Guidelines for designing intelligent bots, using Random Access Navigation

February 23, 2017

by Sabin Ielceanu

At *Intwixt* we see the most value in building intelligent, rule based bots that leverage both the rich, non-chat based UI experience (such the one offered by Messenger) and certain AI aspects such as NLP. While many platforms promote one approach over another, we see value in providing a framework that encapsulates the best of both: the *programmed intelligence* provided by the guided interactions and the *artificial intelligence* of NLP. Our process-first approach is capable of delivering the rules you need to define your bot's intelligence, while our integration-based architecture lets you tightly integrate the best NLP platforms available.

With this approach, the conversations are not modeled inside and served by the NLP service (API.AI, Wit.AI, etc.) but rather in a platform that provides a tight integration with the NLP service. The bots are fully functional and provide a very good, guided user experience without using NLP. However, NLP is critical for ensuring a great user experience even for rule based bots. That is, mainly for two reasons. First, it enables speech recognition. Second, it provides a user experience where the end user can, at any point in the conversation, break the conversation flow, change the conversation context and then resume the conversation flow without restarting it from the beginning. This feature has been coined Random Access Navigation by Shane Mac of Assist. A great post that talks about its benefits in more details, can be found here.

This style requires a different way of thinking about your bot implementation. First, you need to separate out and centralize the logic that deals with NLP from the rest of the logic. Then you have to think more about modularity and reusability and make sure that the conversation logic is “reentrant” (see below). Most of these things are quite difficult to get right from the beginning. Therefore, the platform that you're choosing for your bot implementation has to provide a very agile development model especially around designing conversations.

To help with the process of designing bots that combine programmed intelligence and artificial intelligence (NLP), we put together a few guidelines and tips.

1. Identify the actions of the bot

Focus on the utility aspect of the chatbot. Its ability to "chat" should not be seen as an end unto itself; rather, it should be used for gathering information. The bot's skills are what matter most. In the same way a seasoned call center agent is most helpful in providing details on navigating the ins and outs of a company, chatbots are most useful to customers when they have tangible skills that focus on task completion. Look at the ability to "chat" as a way to remove the friction for fulfilling the action.

At this stage, the most important question to ask is "*What does the bot do?*" What are the low-level actions that the bot can fulfill? For example, a *shopping district concierge* bot could model the following low-level actions:

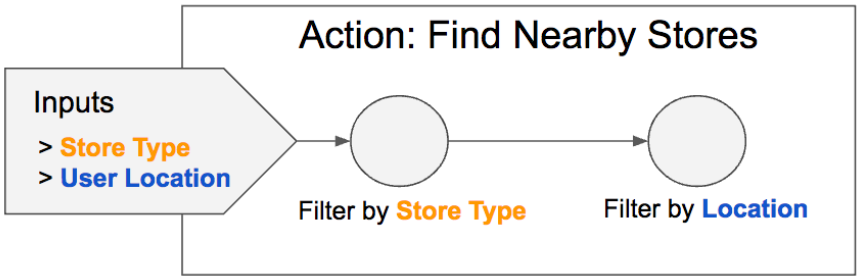
- get-all-stores
- get-stores-by-type
- find-nearby-stores
- get-store-details
- get-store-schedule
- call-store, etc

These actions represent, collectively, the skill(s) of your bot. Think about these actions as *the API* of your bot. Model their input, output and fault data types as you would model a traditional API. Even if you do not end up implementing them as an API, this exercise will help you rationalize the implementation of your bot.

Most, if not all of these actions are likely modeled as "intents" in a natural language processing platform such as API.AI. Each one would require chatbot training with different utterances. In general, you will be able to expand the list of utterances without affecting the implementation of the action (i.e. the bot implementation).

2. Implement the actions

Implement each individual action as a separate, reusable entity that requires very specific data to fulfill. When you define the data types, pay attention to the cardinality of the parameters and make sure they reflect the requirements of the implementation. At this level, you should avoid adding logic that interacts with the user in order to gather the missing/additional data. That is usually done inside data collectors.



