# AI Platform System Requirements

# Introduction

A cheeky definition of Artificial Intelligence (AI) is “the technology just around the corner”. Twenty-five years ago, capabilities such as self-driving cars, live translation of a foreign street sign, and asking a pocket device for directions to the nearest petrol station, would be science fiction. Today, we treat these innovations as more automation than superlative examples of AI. The expectations of AI have moved on:

* Holding a conversation that requires common sense knowledge and remembering previous facts
* Coordinating the events of a group – balancing individual preferences against reaching group consensus
* Reading documents and being able to answer questions related to their contents

Once these skills are mastered, we might expect the goals of AI to move onto making ethical decisions, defining policy, and psychological mirroring.

Despite what seems like venturing into areas that we regard as science fiction (from our current perspective in time), there are many practical benefits of AI-related technologies that can be realized today, including:

* Lowering the cost of customer support by automating routine queries
* Automatically fetching and presenting information in context
* Automating and optimising routine process tasks

Because of AI’s ability to be always on and to navigate multiple digital channels, it has both the promise to augment human activities to get more done for the same cost, and to create a superior customer experience through convenience, timeliness, and personalisation.

The dominant AI-related technologies today include:

* Machine learning, and a branch of machine learning called Deep Learning, which is augmenting data science through automation in feature engineering and selection, as machine learning has been to the automation of decisioning processes
* Bots and Conversational Agents
* Analytical factories capable of processing big data and getting insights to the machine learning-enabled systems and platforms

An investment in AI is also a bet on its potential to disrupt industries and to create new ones. For example, AI could be:

* The next major computing paradigm, changing the way people interact with computers, and in turn, how customers engage with commercial organisations
* A revolution in customer experience, one based on conversations whereby consumers exert more control, e.g. customers will expect to message a company.
* The large-scale automation of some “white collar” activities, creating the same level of disruption as occurred in many “blue collar” jobs such as car manufacturing.

Digital transformation isn’t about converting physical processes and interactions into virtual ones. For example, being able to message a company and get a response when and where convenient is transformative in a way that waiting around to initiate a live chat session is not.

## Commercialisation of AI

The key practical benefits of AI today include:

* An always-on customer agent providing instant response with knowledge of the customer’s complete interaction journey
* Responding to plain language requests rather than forcing customers through a rigid corporate process
* Deeper personalisation of our services and messages

The main benefits of the AI Platform are:

1. Make use of the best AI services available from companies such as Amazon, Google, and IBM
2. Connect to any messaging platform per the preferences of customers
3. Retain and enhance the customer data, which enables sustainable competitive advantage

To achieve these benefits, the system must support the following key requirements:

1. Understand natural voice and text requests (utterances)
   1. Train the system by example using equivalent texts – a system that does not require every word and word combination to be enumerated and tested by explicit rules. The system must be adaptable to variations in word order, keywords vs. full questions, synonyms, tense, common spelling mistakes, and so on.
   2. Determine customer intent (their goal or task).
   3. Extract specific entities required to perform a task, e.g. time or interval, quantity, location, and product type.
2. Engage in a multi-exchange conversation (beyond a simple request-response interaction) to:
   1. Qualify the customer intent or request, e.g. ask additional questions to satisfy information requirements, or to disambiguate meaning
   2. Guide the customer through a process
   3. Respond to questions or concerns
3. Personalise the experience by making use of information that the customer has previously given to us, or which we can infer, such as existing products and accounts, communication preferences, scheduling preferences, receptivity to new or alternate products, etc.



## Natural Language Understanding

This supports training a Language Model to understand and extract information from utterances using natural language text or speech. The system must be adaptable to variations in word order, keywords vs. full questions, synonyms, tense, common spelling mistakes, and so on.

### Intents

Intents are defined through a set of sample utterances. Sample utterances may be created by the implementation team, collected from historical transcripts, or pre-defined vendor datasets or models.

