ChatScript Practicum: Messaging

Revision 6/9/2018 cs8.3

"'There's more than one way to skin a cat"'. A problem often has more than one solution. This is certainly true with ChatScript. The purpose of the Practicum series is to show you how to think about features of ChatScript and what guidelines to follow in designing and coding your bot.

Sending and receiving messages is what ChatScript and bots are all about. One of ChatScript's unique virtues is that it runs two communcation channels simultaneously. One channel talks with the user. The other talks with the application (which may be just the browser). That second channel is referred to as "out-of-band" communication or OOB. Messages can be sent on either or both channels.

ChatScript tracks the history of communcations with the user from day 1 to infinity. It is a relationship. This relationship consists of one or more conversations. Just as when you come together with a friend, chat for a while, and then separate. Only problem is that ChatScript does not know when you separate, only when you come together anew. The application is responsible for sending a null message indicating the start of a new conversation, like when the user arrives on a conversation page. If no null message is sent, then ChatScript will imagine it is one long continuous conversation. The engine can compute the passage of time since its last volley if the script wants to. But it is perfectly reasonable that a user leaves his computer to go to the bathroom and returns later, and it's still the same conversation.

ChatScript itself is a passive server. If a message is sent to it, it can respond (a volley). Otherwise it sits idle. It does not maintain a connection to anyone beyond the moment of the volley so it cannot initiate a communication. If the bot thinks it might want to initiate a message, it has to have replied to some earlier volley with an OOB request for a callback - it asks the application to send a new message at some specified moment in the future. That message may be a simple dummy message and not involve the user at all. The server cannot do anything itself based on the passage of time - it cannot set a timer and launch a communication. It asks the application to do that on its behalf.

A volley consists of any number of "sentences" of input and any number of sentences of output. I say "sentence" because it may not be a traditional view of a sentence. For example, if you have turned on interjection splitting, then

```
Hi, how are you.
```

becomes two sentences, splitting off the interjection. And ChatScript limits a sentence to 254 words. So if you provide 300 words of input, that will get split into two sentences arbitrarily at the limit.

Input OOB

OOB input is always encased in [ ] and comes before any user input. How you interpret the content of the [] is entirely up to your script, but ChatScript supports some styles better than others. The content of [] is treated as a sentence and is therefore the first sentence of the volley. ChatScript will not attempt spell-correction or parsing, because this is not from the user.

Normally your control script will test for this kind of sentence first, and when detected, process it and move on to the next sentence for real user behavior.

```
topic: ~main_control system repeat ()

u: () $$sentenceCount += 1

#! [ emotion: annoyed pose: standing]
u: ($$sentenceCount==1 < \[ * \]  )
    ^respond(~handle_oob)
    ^end(SENTENCE)

.... real processing for user input ...
```

The responder tests two things. First it checks that this is the first sentence of the volley. If your application always sends OOB, then no user can provide input that would be confused as OOB. If not, then maybe you should provide some data in OOB that users will never stumble onto themselves. Given that, you don't really need to track which sentence it is. Second, of course, the form of an OOB message is detected. I always make a topic to handle processing oob messages and then end the sentence because it should not be subject to responses from the bot.

There are two ways of putting data in an OOB message. One is to put whatever your script may detect. In my sample input I showed a keyword pairs approach of data that might have been detected about the user. One could process this OOB data like this inside the OOB topic:

```
u: ( emotion: _*1)   # do something with this emotion
u: ( pose: _*1) # do something with this posture
```

When your data is small and can fit in 254 words, this is simple enough. But when you get serious amounts of data being passed in, you need to bypass the word limit. For that, you make the content of the OOB be, in fact, a JSON blob. In your bot macro have a line like this to enable use:

```
    $cs_token += #JSON_DIRECT_FROM_OOB
```

ChatScript will detect JSON in OOB, process all the JSON during tokenization, and return instead a single token which is the JSON id. You should not mix these two styles.

```
#! [ [ { name: entity1} { name: entity2} ]]
#! [ { name: entity1 price: $5} ]
u: (\[ _*1 \])  $$data = _0
    # $$data holds the json entity name
    # $$data[0].name is the first entity name if the array blob was used
```

Your top level JSON can be an object or an array. ChatScript accepts "lazy" JSON so while you can use doublequotes around field names and string values, if your names and values don't have special characters like spaces in them, you can omit the quotes. The JSON entity returned is always created as transient facts, so you need to copy out anything you want to keep long term into permanent variables or permanent JSON.

```
u: (\[ _*1 \])  $$data = _0
    $name = $$data.name                      # individual datum
    $jsoncopy = ^jsoncopy(PERMANENT $$data) # full blob
```

Note if you make a copy, it will survive even if you set the variable to null in the future. Facts do not need anyone pointing to them to survive. To remove this permanent JSON, you would need to use `^delete($jsoncopy)`. Or you could force the user to do a reset and lose EVERYTHING about the user.

