Rasa Certification Workshop

May 2020

Mady Mantha, Juste Petraityte, Karen White

Agenda

Day 1: Deep dive into NLU and dialogue management with Rasa Open Source, livecoding and testing

Day 2: Deep dive into DIET (Dual Intent and Entity Transformer) and TED (Transformer Embedding Policy) and create an MVP assistant

Day 3: Adding custom actions and implementing forms

Day 4: Deploying and improving your assistant using Rasa X

Day 5: Recap and certification

*More info blablabla

The Rasa Team

Mady Mantha
Sr. Technical Evangelist

Juste Petraityte
Head of Developer Relations

Karen White
Developer Marketing Manager

How to get help

- Please ask your questions in the **#workshop-help** Slack channel rather than the Zoom chat. Slack is the place the Rasa team will be monitoring most closely.
    - Karen, Mady, and Juste will be in Slack answering questions, as well as Arjaan, Melinda, and Ella from our Customer Success Engineering team
- Monday - Friday, the Rasa team will be dedicating time to answering your questions in Slack from 4 pm - 6 pm CEST (Central European Summer Time)
- Feel free to ask questions outside of these hours, but responses may be a little slower
- A note on time zones:
    - The Rasa team is based across the US and in Berlin. We'll do our best to answer questions within team members' working hours, but please keep in mind, some discussions may need to take place async rather than in real time.

Adding custom actions and implementing forms

Agenda

Day 1: Deep dive into NLU and dialogue management with Rasa Open Source

Day 2: Create an MVP assistant, end-to-end testing, DIET architecture, and TED policy

Day 3: Adding custom actions and implementing forms

Day 4: Deploying and improving your assistant using Rasa X

Day 5: Recap + Q&A Panel

How to get help

- Please ask your questions in the **#workshop-help** Slack channel rather than the Zoom chat. Slack is the place the Rasa team will be monitoring most closely.
  - Karen, Mady, and Juste will be in Slack answering questions, as well as Arjaan, Melinda, and Ella from our Customer Success Engineering team
- Tuesday - Friday, the Rasa team will be dedicating time to answering your questions in Slack from 8am - 10:00 am PDT (12:00 pm - 2 pm EDT)
  - Feel free to ask questions outside of these hours, but responses may be a little slower
- A note on time zones:
  - The Rasa team is based across the US and in Berlin. We'll do our best to answer questions within team members' working hours, but please keep in mind, some discussions may need to take place async rather than in real time.

Day 3 Roadmap

Part 1:

- Quick overview of what you have learned yesterday

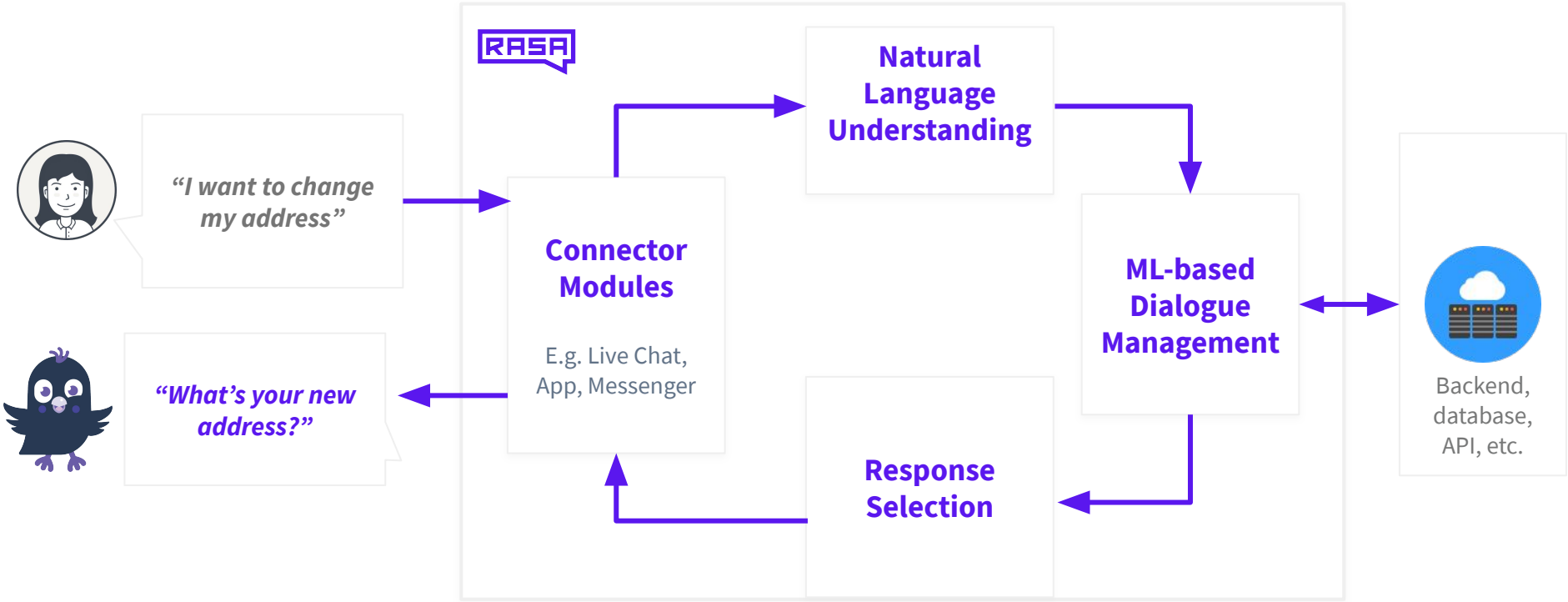
Part 2:

- Rasa events and slots: talk and coding together
- Custom actions: talk and coding together
- Short break 🧁

Part 3:

- Forms: talk and coding together
- Recap and Q&A

Rasa Open Source



Project setup: Files

`__init__.py`	an empty file that helps python find your actions
`actions.py`	code for your custom actions
`config.yml`	configuration of your NLU and Core models
`credentials.yml`	details for connecting to other services
`data/`**`nlu.md`**	your NLU training data
`data/`**`stories.md`**	your stories
`domain.yml`	your assistant's domain
`endpoints.yml`	details for connecting to channels like fb messenger
`models/<timestamp>.tar.gz`	your initial model

NLU Pipeline

Defines how a user message is processed & what information is extracted

```
language: "en"

pipeline:
  - name: "Whitespacetokenizer"
  - name: "ConveRTFeaturizer"
  - name: "RegexFeaturizer"
  - name: "LexicalSyntacticFeaturizer"
  - name: "CountVectorsFeaturizer"
  - name: "DietClassifier"
    epochs: 100
```

Components

NLU Pipeline: Components

Built-in component types:

- Tokenizers
- Word Vector Sources
- Text Featurizers
- Entity Extractors
- Intent Classifiers
- Response Selectors

You can also add custom components for e.g.

- Sentiment analysis
- Spell checking

Rule Based Policies

- **MemoizationPolicy:** Memorizes your stories
  - Makes predictions based on your max_history value
- **AugmentedMemoizationPolicy:** Memorizes your stories with a twist
  - As above, but if no match is found, it will forget certain events until a match is found
- **FormPolicy:** Fills required slots
  - Asks for information until all required slots are filled
- **MappingPolicy:** Intents trigger actions
  - Will execute the specified action regardless of context unless superseded by another policy
- **FallbackPolicy**: Failing gracefully
  - If model confidence is below a certain value, it will send the user a “fallback message”, e.g. “Sorry, I didn’t understand you”
- **TwoStageFallbackPolicy**: Failing (more) gracefully
  - Attempts to disambiguate the user’s intent before sending a fallback message

