Chapter 7: Building Chatbots

In this chapter, we will build chatbots using two different frameworks, the **`nltk.chat`** package and the Rasa framework. The first recipe talks about the **`nltk.chat`** package, where we build a simple keyword matching chatbot, and the rest of the chapter is devoted to Rasa. **Rasa** is a complex framework that allows the creation of very sophisticated chatbots, and we will have a basic introduction to it. We will build a default bot using Rasa, and then we will modify it to do simple interactions.

Here is the list of this chapter's recipes:

- Building a basic chatbot with keyword matching
- Building a basic Rasa chatbot
- Creating question-answer pairs with Rasa
- Creating and visualizing conversation paths with Rasa
- Creating actions for the Rasa chatbot

Technical requirements

In this chapter, we will use two different packages for our chatbots; one is **`nltk.chat`** and the other is the Rasa framework. To install these packages, use the following commands:

```
pip install nltk
pip install rasa
```

Building a basic chatbot with keyword matching

In this recipe, we will build a very basic chatbot that uses keywords to match inquiries and responses. This is based on the **`NLTK Chat`** class.

Getting ready

We will start by creating a new chat class and initializing it with pairs of responses. Then, we will run it on the command line. If you haven't al-

ready, you will need to install the **nltk** package:

```
pip install nltk
```

How to do it...

We will use the NLTK chat framework to create a simple chatbot. First, we will create the question-answer pairs and then create the **Chat** object. We will then use it to converse with the bot:

1. Do the necessary imports:
```
from nltk.chat.util import Chat, reflections
```
2. Initialize the conversation pairs:
```
pairs = [[r".*hi|hello|hey|what's up.*",
          ["Hello, I am a simple chatbot. How are
you?"]],
          [r'.*(how are you|and you).*',
           ["Doing well, thank you.",
            "I'm ok, what's new?"]],
          [r"quit",
           ["Bye, nice talking to you!",
            "Talk to you later!"]],
           [r".*weather|rain|snow|sun.*",
            ["Isn't it nice out today?",
             "Hopefully, tomorrow the weather will
get
              better.",
             "I don't know about you, but I am glad
I am \
              inside today."]],
          [r"I like (.*)",
           ["Why do you like %1?",
            "What about %1 do you like?"]],
          [r".*(tasty).*", ["Yeah, I think so
too."]],
          [r".*",
           ["I am sorry, I don't understand. \
             I am a very simple chatbot!"]]
         ]
```
3. Create a new chatbot:
```
chatbot = Chat(pairs, reflections)
```
4. Use the chatbot's **converse** function to carrv out the conversation:

```
chatbot.converse()
```

It might go something like this, where lines preceded by > are user input:

```
>hello
Bot: Hello, I am a simple chatbot. How are you?
>Good, and you?
Bot: I'm ok, what's new?
>I like pizza
Bot: What about pizza do you like?
>It's tasty.
Bot: Yeah, I think so too.
>Do you speak other languages?
Bot: I am sorry, I don't understand. I am a very
simple chatbot!
>quit
Bot: Bye, nice talking to you!
```

How it works...

In *step 1*, we import the **`Chat`** class and the **`reflections`** dictionary. This dictionary maps first-person phrases to their corresponding second-person expressions. For example, it will map *I am to you are*.

In *step 2*, we create inquiry-response pairs for the chatbot. The variable is a list of lists. Each sub list has two elements: the first one is the regular expression that we use to match on the user input, and the second one is a list of possible responses, where one is chosen at random. The regular expressions can include groups that can be referenced in the response, as in the **`[r"I like (.*)", ["Why do you like %1?", "What about %1 do you like?"]]`**list. The **`%1`** element in the response refers to the **`(.*)`**group in the input. If you have more than one group, you can refer to them by numbers, as in **`%2`**, **`%3`**, and so on. We also include a list with the **`.*`** input regular expression that has the line **`I am sorry, I don't understand. I am a very simple chatbot!`** as the response. This regular expression will match anything but only after it tries other expressions, so this is the default option when we don't really know what to say.

