

THE RASA MASTERCLASS HANDBOOK

A COMPANION GUIDE TO THE
RASA MASTERCLASS VIDEO SERIES

THE RASA MASTERCLASS HANDBOOK

Foreword

In September 2019, we launched the Rasa Masterclass, a twelve-video tutorial series on building AI assistants with Rasa. The Masterclass provides a complete roadmap for building AI assistants—all the way from installing Rasa for the first time to deploying a finished project on Kubernetes. Along the way, we cover important machine learning concepts and practical coding examples to give you a solid foundation in conversational AI.

The series is hosted by Juste Petraityte, Head of Developer Relations at Rasa. Over the course of the Masterclass, we build an advanced assistant called the Medicare Locator, which uses the medicare.gov API to locate nearby medical facilities.

In addition to learning how to build the Medicare Locator, the Masterclass also covers:

- NLU and dialogue management components and how to configure your NLU pipeline to get the best performance with your dataset
- Adding business logic using forms and integrating with backend systems
- Connecting with messaging channels and deploying the assistant

First published on the blog and now available as an ebook, the Masterclass Handbook is the companion guide to the Masterclass video series. You can follow along with the Handbook as you watch the videos, or return to it later as a quick reference guide. At the end of each chapter, you'll find links to additional resources to help you along your journey.

Whether you're brand new to Rasa or you've built simple AI assistants before, the Rasa Masterclass is a great resource to level up and deepen your expertise as a Rasa developer. We're excited to have you along—we'll learn a lot and apply our new skills to building AI assistants that really help users.

Let's get started!

TABLE OF CONTENTS

Chapter 1: Intro to Conversational AI and Rasa

What are Contextual Assistants?

Exploring Rasa

Getting Started

Chapter 2: Creating NLU Training Data

What We're Building

Conversation Design

Generating NLU Training Data

Chapter 3: NLU Model, Part 1: Pre-configured Pipelines

Key Concepts

Choosing a Pipeline Configuration

Training the Model

Testing the Model

Chapter 4: NLU Model, Part 2: Pipeline Components

Training Pipeline Overview

Training Pipeline Components

SpacyNLP

Tokenizer

Named Entity Recognition

Intent Classification

Featurizers

Intent Classifiers

FAQ



TABLE OF CONTENTS

Chapter 5: Intro to Dialogue Management

Machine Learning vs State Machines
Stories

Adding Stories to Medicare Locator
Training Data Tips

Chapter 6: Domain, Custom Actions, and Slots

Domain File in Rasa
Building a Domain for Medicare Locator
Custom Actions in Rasa

Slots in Rasa
Slot Types
Using Slots in Medicare Locator

Training Your Rasa Assistant

Chapter 7: Dialogue Policies

Policy Configuration in Rasa

Dialogue Policies
Memoization Policy
Mapping Policy
Keras Policy
TED Policy
Form Policy
Fallback Policy



TABLE OF CONTENTS

[Chapter 8: Integrations, Forms, and Fallbacks](#)

[Real-world Dataset for Medicare Locator](#)

[Improving the NLU](#)

[Regex in Entities](#)

[Synonyms](#)

[Implementing a Form Action](#)

[Failing Gracefully in Rasa](#)

[Chapter 9: Improving the Assistant](#)

[What is Rasa X?](#)

[Deploying Rasa X](#)

[Configure the VM](#)

[Install Rasa X](#)

[Connecting the Assistant](#)

[Set up the Action Server](#)

[The Rasa X Dashboard](#)

[Chapter 10: Sharing with Test Users](#)

[Opening Your Assistant to Testers](#)

[Reviewing Test Conversations](#)

[Improving Your Assistant](#)

[Chapter 11: Connecting to Messaging Channels](#)

[Configuring DNS and SSL](#)

[Telegram](#)

[Webchat](#)



TABLE OF CONTENTS

[Chapter 12: Deploying on Kubernetes](#)

[What is Kubernetes?](#)

[Create a New Cluster](#)

[Connect to the Cluster](#)

[Set up the Custom Action Server](#)

[Deploy Rasa X](#)





CHAPTER ONE

Intro to Conversational AI and Rasa

INTRO TO CONVERSATIONAL AI AND RASA



Introduction

Welcome to [episode 1](#) of the Rasa Masterclass. In this episode, we lay the foundation for this video series by introducing you to contextual assistants and Rasa. By the end of this episode, you'll be able to identify what separates contextual assistants from simple FAQ assistants and the components that make up the Rasa stack: Rasa Open Source, which handles NLU and dialogue management, and Rasa X. You'll also install Rasa and create your first starter project.

Chapter 1: Intro

What are contextual assistants?

At Rasa, we use the concept of [5 Levels of Assistants](#) to describe the capabilities of AI assistants and show how the technology has evolved over time.

Briefly, these are the definitions:

- **Level 1: Notification Assistants**
 - Capable of sending simple notifications, like a text message, push notification, or WhatsApp message.
- **Level 2: FAQ Assistants**
 - Can answer simple questions, like FAQs.
 - The most common type of assistant today
 - Often constructed around a set of rules or a state machine.
- **Level 3: Contextual Assistants**
 - Able to understand the context of the conversation, i.e. what the user has said previously and when/where/how they said it.
 - Capable of understanding and responding to different and unexpected inputs
 - Can learn from previous conversations and improve in accuracy over time
 - Buildable today with Rasa
- **Level 4: Personalised Assistants**
 - The next generation of AI assistants, that will get to know you better over time
 - Theoretical only
- **Level 5: Autonomous Organization of Assistants**
 - AI assistants that know every customer personally
 - Capable of running large parts of a company's operations—from lead generation to sales, HR, or finance.
 - Long-term vision for the industry

In the Rasa Masterclass, we'll be focused on building Level 3 assistants, using Rasa's machine learning-based approach, which uses data from real conversations to improve accuracy over time.

Chapter 1: Intro

Exploring Rasa

Rasa has three major components that work together to create contextual assistants:

Note: As of Dec 2019, Rasa Core is now known as dialogue management, to more accurately describe its function. Together, the NLU and dialogue management libraries make up Rasa Open Source.

Rasa NLU

Rasa NLU is like the “ear” of your assistant—it helps your assistant understand what’s being said. Rasa NLU takes user input in the form of unstructured human language and extracts structured data in the form of intents and entities.

- **Intents** are labels that represent the goal, or meaning, of a user’s specific input. For example, the message ‘Hello’ could have the label ‘greet’ because the meaning of this message is a greeting.
- **Entities** are important keywords that an assistant should take note of. For example, the message ‘My name is Juste’ has the name ‘Juste’ in it. An assistant should extract the name and remember it throughout the conversation to keep the interaction natural.
 - Entity extraction is achieved by training a named entity recognition model to identify and extract the entities (in this example, names) for unstructured user messages

Rasa Core

Core is Rasa’s dialogue management component. It decides how an assistant should respond based on 1) the state of the conversation and 2) the context. Rasa Core learns by observing patterns in conversational data between users and an assistant.

Rasa X

Rasa X is a toolset for developers to build, improve and deploy contextual assistants with the Rasa framework. You can use Rasa X to:

- View and annotate conversations
- Get feedback from testers
- Version and manage models

With Rasa X, you can share your assistant with real users and collect the conversations they have with the assistant, allowing you to improve your assistant without interrupting the assistant running in production.

Chapter 1: Intro

Getting Started

The fastest way to begin building an AI assistant with Rasa is on the command line, with a few simple steps:

Install Rasa

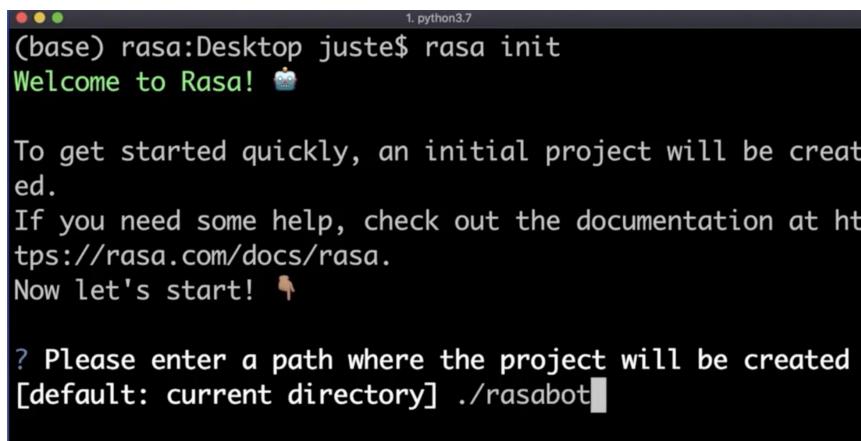
You can install both Rasa (NLU and Core) and Rasa X with a single command:

```
pip3 install rasa-x --extra-index-url https://pypi.rasa.com/simple
```

For detailed step-by-step instructions on installing Rasa, along with system requirements and dependencies, see the [Installation Guide](#).

Create the starter project

Next, we can create the Rasa example starter project. Open a terminal and run the command `rasa init`.



```
1. python3.7
(base) rasa:Desktop juste$ rasa init
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] ./rasabot
```

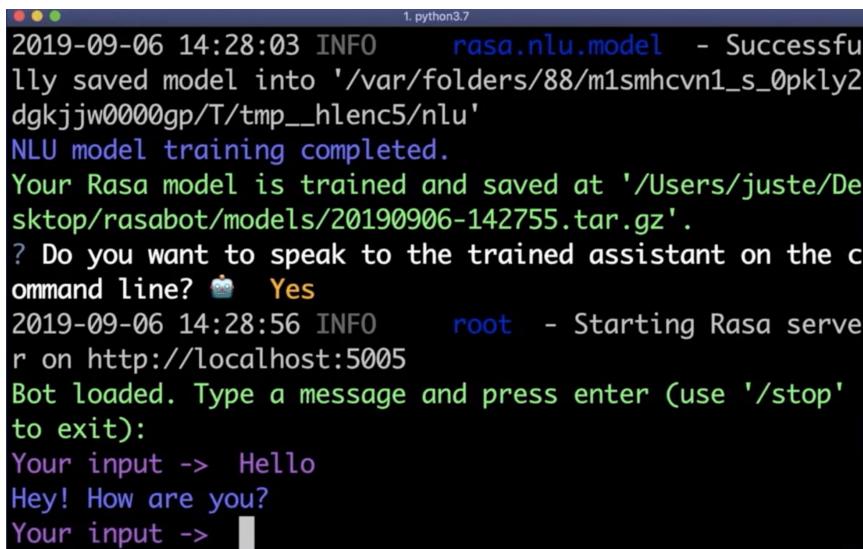
This command creates a new Rasa project in a local directory, which you will specify by providing the directory name. Once the directory is initialised, Rasa will automatically populate it with the project files and example training data, and it will train the NLU and dialogue models.

By default, `rasa init` trains a simple assistant called moodbot which will ask you how you feel, and if you are unhappy it will try to cheer you up by sending you a picture of a cute tiger cub.

Chapter 1: Intro

Interacting with moodbot

You can use moodbot immediately. In your terminal, type `Hi`. The assistant responds with, `Hey, how are you.`



```
1. python3.7
2019-09-06 14:28:03 INFO    rasa.nlu.model - Successfully saved model into '/var/folders/88/m1smhcvn1_s_0pkly2dgkjjw0000gp/T/tmp__hlenc5/nlu'
NLU model training completed.
Your Rasa model is trained and saved at '/Users/juste/Desktop/rasabot/models/20190906-142755.tar.gz'.
? Do you want to speak to the trained assistant on the command line? 🤖 Yes
2019-09-06 14:28:56 INFO    root - Starting Rasa server on http://localhost:5005
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input -> Hello
Hey! How are you?
Your input -> 
```

If you reply that you are a bit sad, moodbot understands that you are unhappy and will send you a picture to cheer you up.

The `rasa init` function is a great way to test how a Rasa-powered assistant works. You can also use moodbot as a boilerplate project for building your own custom assistant.

Next Steps

Now that we've laid the foundation by establishing what contextual assistants are, identifying the components of Rasa, and creating a starter project, we'll build on that knowledge in the next episode: [Creating the NLU Training Data](#).

Chapter 1: Intro

Additional Resources

- [Intro to conversational AI and Rasa: Rasa Masterclass Ep#1](#) (YouTube)
- [Conversational AI: Your Guide to Five Levels of AI Assistants in Enterprise](#) (Rasa Blog)
- [Level 3 Contextual Assistants: Beyond Answering Simple Questions](#) (Rasa Blog)
- [Step-by-step Installation Guide](#) (Rasa docs)
- [Getting Started with Rasa](#) (Rasa docs)



CHAPTER TWO

Creating NLU Training Data

CREATING NLU TRAINING DATA



Introduction

In Episode 2 of the Rasa Masterclass, we focus on generating NLU training data. In the previous episode, we installed Rasa and created moodbot, the Rasa starter project. Now, we'll start to build your assistant's vocabulary.

We'll begin with the basics of conversation design, including techniques you can use to script dialogues between your assistant and your users. Then, we'll learn how to format your training data and define the intents and entities your assistant can understand.

Chapter 2: NLU Data

What We're Building

Before we get into the details of generating NLU training data, let's briefly discuss what we'll be building over the course of the Masterclass series. The Medicare Locator is an AI assistant that uses the Medicare.gov API to locate hospitals, nursing homes, and home health agencies in US cities. We'll be building this assistant from beginning to end throughout the series. When we're finished, the assistant will be able to answer requests like "Give me the address of a hospital in San Francisco."

In this episode, we'll cover the basics of conversational design and generating training data using the Medicare Locator as an example.

Conversation Design

The first step to building a successful contextual assistant is planning the types of conversations your assistant will be able to have, a process known as conversation design. Conversation design should start with three important planning steps to ensure your assistant will meet the needs of your users:

1. Asking who your users are
2. Understanding the assistant's purpose
3. Documenting the most typical conversations users will have with the assistant

Gathering possible questions

Once you've considered who your users are and the intended purpose of your assistant, start assessing what you already know about your potential audience. The goal is to begin compiling a list of common questions your users are likely to ask your assistant.

Do this by:

- Leveraging the knowledge of domain experts
- Looking at common search queries on your website
- Asking your customer service team about their most common requests

Chapter 2: NLU Data

If you don't have access to historical conversation data, you can use the Wizard of Oz approach to gather information. This technique gets its name from a classic scene in the film *The Wizard of Oz*, where the "wizard" is revealed to be a man behind a curtain. When you use the Wizard of Oz approach, you are the man behind the curtain, so to speak. Recruit a volunteer to play the part of your user while you play the role of the bot. By simulating a human-bot chat interaction and recording the conversations, you can put together a realistic estimate of the questions your real users are likely to ask.

Outlining the conversational flow

Conversations tend to follow patterns we can use to identify common intents our assistant should anticipate. For example, many conversations follow this structure:

1. Greeting
2. Assistant states what it is capable of (this is a good practice for a better user experience)
3. User states what they are looking for
4. Assistant asks for more details OR
5. Answers the query if enough information has been provided
6. User says thank you
7. Assistant says "you're welcome" and goodbye

This sequence may seem simple, but conversation design is actually a challenging task. Real-life conversations have more back and forth interactions, and it's difficult to anticipate everything users might ask. When you try to invent a large number of hypothetical conversations to train your model, you risk introducing bias into your data. Because of this, you should only rely on hypothetical conversations in the early stages of development and train your assistant on real conversations as soon as possible.

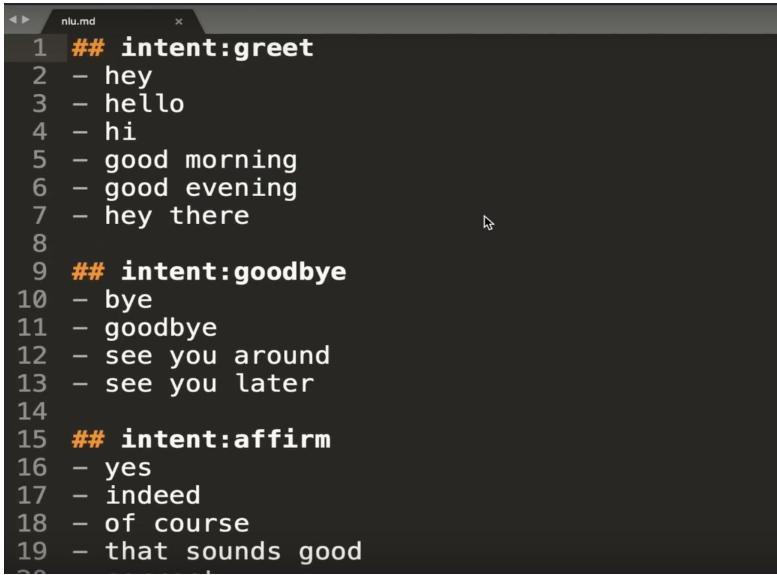
Generating NLU training data for the Medicare Locator

The moodbot starter project contains a Data directory, where you'll find the training data files for NLU and dialogue management models. The Data directory contains two files:

- **nlu.md** - the file containing NLU model training examples. This includes intents, which are user goals, and example utterances that represent those intents. The NLU training data also labels the entities, or important keywords, the assistant should extract from the example utterance.
- **stories.md** - the file containing story data. Stories are example end-to-end conversations.

Chapter 2: NLU Data

For now, we'll concentrate on the nlu.md file:

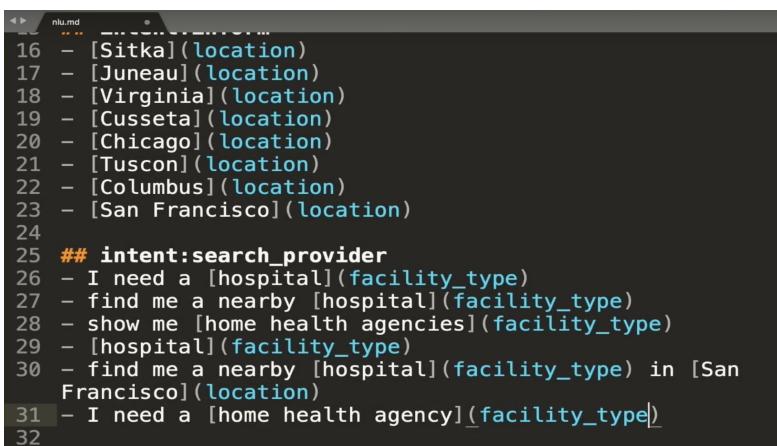


```
1 ## intent:greet
2 - hey
3 - hello
4 - hi
5 - good morning
6 - good evening
7 - hey there
8
9 ## intent:goodbye
10 - bye
11 - goodbye
12 - see you around
13 - see you later
14
15 ## intent:affirm
16 - yes
17 - indeed
18 - of course
19 - that sounds good
20
```

Intents are defined using a double hashtag. Each intent is followed by multiple examples of how a user might express that intent.

Entities are labeled with square brackets and tagged with their type in parentheses.

For example, in the nlu.md file for the Medicare Locator, we've created an intent called search_provider, which represents a user's request to locate a healthcare facility. In each example utterance, we've labeled entities for location and facility type.



```
16 - [Sitka](location)
17 - [Juneau](location)
18 - [Virginia](location)
19 - [Cusseta](location)
20 - [Chicago](location)
21 - [Tuscon](location)
22 - [Columbus](location)
23 - [San Francisco](location)
24
25 ## intent:search_provider
26 - I need a [hospital](facility_type)
27 - find me a nearby [hospital](facility_type)
28 - show me [home health agencies](facility_type)
29 - [hospital](facility_type)
30 - find me a nearby [hospital](facility_type) in [San
  Francisco](location)
31 - I need a [home health agency](facility_type)
32
```

Chapter 2: NLU Data

Each intent your assistant is capable of understanding will need to be defined in the nlu.md file.

There are a few best practices to keep in mind:

- You don't need to write every possible utterance to train an intent, but you should provide 10-15 examples.
- Make sure you provide high-quality data to train your model. Examples should be relevant to the intents, and be sure that there's plenty of diversity in the vocabulary you use in your examples.

Next Steps

Take some time to practice what you've learned by defining a few new intents in your nlu.md file. Then, continue on to [Episode 3](#), where we'll discuss the NLU training pipeline.

Additional Resources

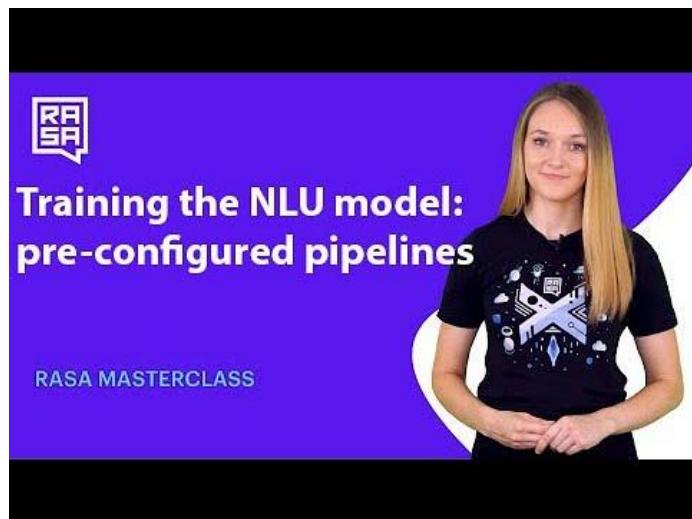
- [Creating the NLU training data - Rasa Masterclass Ep.#2](#) (YouTube)
- [NLU Training Data Format](#) (Rasa docs)
- [Rasa X - NLU Training](#) (Rasa docs)



CHAPTER THREE

NLU MODEL: PART 1 PRE-CONFIGURED PIPELINES

NLU MODEL: PART 1 PRE-CONFIGURED PIPELINES



Introduction

In episode 3 of the Rasa Masterclass, we tackle the first of a two-part module on training NLU models. In this part, we'll focus on choosing a training pipeline configuration, training the model, and testing the model. In our next episode, we'll be back to do a deep dive into each of the components that make up the NLU pipeline.

CHAPTER 3: PRE-CONFIGURED PIPELINES

Key Concepts

Let's begin with a few important definitions.

NLU model - An NLU model is used to extract meaning from text input. In our previous episode, we discussed how to create training data, which contains labeled examples of intents and entities. Training an NLU model on this data allows the model to make predictions about the intents and entities in new user messages, even when the message doesn't match any of the examples the model has seen before.

Training pipeline - NLU models are created by a training pipeline, also referred to as a processing pipeline. A training pipeline is a sequence of processing steps which allow the model to learn the training data's underlying patterns.

In our next episode, we'll dive deeper into the inner workings of the individual pipeline components, but for now, we'll focus on the two pre-configured pipelines included with Rasa out-of-the-box. These pre-configured pipelines are a great fit for the majority of general use cases. If you're looking for information on configuring a custom training pipeline, we'll cover the topic in Episode 4.

Word embeddings - Word embeddings convert words to vectors, or dense numeric representations based on multiple dimensions. Similar words are represented by similar vectors, which allows the technique to capture their meaning. Word embeddings are used by the training pipeline components to make text data understandable to the machine learning model.

CHAPTER 3: PRE-CONFIGURED PIPELINES

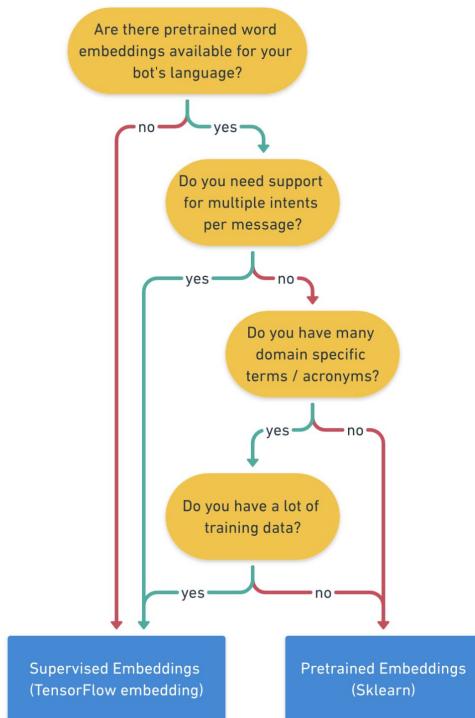
Choosing a Pipeline Configuration

Rasa comes with two default, pre-configured pipelines. Both pipelines are capable of performing intent classification and entity extraction. In this section, we'll compare and contrast the two options to help you choose the right pipeline configuration for your assistant.

1. **Pretrained_embeddings_spacy** - Uses the [spaCy](#) library to load pre-trained language models, which are used to represent each word in the user's input as word embeddings.
 - a. Advantages
 - i. Boosts the accuracy of your models, even if you have very little training data
 - ii. Training doesn't start from scratch, which makes it blazing fast. This encourages short iteration times, so you can rapidly improve your assistant.
 - b. Considerations
 - i. Complete and accurate word embeddings are not available for all languages. They're trained on publicly available datasets, which are mostly in English.
 - ii. Word embeddings don't cover domain-specific words, like product names or acronyms, because they're often trained on generic data, like Wikipedia articles.
2. **Supervised_embeddings** - Unlike pre-trained embeddings, the supervised_embeddings pipeline trains the model from scratch using the data provided in the NLU training data file.
 - a. Advantages
 - i. Can adapt to domain-specific words and messages, because the model is trained on your training data.
 - ii. Language-agnostic. Allows you to build assistants in any language.
 - iii. Supports messages with multiple intents.
 - b. Considerations
 - i. Compared to pre-trained embeddings, you'll need more training examples for your model to start understanding unfamiliar user inputs. The recommended number of examples is 1000 or more.

Generally speaking, the pretrained_embeddings_spacy pipeline is the best choice when you don't have a lot of training data and your assistant will be fairly simple. The supervised_embeddings pipeline is the best choice when your assistant will be more complex, especially if you need to support non-English languages. This decision tree illustrates the factors you will want to consider when deciding which of these two pre-configured pipelines is right for your project:

CHAPTER 3: PRE-CONFIGURED PIPELINES



Training the Model

After you've created your training data (see [Episode 2](#) for a refresher on this topic), you are ready to configure your pipeline, which will train a model on that data. Your assistant's processing pipeline is defined in the `config.yml` file, which is automatically generated when you create a starter project using the `rasa init` command.

This example shows how to configure the `supervised_embeddings` pipeline, by defining the language indicator and the pipeline name:

```
language: "en"  
pipeline: "supervised_embeddings"
```

To train an NLU model using the `supervised_embeddings` pipeline, define it in your `config.yml` file and then run the Rasa CLI command `rasa train nlu`. This command will train the model on your training data and save it in a directory called `models`.

CHAPTER 3: PRE-CONFIGURED PIPELINES

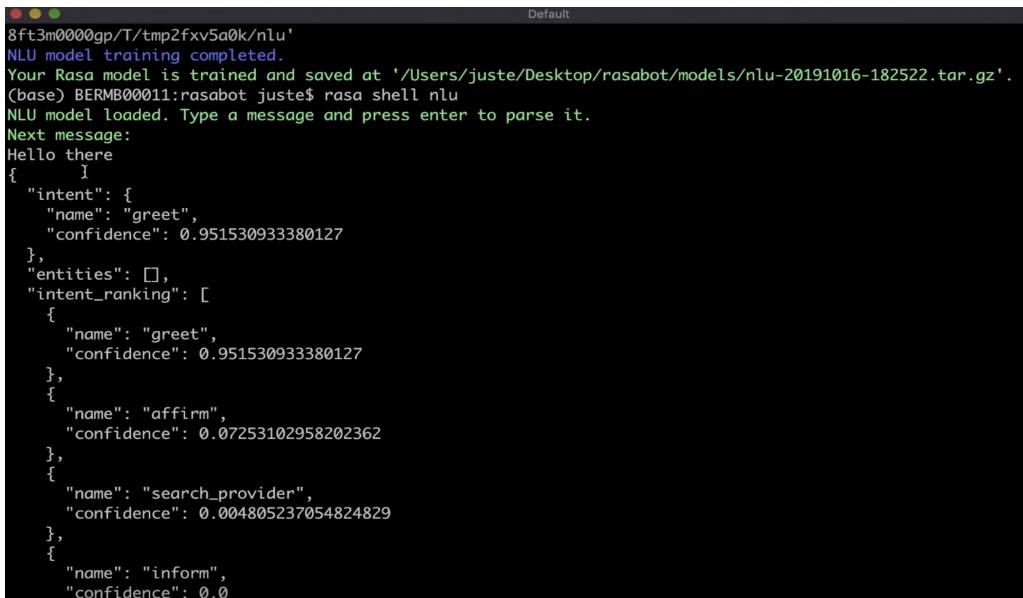
To change the pipeline configuration to pretrained_embeddings_spacy, edit the language parameter in config.yml to match the appropriate spaCy language model and update the pipeline name. You can now retrain the model using the `rasa train nlu` command.

Testing the Model

Test the newly trained model by running the Rasa CLI command, `rasa shell nlu`. This loads the most recently trained NLU model and allows you to test its performance by conversing with the assistant on the command line.

While in test mode, type a message in your terminal, for example, ‘Hello there.’ Rasa CLI outputs a JSON object containing several useful pieces of data:

- The intent the model thinks is the most likely match for the message.
 - For example: {“name: greet”, “confidence: 0.95347273804”}. This means the model is 95% certain “Hello there” is a greeting.
- A list of extracted entities, if there are any.
- A list of intent_rankings. These results show the intent classification for all of the other intents defined in the training data. The intents are ranked according to the intent match probability predictions generated by the model



The screenshot shows a terminal window with the following text:

```
8ft3m0000gp/T/tmp2fxv5a0k/nlu
NLU model training completed.
Your Rasa model is trained and saved at '/Users/juste/Desktop/rasabot/models/nlu-20191016-182522.tar.gz'.
(base) BERMB00011:rasabot juste$ rasa shell nlu
NLU model loaded. Type a message and press enter to parse it.
Next message:
Hello there
{
  "intent": {
    "name": "greet",
    "confidence": 0.951530933380127
  },
  "entities": [],
  "intent_ranking": [
    {
      "name": "greet",
      "confidence": 0.951530933380127
    },
    {
      "name": "affirm",
      "confidence": 0.07253102958202362
    },
    {
      "name": "search_provider",
      "confidence": 0.004805237054824829
    },
    {
      "name": "inform",
      "confidence": 0.0
    }
  ]
}
```

CHAPTER 3: PRE-CONFIGURED PIPELINES

You can use this output to compare the performance of models generated by different pipeline configurations.

Next Steps

Pre-configured pipelines are a great way to get started quickly, but as your project grows in complexity, you will likely want to customize your model. Similarly, as your knowledge and comfort level increases, it's important to understand how the components of the processing pipeline work under the hood. This deeper understanding will help you diagnose why your models behave a certain way and optimize the performance of your training data.

Continue on to our next episode, where we'll explore these topics in part 2 of our module on NLU model training: [Training the NLU models: understanding pipeline components \(Rasa Masterclass Ep.#4\)](#).

Additional Resources

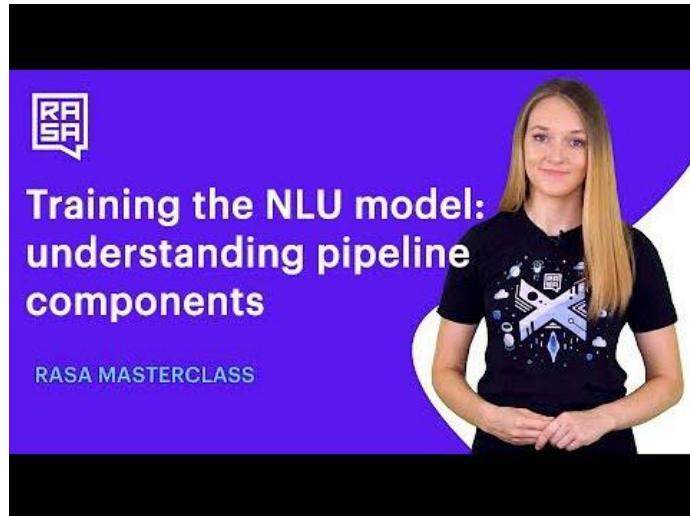
- [Training the NLU model: pre-configured pipelines - Rasa Masterclass ep.#3](#) (YouTube)
- [Choosing a Pipeline](#) (Rasa docs)
- [Supervised Word Vectors from Scratch in Rasa NLU](#) (Rasa blog)
- [Spacy 101](#) (Spacy docs)



CHAPTER FOUR

NLU MODEL: PART 2 PIPELINE COMPONENTS

NLU MODEL: PART 2 PIPELINE COMPONENTS



Introduction

[Episode 4](#) of the Rasa Masterclass is the second of a two-part module on training NLU models. This episode builds upon the material we covered previously, so if you're just joining, head back and watch [Episode 3](#) before proceeding.

Let's do a quick recap of the most important concepts covered in Episode 3:

- **NLU models** accept user messages as input and output predictions about the intents and entities contained in those messages.
- A **training pipeline** trains a new NLU model using a sequence of processing steps that allow the model to learn the training data's underlying patterns.

•

CHAPTER 4: PIPELINE COMPONENTS

- Rasa comes with two [pre-configured pipelines](#):
 - a. **Pretrained_embeddings_spacy** - Uses the spaCy pre-trained language model. Pre-trained models allow you to supply fewer training examples and get started quickly; however, they're trained primarily on general-purpose English data sets, so support for domain-specific terms and non-English languages is limited.
 - b. **Supervised_embeddings** - Trains the model from scratch using the data provided in the NLU training data file. Supervised training supports any language that can be tokenized and can be trained to understand domain-specific terms, but a greater number of training examples is required.
- In a Rasa assistant, the training pipeline is defined in the config.yml file:

```
language: "en"
pipeline: "pretrained_embeddings_spacy"
```

In addition to defining which pipeline you want to use, you can also define which individual pipeline *components* you want to use, to completely customize your NLU model. In Episode 4, we'll examine what each component does and what's happening under the hood when a model is trained.

Training Pipeline Overview

Before getting into the details of individual pipeline components, it's helpful to step back and take a birds-eye view of the process.

As mentioned earlier, a training pipeline consists of a sequence of steps that train a model using NLU data. Each pipeline step executes one after the other, and the order of the steps matters. Some steps produce output that a later step needs to accept as input. Imagine an assembly line in a factory: the worker at the end of the line can't attach the final piece until other workers have attached their pieces. So the pipeline doesn't just define which components should be present, but also the order in which they should be arranged.

No matter which pipeline you choose, it will follow the same basic sequence. We'll outline the process here and then describe each step in greater detail in the Components section.

1. Load pre-trained language model (optional). Only needed if you're using a pre-trained model like [spaCy](#).
2. Tokenize the data. Splits the training data text into individual words, or tokens.
3. Named Entity Recognition. Teaches the model to recognize which words in a message are entities and what type of entity they are.

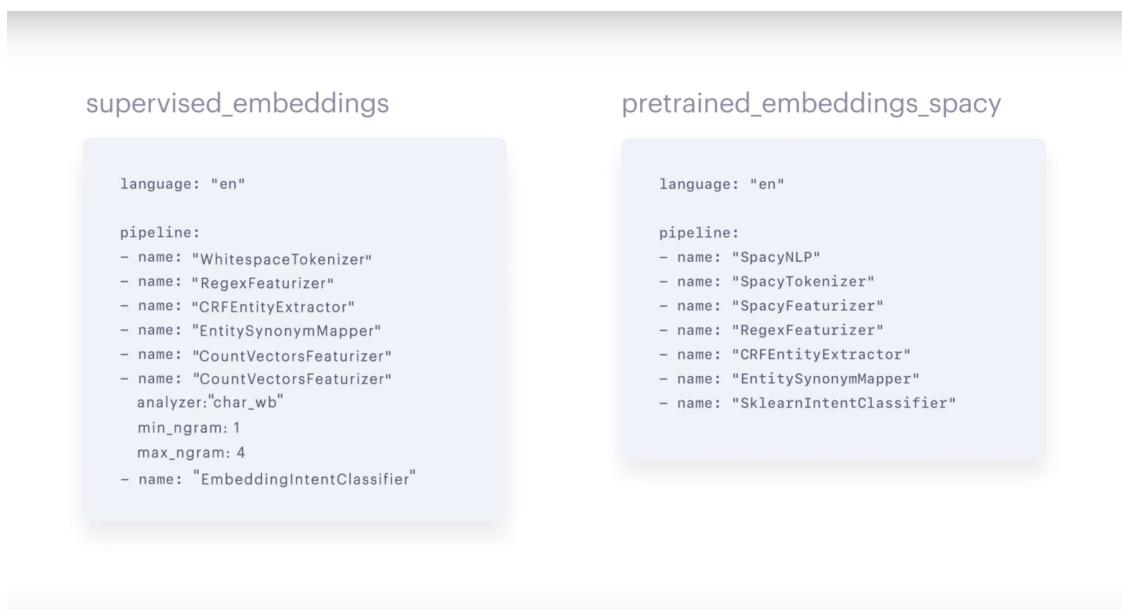
CHAPTER 4: PIPELINE COMPONENTS

4. Featurization. Converts tokens to vectors, or dense numeric representations of words. This step can be performed before or after Named Entity Recognition, but must come after tokenization and before Intent Classification.
5. Intent Classification. Trains the model to make a prediction about the most likely meaning behind a user's message

After a model has been trained using this series of components, it will be able to accept raw text data and make a prediction about which intents and entities the text contains.

Training Pipeline Components

So far, we've talked about two processing pipelines: `supervised_embeddings` and `pretrained_embeddings_spacy`. Both consist of components that are each responsible for a different task. Let's examine each component in greater detail.



SpacyNLP

The `pretrained_embeddings_spacy` pipeline uses the [SpacyNLP](#) component to load the Spacy language model so it can be used by subsequent processing steps. You only need to include this component in pipelines that use spaCy for pre-trained embeddings, and it needs to be placed at the very beginning of the pipeline.

CHAPTER 4: PIPELINE COMPONENTS

Tokenizer

Tokenizers take a stream of text and split it into smaller chunks, or tokens; usually individual words. The tokenizer should be one of the first steps in the processing pipeline because it prepares text data to be used in subsequent steps. All training pipelines need to include a tokenizer, and there are several you can choose from:

WhitespaceTokenizer - The way a [whitespace tokenizer](#) works is very simple: it looks for whitespace in a stream of text and uses it as a delimiter to separate each token, or word. A whitespace tokenizer is the default tokenizer used by the supervised_embeddings pipeline, and it's a good choice if you don't plan on using pre-trained embeddings.

Jieba - Whitespace works well for English and many other languages, but you may need to support languages that require more specific tokenization rules. In that case, you'll want to reach for a language-specific tokenizer, like [Jieba](#) for the Chinese language.