Machine Learning Based Policies

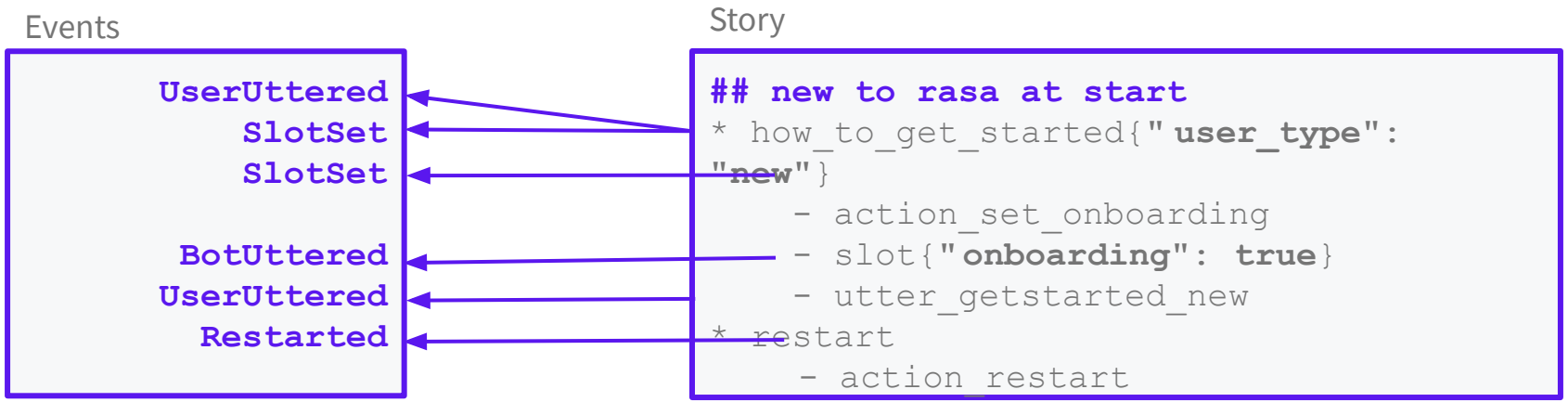
These policies should be used in conjunction with rule-based policies

- **KerasPolicy:** Uses a standard LSTM to predict the next action
  - Learns the patterns of your stories
  - Good for handling stories that don't exactly match your training data
- **TED Policy:** Uses Attention to Handle Uncooperative Dialogue
  - Requires fewer story examples of uncooperative user dialogue
    - e.g. users who go off on tangents instead of providing the requested information
  - Effectively "ignores" irrelevant parts of the dialogue

