Chatbots in eGovernance

Exploring Possibilities and Challenges

Sebastian Hansmann, Marcel Müller

October 5, 2017

Contents

List of Figures

1

Introduction

In the past two years "many private-sector businesses have started building chatbots into their products to improve customer service" [Leo16] based on a small variety of underlaying chatbot systems. As a tool for improving customer services "in the application of the tools and techniques of e-Commerce to the work of government" [How01] called *eGovernment*, chatbots have a huge potential in automating repetitive administration tasks to help governments around the globe to fulfill their main challenge of reaching to masses and reducing bureaucracy.

The objective of this work is to apply the use case of a chatbot for serving as a general purpose information chatbot in the context of the public administration of the City of Berlin (`berlin.de`) and the evaluation of the suitability of already existing chatbot building frameworks such as API.AI (see `www.api.ai`) and AWS Lex (see `aws.amazon.com/lex/`).

2

Frameworks

In this chapter we are comparing two chatbot frameworks to evaluate, which one we might use for the implementation of our own chatbot.

2.1 Amazon Lex

Amazon Lex is as a product part of AWS (see `aws.amazon.com`) and a a service for building conversational interfaces which can be deployed to any application invoking text or voice interaction. Lex provides Machine Learning and Deep Learning capabilities, adverted as being the same as used for Amazon Alexa. advertised

Programming Model

Lex works as figure 2.1 shows with the following concepts:

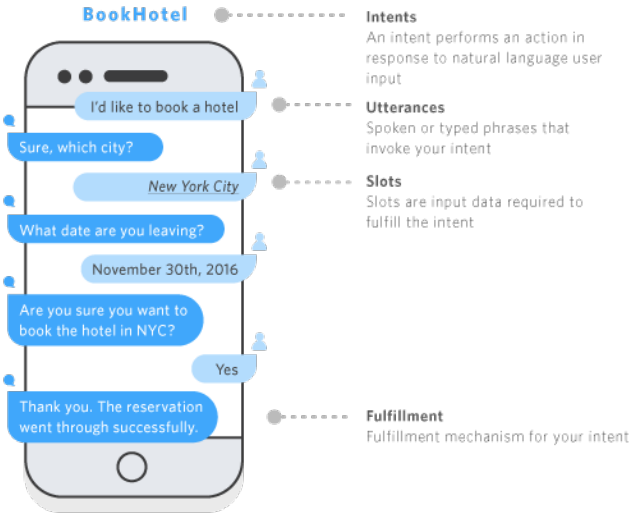

Figure 2.1: Schematic illustration as seen on the official AWS webpage for Lex `aws.amazon.com/lex/details/`

- **Intents**: Intents represent actions users might want to perform. Chatbots can have multiple intents. An example intent (for a bot build for making reservations for hotels) would be: book a room.
- **Utterances**: Are natural language sentences standing for how the user might phrase the intent. For the aforementioned intent this would be: "I would like to book a room".
- **Slots**: Slots can be seen as the parameters for an intent, which are automatically inferred by Lex. In Figure 2.1 an example slot would be the question for the city the user wants to book a hotel room in. Lex automatically inferes the users answers (New York City) with the city.
- **Fulfillments**: Fulfillments are code hooks which are triggered to perform an action invoking other services. In our example this would be the interface to the real room booking system of the hotel.

Maintenance

The chatbot can be maintained through a online console within the AWS platform. Adding new functionalities to the bot can also be performed through a RESTful interface.

Language Support

The only supported language at the start of the project is English.

Pricing

Lex offers a pay-as-you go model measured with the amount with processed requests. The first 10,000 text requests and 5,000 speech requests per month are free in the first year of usage. After this free tier is exceeded the pricing scales at $0.004 per voice request, and $0.00075 per text request.

Availability, Distribution and Scalability

AWS divides its computation resources into availability zones (see figure 2.2). At the start of this project Lex was only available on the North-Virginia Cluster[1].

[1] as seen in this table: https://aws.amazon.com/de/about-aws/global-infrastructure/regional-product-services/

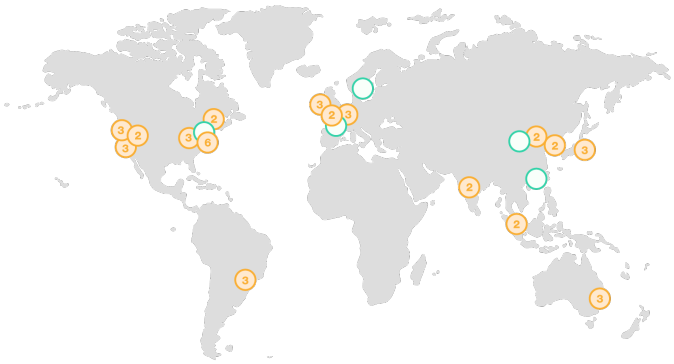

Figure 2.2: AWS availability zones as seen on aws.amazon.com/de/about-aws/global-infrastructure/. Orange circles are locations of actual, green circles are locations of planned data center locations.

2.2 API.AI

API.AI was initially a startup which got acquired by Google in 2016 [2]. It supports conversational interactions via chat and has a huge integration into existing chatbot environments.