SpacyTokenizer - Pipelines that use spaCy come bundled with the [SpacyTokenizer](#), which segments text into words and punctuation according to rules specific to each language. This is a good option if you're using pre-trained embeddings.

Tokenizer	
Supervised embeddings	Whitespace Jieba (Chinese)
Pre-trained embeddings	SpacyTokenizer

Named Entity Recognition (NER)

NLU models use named entity recognition components to extract entities from user messages. For example, if a user says “What’s the best coffee shop in San Francisco?” the model should extract the entities ‘coffee shop’ and ‘San Francisco’, and identify them as a type of business and a location. There are a few named entity recognition components you can choose from when assembling your training pipeline:

CHAPTER 4: PIPELINE COMPONENTS

CRFEntityExtractor - [CRFEntityExtractor](#) works by building a model called a Conditional Random Field. This method identifies the entities in a sentence by observing the text features of a target word as well as the words surrounding it in the sentence. Those features can include the prefix or suffix of the target word, capitalization, whether the word contains numeric digits, etc. You can also use part of speech tagging with CRFEntityExtractor, but it requires installing spaCy. Part of speech tagging looks at a word's definition and context to determine its grammatical part of speech, e.g. noun, adverb, adjective, etc.

Unlike tokenizers, whose output is fed into subsequent pipeline components, the output produced by CRFEntityExtractor and other named entity recognition components is actually expressed in the final output of the NLU model. It outputs which words in a sentence are entities, what kind of entities they are, and how confident the model was in making the prediction.

SpacyEntityExtractor - If you're using pre-trained word embeddings, you have the option to use [SpacyEntityExtractor](#) for named entity recognition. Even when trained on small data sets, SpacyEntityExtractor can leverage part of speech tagging and other features to locate the entities in your training examples.

DucklingHttpExtractor - Some types of entities follow certain patterns, like dates. You can use specialized NER components to extract these types of structured entities. [DucklingHttpExtractor](#) recognizes dates, numbers, distances and data types.

Regex_featurizer - The [regex_featurizer](#) component can be added before CRFEntityExtractor to assist with entity extraction when you're using regular expressions and/or lookup tables. Regular expressions match certain hardcoded patterns, like a 10-digit phone number or an email address. Lookup tables provide a predefined range of values for an entity. They're useful if your entity type has a finite number of possible values. For example, there are 195 possible values for the entity type 'country,' which could all be listed in a lookup table.

Named Entity Recognition	
Supervised embeddings	CRFEntityExtractor DucklingHttpExtractor Regex_featurizer
Pre-trained embeddings	SpacyEntityExtractor

CHAPTER 4: PIPELINE COMPONENTS

Intent Classification

There are two types of components that work together to classify intents: featurizers and intent classification models.

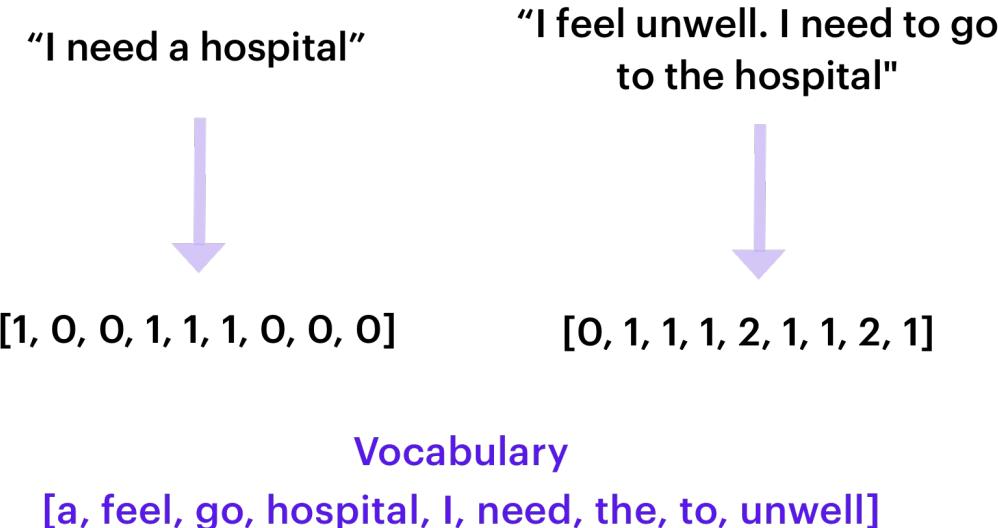
Featurizers take tokens, or individual words, and encode them as vectors, which are numeric representations of words based on multiple attributes. The intent classification model takes the output of the featurizer and uses it to make a prediction about which intent matches the user's message. The output of the intent classification model is expressed in the final output of the NLU model as a list of intent predictions, from the top prediction down to a ranked list of the intents that didn't "win."

```
{
  "intent": {"name": "greet", "confidence": 0.8343},
  "intent_ranking": [
    {
      "confidence": 0.385910906220309,
      "name": "goodbye"
    },
    {
      "confidence": 0.28161531595656784,
      "name": "restaurant_search"
    }
  ]
}
```

Featurizers

CountVectorsFeaturizer - This featurizer creates a bag-of-words representation of a user's message using sklearn's [CountVectorizer](#). The bag-of-words model disregards the order of words in a body of text and instead focuses on the number of times words appear in the text. So the [CountVectorsFeaturizer](#) counts how often certain words from your training data appear in a message and provides that as input for the intent classifier.

CHAPTER 4: PIPELINE COMPONENTS



CountVectorsFeaturizer can be configured to use either word or character n-grams, which is defined using the analyzer config parameter. An n-gram is a sequence of n items in text data, where n represents the linguistic units used to split the data, e.g. by characters, syllables, or words.

By default, the analyzer is set to word n-grams, so word token counts are used as features. If you want to use character n-grams, set the analyzer to char or char_wb. You can also use character n-gram counts by changing the analyzer property of the intent_featurizer_count_vectors component to char. This makes the intent classification more resilient to typos, but also increases the training time.

```
-   name: "CountVectorsFeaturizer"  
     analyzer:"char_web"  
     min_ngram: 1  
     max_ngram: 4
```

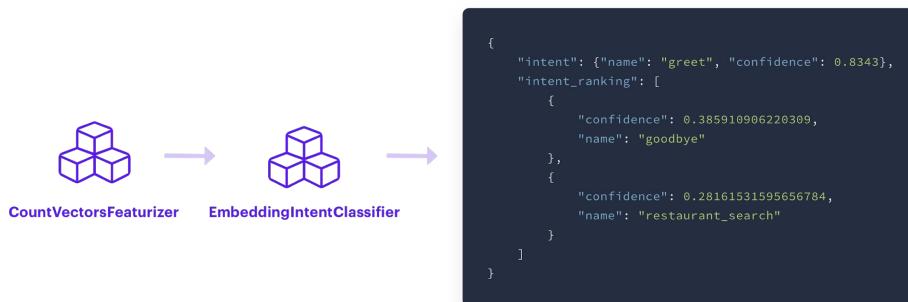
SpacyFeaturizer - If you’re using pre-trained embeddings, [SpacyFeaturizer](#) is the featurizer component you’ll likely want to use. It returns spaCy word vectors for each token, which is then passed to the SklearnIntent Classifier for intent classification.

CHAPTER 4: PIPELINE COMPONENTS

Intent Classifiers

EmbeddingIntentClassifier - If you're using the CountVectorsFeaturizer in your pipeline, we recommend using the [EmbeddingIntentClassifier](#) component for intent classification. The features extracted by the CountVectorsFeaturizer are transferred to the EmbeddingIntentClassifier to produce intent predictions.

The EmbeddingIntentClassifier works by feeding user message inputs and intent labels from training data into two separate neural networks which each terminate in an embedding layer. The cosine similarities between the embedded message inputs and the embedded intent labels are calculated, and supervised embeddings are trained by maximizing the similarities with the target label and minimizing similarities with incorrect ones. The results are intent predictions that are expressed in the final output of the NLU model.



SklearnIntentClassifier - When using pre-trained word embeddings, you should use the [SklearnIntentClassifier](#) component for intent classification. This component uses the features extracted by the SpacyFeaturizer as well as pre-trained word embeddings to train a model called a [Support Vector Machine](#) (SVM). The SVM model predicts the intent of user input based on observed text features. The output is an object showing the top ranked intent and an array listing the rankings of other possible intents.

CHAPTER 4: PIPELINE COMPONENTS

Intent Classification		
Pipeline	Featurizer	Intent Classifier
Supervised embeddings	CountVectorsFeaturizer	EmbeddingIntentClassifier
Pre-trained embeddings	SpacyFeaturizer	SklearnIntent Classifier

FAQ

Now that we've discussed the components that make up the NLU training pipeline, let's look at some of the most common questions developers have about training NLU models.

Q. Does the order of the components in the pipeline matter?

A. The short answer: yes! Some components need the output from a previous component in order to do their jobs. As a rule of thumb, your tokenizer should be at the beginning of the pipeline, and the featurizer should come before the intent classifier.

Q. Should I worry about class imbalance in my NLU training data?

A. Class imbalance is when some intents in the training data file have many more examples than others. And yes—this can affect the performance of your model. To mitigate this problem, Rasa's supervised_embeddings pipeline uses a balanced batching strategy. This algorithm distributes classes across batches to balance the data set. To prevent oversampling rare classes and undersampling frequent ones, it keeps the number of examples per batch roughly proportional to the relative number of examples in the overall data set.

Q. Does the punctuation in my training examples matter?

A. Punctuation is not extracted as tokens, so it's not expressed in the features used to train the models. That's why punctuation in your training examples should not affect the intent classification and entity extraction results.

CHAPTER 4: PIPELINE COMPONENTS

Q. Are intent classification and entity extraction case sensitive?

A. It depends on the task. Named Entity Recognition does observe whether tokens are upper- or lowercase. Case sensitivity also affects the results of entity extraction models. CountVectorsFeaturizer, however, converts characters to lowercase by default. For that reason, upper- or lowercase words don't really affect the performance of the intent classification model, but you can customize the model parameters if needed.

Q. Some of the intents in my training data are pretty similar. What should I do?

A. When the intents in your training data start to seem very similar, it's a good idea to evaluate whether the intents can be combined into one. For example, imagine a scenario where a user provides their name or a date. Intuitively you might create a provide_name intent for the message "It is Sara," and a provide_date intent for the message "It is on Monday." However, from an NLU perspective, these messages are very similar except for their entities. For this reason it would be better to create an intent called *inform* which unifies provide_name and provide_date. Later on, in your dialogue management training data, you can define different story paths depending on which entity Rasa NLU extracted.

Q. What if I want to extract entities from one-word inputs?

A. Extracting entities from one-word user inputs is still quite challenging. The best technique is to create a specific intent, for example *inform*, which would contain examples of how users provide information, even if those inputs consist of one word. You should label the entities in those examples as you would with any other example, and use them to train intent classification and entity extraction models.

Q. Can I specify more than one intent classification model in my pipeline?

A. Technically yes, but there is no real benefit. The predictions of the last specified intent classification model will always be what's expressed in the output.

Q. How do I deal with typos in user inputs?

A. Typos in user messages are unavoidable, but there are a few things you can do to address the problem. One solution is to implement a custom spell checker and add it to your pipeline configuration. This is a nice way to fix typos in extracted entities. Another thing you can do is to add some examples with typos to your training data for your models to pick up.

CHAPTER 4: PIPELINE COMPONENTS

Conclusion

Choosing the components in a custom pipeline can require experimentation to achieve the best results. But after applying the knowledge gained from this episode, you'll be well on your way to confidently configuring your NLU models.

After you finish this episode of the Rasa Masterclass, keep up the momentum. Watch the next installment in the series: [Episode 5, Intro to dialogue management](#). Then, join us in the [community forum](#) to discuss!

Additional Resources

- [Training the NLU models: understanding pipeline components - Rasa Masterclass Ep.#4](#) (YouTube)
- [Entity Extraction](#) (Rasa docs)
- [NLU Components](#) (Rasa docs)
- [Rasa NLU In Depth: Part 1 - Intent Classification](#) (Rasa Blog)
- [Rasa NLU in Depth: Part 2 - Entity Recognition](#) (Rasa Blog)



CHAPTER FIVE

INTRO TO DIALOGUE MANAGEMENT

INTRO TO DIALOGUE MANAGEMENT



Introduction

In Episode 5 of the Rasa Masterclass, we introduce dialogue management, which is controlled by a component called Rasa core. Dialogue management is the function that controls the next action the assistant takes during a conversation. Based on the intents and entities extracted by Rasa NLU, as well as other context, like the conversation history, Rasa core decides which text response should be sent back to the user or whether to execute custom code, like querying a database.

Previously in the Rasa Masterclass, we covered NLU models, including how to [format NLU training data](#), how to [choose a pipeline configuration and train a model](#), and an [in-depth examination of NLU pipeline components](#). If you're just joining, be sure to catch up on previous episodes before moving on to Episode 5.

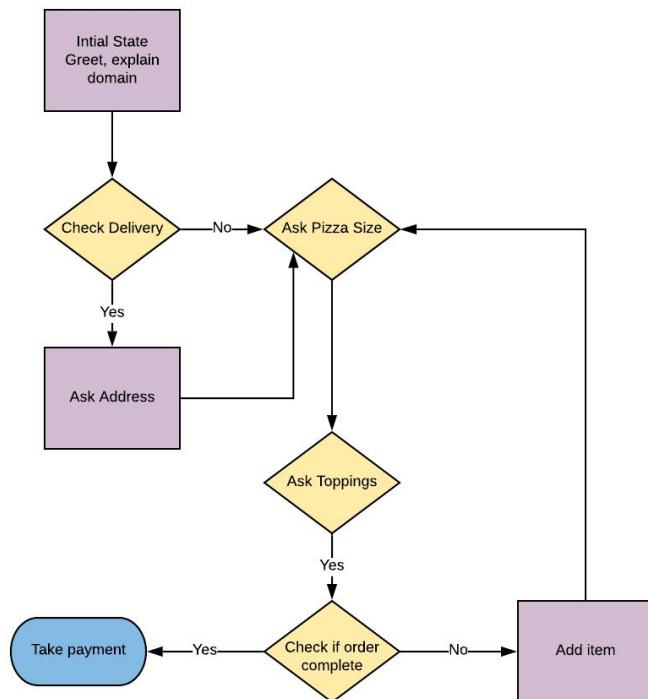
CHAPTER 5: INTRO TO DIALOGUE

Machine Learning: a Better Alternative to State Machines

When building AI assistants, the most common approach to handling dialogue management is to use a set of rules, known as a state machine. Let's examine how a state machine works, and then discuss why Rasa does things differently.

What is a state machine?

A state machine exists in exactly one of a predefined set of states at any given time. The state machine consists of rules that determine the specific set of inputs or circumstances necessary to transition from one state to another. When we apply this concept to a conversation between a bot and a user, we can see how the state machine can control the way the conversation shifts from one phase to the next: 1) in the initial state, the bot greets the user and describes a task it can help with, like ordering a pizza 2) if the user confirms 3) the bot transitions to a new state and asks the user whether they would like a pizza for delivery or pickup.



CHAPTER 5: INTRO TO DIALOGUE

The important thing to keep in mind about state machines is that every state, and the user input needed to transition from state to state, must be explicitly programmed. The assistant won't be able to transition to the next conversational state if it doesn't receive the expected input, which can create a very undesirable user experience. Let's imagine that instead of saying "delivery" in response to the bot's question, the user instead typed "This order is actually for next week." An assistant controlled by a state machine might find it difficult to recover from this unexpected input and transition to the proper state.

You could try to code a rule for every possible turn the conversation might take, but this gets complex very quickly, and you *still* might not anticipate everything users might say.

Dialogue Management with Rasa

Rasa takes a different approach. Instead of a state machine, Rasa core uses machine learning to pick up conversational patterns from example conversations. These example conversations are supplied as training data, much like [the method we used to train the NLU model](#). Based on these patterns, the assistant can generalize, allowing the model to predict the next best action to take in a conversation. This allows the model to provide an appropriate response, even when the conversation doesn't exactly match any of the training examples seen before.

Importantly, this means you don't need to program every possible conversational turn into your assistant up front. You can supply training data containing a few conversation examples when you first build the assistant, and then gather new conversation data directly from real user interactions. Using machine learning, the assistant is able to improve over time, based on the things that users are actually saying.

Stories

In Rasa core, the basic unit of dialogue training data is called a **story**. You can think of a story as a script detailing the back and forth conversation between user and assistant, from beginning to end. Stories are written in a specific format and stored in the stories.md file. When you run the rasa init command to create a new starter project, the stories.md file is automatically created in the data directory, along with a few simple training stories.

CHAPTER 5: INTRO TO DIALOGUE

Let's look at an example:

```
## greet + location/price + cuisine + num people      <!-- name of the story - just for debugging -->
* greet
  - action_ask_howcanhelp
* inform{"location": "rome", "price": "cheap"}  <!-- user utterance, in format intent{entities} -->
  - action_on_it
  - action_ask_cuisine
* inform{"cuisine": "spanish"}
  - action_ask_numpeople      <!-- action that the bot should execute -->
* inform{"people": "six"}
  - action_ack_dosearch
```

Story Name

The beginning of a story is marked with a double hashtag (##), followed by its name. Naming your stories is not required, but it makes debugging much easier. For the same reason, it's a good idea to make story names descriptive so you can see at a glance what the story is about.

Messages from Users

Stories are structured as a series of messages from users, and your assistant's response to those messages. User messages are marked by lines starting with an asterisk (*).

As you can see, the user message is not the actual text of the user message. Instead, the story contains the intent labels and entities extracted by the NLU model.

CHAPTER 5: INTRO TO DIALOGUE

Adding Stories to the Medicare Locator Assistant

If you've been following along with the Masterclass, we've been building a contextual AI assistant that uses the Medicare.gov API to locate medical facilities in the US, like hospitals or nursing homes. For reference, you can find the completed code for the Medicare Locator assistant in the [Rasa Masterclass GitHub repository](#).

In the meantime, let's keep working on building our medicare locator from scratch. Let's update the assistant's stories.md file and add some training data.

Locate the stories.md file in the data directory. Since we used the rasa init command to create the assistant earlier in the tutorial, we should see a few example stories used to train moodbot.

Name	Date Modified	Size	Kind
__init__.py	22. Sep 2019 at 23:12	Zero bytes	Python
__pycache__	3. Oct 2019 at 15:41	--	Folder
actions.py	22. Sep 2019 at 23:12	752 bytes	Python
config.yml	16. Oct 2019 at 18:27	286 bytes	YAML
credentials.yml	22. Sep 2019 at 23:12	658 bytes	YAML
data	3. Oct 2019 at 15:41	--	Folder
nlu.md	16. Oct 2019 at 18:22	1 KB	Markdown
stories.md	22. Sep 2019 at 23:12	355 bytes	Markdown
domain.yml	22. Sep 2019 at 23:12	477 bytes	YAML
endpoints.yml	22. Sep 2019 at 23:12	1 KB	YAML
models	16. Oct 2019 at 18:27	--	Folder

CHAPTER 5: INTRO TO DIALOGUE

We'll add a few new stories to train the Medicare Locator.

Story 1: hospital search happy path

We'll start with a story where the user supplies all of the required information and the conversation goes as expected (what we'd call the "happy path").

If we were to write out the full conversation, it might look something like this:

User: Hello

Medicare Locator: Hi. I am a Medicare Locator. I can help you find hospitals, nursing homes or other medical facilities in your preferred location. How can I help?

User: I need a hospital in San Francisco

Medicare Locator: The address is 1001 Potrero Ave.

User: Thank you.

Medicare Locator: You are very welcome. Goodbye.

Let's translate the conversation above into the training story format.

```
↳ stories.md ●  
data > ↳ stories.md > abc ## hospital search happy path  
1  ## hospital search happy path  
2  * greet  
3  | - utter_how_can_i_help  
4  * search_provider{"facility_type": "hospital", "location": "San Francisco"}  
5  | - action_facility_search  
6  * thanks  
7  | - utter_goodbye  
8
```

First, we'll name the story `hospital search happy path`.

```
## hospital search happy path
```

Because the user will likely start out with a greeting like 'Hello' or 'Hi there,' we start the story with the user intent `greet`, which is an intent we defined when we created the assistant's NLU training data. In response, the assistant responds with an utterance: a hard-coded message that prints out "Hi. I am a Medicare Locator. I can help you find hospitals, nursing homes or other medical facilities in your preferred location. How can I help?"

CHAPTER 5: INTRO TO DIALOGUE

```
## hospital search happy path
* greet
  - utter_how_can_i_help
```

The user responds with “I need a hospital in San Francisco,” which matches the `search_provider` intent we defined in the NLU training data. The user provides the two entities needed to conduct the search: the `facility_type` and the location. Next, the assistant responds by executing a custom action, in this case, making an API call to locate hospitals in San Francisco.

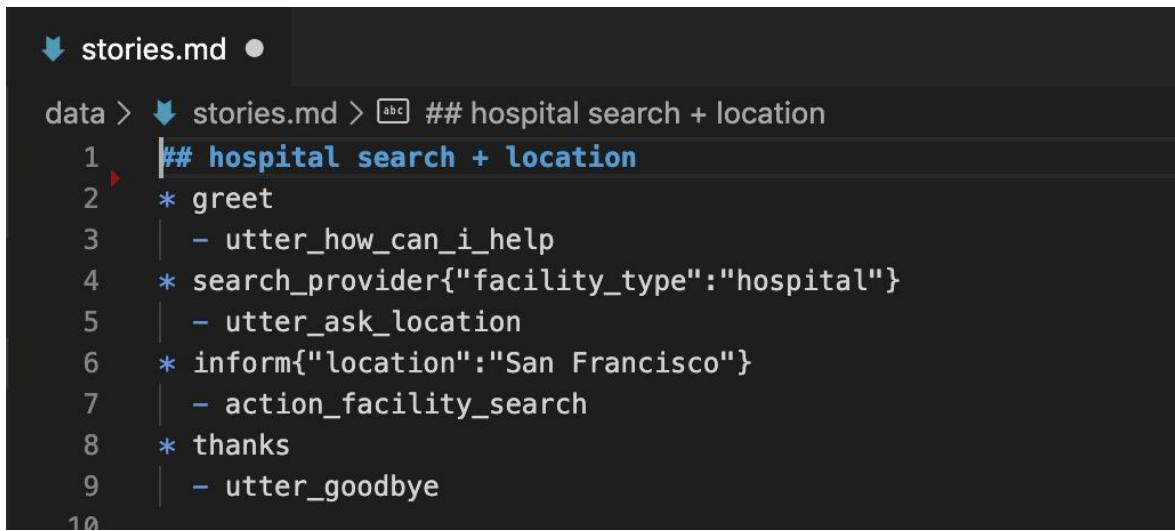
```
* search_provider{"facility_type": "hospital", "location": "San
Francisco"}
  - action_facility_search
```

We end the conversation with the user thanking the assistant and the assistant uttering goodbye.

```
* thanks
  - utter_goodbye
```

Story 2: hospital search + location

In order for the assistant to learn, we need to supply more than one training story. So let’s add another conversation to our training data.



```
stories.md ●
data > stories.md > ## hospital search + location
1  ## hospital search + location
2  * greet
3  |  - utter_how_can_i_help
4  * search_provider{"facility_type": "hospital"}
5  |  - utter_ask_location
6  * inform{"location": "San Francisco"}
7  |  - action_facility_search
8  * thanks
9  |  - utter_goodbye
10
```

CHAPTER 5: INTRO TO DIALOGUE

```
## hospital search happy path
* greet
  - utter_how_can_i_help
```

The user responds with “I need a hospital in San Francisco,” which matches the `search_provider` intent we defined in the NLU training data. The user provides the two entities needed to conduct the search: the `facility_type` and the location. Next, the assistant responds by executing a custom action, in this case, making an API call to locate hospitals in San Francisco.

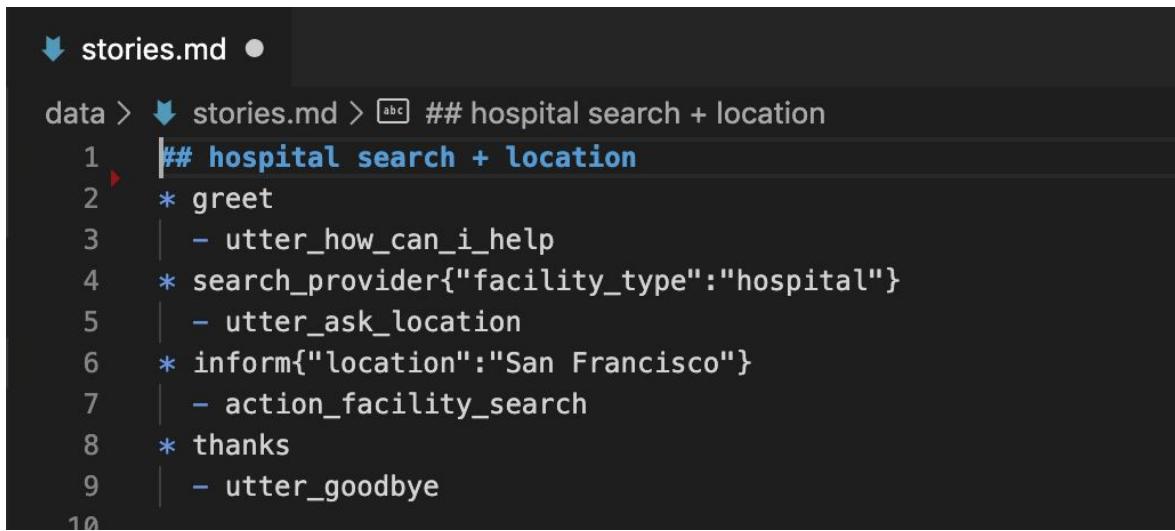
```
* search_provider{"facility_type": "hospital", "location": "San
Francisco"}
  - action_facility_search
```

We end the conversation with the user thanking the assistant and the assistant uttering goodbye.

```
* thanks
  - utter_goodbye
```

Story 2: hospital search + location

In order for the assistant to learn, we need to supply more than one training story. So let’s add another conversation to our training data.



```
stories.md ●
data > stories.md > ## hospital search + location
1  ## hospital search + location
2  * greet
3    - utter_how_can_i_help
4  * search_provider{"facility_type": "hospital"}
5    - utter_ask_location
6  * inform{"location": "San Francisco"}
7    - action_facility_search
8  * thanks
9    - utter_goodbye
10
```

CHAPTER 5: INTRO TO DIALOGUE

In this story, the user doesn't specify their location right away, so the assistant will need to ask for it. We'll call this story `hospital search + location`.

You'll notice this story looks much like the story we wrote for `hospital search happy path`, except this time, the user's `search_provider` message only includes one entity: `facility_type`. Instead of executing the `action_facility_search` custom action, the assistant responds with an utterance prompting the user for their location. Once the location entity has been supplied, the assistant executes the custom action searching the facility, and the conversation ends with a thank you and a goodbye.

Training Data Tips

While there isn't a firm rule for the number of stories you should have in your training data, there are 2 things to keep in mind:

The more training examples you have, the better. Remember, you don't need to provide a training example for every possible conversational turn, but you do need diverse training examples so the model can learn, start to generalize, and handle previously unseen user inputs.

Examples from real user interactions are preferable to training stories you make up yourself. With this in mind, you should focus first on building an assistant that can handle a simple skill and then share the assistant with real users as soon as possible. Data generated by conversations with real users is the best training data you can possibly get.

It can be challenging to improve your assistant with real user input, which is why we created Rasa X. The Rasa Masterclass will cover Rasa X in detail in later episodes. For now, create as many training stories as you need to cover what you want your assistant to be able to do, and know that you'll need to test it with real users to make it better, later.

CHAPTER 5: INTRO TO DIALOGUE

Conclusion

If you're coding along, try adding a few more training story examples to your stories.md file. Start by writing down different conversations you think the assistant might have with users and then convert the dialogues into the Rasa training data format.

So far, we've just scratched the surface of dialogue management. In [Episode 6](#), we'll go deeper, covering domain, custom actions, and slots. And in [Episode 7](#), we'll talk dialogue policies—the machine learning components that determine the behavior of Rasa core. Keep up the progress, and if you get stuck, ask us a question in the [Community forum](#).

Additional Resources

- [Ep #5 - Rasa Masterclass - Intro to dialogue management with Rasa](#) (YouTube)
- [About Rasa Core](#) (Rasa docs)
- [Stories](#) (Rasa docs)
- [Rasa: Open Source Language Understanding and Dialogue Management](#) (arXiv)



CHAPTER SIX

DOMAIN, CUSTOM ACTIONS, AND SLOTS

DOMAIN, CUSTOM ACTIONS, AND SLOTS



Introduction

In Episode 6 of the Rasa Masterclass, we cover essential components in building dialogue management models with Rasa, including domain, custom actions, slots, and more.

This episode builds heavily on Episode 5, in which we covered [dialogue management](#), handled by Rasa core. Dialogue management is the function that controls the next action the assistant takes during a conversation. Based on the intents and entities extracted by Rasa NLU, as well as other context, like the conversation history, Rasa core decides which text response should be sent back to the user or whether to execute custom code, like querying a database.

If you're just joining, be sure to catch up on previous episodes before moving on to Episode 6.

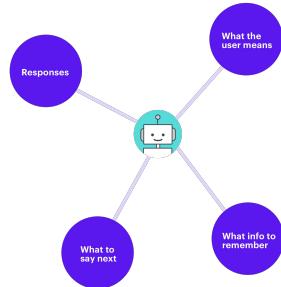
Want to skip straight to the code? Get the [full code for this episode on GitHub](#).

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Domain File in Rasa

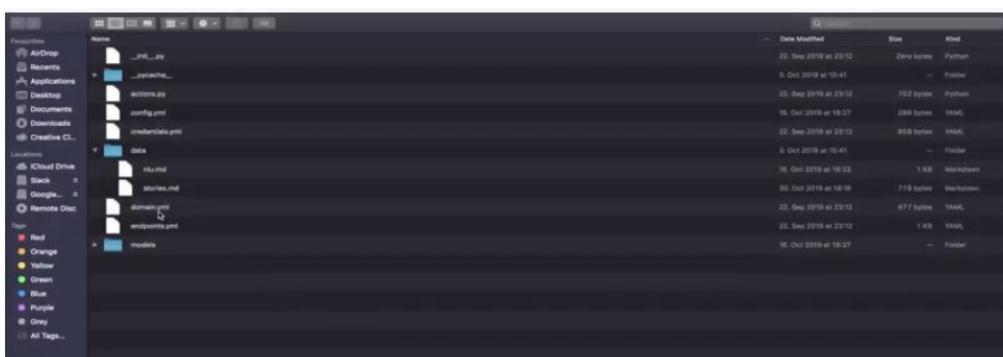
The domain is an essential component of a Rasa dialogue management model. It defines the environment in which the assistant operates, including:

- **What the user means:** specifically, what **intents** and **entities** the model can understand
- **What responses the model can provide:** such as utterances or custom actions
- **What to say next:** what the model should be ready to respond with
- **What info to remember:** what information an assistant should remember and use throughout the conversation



Building a Domain for Medicare Locator

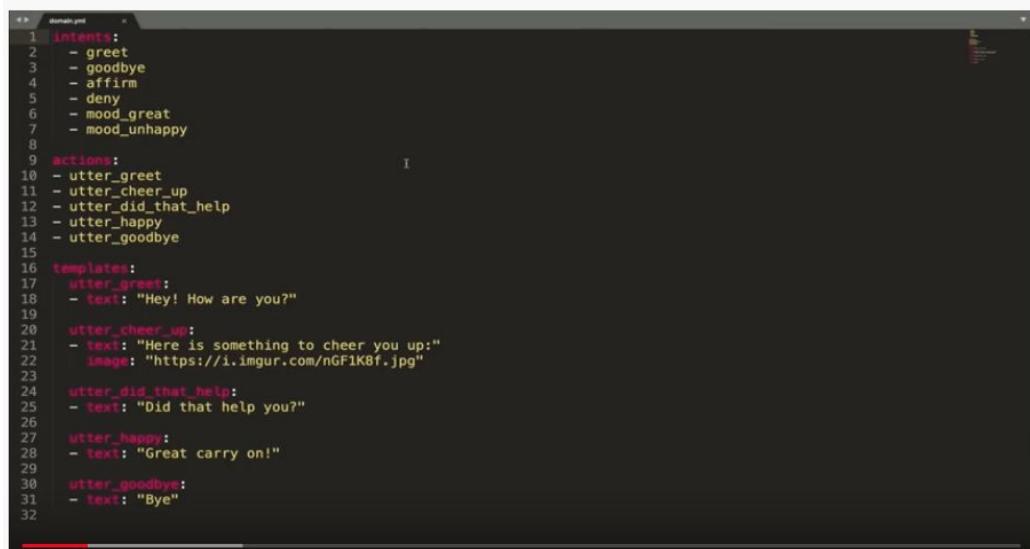
To understand the domain better, let's build out an actual example domain for our medicare locator assistant.



CHAPTER 6: DOMAIN, ACTIONS, SLOTS

The domain of an assistant is usually specified in a domain.yml file of the project directory.

Earlier, we ran the `rasa init` function and a domain was automatically created for our Moodbot assistant. This domain is a nice starting point for creating a custom domain for our medicare locator assistant.



A screenshot of a code editor window showing the `domain.yml` file. The file contains YAML configuration for a Rasa domain. It includes sections for intents, actions, and templates. The `intents` section lists common intents like greet, goodbye, affirm, deny, mood_great, and mood_unhappy. The `actions` section lists various utterances such as `utter_greet`, `utter_cheer_up`, `utter_did_that_help`, `utter_happy`, and `utter_goodbye`. The `templates` section defines responses for these utterances, including text and optional images. For example, the `utter_greet` template has the text "Hey! How are you?" and the `utter_cheer_up` template has the text "Here is something to cheer you up:" and an image URL.

```
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8
9 actions:
10  - utter_greet
11  - utter_cheer_up
12  - utter_did_that_help
13  - utter_happy
14  - utter_goodbye
15
16 templates:
17   utter_greet:
18     - text: "Hey! How are you?"
19
20   utter_cheer_up:
21     - text: "Here is something to cheer you up:"
22     - image: "https://i.imgur.com/nGF1K8f.jpg"
23
24   utter_did_that_help:
25     - text: "Did that help you?"
26
27   utter_happy:
28     - text: "Great carry on!"
29
30   utter_goodbye:
31     - text: "Bye"
32
```

In this Moodbot domain, you can see three sections which are necessary to build an assistant with Rasa: **intents**, **actions** and **templates**. We'll start by discussing these sections, and then add two more sections that are necessary to build many assistants, including our medicare locator: **entities** and **slots**.

Intents

The section labeled **intents** defines a list of intents that the assistant is able to understand. These details come from the NLU model (`nlu.md`). The intents section is where you need to provide the labels of all the intents that you trained your NLU model to understand. In earlier episodes of the Rasa Masterclass, we created new intents for the medicare locator assistant, which you can see below in the `nlu.md` file. Those two intents (`inform` and `search_provider`) need to be added to the domain file for the medicare assistant.

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

```
domain.yml
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - inform
9   - search_provider
10 actions:
11   - utter_greet
12   - utter_cheer_up
13   - utter_did_that_help
14   - utter_happy
15   - utter_goodbye
16 templates:
17   utter_greet:
18     - text: "Hey! How are you?"
19   utter_cheer_up:
20     - text: "Here is something to cheer you up:"
21     - image: "https://i.imgur.com/nGF1K8f.jpg"
22   utter_did_that_help:
23     - text: "Did that help you?"
24   utter_happy:
```

```
stories.md
1 ## intent:greet
2 - hey
3 - hello
4 - hi
5 - good morning
6 - good evening
7 - hey there
8
9 ## intent:goodbye
10 - bye
11 - goodbye
12 - see you around
13 - see you later
14
15 ## intent:inform
16 - [Sitka](location)
17 - [Juneau](location)
18 - [Virginia](location)
19 - [Cusseta](location)
20 - [Chicago](location)
21 - [Tucson](location)
22 - [Columbus](location)
23 - [San Francisco](location)
24
25 ## intent:search_provider
26 - i need a [hospital](facility_type)
27 - find me a nearby [hospital](facility_type)
28 - show me [home health agencies](facility_type)
29 - [hospital](facility_type)
```

Actions

The section called actions should contain the list of all utterances and custom actions an assistant should use to respond to user's inputs. These should come from your stories data in the stories.md file.

```
domain.yml
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - inform
9   - search_provider
10 entities:
11   - location
12   - facility_type
13 actions:
14   - utter_greet
15   - utter_cheer_up
16   - utter_did_that_help
17   - utter_happy
18   - utter_goodbye
19   - utter_how_can_i_help
20   - utter_ask_location
21   - action_facility_search
22   - search_provider
23   - thanks
24 templates:
25   utter_greet:
26     - text: "Hey! How are you?"
27   utter_cheer_up:
28     - text: "Here is something to cheer you up:"
29     - image: "https://i.imgur.com/nGF1K8f.jpg"
```

```
stories.md
1 ## hospital search happy path
2 * greet
3   - utter_greet
4   * search_provider("facility_type":"hospital",
5     "location": "San Francisco")
6   - action_facility_search
7   * thanks
8   - utter_goodbye
9
10 ## hospital search + location
11 * greet
12   - utter_how_can_i_help
13   * search_provider("facility_type":"hospital")
14   - utter_ask_location
15   * inform("location": "San Francisco")
16   < action_facility_search
17   * thanks
18   - utter_goodbye
19
20 ## happy path
21 * greet
22   - utter_greet
23   * mood_great
24   - utter_happy
25
26 ## sad path 1
27 * greet
28   - utter_greet
29   * mood_unhappy
30   - utter_cheer_up
31   - utter_did_that_help
32   * affirm
```

Here we took the actions created earlier for the medicare locator assistant from the stories file and added them to the domain.

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Templates

The final section in the domain is called templates. The templates section is where you can define the specific text responses that your assistant will provide to the user, based on which utterance is predicted by the dialogue management model. Each utterance can have more than one template, that is, more than one specific text response that captures the meaning of the utterance. Utterances can also go beyond simple messages, and can include things like images, buttons, [custom payloads \(for a datepicker, for example\)](#), and more.