Rasa events and slots

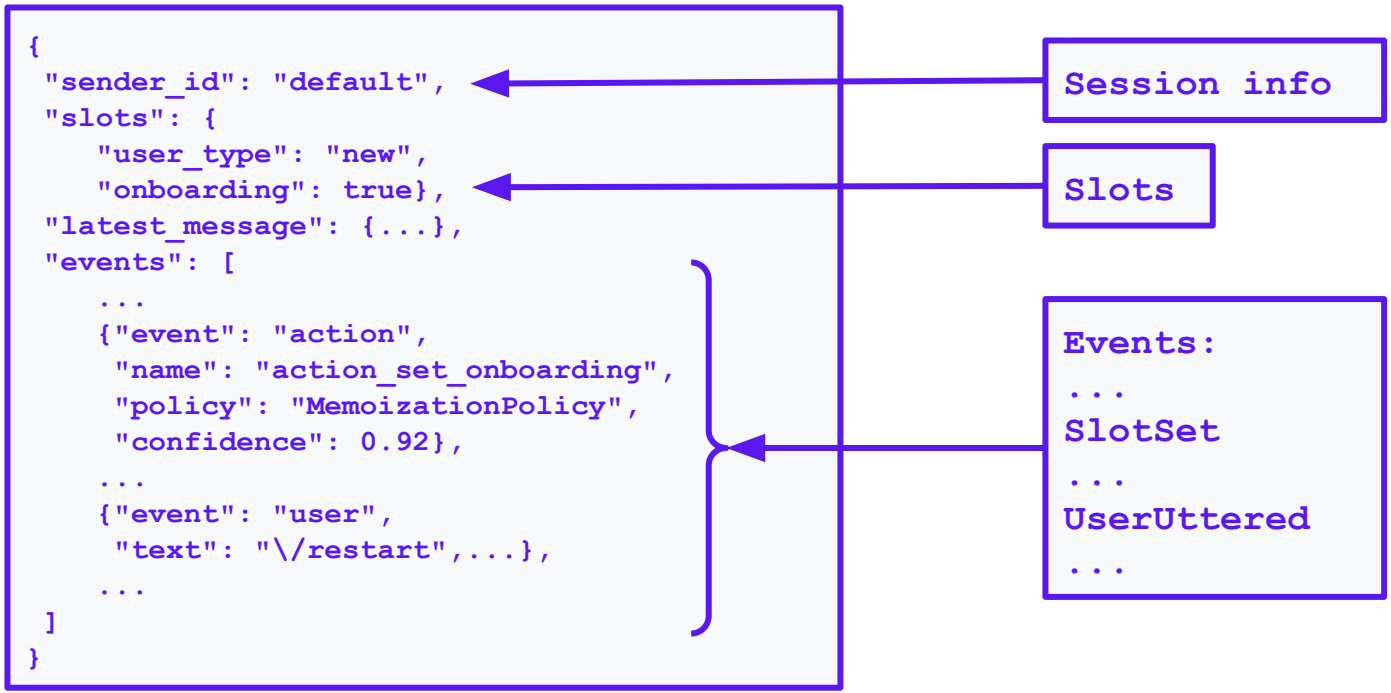
Events

- **Internally**, conversations are represented as a sequence of **events.**
- Some events are automatically tracked



Tracker

- Trackers maintain the **state of a dialogue** between the assistant and the user.
- It keeps track of events, slots, session info etc.



Slots

Your bot's memory

Hi, my name is Tom

Hello! How can I help?

I would like to check my account balance

Sure. Can you tell me your surname?

Hi

Hello! How can I help?

I would like to check my account balance

What is your name?

Slots

Your bot's memory

- Can store:
  - user-provided info
  - info from the outside world
- Can be set by:
  - NLU (from extracted entities, or buttons)
  - Custom Actions
- Can be configured to affect or not affect the dialogue progression

Slots types

Text and List slots influence conversation path based on **whether it is set or not.**

Text Slot

```
text
```

Use For: User preferences where you only care whether or not they've been specified.

Example:

```
slots:
   location:
      type: text
```

"I am based in London"

List Slot

```
list
```

Use For: Lists of values

Example:

```
slots:
   appointment_times:
      type: list
```

"Are there available appointments on Monday or Wednesday this week?"

Core Deep Dive

Slots types

Categorical, Boolean and Float slots influence conversation path based on **the value of the slot.**

Categorical Slot

```
categorical
```

Use For: Slots which can take one of N values

Example:

```
slots:
   price_level:
      type: categorical
      values:
      - low
      - medium
      - high
```

"I am looking for a restaurant in a low price range."

Boolean Slot

```
bool
```

Use For: True or False

Example:

```
slots:
   is_authenticated:
      type: bool
```

Float Slot

```
float
```

Use For: Continuous values

Example:

```
radius:
   type: float
   min_value:  0.0
   max_value:  50.0
```

"Find me a restaurant within 2 mile radius."

RASA

Slots types

Unfeaturized slots don’t have any influence on the dialogue.

Unfeaturized Slot

```
unfeaturized
```

Use For: Continuous values

Example:

```
slots:
   patient_name:
      type: unfeaturized
```

“My name is Sarah.”