Samples should capture variation in:

* Length
* Formality
* Use of emoticons or stickers

|  |  |
| --- | --- |
| Requirement | Priority |
| Create, modify or delete an Intent | 1 |
| Create, modify or delete a sample Utterance | 1 |
| Publish intents and utterances to the AI Platform | 1 |
| Import utterances from a CSV file | 1 |
| Export intents and utterances to a CSV file | 1 |
| Search for intents or utterances | 2 |
| Classify unlabelled utterances using an unsupervised machine learning classifier | 2 |
| Review unmatched utterances to manually assign an Intent label | 2 |
| Process multiple questions in a single utterance | 3 |

See appendix for sample CSV exchange format.

### Entities

Entities are defined by enumerating their instances and any synonyms of the instance.

For example, an “Account Type” entity may consist of the following instances and synonyms:

|  |  |
| --- | --- |
| Instance | Synonyms |
| Savings |  |
| Checking | Cheque, Check |

Synonyms should capture variation in:

* Spelling
* Colloquial terms

|  |  |
| --- | --- |
| Requirement | Priority |
| Create, modify or delete an Entity | 1 |
| Create, modify or delete an Instance and set of optional Synonyms | 1 |
| Publish entities and instances to the AI Platform | 1 |
| Import entities and instances from a CSV file | 1 |
| Export entities and instances to a CSV file | 1 |
| Search for entities, instances or synonyms | 2 |

See appendix for sample CSV exchange format.

## Conversational Logic

Conversational logic handles the dialog once the intent is matched. Any unmatched utterance is processed by the default handler, which may, for example, transfer the request to a human agent.

A Conversational Flow defines the handling of customer intents.

The conversational flow resembles a flowchart with important differences:

* The flow doesn’t necessarily begin at the start or top. Conversational logic typically flows from a general understanding to a more specific one until some action can be fulfilled. Depending on the specificity of the utterance, the flow can jump ahead. For example, if I want to check my account balance, the conversational agent will ask which account I want the balance for. However, if I mention “transaction account” in my initial request, the conversational agent can skip that question. Thus, a conversation can start at any point in the flow that matches the entry conditions.
* As explained in the previous point, the flow can skip steps. In addition, to matching a more specific request, the flow will skip questions for which it already knows or can infer the answer based on known customer information and preferences, and previous interactions.
* The customer may initiate a new intent mid-flow, causing a new conversation flow to take over. The current state of every conversation is persisted. Should the customer re-enter a conversation, the flow is picked up at the same point where the customer left.

A Flow consists of Nodes, which are equivalent to steps in the flow, or flow states if we consider the flow as a “state machine”.

Each node describes:

* Entry conditions – the conditions that will trigger this node. The node with the most specific matching conditions will be triggered. For example, a top-level node may be triggered by a matching “balance enquiry” intent, and a child node is triggered by the combination of a matching “balance enquiry” intent and a “savings account” entity. An utterance such as “what is the balance of my savings account?”, which matches both conditions will trigger the child node.
* An optional Action such as an API call or executing some code.
* A Reply such as a simple text reply or a Card that incorporates rich media and “quick reply” buttons.

A Flow also consists of Links that connect one node to another. Links provide useful visual information to indicate a hierarchy from the general to the specific, or a natural sequence of steps in the conversation. They also specify soft constraints to optimize resolution of the next step in the conversation. However, keep in mind that jumps in the flow can occur based on pattern matching of the entry conditions.

The flow is a Directed Acyclic Graph (DAG).

### Flow Procedure

1. Utterance received
2. If in ‘Lookup Intent’ State:
   1. For each registered ‘Intent Resolver’:
      1. Call ‘Intent Resolver’.
      2. If ‘First Resolution’ Setting and confidence score of response is above ‘Intent Confidence Threshold’ Setting, then use Intent.
   2. If ‘Top Score’ Setting, then use Intent with highest confidence score.
   3. For each Flow, match Intent and/or Entities against root nodes.
   4. If more than one match, then there is a configuration error. Invoke the Fallback Procedure.
   5. Else If no match, then invoke the Fallback Procedure.
   6. Else Execute the actions of the matched node.
   7. If a ‘Quick Reply’, ‘Card’, ‘Prompt’, or ‘Form’ Reply then transition to the ‘No Lookup Intent’ State.
3. If in ‘No Lookup Intent’ State:
   1. If a Prompt was defined in the last reply:
      1. Validate the response using the Prompt and enrich the Conversation Context with structured data extracted from the response
   2. Evaluate the response against the set of nodes linked from the last matched node.
   3. If more than one match, then there is a configuration error. Invoke the Fallback Procedure.
   4. Else If no match, then back out of the conversation, and apply Step 2 as if in ‘Lookup Intent’ State
   5. Else Execute the actions of the matched node.
   6. If a ‘Quick Reply’, ‘Card’, ‘Prompt’, or ‘Form’ Reply then transition to the ‘No Lookup Intent’ State.