Below you can see some templates created for our medicare locator assistant.

```
14
15 actions:
16 - utter_greet
17 - utter_cheer_up
18 - utter_did_that_help
19 - utter_happy
20 - utter_goodbye
21 - utter_how_can_i_help
22 - utter_ask_location
23 - action_facility_search
24
25 templates:
26   utter_greet:
27     - text: "Hey! How are you?"
28
29   utter_cheer_up:
30     - text: "Here is something to cheer you up:"
31     - image: "https://i.imgur.com/nGF1K8f.jpg"
32
33   utter_did_that_help:
34     - text: "Did that help you?"
35
36   utter_happy:
37     - text: "Great carry on!"
38
39   utter_goodbye:
40     - text: "Bye"
41
42   utter_how_can_i_help:
43     - text: "Hello. I am a medicare locator. I can help you find nearest hospital. How can I help?""
44     - text: "Hi. My name is medicare locator. How can I help you today?"
```

Entities

Another important section of the domain file is for **entities**. While the Moobot assistant domain does not have this section, the NLU model for our medicare locator assistant does extract entities, like location and facility type. Since entities influence how an assistant responds to a user's input, entities must be included in the domain file.

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

```
domain.yml
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - inform
9   - search_provider
10
11 entities:
12   - location
13   - facility_type
14
15 actions:
16   - utter_greet
17   - utter_cheer_up
18   - utter_did_that_help
19   - utter_happy
20   - utter_goodbye
21
22 templates:
23   utter_greet:
24     - text: "Hey! How are you?"
25
26 utter_cheer_up:
27   - text: "Here is something to cheer you up!"
```

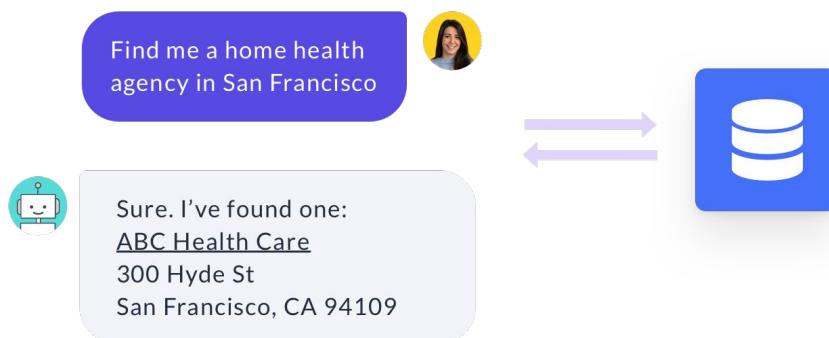
```
nlu.md
1 ## intent:greet
2 - hey
3 - hello
4 - hi
5 - good morning
6 - good evening
7 - hey there
8
9 ## intent:goodbye
10 - bye
11 - goodbye
12 - see you around
13 - see you later
14
15 ## intent:inform
16 - [Sitka](location)
17 - [Juneau](location)
18 - [Virginia](location)
19 - [Cusseta](location)
20 - [Chicago](location)
21 - [Tucson](location)
22 - [Columbus](location)
23 - [San Francisco](location)
24
25 ## intent:search_provider
26 - I need a [hospital](facility_type)
27 - find me a nearby [hospital](facility_type)
```

As you can see, we've created the entities section in the domain file, and listed the entity labels that the medicare locator NLU model (nlu.md) was trained to extract. In our case, we have two entities, location and facility_type.

Custom Actions in Rasa

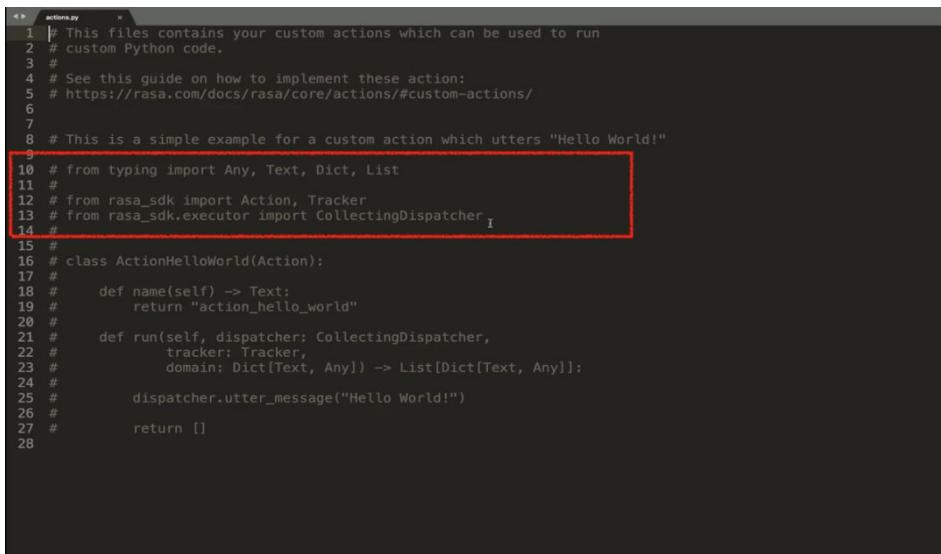
Adding response templates directly to the domain file is the easiest way to define the message an assistant sends the user once a specific utterance is predicted. But there is another way to achieve the same result - by creating custom actions.

Custom actions are response actions which include custom code. That custom code can define anything from a simple text response to a backend integration - an API call, connecting to the database, or anything else your assistant needs to do.



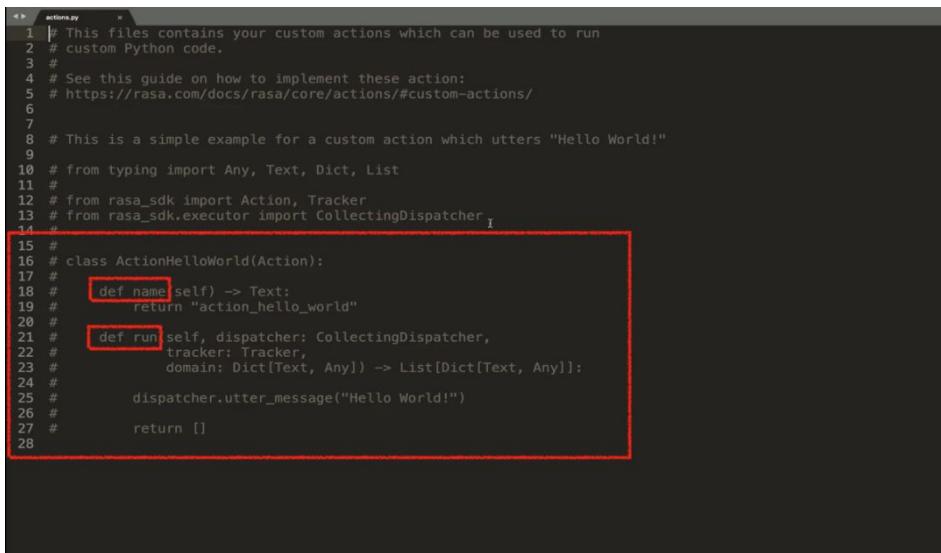
CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Custom actions are defined in a file called [actions.py](#), containing python code, as the file extension suggests. Again, the `rasa init` function created a sample file for us, this time including the code for a simple Hello World custom action, which we will examine to better understand custom actions.



```
actions.py
1 # This files contains your custom actions which can be used to run
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/
6 #
7 #
8 # This is a simple example for a custom action which utters "Hello World!"
9 #
10 # from typing import Any, Text, Dict, List
11 #
12 # from rasa_sdk import Action, Tracker
13 # from rasa_sdk.executor import CollectingDispatcher
14 #
15 #
16 # class ActionHelloWorld(Action):
17 #
18 #     def name(self) -> Text:
19 #         return "action_hello_world"
20 #
21 #     def run(self, dispatcher: CollectingDispatcher,
22 #            tracker: Tracker,
23 #            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
24 #
25 #         dispatcher.utter_message("Hello World!")
26 #
27 #         return []
28
```

The import statement above imports modules (like `rasa sdk`) that are necessary to ensure that the custom action server and the server running your assistant can exchange information.



```
actions.py
1 # This files contains your custom actions which can be used to run
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/
6 #
7 #
8 # This is a simple example for a custom action which utters "Hello World!"
9 #
10 # from typing import Any, Text, Dict, List
11 #
12 # from rasa_sdk import Action, Tracker
13 # from rasa_sdk.executor import CollectingDispatcher
14 #
15 #
16 # class ActionHelloWorld(Action):
17 #
18 #     def name(self) -> Text:
19 #         return "action_hello_world"
20 #
21 #     def run(self, dispatcher: CollectingDispatcher,
22 #            tracker: Tracker,
23 #            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
24 #
25 #         dispatcher.utter_message("Hello World!")
26 #
27 #         return []
28
```

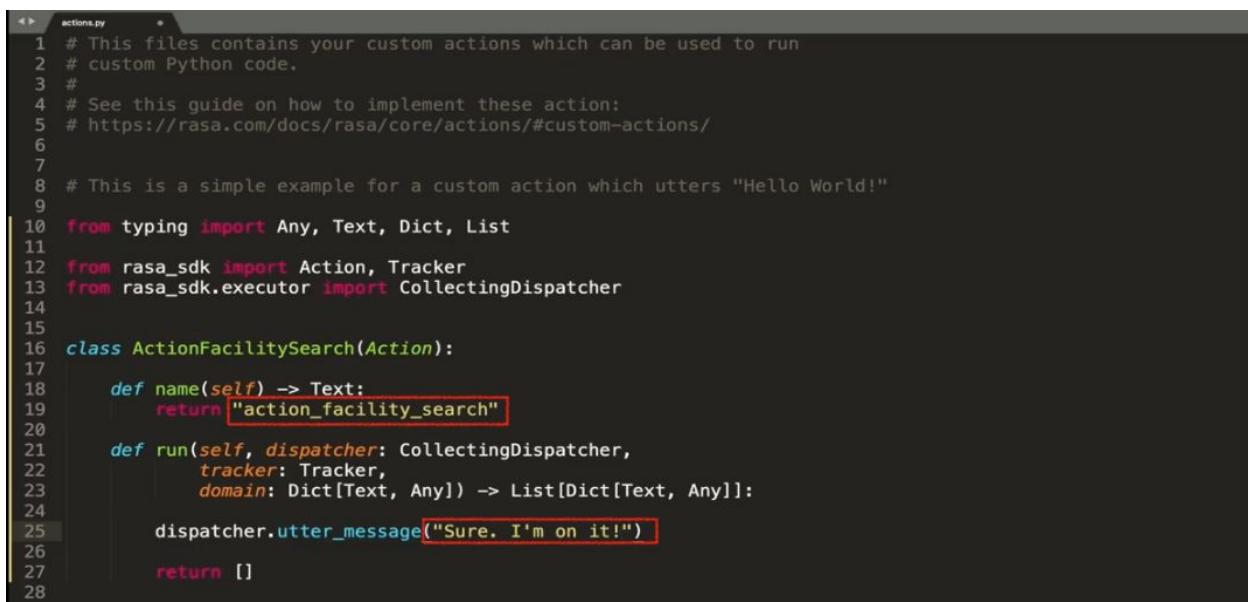
CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Next, we have the class of the custom action. The class consists of two functions - name and run. The function name in the class must match the name of the custom action in your training stories (in stories.md). For example, when the custom action `action_hello_world` is included in a story, Rasa knows to run the code defined in the custom action class named `action_hello_world`.

The `run` function within the class contains the code to be executed, once the custom action is predicted. The run function is where you can define what the custom action actually does. Note the `tracker` and `dispatcher` elements, which are very useful and important pieces of the run function:

- `tracker` keeps track of what happens at each point within a dialogue - what intents were predicted, which entities were extracted, as well as other information
- `dispatcher` is the element that sends the response back to the user.

Updating this actions file for our medicare locator would look like this. We created a class called `ActionFacilitySearch` that, when the action called `action_hello_world` is predicted, the assistant responds, "Sure, I'm on it!"



```
actions.py
1 # This files contains your custom actions which can be used to run
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/
6
7
8 # This is a simple example for a custom action which utters "Hello World!"
9
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14
15
16 class ActionFacilitySearch(Action):
17
18     def name(self) -> Text:
19         return "action_facility_search"
20
21     def run(self, dispatcher: CollectingDispatcher,
22             tracker: Tracker,
23             domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
24
25         dispatcher.utter_message("Sure. I'm on it!")
26
27         return []
28
```

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

An important thing to remember about custom actions is that the names of these actions must match the actions included in the domain file.

```
actions.py
1 # This file contains your custom actions which can
2 # be implemented in Python
3 #
4 # See this guide on how to implement these actions:
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions
6
7 # This is a simple example for a custom action which
8 # returns a message
9
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14
15
16 class ActionFacilitySearch(Action):
17
18     def name(self) -> Text:
19         return "action_facility_search"
20
21     def run(self, dispatcher: CollectingDispatcher,
22            tracker: Tracker,
23            domain: Dict[Text, Any]) -> List[Dict[Te
24
25         dispatcher.utter_message("Sure. I'm on it!")
26
27         return []
28
```

```
domain.yml
1 intents:
2     - greet
3     - goodbye
4     - affirm
5     - deny
6     - mood_great
7     - mood_unhappy
8     - inform
9     - search_provider
10
11 entities:
12     - location
13     - facility_type
14
15 actions:
16     - utter_greet
17     - utter_cheer_up
18     - utter_did_that_help
19     - utter_happy
20     - utter_goodbye
21     - utter_how_can_i_help
22     - utter_ask_location
23     - action_facility_search
24
25 templates:
26     utter_greet:
27         - text: "Hey! How are you?"
28
29     utter_cheer_up:
30         - text: "Here is something to cheer you up:"
31         - image: "https://i.imgur.com/nGF1K8f.jpg"
32
```

As you can see, the domain, NLU training data, and stories data files are very closely connected. There is no specific rule for which one should come first, but you will notice that changes in one file will result in changes in other files. Developing an assistant with Rasa usually takes a number of iterations; that's why you should expect to constantly go back and forth between these files and make modifications as you go.

Slots in Rasa

Another important element of the domain file - very important for dialogue management in Rasa - is slots. Slots function as the assistant's memory, and are used by your assistant to remember important details throughout the conversation and apply those details in context to drive the conversation. Slots act as a key-value pair to store information critical to the conversation with the user. This information can be provided by the user (e.g., entity values extracted by the NLU model) or gathered from outside the conversation (e.g., results extracted from the external database).

CHAPTER 6: DOMAIN, ACTIONS, SLOTS



Earlier in the Masterclass, we covered situations in which details provided by the user influence the path of the conversation. For our medicare locator assistant, if the user wants help finding a medicare facility, the assistant needs both location and facility type to perform the search. We trained the NLU model to extract the location and the facility type from the user requests, if provided. If the user asks for a facility search, but hasn't yet provided location and facility type, the assistant directs the conversation to ask for them, before performing the search. In this way, the dialogue management model uses the provided information to drive the conversation.

This is implemented in Rasa using slots. The details of location and facility type are stored as slots and allow the dialogue management model to use the content of the slots to determine the next step in the conversation.

```
domain.yml
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - inform
9   - search_provider
10
11 entities:
12   - location
13   - facility_type
14
15 slots:
16   location:
17     type: text
18
19
20 actions:
21   - utter_greet
22   - utter_cheer_up
23   - utter_did_that_help
24   - utter_happy
25   - utter_goodbye
26   - utter_how_can_i_help
27   - utter_ask_location
28   - action_facility_search
29
30 templates:
31   utter_greet:
32     - text: "Hey! How are you?"
```

Slots have a dedicated section in a domain file. To define a slot, you have to provide two pieces of information - slot name and a slot type. Slot names can match the names of the entities. If there is a match, the values of extracted entities are automatically set as slots. (You can disable this behavior by setting the `auto_fill` attribute to `False` for specific slots.)

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Slot Types

Slot types are extremely important and have a direct influence on the predictions the dialogue management makes. For some types, only the presence or the absence of the slot matters, in other cases the values of the slots matter too.

Slot Type: text

- Use for: user preferences where you care only whether or not the preference has been specified
- A slot of the type `text` only tells Rasa whether the slot has been set or not. The actual value doesn't make any difference.
- Slots with type `text` are useful when modelling the situation where the dialogue should take different turns, depending on whether or not users provided specific details.
- Example: if the location slot has not been set, an assistant should ask for this detail, otherwise, move forward and drive the conversation further.

Slot Type: bool

- Use for: true or false
- When using this slot type, not only the presence or absence of this slot matters, but also the value
Example: the dialogue management model checks if a slot value is true and based on that, drives the conversation further.

Slot Type: categorial

- Use for: slots which can take one of N values
- Both the presence of and the value of the categorical slot matters.
- Categorical slots are useful when a piece of information can take one of N possible values.
- For example: slot with possible values of low, medium, and high. A dialogue management model can take the value of the slot into account and use it to make the prediction for the next best action to respond with.

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Slot Type: float

- Use for: continuous values
- Useful to store continuous values like float numbers
- Both the presence of the slot and the value matters
- The float slot has the parameters `min_value` and `max_value`, which lets you define the highest and lowest possible values for the slot. Anything that is above `max_value` will be set to `max_value` while everything that is below `min_value` will be set to `min_value`.

Slot type: list

- Use for: list of values
- If the NLU model extracts more than one value for an entity, you might want to store all the provided values.
- A slot with the type list is designed to store details with multiple values.
- The length and values of a list slot doesn't really matter; what matters is if the slot is empty or filled.

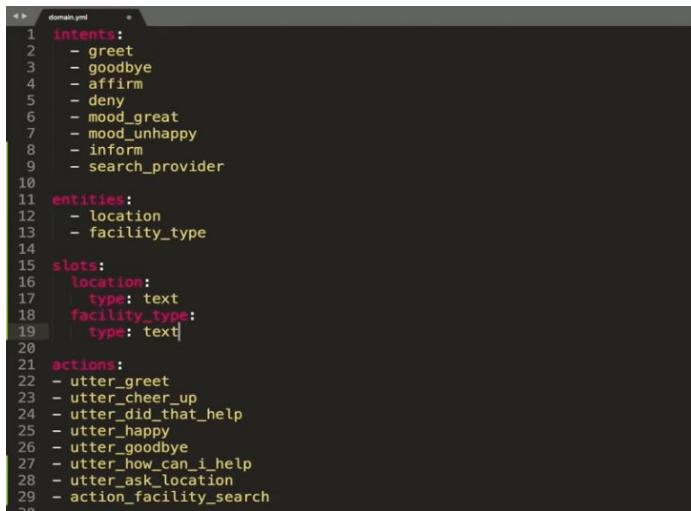
Slot type: unfeaturized

- Use for: continuous values
- Useful when you want to use slots to store the information as extracted, but not to drive the dialogue.
- Slot type unfeaturized means that neither the value of the slot nor whether it's filled will influence the dialogue management model's predictions.

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Using Slots in the Medicare Locator

Here we will define the slots for the medicare locator assistant.

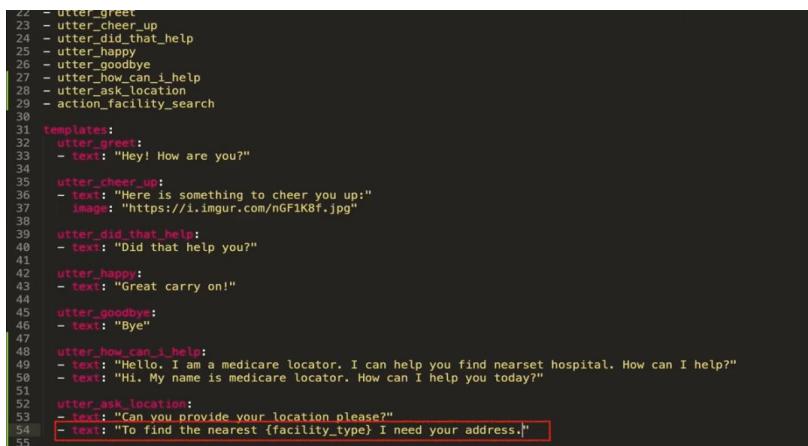


```
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - inform
9   - search_provider
10
11 entities:
12   - location
13   - facility_type
14
15 slots:
16   location:
17     type: text
18   facility_type:
19     type: text
20
21 actions:
22   - utter_greet
23   - utter_cheer_up
24   - utter_did_that_help
25   - utter_happy
26   - utter_goodbye
27   - utter_how_can_i_help
28   - utter_ask_location
29   - action_facility_search
30
31 templates:
32   utter_greet:
33     - text: "Hey! How are you?"
34
35   utter_cheer_up:
36     - text: "Here is something to cheer you up:"
37     - image: "https://i.imgur.com/nGF1K8f.jpg"
38
39   utter_did_that_help:
40     - text: "Did that help you?"
41
42   utter_happy:
43     - text: "Great carry on!"
44
45   utter_goodbye:
46     - text: "Bye"
47
48   utter_how_can_i_help:
49     - text: "Hello. I am a medicare locator. I can help you find nearest hospital. How can I help?"
50     - text: "Hi. My name is medicare locator. How can I help you today?"
51
52   utter_ask_location:
53     - text: "Can you provide your location please?"
54     - text: "To find the nearest {facility_type} I need your address."
```

In addition to driving conversations, slots can be useful in other situations when developing your assistant.

Using Slots to Customize Templates

Slots can be used when creating the responses of an assistant. Slots hold the values of important details, which can be added to utterance templates to make them more personal. Simply provide the slot name in curly brackets within the template.



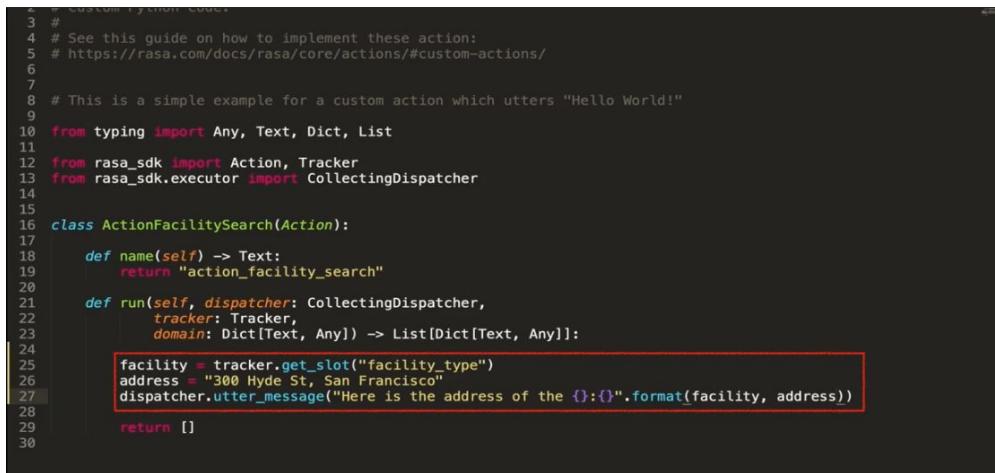
```
22   - utter_greet
23   - utter_cheer_up
24   - utter_did_that_help
25   - utter_happy
26   - utter_goodbye
27   - utter_how_can_i_help
28   - utter_ask_location
29   - action_facility_search
30
31 templates:
32   utter_greet:
33     - text: "Hey! How are you?"
34
35   utter_cheer_up:
36     - text: "Here is something to cheer you up:"
37     - image: "https://i.imgur.com/nGF1K8f.jpg"
38
39   utter_did_that_help:
40     - text: "Did that help you?"
41
42   utter_happy:
43     - text: "Great carry on!"
44
45   utter_goodbye:
46     - text: "Bye"
47
48   utter_how_can_i_help:
49     - text: "Hello. I am a medicare locator. I can help you find nearest hospital. How can I help?"
50     - text: "Hi. My name is medicare locator. How can I help you today?"
51
52   utter_ask_location:
53     - text: "Can you provide your location please?"
54     - text: "To find the nearest {facility_type} I need your address."
```

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Slots can also be used in custom actions when running backend integrations, querying database, making API calls, and more.

Slots & Custom Actions

Using slots, let's update our custom action `action_facility_search`.



```
# See this guide on how to implement these actions:
# https://rasa.com/docs/rasa/core/actions/#custom-actions/
# This is a simple example for a custom action which utters "Hello World!"
from typing import Any, Text, Dict, List
from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher

class ActionFacilitySearch(Action):
    def name(self) -> Text:
        return "action_facility_search"
    def run(self, dispatcher: CollectingDispatcher,
           tracker: Tracker,
           domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        facility = tracker.get_slot("facility_type")
        address = "300 Hyde St, San Francisco"
        dispatcher.utter_message("Here is the address of the {}:{}.".format(facility, address))
        return []


```

A tracker keeps track of the slots set at each dialogue state. The values of these slots can be returned using the `get_slot` method, and providing the name of the slot we need (“`facility_type`”). Then we can enable the custom action to extract the user’s requested facility type and use this slot to find an address of the facility that meets the user’s requirements.

For the sake of simplicity in this example, we hard-coded a suggested facility address (300 Hyde St, San Francisco).

Slots can be set by the NLU model as well as by custom actions. This is useful when important information is provided by running a backend integration, which should be saved and used later on throughout the conversation. For example, when our medicare locator assistant finds the address of a facility for the user, it is useful to save this piece of information, because the user may ask follow-up questions about the facility. Saving the address as a slot lets your assistant handle this situation. You can set slots in custom actions using the `SetSlot` event.

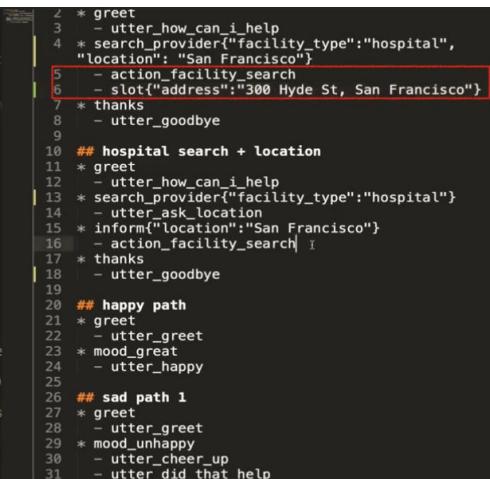
CHAPTER 6: DOMAIN, ACTIONS, SLOTS

In our medicare locator, we can import the SlotSet event, and update the return statement of the run function with the SlotSet event, providing the slot name and the value.

```
3 #  
4 # See this guide on how to implement these action:  
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/  
6  
7 # This is a simple example for a custom action which utters "Hello World!"  
8  
9 from typing import Any, Text, Dict, List  
10  
11 from rasa_sdk import Action, Tracker  
12 from rasa_sdk.executor import CollectingDispatcher  
13 from rasa_sdk.events import SlotSet  
14  
15  
16 class ActionFacilitySearch(Action):  
17     def name(self) -> Text:  
18         return "action_facility_search"  
19     def run(self, dispatcher: CollectingDispatcher,  
20             tracker: Tracker,  
21             domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:  
22         facility = tracker.get_slot("facility_type")  
23         address = "300 Hyde St, San Francisco"  
24         dispatcher.utter_message(f"Here is the address of the {facility}: {address}")  
25  
26         return [SlotSet("address", address)]  
27  
28  
29  
30  
31
```

An important note about slots set using a custom action: Rasa requires that these slot events be reflected in the training stories (stories.md). So, in our medicare locator assistant, `action_facility_search` sets a slot `address`, and so in our stories, this action must be followed by an event slot. This is necessary so that the dialogue management model can access these details to make the right decision on how to respond going forward.

```
2 # Custom Python code.  
3 #  
4 # See this guide on how to implement these action:  
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/  
6  
7 # This is a simple example for a custom action which  
8 # utters "Hello World!"  
9  
10 from typing import Any, Text, Dict, List  
11  
12 from rasa_sdk import Action, Tracker  
13 from rasa_sdk.executor import CollectingDispatcher  
14 from rasa_sdk.events import SlotSet  
15  
16 class ActionFacilitySearch(Action):  
17     def name(self) -> Text:  
18         return "action_facility_search"  
19     def run(self, dispatcher: CollectingDispatcher,  
20             tracker: Tracker,  
21             domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:  
22         facility = tracker.get_slot("facility_type")  
23         address = "300 Hyde St, San Francisco"  
24         dispatcher.utter_message(f"Here is the address of the {facility}: {address}")  
25  
26         return [SlotSet("address", address)]  
27  
28  
29  
30  
31
```



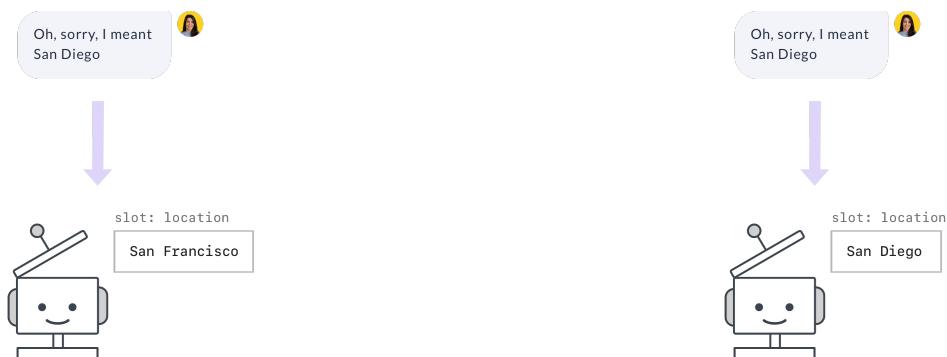
CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Lastly, remember that all slots have to be listed in the domain. Since we created a new slot address, it needs to be added to the domain file as well.

```
 1 # Custom Python code.
 2 #
 3 # See this guide on how to implement these actions:
 4 # https://rasa.com/docs/rasa/core/actions/#custom-actions
 5 #
 6 #
 7 #
 8 # This is a simple example for a custom action which
 9 #
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14 from rasa_sdk.events import SlotSet
15
16
17 class ActionFacilitySearch(Action):
18
19     def name(self) -> Text:
20         return "action_facility_search"
21
22     def run(self, dispatcher: CollectingDispatcher,
23             tracker: Tracker,
24             domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
25
26         facility = tracker.get_slot("facility_type")
27         address = "300 Hyde St, San Francisco"
28         dispatcher.utter_message("Here is the address")
29
30         return [SlotSet("address", address)]
31
32
33 # greet
34 # goodbye
35 # affirm
36 # deny
37 # mood_great
38 # mood_unhappy
39 # inform
40 # search_provider
41
42 entities:
43 - location
44 - facility_type
45
46 slots:
47 - location:
48     type: text
49     facility_type:
50     type: text
51     address:
52     type: unfeaturized
53
54 actions:
55 - utter_greet
56 - utter_cheer_up
57 - utter_did_that_help
58 - utter_happy
59 - utter_goodbye
60 - utter_how_can_i_help
61 - utter_ask_location
62 - action_facility_search
```

Resetting Slots

Slot values are kept in memory until reset. If the slot is set by the NLU model, then every time a new value is extracted, the slot value will be updated with the most recently extracted one. The same is true of slots set by the custom actions. In some situations you may want to reset specific or all slots. To achieve that, you can use Rasa in-built events like `reset slot`.



CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Training your Rasa Assistant

With domain and custom actions in place, we can train the first version of our assistant and see how it works.

To do that we can use the command called `rasa train` which will retrain the NLU model alongside the dialogue management model. Once the model is trained, we can test it on our command line by running a command `rasa shell`. The `rasa train` command will also tell us if we forgot any details in the domain file.

Custom actions run on a separate server than the one the models run on. Rasa will call an endpoint, which you specify, when a custom action is predicted. This endpoint should be a web server that reacts to the call, runs the code, and optionally, returns information to modify the dialogue state. The full configuration of the custom action server is provided in the file in the project directory named `endpoints.yml`.



Let's update the file to use the endpoint for our medicare locator model.

```
3 # Server where the models are pulled from.
4 # https://rasa.com/docs/rasa/user-guide/running-the-server/#fetching-models-from-a-server/
5
6 #models:
7 # url: http://my-server.com/models/default_core@latest
8 # wait_time_between_pulls: 10 # [optional] (default: 100)
9
10 # Server which runs your custom actions.
11 # https://rasa.com/docs/rasa/core/actions/#custom-actions/
12
13 action_endpoint:
14 | url: "http://localhost:5055/webhook"
15
16 # Tracker store which is used to store the conversations.
17 # By default the conversations are stored in memory.
18 # https://rasa.com/docs/rasa/api/tracker-stores/
19
20 #tracker_store:
21 #   type: redis
22 #   url: <host of the redis instance, e.g. localhost>
23 #   port: <port of your redis instance, usually 6379>
24 #   db: <number of your database within redis, e.g. 0>
25 #   password: <password used for authentication>
26
```

CHAPTER 6: DOMAIN, ACTIONS, SLOTS

Now we can start the custom action server using the Rasa CLI function `rasa run actions` and load the assistant for us to test, using the Rasa CLI function `rasa shell`. With this, we have the very first version of our medicare locator assistant.

Conclusion

Here, we used a default dialogue management model created by `rasa init` function. A model is defined using training policies. We will talk about them in [Episode 7](#) of the Rasa Masterclass, where we will understand better what the training policies are, how they work, how to define them in the configuration file, and in what situations one policy may be better than another.

If you're coding along, now would be the time to start having conversations with your assistant, to see how it works. Keep up the progress, and if you get stuck, ask us a question in the [Community forum](#).

Additional Resources

- [Ep #5 - Rasa Masterclass - Intro to dialogue management with Rasa](#) (YouTube)
- [Ep. #6 - Rasa Masterclass - Dialogue management with domains, custom actions, and slots](#) (YouTube)
- [About Rasa Core](#) (Rasa docs)
- [Stories](#) (Rasa docs)
- [Rasa: Open Source Language Understanding and Dialogue Management](#) (arXiv)



CHAPTER SEVEN

DIALOGUE POLICIES

DIALOGUE POLICIES



Introduction

In [Episode 7](#) of the Rasa Masterclass, we cover dialogue policies. In our last few episodes, we've been discussing dialogue management—the process that determines what an assistant does next in response to user input. Policies are components that train the dialogue model, and they play a very important role in determining its behavior. Some policies are quite simple, like those that mirror the conversations they've been trained on, and some are quite complex, like those that rely on sophisticated machine learning to predict the next action based on the context of the conversation.

In this episode, we'll cover the policies that are available in Rasa, how developers can configure them, and how to decide which policies to use. Because we'll be building on concepts covered in previous episodes, be sure to watch [Episode 5](#) and [Episode 6](#) before proceeding.

CHAPTER 7: DIALOGUE POLICIES

Policy Configuration in Rasa

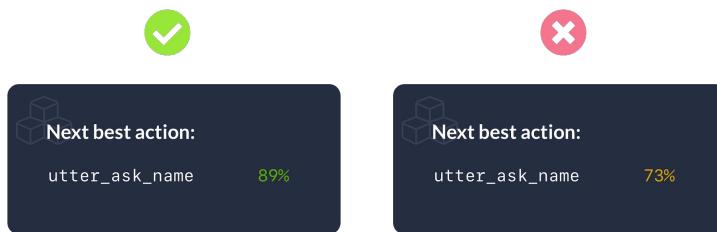
Just like the NLU training pipeline we configured in [Episode 4](#), dialogue policies are also configured in the config.yml file, which you'll find in your main project directory.

Let's take a look at the default configuration generated by `rasa init`:

```
config.yml
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: en
pipeline: pretrained_embeddings_spacy

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  - name: MemoizationPolicy
  - name: KerasPolicy
  - name: MappingPolicy
```

The policy configuration is defined by a list of policy names, along with optional parameters that can be configured by developers. Unlike the NLU training pipeline, which runs components sequentially, dialogue policies run in parallel. At each conversational turn, each policy in the configuration makes its own prediction about the next best action. The policy that predicts the next action with the highest confidence level determines the assistant's next action.



CHAPTER 7: DIALOGUE POLICIES

In cases where two policies predict with equal confidence, a priority system kicks in to decide which policy will “win.” Rasa automatically weights policies according to default priority levels, which are tuned to make sure the assistant chooses the most appropriate action when there’s a tie. Higher numbers are given higher priority, and machine learning policies are given a priority of 1:

5. FormPolicy
4. FallbackPolicy and TwoStageFallbackPolicy
3. MemoizationPolicy and AugmentedMemoizationPolicy
2. MappingPolicy
1. EmbeddingPolicy, KerasPolicy, and SklearnPolicy

Generally speaking, you should only have one policy per priority level to avoid conflicts, and some policies, like Fallback and TwoStageFallback, explicitly cannot be used together. We’ll discuss configuration in greater detail when we cover each policy in depth. For now, keep in mind that multiple policies can be used together, and the highest confidence, highest priority policy predicts the assistant’s next action.

Hyperparameters

While individual policies have their own unique parameters that can be configured to adjust the model’s behavior, there are two important parameters that are common to all Rasa core dialogue policies: **max-history** and **data augmentation**.

Max-history

When a policy makes a prediction about the next action to take, it doesn’t just look at the last thing the user said. It also considers what’s happened in previous conversational turns. The max-history parameter controls how many previous conversational turns the policy should look at when making a prediction. This can be especially useful if you want to train your assistant to respond to certain patterns in user messages, like repeated off topic requests.

Let’s use our medicare locator assistant, which helps users locate nearby medical facilities, as an example. If a user asked the assistant several times in a row where to find the nearest Italian restaurant, we’d probably want the assistant to respond with a help message telling the user what it *can* help with. Assuming we’d defined an `out_of_scope` intent to catch off-topic requests, the story might look like this:

CHAPTER 7: DIALOGUE POLICIES

```
* out_of_scope
  - utter_default
* out_of_scope
  - utter_default
* out_of_scope
  - utter_help_message
```

In order for the assistant to pick up this pattern, we'd need to set the `max_history` to at least 3.

So a higher `max_history` is always better, right? Not so fast. A higher `max_history` creates a larger model, which can cause performance issues. If you need your assistant to remember certain details from farther back in the conversation, it's better to save those details as slots instead of setting `max_history` to a very large number.

Data augmentation

By default, Rasa core combines randomly-selected stories in your training data file, a process known as data augmentation. Data augmentation is used to teach the model to ignore conversation history when it's not relevant. For example, with short stories like these, the next action should be the same, no matter what happened before in the conversation:

```
# thanks
* thankyou
  - utter_youarewelcome
```

```
# bye
* goodbye
  - utter_goodbye
```

You can control this behavior using the `--augmentation` flag in the policy configuration. By default, augmentation is set to a factor of 20, which creates 200 augmented stories. A higher factor increases the amount of data the model has to process and results in higher training times. Setting `--augmentation 0` disables the behavior, which you might want to do if you have a lot of training examples.