Slots

There are a few different ways how slots can be set

- Slots set from NLU

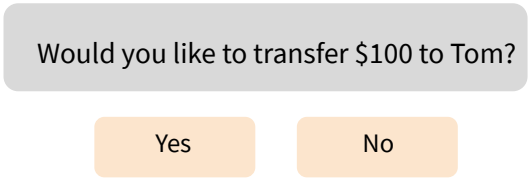
```
## intent:search_transactions
- how much did I spend at [Target](vendor_name) this week?
- what is my typical spending at [Amazon](vendor_name)?
```

Slots

There are a few different ways how slots can be set

- Slots set by clicking buttons

```
utter_ask_transfer_form_confirm:
 -   text: "Would you like to transfer $100 to Tom?"
     buttons:
       -   title: Yes
           payload: /affirm
       -   title: No, cancel the transaction
           payload: /deny
```



Slots

There are a few different ways how slots can be set

- Slots set by custom actions

```
## account balance story
* check_account_balance
    - action_account_balance
    - slot{"account_balance": 1000}
    - slot{"amount_transferred": None}
```

Livecoding

Let’s define the slots for our Financial Bot assistant!

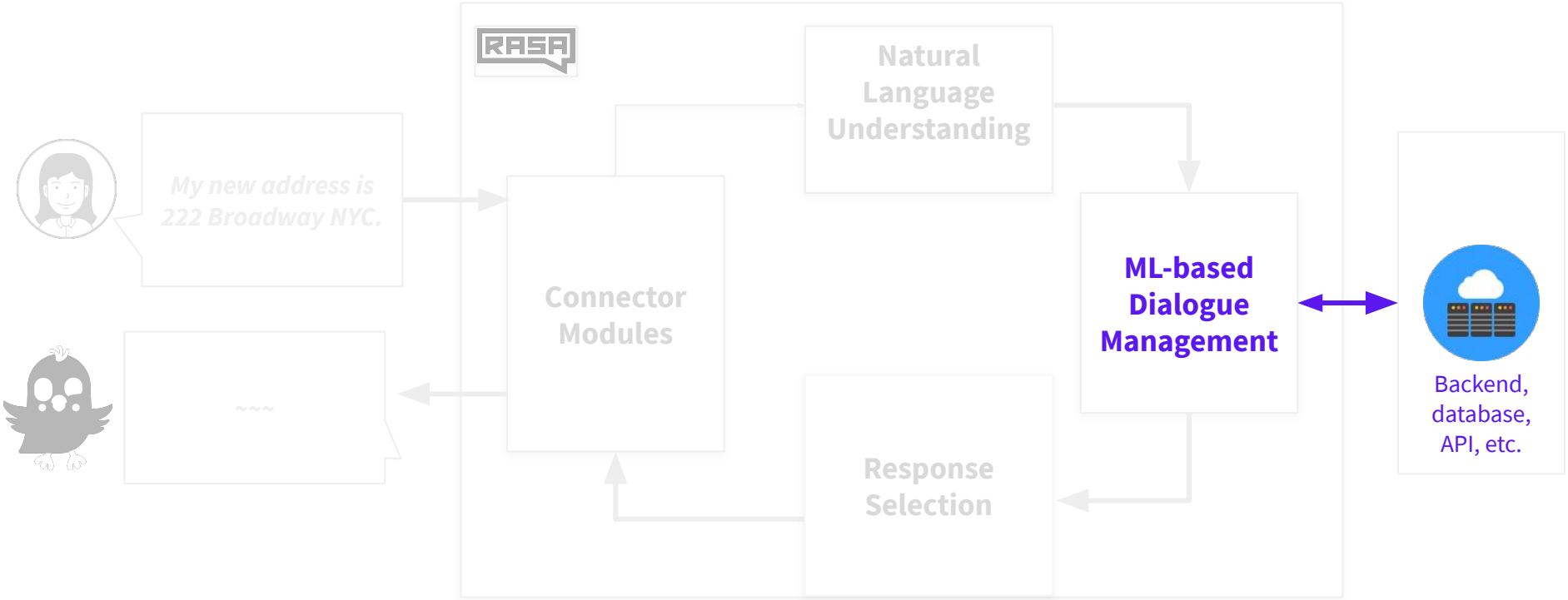
1. Let’s use a sandbox branch for today’s exercise. First, make sure your are in a financial bot directory:
   cd financial-demo