Example Walkthroughs

Greeting



### Fallback Procedure

1. Increment ‘Fallback Count’
2. If ‘Fallback Count’ greater than ‘Fallback Count Threshold’ Setting:
   1. Execute

### Entry Conditions

The flow is entered given the interpreted intent of the utterance. In the flow, the node with the most specific matching entry conditions, starting from the “top” of the DAG, is triggered.



Entry conditions may include one or more of the following:

* Intent, e.g. Balance Enquiry
* Mention of Entity Type, e.g. Account
* Mention of Entity Instance, e.g. Savings Account
* Text or regular expression that matches the utterance text
* Custom code that evaluates the utterance and returns a non-empty Context Object.

#### Intent and/or Entity Match

(1)



(2)





#### Text or Regular Expression Rule

This is used when a specific response is expected in answer to the last reply. For example, the time for an account enquiry.

Prompts (see below) can assist here by validating the response for an expected type, such as a ‘yes-no’ or numeric response.

A Text Rule will test whether the response text is equal to the rule text. A more sophisticated comparison can be performed using a regular expression.

A regular expression (regex for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids.

Before text or regex matching, utterances are pre-processed as follows:

* All text is converted to lower case, so "I said" becomes "i said".
* Leading/trailing spaces are trimmed, so " hello " becomes "hello"

This simplifies rule construction.

A match results in the corresponding action being performed and reply sent. A non-match will be handled by the fallback procedure.

#### Code Rule

Custom code can be supplied to perform any logic, including regex matching. The signature of the required function is:

**function entry(text, sticker, emoticon, user, conversation, callback)**

The ‘text’ argument is the raw response text. If the customer responded with a sticker or emoticon, the text translation is provided as the ‘text’ argument, and metadata for the next two arguments depending on type.

Known customer information is provided as the fourth argument, which may include:

* User id
* Profile name
* Real name
* Preferences and analytical features

Conversation-scoped data is provided as the fifth argument.

The sixth argument is a callback function, which must be called at the end. The first argument of the callback is an optional ‘error’ object to communicate an error in processing. The second argument is an optional ‘context’ object, which may contain structured data extracted from the response.

If ‘false’ is returned instead of a context object, the rule will be deemed to be a non-match, and therefore the response is handled by the fallback procedure.

If an ‘error’ object is returned, then the response will be handled by the fallback procedure as well as additional logging of the error.

An ‘error’ object has the following properties:

{

code: "<optional short code>",

description: "<optional technical description>",

message: "<error message>"

}

### Action

An action may be taken in response to a customer utterance or on reaching a point in the conversational flow. Actions include:

* Transferring the conversation to a human agent
* Calling an internal or external API
* Executing custom code
* Enriching the customer or conversation context – creating variables

#### Transferring to an Agent

If a conversation has reached a point that can no longer be automated, it can be transferred to the next available human agent with a custom message.

Transfers may occur implicitly because of failing to handle a response or signs that the customer would prefer to talk with a human agent.

This is an explicit transfer. The supplied message is sent to the receiving agent along with a complete history of interactions and information stored against the user.

#### Calling an API

An API may be called to fulfil a transaction, such as to activate a SIM, or to request additional information which is added to the conversation context.

A test facility will be provided.

Any data returned by the API is added to the Conversation Context, which can be accessed by a reply or custom code. (See Contexts section below.)

#### Action Code

Custom code can be supplied to perform any logic including interacting with one or more APIs. The signature of the required function is:

**function action(text, sticker, emoticon, user, conversation, callback)**

The ‘text’ argument is the raw response text. If the customer responded with a sticker or emoticon, the text translation is provided as the ‘text’ argument, and metadata for the next two arguments depending on type.