Input User

User sentences are just normal stuff that you are probably already familiar with. You use a control script, it calls topics which in turn have responders and gambits. But the real problems a control script has to manage is what to do with multiple sentences from the user. What I usually do is scan through all of the sentences first, gathering data that I will use later. And maybe characterize why the user has all these sentences. Are they all statements? Are they statements followed by a terminating question? Are they a bunch of questions? What is the user trying to do?

So my standard way of leafing thru all sentences involves getting a sentence, saving it, calling a topic to analyze it that does NOT generate a response, and then having seen all the input, restoring each sentence and doing real output processing. Like this:

```
topic: ~main_control system repeat()
    ... # OOB and other stuff

# save input to retrieve later
u: GLEAN() $$sentenceCount += 1
    $_tmp = ^saveSentence($$sentenceCount)
    ^nofail(TOPIC ^respond(~glean_data))
    if (%more) {^end(SENTENCE)}
```

```
u: OUTPUT()
    $_count = 0
    loop($$sentenceCount)
    {
        $_count += 1
        ^restoreSentence($_count)
        ^respond(~phase2)
    }
```

The GLEAN rule counts how many sentences there are, saves them, and gleans them until there are no more sentences. The OUTPUT rule restores each sentence and passes it off to a more normal control topic to handle the user output determination. One assumes that the GLEAN topic will have set various facts and variables to remember what it found so the ~phase2 topic can be guided in what it does.

Output OOB

Output OOB follows the same style as input. You put brackets around the output and you make it the first sentence of the output. What is contained in the OOB is whatever agreement you have with the application. It can be keyword pairs, JSON, or whatever.

The requirement that it occur first might lead you to this style of coding to control a bot avatar...

```
u: (I like meat) \[ action: smile \] I like meat, too!
```

This is appropriate enough if you know you will only have one output. But if you may have multiple outputs then things may stack up badly. Also, it's tedious writing this code. I prefer this style:

```
outputmacro: ^action($_act)
    $$oob = ^"$$oob action: $_act"}
outputmacro: ^emote($_emotion)
    $$oob = ^"$$oob emotion: $_emotion"}
```

These create and augment a transient variable with keyword pairs. Or you could create a single macro with two arguments, your choice. Then the script becomes

```
u: (I like meat) ^action(smile) ^emote(happy) I like meat, too!
```

Later you need to have a postprocessing topic like this:

```
topic: ~POSTPROCESS system repeat ()
t: ($$oob)  ^postprintbefore( \[ $$oob \])
```

This will put out OOB data once, in front of all other outputs your bot may have generated. Note that postprocess topics use gambit rules, not responder

rules, since the postprocess happens once after all sentences have been seen, so it has not responder context.

Output user

Output to the user is whatever you want write. There are a couple of tricky situations to consider, however.

Random choices

First, the use of randomized choice [].

```
?: (do you like meat) [Yes.] [No.] [Maybe.]
```

By default rules erase themselves after use. If you have not used KEEP on the topic or rule, then the user will only ever see this rule firing once. Providing randomized responses is overkill. Worse, if your script is part of a corporate bot with regression systems, randomization will play havoc with their expected test results. You should save randomized answers for rules that do not erase themselves AND the user is likely to see often. Something which the user may seen once after every long conversation is probably not going to be an issue. Like a closing "ta ta for now" as a goodbye. It becomes more of a signature than evidence of robotic response.

Note that if you reset the user, you will always get the same random choice at first. The random value used by the system at the start of a volley is a combination of the user's name and what volley count it is. Thereafter during the volley, the value fluctuates "randomly" along a course set by that current seed. So when you reset the user, you start the same sequence all over again if you say the same inputs. You `can` force a different value, but really there is no reason to usually. Users will not usually see this repeat, since they have a continuous existence with volley count always rising. It's only developers who see this and repeatability is useful for testing.

Formatting

Normally ChatScript automatically handles formatting output. You don't have to pay attention to whether your comma is freestanding or attached at the end of a word. It doesn't matter how many spaces occur between words. Numbers can automatically have commas added (in varying nationalities with the right controls).