In *step 3*, we initialize the chatbot using the reflections dictionary and the inquiry-response pairs. In *step 4*, we use the **`converse`** function to start the conversation. We see that it matches the inputs that we specified and

asks us whether we like pizza. It also uses the default response when it doesn't have an entry for the input **`Do you speak other languages?`**

There's more...

NLTK has other chatbots that it already created: **`eliza_chat`**, **`iesha_chat`**, **`rude_chat`**, **`suntsu_chat`**, **`zen_chat`**. For example, you can try the ELIZA bot as follows:

1. Do the necessary imports:
```
from nltk.chat.eliza import eliza_chat
```
2. Initialize the conversation pairs:
```
eliza_chat()
```
The output might look like this:
```
Therapist
---------
Talk to the program by typing in plain English,
using normal upper-
and lower-case letters and punctuation.  Enter
"quit" when done.
==========================================================
====================
Bot: Hello.  How are you feeling today?
>Good
Bot: Very interesting.
>I like pizza
Bot: How does that make you feel?
>Hungry
Bot: How does that make you feel?
>quit
Thank you, that will be $150.  Have a good day!
```

Read more about the ELIZA chatbot at **https://en.wikipedia.org/wiki/ELIZA**.

Building a basic Rasa chatbot

In this recipe, we will use a popular chatbot framework, Rasa, to build a default chatbot. In the coming recipes, we will make the chatbot better.

Rasa is an open source deep learning framework for building chatbots. It uses Keras and Tensorflow to implement the model. You can read more about the implementation here: **https://blog.tensorflow.org/2020/12/how-rasa-open-source-gained-layers-of-flexibility-with-tensorflow-2x.html**.

Getting ready

We will initialize the Rasa framework and use it to build and initialize a default Rasa chatbot and then we will explore its structure. If you haven't already, install the **rasa** package:

```
pip install rasa
```

How to do it...

After installing the **rasa** package, there are new commands available through the Rasa interface. We will use them to create a default chatbot. The steps for this recipe are as follows:

1. On the command line, enter this:

   ```
   rasa init
   ```

   Rasa will start and will produce some colorful output. After that, it will ask you for the path you want to create the new chatbot in. The program output might look like this:

   ```
   2020-12-20 21:01:02.764647: I tensorflow/
   stream_executor/platform/default/dso_loader.cc:48]
   Successfully opened dynamic library
   cudart64_101.dll
   ┌──────────────────────────────────────────────────
   ──────────────────────────────────┐
   │ Rasa Open Source reports anonymous usage
   telemetry to help improve the product │
   │ for all its
   users.
                  │
   ||
   │ If you'd like to opt-out, you can use `rasa
   telemetry disable`.                  │
   │ To learn more, check out https://rasa.com/docs/
   rasa/telemetry/telemetry.        │
   └──────────────────────────────────────────────────
                                    ┘
   ```

```
…
Welcome to Rasa! 🤖
To get started quickly, an initial project will be
created.
If you need some help, check out the documentation
at https://rasa.com/docs/rasa.
Now let's start! 👇🏽
? Please enter a path where the project will be
created [default: current directory]
```

2. Enter the path and press *Enter*:

```
./Chapter07/rasa_bot
```

3. If the path does not exist, Rasa will ask if it needs to be created:

```
? Path './Chapter07/rasa_bot' does not exist 🧐.
Create path?
```

4. Answer **Yes** and then the program will ask you if you want to have an initial model trained:

```
Created project directory at 'C:
\Users\zhenya\Documents\Zhenya\consulting\book\code
\python-natural-language-processing--
cookbook\Chapter07\rasa_bot'.
Finished creating project structure.
? Do you want to train an initial model? 💪  Yes
```