2. Fetch the remote branches:
   git fetch origin

3. Checkout the sandbox branch named “workshop_day_3_sandbox”:
   git checkout -b workshop_day_3_sandbox origin/workshop_day_3_sandbox

Custom actions

Custom Actions: Connecting to the outside world



Actions

Things your bot runs in response to user input.

Four different action types:

- **Utterance actions**: `utter_`
  - send a specific message to the user
  - specified in `responses` section of the domain
- **Retrieval actions**: `respond_`
  - send a message selected by a retrieval model
- **Custom actions**: `action_`
  - run arbitrary code and send any number of messages (or none).
  - return events
- **Default actions**:
  - built-in implementations available but can be overridden
  - E.g. action_listen, action_restart, action_default_fallback

Custom Actions: The Action Server

Custom action code is run by a **webserver** called the **action server**

- **How custom actions get run:**
  - When a custom action is predicted, Rasa will call the **endpoint** to your action server
  - The action server:
    - runs the code for your custom action
    - [optionally] returns information to modify the dialogue state
- **How to create an action server:**
  - You can create an action server in any language you want!
  - You can use the **Rasa-SDK** for easy development in Python

Custom Actions: Examples

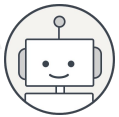
Custom actions can do whatever you want them to! e.g.

- **Send multiple messages**
- **Query a database**
- **Make an API call to another service**
- **Return events e.g.**
  - Set a slot
    - e.g. based on a database query)
  - Force a follow up action
    - force a specific action to be executed next
  - Revert a user utterance
    - I.e. remove a user utterance from the tracker

Custom Action Example: Query a Database

using the Rasa SDK

"Should I also update your mailing address?"



"What do you have on file?"

```
class ActionCheckAddress(Action):
   def name(self) -> Text:
      return "action_check_address"

   def run(self,
           dispatcher: CollectingDispatcher,
           tracker: Tracker,
           domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

      idnum = tracker.get_slot('person_id')
      q = "SELECT Address FROM Customers \
          \WHERE CustomerID=1;'{0}'".format(idnum)
      result = db.query(q)

      return [SlotSet(
          "address",
          result if result is not None else "NotOnFile")]
```

Time for a short break (10 minutes)

Forms

Forms

Rasa Forms allow you to describe all happy paths with a single story

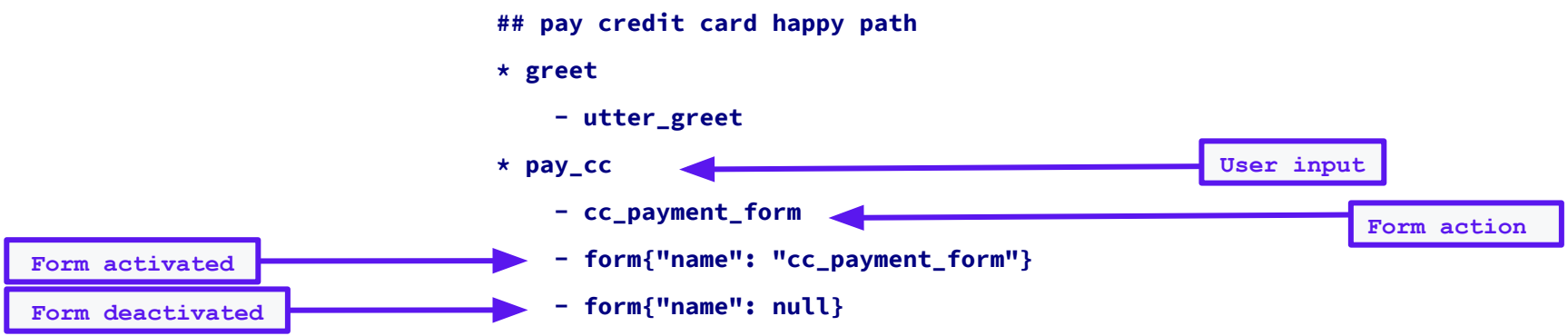
Happy path 1:
U: I would like to make a money transfer
B: Towards which credit card would you like to make a payment?
U: Towards my justice bank credit card
B: How much do you want to pay?
U: $100
B: The transaction has been scheduled.

