion of the code explaining what it does and linking to resources where necessary.

I've written more about the value of Interactive Personal Assistants on the Distilled blog, so this post won't spend any longer focusing on why you should develop chat bots. Instead, I'll share everything I've learned.

What my built-from-scratch bot does

Ever since I started investigating chat bots, I was particularly interested in finding out the answer to one question: **What does it take for someone with little-to-no programming experience to create one of these chat applications from scratch?** Fortunately, I have direct access to someone with little-to-no experience (before February, I had no idea what Python was). And so I set about designing my own bot with the following hard conditions:

1. **It had to have some kind of real-world application.** It didn't have to be critical to a business, but it did have to bear basic user needs in mind.

2. **It had to be easily distributable** across the immediate intended users, and to have reasonable scope to distribute further (modifications at most, rather than a complete rewrite).
3. **It had to be flexible enough** that you, the reader, can take some free code and make your *own* chat bot.
4. **It had to be possible to adapt the skeleton of the process** for much more complex business cases.
5. **It had to be free to run,** but could have the option of paying to scale up or make life easier.
6. **It had to send messages** confirming when important steps had been completed.

The resulting program is "Vietnambot," a program that communicates with Slack, the API.AI linguistic processing platform, and Google Sheets, using real-time and asynchronous processing and its own database for storing user credentials.

If that meant nothing to you, don't worry — I'll define those things in a bit, and the code I'm providing is obsessively commented with explanation. The thing to remember is it does all of this to write down food orders for our favorite Vietnamese restaurant in a shared Google Sheet, probably saving tens of seconds of Distilled company time every year.

It's deliberately mundane, but it's designed to be a template for far more complex interactions. The idea is that whether you want to write a no-code-needed back-and-forth just through API.AI; a simple Python program that receives information, does a thing, and sends a response; or something that breaks out of the limitations of linguistic processing platforms to perform complex interactions in user sessions that can last days, this post should give you some of the puzzle pieces and point you to others.

What is API.AI and what's it used for?

API.AI is a linguistic processing interface. It can receive text, or speech converted to text, and perform much of the comprehension for you. You can see my Distilled post for more details, but essentially, it takes the phrase “My name is Robin and I want noodles today” and splits it up into components like:

- **Intent:** food_request
- **Action:** process_food
- **Name:** Robin

- **Food:** noodles
- **Time:** today

This setup means you have some hope of responding to the hundreds of thousands of ways your users could find to say the same thing. It's your choice whether API.AI receives a message and responds to the user right away, or whether it receives a message from a user, categorizes it and sends it to your application, then waits for your application to respond before sending your application's response back to the user who made the original request. In its simplest form, the platform has a bunch of one-click integrations and requires *absolutely no code*.

I've listed the possible levels of complexity below, but it's worth bearing some hard limitations in mind which apply to most of these services. They cannot remember anything outside of a user session, which will automatically end after about 30 minutes, they have to do everything through what are called POST and GET requests (something you can ignore unless you're using code), and if you *do* choose to have it ask your application for information before it responds to the user, you have to do everything and respond within five seconds.

What are the other things?

Slack: A text-based messaging platform designed for work (or for distracting people from work).

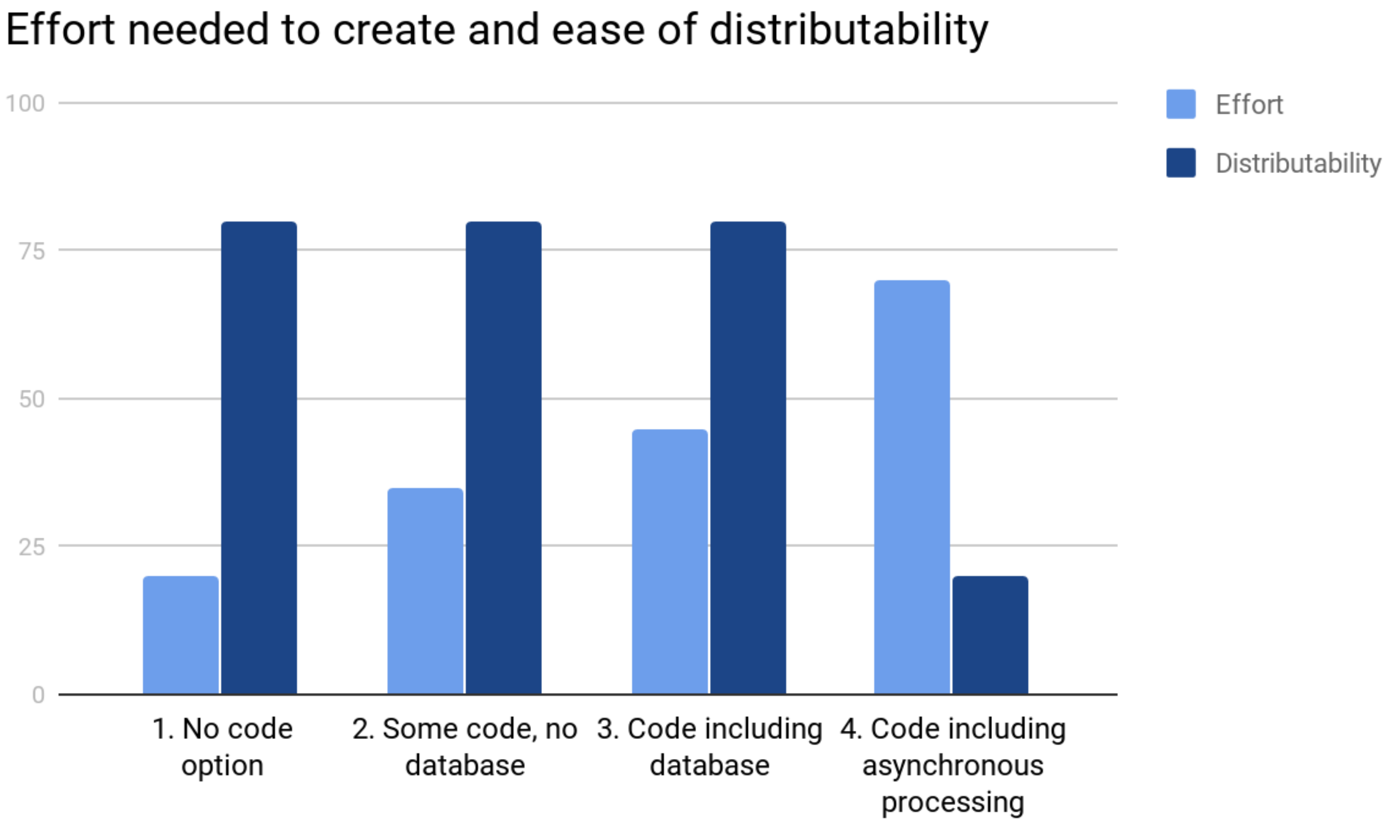
Google Sheets: We all know this, but just in case, it's Excel online.

Asynchronous processing: Most of the time, one program can do one thing at a time. Even if it asks another program to do something, it normally just stops and waits for the response. Asynchronous processing is how we ask a question and continue without waiting for the answer, possibly retrieving that answer at a later time.