5. The model will train and Rasa will ask you if you want to interact with the bot; answer **Yes**:

```
? Do you want to speak to the trained assistant on
the command line?   Yes
2020-12-21 07:20:53 INFO     rasa.model  - Loading
model
Chapter07\rasa_bot\models\20201221-071928.tar.gz...
2020-12-21 07:20:55 INFO     root  - Starting Rasa
server on http://localhost:5005
2020-12-21 07:20:55 INFO     rasa.model  - Loading
model
Chapter07\rasa_bot\models\20201221-071928.tar.gz...
2020-12-21 07:21:03 INFO     root  - Rasa server is
up and running.
Bot loaded. Type a message and press enter (use '/
stop' to exit):
```

Our conversation might go something like this:

```
Your input ->  Hello
```

```
Hey! How are you?
Your input ->  Good, and you?
Bye
Your input ->  wow
Great, carry on!
Your input ->  where are you located?
I am a bot, powered by Rasa.
Your input ->  where do you live?
Hey! How are you?
Your input ->  /stop
```

6. Once we input **/stop**, the program will stop execution and will return to the command line. To power up the chatbot again, change the working directory to the **bot** path and start it up again:

```
cd Chapter07
cd rasa_bot
rasa shell
```

It will load up again:

```
…
Bot loaded. Type a message and press enter (use '/
stop' to exit):
```

How it works…

In *step 1*, we initialize Rasa. This command is used to create a new chatbot project. In the last line, Rasa asks us about the location of the project files. In *step 2*, we enter a path, an example path is provided, but you can use any path you like. The path specification should follow Unix formatting, so the dot in the path signifies the current directory.

In *step 3*, Rasa asks us if the directory needs to be created. In *step 4*, after we answer **Yes**, Rasa creates the necessary directories and then asks us if we want to have an initial model trained.

In *step 5*, Rasa asks us if we want to interact with the trained assistant. After we answer **Yes**, we try out different inputs to see what the bot knows and how it answers. We see that at this point it pretty much recognizes greetings. In the next recipe, we will add some more utterances that it will be able to handle.

In *step 6* we stop the bot. Then we change to the **bot** directory and power it up again.

There's more...

Let's now look at the file structure of the project:

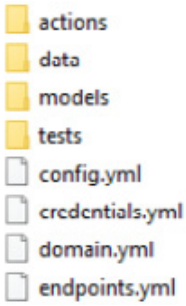


Figure 7.1 – File structure of the project

At the top level, the two most important files are **`config.yml`** and **`domain.yml`**. The configuration file specifies how the chatbot should be created and trained, and the **`domain.yml`** file lists the possible intents it can handle and which responses it should provide for those intents. For example, there is a *greet* intent, and the response to that is **`Hey! How are you?`** We can modify this file to create our own custom intents and custom responses to those intents.

In addition to the **`config.yml`** and **`domain.yml`** files, there are important files in the **`data`** directory:

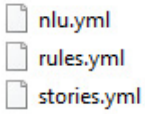


Figure 7.2 – Files in the data directory

The **`nlu.yml`** file contains example inputs for each intent. The **`rules.yml`** file specifies what should be said when. For example, it pairs up the goodbye response with the goodbye intent of the user. Finally, the **`stories.yml`** file defines conversation paths that might happen during interactions with the user.

See also

Rasa has excellent documentation that can be found at **https://rasa.com/docs/**.

Even this simple bot can be connected to different channels, such as your website, social networks, Slack, Telegram, and so on. See the Rasa docu-

mentation on how to do that at**https://rasa.com/docs/rasa/connectors/your-own-website**.

Creating question-answer pairs with Rasa

Now we will build on the simple chatbot that we built in the previous recipe and create new conversation pairs. Our bot will answer simple, frequently asked questions for a business, such as questions about hours, address, and so on.

Getting ready

We will continue using the bot we created in the previous recipe. Please follow the installation instructions specified there.

How to do it...