CHAPTER 7: DIALOGUE POLICIES

Dialogue Policies

Now, we'll discuss the available Rasa training policies, one by one. The policies are:

- Memoization Policy
- Mapping Policy
- Keras Policy
- Embedding Policy (TEDP)
- FormPolicy
- FallbackPolicy

For each policy, we'll discuss how the policy makes predictions and influences the behavior of Rasa core, as well as the parameters available for developers to configure.

Memoization Policy

The Memoization Policy is one of the simpler dialogue policies we'll discuss. It snips the part of the conversation defined by the `max_history` (so if `max_history: 3`, 3 turns back), and looks for a matching story fragment in the training data set. If it finds a match, it predicts the same next action seen in the training data.



An important detail about the Memoization Policy is that it predicts with 100% certainty: if a match is found in the training data, it predicts the next action with a confidence level of 1, otherwise it predicts None, with a confidence level of 0.

CHAPTER 7: DIALOGUE POLICIES

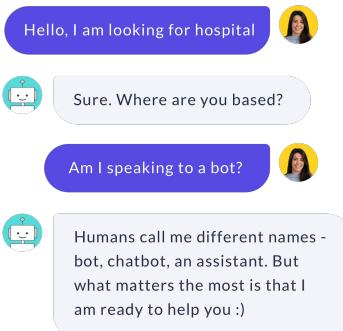
The Memoization policy is very useful for making sure your assistant follows the stories you've provided in your training data. It enforces the patterns and conversational paths you want your assistant to follow, much like a set of rules. However, when a match isn't found in the training data, the Memoization policy can't provide a next action. That's why the Memoization Policy isn't meant to be used on its own—it's used with other policies that fill in the gaps when an exact match isn't available.

Configuration

- Max_history - Determines the length of story fragments an assistant should consider when making the next prediction. Default configuration is 5
- Priority - While you can reset the policy's priority, we recommend leaving this as the default.

Mapping Policy

The Mapping Policy maps an intent to a specific action. This is useful when you know that an intent should always be followed by a certain response, regardless of what has happened previously in the conversation. For example, if the user asks if they're speaking to a bot in the middle of the conversation, the assistant should always respond with the same message, without breaking the flow of conversation.



Because the Mapping Policy is so specialized, it will always need to be used in combination with other policies, to generate action predictions outside of those that have been explicitly mapped.

CHAPTER 7: DIALOGUE POLICIES

Configuration

To enable the mapping policy, include `MappingPolicy` in your `config.yml` policies and then map intents to actions in the `domain.yml` file, using the `triggers` property. An intent can be mapped only to a single action.

Here, we're mapping the `ask_is_bot` intent to a custom action that prints a response message.

```
intents:  
- ask_is_bot:  
  triggers: action_is_bot
```

Preventing mapped actions from influencing later predictions

When a mapped action has been triggered, that action is taken into account by other policies making later predictions, just like any other conversational turn. If you want to completely remove the mapped action from the conversational flow, so it doesn't influence later predictions, you can disable the behavior by adding the `UserUtteranceReverted()` event to your custom action. This deletes the interaction from memory so it won't affect the predictions of other policies later in the conversation.

```
class ActionIsBot(Action):  
    """Revertible mapped action for utter_is_bot"""\n\n    def name(self):  
        return "action_is_bot"  
  
    def run(self, dispatcher, tracker, domain):  
        dispatcher.utter_template("utter_is_bot", tracker)  
        return [UserUtteranceReverted()]
```

Keras Policy

The Keras Policy applies the power of machine learning to dialogue management. It learns from training data, and over time, your assistant can learn to handle advanced conversations. The policy uses the neural network implemented in [Keras](#), a Python deep learning library. The default architecture is based on [Long Short Term Memory](#) (LSTM), a type of recurrent neural network (RNN) but this can be overridden in the [Keras Policy model_architecture method](#).

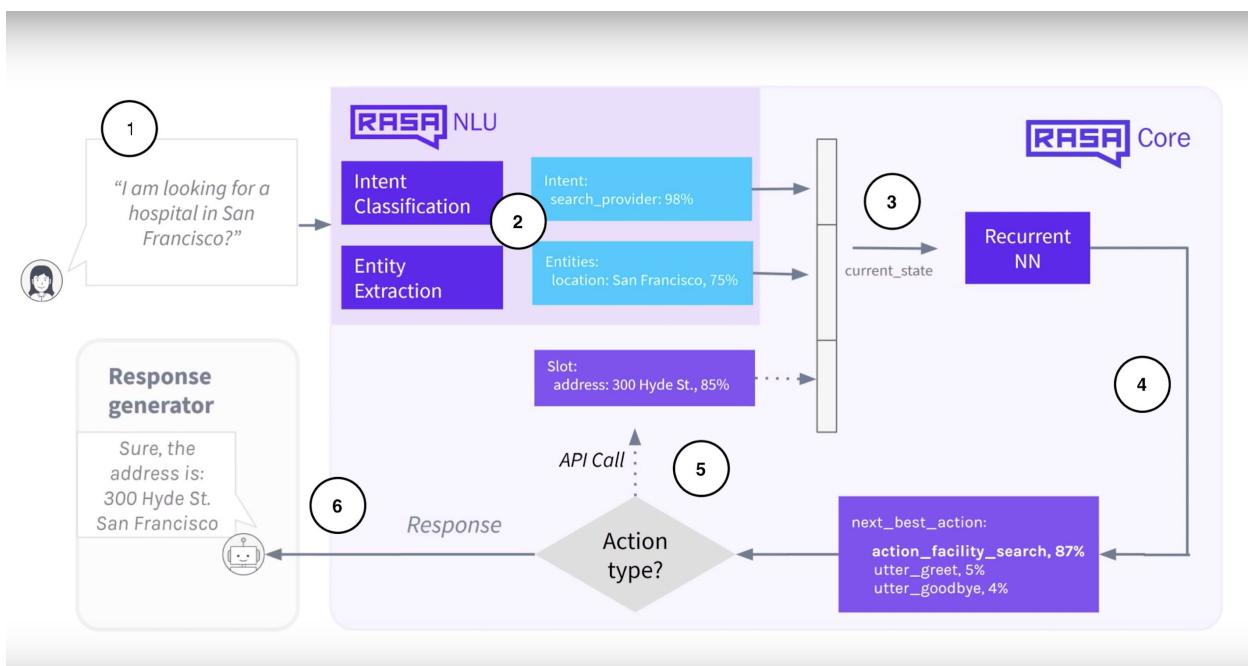
CHAPTER 7: DIALOGUE POLICIES

The Keras policy considers multiple factors when making a prediction, including:

- The last action
- The intents and entities extracted by the NLU model
- The slots that have been set
- Previous conversational turns

Let's walk through the process step by step to understand how the Keras Policy decides which action to predict.

1. First, the user asks a question.
2. Rasa NLU extracts the intents and entities from the user's message, which are used to create the feature vector passed to the dialogue model. A feature vector is a numeric representation that describes important characteristics of the message.
3. The dialogue model takes in previous conversational states, as set by the `max_history` hyperparameter, and the feature vector of the current state is also passed into the model.
4. The model predicts the next best action.
5. If the next action includes a call to a backend integration that fetches additional details, those details can be featurized and used to predict future actions.
6. The predicted response is sent back to the user.



CHAPTER 7: DIALOGUE POLICIES

Configuration

- Max_history - Defines the number of previous conversational states the model should take into account
- Epochs - The number of times the algorithm should pass through the training data, where an epoch equals one forward pass and one backward pass of all training examples.
- Validation_split - The portion of training data set apart and not used for training, to evaluate the model's performance
- Random_seed - Helps the model achieve reproducible results, when set to an integer value. By design, neural networks use randomness to achieve the best performance: in the initial weights set in a neural network, the training data that's sampled, and in other areas. The seed number is the starting point used by the random number generator to generate a random sequence; when we specify a random_seed value, the same starting point is used each time, resulting in the same random number sequence.

```
config.yml
1 # Configuration for Rasa NLU.
2 # https://rasa.com/docs/rasa/nlu/components/
3 language: en
4 pipeline: pretrained_embeddings_spacy
5
6 # Configuration for Rasa Core.
7 # https://rasa.com/docs/rasa/core/policies/
8 policies:
9   - name: MemoizationPolicy
10    - name: KerasPolicy
11      max_history: 3
12      epochs: 200
13      validation_split: 0.1
14      random_seed: 1
15    - name: MappingPolicy
16
17
```

CHAPTER 7: DIALOGUE POLICIES

Embedding Policy (TEDP)

Transformer Embedding Dialogue Policy, also known as TED Policy, is Rasa's default machine learning based policy. Compared to other machine learning policies like Keras, our research has shown that TED Policy performs better, especially when it comes to modelling multi-turn conversations.

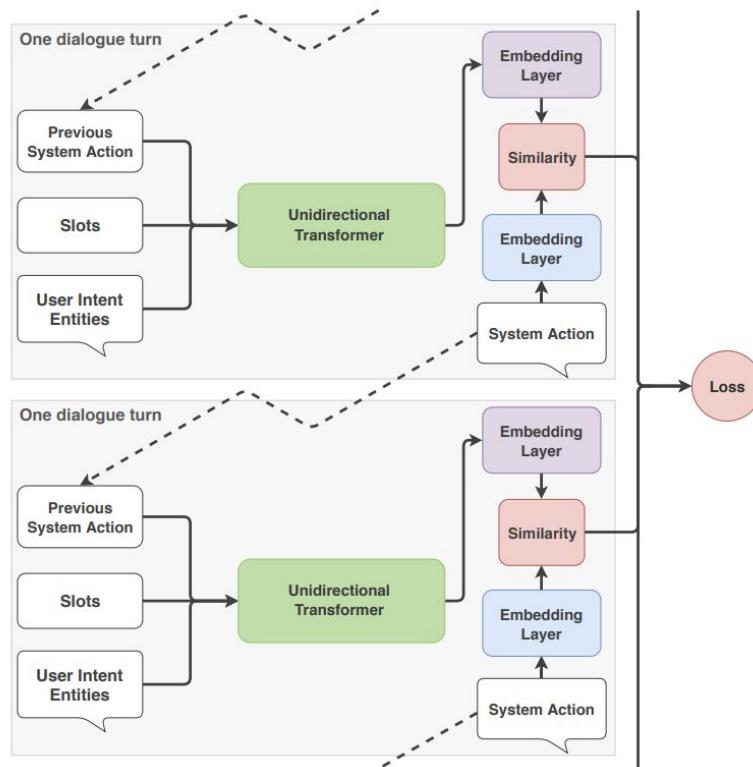
Instead of using a recurrent neural network like Keras Policy, TED Policy uses Transformer, a deep machine learning model that is now overtaking RNNs in popularity. Transformers produce accurate results across a variety of corpora, or datasets, and they also deal well with unexpected user input, like adversarial messages or chitchat.

How does the TED Policy make predictions? Let's walk through the high level architecture (shown in the diagram below). The diagram depicts two time steps, or dialogue turns.

1. First, input is aggregated from a variety of sources:
 - a. Intent and entities extracted from the user's message by the NLU model
 - b. Any previous system actions
 - c. Data saved as slots.

These are concatenated into an input vector and fed into the transformer deep learning model.
2. A dense layer is applied to the transformer's output to get dialogue embeddings for each time stamp.
3. A dense layer is applied to create embeddings for each categorical system action, for each time step.
4. The similarity between the dialogue embedding and the embedded system actions is calculated (this concept is based on the [StarSpace](#) idea).

CHAPTER 7: DIALOGUE POLICIES



Configuration

The TED Policy includes configurable hyperparameters that allow developers to adjust conditions that affect the neural network architecture, training, embedding, and more. See the documentation for a [full list of hyperparameters and their default values](#).

Form Policy

The Form Policy is used in situations where the assistant needs to collect specific pieces of information from the user before executing an action. For example, in order for the Medicare Locator assistant to fetch search results for medical facilities, it needs two pieces of information: the type of facility and the location. Without those, it's impossible to complete the search request. If we were to develop the medicare locator assistant further, we might want to ask for additional information, like the patient's name and age, in order to send these details to the hospital ahead of the patient's visit.

CHAPTER 7: DIALOGUE POLICIES

Another common use case is account creation. You might have a minimum number of fields that are required to create an account in a CRM or billing system—things like name, address, phone number, etc. If you were building an assistant that helps users create a new account, you would want to be sure that the assistant collected all of the required pieces of information from the user before moving forward in the conversation.

You could enable this behavior by writing stories with slots, but that would require a lot of training data, even just to handle the happy path. A better solution is to use the Rasa Form Policy to collect the information. The Form Policy allows you to define which pieces of information the assistant needs to collect. It activates a form action, which runs continually until all of the required data has been provided by the user.



Forms are a powerful feature in Rasa, and there's more to the topic than we can cover here. We'll dedicate the next episode to covering forms in greater detail.

Fallback Policy

Even the best contextual assistants get stumped from time to time, if the user's message is outside of the assistant's domain or off-topic. Since it's impossible to train your assistant to handle every possible situation, it's important to have a strategy for gracefully handling requests the assistant doesn't understand.

CHAPTER 7: DIALOGUE POLICIES

The Fallback Policy is activated when the assistant can't predict an intent or next action with certainty above a certain threshold. The Fallback Policy also kicks in when the confidence levels of two intents are very close together. The confidence levels that should trigger the fallback policy are configured as hyperparameters. When the Fallback Policy is triggered, [an action is executed](#)—usually an utterance letting the user know the assistant didn't understand and asking them to rephrase their message.

Configuration

The Fallback Policy accepts four hyperparameters that set the minimum confidence thresholds for triggering the fallback. These are:

- nlu_threshold - Min confidence needed to accept an NLU prediction
- ambiguity_threshold - Min amount by which the confidence of the top intent must exceed that of the second highest ranked intent.
- core_threshold - Min confidence needed to accept an action prediction from Rasa Core
- fallback_action_name - Name of the fallback action to be called if the confidence of intent or action is below the respective threshold. You can use the default fallback action, or you can configure your own. If you specify your own custom action, be sure to include it in your domain.yml and stories.md files.

Two-stage Fallback Policy

The Two-stage Fallback Policy is a more sophisticated variation of the Fallback Policy we just discussed. Instead of immediately executing the fallback action, the Two-stage Fallback Policy asks the user to verify the predicted intent. If the user verifies the intent was correct, the story continues. If the user tells the assistant the predicted intent was *not* what they meant, the assistant asks the user to rephrase the message.

CHAPTER 7: DIALOGUE POLICIES

Besides providing a better user experience, the Two-stage Fallback Policy also allows the assistant to correct itself and recover in cases when it's not confident in a prediction.

While dialogue policies are meant to be used together, a notable exception is the Fallback Policy and the Two-stage Fallback Policy. You can't choose both when configuring your dialogue policies in the config.yml file—it has to be one or the other.

Configuration

In addition to the nlu_threshold, ambiguity_threshold, and core_threshold hyperparameters, the Two-stage Fallback Policy also includes three more hyperparameters that define the actions executed at each stage of the fallback flow:

- fallback_core_action_name - Name of the fallback action called when the confidence of the Rasa Core action prediction is below the core_threshold. This action suggests possible intents for the user to choose from.
- fallback_nlu_action_name - Name of the fallback action to be called if the confidence of the Rasa NLU intent classification is below the nlu_threshold. This action is called when the user denies the second time
- deny_suggestion_intent_name - The name of the intent used to detect that the user has denied the suggested intents



CHAPTER 7: DIALOGUE POLICIES

Conclusion

Customizing your policy configuration and fine-tuning parameters is a powerful way to take your Rasa assistant to the next level. Although the default configuration provided with moodbot is a great place to get started, over time, you can layer on additional policies to enhance your assistant's performance.

In our next episode, we'll dive deeper into developing the medicare locator assistant, focusing on implementing custom actions with backend integrations, forms and fallback. Keep up the momentum, and if you get stuck, ask us a question in the [Community forum](#).

Additional Resources

- [Ep #7 Rasa Masterclass - Dialogue Policies](#) (YouTube)
- [Dialogue Policies](#) (Rasa docs)
- [Dialogue Transformers](#) (arXiv)
- [StarSpace: Embed All the Things!](#) (arXiv)
- [Fallback Actions](#) (Rasa docs)
- [Failing Gracefully with Rasa](#) (Rasa blog)



CHAPTER EIGHT

INTEGRATIONS, FORMS, AND FALLBACKS

INTEGRATIONS, FORMS, AND Fallbacks



Introduction

In this edition of the Rasa Masterclass + Handbook, we will take the theoretical learnings from the prior 7, and apply them in practice. In the previous episodes we covered a great deal of theoretical concepts involved in the development of AI assistants with Rasa. Episode #8 of the Rasa Masterclass takes much of what you've learned in prior episodes and helps you use these learnings in a hands-on way. In this episode, we will focus more on the actual development of our medicare locator assistant to show you how things work in practice.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

At the end of this tutorial, we will have completed the following for our medicare locator assistant:

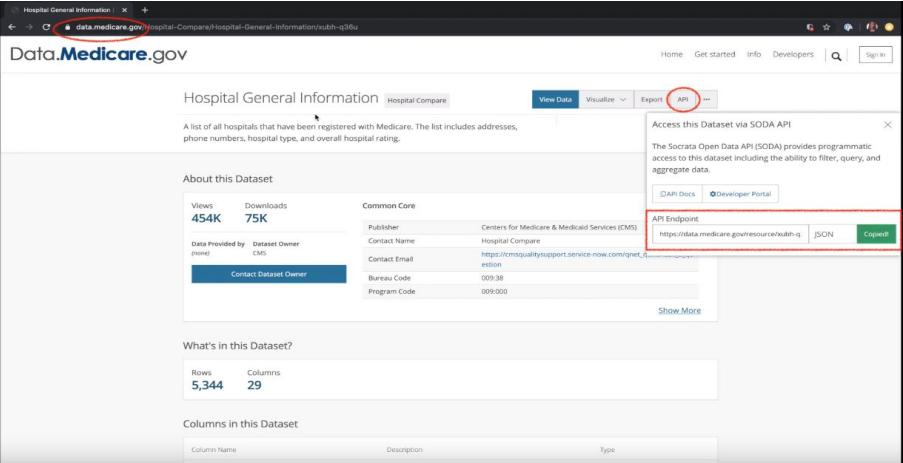
- Update the NLU and stories data to expand the knowledge and skills of our medicare locator assistant
- Implement custom actions which will leverage API calls and connections to the database to extract necessary information
- Implement form action
- Enable our assistant to fail gracefully

Let's get started.

Real-World Dataset for our Medicare Locator

At the moment, the medicare locator assistant is capable of understanding some simple inputs like greetings & goodbyes, handling some simple interactions with a user, such as a request to find a specific health facility, and dialogue in which the user provides some information, like their location or the type of facility they are seeking. With this information, we enabled our assistant to run a simple action (`action_facility_search`) once the user asks for a suggestion. In this section, we will upgrade this ability to enable the assistant to collect all necessary information from the user, and run a real backend integration to provide the location of the type of facility requested.

We will start with the data source for the backend integration. We will use a publically available database from www.medicare.gov, which provides a variety of open datasets, including information about different health facilities across the U.S.



The screenshot shows the Data.Medicare.gov interface for the 'Hospital General Information' dataset. At the top, there are links for 'View Data', 'Visualize', 'Export', and 'API'. The 'API' link is circled in red. Below it, there's a section titled 'About this Dataset' with statistics: 'Views 454K' and 'Downloads 75K'. To the right, there's a 'Common Core' section with fields for Publisher (Centers for Medicare & Medicaid Services (CMS)), Contact Name (Hospital Compare), Contact Email, Bureau Code (009-38), and Program Code (009-000). A red box highlights the 'API Endpoint' field, which contains the URL <https://data.medicare.gov/resource/hlth-q36u>. Other options shown are 'CSV' and 'JSON'. At the bottom, there are sections for 'What's in this Dataset?' (Rows 5,344, Columns 29) and 'Columns in this Dataset'.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

The data the assistant needs can be pulled using the provided API. You can see above where the endpoint is specified and can be copied. It looks like this:

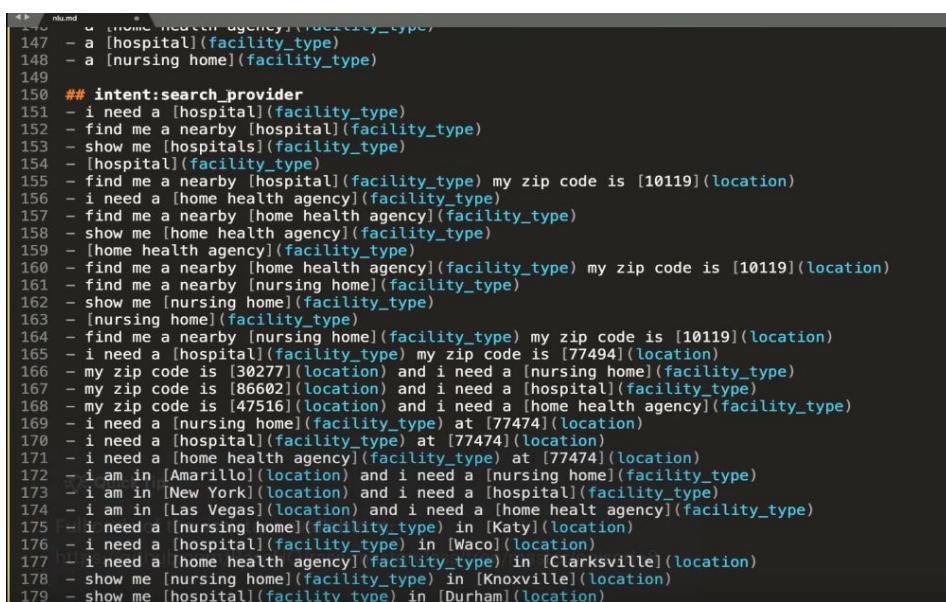
<https://data.medicare.gov/resource/xubh-q36u.json>

There are different endpoints for different types of facilities, and they are differentiated using a resource code. In the example above, the endpoint contains `xubh-q36u` which specifies the dataset for hospitals. Two other important resource codes (that we will use later) are `f7df-2ac7` for home health agencies and `b27b-2uc7` for nursing homes.

In the API call itself, we can further narrow the search using parameters like city name or zip code. This is a perfect dataset to provide real-world data about specific healthcare facility locations to our assistant.

Improving the NLU

Previously, we built a simple NLU model capable of classifying intents and extracting a few entities. The model works, but is prone to mistakes: our limited training data doesn't provide enough information for the NLU to be very accurate. A good next step will be to update the NLU data and add more training examples to all of the intents in our training data file to improve the performance of the model.



```
nlu.md
146 - [home health agency](facility_type)
147 - a [hospital](facility_type)
148 - a [nursing home](facility_type)
149
150 ## intent:search_provider
151 - i need a [hospital](facility_type)
152 - find me a nearby [hospital](facility_type)
153 - show me [hospitals](facility_type)
154 - [hospital](facility_type)
155 - find me a nearby [hospital](facility_type) my zip code is [10119](location)
156 - i need a [home health agency](facility_type)
157 - find me a nearby [home health agency](facility_type)
158 - show me [home health agency](facility_type)
159 - [home health agency](facility_type)
160 - find me a nearby [home health agency](facility_type) my zip code is [10119](location)
161 - find me a nearby [nursing home](facility_type)
162 - show me [nursing home](facility_type)
163 - [nursing home](facility_type)
164 - find me a nearby [nursing home](facility_type) my zip code is [10119](location)
165 - i need a [hospital](facility_type) my zip code is [77494](location)
166 - my zip code is [30277](location) and i need a [nursing home](facility_type)
167 - my zip code is [86602](location) and i need a [hospital](facility_type)
168 - my zip code is [47916](location) and i need a [home health agency](facility_type)
169 - i need a [nursing home](facility_type) at [77474](location)
170 - i need a [hospital](facility_type) at [77474](location)
171 - i need a [home health agency](facility_type) at [77474](location)
172 - i am in [Amarillo](location) and i need a [nursing home](facility_type)
173 - i am in [New York](location) and i need a [hospital](facility_type)
174 - i am in [Las Vegas](location) and i need a [home health agency](facility_type)
175 - i need a [nursing home](facility_type) in [Katy](location)
176 - i need a [hospital](facility_type) in [Waco](location)
177 - i need a [home health agency](facility_type) in [Clarksville](location)
178 - show me [nursing home](facility_type) in [Knoxville](location)
179 - show me [hospital](facility_type) in [Durham](location)
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Remember, you can download the full code of the assistant from:

<https://github.com/RasaHQ/rasa-masterclass/tree/master/episode8>

If you examine the nlu.md file, you will see we added many new intents (e.g., intent: affirm, intent: out_of_scope), as well as additional examples for existing intents (e.g., catch you later, gotta go for the intent:goodbye). We also added more examples for entity values (e.g., increasing from 8 to more than 50 examples for the entity location). Adding more examples will improve performance, but there's a lot more we can do to improve entity extraction.

Using Regex in Entities

Let's start with the location entity. Users can provide this information by responding with either a city name or a zip code. Since standard U.S. zip codes follow a specific pattern - 5 digits, an easy and effective way to improve the extraction of zip codes is to allow the NLU model to use regex features. Specifying a regex for the location entity allows the model to learn that certain patterns should be associated with specific entities. This can be achieved by defining a regex for an entity location and including the pattern. The regex for extracting zip codes is entered into the nlu.md file as follows:

```
## regex:location
- [0-9]{5}
```

Using Synonyms

Another thing we can do to improve the NLU model is use **synonyms**. The medicare.gov site, where we get our facility information, uses resource codes to specify the type of medical facility.

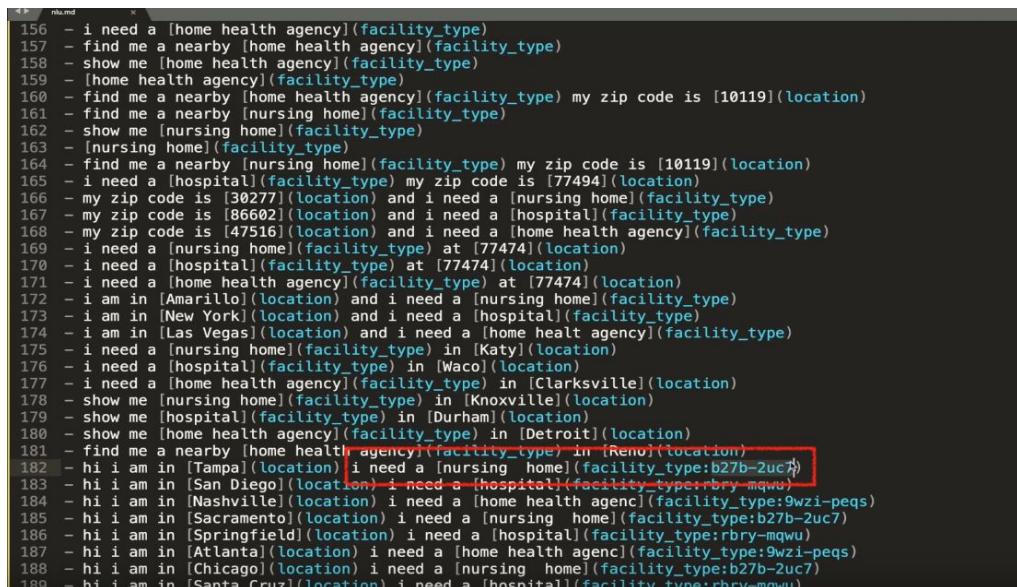
Resource code	Type of facility
xubh-q36	hospital
f7df-2ac7	Health agency
b27b-2uc7	Nursing home

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Providing the request using words rather than resource codes is more natural for a user talking to our assistant, so we will do some data normalization to make sure that the extracted values (words from users) can be used to query the database correctly (using the appropriate resource codes).

To do this, we will define synonyms that map specific values for the entity `facility_type` to corresponding resource codes. This can be achieved using a Rasa NLU feature called `synonyms`.

To define the synonym, we have to update our training data and specify the mapping between entities and synonym values. There are two ways to define a synonym. The first is within the intent itself - in this case, `## intent:search_provider`. Within the examples under this intent, we can connect the entity `facility_type` with the value `nursing home` to the specific resource code `b27b-2uc7`.



```
156 - i need a [home health agency](facility_type)
157 - find me a nearby [home health agency](facility_type)
158 - show me [home health agency](facility_type)
159 - [home health agency](facility_type)
160 - find me a nearby [home health agency](facility_type) my zip code is [10119](location)
161 - find me a nearby [nursing home](facility_type)
162 - show me [nursing home](facility_type)
163 - [nursing home](facility_type)
164 - find me a nearby [nursing home](facility_type) my zip code is [10119](location)
165 - i need a [hospital](facility_type) my zip code is [77494](location)
166 - my zip code is [30277](location) and i need a [nursing home](facility_type)
167 - my zip code is [86602](location) and i need a [hospital](facility_type)
168 - my zip code is [47516](location) and i need a [home health agency](facility_type)
169 - i need a [nursing home](facility_type) at [77474](location)
170 - i need a [hospital](facility_type) at [77474](location)
171 - i need a [home health agency](facility_type) at [77474](location)
172 - i am in [Amarillo](location) and i need a [nursing home](facility_type)
173 - i am in [New York](location) and i need a [hospital](facility_type)
174 - i am in [Las Vegas](location) and i need a [home health agency](facility_type)
175 - i need a [nursing home](facility_type) in [Katy](location)
176 - i need a [hospital](facility_type) in [Waco](location)
177 - i need a [home health agency](facility_type) in [Clarksville](location)
178 - show me [nursing home](facility_type) in [Knoxville](location)
179 - show me [hospital](facility_type) in [Durham](location)
180 - show me [home health agency](facility_type) in [Detroit](location)
181 - find me a nearby [home health agency](facility_type) in [Reno](location)
182 - hi i am in [Tampa](location) i need a [nursing home](facility_type:b27b-2uc7)
183 - hi i am in [San Diego](location) i need a [hospital](facility_type:rbrv-mqwu)
184 - hi i am in [Nashville](location) i need a [home health agency](facility_type:9wzi-peqs)
185 - hi i am in [Sacramento](location) i need a [nursing home](facility_type:b27b-2uc7)
186 - hi i am in [Springfield](location) i need a [hospital](facility_type:rbrv-mqwu)
187 - hi i am in [Atlanta](location) i need a [home health agency](facility_type:9wzi-peqs)
188 - hi i am in [Chicago](location) i need a [nursing home](facility_type:b27b-2uc7)
189 - hi i am in [Santa Cruz](location) i need a [hospital](facility_type:rbrv-mqwu)
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

You can also create a synonym directly in the nlu.md file, using the following format. This defines a specific value for the synonym (in this case, the resource code) and makes a list of possible entity values that the synonym should be mapped to.

```
## synonym:rbry-mqwu
- hospital
- hospitals
```

This synonym will make sure that whenever a `facility_type` entity is extracted with the value ‘hospital’ or ‘hospitals’, those values will be mapped to the specified resource code, to be used to query the database later on. We will do the same with other entity values and resource codes.

```
293
294 ## synonym:xubh-q36u
295 - hospital
296 - hospitals
297
298 ## synonym:9wzi-peqs
299 - home health agency
300 - home health agencies
301
302 ## synonym:b27b-2uc7
303 - nursing home
304 - nursing homes
```

Lastly, to ensure that the NLU model learns to map the synonyms, we need to include the component `EntitySynonymMapper` in our pipeline, in the config.yml file. Going forward, we will be using the Supervised Embeddings Pipeline for our model, and this pipeline (by default) includes the `EntitySynonymMapper` component in its configuration.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

```
config.yml
1 # Configuration for Rasa NLU.
2 # https://rasa.com/docs/rasa/nlu/components/
3 language: "en"
4
5 pipeline:
6   - name: "WhitespaceTokenizer"
7   - name: "RegexFeaturizer"
8   - name: "CRFEntityExtractor"
9   - name: "EntitySynonymMapper" highlighted
10  - name: "CountVectorsFeaturizer"
11  - name: "CountVectorsFeaturizer"
12  - name: "EmbeddingIntentClassifier"
13
14 # Configuration for Rasa Core.
15 # https://rasa.com/docs/rasa/core/policies/
16 policies:
17   - name: MemoizationPolicy
18   - name: KerasPolicy
19   - name: MappingPolicy
20   - name: EmbeddingPolicy
21
22
23
```

Retraining the NLU Model

Since we've added a number of items to the NLU model, let's retrain the model to see how it works. Once again, we will use the `rasa train nlu` command to train the model, and the `rasa shell nlu` command to talk to our model.

If we type `I need a hospital`, we can see the model correctly classifies this input as `intent: search_provider`. Even better, take a look at the extracted entities. The entity `hospital` was identified, but the extracted value was a medicare.gov resource code, which was mapped to the entity value `hospital` using the synonym feature.

```
(base) BERMB00011:rasabot juste$ rasa shell nlu
NLU model loaded. Type a message and press enter to parse it.
New message:
I need a hospital
{
  "intent": {
    "name": "search_provider",
    "confidence": 0.9695162773132324
  },
  "entities": [
    {
      "start": 9,
      "end": 17,
      "value": "xuh-q36u",
      "entity": "facility_type",
      "confidence": 0.9933899593082041,
      "extractor": "CRFEntityExtractor",
      "processors": [
        "EntitySynonymMapper"
      ]
    }
  ],
  "intent_ranking": [
    {
      "name": "search_provider",
      "confidence": 0.9695162773132324
    },
    {
      "name": "inform",
      "confidence": 0.0442807674407959
    }
  ]
}
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Implementing a Form Action in Rasa

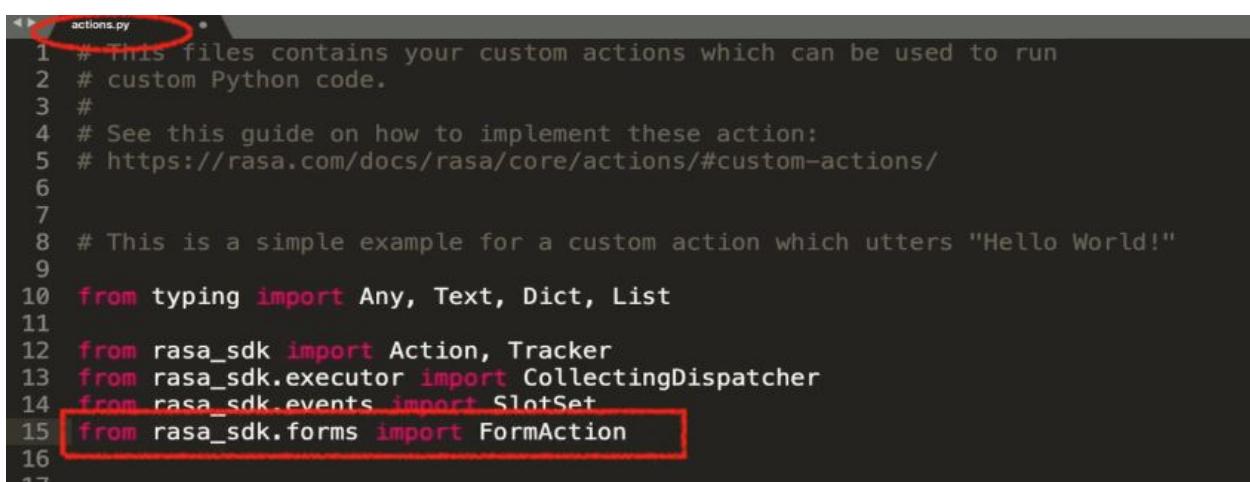
Next, we will improve the dialogue management in our assistant by using the Form Action in Rasa. In previous episodes, we built a simple dialogue management model capable of using slots. The model is capable of driving the conversation based on whether or not the user provided specific details like location or facility type.

Going forward, we will be using the database from medicare.gov to supply information about facility locations. We want the medicare locator assistant to query this database when the user asks the assistant for information about a specific health facility, via an action. To make sure that the API call to the database is correct, we need to make sure that our assistant collects and uses the details of `location` and `facility_type` *before* the call is made. This situation is a perfect use case for Rasa forms - a component which can be used to ensure the assistant collects the necessary details *before* running a specific action.

Defining a Form Action

Form actions have to be defined in the project's `actions.py` file. All the work in this section will be within this file.

First, we need to import the `FormAction` method from the Rasa SDK. Form action is a class, just like other custom actions in Rasa and so it follows a similar overall structure.



```
actions.py
1 # This files contains your custom actions which can be used to run
2 # custom Python code.
3 #
4 # See this guide on how to implement these action:
5 # https://rasa.com/docs/rasa/core/actions/#custom-actions/
6 #
7 # This is a simple example for a custom action which utters "Hello World!"
8
9
10 from typing import Any, Text, Dict, List
11
12 from rasa_sdk import Action, Tracker
13 from rasa_sdk.executor import CollectingDispatcher
14 from rasa_sdk.events import SlotSet
15 from rasa_sdk.forms import FormAction
16
17
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Forms generally consist of a few main functions - name, required slots, slot mappings, and submit.

The function called `name` is very simple - it defines the name of the form action. When an action `facility_form` is predicted, Rasa knows that it should run the code defined in this class `FacilityForm(FormAction)`:

```
class FacilityForm(FormAction):
    """Custom form action to fill all slots required to find specific type
    of healthcare facilities in a certain city or zip code."""

    def name(self) -> Text:
        """Unique identifier of the form"""

        return "facility_form"

    @staticmethod
    def required_slots(tracker: Tracker) -> List[Text]:
        """A list of required slots that the form has to fill"""

        return ["facility_type", "location"]

    def slot_mappings(self) -> Dict[Text, Any]:
        return {"facility_type": self.from_entity(entity="facility_type",
                                                   intent=[{"inform",
                                                       "search_provider"}]),
                "location": self.from_entity(entity="location",
                                             intent=[{"inform",
                                                       "search_provider"}])}

    def submit(self,
              dispatcher: CollectingDispatcher,
              tracker: Tracker,
              domain: Dict[Text, Any]
              domain: Dict[Text, Any]
```

All form actions must have a method called `required_slots`. This method is used to define which slots must be filled in before an assistant can continue with the dialogue. In the conversation with the user, as long as there are unfilled slots, the model will continue predicting `FormAction`, and the assistant will continue to ask the user for information to fill in the details, until all the slots are filled. In the case of our medicare locator, we need two slots to be provided for an assistant to query the database: `location` and `facility_type`.

```
class FacilityForm(FormAction):
    """Custom form action to fill all slots required to find specific type
    of healthcare facilities in a certain city or zip code."""

    def name(self) -> Text:
        """Unique identifier of the form"""

        return "facility_form"

    @staticmethod
    def required_slots(tracker: Tracker) -> List[Text]:
        """A list of required slots that the form has to fill"""

        return ["facility_type", "location"]

    def slot_mappings(self) -> Dict[Text, Any]:
        return {"facility_type": self.from_entity(entity="facility_type",
                                                   intent=[{"inform",
                                                       "search_provider"}]),
                "location": self.from_entity(entity="location",
                                             intent=[{"inform",
                                                       "search_provider"}])}

    def submit(self,
              dispatcher: CollectingDispatcher,
              tracker: Tracker,
              domain: Dict[Text, Any]
              domain: Dict[Text, Any]
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

`slot_mappings` is an optional method used with Form Action, but it's a very useful one. Sometimes, required slots come from very different user inputs which, naturally, will have different intent or entity labels. By default, Form Action will fill in required slots using only the values extracted from intents or entities with exactly the same name as the slot. Slot mapping allows you to define how the values from other intents and entities can be mapped to the required slots. In our case, the names of the required slots match the names of the corresponding entities. We can use `slot_mappings` to specify which intents those values come from. For example, the slot `location` can be filled using the entity values from two different intents: `inform` and `search_provider`.