Database: Again, it's likely you know this, but if not: it's Excel that our code will use (different from the Google Sheet).

Heroku: A platform for running code online. *(Important to note: I don't work for Heroku and haven't been paid by them. I couldn't say that it's the best platform, but it can be free and, as of now, it's the one I'm most familiar with).*

How easy is it?



This graph isn't terribly scientific and it's from the perspective of someone who's learning much of this for the first time, so here's an approximate breakdown:

Label	Functionality	Time it took me
1	You set up the conversation purely through API.AI or similar, no external code needed. For instance, answering set questions about contact details or opening times	Half an hour to distributable prototype
2	A program that receives information from API.AI and uses that information to update the correct cells in a Google Sheet (but can't remember user names and can't use the slower Google Sheets integrations)	A few weeks to distributable prototype

Label	Functionality	Time it took me
3	A program that remembers user names once they've been set and writes them to Google Sheets. Is limited to five seconds processing time by API.AI, so can't use the slower Google Sheets integrations and may not work reliably when the app has to boot up from sleep because that takes a few seconds of your allocation*	A few weeks on top of the last prototype
4	A program that remembers user details and manages the connection between API.AI and our chosen platform (in this case, Slack) so it can break out of the five-second processing window.	A few weeks more on top of the last prototype (not including the time needed to rewrite existing structures to work with this)

*On the Heroku free plan, when your app hasn't been used for 30 minutes it goes to sleep. This means that the first time it's activated it takes a little while to start your process, which can be a problem if you have a short window in which to act. You could get around this by (mis)using a free "uptime monitoring service" which sends a request every so often to keep your app awake. If you choose this method, in order to avoid using all of the Heroku free hours allocation by the end of the month, you'll need to register your card (no charge, it just gets you extra hours) and only run this application on the account. Alternatively, there are any number of companies happy to take your money to keep your app alive.

For the rest of this post, I'm going to break down each of those key steps and either give an overview of how you could achieve it, or point you in the direction of where you can find that. The code I'm giving you is Python, but as long as you can receive and respond to GET and POST requests, you can do it in pretty much whatever format you wish.

---

1. Design your conversation

Conversational flow is an art form in itself. Jonathan Seal, strategy director at Mando and member of British Interactive Media Association's AI thinktank, has given some great talks on the topic. Paul Pangaro has also spoken about conversation as more than interface in multiple mediums.

Your first step is to create a flow chart of the conversation. Write out your ideal conversation, then write out the most likely ways a person might go off track and how you'd deal with them. Then go online, find existing chat bots and do everything you can to break them. Write out the most difficult, obtuse, and nonsensical responses you can. Interact with them like you're six glasses of wine in and trying to order a lemon engraving kit, interact with them as though you've found charges on your card for a lemon engraver you definitely didn't buy and you are *livid*, interact with them like you're a bored teenager. At every point, write down what you tried to do to break them and what the response was, then apply that to your flow. Then get someone *else* to try to break your flow. Give them no information whatsoever apart from the responses you've written down (not even what the bot is designed for), refuse to answer any input you don't have written down, and see how it goes. David Low, principal evangelist for Amazon Alexa, often describes the value of printing out a script and testing the back-and-forth for a conversation. As well as helping to avoid gaps, it'll also show you where you're dumping a *huge* amount of information on the user.

While "best practices" are still developing for chat bots, a common theme is that it's not a good idea to pretend your bot is a person. Be upfront that it's a bot — users will find out anyway. Likewise, it's incredibly frustrating to open a chat and have no idea what to say. On text platforms, start with a welcome message making it clear you're a bot and giving examples of things you can do. On platforms like Google Home and Amazon Alexa users will expect a program, but the "things I can do" bit is still important enough that your bot **won't be approved without this opening phase.**

I've included a sample conversational flow for Vietnambot at the end of this post as one way to approach it, although if you have ideas for alternative conversational structures I'd be interested in reading them in the comments.

A final piece of advice on conversations: The trick here is to find organic ways of controlling the possible inputs and preparing for unexpected inputs. That being said, the Alexa evangelist team provide an example of terrible user experience in which a bank's app said: "If you want to continue, say *nine*." Quite often *questions*, rather than *instructions*, are the key.

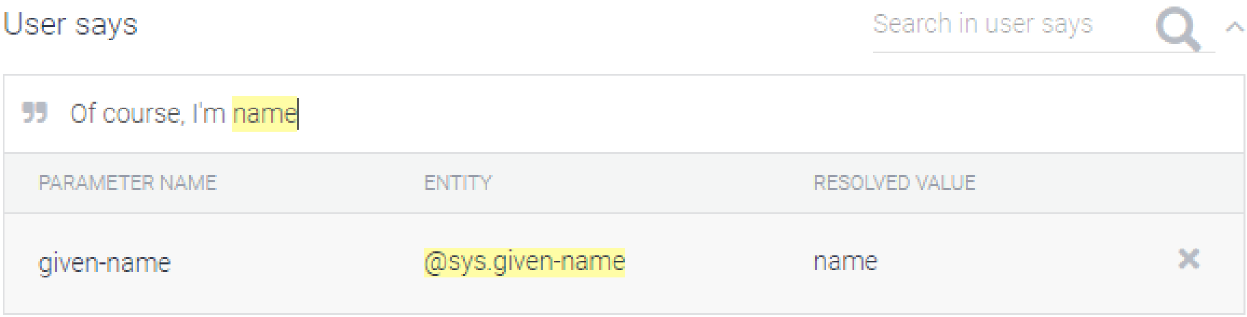
2. Create a conversation in API.AI

API.AI has quite a lot of documentation explaining how to create programs here, so I won't go over individual steps.

Key things to understand:

You create **agents**; each is basically a different program. **Agents** recognize **intents**, which are simply ways of triggering a specific **response**. If someone says the **right things** at the **right time,** they meet criteria you have set, fall into an **intent**, and get a pre-set **response**.

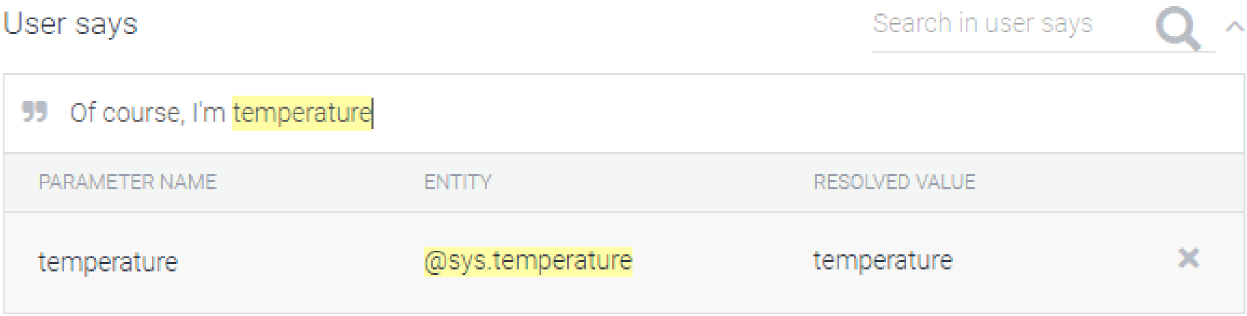
The **right things** to say are included in the "**User says**" section (screenshot below). You set either exact phrases or lists of options as the necessary input. For instance, a user could write "Of course, I'm [any name]" *or* "Of course, I'm [any temperature]." You could set up one intent for name-is which matches "Of course, I'm [given-name]" and another intent for temperature which matches "Of course, I'm [temperature]," and depending on whether your user writes a name or temperature in that final block you could activate either the "name-is" or "temperature-is" **intent**.