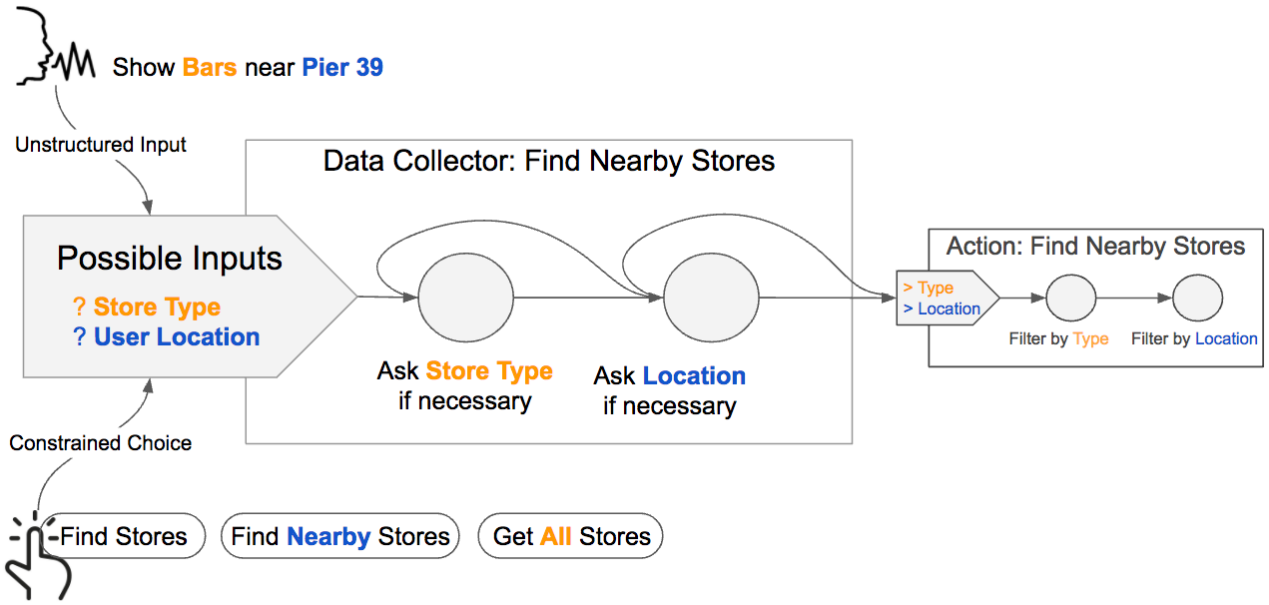
An action implementation encapsulates the business logic of the action. In general, it orchestrates interactions with backend services to fulfill the action. Pay attention to using UI primitives in action implementations. If you see them, ask yourself whether they really need to be there. They will impact the ability to share the action implementations across the platforms you deploy the bot on (Messenger, Slack, etc.). Ideally, you want the action implementations to be independent of the messaging platform.

The data that is passed to the action is very likely modeled as "entities" in API.AI. Each entity has a name, a list of values and each value has a list of synonyms. For example, a "store-type" entity could have a "restaurant" value with ["diner", "place that sells food"] as synonyms. It could also have a "bar" value with ["pub", "place that sells drinks"] as synonyms.

3. Implement the Action Data Collectors

The end user should be able to trigger an action in different ways: by

clicking on a button, typing in a text message or by issuing a voice command. From an implementation perspective, these represent different entry points into the process that validates the consistency of the data that is passed to the action. We call these *action data collectors* or simply *collectors*. This process should *not* expect that all the data required to fulfill the action is provided by the user. It is very likely that a guided interaction would ask the user to provide every single required data whereas a text or voice command will not. Therefore, the bot should be ready to ask the appropriate questions to collect the missing data. The fulfillment logic should be the same regardless of how the data is collected.



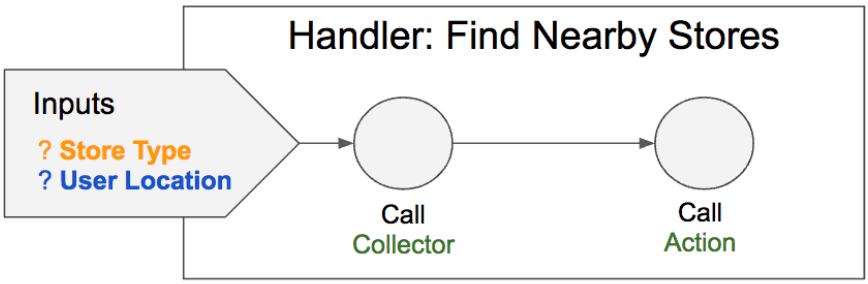
It is worth noting that data collectors may use complex user interface controls to collect data such as webviews or they may combine a series of Q&A style inputs to collect the information sequentially. The conversation primitives available depends upon the messaging platform. For example, Messenger has a very different set of conversation primitives than Twitter.

The platform you use for modeling your bot must provide the tools necessary for modeling conversations. Conversations can grow in complexity quite fast, so you need tools that help you easily build, visualize, tear down and rebuild the conversation logic. We designed the

Intwixt platform and tools with this in mind.

4. Implement the Action Handlers

We recommend that you do not call an action directly from a collector but rather you use a different entity that calls both the collector and the action. That way you can reuse and chain collectors to form more complex ones. We call these *action handlers* or simply *handlers*.