Known customer information is provided as the fourth argument.

Conversation-scoped data is provided as the fifth argument.

The sixth argument is a callback function, which must be called at the end. The first argument of the callback is an optional ‘error’ object to communicate an error in processing. The second argument is an optional ‘context’ object, which may contain structured data extracted from the response.

If ‘false’ is returned instead of a context object, the rule will be deemed to be a non-match, and therefore the response is handled by the fallback procedure.

If an ‘error’ object is returned, then the response will be handled by the fallback procedure as well as additional logging of the error.

An ‘error’ object has the following properties:

{

code: "<optional short code>",

description: "<optional technical description>",

message: "<error message>"

}

Implementation notes. Current thinking is to use the Nashorn engine in Java 8. As this doesn’t support NPM modules, nor has HTTP capabilities through XMLHttpRequest (a DOM API), we’ll provide out-of-the-box functions using Java, based on the ‘request’ API (https://github.com/request/request).

request('http://www.google.com', function (error, response, body) {

if (!error && response.statusCode == 200) {

console.log(body)

}

})

request.post({url:'http://service.com/upload', form: {key:'value'}}, function(err,httpResponse,body){ /\* ... \*/ })

// HTTP Authentication

request.get('http://some.server.com/', {

auth: {

user: 'username',

pass: 'password'

}

})

// or

request.get('http://some.server.com/', {

auth: {

bearer: 'bearerToken'

}

})

// Custom HTTP Headers

var options = {

url: 'https://api.github.com/repos/request/request',

headers: {

'User-Agent': 'request'

}

};

function callback(error, response, body) {

if (!error && response.statusCode == 200) {

var info = JSON.parse(body);

console.log(info.stargazers\_count + ' stars');

console.log(info.forks\_count + ' forks');

}

}

request(options, callback)

#### Custom Variables

Adds new variables to the Conversation Context, which can be used within custom code blocks and text replies, and can pass information to next steps in the conversation.

### Replies

A Reply defines the response back to the customer.

#### Text Reply

A text reply can be selected from the CMS, or defined as inline text. Inline text can include variables, including content items from the CMS and any data from the User or Conversation Contexts.

The text reply can be a template using the Handlebars format.

Embedded expressions within double braces `{{ }}` will be rendered. Object properties can be accessed using dot notation, e.g.

Hello {{context.user.firstName}}!

Lists can be rendered using:

{{#list context.conversation.reply.multichoice.items}}

{{label}}

{{/list}}

Conditional text can be handled using:

{{#if context.user.offer}}

{{context.user.name}}

{{/if}}

#### Prompts

Prompts can be used in combination with a Text Reply. It is the expected type of response to the current reply. For example, if we expect a time from the customer, then the Time prompt can be selected. This will validate the customer response for containing time information, and extract time mentions as a structured data object to be added to the Conversation Context.

The following Prompt types are supported:

* time, e.g. “today” or “Monday, Feb 18” - both points in time and intervals
* temperature, e.g. “65 degrees”
* number, e.g. “100K” or “four”
* ordinal, e.g. “first”
* distance, e.g. “7 km” or “2 inches” and conversion to a normalized unit (metre)
* volume, e.g. “250ml” or “1 gallon” and conversion to a normalized unit (litre)
* money, e.g. “ten dollars” or “$20.30”
* duration, e.g. “2 hours” or “3 min” and conversion to a normalized unit (seconds)
* email
* URL
* phone number, e.g. “415-123-3444” or “(650)-283-4757 ext 897”
* selection from a defined list, either by ordinal number of the item’s index in the list, or nearest textual match (using Levenshtein distance)

#### Multi-choice

#### Cards

#### Forms

With a bot interface, the framework should:

* Ask only for information not known, and remember details for a later session
* Confirm some items even if known
* Process unstructured input and natural language
* Enable a composite response to fill in multiple required data elements, e.g. ask for a complete address instead of separate requests for street, city, state, postcode, etc.
* Handle sudden departures from script and left-field questions from the user
* Enable custom functions or API calls to be used to validate or parse input
* Think of a form as a sub-conversational flow. Once an intent or goal requiring additional information is determined, the bot should be able to hand-off to a sub-routine or specialised bot that navigates the user through the information gathering process.