User says

Of course, I'm name

PARAMETER NAME	ENTITY	RESOLVED VALUE
given-name	@sys.given-name	name

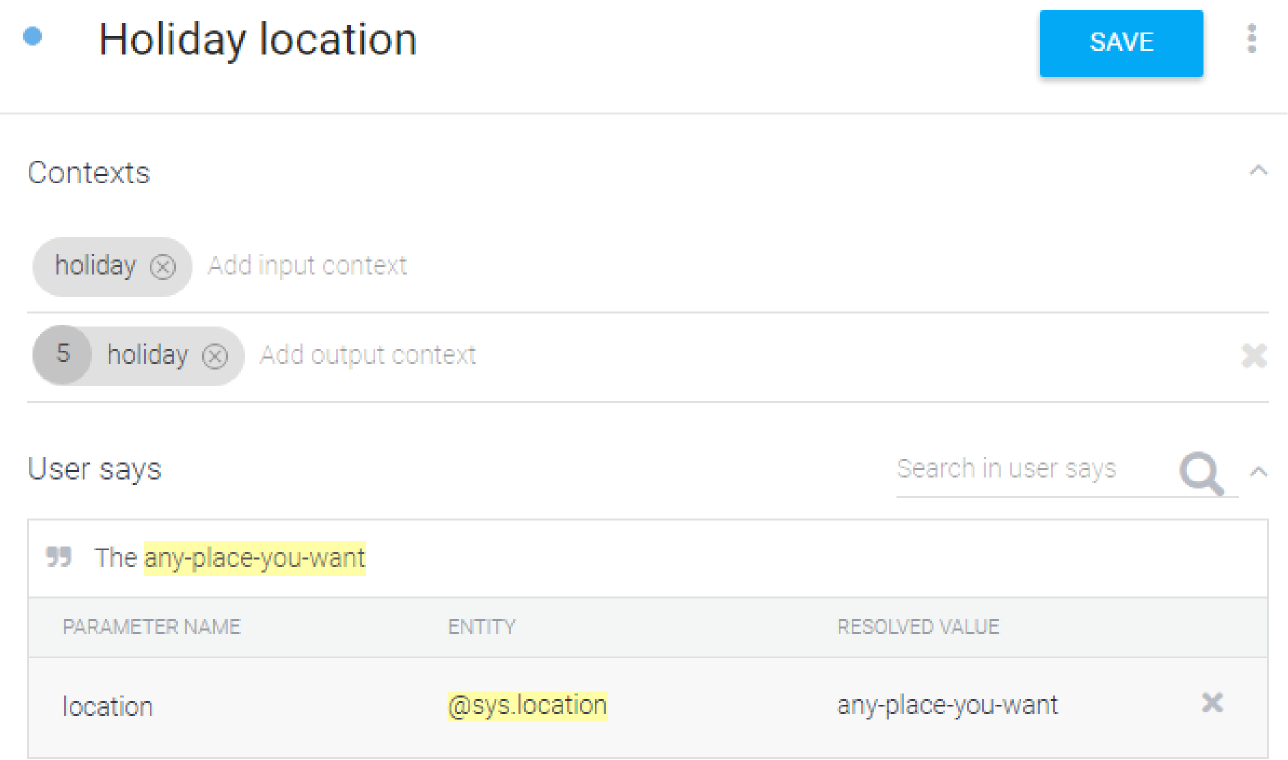


User says

Of course, I'm temperature

PARAMETER NAME	ENTITY	RESOLVED VALUE
temperature	@sys.temperature	temperature

The "right time" is defined by **contexts. Contexts** help define whether an **intent** will be activated, but are *also created* by certain **intents**. I've included a screenshot below of an example interaction. In this example, the user says that they would like to go to on holiday. This activates a **holiday intent** and sets the **holiday context** you can see in input contexts below. After that, our service will have automatically responded with the question "where would you like to go?" When our user says "The" and then any location, it activates our **holiday location intent** because it matches both the **context**, and what the **user says.** If, on the other hand, the user had initially said "I want to go to the theater," that might have activated the **theater intent** which would set a **theater context** — so when we ask "what area of theaters are you interested in?" and the user says "The [location]" or even just "[location]," we will take them down a completely different path of suggesting theaters rather than hotels in Rome.



holiday

5 holiday

The any-place-you-want

PARAMETER NAME	ENTITY	RESOLVED VALUE
location	@sys.location	any-place-you-want

The way you can create conversations without ever using external code is by using these **contexts**. A user might say "What times are you open?"; you could set an **open-time-inquiry context**. In your response, you could give the times and ask if they want the phone number to contact you. You would then make a **yes/no intent** which matches the context you have set, so if your user says "Yes" you respond with the number. This could be set up within an hour but gets exponentially more complex when you need to respond to specific parts of the message. For instance, if you have different shop locations and want to give the right phone number without having to write out every possible location they could say in API.AI, you'll need to integrate with external code (see section three).

Now, there will be times when your users don't say what you're expecting. Excluding contexts, there are three very important ways to deal with that:

1. Almost like keyword research — plan out as many possible variations of saying the same thing as possible, and put them all into the intent
2. Test, test, test, test, test, test, test, test, test, test, test, test, test, test, test (when launched, every chat bot will have problems. Keep testing, keep updating, keep improving.)
3. Fallback contexts

Fallback contexts don't have a **user says** section, but can be boxed in by **contexts**. They match anything that has the right context but *doesn't* match any of your **user says**. It could be tempting to use fallback intents as a catch-all. Reasoning along the lines of "This is the only thing they'll say, so we'll just treat it the same" is understandable, but it opens up a massive hole in the process. Fallback intents are designed to be a conversational safety net. They operate exactly the same as in a normal conversation. If a person asked what you want in your tea and you responded "I don't want tea" and that person made a cup of tea, wrote the words "I don't want tea" on a piece of paper, and put it in, that is not a person you'd want to interact with again. If we are using fallback intents to do anything, we need to preface it with a check. If we had to resort to it in the example above, saying "I think you asked me to add *I don't want tea* to your tea. Is that right?" is clunky and robotic, but it's a big step forward, and you can travel the rest of the way by perfecting other parts of your conversation.