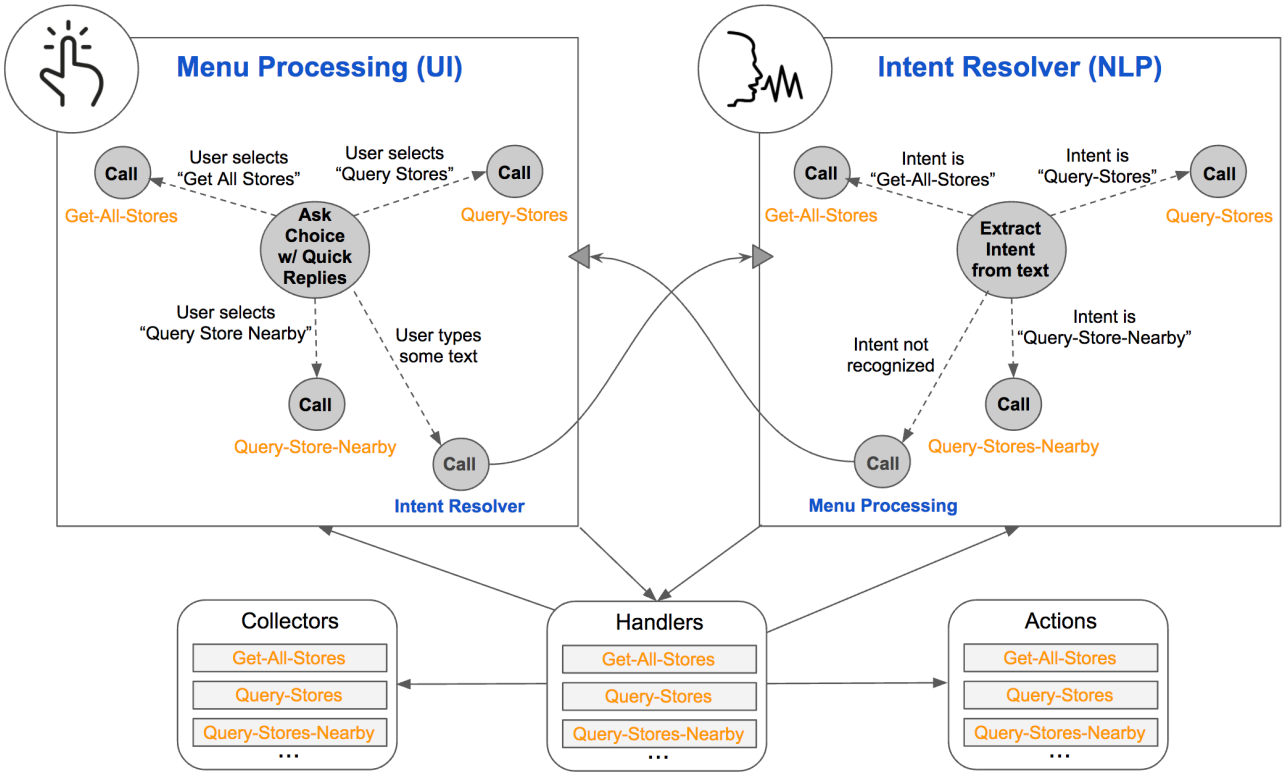
The input data of the handler mirrors the input data of the collector. It is very common for a handler to have more activities than the two invocations shown in this example. For example, a handler could have the logic that deals with what happens next after the action is executed (e.g. provide a menu of options to the user, etc.)

The handler represents all the data collection, validation and execution logic for a given action. It is the action-specific entity that is shared across different interaction models that the bot exposes to users (e.g. rich UI primitives, NLP).

5. Separate the NLP logic from the UI logic

Remember that the bot has both *static aspects* like the actions it can fulfill, or the questions it needs to ask to fulfill a specific action, as well as *AI aspects* like its ability to understand what a user is asking for and its ability to learn new ways of of asking the same thing. You want to separate them and make sure they can call each other. You want to make sure the user is

allowed to interrupt the normal conversation flow driven with UI controls by using a text command. At the same time, after extracting the intent from a text command, you may need to take the user through a conversation to gather more data. From a dependency perspective, you end up with intertwined, cyclic dependencies between some of these entities. This is OK. Don't fight it. Embrace it. After all, isn't this the way we communicate?