```
class FacilityForm(FormAction):
    """Custom form action to fill all slots required to find specific type
    of healthcare facilities in a certain city or zip code."""

    def name(self) -> Text:
        """Unique identifier of the form"""

        return "facility_form"

    @staticmethod
    def required_slots(tracker: Tracker) -> List[Text]:
        """A list of required slots that the form has to fill"""

        return ["facility_type", "location"]

    def slot_mappings(self) -> Dict[Text, Any]:
        return {"facility_type": self.from_entity(entity="facility_type",
                                                intent=["inform",
                                                        "search_provider"]),
                "location": self.from_entity(entity="location",
                                             intent=["inform",
                                                     "search_provider"])}

    def submit(self,
              dispatcher: CollectingDispatcher,
              tracker: Tracker,
              domain: Dict[Text, Any]):
```

All form actions must have a method called `required_slots`. This method is used to define which slots must be filled in before an assistant can continue with the dialogue. In the conversation with the user, as long as there are unfilled slots, the model will continue predicting `FormAction`, and the assistant will continue to ask the user for information to fill in the details, until all the slots are filled. In the case of our medicare locator, we need two slots to be provided for an assistant to query the database: `location` and `facility_type`.

Finally, we need to define what should happen when all required slots are filled. This is specified in the `submit` method. In the medicare locator, we want our assistant to send the facility query to medicare.gov database using the provided details and return the options to the user.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Let's walk through the code of the `submit` method in detail.

With the `tracker.get_slot` method, our assistant will pull the current values of the `location` and `facility_type` slots, and use them to call the (still-to-be-created) custom action `find_facilities`. (We will create the `find_facilities` action in a later masterclass.)

```
def submit(self,
           dispatcher: CollectingDispatcher,
           tracker: Tracker,
           domain: Dict[Text, Any]
           ) -> List[Dict]:
    """Once required slots are filled, print buttons for found facilities"""

    location = tracker.get_slot('location')
    facility_type = tracker.get_slot('facility_type')

    results = _find_facilities(location, facility_type)
    button_name = _resolve_name(FACILITY_TYPES, facility_type)
    if len(results) == 0:
        dispatcher.utter_message(
            "Sorry, we could not find a {} in {}".format(button_name,
                                                          location.title()))
    return []
```

If no results were found by `find_facilities`, the assistant will send a message to the user saying the requested facility could not be found in the specified location.

```
def submit(self,
           dispatcher: CollectingDispatcher,
           tracker: Tracker,
           domain: Dict[Text, Any]
           ) -> List[Dict]:
    """Once required slots are filled, print buttons for found facilities"""

    location = tracker.get_slot('location')
    facility_type = tracker.get_slot('facility_type')

    results = _find_facilities(location, facility_type)
    button_name = _resolve_name(FACILITY_TYPES, facility_type)
    if len(results) == 0:
        dispatcher.utter_message(
            "Sorry, we could not find a {} in {}".format(button_name,
                                                          location.title()))
    return []
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

If `find_facilities` returns results, the assistant will format the first three returned facilities as buttons, to be returned to the user. The message to the user is sent using the Rasa dispatcher method.

```
        # limit number of results to 3 for clear presentation purposes
    for r in results[:3]:
        if facility_type == FACILITY_TYPES["hospital"]["resource"]:
            facility_id = r.get("provider_id")
            name = r["hospital_name"]
        elif facility_type == FACILITY_TYPES["nursing_home"]["resource"]:
            facility_id = r["federal_provider_number"]
            name = r["provider_name"]
        else:
            facility_id = r["provider_number"]
            name = r["provider_name"]

        payload = "/inform{{\"facility_id\":\"{}\" + facility_id + \"\"}}"
        buttons.append(
            {"title": "{}".format(name.title()), "payload": payload})

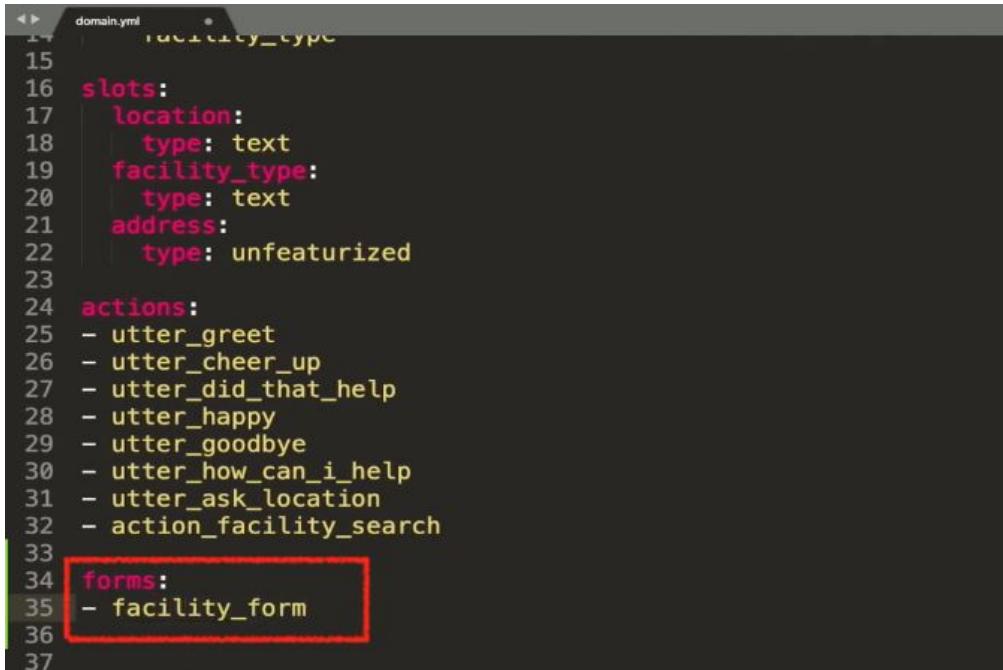
    if len(buttons) == 1:
        message = "Here is a {} near you:".format(button_name)
    else:
        if button_name == "home health agency":
            button_name = "home health agencie"
        message = "Here are {} {}s near you:".format(len(buttons),
                                                     button_name)

    dispatcher.utter_button_message(message, buttons)

    return []
    return []
```

Updating the Domain File and Model Configuration for Forms

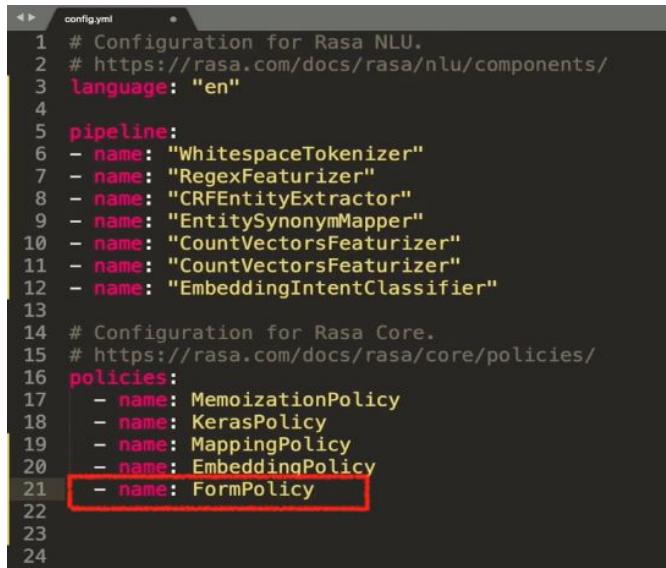
For our assistant to use this newly created form, it has to be included in the domain file.



```
domain.yml
15
16 slots:
17   location:
18     type: text
19   facility_type:
20     type: text
21   address:
22     type: unfeaturized
23
24 actions:
25 - utter_greet
26 - utter_cheer_up
27 - utter_did_that_help
28 - utter_happy
29 - utter_goodbye
30 - utter_how_can_i_help
31 - utter_ask_location
32 - action_facility_search
33
34 forms:
35 - facility_form
36
37
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

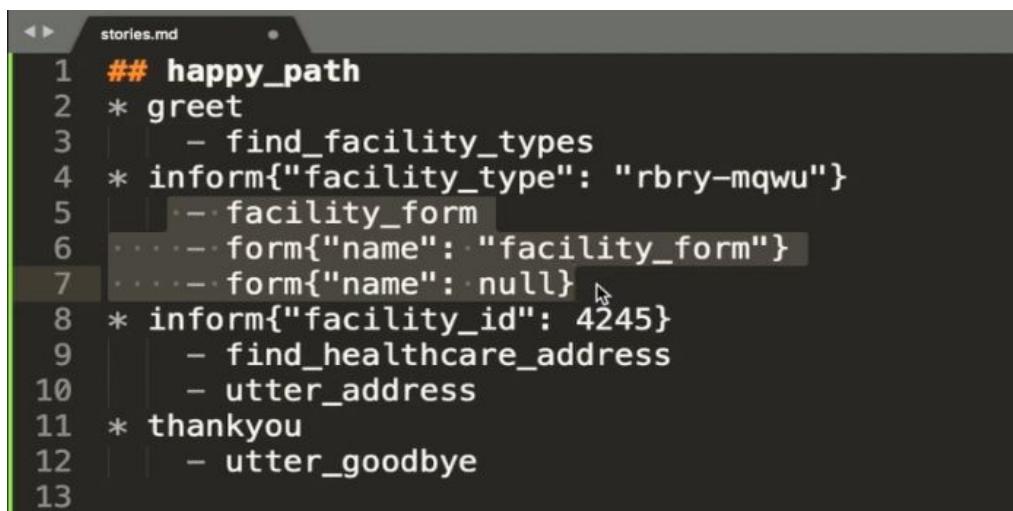
We also need to update the model config.yml file to include `FormPolicy`, an extension of the `MemoizationPolicy` that handles the filling of forms. `FormPolicy` will predict the new `FormAction` until all the required slots in the form are filled.



```
config.yml
1 # Configuration for Rasa NLU.
2 # https://rasa.com/docs/rasa/nlu/components/
3 language: "en"
4
5 pipeline:
6 - name: "WhitespaceTokenizer"
7 - name: "RegexFeaturizer"
8 - name: "CRFEntityExtractor"
9 - name: "EntitySynonymMapper"
10 - name: "CountVectorsFeaturizer"
11 - name: "CountVectorsFeaturizer"
12 - name: "EmbeddingIntentClassifier"
13
14 # Configuration for Rasa Core.
15 # https://rasa.com/docs/rasa/core/policies/
16 policies:
17 - name: MemoizationPolicy
18 - name: KerasPolicy
19 - name: MappingPolicy
20 - name: EmbeddingPolicy
21 - name: FormPolicy
22
23
24
```

Updating Training Stories with Forms

Lastly, we need to update the stories.md file with the form action that we created, `facility_form`.



```
stories.md
1 ## happy_path
2 * greet
3   - find_facility_types
4 * inform{"facility_type": "rbry-mqwu"}
5   -- facility_form
6   ... -- form{"name": "facility_form"}
7   ... -- form{"name": null} -->
8 * inform{"facility_id": 4245}
9   - find_healthcare_address
10  - utter_address
11 * thankyou
12   - utter_goodbye
13
```

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

We add three lines, which represent three actions that should happen when the user chooses a facility type for the medicare locator assistant to find:

- `facility_form` tells the model to activate the form, to begin gathering information to complete the form
- `form{"name": "facility_form"}` indicates that the FormPolicy will run the form action until all slots are filled
- `form{"name": "null"}` ends this small story, and indicates that the form is filled and the assistant can move on with the conversation. These three lines are enough to allow our assistant to handle all the happy paths the user might take when filling in the form. In this case, by “happy path”, we mean that whenever the assistant asks the user for some information, they eventually respond with the information you asked for. There are many paths for this to happen - situations where the user provides one detail but not the other, or responds with all details at once, or even when the user doesn’t provide any of the required information.

Let’s add another story which will teach our assistant how to respond when the user provides all information with their initial request.

```
14
15 ## happy_path2
16 * search_provider{"location": "Austin", "facility_type": "rbry-mqwu"}
17   - facility_form
18   - form{"name": "facility_form"}
19   - form{"name": null}
20 * inform{"facility_id": "450871"}
21   - find_healthcare_address
22   - utter_address
23 * thankyou
24   - utter_noworries
25
```

We also want our medicare locator assistant to be able to handle situations in which a user asks for multiple recommendations. A user may ask additional questions after the assistant has provided information to answer an initial question. We can enable our assistant to handle this behavior by adding stories with more dialogue turns, like the following:

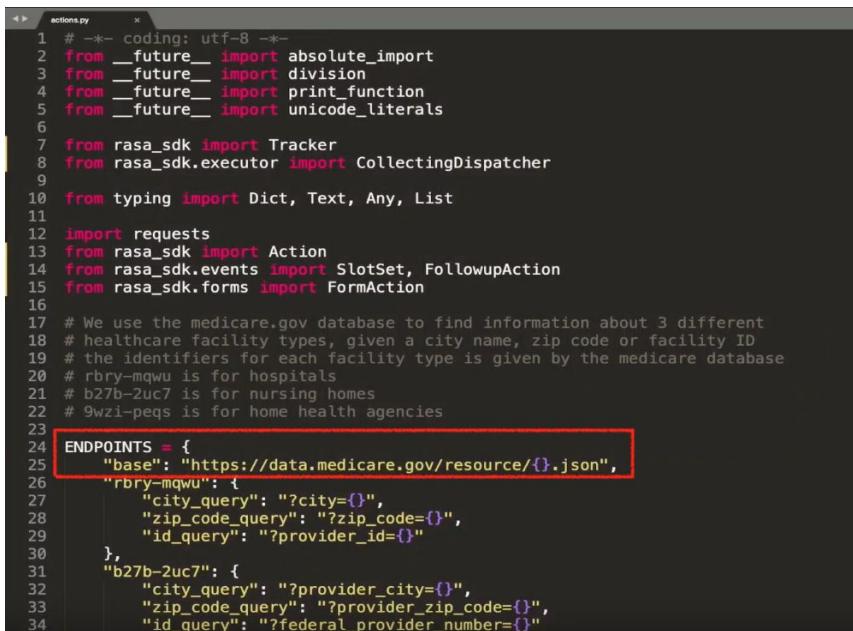
```
## happy_path_multi_requests
* greet
  - find_facility_types
* inform{"facility_type": "xubh-q36u"}
  - facility_form
  - form{"name": "facility_form"}
  - form{"name": null}
* inform{"facility_id": "747604"}
  - find_healthcare_address
  - utter_address
* search_provider{"facility_type": "xubh-q36u"}
  - facility_form
  - form{"name": "facility_form"}
  - form{"name": null}
* inform{"facility_id": 4245}
  - find_healthcare_address
  - utter_address
```

<!-- Story with multiple dialogue turns -->
<!-- User says hello, assistant displays buttons with facility types, for user to choose from -->
<!-- User chooses hospital, assistant initiates form action to gather info and return hospital options -->
<!-- User chooses specific hospital, assistant finds address & shares -->
<!-- User asks to find a hospital, assistant starts form action, returns hospital options -->
<!-- User chooses different hospital, assistant finds address & shares -->

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

With forms, we enabled our assistant to collect some crucial details needed to run the facility search. Now, let's update the `facility_search` custom action to enable our medicare assistant to use the extracted details, run the backend integrations, and return the requested information.

Let's go back to our `actions.py` file.

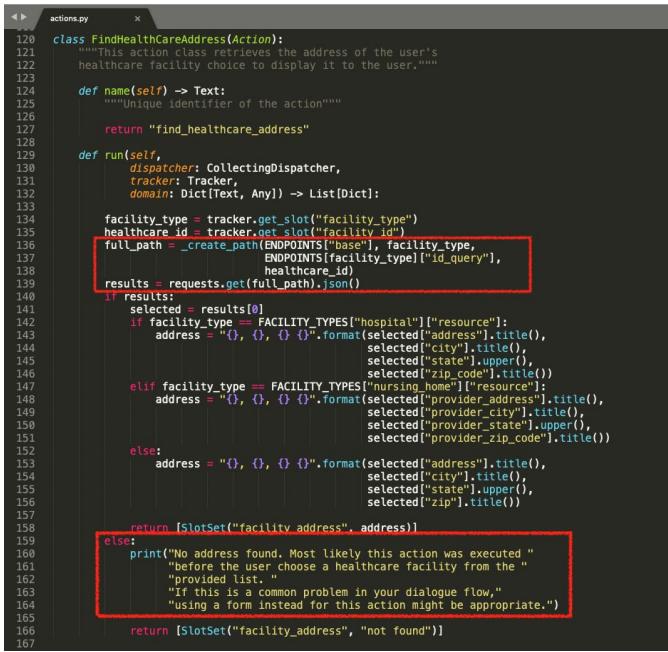


```
actions.py
1 # -*- coding: utf-8 -*-
2 from __future__ import absolute_import
3 from __future__ import division
4 from __future__ import print_function
5 from __future__ import unicode_literals
6
7 from rasa_sdk import Tracker
8 from rasa_sdk.executor import CollectingDispatcher
9
10 from typing import Dict, Text, Any, List
11
12 import requests
13 from rasa_sdk import Action
14 from rasa_sdk.events import SlotSet, FollowupAction
15 from rasa_sdk.forms import FormAction
16
17 # We use the medicare.gov database to find information about 3 different
18 # healthcare facility types, given a city name, zip code or facility ID
19 # the identifiers for each facility type is given by the medicare database
20 # rbry-mqwu is for hospitals
21 # b27b-2uc7 is for nursing homes
22 # 9wzi-peqs is for home health agencies
23
24 ENDPOINTS = {
25     "base": "https://data.medicare.gov/resource/{}.json",
26     "rbry-mqwu": {
27         "city_query": "?city={}",
28         "zip_code_query": "?zip_code={}",
29         "id_query": "?provider_id={}"
30     },
31     "b27b-2uc7": {
32         "city_query": "?provider_city={}",
33         "zip_code_query": "?provider_zip_code={}",
34         "id_query": "?federal_provider_number={}"
35 }
```

Custom actions with real backend integrations usually consist of quite a few methods and they will highly depend on what database or API you are using. In the medicare locator, we have quite a few helper methods - for example, defining the endpoints for our APIs.

The most important part of our `actions.py` file is a function called `FindHealthCareAddress`. This function retrieves the current slot values for key parameters, and uses them to create a full path for an endpoint. Our assistant will send a request to this endpoint to retrieve the data - the health care facility address. A method `requests.get` sends a GET request to the API, which returns the data in JSON format. If any results were returned, our assistant returns a response, based on what was requested. If no results were returned, the assistant sends a message back to the user suggesting what went wrong.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS



```
actions.py
120 class FindHealthCareAddress(Action):
121     """This action class retrieves the address of the user's
122     healthcare facility choice to display it to the user."""
123
124     def name(self) -> Text:
125         """Unique identifier of the action"""
126
127         return "find_healthcare_address"
128
129     def run(self,
130             dispatcher: CollectingDispatcher,
131             tracker: Tracker,
132             domain: Dict[Text, Any]) -> List[Dict]:
133
134         facility_type = tracker.get_slot("facility_type")
135         healthcare_id = tracker.get_slot("facility_id")
136
137         full_path = _create_path(ENDPOINTS["base"], facility_type,
138                                 ENDPOINTS[facility_type]["id_query"],
139                                 healthcare_id)
140
141         results = requests.get(full_path).json()
142
143         if results:
144             selected = results[0]
145
146             if facility_type == FACILITY_TYPES["hospital"]["resource"]:
147                 address = "{}; {}; {}".format(selected["address"].title(),
148                                              selected["city"].title(),
149                                              selected["state"].upper(),
150                                              selected["zip_code"].title())
151
152             elif facility_type == FACILITY_TYPES["nursing_home"]["resource"]:
153                 address = "{}; {}; {}; {}".format(selected["provider_address"].title(),
154                                              selected["provider_city"].title(),
155                                              selected["provider_state"].upper(),
156                                              selected["provider_zip_code"].title())
157
158             else:
159                 address = "{}; {}; {}".format(selected["address"].title(),
160                                              selected["city"].title(),
161                                              selected["state"].upper(),
162                                              selected["zip"].title())
163
164
165             return [SlotSet("facility_address", address)]
166
167         else:
168             print("No address found. Most likely this action was executed "
169                  "before the user choose a healthcare facility from the "
170                  "provided list."
171                  "\nIf this is a common problem in your dialogue flow,"
172                  "\nusing a form instead for this action might be appropriate.")
173
174
175         return [SlotSet("facility_address", "not found")]
176
```

Failing Gracefully in Rasa

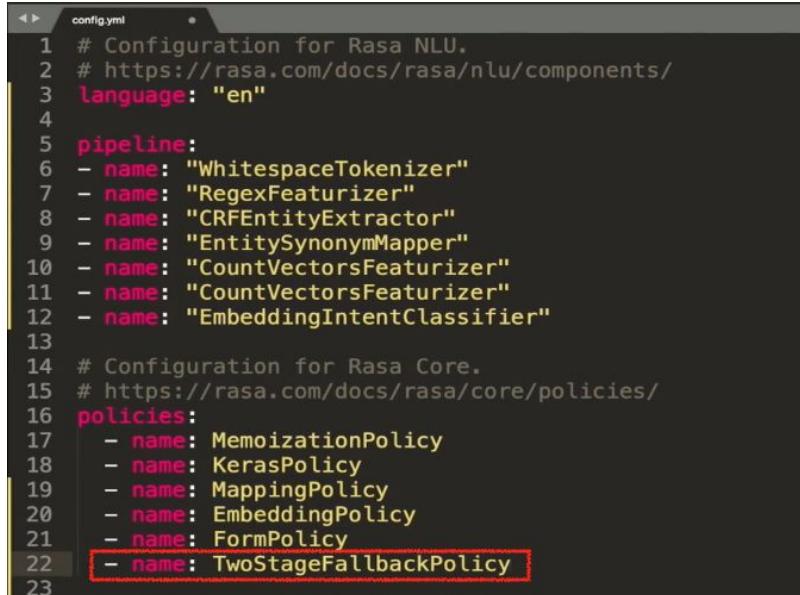
We've updated our assistant to be better at intent classification and entity extraction, implemented form actions, and added a custom action for the facility search, with some backend integrations.

Another valuable improvement we can make to our assistant is to implement a fallback policy. The fallback policy helps to make sure that if our assistant makes a mistake, it handles the situation gracefully.

As we discussed in the [Rasa Masterclass Episode #7](#), there are two fallback policies in Rasa: the fallback policy and the two-stage fallback policy. For our assistant, we will implement the `TwoStageFallbackPolicy`.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

We add the `TwoStageFallbackPolicy` to our assistant in the `config.yml` file, by listing this policy as one of the components of the policy configuration. We will keep the default hyperparameters of this policy for now.



```
1 # Configuration for Rasa NLU.
2 # https://rasa.com/docs/rasa/nlu/components/
3 language: "en"
4
5 pipeline:
6 - name: "WhitespaceTokenizer"
7 - name: "RegexFeaturizer"
8 - name: "CRFEntityExtractor"
9 - name: "EntitySynonymMapper"
10 - name: "CountVectorsFeaturizer"
11 - name: "CountVectorsFeaturizer"
12 - name: "EmbeddingIntentClassifier"
13
14 # Configuration for Rasa Core.
15 # https://rasa.com/docs/rasa/core/policies/
16 policies:
17 - name: MemoizationPolicy
18 - name: KerasPolicy
19 - name: MappingPolicy
20 - name: EmbeddingPolicy
21 - name: FormPolicy
22 - name: TwoStageFallbackPolicy
23
```

Since we've made many additions to the model, it would be a good time to re-train the models and test how they work. Again, we use the command line function `rasa train`, and after the models are trained, we can load the assistant using the functions `rasa run actions` and `rasa shell`. Then we can have a conversation with the newly trained assistant.



```
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input -> Hello
127.0.0.1 - - [2019-11-27 19:26:56] "POST /webhook HTTP/1.1" 200 447 0.000740
? Hey there! Please choose one of the healthcare facility options: 1: Hospital (/inform{"facility_type": "rbry-mqwu"})
127.0.0.1 - - [2019-11-27 19:26:58] "POST /webhook HTTP/1.1" 200 599 0.001052
Please enter your zip code or city name to find local providers.
Your input -> San Francisco
127.0.0.1 - - [2019-11-27 19:27:03] "POST /webhook HTTP/1.1" 200 657 1.340564
? Here are 3 hospitals near you: 1: St Mary's Medical Center (/inform{"facility_id": "050457"})
127.0.0.1 - - [2019-11-27 19:27:06] "POST /webhook HTTP/1.1" 200 279 0.328426
The address is 450 Stanyan St, San Francisco, CA 94117.
```

Our assistant can now pull real-world data and use it to respond to users' queries.

CHAPTER 7: INTEGRATIONS, FALLBACKS, FORMS

Conclusion

So after all this hard work, our assistant improved quite a bit - we managed to improve the performance of the NLU model, we implemented the form action and quite an advanced custom action with back-end integrations, and we even created a fallback policy.

In future episodes of the Masterclass, we will expand on these updates and work on adding new features and skills to our assistant. We hope you will implement all the features that you learned in this episode of the Rasa Masterclass, to improve your custom assistants. See you in the next episode!

Additional Resources

- [Ep. #6 - Rasa Masterclass - Dialogue management with domains, custom actions, and slots](#) (YouTube)
- [Ep. #7 - Rasa Masterclass - Dialogue Policies](#)
- [Form Basics](#) (Rasa docs)
- [Custom Actions](#) (Rasa docs)
- [Fallback Actions](#) (Rasa docs)



CHAPTER NINE

IMPROVING THE ASSISTANT

IMPROVING THE ASSISTANT



Introduction

In [Episode 9](#) of the Rasa Masterclass, we introduce Rasa X, a toolset for improving AI assistants. We'll discuss why you should use Rasa X, demonstrate how to deploy Rasa X to a server, and connect the Rasa X instance to the medicare locator assistant.

Before beginning this tutorial, use the exercises in the [previous 8 episodes](#) to build the medicare locator assistant, a contextual assistant that can find the addresses of nearby hospitals, home health agencies, and nursing homes. You can also fast track by downloading the [completed medicare locator assistant code on GitHub](#).

To follow along, you'll need to [push your medicare locator project files to GitHub](#).

CHAPTER 9: IMPROVING THE ASSISTANT

The challenge: getting good training data is hard

Up until this point, we've been [creating training data](#) for the medicare locator assistant by imagining things users might say and writing them down as training examples.

That's a fine way to start, but it has some major limitations. The data set we've generated is pretty small, and it's inherently biased—our real users are likely to say things to the assistant that we didn't anticipate. Because of this, the assistant will have a hard time generalizing. It can handle utterances and story paths that are pretty close to what we have in our training data, but it's likely to fall down if presented with an unexpected user message.

The best training data we could use would be a large data set made up of real conversations. There are publicly available conversational data sets, but they tend to be general purpose: they're not conversations with *your users*, and they don't relate to *your domain*.

Rasa X solves this problem by helping you build the ideal data set: a large body of conversations that have happened between your users and your assistant. It allows you to share the assistant with test users to collect their conversations, and later on, when the assistant has been deployed, collect conversations from real users. Rasa X converts these conversations into high quality data which can be used to re-train and improve the assistant.

It's important to get your assistant into the hands of real users to start collecting this training data as soon as possible. The perfect time to start using Rasa X is when you've built an assistant that can handle what we call the most basic happy paths, meaning the assistant can handle some basic conversations and accomplish its main purpose: in our case, to look up medical facilities in the United States. The medicare locator is still a simple application at this point, but it's far enough along to start benefiting from real-world data.

What is Rasa X?

Rasa X is a UI tool for developers, used to improve assistants built with Rasa Open Source. It's intended to solve two problems:

- First, to make it easier to leverage real conversations as training data.
- Second, to provide a way to review past conversations for patterns or errors.

CHAPTER 9: IMPROVING THE ASSISTANT

Without a UI tool, these workflows can be challenging. Rasa X collects conversations with users so you can review how they went and make decisions about the best way to improve the assistant. For example, if you see a conversation where the assistant performed particularly well, you can save it directly to your training stories. If you see an intent that was mis-classified, you can correct the classification and save the annotated data to your training file.

But, not every type of update should be made directly in Rasa X. While reviewing past conversations, you might conclude that two intents should be merged into one or the behavior of a custom action needs to change. In these cases, Rasa X is a useful tool for identifying which changes need to happen, but it's easier to make major updates like these in a text editor on your local machine, rather than in Rasa X.

Deploying Rasa X

Now that we've discussed why Rasa X is an important part of the assistant-building workflow, let's move on to installation. The proper way to run Rasa X is to deploy it to a server. It's possible to run it in local mode, but in order to really start collecting conversations with users, Rasa X needs to be accessible on the public internet and running at all times.

In this tutorial, we'll be using Google Cloud Platform (GCP). You can sign up for a GCP account (and get \$300 free credit) [here](#).

It's also possible to deploy Rasa X to other hosting services, like Azure, Digital Ocean, AWS, or to your own dedicated server. You'll just want to make sure your machine meets the minimum requirements:

- At least 4GB RAM
- 2-6 vCPUs
- 100 GB disc space
- Linux distribution that can run Docker (e.g. Debian 9, Ubuntu 16.04 / 18.04)

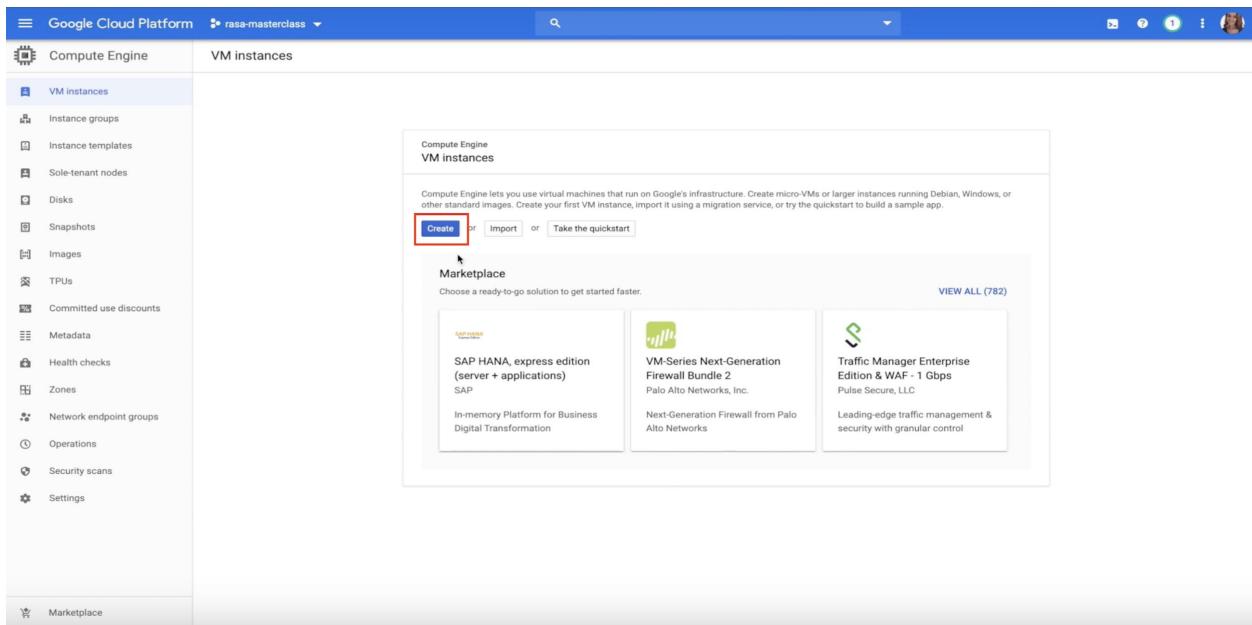
Docker-compose

The most lightweight method for installing Rasa X is to use docker-compose, which is the method we'll be using today. For larger-scale assistants running in production, Rasa X also provides Helm charts for deploying to Kubernetes/OpenShift. We'll cover cluster deployments in a later episode of the Masterclass.

CHAPTER 9: IMPROVING THE ASSISTANT

Configure the VM instance

Log in to your GCP Console and navigate to Compute Engine>VM instances. Click Create.



On the configuration page, specify a name for the instance. We'll call ours rasax-server. Then, choose a region near where you are located. In our case, that's Belgium.

Under Machine configuration>Machine type, choose the standard option with 2 CPUs: n1-standard-2 (2 vCPU, 7.5 GB memory).

CHAPTER 9: IMPROVING THE ASSISTANT

[← Create an instance](#)

To create a VM instance, select one of the options:

- New VM instance**
Create a single VM instance from scratch
- New VM instance from template**
Create a single VM instance from an existing template
- Marketplace**
Deploy a ready-to-go solution onto a VM instance

Name ?
Name is permanent

Region ?
Region is permanent

Zone ?
Zone is permanent

Machine configuration ?

Machine family
 General-purpose Memory-optimized

Machine types for common workloads, optimized for cost and flexibility

Series

Powered by Intel Skylake CPU platform or one of its predecessors

Machine type

	vCPU	Memory
	2	7.5 GB

[▼ CPU platform and GPU](#)

Next, specify the boot disk. You'll need to be running a modern Linux distribution that can run Docker, like Debian 9 or Ubuntu 16.04 / 18.04. We'll go with Ubuntu 16.04 LTS. Increase the disc size to 100 GB.

Boot disk

Select an image or snapshot to create a boot disk; or attach an existing disk. Can't find what you're looking for? Explore hundreds of VM solutions in [Marketplace](#).

Public images Custom images Snapshots Existing disks

Show images with Shielded VM features ?

Operating system

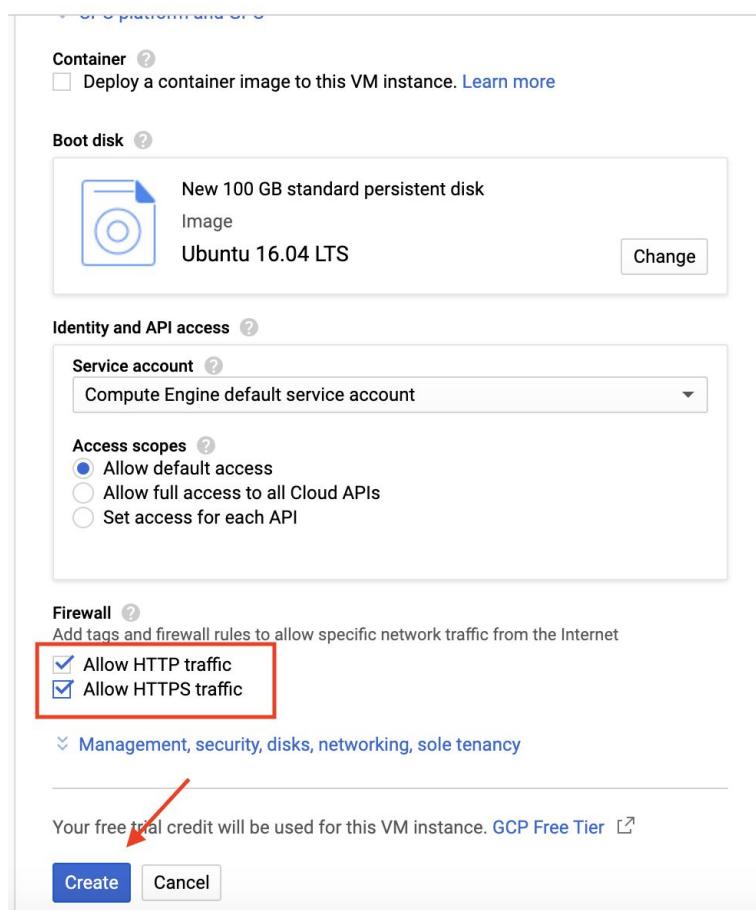
Version

amd64 xenial image built on 2020-01-08

Boot disk type ? **Size (GB)** ?

CHAPTER 9: IMPROVING THE ASSISTANT

Finally, under the Firewall settings, check the boxes to allow both HTTP and HTTPS traffic, and click Create.

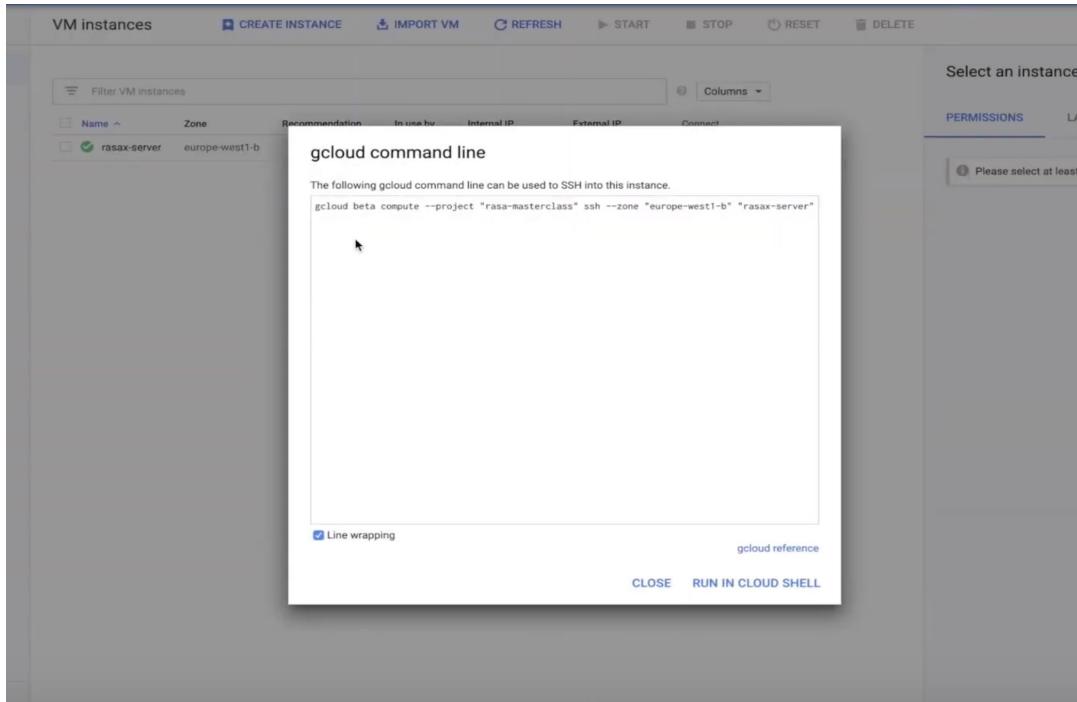


Install Rasa X

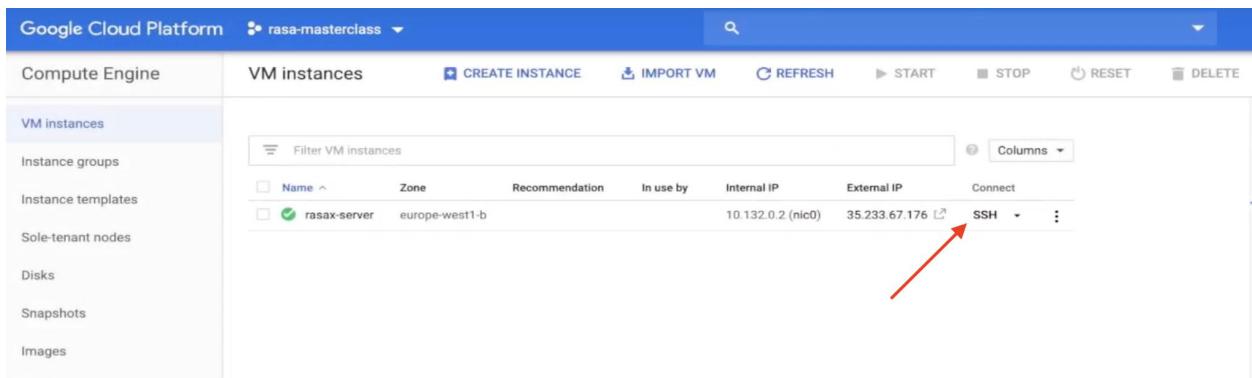
We've created our VM, but right now, it's empty. Let's connect to the new instance using SSH, so we can download and install Rasa.

If you've installed the [Google Cloud SDK](#), you can run the following command in your terminal to connect to the VM instance.

CHAPTER 9: IMPROVING THE ASSISTANT



Otherwise, you can click the SSH button next to your newly created instance to start a new shell session and connect.



In your terminal, run the install script using the following commands. First, run:

```
curl -sSL -o install.sh  
https://storage.googleapis.com/rasa-x-releases/0.23.3/install.sh
```

CHAPTER 9: IMPROVING THE ASSISTANT

Note that we're specifying version 0.23.3. At the time this video was made, this is the latest version. Be sure to update to the most recent Rasa X version number if you're watching at a later date.

Then, run:

```
sudo bash ./install.sh
```

```
ability to manage packages. You may install the locales by running:  
  sudo apt-get install language-pack-UTF-8  
  or  
  sudo locale-gen UTF-8  
  
To see all available language packs, run:  
  apt-cache search "^language-pack-[a-z][a-z]$"  
To disable this message for all users, run:  
  sudo touch /var/lib/cloud/instance/locale-check.skip  
  
juste@rasax--server:~$ curl -sSL -o install.sh https://storage.googleapis.com/rasa-x-releases/0.23.3/install.sh  
juste@rasax--server:~$ sudo bash ./install.sh  
Installing pip and ansible  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
python3 is already the newest version (3.5.1-3).  
The following package was automatically installed and is no longer required:  
  grub-pc-bin  
Use 'sudo apt autoremove' to remove it.  
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.  
% Total    % Received % Xferd  Average Speed   Time   Time  Current  
          Dload  Upload Total Spent   Left  Speed  
0     0    0     0      0      0      0      0      0      0
```

The installation will take a few minutes to complete. Agree to the terms and conditions when prompted in the terminal.

By default, the installation folder for Rasa X is the `/etc/rasa` directory. Navigate into this directory using the following command:

```
cd /etc/rasa
```

We can run the `ls` command to take a look at the files contained in this directory.