3. Integrating with external code

I used Heroku to build my app . Using this excellent weather webhook example you can actually deploy a bot to Heroku within minutes. I found this example particularly useful as something I could pick apart to make my own call and response program. The weather webhook takes the information and calls a yahoo app, but ignoring that specific functionality you essentially need the following if you're working in Python:

```
#start
    req = request.get_json
    print("Request:")
    print(json.dumps(req, indent=4))
#process to do your thing and decide what response should be

    res = processRequest(req)
# Response we should receive from processRequest (you’ll need to write some code ca
{
        "speech": “speech we want to send back”,
        "displayText": “display text we want to send back, usually matches speech”,
        "source": "your app name"
    }

# Making our response readable by API.AI and sending it back to the servic

 response = make_response(res)
    response.headers['Content-Type'] = 'application/json'
    return response
# End
```

As long as you can receive and respond to requests like that (or in the equivalent for languages other than Python), your app and API.AI should both understand each other perfectly — what you do in the interim to change the world or make your response is entirely up to you. The main code I have included is a little different from this because it's also designed to be the step in-between Slack and API.AI. However, I have heavily commented sections like like *process_food* and the database interaction processes, with both explanation and reading sources. Those comments should help you make it your own. If you want to repurpose my program to work within that five-second window, I would forget about the file called *app.py* and aim to copy whole processes from *tasks.py*, paste them into a program based on the weatherhook example above, and go from there.

Initially I'd recommend trying GSpread to make some changes to a test spreadsheet. That way you’ll get visible feedback on how well your application is running (you’ll need to go through the authorization steps as they are explained here).

4. Using a database

Databases are pretty easy to set up in Heroku. I chose the Postgres add-on (you just need to authenticate your account with a card; it won't charge you anything and then you just click to install). In the **import** section of my code I've included links to useful resources which helped me figure out how to get the database up and running — for example, this blog post.

I used the Python library Psycopg2 to interact with the database. To steal some examples of using it in code, have a look at the section entitled "synchronous functions" in either the *app.py or tasks.py files*. *Open_db_connection* and *close_db_connection* do exactly what they say on the tin (open and close the connection with the database). You tell *check_database* to check a specific column for a specific user and it gives you the value, while *update_columns* adds a value to specified columns for a certain user record. Where things haven't worked straightaway, I've included links to the pages where I found my solution. One thing to bear in mind is that I've used a way of including columns as a variable, which Psycopg2 recommends quite strongly against. I've gotten away with it so far because I'm always writing out the specific column names elsewhere — I'm just using that method as a short cut.

5. Processing outside of API.AI's five-second window

It needs to be said that this step complicates things by no small amount. It *also* makes it harder to integrate with different applications. Rather than flicking a switch to roll out through API.AI, you have to write the code that interprets authentication and user-specific messages for each platform you're integrating with. What's more, spoken-only platforms like Google Home and Amazon Alexa don't allow for this kind of circumvention of the rules — you *have* to sit within that 5–8 second window, so this method removes those options. The only reasons you should need to take the integration away from API.AI are:

- **You want to use it to work with a platform that it doesn't have an integration with.** It currently has 14 integrations including Facebook Messenger, Twitter, Slack, and Google Home. It also allows exporting your conversations in an Amazon Alexa-understandable format (Amazon has their own similar interface and a bunch of instructions on how to build a skill — here is an example.
- **You are processing *masses* of information.** I'm talking really large amounts. Some flight comparison sites have had problems fitting within the timeout limit of these platforms, but if you aren't trying to process every detail for every flight for the next 12 months and it's taking more than five seconds, it's probably going to be easier to make your code more efficient than work outside the window. Even if you are, those same flight comparison sites solved the problem by creating a process that regularly checks their full data set and creates a smaller pool of information that's more quickly accessible.

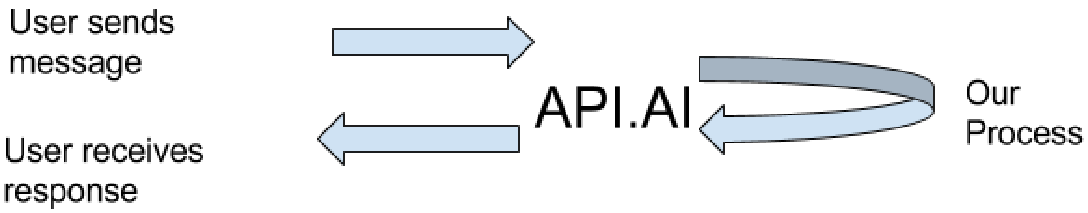
- **You need to send multiple follow-up messages to your user.** When using the API.AI integration it's pretty much call-and-response; you don't always get access to things like authorization tokens, which are what some messaging platforms require before you can automatically send messages to one of their users.
- **You're working with another program that can be quite slow, or there are technical limitations to your setup.** This one applies to Vietnambot, I used the GSpread library in my application, which is fantastic but can be slow to pull out bigger chunks of data. What's more, Heroku can take a little while to start up if you're not paying.

I could have paid or cut out some of the functionality to avoid needing to manage this part of the process, but that would have failed to meet number 4 in our original conditions: *It had to be possible to adapt the skeleton of the process for much more complex business cases*. If you decide you'd rather use my program within that five-second window, skip back to section 2 of this post. Otherwise, keep reading.

When we break out of the five-second API.AI window, we have to do a couple of things. First thing is to flip the process on its head.

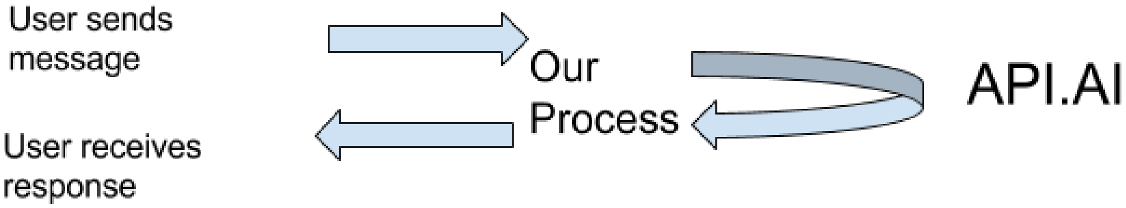
What we were doing before:

User sends message -> API.AI -> our process -> API.AI -> user



What we need to do now:

User sends message -> our process -> API.AI -> our process -> user



Instead of API.AI waiting while we do our processing, we do some processing, wait for API.AI to categorize the message from us, do a bit more processing, then message the user.