Programming Model

API.AI has the following concepts:

- **Intents:** Similar to Lex API.AI also supports intents to model the real world intent of a user, so an abstract mapping between what a user says and what action should be taken by the chatbot. Every intent can have the following:
  - *User Says:* An intent can have many user says expressions, each on of them can be either in *example* or in *template* mode. In example mode, an user says expression is written in natural language and annotated in a way, that the entities extracted by the framework, then later can serve as input parameters for programming logic. Also there is an automatic annotation, where the framework infers entities to words in the expression by mapping them to examples and later to be enhanced with machine learning.

[2] See that interesting article on TechCrunch: techcrunch.com/2016/09/19/google-acquires-api-ai-a-company-helping-developers-build-bots-that-arent-awful-to-talk-to/

  - *Action:* An action is the next step after the mapping and extraction of user inputs in a specific intent to a action. The before mentioned parameters can serve as inputs to actions. These actions are the interface to any programming logic, sending requests to a RESTful backend.
  - *Response:* Direct Responses are also possible to use instead of actions being triggered by a specific intent. In contrast to actions no backend requests will be sent, answers will be saved there hard coded instead.
  - *Contexts:* Contexts are a special kind of parameter designed to pass information from previous conversation parts or external sources. They are used to build the conversation flow.

- **Entities:** Entities are the programming concepts empowering the framework to extract parameter values from user says expressions. There are three different kinds of entities:

  - *System Entities:* These entities are pre-built provided by the API.AI framework for the most common concepts such as date and time, numbers, amounts with units, unit names, geography, contacts, names, music, color, language and generic concepts as `@sys.any`.
  - *Developer Entities:* Also developer can define custom entities, which are usually reflecting the core of the chatbots application domain.
  - *User Entities:* User entities can be redefined on special information on a session ID level, to act as a cache for encapsulation user related information.

  All entities are also either an mapping, an enum, so an entity, consisting of other entities or a composite entity.

More basic concepts can be found at `api.ai/docs/getting-started/basics`.

Maintenance

API.AI chatbots can be maintained through an online console. Adding new functions to the bot can also be performed through a RESTful interface or via json-objects uploaded using the api.

Language Support

14 different languages are supported including English, Chinese, Spanish, Dutch and German.

Pricing

The usage of API.AI is free [3].

2.3 Evaluation and Decision

Weighting pros an cons for both models the decision is relatively easy, as long as we reason based on this very high level overview. Since we want to write our user interactions in German language and all our data is German, API.AI is the only possibility, since Amazon Lex only supports English. Also when it comes to pricing, the always free model of API.AI is unbeatable. Although we don't know where API.AI is physically hosted, Lex position in the U.S. might lead to high latencies resulting in a bad user experience or to data privacy issues. We proceed any further studies with our chosen framework API.AI.

[3] As to see in https://api.ai/pricing/

3

Implementation

Once we decided to take our explorations further with API.AI, we first need to get an overview over the information provided in order to get an idea of how mapping between data and the programming concepts should be done.

3.1 Data

For the content of our chatbot, which we named *Berlina*, we were provided a collection of different json and csv documents. As a first step we did some data exploration to see where we could get with the data.

Exploration

In this section we are getting a first glance at the data provided, which we are using to build our bot.

dienstleistungen.json: The `dienstleistungen.json` provides us with the most important data as a json list of service items. Each of the items represents one service and contains up to 22 properties:

- *authorities:* This field contains information about the authorities offering the specific service and has fields for *id*, *name* and *webinfo* which is the link to the described authority.
- *locations:* Stands for the location where the required service can be fulfilled and has also fields itself for *location* as an ID, *url*, *hint* and *appointment*.
- *meta:* Contains fields for the *lastupdate*, *url*, *locale* and some *keywords* for the service.
- *process time:* Is a natural language description of the estimated process time.
- *requirements:* Symbolizes requirements for the fulfillment of the specific service and contains a *name*, *link* and a *description*.

- *forms:* States the needed forms for the fulfillment of the service described by a *name*, *description* and *link*.
- *fees:* Describes the fees for the service in natural language.
- *prerequisites:* Contains fields for *name*, *description* and *link* to describe what needs to be done upfront.
- *id:* An unique ID to identify the service.
- *description:* A natural language description of the service, formatted with HTML tags.
- *LeiKa:* The **LeiKa** service code used to categorize the services.
- *links:* A list of *weblinks* for the service.
- *responsibility all*: This is a boolean value, with to us unknown semantics.
- *name:* The correct natural language name of the service.
- *onlineprocessing:* Contains fields for *description* and *link* if an online processing of the service is available. If not, the description says *false*.
- *legal:* Giving the legal backgrounds with fields for *description*, *name* and *link*.
- *representation:* A numeric value.
- *residence:* A numeric value.
- *relation:* Contains the field *root topic* which is a numeric value.
- *appointment:* Contains a link to the online appointment manager with the respective service pre configured.