In order to create new question-answer pairs, we will modify the following files: **`domain.yml`**, **`nlu.yml`**, and **`rules.yml`**. The steps are as follows:

1. Open the **`domain.yml`** file and in the section named **`intents`**, add an intent named **`hours`**. The section should now look like this:

```
intents:
  - greet
  - goodbye
  - affirm
  - deny
  - mood_great
  - mood_unhappy
  - bot_challenge
  - hours
```

2. Now we will create a new response for a question about hours. Edit the section named **`responses`** and add a response named **`utter_hours`** that has the text **`Our hours are Monday to Friday 9 am to 8 pm EST`**. The **`responses`** section should now look like this:

```
responses:
utter_greet:
  - text: "Hey! How are you?"
utter_cheer_up:
```

```
    - text: "Here is something to cheer you up:"
      image: "https://i.imgur.com/nGF1K8f.jpg"
  utter_did_that_help:
    - text: "Did that help you?"
  utter_happy:
    - text: "Great, carry on!"
  utter_goodbye:
    - text: "Bye"
  utter_iamabot:
    - text: "I am a bot, powered by Rasa."
  utter_hours:
    - text: "Our hours are Monday to Friday 9 am to 8
  pm EST."
```

3. Now we will add the user's possible utterances. Open the **`nlu.yml`** file in the **`data`** folder and add a section under **`nlu`** where the intent is **`hours`** and there are some examples of how the user might inquire about hours. It should look like this (feel free to add more examples):

```
- intent: hours
  examples: |
    - what are your hours?
    - when are you open?
    - are you open right now?
    - hours
    - open
    - are you open today?
    - are you open
```

4. We can also use regular expressions to express utterances that contain certain words:

```
- regex: hours
  examples: |
    - \bopen\b
```

5. Now we will add a rule that will make sure that the bot answers about hours when asked about them. Open the **`rules.yml`** file in the **`data`** folder and add a new rule:

```
- rule: Say hours when asked about hours
  steps:
  - intent: hours
  - action: utter_hours
```

6. Now we will retrain the model. On the command line, enter the following:

```
rasa train
```

```
rasa train
```

7. We can now test the chatbot. Enter the following on the command line:

```
rasa shell
```

The conversation might go something like this:

```
Bot loaded. Type a message and press enter (use '/
stop' to exit):
Your input ->  hello
Hey! How are you?
Your input ->  what are your hours?
Our hours are Monday to Friday 9 am to 8 pm EST.
Your input ->  thanks, bye
Bye
Your input ->  /stop
2020-12-24 12:43:08 INFO     root  - Killing Sanic
server now.
```

How it works...

In *step 1*, we add an additional intent, hours, and a response that will follow a user's utterance classified as this intent. In *step 2*, we add a response to that intent.

In *step 3*, we add possible ways a user might inquire about hours of operation. In *step 4*, we connect the user's inquiry and the bot's response by adding a rule that ensures that the **`utter_hours`** response is given for the **`hours`** intent.

In *step 5*, we retrain the model, and in *step 6*, we launch the retrained chatbot. As we see from the conversation, the bot answers correctly to the **`hours`** inquiry.

Creating and visualizing conversation paths with Rasa

We will now upgrade our bot to create conversation paths that start and end with greetings and will answer the user's questions about the business' hours and address.

Getting ready

In this recipe, we continue using the chatbot we created in the *Building a basic Rasa chatbot* recipe. Please see that recipe for installation information.

How to do it...

We will add new intents and new replies and create a conversation path that can be visualized. The steps are as follows:

1. We start by editing the **`domain.yml`** file. We will first add two intents, **`address`** and **`thanks`**. The intents section should now look like this:

```
intents:
  - greet
  - goodbye
  - affirm
  - deny
  - mood_great
  - mood_unhappy
  - bot_challenge
  - hours
  - address
  - thanks
```