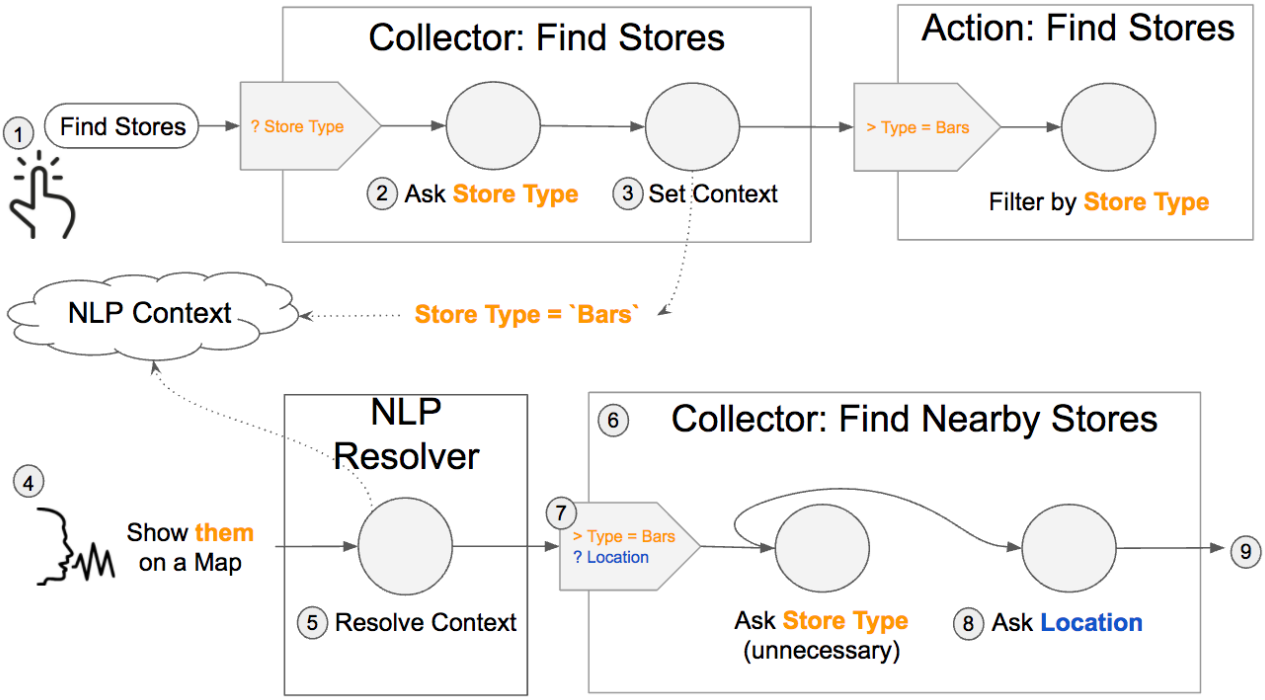
This pattern shows how effective the dual nature of the conversation with the user can be, when it leverages both deterministic UI controls (such as buttons and carousels) and NLP. A smart bot should anticipate what the user wants to do next and offer those choices to the user. Most of the times, that cannot be done effectively in text. User-guided interactions through UI controls are superior interactions in many cases. Pushing a button is simpler than typing in or even dictating a command. However, at any point the user should be able to break the conversation and request an action by typing in a text or issuing a voice command. The action can be either a new action, that is not part of the choices provided to the user or it can be an action that is in fact provided to the user in the menu of choices. Moreover, this action can also revert or change the outcome of a previous

interaction, which can be a very effective way to change the state of the conversation without asking the user to start over.

When the NLP logic doesn't understand the command (i.e. the NLP service is not able to extract the proper intent/action from the text), the bot could send an appropriate message and present a menu of options to the user.

6. Share context across the bot's implementation layers

Context is King. Your bot's intelligence heavily depends on how effectively you model, manage and share context across all the layers that participate in the decision making process. For example, what does a user expect when he/she says "*Show me more*"? Are they asking for more stores or more details about a specific store? Unless you share context between the NLP Service, the NLP logic of your bot, and the rest of your bot's conversation logic, it is very likely that your bot won't be able to properly answer this question.



As the user navigates, and issues text/voice commands, the bot accumulates context that must be shared with every layer that can be affected by it.

If you are using a platform such as API.AI, you must store the context in its space for it to be able to properly perform the intent resolution. Ensure that the user experience is consistent regardless of how the user issues the command, whether through text/voice or by selecting a menu option.

7. Hybrid (bot + human) interactions

When the bot does not understand what the user is asking for or it does not know how to fulfill the requested action, it can either request a human being to take over the conversation or it can flag the conversation and respond with a static message. This is a design decision that is usually driven by a strong product management requirement. Regardless of what approach you take, make sure you monitor the bot to understand its deficiencies. As you do that and discover utterances that you missed, you should be able to train your bot with them in the NLP Service usually without changing your bot implementation.

Final Thoughts

Bots that combine programmed intelligence with artificial intelligence provide a far superior user experience than the bots built with only one in mind. You can build them today. If you have any questions or you need help, please do not hesitate to reach out. We would love to learn about your use-case and help you build amazing bots.

Comments (0)

Newest First Subscribe via e-mail

Preview POST COMMENT...

tagged with Natural Language Processing, NLP, NLP chatbot, random access navigation

Newer / Older

TERMS OF SERVICE PRIVACY POLICY VIDEOS

email info@intwixt.com
phone 415.570.8502