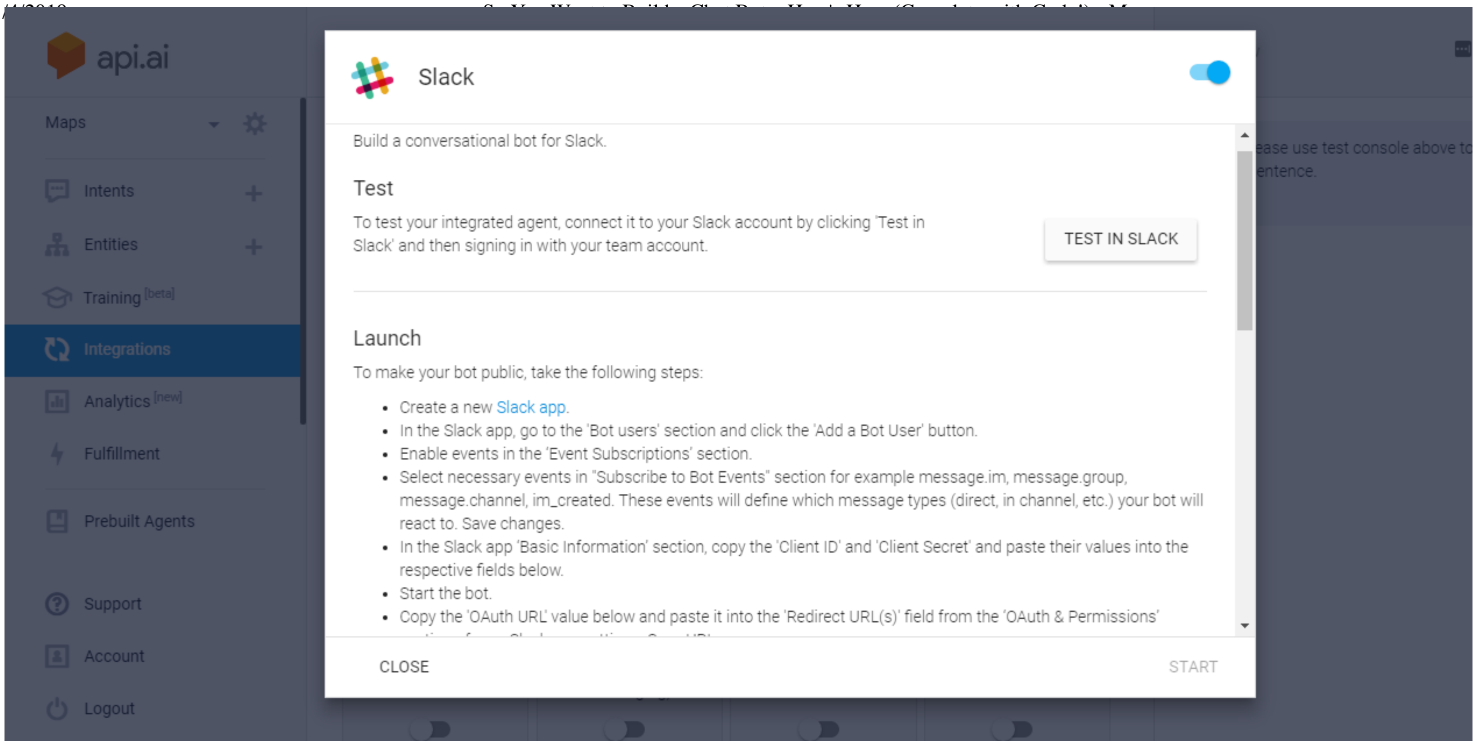
The way this applies to Vietnambot is:

1. User says “I want [food]”
2. Slack sends a message to my app on Heroku
3. My app sends a “swift and confident” 200 response to Slack to prevent it from resending the message. To send the response, my process has to shut down, so before it does that, it activates a secondary process using "tasks."
4. The secondary process takes the query text and sends it to API.AI, then gets back the response.
5. The secondary process checks our database for a user name. If we don’t have one saved, it sends *another* request to API.AI, putting it in the “we don’t have a name” context, and sends a message to our user asking for their name. That way, when our user responds with their name, API.AI is already primed to interpret it correctly because we’ve set the right context (see section 1 of this post). API.AI tells us that the latest message is a user name and we save it. When we have both the user name and food (whether we’ve just got it *from* the database or just saved it *to* the database), Vietnambot adds the order to our sheet, calculates whether we’ve reached the order minimum for that day, and sends a final success message.

6. Integrating with Slack

This won’t be the same as integrating with other messaging services, but it could give some insight into what might be required elsewhere. Slack has two authorization processes; we’ll call one "challenge" and the other "authentication."

Slack includes instructions for an app lifecycle here, but API.AI actually has excellent instructions for how to set up your app; as a first step, create a simple back-and-forth conversation in API.AI (not your full product), go to integrations, switch on Slack, and run through the steps to set it up. Once that is up and working, you’ll need to change the OAuth URL and the Events URL to be the URL for your app.



Build a conversational bot for Slack.

Test

To test your integrated agent, connect it to your Slack account by clicking 'Test in Slack' and then signing in with your team account.

Launch

To make your bot public, take the following steps:

- Create a new Slack app.
- In the Slack app, go to the 'Bot users' section and click the 'Add a Bot User' button.
- Enable events in the 'Event Subscriptions' section.
- Select necessary events in "Subscribe to Bot Events" section for example message.im, message.group, message.channel, im_created. These events will define which message types (direct, in channel, etc.) your bot will react to. Save changes.
- In the Slack app 'Basic Information' section, copy the 'Client ID' and 'Client Secret' and paste their values into the respective fields below.
- Start the bot.
- Copy the 'OAuth URL' value below and paste it into the 'Redirect URL(s)' field from the 'OAuth & Permissions'

Thanks to github user karishay, my app code includes a process for responding to the challenge process (which will tell Slack you're set up to receive events) and for running through the authentication process, using our established database to save important user tokens. There's also the option to save them to a Google Sheet if you haven't got the database established yet. However, be wary of this as anything other than a first step — user tokens give an app a lot of power and have to be guarded carefully.

7. Asynchronous processing

We are running our app using Flask, which is basically a whole bunch of code we can call upon to deal with things like receiving requests for information over the internet. In order to create a secondary worker process I've used Redis and Celery. Redis is our "message broker"; it makes makes a list of everything we want our secondary process to do. Celery runs through that list and makes our worker process do those tasks in sequence. Redis is a note left on the fridge telling you to do your washing and take out the bins, while Celery is the housemate that bangs on your bedroom door, note in hand, and makes you do each thing. I'm sure our worker process doesn't like Celery very much, but it's really useful for us.

You can find instructions for adding Redis to your app in Heroku here and you can find advice on setting up Celery in Heroku here. Miguel Grinberg's *Using Celery with Flask* blog post is also an excellent resource, but using the exact setup he gives results in a clash with our database, so it's easier to stick with the Heroku version.

Up until this point, we've been calling functions in our main app — anything of the form *function_name(argument_1, argument_2, argument_3)*. Now, by putting "tasks." in front of our function, we're saying "don't do this now — hand it to the secondary process." That's because we've done a few things:

- We've created *tasks.py* which is the secondary process. Basically it's just one big, long function that our main code tells to run.
- In *tasks.py* we've included Celery in our imports and set our app as *celery.Celery()*, meaning that when we use "app" later we're essentially saying "this is part of our Celery jobs list" or rather "*tasks.py* will only do anything when its flatmate Celery comes banging on the door"
- For every time our main process asks for an asynchronous function by writing *tasks.any_function_name(),* we have created that function in our secondary program just as we would if it were in the same file. However in our secondary program we've prefaced with "@app.task", another way of saying "Do *wash_the_dishes* when Celery comes banging the door yelling *wash_the_dishes(dishes, water, heat, resentment)*".
- In our "procfile" (included as a file in my code) we have listed our worker process as *--app=tasks.app*

All this adds up to the following process:

1. Main program runs until it hits an asynchronous function
2. Main program fires off a message to Redis which has a list of work to be done. The main process doesn't wait, it just runs through everything after it and in our case even shuts down
3. The Celery part of our worker program goes to Redis and checks for the latest update, it checks what function has been called (because our worker functions are named the same as when our main process called them), it gives our worker all the information to start doing that thing and tells it to get going
4. Our worker process starts the action it has been told to do, then shuts down.