2. Now we will add three new chatbot utterances to the **`responses`** section, so it will look like this:

```
responses:
utter_greet:
  - text: "Hey! How are you?"
utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://i.imgur.com/nGF1K8f.jpg"
utter_did_that_help:
  - text: "Did that help you?"
utter_happy:
  - text: "Great, carry on!"
utter_goodbye:
  - text: "Bye"
utter_iamabot:
  - text: "I am a bot, powered by Rasa."
utter_hours:
  - text: "Our hours are Monday to Friday 9 am to 8
pm EST."
```

```
pm 2b1:
utter_address:
  - text: "Our address is 123 Elf Road North Pole,
88888."
utter_help:
  - text: "Is there anything else I can help you
with?"
utter_welcome:
  - text: "You're welcome!"
```

3. In the **`nlu.yml`** file in the **`data`** folder, we will add the possible user utterances for the **`address`** and **`thanks`** intents:

```
- intent: address
  examples: |
    - what is your address?
    - where are you located?
    - how can I find you?
    - where are you?
    - what's your address
    - address
- intent: thanks
  examples: |
    - thanks!
    - thank you!
    - thank you very much
    - I appreciate it
```

4. Now we will create stories for possible scenarios. The first scenario will have the user asking about hours and then address, and the other story will have the user first asking about the address and then hours. In the **`stories.yml`** file, enter the two stories:

```
- story: hours address 1
  steps:
  - intent: greet
  - action: utter_greet
  - intent: hours
  - action: utter_hours
  - action: utter_help
  - or:
    - intent: thanks
    - intent: goodbye
  - action: utter_goodbye
- story: hours address 2
```

```
  steps:
  - intent: greet
  - action: utter_greet
  - intent: address
  - action: utter_address
  - action: utter_help
  - intent: hours
  - action: utter_hours
  - intent: thanks
  - action: utter_welcome
  - intent: goodbye
  - action: utter_goodbye
```

5. Now we will train the model and start up the bot:

```
rasa train
rasa shell
```

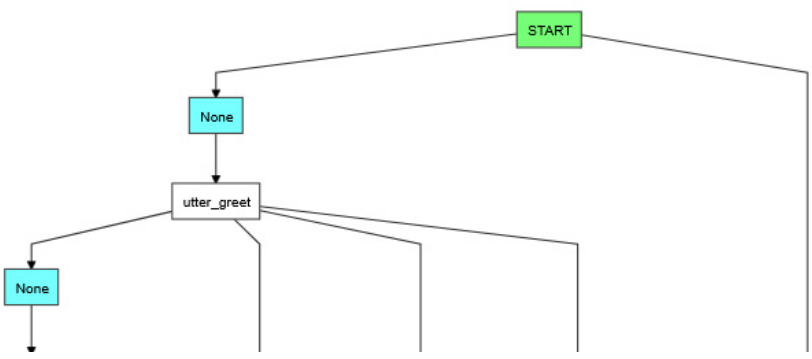
Our conversation might look like this:

```
Your input ->  hello
Hey! How are you?
Your input ->  what is your address?
Our address is 123 Elf Road North Pole, 88888.
Is there anything else I can help you with?
Your input ->  when are you open?
Our hours are Monday to Friday 9 am to 8 pm EST.
Your input ->  thanks
You're welcome!
Your input ->  bye
Bye
Your input ->  /stop
```

6. We can visualize all the stories that this bot has. To do that, enter the following command:

```
rasa visualize
```

The program will create the graph and open it in your browser. It will look like this:



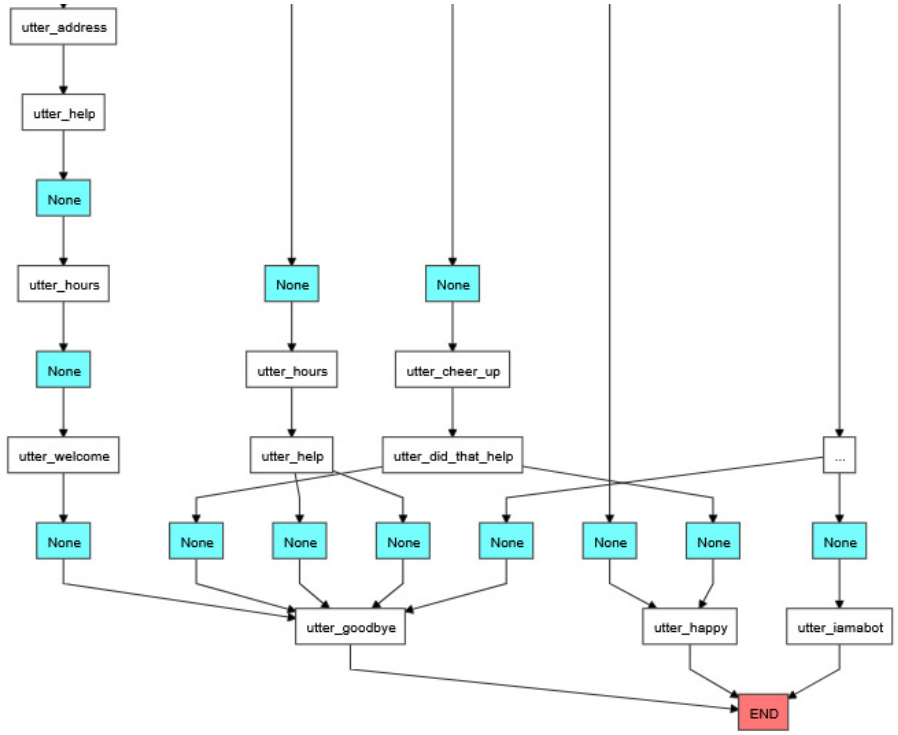


Figure 7.3 – The story graph

How it works...

In *step 1*, we add new intents to the chatbot, and in *step 2*, we add responses to those intents. In *step 3*, we add possible user's utterances for each of the intents added in *step 1*.

In *step 4*, we create possible stories that might come up. In the first story, the user greets the bot and inquires about the hours. After that, one of two intents might come up: either **`thanks`** or **`goodbye`**, which is handled with an **`or`** statement. In the second story, the user first asks about the address and then the hours.

In *step 5*, we train the new model and load the chatbot. The conversation in the example follows the second story.

In *step 6*, we create a visualization graph of all the chatbot stories, including the ones that were predefined. You see that all three stories start with the greeting and continue from there. There is also a rule that is not in any of the stories, and that is answering with the **`utter_iamabot`** response to a user's **`bot_challenge`** intent.

Creating actions for the Rasa chatbot

In this recipe, we will add a custom action and greet the user by name.

Getting ready

In order to create custom actions, we will need to install the **`rasa_core_sdk`** package:

```
pip install rasa_core_sdk
```

How to do it...

We will first edit the configuration files, adding necessary information. Then, we will edit the **`actions.py`** file, which programs the necessary actions. We will then start the **`actions`** server and test the chatbot:

1. First, in the **`domain.yml`** file, add a special intent called **`inform`** that may contain entities. The section will now look like this:
```
intents:
  - greet
  - goodbye
  - affirm
  - deny
  - mood_great
  - mood_unhappy
  - bot_challenge
  - hours
  - address
  - thanks
  - inform
```
2. In the same file, add a new section called **`entities`** where **`name`** is the entity:
```
entities:
  - name
```
3. Add a new chatbot utterance that will ask the question about the user's name:
```
responses:
…
utter_welcome:
  - text: "You're welcome!"
utter_ask_name:
  - text: "What's your name?"
```

4. Next, add a section called **`actions`** in the same file, **`domain.yml`**, where the one action will be **`action_save_name`**:

```
actions:
  - action_save_name
```

5. Next, in the **`nlu.yml`** file, add some examples for the **`inform`** intent:

```
- intent: inform
  examples: |
    - my name is [Zhenya](name)
    - [Philip](name)
    - [Michelle](name)
    - [Mike](name)
    - I'm [Helen](name)
```

6. In the **`stories.yml`** file, add the following to each story after the greeting:

```
  - action: utter_ask_name
  - intent: inform
    entities:
    - name: "name"
  - action: action_save_name
```