It is worth noting that the JSON document is very unclean. The field fee for instance, contains natural language as well as numeric descriptions of the costs for using that service. Some fields even contain inline html tags, suggesting it was designed with an other use case in mind. From a reverse engineering perspective we suspect, that the JSON is used to render information presented on www.berlin.de.

d115toLeiKa.csv: This csv gives a mapping between services according to D115 and the LeiKa (Leistungskatalog). LeiKa helps to categorize and structuring services. The csv has the following columns:

- *D115 ID:* The ID according to D115.
- *LeiKa Nummber:* The LeiKa number.
- *LeiKa Gruppe:* A top level grouping of the services.
- *LeiKa Kennung:* A natural language description of the service.
- *LeiKa Verrichtung:* A single noun describing the objective of the service.
- *LeiKa Synonyme:* A list of synonyms for the service.

These two documents where sufficient for us to build *Berlina*.

LeiKa

A main key to get a categorization of different services was the LeiKa number. LeiKa (Leistungskatalog der öffentlichen Verwaltung) is an exhaustive catalog of administrative services among all levels and authorities in Germany. LeiKa categorizes and encodes every service into the following, as seen in figure 3.1.

- *Instanz/instance:* Identifies who created the described LeiKa object.
- *Leistungsobject/service object:* Describes the legal service object according to existing laws.
  - *Leistungsgruppierung/service group:* Grouping of services into bigger categories.
  - *Leistungskennung/service identifier:* Uniquely identifies the service.
- *Verrichtung/fulfillment:* Describes what administrative action belong to the specific service in the specific LeiKa object.
  - *Verrichtungskennung/fulfillment identifier:* Identifies the action to resolve a fulfillment.
  - *Verrichtungsdetail/fulfillment detail:* Adds more information to the fulfillment in detail concerning the whole workflow.

Since the LeiKa names and codes are too many, confusing and not very intuitive, we came up with a simplified solution.

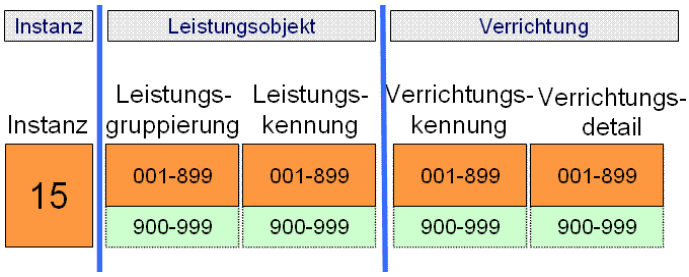

Figure 3.1: LeiKa 2.0 Schlüssel from `de.wikipedia.org/wiki/LeiKa`

Data Structures

We structured the data based on the LeiKa numbers in a more advanced way and merged them with related data for service descriptions in the `dienstleistungen.json`, to utilize a tree as the data structure of our choice as seen in figure 3.2.

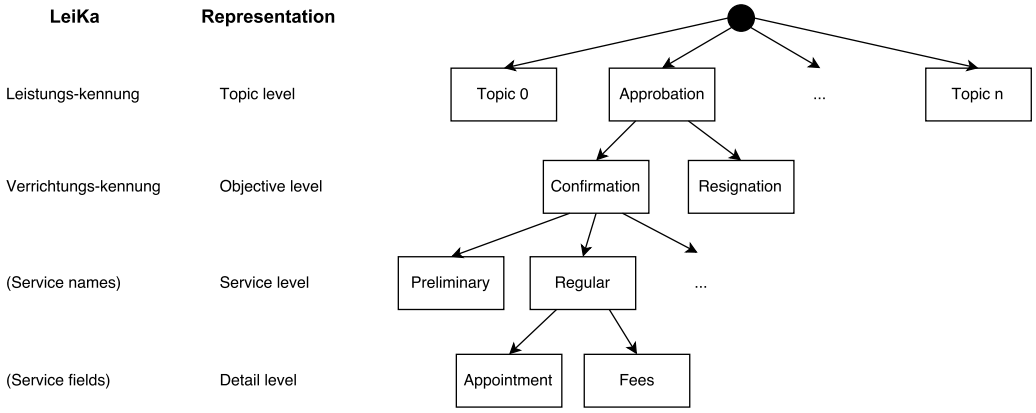

Figure 3.2: Topic-Objective-Service-Detail Tree (TOSD tree)

The tree has the following four levels:

- *Topic level:* Is the equivalent of a natural language term of the LeiKa service object. In the shown example of getting an approbation as a doctor the topic describes the approbation itself.
- *Objective level:* Is equivalent to a fulfillment encoded in the LeiKa number. An objective tells us what we want to do with a topic. In the example we need a confirmation or a resignation for an approbation.
- *Service level:* Is equivalent to the service title in `dienstleistungen.json`.

- *Detail level:* Adds more natural language details what a user might want to know about the specified topic and objective. In our example it might be the location, where a service can be fulfilled or information on how to get an appointment. Its origin are the service fields in the `dienstleistungen.json`.

This tree structure helps us to handle the huge amount of data the jsons provides us with and also lays the base for dialogues.

3.2 Dialogues

After imposing our improved data structure, we continue reasoning about how to utilize it within the API.AI dialogues and their corresponding intents.