Happy path 2:
U: I would like to make a 100$ money transfer
B: Towards which credit card would you like to make a payment?
U: Towards my justice bank credit card
B: The transaction has been scheduled.

```
## pay credit card happy path
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
    - form{"name": null}
```

Forms

The structure of the form



Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- name - defines the name of the form actions

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

Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- required slots - sets the list of required slots

```
def required_slots(tracker: Tracker) -> List[Text]:
    " " "A list of required slots to fill in" " "
    return [“credit_card”, “amount_of_money”]
```

Forms

FormAction is a custom action which allows you to set the required slots and determine the behaviour of the form

The main components of the form action:

- submit - defines the output of the form action once all slots are filled in

```
def submit(self, args) -> List[Dict]:
    " " "Defines what the form has to do after all slots are filled in" " "
    dispatcher.utter_message("The payment is confirmed")
    return [AllSlotsReset()]
```

Forms: Custom slot mappings

The slot mappings define how to extract slot values from user inputs.

With slot mappings you can define how certain slots can be extracted from:

- Entities
- Intents
- Support free text input
- Support yes/no inputs

```
def slot_mappings(self) -> List[Text, Union[Dict, List[Dict]]]:
    " " "Defines what the form has to do after all slots are filled in" " "
    return {
        “confirm”: [
                self.from_intent(value:True, intent=”affirm”),
                self.from_intent(value:False, intent=”deny”)
                ]
            }
```

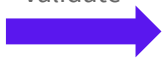
Forms: Validating user input

Validate method in form action allows you to check the value of the slot against a set of values.

I would like to make a $100 payment.

slot: amout_of_money

validate



“Minimum balance”: 85
“Current balance”: 550

Forms

Custom Form Action

Custom Form Action allows you to add new methods to a regular FormAction.

```
class CustomFormAction(FormAction):

    def name(self):
        return “”

    def custom_method(self, args):
            ...
        return
```

Forms

Handling unhappy paths

Users are not always cooperative - they can change their mind or interrupt the form.

- Create stories to handle the interruptions:

```
## pay credit card unhappy path
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
* chitchat
    - utter_chitchat
    - cc_payment_form
    - form{"name": null}
```

Forms

Handling unhappy paths

Users are not always cooperative - they can change their mind or interrupt the form.

- Use action `action_deactivate_form` to handle the situation where users decide not to proceed with the form:

```
## chitchat
* pay_cc
    - cc_payment_form
    - form{"name": "cc_payment_form"}
* stop
    - utter_ask_continue
* deny
    - action_deactivate_form
    - form{"name": null}
```

Forms

How this affect domain and policy configuration?

Domain

```
forms:
  cc_payment_form
```

Policy configuration

```
policies:
  -   name: FormPolicy
```

Livecoding

Let’s implement a form!

A little homework for you

Talk to Financial Bot assistant
http://finbot.rasaworkshop.com/guest/conversations/production/60d883db84d9407da8b36819a0bd62d8

Agenda

Day 1: Deep dive into NLU and dialogue management with Rasa Open Source

Day 2: Create an MVP assistant, end-to-end testing, DIET architecture, and TED policy

Day 3: Adding custom actions and implementing forms

Day 4: Deploying and improving your assistant using Rasa X

Day 5: Recap + Q&A Panel

*More info blablabla