   Our FAQ story will now look like this:

```
- story: hours address 2
  steps:
  - intent: greet
  - action: utter_greet
  - action: utter_ask_name
  - intent: inform
    entities:
    - name: "name"
  - action: action_save_name
  - intent: address
  - action: utter_address
  - action: utter_help
  - intent: hours
  - action: utter_hours
  - intent: thanks
  - action: utter_welcome
  - intent: goodbye
  - action: utter_goodbye
```

7. Now open the **`endpoints.yml`** file and add or uncomment the following two lines:

```
action_endpoint:
```

```
action_endpoint:
  url: "http://localhost:5055/webhook"
```

8. Now we will edit the **`actions.py`** file and add our custom action. Open the **`actions.py`** file, which is located in the **`actions`** folder of your bot. Add the following code to it:

```
from typing import Any, Text, Dict, List
from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher
class ActionSaveName(Action):
    def name(self) -> Text:
        return "action_save_name"
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) ->
List[Dict[Text,
                                                A
ny]]:
        name = \
        next(tracker.get_latest_entity_values("name
"))
        dispatcher.utter_message(text=f"Hello,
{name}!")
        return []
```

9. Now open a new terminal, activate the custom environment where Rasa is installed, and run the **`actions`** endpoint. This will start up the Rasa **`actions`** server:

```
rasa run actions
```

10. Now train your new model in a terminal window different from the one in *step 8*:

```
rasa train
```

11. Now we can test the bot:

```
rasa shell
```

The conversation might go something like this:

```
Your input ->  hello
Hey! How are you?
What's your name?
Your input ->Zhenya
Hello, Zhenya!
Your input ->  where are you located?
Our address is 123 Elf Road North Pole, 88888.
Is there anything else I can help you with?
```

```
Your input ->  what are your hours?
Our hours are Monday to Friday 9 am to 8 pm EST.
Your input ->  bye
Bye
```

How it works...

In *step 1*, we add the **inform** intent to the list of possible intents. In *step 2*, we add an entity that will be recognized, **name**. In *step 3*, we add a new utterance that asks about the user's name. In *step 4*, we add a new action that will be triggered when the user answers the question.

In *step 5*, we add sample user inputs for the **inform** intent. You will notice that the names are listed in parentheses and the entity type, **name**, is listed in square brackets right after the name. The entity name should be the same as the one we listed in *step 2* in the **domain.yml** file.

In *step 6*, we add a piece that asks the user's name and triggers **action_save_name** to each story. The action name should be the same as the one defined in *step 4* in the **domain.yml** file. You will notice that after the **inform** intent, we also list the entities that should be parsed from the user's response, in this case, **name**.

In *step 7*, we uncomment the lines that tell the Rasa chatbot where to look for the **actions** server, which we will start up in *step 9*.

In *step 8*, we define the **ActionSaveName** class, which defines what should happen when the action is triggered. Each action requires a class that is a subclass of the **Action** class and that overrides two methods, **name** and **run**. The **name** method defines the class name, and that name should be the same as the name we defined in *step 4* in the **domain.yml** file. The **run** method defines the action that should be taken once the action is triggered. It is given several arguments, including **tracker** and **dispatcher**. The **tracker** object is the chatbot's *memory* and lets us access parsed entities and other information. The **dispatcher** object generates and sends responses back to the user. We get the user's name from the tracker and send the **Hello, {name}!** response.

In *step 9*, we start the **actions** server, which will be accessed during the bot execution. It should be started in a terminal window different from the one where the chatbot is being run.

In *step 10*, we train the bot model, and in *step 11*, we start it up. In the conversation, we now ask the user their name and greet them accordingly.

See also

In addition to parsing entities out, it is possible to parse out information that fills certain slots, for example, departure and destination cities in a bot that provides flight information. This information can be used for lookup and to provide relevant answers to the user. Please see the Rasa documentation for instructions: **https://rasa.com/docs/rasa/forms**.