Small Talk Dialogues

To build a foundation for Berlina, we trained her with common Small Talk capabilities. API.AI provides a series of small talk intent, pre-configured where developers just have to fill in their answers. Overall we imported 80 small talk intents and adjusted them to Berlina's needs. The following example shows the `smalltalk.agent.marryuser` intent:

- **User says:**
  - "willst du meine Frau werden",
  - "wollen wir uns das Jawort geben",
  - "lass uns heiraten",
  - "wollen wir heiraten",
  - "werde meine Frau",
  - "du bist meine große Liebe, ich will dich heiraten",
  - "bitte heirate mich",
  - "willst du meine Frau sein",
  - "willst du mich heiraten",
  - "heirate mich"
- **Events:** -
- **Actions:** -

- **Response:** “Mein Herz gehört meiner Arbeit”

The User Says expressions are in this case in pure example mode, not a single template (to be identified with an @ sign) is used. This works because in API.AI machine learning is enabled per default for the important intents along with training data in the background, without even letting the developer explicitly know.

Information Desk Dialogues

The Information Desk Dialogues to get the answers for the specific administrative questions is the heart of Berlina. Analyzing the structure of dialoges, one objective of the system should be to minimize the number of questions required for getting the right answer. Considering the Topic-Objective-Service-Detail Tree (see figure 3.2) from a technical view, we need to determine all levels to ensure that a correct answer can be given. Splitting the conversation into sentences, a User says expression can contain:

- Only entities from one level. For the topic level entity for example "I need to know something about an Approbation as a doctor or for the objective level "I need a Confirmation" and so on.
- Entities from two levels, for example "I need information on a confirmation of my approbation as a doctor".
- Entities from three levels, for example "I need information about a regular confirmation of my approbation as a doctor".
- Entities from all levels like, "At what locations can I get a regular confirmation of my approbation as a doctor?"

Also all combinations are theoretically possible, even though not all are meaningful. No one would ask "I need a location", without specifying the other two levels.

3.3 Implementation of Intents, Entities and Contexts

Restructuring the data into smaller pieces being more easy to handle we are able to build the core objects of API.AI: intents, entities and contexts.

Dialog mappings to intents

Resulting from our structure we have different possibilities to implement our intents

Single Intents In this approach every value of every level along with its synonyms will be an entity and for every topic, every objective and every detail we need to have one intent. Then for every combination of the three levels one single intent is needed. For our example we would then need the following intents:

- approbation intent
- approbation.confirmation, approbation.resignation intents ...
- approbation.confirmation.regular, approbation.confirmation.preliminary intents ...
- approbation.confirmation.regular.location, approbation.confirmation. regular.appointment, approbation.resignation.regular.location, approbation. resignation.regular.appointment ... intents

We see that the amount of intents are exponentially increasing while stepping into deeper levels of the Topic-Object-Service-Detail Tree. This structure gives us the possibility to hard code every answer to the intents into the API.AI response field. The architecture resulting from that approach would have nothing more than API.AI as our interface to the user, entity, intent mapping and further the storage of data.

Generic Intents In contrast to the Single Intent approach Generic intents limit the amount of intents very strictly and do not increase with every new service being added to the knowledge base. We would have only the following intents:

- topic intent
- objective intent
- detail intent
- topic.objective intent
- topic.objective.service intent

- topic.objective.service.detail intent

We are skipping the topic.detail intent, since we assume that this would be a rare use case.

With the Generic Intents approach we would also have to structure the entities differently. We still have entities for all possible values of all three levels, but in addition we also have enum entities. For example the enum entity `topic` consists of all topic intents (e.g. `topic` = `Approbation`, `Passport`, `Lost driver license` ...). Since the templates of the intents then would have to contain these enum entities, we cannot hard code the answers into API.AI's response field anymore. We need to extract the values from the enum intents as parameters and pass them into a RESTful backend and program the mapping to the possible answers ourselves, before we pass the response back to API.AI, which is forwarding it to the user.

Weighting the strength and weaknesses of the approaches, we decided to implement the system using generic intents, since they make the whole architecture easier to maintain. New services and answers could be inserted very easily even without touching API.AI and the data remains in a database of our choice. Also the amount of intents we need for our large knowledge base would exceed the limitation of API.AI to only 250 intents per bot. Since we decided to get our User Interface hosted through the Facebook Messenger (`messenger.com`) and a Python Flask backend, our logic sequence looks like that.

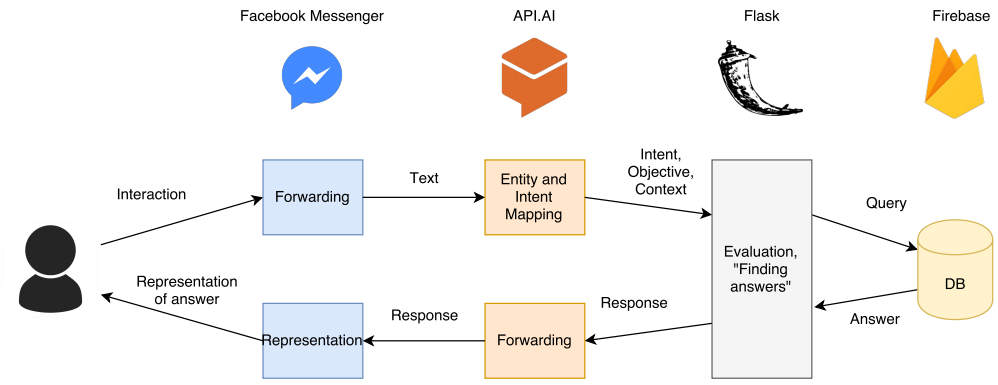

Figure 3.3: Sequence Diagram

Entities

Since we decided to use generic intents, we have to group our entities so that every API.ai entity represents are real entity of either the topic, the objective