```
juste@rasax--server:~$ cd /etc/rasa  
juste@rasax--server:/etc/rasa$ ls  
auth certs credentials.yml db docker-compose.yml endpoints.yml environments.yml logs models rasa_x_commands.py scripts terms
```

We see everything Rasa X needs to run properly: credentials, the models directory, a database file (which will be used to store the conversations between users and the assistant), container logs, and a docker-compose file.

Let's start Rasa X with the following command:

```
sudo docker-compose up -d
```

CHAPTER 9: IMPROVING THE ASSISTANT

Once the container is up and running, set the password for the admin user with the following command. In the example below, we're setting the password to rasarasa.

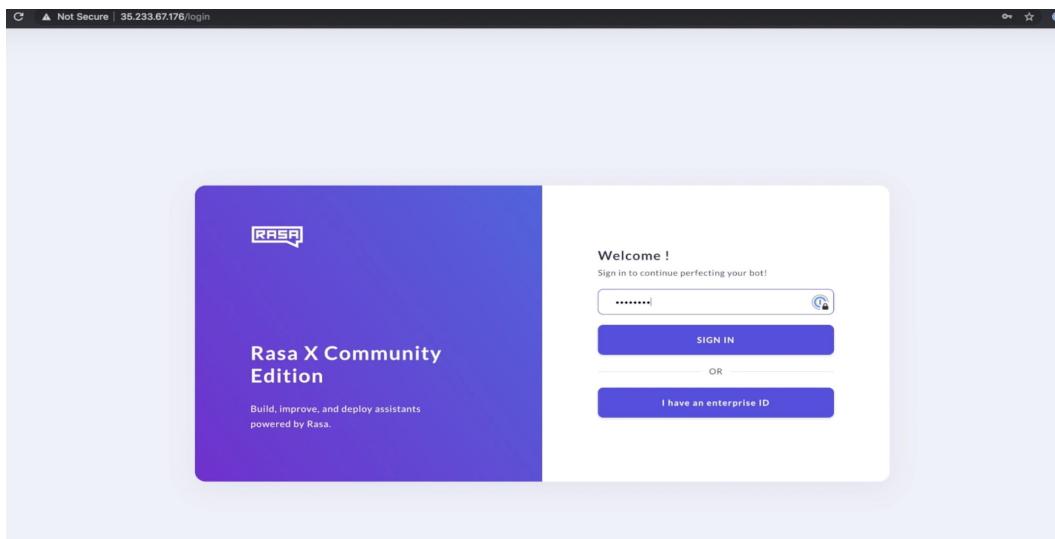
```
sudo python rasa_x_commands.py create --update admin me rasarasa
```

Launch Rasa X

We're now ready to launch Rasa X in the browser. Head back to your GCP console and click the external IP address for your VM instance.

The screenshot shows the 'VM instances' section of the GCP console. A table lists one instance named 'rasax-server' in the 'europe-west1-b' zone. The 'External IP' column for this instance contains '35.233.67.176'. This IP address is highlighted with a red rectangular box. The 'Connect' column for the same row has an 'SSH' button, which is also highlighted with a red box. Other columns in the table include 'Name', 'Zone', 'Recommendation', 'In use by', 'Internal IP', and 'External IP'.

At first, the page will appear to be broken, but not to worry—that's expected at this stage. By default, the link opens with HTTPS, but we haven't installed an SSL certificate on the server. We'll change the URL to http://<your_External_IP> in the address bar for now. Now we see the Rasa X login page and can enter the password we just set for the admin user.

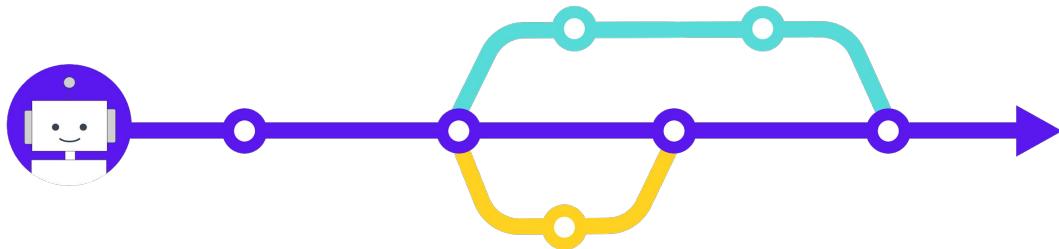


CHAPTER 9: IMPROVING THE ASSISTANT

Connecting the Assistant to Rasa X

Integrated Version Control allows you to connect your Rasa X instance to a remote git repository, like GitHub, to keep track of the version history for your training data. In addition to keeping a record of all changes, Integrated Version Control allows teams to take advantage of the downstream benefits of a git-based development workflow: reviewing changes before they go into production, integrating CI/CD, and running tests on proposed changes.

Integrated Version Control works by performing a two-way sync with the remote Git repository. If changes exist in Rasa X that haven't been pushed to the remote repository, Integrated Version Control lets you commit those changes to an existing branch or create a new one. Read more about Integrated Version Control [on the Rasa blog](#) and [in the documentation](#).



Project Structure

To set this up, we first need to upload our medicare locator project files to a remote GitHub repository. See GitHub's documentation for [step-by-step instructions](#).

File	Commit	Time
__pycache__	initial commit	2 days ago
data	initial commit	2 days ago
models	initial commit	2 days ago
DS_Store	initial commit	2 days ago
README.md	Initial commit	2 days ago
actions.py	initial commit	2 days ago
config.yml	initial commit	2 days ago
domain.yml	initial commit	2 days ago
endpoints.yml	initial commit	2 days ago
events.db	initial commit	2 days ago
rasa.db	initial commit	2 days ago
medicare-assistant		

CHAPTER 9: IMPROVING THE ASSISTANT

In order for Integrated Version Control to sync with the remote repository, the file structure needs to follow the same structure generated when you create a new project using the `rasa init` command. If you've been following along with this series to build your assistant, you should be good to go.

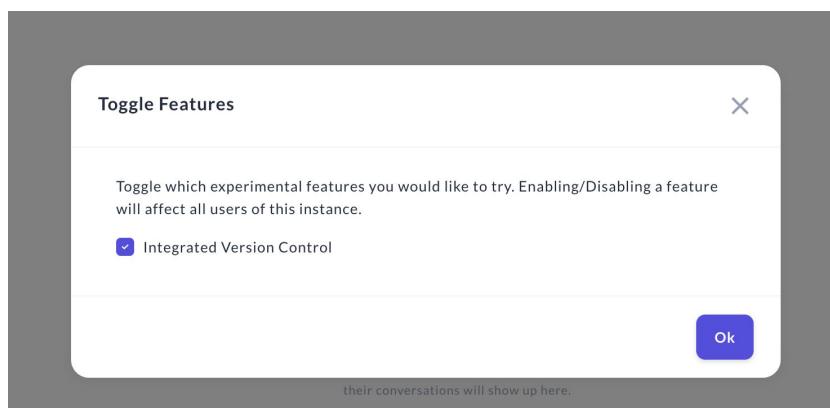


Activate Integrated Version Control

At the time this video was released, Integrated Version Control is an experimental feature that must be manually toggled on. Let's go ahead and activate it.

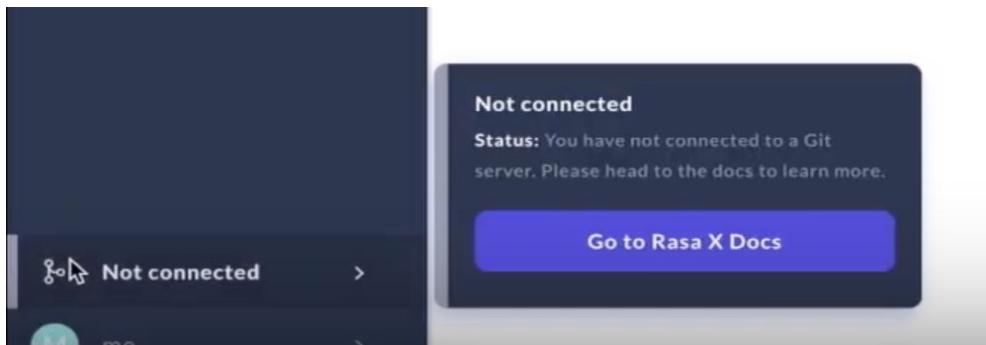
Note: as of Rasa X 0.26.0, Integrated Version Control is no longer an experimental feature and we've simplified the setup process. [Check the docs](#) for the most up-to-date instructions.

In the Rasa X dashboard, click on the user icon in the lower left hand corner and select Experimental. Check the box next to Integrated Version Control.



CHAPTER 9: IMPROVING THE ASSISTANT

Once Integrated Version Control has been activated, you will see a git icon appear in the bottom left corner, just above your user icon, with a message indicating the feature is not connected.



Next, we'll authenticate our server with GitHub to establish the connection, by generating an SSH key on the server, saving the public key in our GitHub repository settings, and sending the private key to Rasa X via an API call.

Generate SSH keys

Navigate back to your terminal. If you've closed the connection to your VM instance, log back in.

Run the following command to generate a public and private SSH key.

```
ssh-keygen -t rsa -b 4096 -f git-deploy-key
```

When prompted, do not set a passphrase. After the key has finished generating, you can run the `ls` command in the `/rasa/etc` directory to see the newly created keys: `git-deploy-key` (the private key) and `git-deploy-key.pub` (the public key).

A terminal window showing the generation of an RSA 4096 key pair. An arrow points from the terminal text to the "git-deploy-key" file in the ls output, which is highlighted with a red box. The terminal shows the following output:

```
root@rasax--server:/etc/rasa# ssh-keygen -t rsa -b 4096 -f git-deploy-key
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in git-deploy-key.
Your public key has been saved in git-deploy-key.pub.
The key fingerprint is:
SHA256:fd1AnGhN8jqmhZ+Vu9o5SZDpjKzX/NAD5DG7mdjTuQo root@rasax--server
The key's randomart image is:
+---[RSA 4096]----+
|   .*= |
|   o.* . |
|   .**+o |
|   .=oBo. |
|   +SoXo=. |
|   ..+o.o.o |
|   .oE.o... |
|   = *+ |
|   . +*o |
+---[SHA256]----+
root@rasax--server:/etc/rasa# ls
auth_credentials.yml db endpoints.yml git-deploy-key logs rasa_x_commands.py terms
certs credentials.yml docker-compose.yml environments.yml git-deploy-key.pub models scripts
root@rasax--server:/etc/rasa#
```

CHAPTER 9: IMPROVING THE ASSISTANT

Save the public key in GitHub

We'll print the public key to the terminal so we can copy and save it in our GitHub settings.

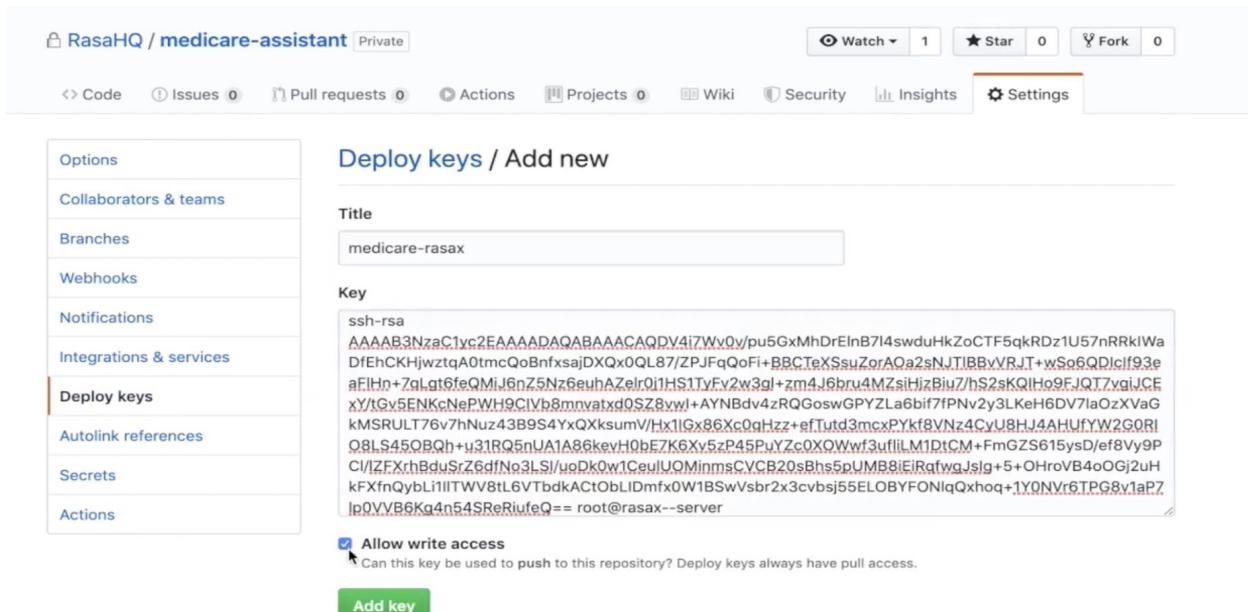
Run the following command to view the public key:

```
cat git-deploy-key.pub
```

Copy the entire contents.

```
[root@rasax--server: /etc/rasa# ls
auth credentials db endpoints.yml git-deploy-key logs rasa_x_commands.py terms
certs credentials.yml docker-compose.yml environments.yml git-deploy-key.pub models scripts
root@rasax--server:/etc/rasa# cat git-deploy-key.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQACQDV4i7Wv0/v/pu5GxMhDrElnB714swduHkZoCTF5qkRDz1U57nRRkIWaDfEhCKHjwztqA0tmcQo8nfxsajDXQx0QL87/ZPJFqQoFi+BBCTeXSsuZorA
0a2sNT1LBvvRJt+wSo6QDlclf93eaflhn-7qlgt6feQm1J6nZ5nZ6euhA2elr0j1HS1TyFv2w3gl+zm4J6bru4M2s1hjzBiu7/h52sKQlHo9FJQT7vgiJCeXy/TgvsENKcnePWH9C1Vb8mnvatxd0S
Z8w1+AYNbvd4zRQGoswGPYZLa6bfif7fPNv2y3LKeH6DV7la0zXVaGkMSRULT76v7hNu43B9S4YxQXksumV/Hx1IGx86xC0Hzz+efTutd3mcxPYkF8VNz4CyU8HJ4AHUFYW2G0RI08LS45OBQh+u3
1RQ5nUA1A86kevH0bE7K6xv5zP45PuYzC0X0Wwf3uf1lM1DtCM+FmGZS615ysD/ef8Vy9PC1/ZFXrbduSrZ6dfno3LS1/uodk0w1CeulU0MinmsCVB20sBhs5pUMB8iEiRqfwgJsIg+5+OhrvB
4oGj2uHkFXfnQyblLi1lTWV8tL6VTbdkActObLIDmf0xW1BSwVsbr2x3cvbsj55ELOBYFONlqQxhoq+1Y0NvR6TPG8v1aP7lp0VV6Kg4n54SReRiufeQ== root@rasax--server#
root@rasax--server:/etc/rasa# ]
```

In your GitHub repository, navigate to Settings>Deploy keys. Click the **Add deploy key** button and paste your public key into the Key box. Give the key a title to identify it, like medicare-rasax, and be sure to check the box to allow Write permissions. Click **Add key**.



CHAPTER 9: IMPROVING THE ASSISTANT

Connect the repository to Rasa X

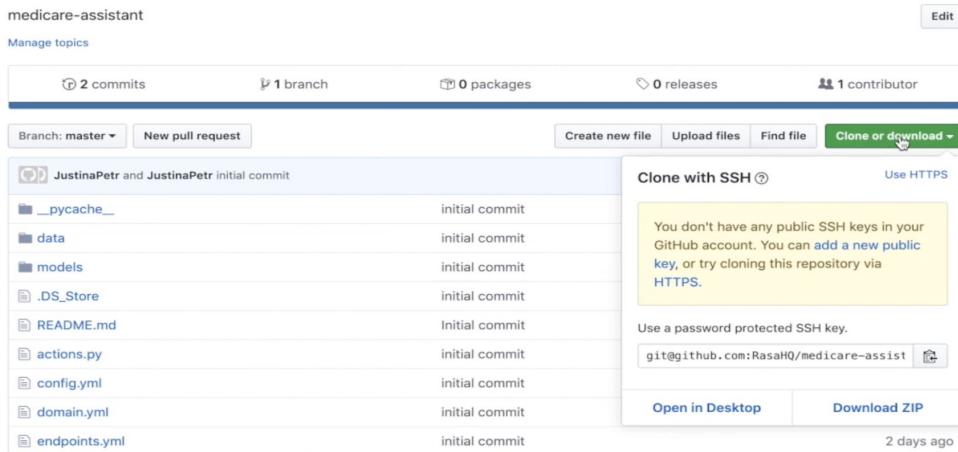
We'll establish the connection between the Rasa X instance and GitHub repository by making a POST request to this Rasa X API endpoint:

```
https://<Rasa X server  
host>/api/projects/default/git_repositories?api_token=<your api  
token>
```

The JSON request body contains three pieces of information:

- **repository_url** - The SSH URL for your GitHub repository, e.g.
<git@github.com>:RasaHQ/rasa-demo.git

To get the URL for your repo, click the **Clone or download** button on your GitHub repository and select the **Use SSH** link



- **target_branch** - The GitHub repository branch where Rasa X should push and pull changes, e.g. master
- **ssh_key** - The private SSH key generated on your server.

To copy the private key, run the following command in the /etc/rasa folder on your server:

```
cat git-deploy-key
```

Copy the entire contents of the key, including the lines -----BEGIN RSA PRIVATE KEY----- and -----END RSA PRIVATE KEY-----

CHAPTER 9: IMPROVING THE ASSISTANT

Once you've assembled the JSON object, you'll have something like this:

```
{  
    "repository_url": "git@github.com:RasaHQ/rasa-demo.git",  
    "target_branch": "master",  
    "ssh_key": "-----BEGIN RSA PRIVATE KEY-----  
b3B1bnNzaC1rZXktdjEAAAAABG5vbmUAAAAEb9uZQAAAAAAAABAAACFwAAAAdzc2gtc  
n  
NhAAAAAwEAAQAAgEAu/Giin7t8DFMxsatbyy1To2EQpLIAhpAIgpyC/e45NYVTwKRGCB  
1  
mxHzt5IWoh7GSWry3pKFBM74UpXxrRPBdCmFeUIiJoslAukNkRSckAUj0VEfoIZLf2SSP  
g  
...  
CDHniFksE1SjkAAAEBAJacZeM2Qdk/vditmBQV97Ac2VJL/Btt8Rks2Vb3CORyXQn3Bp  
b  
+5ZONhmPEoCg4FcZbAm02gYw3dSoBBWz2i8mmAv71mVsNoddWKpDngRFv4PUaITnYYxrZ  
4  
-----END RSA PRIVATE KEY-----"  
}
```

We'll save this JSON object in a file called `repository.json`, in the `/rasa/etc` folder on the server. First, let's create that file:

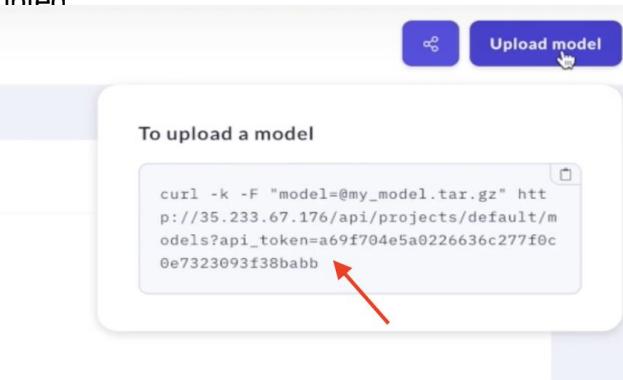
```
touch repository.json
```

Open the file to edit it:

```
nano repository.json
```

Paste the JSON object into the file. Press **Control + X** to exit the editor, and confirm **Y** to save your changes when prompted

We need one more thing to get the easiest way to get the API token. The easiest way to get the API token is to upload a model to the Models JRL.



X API token. The easiest way to get the API token is to upload a model to the Models JRL.

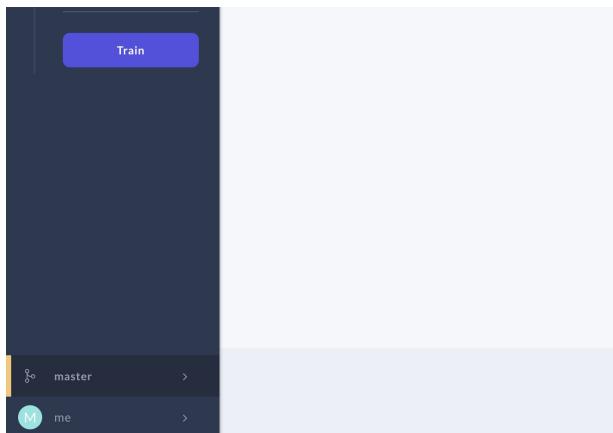
CHAPTER 9: IMPROVING THE ASSISTANT

Head back to the terminal. Still in the `/etc/rasa` directory, run the following cURL command, replacing the Rasa X server URL and API key values with your own:

```
curl --request POST \
      --url http://<Rasa X server
host>/api/projects/default/git_repositories?api_token=<your api
token> \
      --header 'content-type: application/json' \
      --data-binary @repository.json
```

```
root@rasax--server:/etc/rasa# curl --request POST \
>     --url http://35.233.67.176/api/projects/default/git_repositories?api_token=a69f704e5a0226636c277f0c0e7323093f38bab \
>     --header 'content-type: application/json' \
>     --data-binary @repository.json
```

Check the connection by navigating back to the Rasa X dashboard in your browser and checking the Integrated Version Control icon in the bottom left corner. If the connection was successful, you'll see either a green indicator, meaning Rasa X is up to date with the GitHub repository, or a yellow indicator, meaning Rasa X has changes that need to be pushed to GitHub.



Set up the Actions Server

We have one more thing to configure: the assistant's custom action server. To do this, we'll place the assistant's custom action code within an `actions` directory on the server.

CHAPTER 9: IMPROVING THE ASSISTANT

Connect to your server and make sure you're in the /etc/rasa directory. In your terminal, run the following commands to create the actions directory and two files inside it: `__init__.py` and `actions.py`:

```
root@rasax--server:/etc/rasa# mkdir actions
root@rasax--server:/etc/rasa# touch actions/__init__.py
root@rasax--server:/etc/rasa# touch actions/actions.py
```

Run `nano actions/actions.py` to edit the newly-created `actions.py` file. Paste the code from your assistant's `actions.py` file into the blank file, save, and close the editor.

Then, we need to create a `docker-compose.override.yml` file. This file instructs `docker-compose` to spin up a custom action server when the Rasa X server starts up. Let's create that file:

```
touch docker-compose.override.yml
```

Open the file editor:

```
nano docker-compose.override.yml
```

And add the following contents:

```
version: '3.4'
services:
  app:
    image: 'rasa/rasa-sdk:latest'
    volumes:
      - './actions:/app/actions'
    expose:
      - '5055'
    depends_on:
      - rasa-production
```

CHAPTER 9: IMPROVING THE ASSISTANT

Here, we're using the rasa-sdk image to run our custom actions, and we're specifying that the actions server will listen on port 5055. The actions server depends on the rasa-production service, which is responsible for running the trained model, parsing intent messages, and predicting actions.

Once you've saved the file, you can restart the Rasa X docker container and the assistant will be fully functional on Rasa X.

```
sudo docker-compose up -d
```

Let's head back to the Rasa X dashboard where we can have a look and try it out!

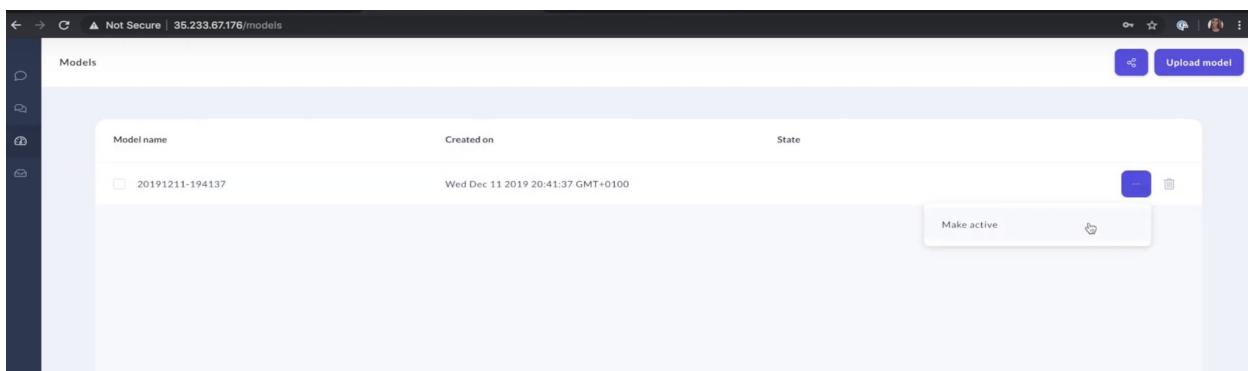
The Rasa X Dashboard

When you log in to Rasa X, you'll see four tabs on the left side of the screen: Models, Talk to your bot, Training, and Conversations. In this section, we'll take a tour of each area.

Models

The Models screen is where you view and manage your models. Now that we've connected Integrated Version Control, all of the files from the GitHub repository have been pulled into the Rasa X instance. Let's hit the **Train** button on the sidebar to train a new model on the latest training data and file configurations.

Once the model has finished training, we can make it active on the Models screen.

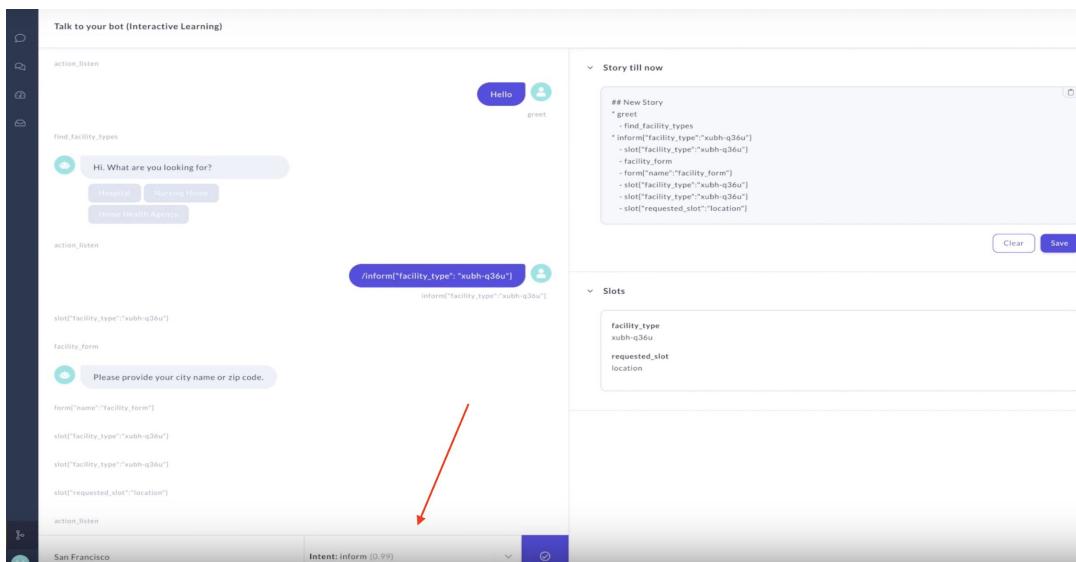


CHAPTER 9: IMPROVING THE ASSISTANT

Talk to your bot

The Talk to your bot screen allows you to test and improve your assistant by talking to it. You can use the Talk to your bot screen in two modes:

- **Simple chat** - Allows you to test the performance of the assistant by having a conversation. This is a nice way to test your assistant and generate new training examples at the same time.
- **Interactive learning** - As you chat with your assistant, you'll be asked to validate the intent classification and action prediction at each step of the conversation. This allows you to finely tune and correct your assistant's behavior.



As you chat with your assistant, you can save new stories and example utterances to your training files. When you've accumulated a few of these changes, you can then retrain the model on the new data.

Training

The Training tab is where you can manage your assistant's training and configuration files, annotate new training examples, and train new models after making changes. Let's look at two important subsections:

CHAPTER 9: IMPROVING THE ASSISTANT

NLU Training

This screen contains the assistant's NLU training examples. If you navigate to this screen, you'll see all of the example utterances we added when we first created the assistant, plus the examples we generated during our interactive learning session on the Talk to your bot screen.

The screenshot shows the NLU Training interface. At the top, there are tabs: 'Annotate new data' (disabled), 'Training data' (selected), and 'Regex & Lookup'. On the left, there's a sidebar with icons for Home, Help, and Log. The main area has a search bar 'Search 271 examples...' and two columns: 'Sentence' and 'Intent'. The 'Sentence' column lists various user inputs, and the 'Intent' column shows the predicted intent for each. A dropdown menu for 'inform' intent is open, listing options like 'Create new', 'affirm', etc. Below the dropdown, there are two more dropdowns, both set to 'thanks'.

Sentence	Intent
/inform{"facility_id": "05045-49"}	inform
/inform{"facility_type": "xubh-q36u"}	inform
cool thank you	affirm
cool, thanks	out_of_scope
amazing, thanks	mood_unhappy
thanks a lot	inform
thanks for the help	goodbye
thanks a bunch for everything	thanks

On the Training data tab, you can see the intent predicted for each utterance. If there's another intent that would be a better match for the utterance, you can change it here.

Finally, under the Annotate new data tab, you'll see the user inputs that were entered during the Talk to your bot session. You can label the entities and intent predictions for each input and add the annotated data to your training files.

CHAPTER 9: IMPROVING THE ASSISTANT

Stories

On the Stories screen, you can select each story in your training data to see a visual representation of the conversation. You can also write new stories, edit existing stories, import stories in bulk, or generate training stories by talking to your assistant in interactive learning mode.

The screenshot shows the 'Stories' screen in a conversational interface. On the left, a sidebar lists several stories: 'happy_path', 'happy_path_multi_requests', 'happy_path2', 'story_goodbye', 'story_thankyou', 'Chat with me', 'New Story', and 'New Story'. The 'happy_path' story is selected, and its content is displayed in the main pane:

