

Build a Bot

Workbook for LCCM Tech Event, 25 April 2017

Welcome to Build a Bot!

In the course of the next couple of hours you will

- Learn about building and running bots, using a simple framework called **botx**.
- Learn about some of the newer constructs and features of JavaScript and Python
- Have fun (we hope...!)

You should work through the tasks and challenges in order. In general, you will be asked to

- run some code and see it working
- look at the code and try to understand what it's doing
- modify the code to add a feature or capability, and run it again
- then read some commentary, talk with your partner and reflect on what you've done
- if you have any questions, ask one of the botmeisters

There is – deliberately – more material here than you'll be able to cover in the time available! Don't worry about this – just work steadily through the tasks and challenges, talk to each other, note any questions and ask for help if need be.

Setup

You can either set up and run the workshop on your own laptop, or use an online IDE to access and run the bot code.

Local installation

You should have been asked to prepare by installing

- nodejs - get the latest (version 7.n) from nodejs.org
- a code editor or IDE you're comfortable with (we recommend WebStorm, from JetBrains. A 30-day trial license will let you try it out: if you're at school or a student you can get a free academic license. www.jetbrains.com/webstorm).

Then download the workshop code from:

<https://github.com/davethehat/botx>

Select Clone or Download, download the source as a zip file, and expand it somewhere useful on your laptop. Then open a terminal or command window, change directory to the top directory of the expanded zip and run:

```
npm install
```

to install the libraries that the workshop code uses.

You will need a Slack API token — this will be provided for you at the workshop, just ask one of the botmeisters.

Cloud9

Alternatively there are accounts on the cloud IDE Cloud9 with environments already set up. Go to:

```
https://c9.io
```

Log in using the email address and password based on your number (see below, Task 1). You should find a single workspace, which has been initialised and is ready for you to edit and run code. Use the **bash** tab in the workspace to run your bot.

Task 1 — up and running

This is just about logging in, running the example bot, and making sure everything's working. We will have demonstrated this on the big screen: you'll need to repeat this.

First, log in to slack. You'll need to point your browser at:

```
https://lccmbot.slack.com
```

Use your bot number to sign in as follows. If (for example) you are number **3**, your login is:

```
lccm.bot+3@gmail.com
```

The password for number **3** is:

```
botdev3!
```

and when you're logged into slack, your name will be **botdev3**.

You will need to open a private chat with your bot, which (if your number is **3**) will (of course) be called **bot3**.

Next, run the simple bot provided. You should be in a command line or shell (either on your own laptop or in the Cloud9 environment as described above), and in the **botx** directory. Type:

```
npm run bot ./work/firstbot.js
```

If everything is setup properly, you should see some text messages which tell you that the bot is running and listening for messages.

Now go back to slack in your browser, and in your bot's private channel type:

bot help

All being well, your bot will respond with a message about what it understands.

Try typing:

hi

and then:

repeat one two three testing

and see what happens.

You can shut down your bot by typing:

shutdown

into slack. Alternatively, in your shell you can type CTRL-C (hold the `ctrl` key down and type C) which will also kill your bot.

Task 2 — a simple bot

Open the file `work/firstbot.js` in your editor and read the code. See if you can understand what it does: talk it over with your partner, note down any questions that you have, and ask one of the nearby botmeisters.

Next, add another simple response to firstbot:

*** * * CHALLENGE 1 * * ***

When your bot hears "how are you", make it reply "Very well, thank you. And you?"

Try this before turning the page and reading about botx and this first, simple bot. Don't forget you'll need to type:

```
npm run bot ./work/firstbot.js
```

to run your bot again. Also, you have to shut the bot down and restart it to introduce any changes you've made. (Remember, you can do this by typing `ctrl-c` in your shell.)

Introducing botx

Botx is designed to make writing simple bots very straightforward. If you're interested, it's an example of a **Domain Specific Language (DSL)**, a set of commands and modifiers that work together to describe a constrained world. JavaScript and Python are both quite frequently used to build these.

The very simplest bot can be created and run as follows:

```
'use strict'; // 1

const botx = require('../src/botx'); // 2

botx(); // 3
```

This creates a bot that responds to **bot help** and **shutdown** only.

Notes (nb anything after `///` on a line of code is a comment, and is ignored by JavaScript):

- (1) tells node.js to be strict about javascript
- (2) loads the botx library
- (3) starts up a bot

How does slack know which bot is running? Good question. You set up a bot in Slack itself: when you do this you get an **API TOKEN** for the bot which your code uses when it starts up and talks to Slack. (The details of this are beyond the scope of this evening's workshop, but if you're interested you can find out more at <https://api.slack.com/bot-users>). Your bot code can be running anywhere: the interaction between slack and your bot is all done via the internet.