or the detail level. We want to make the bot capable of recognizing as many synonyms as possbile so we are querying the Open Thesaurus API[1] with the name of every entity and paste all given synonyms from Open Thesaurus into API.ai. If the name of the objective is a composition of different words, we build the cross product of them. For example:

- **Intent: "Augenärtzliche Beratung"**
- **Synonyms: "Augenärtzliche Beratung"**
  - Augenärztliche Konsultation,
  - Augenärztliche Besprechung,
  - Augenärztliche Consulting,
  - Augenärztliche Mentoring,
  - Augenärztliche Gespräch,
  - Augenärztliche Supervision,
  - Augenärztliche Unterstützung

Augenärztliche Beratung is one element inside the topic entity

Contexts

Contexts help us to identify at which position in the TOSD-tree the user currently is. The transitions from one context to another context can be seen in Figure 3.4.

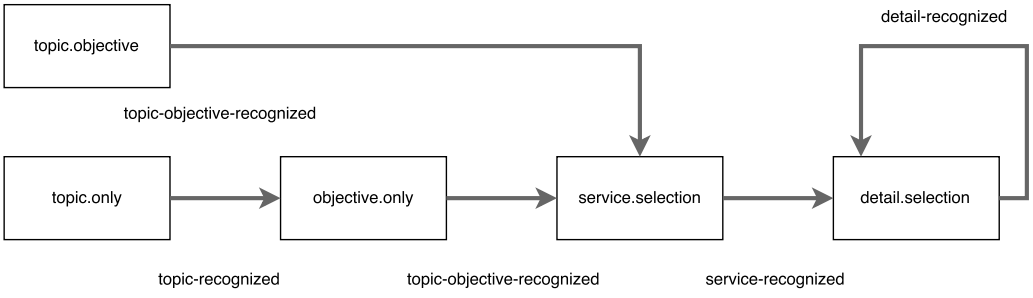

Figure 3.4: All possible transitions between contexts

The context state diagram also shows precisely our dialogue flow. A user can start by saying a topic or a topic and an objective. If the user says the

[1] see `https://www.openthesaurus.de`

topic only, he has to specify the objective in the second step. And in the third step he then has to specify which service he wants to address. Last the user can get as many details to the service he wants to get fulfilled as possible, for instance locations, fees or due dates.

Initial Drawbacks

Our first implementation with generic intents and entities with all possible combinations of synonyms gave us very poor results. The amount of synonyms to instances of entities is limited to 100 each, a limit which got exceeded for intents with very long composite names like “Anerkennung von Stellen für Fahrtschreiber- und Kontrollgeräteprüfungen”. The biggest part of the synonyms does not make any semantically sense. Also the bot had trouble to map many other synonyms to entities since there are collisions between synonyms of different entities. Figure 3.6 shows that after the user has navigated to topic passport (“Personalausweis”) API.ai has the possible objectives "Ausstellung”, “Informationsmitteilung” and “Meldung”, but it mapped to the objective “Mitteilung” which belongs is a synonym for "Meldung” but also an objective itself. These *non-existing* mappings are leading users to dead ends and need to be avoided. In some other cases the synonyms were also mapped to objectives which are not even on the current level of the tree.

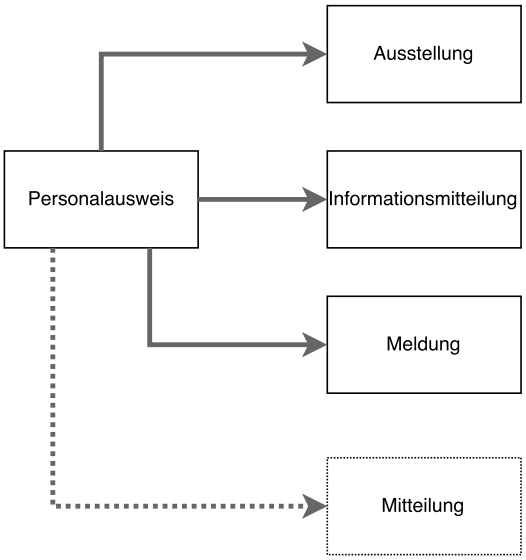

Figure 3.5: The synonyms were initially not collision free.

After a few experiments we recognize that we have no influence to which

objectives recognized words and phrases get mapped. This caused major problems since we are dealing with a very knowledge base with lots of entities which may overlap. Exploring other examples implemented in API.ai like chabots for ordering pizza, flowers or making reservations have shown us that they domains of one single bot in API.ai is usually very small and restricted to a very specific domain with just a few entities.