```
1  ee happy_path
2  * greet
3  * find_facility_types
4  * inform{"facility_type": "rbry-mqueu"}
5  facility_form
6  - form{"name": "facility_form"}
7  - form{"name": null}
8  * inform{"facility_id": 4245}
9  find_healthcare_address
10 - utter_address
11 * thanks
12 - utter_noworries
```

To the right of the story content is a 'Flow' diagram. It starts with a 'Start' node, followed by a blue rounded rectangle labeled 'Agree'. This leads to a 'find_facility_types' node, which then triggers an 'inform{"facility_type": "rbry-mqueu"}' action. From there, it branches into two parallel paths, each leading to an 'inform{"facility_id": "0054541"}' or 'inform{"location": "San Francisco"}' action. These paths converge back to a 'facility_form' node, which then leads to a 'find_healthcare_address' node. This is followed by an 'utter_address' node, a blue rounded rectangle labeled 'thanks', and finally an 'utter_noworries' node, which concludes at an 'End' node.

Conversations

When you navigate to the Conversations screen, you'll notice something: it's empty! That's because we haven't yet shared our assistant with any testers. When we do, those conversations will show up here.

In our next episode, we focus on the process of improving AI assistants, by sharing your assistant with testers and real users.

CHAPTER 9: IMPROVING THE ASSISTANT

Conclusion

In this episode of the Rasa Masterclass, we covered quite a bit of ground: we deployed Rasa X to a server, connected the assistant using Integrated Version Control, and took a tour of the Rasa X dashboard. This sets us up to cover our next topic: further improving your assistant using Rasa X.

Keep up the momentum, and if you get stuck, ask us a question in the [Community forum](#).

Additional Resources

- [Ep #9 Rasa Masterclass - Improving the assistant: Setting up Rasa X](#) (YouTube)
- [Rasa X - Docker-compose Quick Install](#) (Rasa docs)
- [Integrated Version Control](#) (Rasa docs)
- [Integrated Version Control: Linking Rasa X with Git-based Development Workflows](#) (Rasa blog)
- [Rasa Open Source + Rasa X: Better Together](#) (Rasa blog)



CHAPTER TEN

SHARING WITH TEST USERS

SHARING WITH TEST USERS



Introduction

In episode 10 of the Rasa Masterclass, we focus on gathering valuable feedback, by sharing your assistant with test users. We set ourselves up in episode 9 by deploying Rasa X and connecting the assistant. Now, we'll let test users talk to the assistant and explore the ways Rasa X helps you improve based on those conversations.

CHAPTER 10: SHARING WITH TESTERS

Opening your assistant to testers

Once your assistant is capable of handling the most important happy path stories, the next step is to invite testers—real users—to test and have conversations with your assistant. Using the conversations you collect from testing, you will be able to review conversations and make minor, major, and architectural changes to improve your assistant.



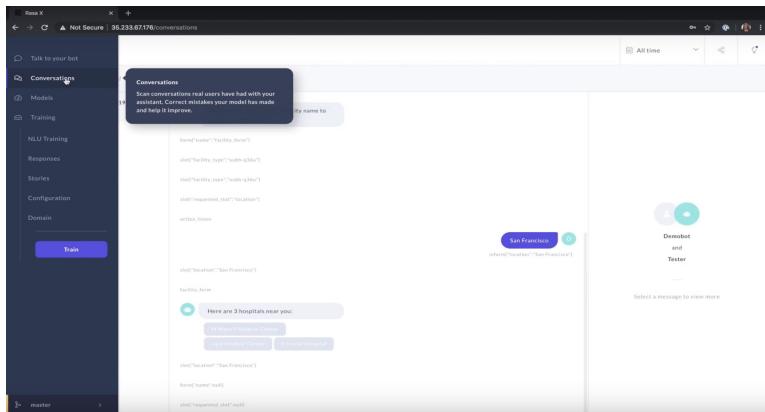
This is an important intermediary step that sits between the initial development – where you rely on manually generated training data – and the deployment stage – where you open up your assistant to a wider audience of users. Inviting guest testers to try out your assistant will help you to spot problems early, and make the improvements necessary to ensure that real users have a good experience when using your assistant in production.

Rasa X comes with a number of useful features that enable you to easily share your assistant with real testers, collect their conversations with your assistant, then review those conversations and decide which improvements you should work on next.

Reviewing the Rasa X UI will help to better understand these features.

CHAPTER 10: SHARING WITH TESTERS

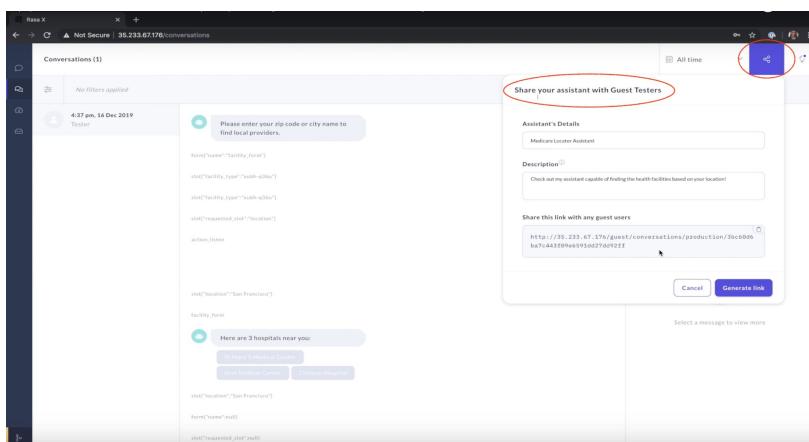
Conversations tab



The “Conversations” tab is accessed on the left navigation within Rasa X. Clicking on this tab will show you the conversations that users have had with your assistant. This tab will also show how many conversations have taken place, when those conversations happened, and the content of the actual conversations themselves. If you haven’t shared your assistant with real users just yet, this section will be empty.

Sharing with guest testers

Rasa X allows you to invite real testers to talk to your assistant via a shareable link. At the top right, you will find an icon to share your assistant. Clicking on the icon will bring up a “Share with guest testers” dialog box.



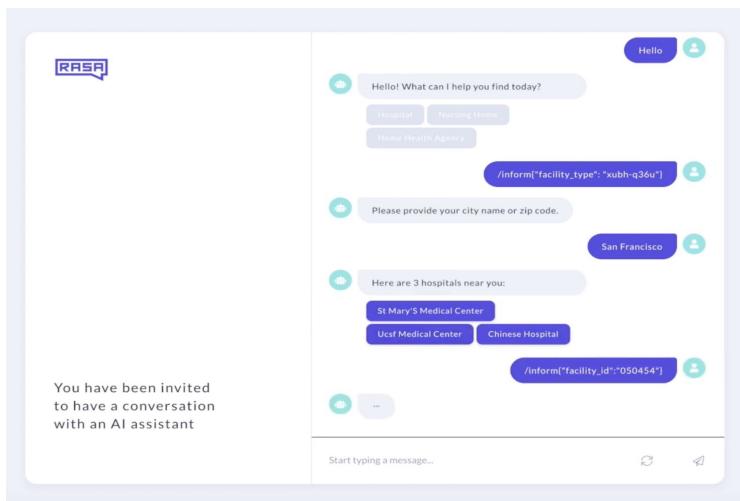
CHAPTER 10: SHARING WITH TESTERS

Here you can provide a name for your assistant and a description about what your assistant does. Once you are happy with the details, hit `Generate link`. This link can be shared with the people that you want to test your assistant - your friends, family, colleagues or a specific group of people that you've selected for a user test.

As you can see, the link in the screenshot above includes our server IP address. Users will be able to test using this URL, but best practice would be to configure a custom domain and SSL for your server, so that the users would see a more familiar URL format. We'll cover these steps in episode 11.

What the guest tester sees

Once an invited test user clicks on the link you shared, they will see a simple chat UI where they can immediately talk to your assistant. The guest tester doesn't need to install software or do any additional setup, making it very simple for them to start testing.



It's important to note that multiple guest testers can talk to your assistant, all at the same time. All of the conversations they have with your assistant are stored in a database on your server and are displayed in the Conversations tab of Rasa X for you to review.

Even more importantly, remember that all of the conversations you collect and view using Rasa X are stored on **your** server, which means that you completely own your data. Conversations are not shared with Rasa. You completely control access to these conversations.

Lastly, remember that you can find all of the files of your assistant on your server - including models, actions, configurations, and the database itself.

CHAPTER 10: SHARING WITH TESTERS

Reviewing test conversations

After your guest testers have had a chance to chat with your assistant, you can review their conversations in Rasa X.

Each conversation has a timestamp of when it took place. To make reviewing the conversations more efficient, you can apply filters to the collected conversations - for example, you can filter by intents or actions included in the stories, or the number of user messages included in a story. .

The screenshot shows the Rasa X interface for managing conversations. At the top, there's a header 'Conversations (5)' and a search bar. Below the header, a message says 'No filters applied'. There are four main sections for filtering:

- Intents (7)**: Contains checkboxes for 'greet', 'inform', 'out_of_scope', 'restart', 'restrt', 'search_provider', and 'thanks'.
- Entities (3)**: Contains checkboxes for 'facility_id', 'facility_type', and 'location'.
- Actions (9)**: Contains checkboxes for 'action_default_ask_affirmation', 'action_default_ask_rephrase', 'action_default_fallback', 'action_restart', 'facility_form', 'find_facility_types', 'find_healthcare_address', and 'utter_address'.
- Other filters**: Contains checkboxes for 'Includes flagged messages' and 'At least [10] user messages'.

At the bottom right are 'Cancel' and 'Apply Filters' buttons.

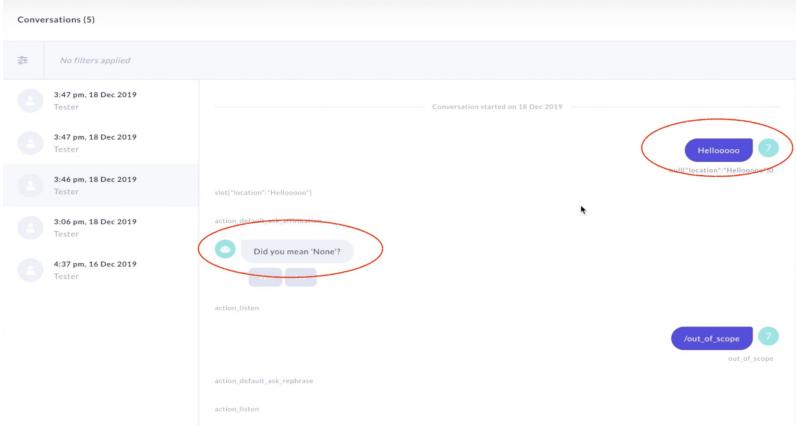
Reviewing these collected conversations should be a great source of learnings about your assistant, and helps in deciding which changes are needed to improve your assistant.

Improving your assistant

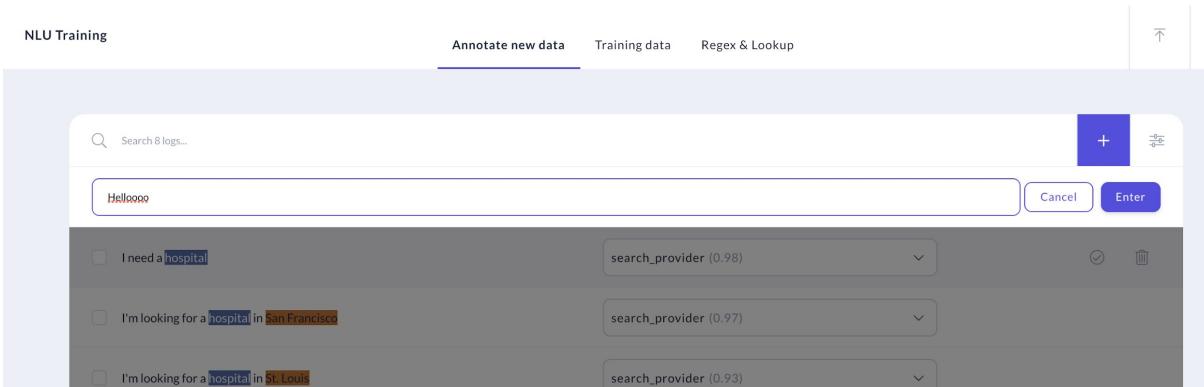
Changes within Rasa X

Minor changes are typically something that can be adjusted within Rasa X, like improving your NLU model to better classify intents. In the example below, the assistant failed to classify the message `Helloooo` as a greeting. This could be an indication that we should include more, and more unusual, examples of greetings in our training data.

CHAPTER 10: SHARING WITH TESTERS



From the NLU Training screen, located under the Training tab, we can add ‘Helloooo’ as a new training example. Rasa X will automatically suggest an intent match, and we can verify that the intent for the message should be `greet`.



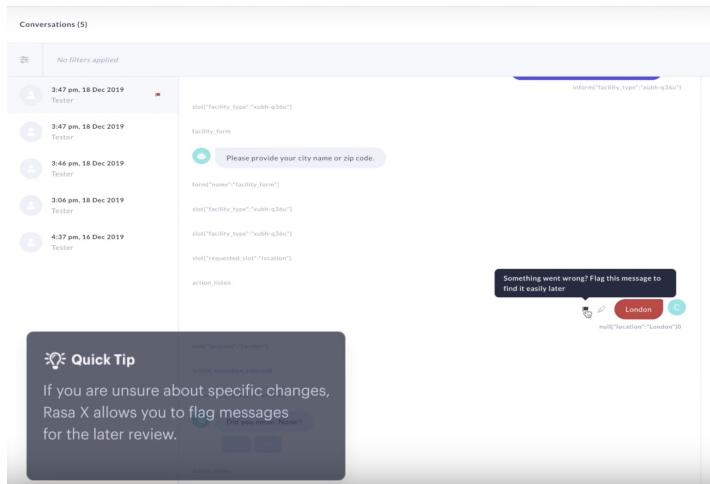
Major changes to your assistant, like creating a new intent or adjusting the logic of your stories, can be done either within Rasa X or on the command line. This [blog post](#) highlights how that works within Rasa X.

Architectural changes

Your test conversations may uncover more substantial issues that require architectural changes to your assistant, such as changing your custom actions. These changes usually require updates to your assistant’s files on your local computer, made via a text editor, and then pushed to GitHub and your production server.

CHAPTER 10: SHARING WITH TESTERS

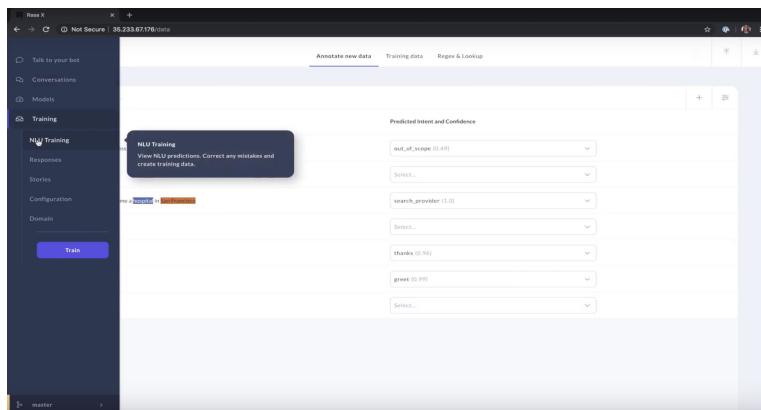
For example, in testing our medicare locator, we collected a conversation with a test user based in London. However, the custom action we created can handle only requests for US-based facilities. We can flag this conversation for later review.



In this case, we would either have to include location validation, or update our custom actions with a new API which would find health facilities in locations outside the US.

Improving the NLU model

The conversations you collect can also be used to improve your NLU model.



CHAPTER 10: SHARING WITH TESTERS

You can access the NLU training section of Rasa X in the left navigation, and there you will find the inputs from test users who talked to your assistant. From this screen, you can review each of the inputs and annotate them. You can improve your NLU model by annotating a user input and its predicted intent as correct.

The screenshot shows the Rasa X interface with the 'NLU Training' tab selected. The main area displays a list of user sentences with their predicted intents and confidences. The first sentence, 'This is useless', has 'out_of_scope (0.49)' selected. The second sentence, 'London', has 'Select...' selected. The third sentence, 'Hellooooo', has 'Select...' selected. The fourth sentence, 'Hey there', has 'greet (0.99)' selected. A 'Mark Correct' button is visible next to the third sentence. A search bar at the top left says 'Search 7 logs...'. Navigation tabs at the top include 'Annotate new data' (which is active), 'Training data', and 'Regex & Lookup'.

If some examples aren't useful – like below, where the user's input Hey was extracted as an entity -- such inputs can simply be deleted.

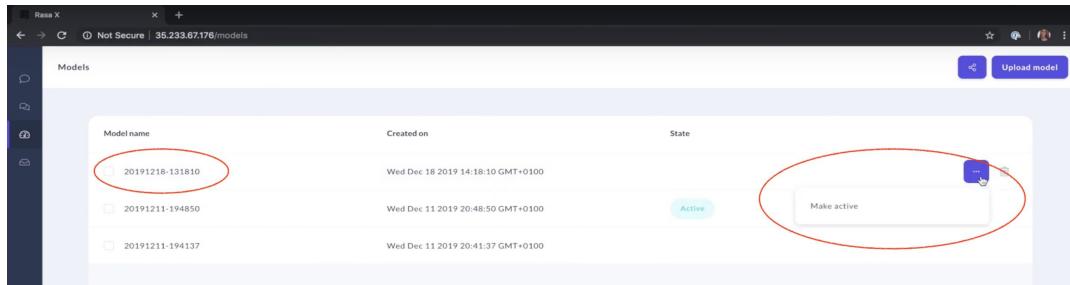
This screenshot shows the same Rasa X interface as above, but with a different set of user inputs. The first sentence, 'This is useless', has 'out_of_scope (0.49)' selected. The second sentence, 'London', has 'Select...' selected. The third sentence, 'Hellooooo', has 'Select...' selected. The fourth sentence, 'Hey', has 'greet' selected. A 'Delete' button is visible next to the fourth sentence. The search bar at the top left says 'Search 4 logs...'. Navigation tabs at the top include 'Annotate new data' (active), 'Training data', and 'Regex & Lookup'.

After you make improvements to your model and annotate new training data, you should train a new model. In prior episodes of the Masterclass, we've been training our model using the command line, but training can also be done directly in Rasa X.

This screenshot shows the Rasa X interface with the 'Training' tab selected in the sidebar. The main area displays the same list of annotated user inputs as the previous screenshots. A prominent blue 'Train' button is located at the bottom left of the main panel. The sidebar includes sections for 'Talk to your bot', 'Conversations', 'Models', and 'Training'. The 'Training' section is expanded, showing 'NLU Training', 'Responses', 'Stories', 'Configuration', and 'Domain'. The 'Train' button is highlighted with a mouse cursor.

CHAPTER 10: SHARING WITH TESTERS

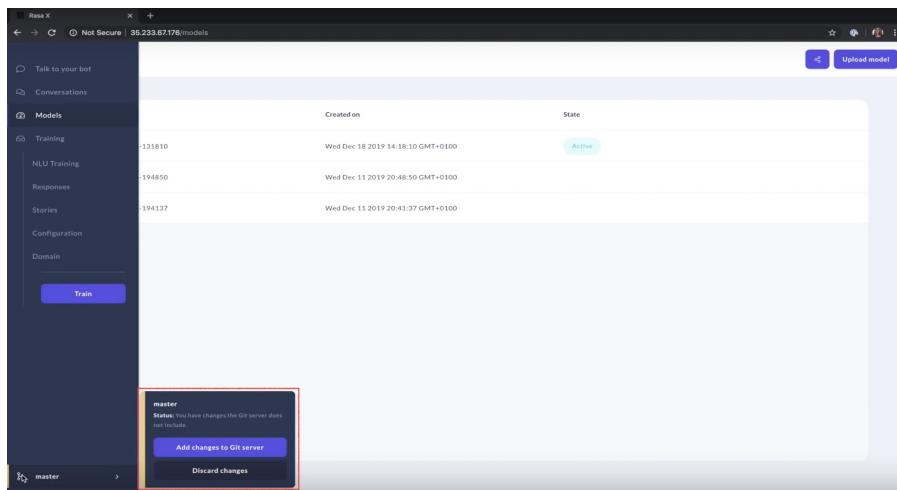
Once you make the new model active, you can start sharing your assistant with real testers again, and start collecting new conversations.



Version control

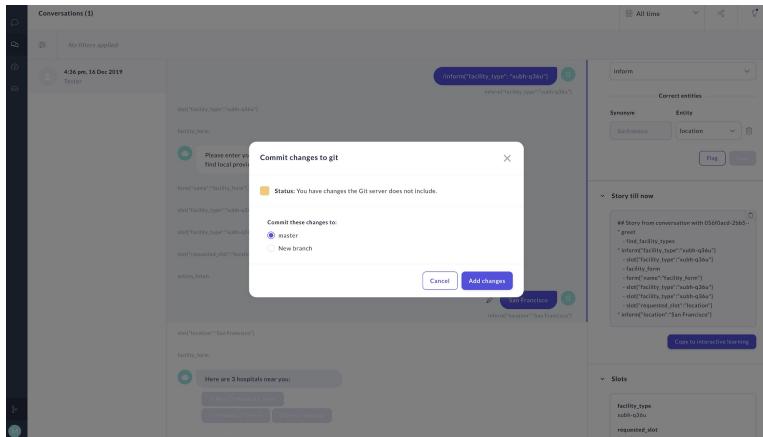
You can version control your assistant using Rasa X's integrated version control feature.

When you make a change to your model in Rasa X, like annotating new training data or changing model configurations, you will see that the integrated version control icon is highlighted to the left with an orange bar. This indicates that there are some changes you made in Rasa X that are not reflected in your git server.



To include those changes, you can commit them to the master branch or create a new one.

CHAPTER 10: SHARING WITH TESTERS



Once the changes are added, you will find a pull request on your assistant's git repository with suggested changes. You can open a PR, invite team members to review and merge it, and continue improving your assistant.

Conclusion

Rasa X is an essential tool in improving your Rasa Open Source based assistant. The process outlined in this Masterclass should be repeated until you're happy with how your assistant performs:

- Share the assistant with real testers,
- Make improvements based on the collected conversations, and
- Repeat.

Once you feel your assistant can handle conversations with guest testers, it's time to take the next step - connecting to outside messaging channels.

An outside messaging platform can be one of many platforms: Slack, Facebook Messenger, SMS your own website, or any other method where real users will talk to your assistant. You will collect the conversations users have with your assistant, and continue making improvements based on what you can learn from these real-world conversations. We'll cover this in the next episode of the Rasa Masterclass, #11.

CHAPTER 10: SHARING WITH TESTERS

Additional Resources

- [Ep. #9 - Rasa Masterclass - Improving the assistant: Setting up the Rasa X](#) (YouTube)
- Rasa Blog: [Rasa Open Source and Rasa X: Better Together](#)
- Rasa Blog: [Integrated Version Control](#)
- Rasa Docs: [Improve your assistant with Rasa X](#)



CHAPTER ELEVEN

CONNECTING TO MESSAGING CHANNELS

CONNECTING TO MESSAGING CHANNELS



Introduction

In the previous episode of the Rasa Masterclass, we learned how to share the medicare locator assistant with test users. Sharing the assistant with test users is an important middle step for QA, allowing you to build up the training data set with realistic dialogues and review collected conversations for actionable insights.

In this episode, we'll go one step further by making the assistant available to real users on a publicly accessible messaging channel. This is an exciting milestone—our assistant will finally be live.

We'll connect the medicare locator assistant to two channels: Telegram and a website chat widget. Then, we'll head back to Rasa X to view the conversations our users are having on these channels.

CHAPTER 11: MESSAGING CHANNELS

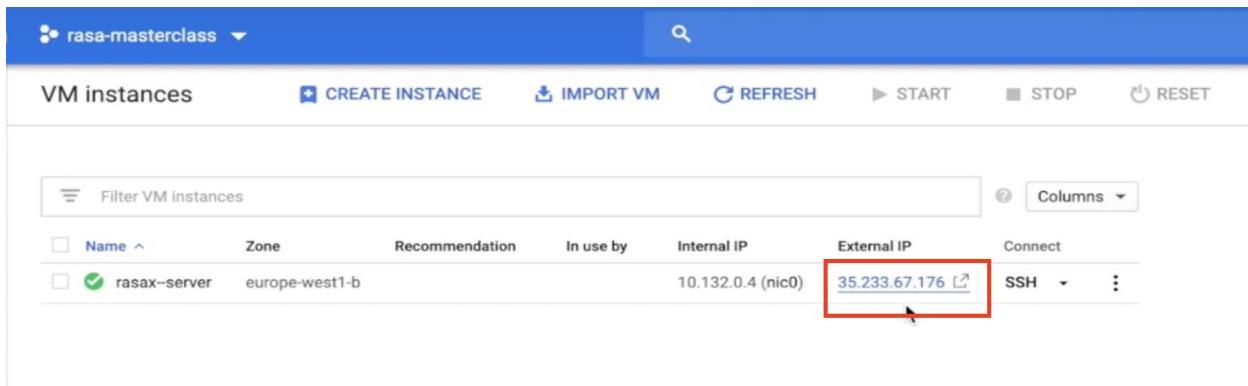
Configuring DNS and SSL

Before we can connect our assistant to messaging channels, we need to do some additional setup on the Rasa X server. As you recall, when we deployed Rasa X, we were able to access Rasa X in the browser by going to an IP address with an http protocol, like <http://34.62.185.11/>. That was fine while we were in early development, but in order to securely connect to external applications—our messaging channels—we need to configure a domain name and SSL. After completing these steps, the Rasa X instance will be accessible at a secure, human-friendly URL like <https://rasamasterclass.com>.

First, we'll assign a domain name to the server instance. Then, we can configure an SSL certificate, bound to the domain.

Assigning a static IP address

If you're following along with this tutorial and using Google Cloud Platform as your host, you can find the IP address for your VM instance in your GCP Console under Compute Engine>VM Instances.



The screenshot shows the Google Cloud Platform Compute Engine interface. At the top, there's a blue header bar with the project name "rasa-masterclass" and various navigation icons. Below the header, a toolbar includes "CREATE INSTANCE", "IMPORT VM", "REFRESH", "START", "STOP", and "RESET". The main area is titled "VM instances" and contains a table. The table has columns for "Name", "Zone", "Recommendation", "In use by", "Internal IP", "External IP", and "Connect". A single row is visible, showing "rasax-server" in the Name column, "europe-west1-b" in the Zone column, "10.132.0.4 (nic0)" in the Internal IP column, and "35.233.67.176" in the External IP column. The "External IP" cell is highlighted with a red box and has a copy icon over it. A cursor arrow is pointing towards the "External IP" cell.

Before you point your domain to this address, you'll want to complete one additional step. By default, the external IP address assigned to a GCP VM instance is ephemeral, meaning that a new IP address is assigned each time the server is stopped or started. This would break the connection with the domain and require you to update your DNS records to point to the new IP.

CHAPTER 11: MESSAGING CHANNELS

To prevent this, you can promote your external IP address from ephemeral to static. Select your VM instance and click **Edit**. Then, scroll down to the Network interfaces settings. In the External IP dropdown, select **Create IP address**.

The screenshot shows the Google Cloud Platform Compute Engine interface. On the left, there's a sidebar with various options like Compute Engine, Images, TPUs, etc. The main area is titled 'VM instance details' and shows 'Network interfaces'. It lists a single interface named 'default' with 'Internal IP' 10.128.0.2 and 'Internal IP type' set to 'Ephemeral'. A dropdown menu is open under 'External IP type', showing 'None' and 'Ephemeral' as options, with 'Create IP address' also visible. Below this, 'IP forwarding' is set to 'Off'. At the bottom, there are 'Public DNS PTR Record' and 'Public IP' sections.

Enter a name, and click **Reserve**. This IP address will now be associated with your server until it is explicitly removed.

Reserve a new static IP address

Name ?
Name is permanent

Description (Optional)

Network Service Tier ?
 Premium (Current project-level tier, [change](#)) ?
 Standard ?

Region
us-central1

i Standard tier uses the same region as your VM instance

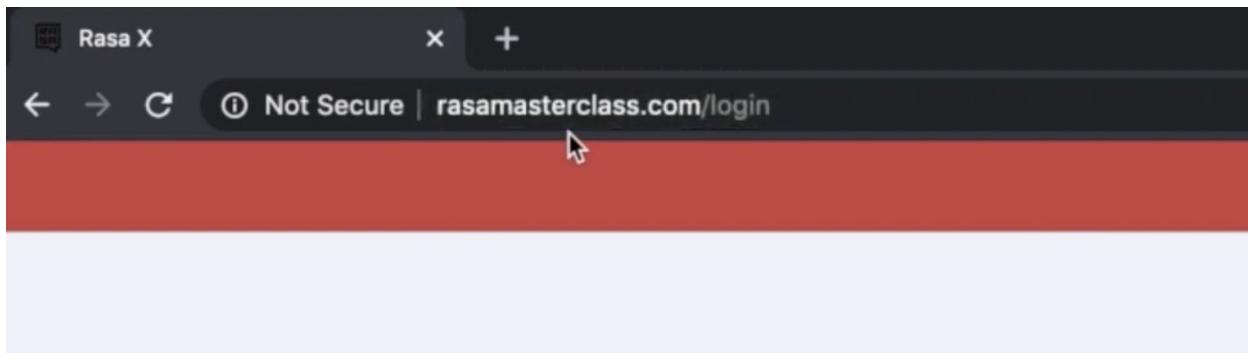
CANCEL RESERVE

CHAPTER 11: MESSAGING CHANNELS

Pointing to a domain

You can purchase the domain name for your Rasa X instance from any registrar you would like; for example, [Google domains](#). Once you've registered the domain, you can apply it to your Rasa X instance by creating an [A record](#) that maps the domain to the external IP address for your server. Be sure to check your domain registrar's documentation for specific instructions on creating DNS records.

After updating DNS records, be sure to allow time for them to propagate. Test the new configuration by visiting <http://yourdomain.com> in the browser.



Installing an SSL certificate

An SSL certificate creates a secure connection between the browser and the server. It verifies that the requested hostname matches the name registered on the certificate.

To secure your setup, you can either purchase a certificate from a certificate authority or you can use a free option like [Let's Encrypt](#). In this tutorial, we'll use [certbot](#): a free, open source tool that makes it easy to install an SSL certificate from Let's Encrypt.

Begin by establishing an SSH connection to your VM instance. In the terminal, stop the docker container:

```
sudo docker-compose down
```

Run the following command to install certbot:

```
sudo apt-get certbot
```

CHAPTER 11: MESSAGING CHANNELS

Start the installation with this command:

```
sudo certbot certonly
```

Certbot asks a series of questions to guide you through the process of configuring the SSL. First, you'll be asked how you would like to authenticate with the ACME CA. Choose option 1, **Spin up a temporary webserver**.

When prompted, provide a valid email address, accept the terms and conditions, and finally, enter the domain name you applied to the Rasa X server.

```
v upgrade, v newly installed, v to remove and is not upgraded.
root@rasax--server:/etc/rasa# sudo certbot certonly
Saving debug log to /var/log/letsencrypt/letsencrypt.log

How would you like to authenticate with the ACME CA?
- -
1: Spin up a temporary webserver (standalone)
2: Place files in webroot directory (webroot)
- -
Select the appropriate number [1-Z] then [enter] (press 'c' to cancel): 1
Plugins selected: Authenticator standalone, Installer None
Enter email address (used for urgent renewal and security notices) (Enter 'c' to
cancel): email@raso.com
Starting new HTTPS connection (1): acme-v02.api.letsencrypt.org

- -
please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf. You must
agree in order to register with the ACME server at
https://acme-v02.api.letsencrypt.org/directory
(Agree/(C)ancel: A

- -
Would you be willing to share your email address with the Electronic Frontier
Foundation, a founding partner of the Let's Encrypt project and the non-profit
organization that develops Certbot? We'd like to send you email about our work
encrypting the web, EFF news, campaigns, and ways to support digital freedom.
- -
(Y)es/(N)o: N
Please enter in your domain name(s) (comma and/or space separated) (Enter 'c'
to cancel): rasamasterclass.com
```

When the process has been completed successfully, you'll see the following message in your terminal:

```
IMPORTANT NOTES:
- Congratulations! Your certificate and chain have been saved at:
  /etc/letsencrypt/live/rasamasterclass.com/fullchain.pem
  Your key file has been saved at:
  /etc/letsencrypt/live/rasamasterclass.com/privkey.pem
  Your cert will expire on 2020-04-28. To obtain a new or tweaked
  version of this certificate in the future, simply run certbot
  again. To non-interactively renew *all* of your certificates, run
  "certbot renew"
- Your account credentials have been saved in your Certbot
  configuration directory at /etc/letsencrypt. You should make a
  secure backup of this folder now. This configuration directory will
  also contain certificates and private keys obtained by Certbot so
  making regular backups of this folder is ideal.
- If you like Certbot, please consider supporting our work by:

  Donating to ISRG / Let's Encrypt:  https://letsencrypt.org/donate
  Donating to EFF:                  https://eff.org/donate-le

root@rasax--server:/etc/rasa#
```

CHAPTER 11: MESSAGING CHANNELS

By default, the certificate is saved to a /letsencrypt/ directory. We'll need to move the files to the /etc/rasa/ directory to make them accessible to the docker container.

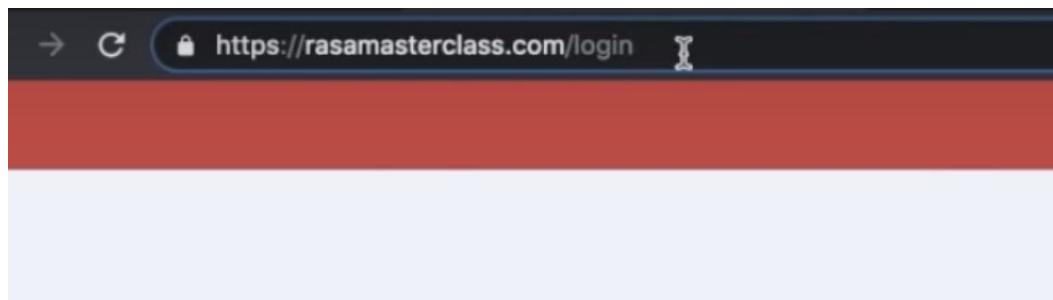
```
sudo cp /etc/letsencrypt/live/rasa.example.com/privkey.pem  
/etc/rasa/certs/
```

```
sudo cp /etc/letsencrypt/live/rasamasterclass.com/fullchain.pem  
/etc/rasa/certs/
```

Re-start the docker container to put the updates into effect:

```
sudo docker-compose up -d
```

Test the changes by navigating to <https://yourdomain.com>. You should now see a secure lock symbol in the address bar.



Note:

Let's Encrypt certificates are valid for 90 days. You can renew the certificate with the following command (either manually or through an automated cron job):

```
sudo certbot renew
```

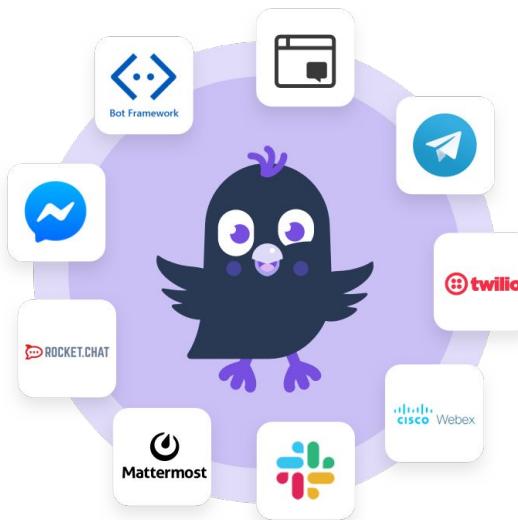
```
root@rasax--server:/etc/rasa# sudo cp /etc/letsencrypt/live/rasamasterclass.com/fullchain.pem /etc/rasa/certs/  
root@rasax--server:/etc/rasa# sudo certbot renew  
Saving debug log to /var/log/letsencrypt/letsencrypt.log  
  
-----  
Processing /etc/letsencrypt/renewal/rasamasterclass.com.conf  
-----  
Cert not yet due for renewal  
-----
```

CHAPTER 11: MESSAGING CHANNELS

Connecting to Messaging Channels

Rasa comes pre-configured to connect with popular messaging channels like Slack, Facebook Messenger, Telegram, and [many more](#). You can also connect to a website chat widget, SMS, or integrate your own custom channel.

In this episode, we demonstrate how to connect the medicare locator assistant to two channels: Telegram and a website chat widget. The configuration process is similar across different channels, but you can find step-by-step instructions for each channel [in the documentation](#).

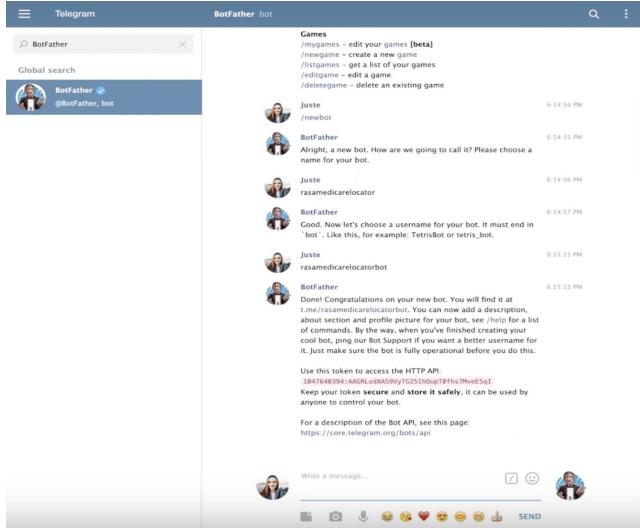


Telegram

Telegram is a popular free messaging app that syncs messages across a number of different devices—smartphones, tablets or laptops. It supports third party integrations like AI assistants, so it's a great platform to build on.

Telegram provides an assistant called [BotFather](#) that helps developers register a new bot integration. Use the /newbot command to get started. When prompted, provide a name and username for your assistant. The name is displayed in contact details. The username is a short name used in mentions and must end in `bot`. When the assistant's profile is complete, you'll be given an API token.

CHAPTER 11: MESSAGING CHANNELS



Now that we have the Telegram API token, we can connect our Rasa assistant. Head back to your server and open a terminal session.

The `credentials.yml` file holds the configuration details for connecting to external applications, like messaging channels. This file is created automatically with the `rasa init` command.

From the `/etc/rasa` directory, open the `credentials.yml` file:

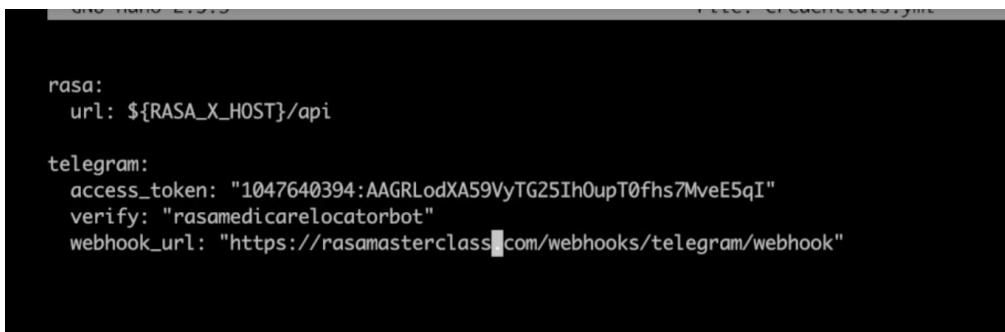
```
sudo nano credentials.yml
```

Edit the contents of the file to include the following details:

```
telegram:  
  access_token: "490161424:AAGlRxinBRTKgb21_r1OEMtDFZMXB16EC0o"  
  verify: "your_bot"  
  webhook_url: "https://your_url.com/core/webhooks/telegram/webhook"
```

Here, we're providing the channel name (`telegram`) and the `access_token`, which we obtained from Telegram's BotFather assistant. In the `verify` field, provide the username you registered with BotFather. Be sure to replace the `your_url.com` placeholder in the `webhook_url` with the domain of your Rasa X server.

CHAPTER 11: MESSAGING CHANNELS



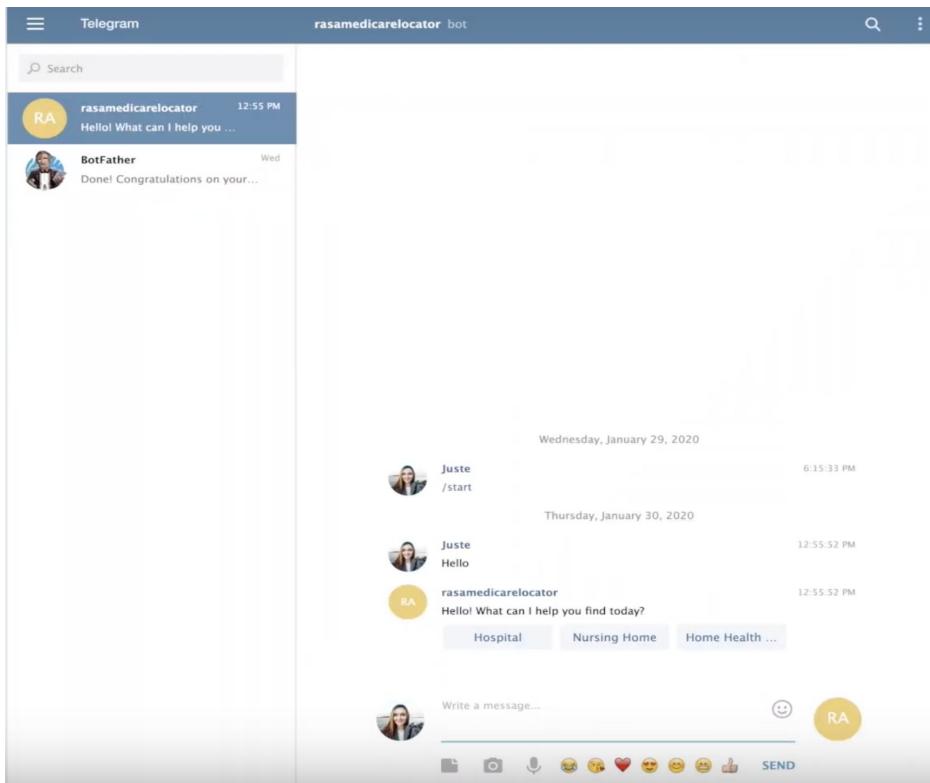
The terminal window displays two configuration files: `rasa.yaml` and `telegram.yaml`.