The simple bot in **firstbot.js** shows a couple more capabilities. Open the file in your editor and read through the code below.

```
'use strict';

const botx = require('../src/botx');

const bot = botx({
  help: {
    messages: [
      'I am but a simple bot...',
      'Say "hello" or "hi" to me, or',
      'ask me to say or repeat something.'
    ]
  }
}); // 1

bot.log.notice("Starting a new BOTX..."); // 2
```

```
bot.when('^hi') // 3
  .or('^hello')
  .thenSay('Well, hello there...')
  .thenSay('How nice to see you again!',
    'It\'s been a while!',
    'What are you doing around these parts?') // 4
  .go();

bot.when('^say (.+)') // 5
  .or('^repeat (.+)')
  .thenSay("{}{}{}! I hope I said that right...")
  .go();
```

- (1) Instead of just calling **botx()**, we're passing the function some extra information, and storing the returned object - this is the bot which you'll be interacting with to set up listeners and responses. Here, the extra information is the reply that the bot will give when you type **bot help** into Slack.
- (2) **bot.log.notice** prints out some text in the shell/command prompt in which your bot is running (NB not in the slack channel itself!)
- (3) This is where things are getting interesting. This block of 5 lines is building a listener and response. When the bot hears 'hi' or 'hello' it will respond as indicated. Having set up the stimulus and responses, you need to tell the bot to use them - this is what the last **go()** does.
- (4) If you pass more than one response to **thenSay**, the bot will choose one at random.
- (5) This is an example of the bot listening for something and then being able to use what it heard in a response. To understand what's happening here, we need to talk about **regular expressions**...

Digression — regular expressions

You may know all about these, in which case skip this section. If not, read on.

Regular expressions (usually called 'regexes' by coders) give us a way of checking and matching text against patterns. They've been around for almost as long as computers, and there are several variants. In programming languages they've become pretty standard, and both JavaScript and Python support them.

In the examples above, the regular expression magic used is:

hello	Matches the exact sequence of characters hello . As it stands, these can appear anywhere in the user's input, so botx would respond to this if you typed "well, hello there!".
^	Matches the beginning of a line. In the code above, '^hello' would match the text "hello", "hello there", "hellooooooo", but not "well, hello!".
.	The full stop matches any character at all.

+	Modifies what precedes it, and says match one or more repetitions. So <code>'+'</code> matches the pattern of one or more <code>'.'</code> , which means it matches one or more of <i>anything</i> , up to the end of the input that it's given
()	The brackets let us capture a match. In the example here, it means that we've saved whatever the user typed to trigger the match after the first word 'say' or 'repeat' . We're using that in our reply by adding <code>'{{1}}'</code> , which signifies the contents of the first captured group . If you had two or more captures in your matching expression, then you could use <code>{{2}}</code> , <code>{{3}}</code> and so on as appropriate.

Here are some other expressions you might find useful:

<code>\w</code>	Match a single word character (defined as letters, numbers, underscore, excluding spaces and punctuation)
<code>(\w+)</code>	A capturing group of one or more word characters. Use this to grab a single word from the user's input.
<code>(\d+)</code>	The same, for just numbers
<code>[abyz]</code>	Square brackets enclose a set of possible matches. This matches any of the four characters a, b, y, z.
<code>[a-z]</code>	Matches a range of characters – in this case anything including and between a to z.

IMPORTANT if you use any of the 'special' matches like `\w`, `\d`, you should type what you want botx to listen to like this:

```
bot.when(/call me (\w+)/)
  .thenSay('OK {{1}}')
  .go();
```

Spot the difference? what you're listening for is enclosed in `/.../`, not `'...'`!

If you need some help with regular expressions, just ask a botmeister.

* * * CHALLENGE 2 * * *

When your bot hears "pirate <anything>" make it reply "<anything>. Arrrrrrr!"

* * * CHALLENGE 3 * * *

When your bot hears "pirate <anything>" make it reply at random with the matched pattern and one of three or four favourite pirate phrases.

Task 3 – more complex responses

We might want our bot to do more than print something out in reply to a user's input. For example, we could run some code to produce a response, or alternatively we might want to respond using something that a user has entered previously.

To do this we need to move beyond simply replying with a fixed string to having the bot respond by executing some code. This is done by **executing a callback**. Take a look at

`work/memorybot.js`

As before, run the bot

```
npm run bot ./work/memorybot.js
```

talk to it in Slack, then look at the code and see if you can make sense of what it's doing.

*** CHALLENGE 1 ***

When your bot hears "shout hello", make it reply "HELLO"
(hint: `string.toUpperCase()`)

```
'use strict';

const botx = require('../src/botx');

let bot = botx({
  help: {
    messages: [
      'I am a bot with a small memory...',
      'If you ask me "call me <x>", I will remember that name',
      'and use it when you say "hello" or "hi" to me.'
    ]
  }
});

let name = 'friend'; // 1

bot.log.notice("Starting a new BOTX...");

bot.when(/^call me (\w+)/) // 2
  .then((b, message) => { // 3
    name = message.match[1]; // 4
    b.reply(message, 'OK, from now on you are ' + name);
  })
  .go();

bot.when(/^hi/)
  .or(/^hello/)
  .then((b, message) => { // 5
    b.reply(m, `Well hi there, ${name}.`); // 6
```

```
}  
.go();
```

- (1) Here we're declaring a variable to store the name entered by the user.
- (2) This is another (and more convenient) way of writing regular expressions in JavaScript
- (3) This construct: **(b, message) => { . . . }** creates a function on the fly, which is passed to **.then** as a **callback**. This function will be called whenever the bot matches the user input with its regex, and will be passed two values: the bot, and the message.
- (4) You can get the whole response that the user typed from **message.match[0]**, the first capture group with **message.match[1]**, and so on.
- (5) Here's another callback function.
- (6) And here's another new JavaScript feature. If you enter a string literal in **backticks** (```) you can drop variables right into the string using **`${variable-name}`**, making it easier to format output.