`$cs_response` is a variable that controls some output processing by ChatScript.

```
$cs_response = #OUTPUT_NOCOMMANUMBER | #ALL_RESPONSES
```

The above line, in the bot macro, makes the system not automatically add commas in numbers. It also requests standard behaviors like uppercasing the

start of a sentence, removing spaces before commands, converting underscores to spaces, and others. You could add `#OUTPUT_NOQUOTES` to automatically take off double quotes from strings. Or `#OUTPUT_FULLFLOAT` to get full precision from a float (normally CS truncates on output to 2 places). Go find documentation on the choices.

But when you really want precision control in layout, then you use active strings.

```
u: (test) ^"I   $verb , meat."
```

This active string will keep the spaces you see here when delivered to the user.

`#OUTPUT_NODEBUG` blocks the system from trying to interpret words starting with : as being debug commands.

Accents

Suppose you want your bot to talk with a French accent. Or use texting. You could, of course, author your output text directly to do this. But then you can't readily switch to a different accent. And it may be a pain to remember to do this on all your outputs. Another way to do an accent is to write code that rewrites your outputs according to the appropriate accent. This is something you can do in postprocessing. You can have it read your outputs and rewrite them.

```
topic: ~POSTPROCESSING system repeat ()
t: ()
    $_count = 0
    Loop(%response)
    {
        $_count += 1
        $_msg = ^response($_count)
        $_newmsg = ^altermessage($_msg)
        ^setResponse($_count $_newmsg)
    }
```

This loop retrieves all the outputs, calls a routine to tamper with the content appropriately and then replaces the response with the new response.

What can you do in `^altermessage`? You could hunt for "yes" and replace it with "oui" and alter spellings to give a French flavor. Or find words and replace them with texting equivalents. whatever you want to do.

Self Reflection

The above loop sees exactly what you wrote to the user from a rule. A message might consist of multiple sentences. If you want to see things on a sentence by sentence basis, you can get a tokenized form of your output. This allows you to look at each actual sentence independently. You might, for example, record your random answers to questions so that you can be consistent in the future. The

user says “do you like meat” and your script randomly picks from `Yes no` and then you detect and record that hear so you give the same answer in the future.

Something I do is automated pronoun analysis. I get the sentence, do a full analysis on it as with any input from a user, then run a topic designed to look for pronoun resolution data. So if my bot says “my mother is English”, it can prepare a reference for `she` to mean `my mother`, should the user say `she` in their input.

```
topic: POSTPROCESSING system repeat ()
t: (^query(direct_v ? chatoutput ? -1 ? @chatoutput  ))
    $_counter = 0
    loop()
    {
        $_counter += 1
        $_utterance = ^last(@chatoutputsubject)
        ^analyze($_utterance)
        nofail(RULE respond(~acquirepronouns))
    }
```

ChatScript creates transient facts for the volley’s output, I retrieve them and look at them with a script topic designed for pronoun reference detection.

Taking the initiative

Normally ChatScript has to wait for a user to reply. But a convention built into the browser pages for ChatScript and recommended for developers of apps, is to support OOB callback mechanisms. This allows your bot to initiate further messaging under various timed conditions. The callbacks are:

```
OUTPUT OOB              INPUT CALLBACK RESPONSE OOB
[callback=10000]    =>   [callback]
[loopback=15000]    =>   [loopback]
[alarm=400000]      =>   [alarm]
```

For output OOB callback, the meaning is if the user does not press a key within the millisecond count given, issue the callback OOB message to ChatScript. If the user is starting to respond, cancel the callback.

For output OOB loopback, create an implied callback every volley, issuing a loopback response if time runs out before the user presses a key. If the user presses a key, suppress loopback for this volley but keep going for future volleys.

For output OOB alarm, set a timer to go off in the milliseconds indicated and when it expires send the alarm OOB to ChatScript as soon as convenient thereafter. If user is midst of a volley, wait for that to complete.

All of these timers can be canceled with a value of 0 to the respective one.

This is a purely voluntary convention. Any app developer is free to create their own conventions or ignore these.

Ex Post Facto

Suppose you want to issue message A normally, but later you have a message B, which if it fires, wants to fire without message A having fired. A will happen first and will not know if B will happen later. What do you do?

As it turns out, with CS introspection, message B can determine that message A happened and can cancel it. It's quite easy. Assume we are at message B.

```
^nofail(RULE $_why = ^ResponseRuleID(1))
if (^substitute(character $_why REASON xx FAIL)) {^reviseoutput(1 "")}
```

The above code hopes there is already a response and gets who generated it (or $_why remains null). Then if the why was generated by message A (here labelled REASON), then message 1 is set to null. So only message B's output will be seen by the user.