As with the other topics mentioned here, I've included all of this in the code I've supplied, along with many of the sources used to gather the information — so feel free to use the processes I have. Also feel free to improve on them; as I said, the value of this investigation was that *I am not a coder*. Any suggestions for tweaks or improvements to the code are very much welcome.

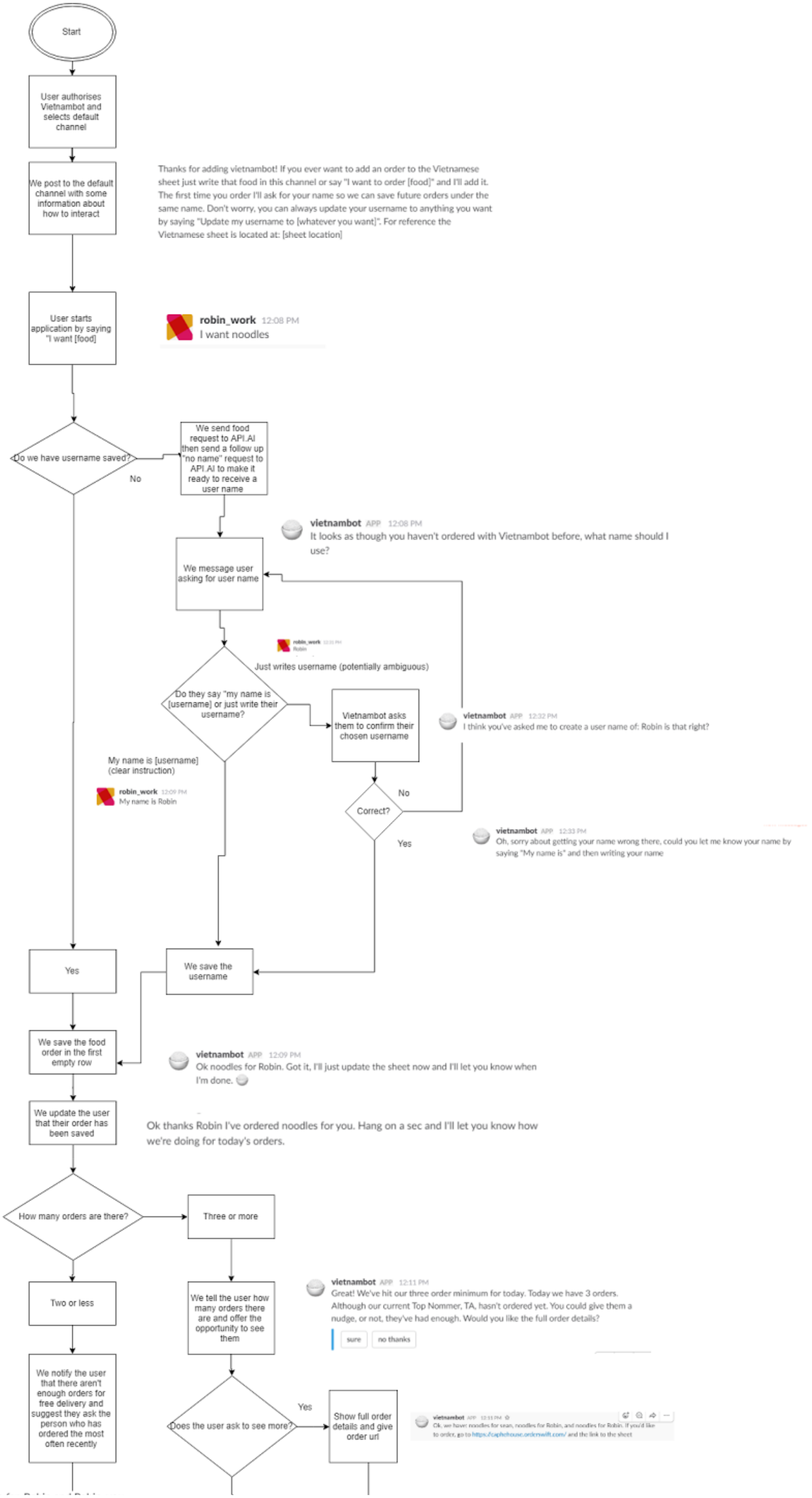
Conclusion

As I mentioned in the introduction to this post, there's huge opportunity for individuals and organizations to gain ground by creating conversational interactions for the general public. For the vast majority of cases you could be up and running in a few hours to a few days, depending on how complex you want your interactions to be and how comfortable you are with coding languages. There are some stumbling blocks out there, but hopefully this post and my obsessively annotated code can act as templates and signposts to help get you on your way.

Grab my code at GitHub

Bonus #1: The conversational flow for my chat bot

This is by no means necessarily the best or only way to approach this interaction. This is designed to be as streamlined an interaction as possible, but we're also working within the restrictions of the platform and the time investment necessary to produce this. Common wisdom is to create the flow of your conversation and then keep testing to perfect, so consider this example layout a step in that process. I'd also recommend putting one of these flow charts together *before* starting — otherwise you could find yourself having to redo a bunch of work to accommodate a better back-and-forth.


Start
User authorises Vietnambot and selects default channel
We post to the default channel with some information about how to interact
User starts application by saying "I want [food]
Do we have username saved?
No
We send food request to API.AI then send a follow up "no name" request to API.AI to make it ready to receive a user name
We message user asking for user name
Just writes username (potentially ambiguous)
Do they say "my name is [username] or just write their username?
Vietnambot asks them to confirm their chosen username
My name is [username] (clear instruction)
Correct?
No
Yes
Yes
We save the username
We save the food order in the first empty row
We update the user that their order has been saved
How many orders are there?
Three or more
Two or less
We tell the user how many orders there are and offer the opportunity to see them
We notify the user that there aren't enough orders for free delivery and suggest they ask the person who has ordered the most often recently
Does the user ask to see more?
Yes
Show full order details and give order url
sure
no thanks

Thanks for adding vietnambot! If you ever want to add an order to the Vietnamese sheet just write that food in this channel or say "I want to order [food]" and I'll add it. The first time you order I'll ask for your name so we can save future orders under the same name. Don't worry, you can always update your username to anything you want by saying "Update my username to [whatever you want]". For reference the Vietnamese sheet is located at: [sheet location]

robin_work 12:08 PM
I want noodles

vietnambot APP 12:08 PM
It looks as though you haven't ordered with Vietnambot before, what name should I use?

robin_work
Robin

vietnambot APP 12:32 PM
I think you've asked me to create a user name of: Robin is that right?

robin_work 12:09 PM
My name is Robin

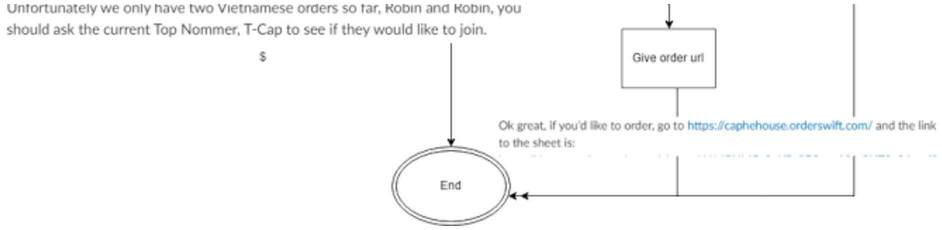
vietnambot APP 12:33 PM
Oh, sorry about getting your name wrong there, could you let me know your name by saying "My name is" and then writing your name

vietnambot APP 12:09 PM
Ok noodles for Robin. Got it, I'll just update the sheet now and I'll let you know when I'm done.