Try one or more of the following challenges, or invent one of your own!

* * * CHALLENGE 2 * * *

When your bot hears "you are **<name>**", make it store the name and reply "OK, I am **<name>**". When you type **hi** add a message that prints "My name is **<name>**" to the responses that the bot already gives.

* * * CHALLENGE 3 * * *

When your bot hears "time now", make it print the current date and time.
(hint; `new Date().toString()`). This is a simple solution: can you think of circumstances in which it would show the wrong date or time?

* * * CHALLENGE 4 * * *

Build a shopping list bot. When your bot hears "shop buy **<something>**", make it add that thing to an array of strings which stores your shopping list. When the bot hears "shop list", print out the current shopping list. When the bot hears "shop reset", clear all the items .
(hint: `array.push(something)`, `array.forEach((item) => {})`)

* * * CHALLENGE 5 * * *

A long time ago there was a program called **Eliza**. This would respond to a user's input as a particular kind of psychotherapist might. You can try it out online, there are numerous versions of it (for example, <http://www.manifestation.com/neurotoys/eliza.php3>). There's no real intelligence here: it's simply finding some common patterns, responding appropriately, remembering some past inputs and making a random prompt (which might incorporate a past response) if it gets stuck. Try writing an Eliza bot.

Task 4 – conversations

So far, our bots listen for things and respond to what they hear. But sometimes we want to structure our bot conversations more carefully, by prompting for responses in a certain order, and switching our questions depending on the answers.

Take a look at

`work/conversationbot.js`

As before, try it out, interact with it, read the code, talk about it, and note any questions.

* * * CHALLENGE 1 * * *

Add a response to the bot for the input "sometimes".

```
'use strict';

const botx = require('../src/botx');

const bot = botx({
  help: {
    messages: [
      'I\'m a slightly insecure bot...',
      'I respond to "good" - please be nice to me!'
    ]
  }
});

bot.log.notice("Starting a new BOTX...");

const areWeGood = bot.conversation() // 1
  .ask('Are we good?')
  .when('yes').thenSay('I\'m so glad we are friends!')
  .when('no').thenSay(['Aww, and there I was hoping...',
    'Oh dear, that makes me sad...'])
  .otherwise('Well, maybe there\'s hope for me yet') // 2
  .create(); // 3

bot.when('^good')
  .thenStartConversation(areWeGood) // 4
  .go();
```

- (1) We create a conversation from the bot by calling **conversation()**, then applying questions and associated patterns and actions.
- (2) If your responses don't deal with all possible inputs, you should always add an 'otherwise' action to deal with things that your bot can't understand
- (3) Call **create()** to actually create the conversation ...

- (4) ... and then have the bot kick off the conversation by asking it to start the conversation you've created.

There's more to conversations than this. You can examine and run the bot code in the examples folder (conversation2.js, conversation3.js) to see what else can be done: briefly:

```
// The statement below is on one line...
const order = 'OK, got your order: {{responses.base}} {{responses.type}} pizza
{{responses.toppings}} topping'; // 1

const pizza = bot.conversation()
  .ask('What sort?')
  .into('type') // 2
  .when('.*')
  .switchTo('base') // 3

  .ask('Thin or deep?', 'base') // 4
  .when('.*')
  .switchTo('toppings')

  .ask('Toppings?', 'toppings')
  .when('.*')
  .thenSay(order)
  .create((responses) => console.log(responses)); // 5

bot.when('pizza')
  .thenStartConversation(pizza)
  .go();
```

- (1) All the responses captured from the user up to any point in the conversation are available to be substituted in the bot's output using this notation, and the name of the thread.
- (2) **into('type')** places the user's response to this question in the field named 'type' in the object that's collecting responses
- (3) This command **switches to another conversation thread ...**
- (4) ... and this is how a **thread is defined**, by giving it a name. This name is also used to store the user's response in the responses object (unless you override that by using **into()**). The effect of this in the code above is to ask the three questions in turn, and gather the responses up under the names **type**, **base** and **toppings**.
- (5) You can pass a callback to the conversation **create** method, which will receive all the responses made during the conversation. In this example you could call out to an online pizza-booking service to get your order delivered.

Try the following challenge, or invent your own:

❖ ❖ ❖ CHALLENGE 2 ❖ ❖ ❖

Write a bot to recommend a book. Ask three questions and work out what sort of book a user might like on the basis of what they've answered and make a recommendation. You'll make life simpler if you constrain the answers to each question (if each question has two responses, then you can propose 8 different books based on their choices).