Improved Implementation

The problems coming from our huge knowledge base and the domain made lead us to another approach. Considering that there is clearly a trade off between the amount of synonyms and the answer quality, due to collisions coming with bigger amounts of synonyms we had to make our synonyms "collision free". Therefore we replaced all synonyms from Open Thesaurus with the synonyms provided to us with the `d115toLeika.csv` with leaving synonyms out, if they are appearing twice. These led us to a smaller synonyms to entity instance ration but with better quality for the detected entities.

To problem with objectives from different levels of the TOSD-tree has been addressed with restricting the possible answers after the recognition of the topic intent by adding quick replies, the user can click instead of typing the answers. This makes sure the user can only select an objective which belongs to the topic he has previously chosen. The same was applied for the service selection as well.

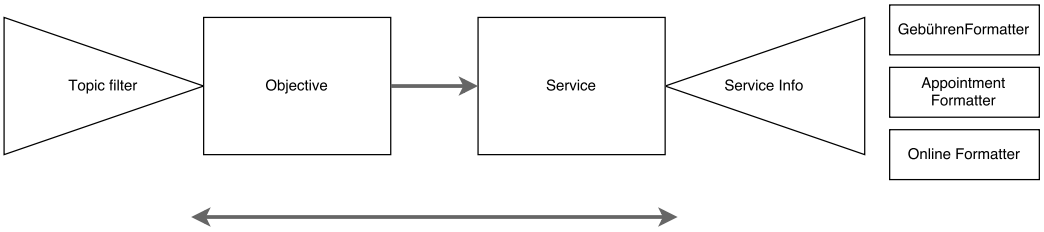

Figure 3.6: Restrictions to the possibilities the user can choose at certain point implemented with quick relies in the Facebook Messenger

With applying these changes we traded better quality for the recognized entities for the amount of entities being recognized (traded precision for recall).

After applying all enhancements to the intents, entities and contexts one example intent with all entities and contexts can be seen here in Figure 3.7.

3.4 Backend

We implemented the backend to evaluate user queries in the *Python* programming language. The *Flask* package was used to create a RESTful service which tries to respond accordingly to given input.

Overview

The backend handles enriched queries send from API.AI and creates responses based on the recognized intent as well as on the displaying clients requirements. A graphical overview of our implemented backend can be seen in figure 3.8.

Incoming messages get first parsed into a message object, containing attributes for topic, objective as well as detail of interest. Based on the intent send from API.AI, we access the emitted JSON and parse those parameters accordingly. Observed and present variables determine the state of the whole message. For instance, is a topic as well as an objective present, we should emit details to choose from to the user. Is only the topic present, we should display existing objectives in that branch of our data tree. To further enhance the communication flow, we decided to implement shortcuts to increase the speed of the underlying tree traversal. If there is only one objective present for the recognized intent, we skip the objective display and continue directly with showing relevant services. After creating a message object as well as determining it's state we continue with handling the message accordingly. Depending on the state of the message and using the previously extracted topics, objectives or details, we query our database for the relevant data. If we request information from the detail level, we still need to parse it by removing HTML tags present in the provided *dienstleistung.json*. Since every field in the detail level is encoded differently, we built a customized parser for most of these attributes.

With the prepared data at hand we are now ready to issue responses back to API.AI and ultimately to facebook's messenger. To improve the user experience, we make use of so called "structured messages" that the chosen Facebook chat client is able to display. We continue with a detailed description of this structured content and its usage for Berlina in the upcoming section.

Structured content for Berlina

To help the user navigate through our data model we chose to implement structured content. The kind of response we want to issue depends on the

state of the message we received. To help the user choose either objective or detail values we opted for quick replies, that display possible choices as predefined buttons (figure 3.9).

This feature does not only improve the user experience, it further helps us to improve the objective mapping by suggesting users possible choices instead of typing in a customized text which would be harder to handle, especially since the user is not aware of possible objectives for that very topic. For displaying the service selection, we decided to implement the general template, which can be used to create either lists or cards, containing the fields *template_type* and *elements*.

The field *template_type* controls which structured message will be issued. The *element* collection contains every card or list item that should be displayed to the user. Each of these elements contains an image link, a title, a subtitle as well as an optional button. Facebook imposes certain limitations to each and every one of these structured messages which had to be respected and specially handled. Lists for example are limited to show 2 up to 4 elements, we therefore had to dynamically decide which kind of response to issue. Fortunately this general structure helps to adapt the response during run time. If there is only one service for a certain topic and objective present, we issue a card representation as can be seen in figure 3.10. The list template on the other hand gets send in every other case, splitting up possible services into multiple lists (figure 3.11).

Dashboard

Besides request handling and response generation we implemented a dashboard to monitor the bot's usage. Especially since we opted for the previously discussed general intents, we are not able to identify and count which topic or detail intent was instantiated using API.AI. We therefore collect queried topics as well as details and save them to a relational database inside our AWS cluster, which was used to host our backend. After logging into the dashboard, these topic and detail counters get queried and displayed on our frontend (see figures 3.12 and 3.13).

The dashboard aims to get an overview over which topics and details are queried most often and should therefore help improving the chatbot over time. One could think of removing unused detail information fields as well as displaying popular topics more present. To further improve the intent recognition for topics of high interest one could also remove or add new synonyms accordingly.