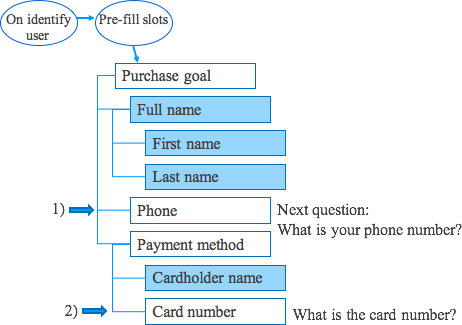
Form flows can be complex. For example, when asking for address details, if no address information is known, then the bot can ask for the full address. However, if city, postcode and state is already known, then the bot should ask for the street only. Defining any multi-exchange bot conversation is a challenge. Every intent that has information requirements to transact will pile on complexity unless there is built-in support for form flows generally.

This framework allows declarative definition of forms - you define what information you need, and how to validate and parse responses where required, the framework takes care of how to execute the flow.

### Core Concepts

A Slot is an item of information. Slots may be organised hierarchically. For example, address is made up of street, city, postcode, state and country. All child slots must be complete for the parent to be treated as complete.

Slots may be filled with known details once the user is identified.



Text forms

All slots must be filled to fulfil goals. Remember information for future tasks.

The Slot API is as follows:

* key - slot name
* question - optional if the slot has child slots, in which case questions will be asked at a lower level
* children - optional if the slot is at the lowest level, otherwise a list of child slots
* value - optional, may be pre-populated
* validateExpr - optional JavaScript function to validate user input
* validateFn - optional Scala code if the form is defined in Scala
* invalidMessage - optional response to invalid input if validateExpr or validateFn is present
* parseApi - optional URL to API used to validate the input
* parseExpr - optional JavaScript function to parse user input, or if parseApi is given, to convert the API response into the required format
* parseFn - optional Scala code if the form is defined in Scala
* confirm - optional response, which if present, is sent to the user to confirm the slot’s details
* caption - optional label for a slot, shown when the parent slot is being confirmed
* Functions/expressions have the following signatures:

validateExpr (JavaScript)

(value: String) => Boolean (true if the input is valid)

validateFn (Scala)

(value: String) => Boolean (true if the input is valid)

parseExpr (JavaScript) - parseApi not present

(value: String) => JSObject (of key-values where key names correspond to keys of child slots)

parseExpr (JavaScript) - parseApi present

(value: JSObject) => JSObject (of key-values where key names correspond to keys of child slots)

parseFn (Scala)

(value: String) => Map[String, Any] (of key-values where key names correspond to keys of child slots)

parseApi - URL String of endpoint containing ‘%s’ where the query string is to be interpolated

The API endpoint is expected to return a JSON response. A response of JSObject, where property names correspond to keys of child slots, can be used directly. Otherwise, parseExpr can be used to translate the format.

An example form definition is as follows:

purchase {

question = "Please provide your full name as <first-name> <last-name>"

name {

firstName {

question = "What is your first name?"

}

lastName {

question = "What is your last name?"

}

parseExpr = """

function (value) {

var re = /(\S+)\s+(.\*)/;

var match = re.exec(value);

return {

firstName: match[1],

lastName: match[2]

};

}

"""

}

phone {

question = "What is your phone number?"

confirm = "Is this number correct?"

}

address {

question = "Please provide your address as <street> <city> <state> <postcode>"

street1 {

question = "What is your street as <street-number> <street-name> <street-type>"

}

city {

question = "What is your city?"

}

state {

question = "What is your state?"

}

postcode {

question = "What is your postcode?"

}

country {

question = "What is your country?"

}

parseApi = $address.api.url,

parseExpr = """

function (value) {

return {

street1: value.street\_1,

city: value.city,

state: value.state,

postcode: value.postal\_code,

country: value.country

};

}

"""

}

}

# Contexts

Contexts are stores of information held within various scopes.