Ok thanks Robin I've ordered noodles for you. Hang on a sec and I'll let you know how we're doing for today's orders.

vietnambot APP 12:11 PM
Great! We've hit our three order minimum for today. Today we have 3 orders. Although our current Top Nommer, TA, hasn't ordered yet. You could give them a nudge, or not, they've had enough. Would you like the full order details?

vietnambot APP 12:11 PM
Ok, we have: noodles for sean, noodles for Robin, and noodles for Robin. If you'd like to order, go to https://caphehouse.orderswift.com/ and the link to the sheet



Unfortunately we only have two Vietnamese orders so far, Robin and Robin, you should ask the current Top Nommer, T-Cap to see if they would like to join.

$

Ok great, if you'd like to order, go to https://caphehouse.orderswift.com/ and the link to the sheet is:

Bonus #2: General things I learned putting this together

As I mentioned above, this has been a project of going from complete ignorance of coding to slightly less ignorance. I am not a professional coder, but I found the following things I picked up to be hugely useful while I was starting out.

1. **Comment everything.** You'll probably see my code is bordering on excessive commenting (anything after a # is a comment). While normally I'm sure someone wouldn't want to include a bunch of Stack Overflow links in their code, I found notes about what things portions of code were trying to do, and where I got the reasoning from, *hugely* helpful as I tried to wrap my head around it all.

2. **Print everything.** In Python, everything within "print()" will be printed out in the app logs (see the commands tip for reading them in Heroku). While printing each action can mean you fill up a logging window terribly quickly (I started using the Heroku add-on *LogDNA* towards the end and it's a huge step up in terms of ease of reading and length of history), often the times my app was falling over was because one specific function wasn't getting what it needed, or because of another stupid typo. Having a semi-constant stream of actions and outputs logged meant I could find the fault much more quickly. My next step would probably be to introduce a way of easily switching on and off the less necessary print functions.

3. **The following commands**: Heroku's how-to documentation for creating an app and adding code is pretty great, but I found myself using these all the time so thought I'd share (all of the below are written in the command line; type *cmd* in on Windows or by running Terminal on a Mac):

    1. **CD """[file location]"""** - select the file your code is in

    2. **"git init"** - create a git file to add to

    3. **"git add ."** - add all of the code in your file into the file that git will put online

    4. **"git commit -m "[description of what you're doing]" "** - save the data in your git file

    5. **"heroku git:remote -a [the name of your app]"** - select your app as where to put the code

    6. **"git push heroku master"** - send your code to the app you selected

    7. **“heroku ps”** - find out whether your app is running or crashed
    8. **“heroku logs”** - apologize to your other half for going totally unresponsive for the last ten minutes and start the process of working through your printouts to see what has gone wrong

4. **POST requests will always wait for a response.** Seems really basic — initially I thought that by just sending a POST request and not telling my application to wait for a response I’d be able to basically hot-potato work around and not worry about having to finish what I was doing. That’s not how it works in general, and it’s more of a symbol of my naivete in programming than anything else.
5. **If something is really difficult, it’s very likely you’re doing it wrong.** While I made sure to do pretty much all of the actual work myself (to avoid simply farming it out to the very talented individuals at Distilled), I was lucky enough to get some really valuable advice. The piece of advice above was from Dominic Woodman, and I should have listened to it more. The times when I made least progress were when I was trying to use things the way they shouldn’t be used. Even when I broke through those walls, I later found that someone didn’t want me to use it that way because it would completely fail at a later point. Tactical retreat *is* an option. (At this point, I should mention he wasn’t the only one to give invaluable advice; Austin, Tom, and Duncan of the Distilled R&D team were a huge help.)

About R0bin_L0rd — Robin is an SEO Analyst at the Distilled London office. He's worked in places from Sussex to Bangalore, India and each time found himself gravitating towards SEO.In his spare time Robin learns as much as possible (Python is his latest fix) and travels whenever he can, often thanks to the seemingly boundless generosity of near total strangers.

Popular posts like this

Overcoming Blockers: How to Build Your Red Tape Toolkit - Whiteboard Friday

Ranking the 6 Most Accurate Keyword Difficulty Tools

How to Get More Keyword Metrics for Your Target Keywords

👍 26 💬 28

Comments 28

Please keep your comments TAGFEE by following the community etiquette.

☐ E-mail me when new comments are posted

Sort: Most Popular ▴▾

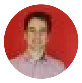
Comments are closed on posts more than 30 days old. Got a burning question? Head to our Q&A section to start a new conversation.

Julie Lichtenberg
a year ago

Have bookmarked this post for future reference! Thank you, Robin, for your descriptive and comprehensive "how-to" article - am not a developer and was able to understand (almost) all the directions!

Julie_Lichtenberg edited a year ago

👍 4 👎 0



Robin Lord
a year ago

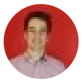
So glad to hear it's of use Julie, good luck!

👍 1 👎 0

Dr. Sadaf Ajmal Ph.D.
a year ago

Thanks Robin, really helpful. :)

👍 3 👎 0



Robin Lord
a year ago

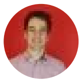
Glad to be of help :-)

👍 2 👎 0

Martin Beneš
a year ago

From the business perspective, chat bots can save tons of man-hours. I remember Emilie talking about this topic on the latest Brighton SEO which was really interesting.

👍 3 👎 0



Robin Lord
a year ago

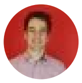
Couldn't agree more Martin, and I really enjoyed that talk too!

👍 1 👎 0

Brendon Ross
a year ago

Thanks Robin. I have seen some interesting cases of bots being used as proof of concepts. Really exciting stuff.

👍 3 👎 0



Robin Lord
a year ago

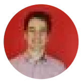
Yea, I think it can be a great way to test things out. Are there any that stand out to you specifically? Always keen to hear more examples.

👍 2 👎 0

DebHughes
a year ago

First I must say, "what a sense of humor you have" Looking forward to checking this out.

👍 3 👎 0



Robin Lord

a year ago

Haha, thanks Deb, hope it goes well! Do come back and comment again if there's anything you want to discuss

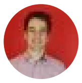
👍 1 👎 0

Cory Beevers

a year ago

How well does this work for CRO? What outcomes have you had form this? Have you got any examples of where this has been used? Also how does this affect rankings? Does Google class this as being user friendly or is this something Google sees negatively?

👍 3 👎 0



Robin Lord

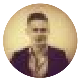
a year ago

Hi Cory, yea I actually just mentioned to Luis (should be below) an example shared by Emilie Reynaud at the most recent Brighton SEO in which chat bot users were 40% more likely to convert.