```
rasa:  
  url: ${RASA_X_HOST}/api  
  
telegram:  
  access_token: "1047640394:AAGRLoxA59VyTG25Ih0upT0fhs7MveE5qI"  
  verify: "rasamedicarelocatorbot"  
  webhook_url: "https://rasamasterclass.com/webhooks/telegram/webhook"
```

Save the changes and then restart the docker container:

```
sudo docker-compose restart
```

We can now chat with the medicare locator assistant on Telegram:



CHAPTER 11: MESSAGING CHANNELS

Webchat

To connect your assistant to a website, you'll need a plugin to create the chat widget—a chat interface where users can talk to your assistant. We'll use an open source option called [Rasa WebChat](#). It uses SocketIO to send and receive messages in real time.

If you don't have a website already, you can try out the WebChat widget with a simple hello world example—a basic HTML document.

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Much like we did when setting up Telegram, we'll need to add configuration details for the SocketIO channel to the `credentials.yml`. Open the `credentials.yml` file in the editor and add the following:

```
socketio:
  user_message_evt: user_uttered
  bot_message_evt: bot_uttered
  session_persistence: true
```

Here, `user_message_evt` and `bot_message_evt` define the event names used by the Rasa dialogue management engine when sending or receiving messages over SocketIO. The third configuration, `session_persistence`, determines how SocketIO should handle session management. By default, the session restarts on every page reload. By setting this parameter to "true" you can persist the session until the browser or tab is closed, or until a specific event is called to clear the session storage.

CHAPTER 11: MESSAGING CHANNELS

Once the changes have been saved to credentials.yml, you can embed the widget by adding a snippet of HTML between the <body></body> tags of your website:

```
<div id="webchat"/>
<script
src="https://storage.googleapis.com/mrbot-cdn/webchat-latest.js"></script>
// Or you can replace latest with a specific version
<script>
  WebChat.default.init({
    selector: "#webchat",
    initPayload: "/get_started",
    customData: {"language": "en"}, // arbitrary custom data. Stay
minimal as this will be added to the socket
    socketUrl: "http://localhost:5500",
    socketPath: "/socket.io/",
    title: "Title",
    subtitle: "Subtitle",
  })
</script>
```

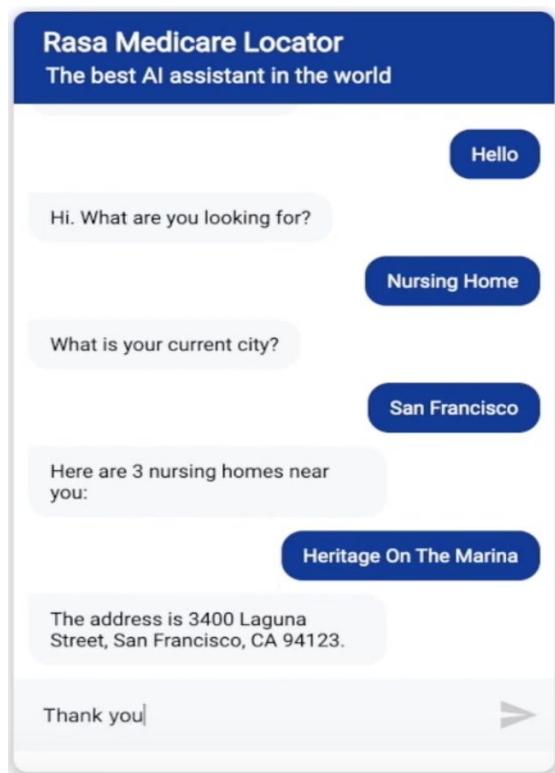
There are three values you'll want to change in the configuration object:

socketURL	Your Rasa X server URL
title	The assistant's name, e.g. Rasa Medicare Locator
subtitle	A description of your assistant, or any additional details you'd like users to see when they begin a conversation

CHAPTER 11: MESSAGING CHANNELS

```
1 <html>
2 <header><title>This is title</title></header>
3 <body>
4 Hello world
5
6 <div id="webchat"/>
7 <script src="https://storage.googleapis.com/mrbot-cdn/webchat-latest.js"></script>
8 // Or you can replace latest with a specific version
9
10 <script>
11   WebChat.default.init({
12     selector: "#webchat",
13     initPayload: "/get_started",
14     customData: {"language": "en"}, // arbitrary custom data. Stay minimal as this will be added to the
15     socketUrl: "https://rasamasterclass.com",
16     socketPath: "/socket.io/",
17     title: "Rasa Medicare Locator",
18     subtitle: "The best AI assistant in the world",
19   })
20 </script>
21
22 </body>
23 </html>
```

Test the new channel by opening the web page in a browser and chatting with the assistant.



CHAPTER 11: MESSAGING CHANNELS

Working with Messaging Channels

A single Rasa assistant can be connected to multiple channels—in fact, you may find it easier to configure the connections for multiple assistants at once in your `credentials.yml` file rather than connecting one at a time.

Conversations users have with the assistant—across all connected channels—are collected in Rasa X for review. In the Rasa X Conversations screen, each conversation is labeled with the channel it originated from.

The screenshot shows the Rasa X interface with the title "rasamasterclass.com/conversations". On the left, there's a sidebar with icons for user, socket, telegram, and Tester. The main area is titled "Conversations (7)" and shows a list of messages. A red box highlights the first three messages, which are from the "Tester" channel. The messages are:

- 6:40 pm, 30 Jan 2020 socketio
- 12:56 pm, 30 Jan 2020 telegram
- 3:47 pm, 18 Dec 2019 Tester

Below these, there are messages from the "Demobot" channel:

- 3:47 pm, 18 Dec 2019 Tester
- 3:46 pm, 18 Dec 2019 Tester
- 3:06 pm, 18 Dec 2019 Tester
- 4:37 pm, 16 Dec 2019 Tester

The right side of the screen shows a detailed view of a message from the "Tester" channel. It includes a timestamp (3:47 pm, 18 Dec 2019), the user icon, the message text "Hi. What are you looking for?", and a list of buttons: Hospital, Nursing Home, and Home Health Agency. Below the message is the Rasa code: `/inform{"facility_type": "b27b-2uc7"}`. To the right, the Demobot's response is shown: "and 485bad637d844d1dbe11c687021f39bf". At the bottom right, there's a link "Select a message to view more".

CHAPTER 11: MESSAGING CHANNELS

Another useful feature when working with multiple channels is the ability to specify [channel-specific utterances](#). Different messaging channels may come with different UI messages you can use, or you may want the assistant to use different responses depending on the channel. To use channel-specific utterances, add the channel key to your response template in domain.yml:

```
responses:  
  utter_ask_game:  
    - text: "Which game would you like to play?"  
      channel: "slack"  
      custom:  
        - # payload for Slack dropdown menu to choose a game  
    - text: "Which game would you like to play?"  
      buttons:  
        - title: "Chess"  
          payload: '/inform{"game": "chess"}'  
        - title: "Checkers"  
          payload: '/inform{"game": "checkers"}'  
        - title: "Fortnite"  
          payload: '/inform{"game": "fortnite"}'
```

CHAPTER 11: MESSAGING CHANNELS

Conclusion

In this episode of the Rasa Masterclass, our assistant finally made the leap from the development and testing stage to interacting with the outside world. Our assistant can now have conversations with real users on real messaging platforms.

Not only is this a gratifying step, it's also crucial for the continued improvement of the assistant. Conversations with real users provide valuable insights and training data. As you review conversations in Rasa X, you should use these conversations to add new training examples and determine the direction of future development. For example, you may decide to add a new intent or action, make adjustments to machine learning algorithms—or even change the scope of the assistant's domain.

As a next step, try connecting your assistant to one of the messaging channels listed in the documentation, and spend some time analyzing your assistant's interactions in Rasa X. When you're ready, meet us back here for the next episode of the Rasa Masterclass.

Additional Resources

- [Messaging and Voice Channels](#) (Rasa docs)
- [Channel-specific Responses](#) (Rasa docs)
- [IP Addresses](#) (Google Cloud docs)
- [DNS Basics](#) (Google support)
- [Certbot](#) (Certbot website)



CHAPTER TWELVE

DEPLOYING ON KUBERNETES

DEPLOYING ON KUBERNETES



Introduction

In [episode 9](#) of the Rasa Masterclass, we learned how to deploy Rasa X to a server using the docker-compose quick installation method. In episode 12, we'll cover a more advanced deployment scenario: deploying Rasa X to a cluster environment.

The docker-compose method is a great deployment option if your goal is to get started quickly or build a Rasa assistant for personal use. However, if your goal is to build a high-availability, mission-critical assistant, a scalable cluster architecture is essential.

Rasa supports cluster deployments on both Kubernetes and OpenShift. In this episode, we'll walk through deploying Rasa X to Kubernetes step-by-step, using Helm to configure the deployment.

CHAPTER 12: KUBERNETES

What is Kubernetes?

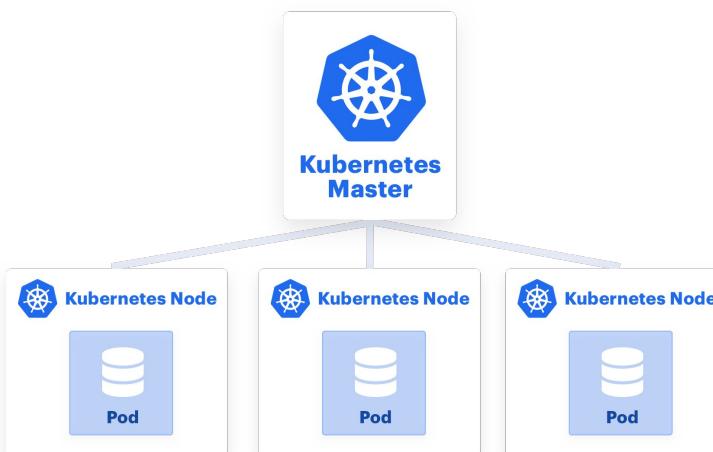
Kubernetes is an open source system for deploying, scaling, and maintaining containerized applications. Whereas the docker-compose method deploys Rasa X to a single container running on a single machine, Kubernetes deploys Rasa X to multiple containers running on multiple host machines, known as nodes. This distributed approach means that there's always a backup system in place—if one container in the cluster goes down, another container can start up to take its place.

Let's take a closer look at the Kubernetes architecture. A pod represents either a single container or multiple containers that share resources, and it's the smallest unit of deployment in Kubernetes. Any container runtime can be used with Kubernetes, but the most common choice is Docker.

A Deployment is a YAML file that declares the deployment configuration. For each pod, it specifies the container image(s), and the number of copies, or replicas, that should be running. The configuration laid out in the deployment YAML file is what's considered the “desired state.”

The Kubernetes cluster service takes the deployment YAML file and schedules the desired number of pods on the host machines, or nodes. It's the scheduler's job to maintain the desired state even if a host machine goes offline. If one machine goes down, the scheduler spins up new pods and redistributes them on the remaining machines to maintain the number of pods defined in the deployment.

It's this self-healing ability that makes cluster deployments so resilient. Similarly, load balancers distribute traffic across the pods and nodes, to automatically scale in response to increased usage.



CHAPTER 12: KUBERNETES

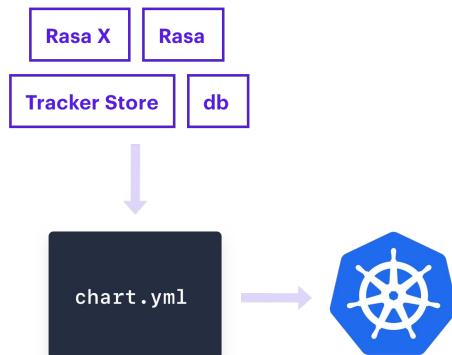
Kubectl

Kubectl is a command line tool for interacting with the Kubernetes cluster services. Kubectl works by sending HTTP requests to the Kubernetes API server, allowing the user to control the cluster from the command line. In this episode we'll use kubectl to access and work with our cluster.

Helm

Helm is an application package manager for Kubernetes, and its purpose is to simplify the process of installing an application on a Kubernetes cluster. A Helm chart is a set of instructions for how an application should be installed on a cluster. It describes the application, including the services and packages it needs to run, and its default configuration values.

Rasa provides an [open source Helm chart](#), which we'll use in this episode to install Rasa X on our cluster.



Prerequisites

Before beginning this tutorial, you'll need to install the following prerequisites on your local machine:

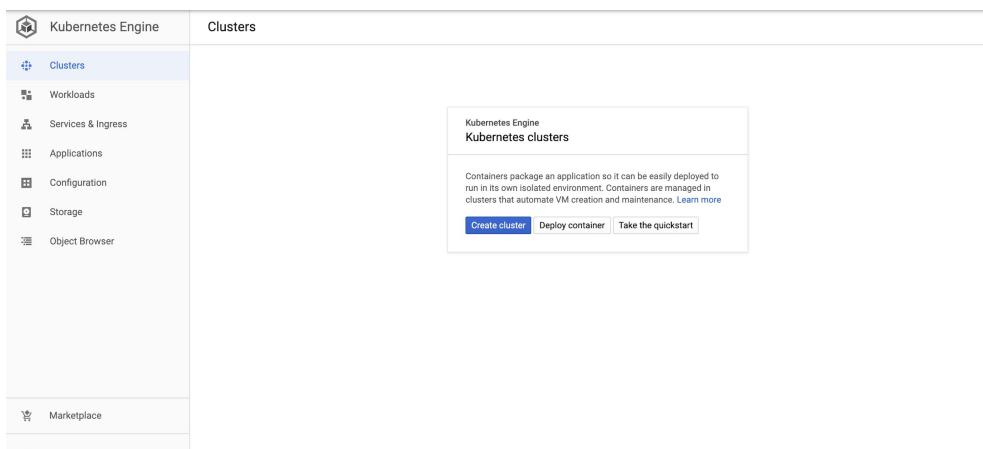
- [gcloud SDK](#)
- [Docker](#) + [create a Docker Hub account](#)
- [Helm](#)

CHAPTER 12: KUBERNETES

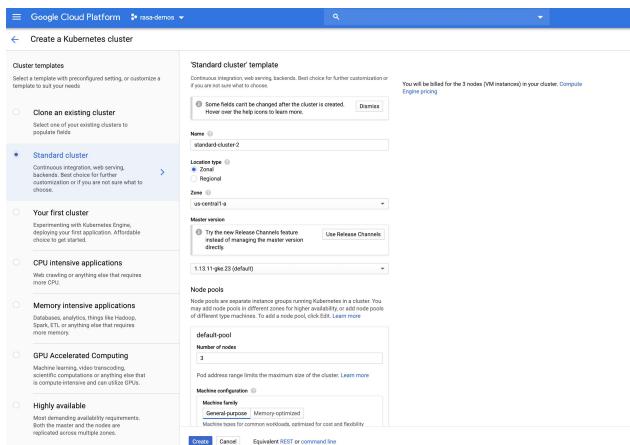
Create a new Kubernetes Cluster

In this tutorial, we'll use Google Kubernetes Engine to create a new cluster, but you can install Rasa on a Kubernetes cluster on any host you choose: Digital Ocean, Microsoft Azure, Amazon Elastic Kubernetes Service, or even a bare metal server.

From the [Google Cloud Console](#), navigate to Kubernetes Engine>Clusters. Click the button to create a new cluster.



You can change the location zone and the machine type if you wish, but in our case, we'll use the standard, default configuration to create our cluster.



Click **Create** to spin up the cluster.

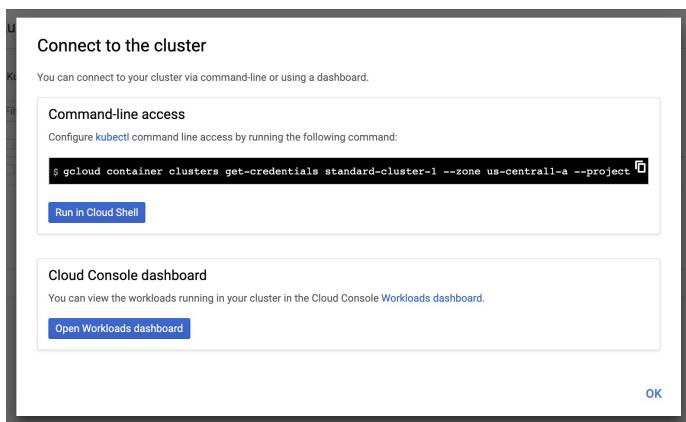
CHAPTER 12: KUBERNETES

Connect to the Cluster

To connect to the cluster, we'll run a gcloud SDK command in the terminal. Note that we'll be using the terminal on your local machine, although you can use the cloud terminal embedded in the browser if you wish.

Before running this command be sure to install the gcloud SDK on your computer. See the Prerequisites section for a quick link to the installation instructions.

Click the Connect button for your cluster and copy the gcloud command.



Run the gcloud command in your terminal, e.g.:

```
gcloud container clusters get-credentials standard-cluster-1 --zone us-central1-a --project replicated-test
```

Once you've connected, check that Helm is installed by running the following command:

```
helm version
```

The minimum required version for this tutorial is 3.0.

```
Fetching cluster endpoint and auth data.
kubeconfig entry generated for standard-cluster-1.

Updates are available for some Cloud SDK components. To install them,
please run:
$ gcloud components update

(base) tremaine:Desktop justest$ helm version
version.BuildInfo{Version:"v3.0.1", GitCommit:"7c22ef9ce89e0ebef7125ba2ebf7d421f3e82ffa", GitTreeState:"clean", GoVersion:"go1.13.4"}
(base) tremaine:Desktop justest$
```

CHAPTER 12: KUBERNETES

Note: Received an error message saying helm is not a recognized command? Check the Prerequisites section of this tutorial for a link to installation instructions, or Mac users can run the brew install helm command.

Next, check to make sure kubectl is connected to the cluster by running:

```
kubectl cluster-info
```

The IP address of the Kubernetes master should match the IP address in the cluster configuration in Google Kubernetes Engine.

```
(base) tremaine:~ juste$ kubectl cluster-info
Kubernetes master is running at https://34.69.97.52
GLBCDefaultBackend is running at https://34.69.97.52/api/v1/namespaces/kube-system/services/default-http-backend:http/proxy
Heapster is running at https://34.69.97.52/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://34.69.97.52/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://34.69.97.52/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy
```

Cluster

Master version	1.13.11-gke.14	Upgrade available
Endpoint	34.69.97.52	Show cluster certificate
Client certificate	Disabled	
Binary Authorization	Disabled	

Run the following command to add the Rasa Helm Chart to your cluster. This prepares us to fetch the chart when we run the command to install later on in this tutorial.

```
helm repo add rasa-x https://rasahq.github.io/rasa-x-helm
```

Let's inspect the list of values you can configure in your deployment. Run this command:

```
helm inspect values rasa-x/rasa-x
```

CHAPTER 12: KUBERNETES

```
# images: Settings for the images
images:
  # pullPolicy to use when deploying images
  pullPolicy: "Always"
  # imagePullSecrets which are required to pull images for private registries
  imagePullSecrets: []

# securityContext to use
securityContext:
  # runAsUser: 1000
  fsGroup: 1000

# nameOverride replaces the Chart's name
nameOverride: ""

# fullnameOverride replace the Chart's fullname
fullnameOverride: ""

# global settings of the used subcharts
global:
  # postgresql: global settings of the postgresql subchart
  postgresql:
    # postgresqlUsername which should be used by Rasa to connect to Postgres
    postgresqlUsername: "postgres"
    # postgresqlPassword is the password which is used when the postgresqlUsername equals "postgres"
    postgresqlPassword: "<password>"
    # existingSecret which should be used for the password instead of putting it in the values file
    existingSecret: ""
    # postgresDatabase which should be used by Rasa X
    postgresDatabase: "rasa"
    # servicePort which is used to expose postgres to the other components
    servicePort: 5432
    # host: postgresql.hostedsomewhere.else
```

Configuration is optional, and we'll go with the default values for most of these. However, for values you do want to change, you'll need to create an override file with a .yml extension. We'll do just that to set the username and password for Rasa X. Create a new file in the directory of your choice and call it values.yml:

```
touch values.yml
```

Open the file in the editor:

```
nano values.yml
```

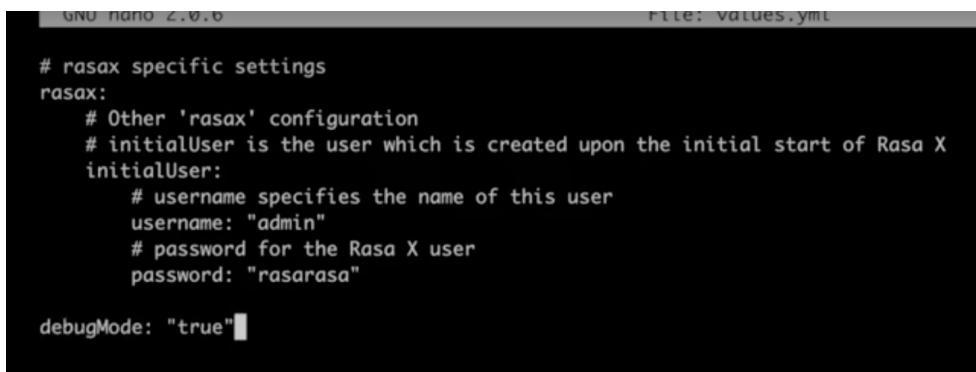
Paste the following into the file contents, replacing the <password> placeholder with an 8-character password of your choice.

```
# rasax specific settings
rasax:
  # Other 'rasax' configuration
  # initialUser is the user which is created upon the initial start
  # of Rasa X
  initialUser:
    # username specifies the name of this user
    username: "admin"
    # password for the Rasa X user
    password: "<safe credential>"
```

CHAPTER 12: KUBERNETES

On a new line, add the following to enable error logging:

```
debugMode: "true"
```



```
GNU nano 2.0.6          File: values.yml

# rasax specific settings
rasax:
  # Other 'rasax' configuration
  # initialUser is the user which is created upon the initial start of Rasa X
  initialUser:
    # username specifies the name of this user
    username: "admin"
    # password for the Rasa X user
    password: "rasarasa"

debugMode: "true"
```

Save and close the file.

Set up the custom action server

Our medicare locator assistant requires a custom action server, so our next step will be to set up a container to run custom actions. We'll do that by creating a custom Docker image and referencing it in our cluster configuration.

Create a directory for your action server, anywhere on your local machine:

```
mkdir action_server
```

```
cd action_server
```

Inside the directory, create an `__init__.py` file and an `actions.py` file:

```
touch __init__.py
touch actions.py
nano actions.py
```

Paste the code from your assistant's `actions.py` file into the `actions.py` file that you just created, save, and close the file.

CHAPTER 12: KUBERNETES

Create the Dockerfile

If your custom actions require any libraries that aren't included in the rasa-sdk image, you'll need to include them in a requirements.txt file. The medicare locator uses the requests library, so we'll need to specify that as a dependency in the image. Create the requirements.txt file:

```
touch requirements.txt
```

Open it in the editor:

```
nano requirements.txt
```

And paste the following into the contents:

```
requests~=2.21.0
```

Save and close the file.

Finally, we'll create a Dockerfile. The Dockerfile contains the commands needed to build the Docker image to our specifications.

Create a new file called Dockerfile and open it in the editor:

```
touch Dockerfile
```

```
nano Dockerfile
```

CHAPTER 12: KUBERNETES

Paste the following into the file contents:

```
FROM rasa/rasa-sdk:latest #lock the version  
  
USER root  
  
WORKDIR /app  
  
COPY actions.py /app  
  
COPY requirements.txt /app  
  
RUN pip3 install -r requirements.txt  
  
CMD ["start", "--actions", "actions"]  
  
USER 1001
```

Let's break down what's happening in the Dockerfile. The first thing we do is specify the parent image. This is the image we'll use as a starting point, before adding the specifications custom to our application on top. In this case, we'll use the latest version of the rasa/rasa-sdk image as the parent.

Next, we'll set the user group permissions. If your custom action contains any external libraries, like ours does, you may need to set the USER group to root.

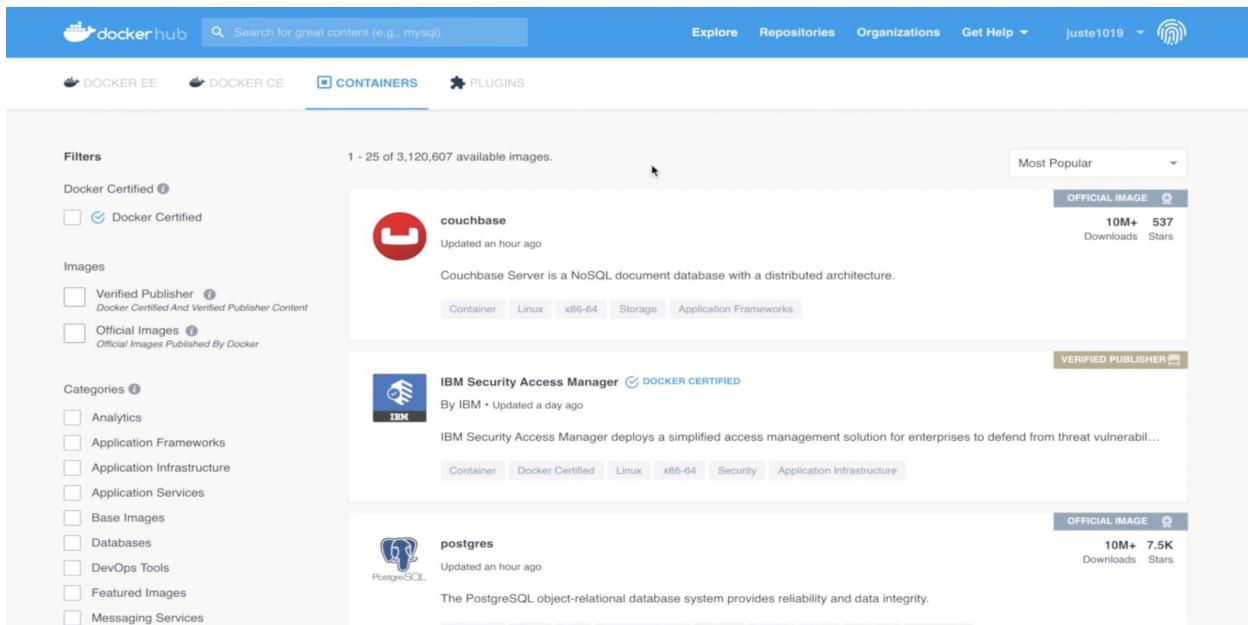
Then, we specify the working directory, which we'll set to app. The lines that follow add the necessary files to the directory: requirements.txt and actions.py. We install the necessary dependencies, and then run the command to spin up the custom action server, referencing the module of our custom action as a parameter. Since we have actions.py in our working directory, all we have to do is reference the module name actions. Lastly, we re-set the user group to default permissions.

Save and close the file.

CHAPTER 12: KUBERNETES

Build and push the image

Now, we're ready to build the image and push it to a container registry. We'll use Docker Hub in this tutorial, but you can use the registry of your choice.



Make sure you're logged into your Docker Hub account (see Prerequisites for a quick link to set this up) and run the following command to build the image:

```
docker build . -t <dockerusername>/<imagename>:<tag>
```

The period in the command means that the Dockerfile is in our current directory. Be sure to replace the <dockerusername> placeholder with your Docker Hub username and <imagename>:<tag> with a name and tag, something like customactionserver:1

Once the build finishes, we can test it by running:

```
docker images
```

```
docker run <image ID>
```

CHAPTER 12: KUBERNETES

Once we've confirmed it's running, we're good to push the image to Docker Hub with the following command:

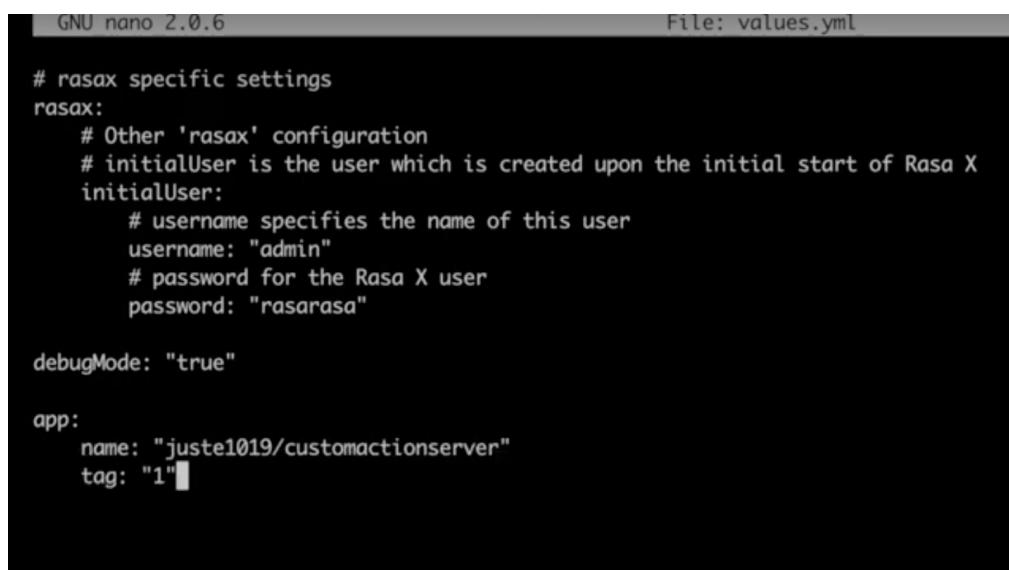
```
docker push <dockerusername>/<imagename>:<tag>
```

Pushing the image to the registry makes the image available to reference in our values.yml file. Here, we're making the image publicly available on Docker Hub, but you also have the option to make the image private. To access a private image, you'll need to configure an image pull secret, following the [Kubernetes guide](#). You can then add the secret to your values.yml file:

```
# images: Settings for the images
images:
  # imagePullSecrets which are required to pull images for private
  # registries
  imagePullSecrets:
    - <name of your pull secret>
```

Since we're using a public image, we can skip creating the image pull secret and simply reference the image using this configuration in the values.yml file:

```
app:
  name: "name of your image"
  tag: "tag you want to use"
```



```
GNU nano 2.0.6                                     File: values.yml

# rasax specific settings
rasax:
  # Other 'rasax' configuration
  # initialUser is the user which is created upon the initial start of Rasa X
  initialUser:
    # username specifies the name of this user
    username: "admin"
    # password for the Rasa X user
    password: "rasarasa"

  debugMode: "true"

app:
  name: "juste1019/customactionserver"
  tag: "1"
```

CHAPTER 12: KUBERNETES

Deploy Rasa X

We recommend installing Rasa X in a separate namespace on your Kubernetes cluster. Namespaces are a way to divide a cluster among multiple users.

Create a new namespace called rasaxkube with this command:

```
kubectl create ns rasaxkube
```

Then, install Rasa X with these two commands:

```
helm repo update
```

```
helm install --generate-name --namespace rasaxkube --values values.yml rasa-x/rasa-x
```

Here, we're pointing to the override YAML file, values.yml, which is in our current directory.

Once the installation is complete, you can check the status of the pods by running:

```
kubectl get pods --namespace=rasaxkube
```

```
Thanks for installing Rasa X !

Check out the Rasa X docs here for more help:
https://rasa.com/docs/rasa-x/
(base) tremaine:Desktop juste$ kubectl get pods --namespace=rasaxkube
NAME                               READY   STATUS    RESTARTS   AGE
rasa-x-1581694459-app-68bdddd65d-h92s6   1/1     Running   0          103s
rasa-x-1581694459-duckling-6d965f896c-qfw4n   1/1     Running   0          103s
rasa-x-1581694459-event-service-7cd668855c-4cgl9   1/1     Running   1          103s
rasa-x-1581694459-nginx-6479f77f9d-swvcr   1/1     Running   0          103s
rasa-x-1581694459-postgresql-0   1/1     Running   0          101s
rasa-x-1581694459-rabbit-0   1/1     Running   0          102s
rasa-x-1581694459-rasa-production-6c86544d48-dvp7l   1/1     Running   0          103s
rasa-x-1581694459-rasa-worker-5c976f9c7c-p2xsw   1/1     Running   0          103s
rasa-x-1581694459-rasa-x-6dd79dc849-mg6qg   1/1     Running   2          102s
rasa-x-1581694459-redis-master-0   1/1     Running   0          101s
(base) tremaine:Desktop juste$ kubectl logs --n
```

CHAPTER 12: KUBERNETES

If you notice that some of the pods are restarting or pending, you can inspect it further by requesting the logs for that pod, for example:

```
kubectl logs rasa-x-1580751734-app-5d4c58c545-dswxn
```

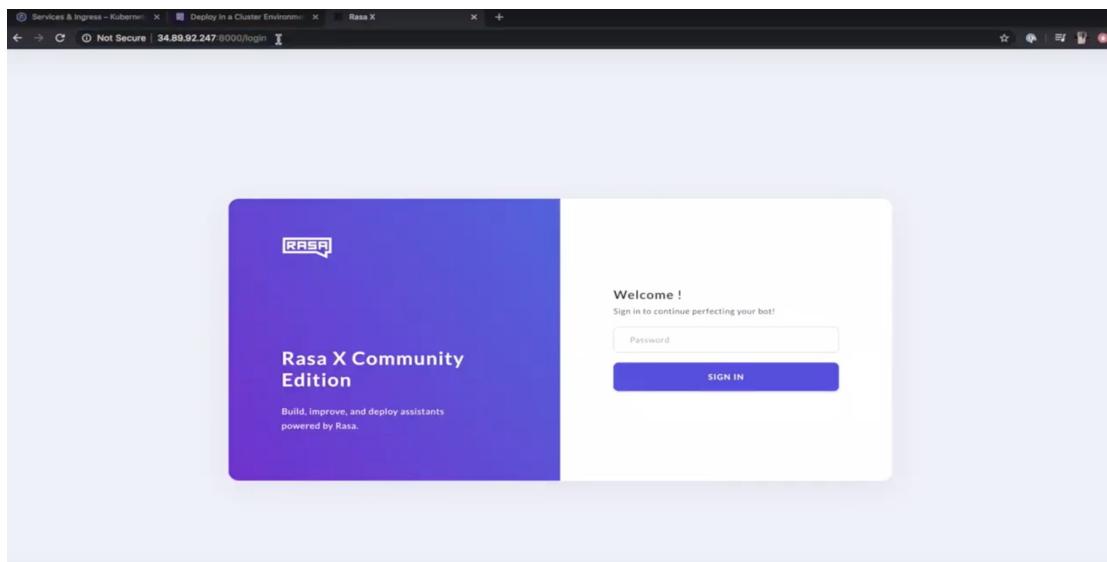
Rasa is now up and running on the Kubernetes cluster! To access it in the browser, find the IP address by running:

```
kubectl get services
```

Look for the Load Balancer service, which lists an external IP address, along with the port the service is running on. Here, it's 34.89.92.247:8000

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
rasa-x-1581694459-app	ClusterIP	10.0.4.240	<none>	5055/TCP, 80/TCP	5m42s
rasa-x-1581694459-duckling	ClusterIP	10.0.8.169	<none>	8000/TCP	5m42s
rasa-x-1581694459-nginx	LoadBalancer	10.0.10.252	34.89.92.247	8000:30741/TCP	5m42s
rasa-x-1581694459-postgresql	ClusterIP	10.0.15.146	<none>	5432/TCP	5m42s
rasa-x-1581694459-postgresql-headless	ClusterIP	None	<none>	5432/TCP	5m42s
rasa-x-1581694459-rabbit	ClusterIP	10.0.13.172	<none>	4369/TCP, 5672/TCP, 25672/TCP, 15672/TCP	5m42s
rasa-x-1581694459-rabbit-headless	ClusterIP	None	<none>	4369/TCP, 5672/TCP, 25672/TCP, 15672/TCP	5m42s
rasa-x-1581694459-rasa-production	ClusterIP	10.0.14.106	<none>	5005/TCP	5m42s
rasa-x-1581694459-rasa-worker	ClusterIP	10.0.13.181	<none>	5005/TCP	5m42s
rasa-x-1581694459-rasa-x	ClusterIP	10.0.4.239	<none>	5002/TCP	5m42s
rasa-x-1581694459-redis-headless	ClusterIP	None	<none>	6379/TCP	5m42s
rasa-x-1581694459-redis-master	ClusterIP	10.0.2.39	<none>	6379/TCP	5m42s

Go to the IP address in your browser with the port number appended (remember to reset the protocol to http), and you should now be able to log in using the password you set in your values.yml file.



CHAPTER 12: KUBERNETES

Next steps

Now that Rasa X has been deployed on Kubernetes, you can go through the process we outlined in earlier episodes to finish setting up your assistant. That includes:

- Connecting [integrated version control](#)
- Configuring DNS
- Installing an SSL certificate
- Connecting messaging channels.

Refer to the documentation for your cloud provider to point a domain name to your cluster. You can find the docs for Google Kubernetes Engine [here](#).

We recommend enabling SSL on your cluster's ingress controller or the load balancer serving the ingress. Check the documentation for your cloud host for detailed instructions. You can find the docs for Google Kubernetes Engine [here](#).

When running your assistant on Kubernetes, messaging channels are configured by adding credentials to the values.yml override file, in this format:

```
# rasa: Settings common for all Rasa containers
rasa:
  # additionalChannelCredentials which should be used by Rasa to
  connect to various
  # input channels
  additionalChannelCredentials: |
    socketio:
      user_message_evt: user_uttered
      bot_message_evt: bot_uttered
      session_persistence: true/false
```

CHAPTER 12: KUBERNETES

Conclusion

In this episode, we learned how to deploy Rasa X on a scalable cluster environment, using Kubernetes. This setup provides the reliable infrastructure our assistant will need to handle increased traffic and usage in production.

And although the deployment is different, the process for improving the assistant is the same process we covered in [episode 10](#): review conversations to identify places where the assistant performs well or not so well, and make updates based on your findings.

And with that, we conclude the Rasa Masterclass. You can watch our [Rasa Masterclass Recap](#) episode to review all of the topics we covered over the course of the series.

This is the end for the Rasa Masterclass, but it's just the beginning of our efforts to help developers build great contextual assistants. Let us know what topics you'd like to see covered in future tutorials by commenting on the Rasa Masterclass Recap video.

Additional Resources

- [Google Kubernetes Engine documentation](#) (Google Cloud Platform)
- [What is Kubernetes](#) (Kubernetes docs)
- [Deploy in a Cluster Environment](#) (Rasa docs)
- [Build and Run Your Image](#) (Docker docs)
- [Helm Quick Start](#) (Helm docs)

CONGRATS!

You've completed the Rasa Masterclass.

Ready to share what you build? Connect with the
Rasa Community in our forum.

[JOIN THE FORUM](#)