Scopes include:

* User Scope – persistent store of customer information available to all conversations
* Conversation Scope – data maintained with the scope of a conversation session
* Exchange Scope – variables passed in processing a single utterance

## User Scope

Data held in user scope is available anytime the user is identified, subject to specific confidentiality and privacy settings. For example, the customer may have opted out of receiving any marketing messages.

Transactions have information requirements, which the platform refers to as “slots”. Filled slots (answers to previous questions) is remembered as user data, so the same questions don’t have to asked in the future, except to confirm where appropriate.

(The structure of user data is expected to be a free-form JSON tree, although we should standardise sections such as profile, preferences, slots, and insights.)

User scoped data includes:

* User Profile
  + Name
  + Email
  + Avatar
  + Location
  + Age
* Preferences
  + Communication channel and app
  + Contact details
  + Payment
* Operational Data
  + Account information
  + Billing Address
  + Shipping Address
  + Payment
* Insights
  + Propensity scores
  + NPS
  + Journey predictions

## Conversation Scope

Conversation-scoped data is maintained against the conversation and includes for example:

* The current messaging platform used

# Example Scenarios

### Scenario A – Simple Text Reply

This is the simplest scenario. A text message is sent in response to a customer utterance matching an intent, entity, rule, or some combination of these.

For example,

**(1)** Utterance: “hello”

🡪 Intent: “Greeting”

🡪 **text\_reply**: “Hi there!”

## Scenario B – Text Response with validated reply (Prompt)

For example,

**(1)** Utterance: “I would like to top up my account”

🡪

Intent: “Pre-paid top-up”

Entity: “Account”

Instance: “Pre-paid”

🡪

text\_reply: “How much would you like to top-up?”

**prompt**: “money”

🡪 **(2)** Utterance: $30

🡪

valid: true

context: { amount: 30.00 }

# Appendix – API Specs

swagger: '2.0'

info:

version: "0.1.0"

title: AI Platform

schemes:

- http

paths:

/utterances:

post:

description: |

Add utterances mapped by intent

consumes:

- application/json

parameters:

-

name: body

in: body

description: utterances mapped by intent

required: true

schema:

title: ArrayOfUtterances

type: array

items:

$ref: "#/definitions/Utterance"

responses:

200:

description: Successful post

/entity-instances:

post:

description: |

Add entity instances

consumes:

- application/json

parameters:

-

name: body

in: body

description: entity instances

required: true

schema:

title: ArrayOfEntityInstances

type: array

items:

$ref: "#/definitions/EntityInstance"

responses:

200:

description: Successful post

/utterance:

get:

description: |

Get the extracted meaning of an utterance

parameters:

-

name: q

in: query

description: utterance text

required: true

type: string

-

name: context

in: query

description: |

optional context object as stringified JSON object

required: false

type: string

-

name: message\_id

in: query

description: optional message id

required: false

type: string

-

name: thread\_id

in: query

description: optional thread id

required: false

type: string

responses:

200:

description: Successful response

schema:

$ref: "#/definitions/Meaning"

post:

description: |

Get the extracted meaning of an utterance

parameters:

-

name: message

in: body

description: message post

required: true

schema:

$ref: "#/definitions/MessagePost"

responses:

200:

description: Successful post

schema:

$ref: "#/definitions/Meaning"

/converse:

post:

description: |

Advances the next step in the conversation

parameters:

-

name: body

in: body

description: conversation post

required: true

schema:

$ref: "#/definitions/ConversationPost"

responses:

200:

description: Successful post

schema:

$ref: "#/definitions/ConversationResponse"

/flows:

post:

description: |

Publish a flow that defines the conversational logic

for one or more intents

parameters:

-

name: body

in: body

description: flow definition

required: true

schema:

$ref: "#/definitions/Flow"

responses:

200:

description: Successful post

definitions:

Action:

type: object

properties:

api:

type: string

code:

type: string

vars:

type: array

items:

$ref: "#/definitions/Var"

Button:

type: object

properties:

button\_type:

type: string

value:

type: string

label:

type: string

required:

- button\_type

- value

- label

Card:

type: object

properties:

title:

type: string

subtitle:

type: string

image\_url:

type: string

buttons:

type: array

items:

$ref: "#/definitions/Button"

required:

- title

ConversationPost:

type: object

properties:

platform\_id:

type: string

sender\_id:

type: string

message:

$ref: "#/definitions/MessagePost"

required:

- platform\_id

- sender\_id

- message

ConversationResponse:

type: object

description: |

Expect one-of "text-reply", "quick\_reply",

or "card"

properties:

reply\_type:

type: string

meaning:

$ref: "#/definitions/Meaning"

confidence:

type: number

text\_reply:

type: string

quick\_reply:

type: object

card:

type: object

EntityInstance:

type: object

properties:

entity:

type: string

instance:

type: string

synonyms:

type: array

items:

type: string

required:

- entity

- instance

EntityResponse:

type: object

properties:

entity:

type: string

value:

type: string

confidence:

type: number

start\_text\_pos:

type: number

end\_text\_pos:

type: number

metadata:

type: string

description: |

optional additional context as stringified JSON object

EntryCondition:

type: object

description: |

Expect one-of "intent", "entity", or "text\_match",

accompanied by "and", "or", or nothing

properties:

intent:

type: string

entity:

type: object

description: |

Expect either "entity\_type" or both "entity\_type"

and "instance"

properties:

entity\_type:

type: string

instance:

type: string

text\_match:

type: string

and:

$ref: "#/definitions/EntryCondition"

or:

$ref: "#/definitions/EntryCondition"

Flow:

type: object

properties:

id:

type: string

name:

type: string

nodes:

type: array

items:

$ref: "#/definitions/FlowNode"

links:

type: array

items:

$ref: "#/definitions/FlowLink"

required:

- id

- name

- nodes

FlowLink:

type: object

properties:

id:

type: string

from:

type: string

to:

type: string

required:

- id

- from

- to

FlowNode:

type: object

properties:

id:

type: string

name:

type: string

description: |

Descriptive title. Not used by AI engine.

A boolean prompt is the same as a "quick reply".

entry\_condition:

$ref: "#/definitions/EntryCondition"

action:

$ref: "#/definitions/Action"

reply:

$ref: "#/definitions/Reply"

required:

- id

- name

- entry\_condition

IntentResponse:

type: object

properties:

intent:

type: string

confidence:

type: number

metadata:

type: string

description: |

optional additional context as stringified JSON object

required:

- intent

Meaning:

type: object

properties:

message\_id:

type: string

utterance:

type: string

intents:

type: array

items:

$ref: "#/definitions/IntentResponse"

entities:

type: array

items:

$ref: "#/definitions/EntityResponse"

required:

- utterance

MessagePost:

type: object

properties:

utterance:

type: string

message\_id:

type: string

thread\_id:

type: string

context:

type: object

required:

- utterance

Reply:

type: object

description: |

Expect one-of "prompt", "multichoice", "form",

"cards", or "agent\_message". Also expect one-of

"text\_reply" or "agent\_message".

properties:

text\_reply:

type: string

prompt:

type: string

enum:

- boolean

- distance

- duration

- email

- money

- number

- ordinal

- phone-number

- selection

- temperature

- text

- time

- url

- volume

multichoice:

type: array

items:

$ref: "#/definitions/Button"

form:

type: string

cards:

type: array

items:

$ref: "#/definitions/Card"

agent\_message:

type: string

Utterance:

type: object

properties:

utterance:

type: string

intent:

type: string

required:

- utterance

- intent

Var:

type: object

properties:

key:

type: string

value:

type: string

required:

- key

- value

# Appendix – Intent CSV Exchange Format

Utterance, Intent

|  |  |
| --- | --- |
| Is it chilly? | temperature |
| What's the current temp in Celsius? | temperature |
| What is the temperature in Fahrenheit? | temperature |
| Is it windy? | conditions |
| Will it rain today? | conditions |

# Appendix – Entity CSV Exchange Format

Type, Instance, Synonym 1, Synonym 2, …

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| weekday | Monday | Mon |  |  |
| weekday | Tuesday | Tue | Tues |  |
| weekday | Wednesday | Wed |  |  |
| weekday | Thursday | Thur | Thu | Thurs |