In terms of rankings - the content your bot shares is unlikely to be read by search engines (although there are rumours of Google testing forms by inputting an approximation of data it's not something that can be relied on in general, and certainly not here). The most direct lever I would expect to move in traditional search is decreasing bounce rates, or preventig users from ever searching your competitors in the first place. By giving people a different and potentially more intuitive way to access what they want on your site you stand a better chance of keeping them there and if they don't leave your site and return to search results that'll benefit you. That being said - I definitely wouldn't recommend replacing a considered navigation strategy with a chat bot pop-up, but it's a valuable component.

What is more interesting is the potential to capitalise on the new breed of search engines. These applications can do a lot of work without users leaving Facebook/ Twitter/ Slack etc. and platforms like Google Home are essentially search engines that provide only one result. So in the case of non-traditional SERP competition the benefit here is the same as building a website is for traditional - if you don’t have something there is no way a person can find you, if you don't offer purchases right there there's no way a person can buy from you unless you want them to go and Google, at which point your competitors have the opportunity to step in.

👍 3 👎 0



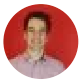
Cory Beevers

a year ago

I was actually at Brighton SEO but didn't see this even unfortunately. I would of liked to of gone.

Thank you for your response! This was very helpful.

👍 3 👎 0

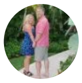


Robin Lord

a year ago

Ah that's unfortunate, it was a good batch, Jonathan Seal was speaking as well. In general if you're looking for chat bot focused talks; Mando has been putting some events together with case studies from organisations like 3. I'd also really recommend getting in touch with the Alexa evangelist team - they have a rolling series of talks and are very open to discussion.

👍 1 👎 0

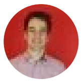


colemanconcierge

a year ago

We are manually working a lot of social media at this time and are considering the options between a VA and a DVA. I keep thinking that a DVA would be the correct path to ensure consistency and scalability. We have been experimenting with "like only" DVAs with some success. The hardest part in my mind is the semantics and conversation flow. I really appreciated your outline of conversation flow, complete with flow chart and references. That helps me advance my bots from mute liking to interacting information gathering machines.

👍 3 👎 0



Robin Lord

a year ago

So glad to hear this is of use! One thing I found interesting is it's definitely a different skill set putting together coherent conversations that achieve a goal, as opposed to designing a website that makes the most of pre-established convention. I'll look forward to seeing your programs in action.

👍 2 👎 0

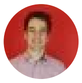
jhines

a year ago

Perfect timing my friend, I'm currently working on a simple chat similar to this with hook ups to an ecommerce listing of products.

Cheers!

👍 3 👎 0



Robin Lord

a year ago

Happy to help :-) good luck! I hope to see your program in action soon

👍 3 👎 0

DescarbonizadoraTresEco

a year ago

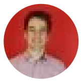
Hi Robin

If the tool is well developed, it saves time and money to a web with thousands and thousands of visits.

Just one question ... How does it affect the upload speed?

Thank you very much for the information

👍 2 👎 0



Robin Lord

a year ago

Hi Luis, you're right that these conversational applications can help get the most out of social followers (and website users if it is integrated with a chat application embedded within the site). Emilie Reynaud spoke at Brighton SEO last week about how her chat bot helped a job site get 40% more job applications for users which interacted with it.

Unfortunately in terms of speed, there are a huge number of variables to consider. My application is running on Heroku, external to and independent from any website, so it'll have no impact. In most cases these applications should respond semi-immediately and should use up negligible resource because API.AI is doing a lot of the heavy lifting in terms of recognition and categorisation, all we need to worry about is (potentially) passing the input and (usually) acting on the output. That being said, it really depends on what your program is doing. As I mentioned in the post, the GSpread plugin is great but a little slow so if you had a large number of users making requests at the same time, that could start to cause issues.

If you wanted to speed up this implementation I'm sure there are a bunch of ways in which the code could have been more elegant but first steps could include sourcing a faster and always live place to host the code, confirming that Slack isn't sending any duplicate messages and removing the deduplication loop, and cutting out the Google Sheets interactions to work purely in the database.

👍 5 👎 0

V **vittoriobanfi**
a year ago

About designing the interaction: I think it's very important not only to have a workflow scheme, but also to properly mockup the whole interaction of your chatbot. Otherwise you may end up in a situation similar to the following:

1. In the flowchart the chatbot makes a question/sends a message which seems ok

2. Once you develop it, you realize the message is either too long/not engaging/it feels weird

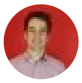
3. You now have to get back to square one and re-design it!

If instead you mockup the whole interaction, adding each message and previewing how it will look like, you will be able to correct your way before building a not optimal chatbot.

ps. Full disclosure: I work at Botsociety, which is the leading tool in the space!

FeliciaCrawford edited a year ago

👍 2 👎 0



Robin Lord
a year ago

Hi there, yep, absolutely! That's what I meant by testing your flow with someone else, and what Amazon's David Low is getting at when he suggests printing out a script and testing it that way. Great idea to have an interactional wireframe, as it were, I'll certainly check it out. Are the limitations of the free plan just how many prototypes and users you can run at a time, or is there any other functionality that a person would have to pay for?

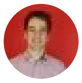
👍 1 👎 0

Angel Blanco
a year ago

If on February you didn't know what Python is and now you're able to use a stack of those kind-of-obscure dev apps and build a chatbot, you're kind of a hero imho.

Jokes aside, that guide was great not only on the practical side, also on the theoric. Thank you very much for that, can't wait to see how the chatbots evolve!

👍 2 👎 0



Robin Lord

a year ago

Thanks, that's really made my day :-) It was a steep learning curve but worth it to feel like I now have some practical experience. I'm really glad you've found the guide useful, I'm looking forward to them evolving too and the more we can spread the word about the relative accessibility of building them, the better!

👍 2 👎 0

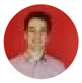
Sandy Maquilin

a year ago

Wonderful post Robin, very technical and well illustrated. A must needed technology for E Commerce website, need to have a chat bot to direct customers to the required page according to their likes and dislikes. The technology of future where human engagement could be minimized by bots, and a day will come where an E Commerce Companies could be run by bots with only a single central human brain.

Sandy-Maquilin edited a year ago

👍 2 👎 0



Robin Lord

a year ago

Thanks Sandy, really glad you enjoyed the post!

👍 2 👎 0

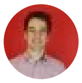
Ozair Akhtar

a year ago

Robin,

It's your first post and I am amazed that you shared the method and the code with all of us :)

👍 2 👎 1



Robin Lord

a year ago

Thanks! I've found the process really interesting and I'm glad to be able to help people get involved

👍 1 👎 0

Get fresh SEO data, insights, and tracking

Learn More About Moz Pro

Copyright 2018 © Moz, Inc. All rights reserved.

Back to Top

Local Marketing

Business Listings Audit

Citation Cleanup

Local Ranking Factors

Local For Enterprise

Marketing Resources

Moz Blog

Beginner's Guide To SEO

Community Q&A

Workshops & Training

Recommended SEO Companies

Get the Moz Top 10

The 10 most valuable pieces of content we can find for SEOs. Every 2 weeks.

Subscribe today