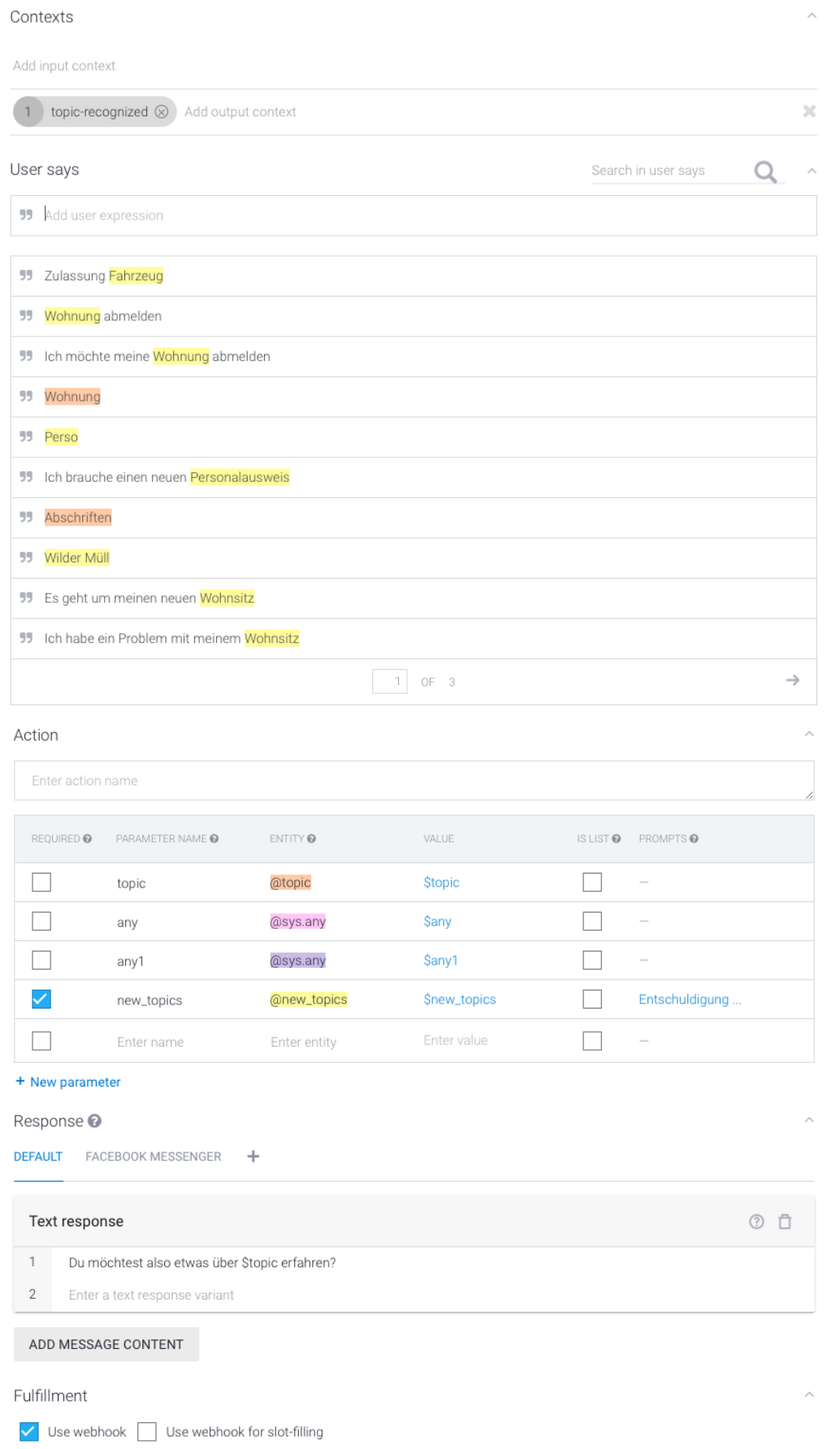

Figure 3.7: Example intent with context and entities

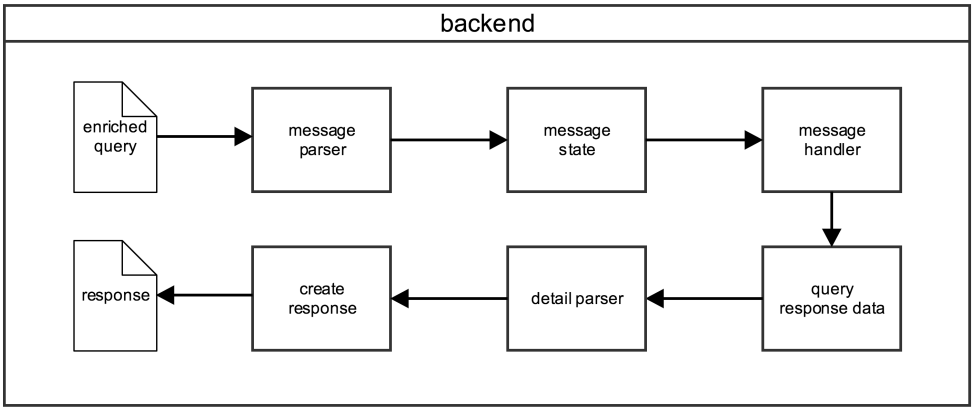

Figure 3.8: Backend overview

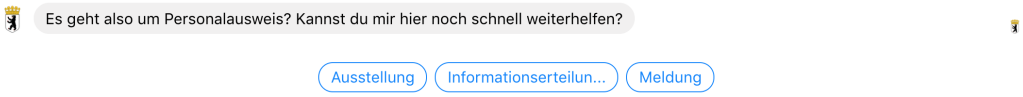

Figure 3.9: Quick replies

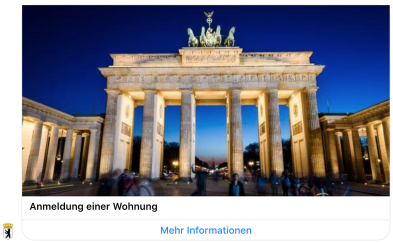

Figure 3.10: Card representation

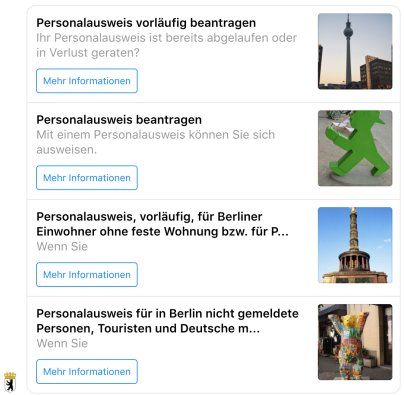

Figure 3.11: List representation

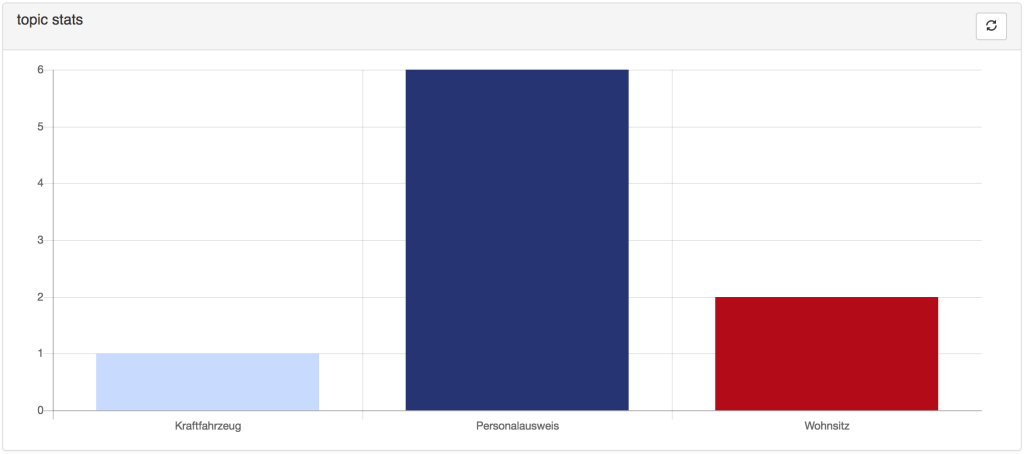

Figure 3.12: Topic stats

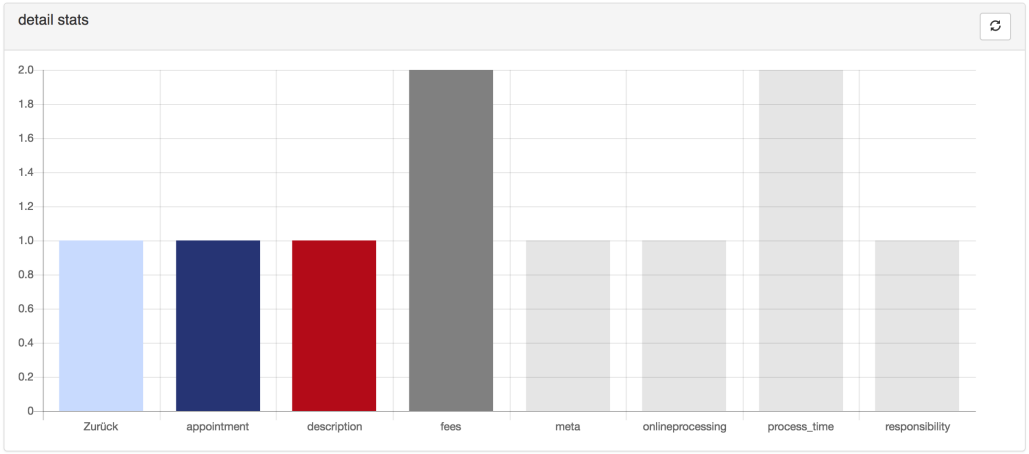

Figure 3.13: Detail stats

4

Conclusion

During the project implemented a huge real world chatbot with a third party framework. We decided to go with API.AI instead of AWS Lex for reasons of language support. Handling a huge base of information, we faced many difficulties while going from a 280k line JSON file with all necessary information being encoded to a fully working Facebook Messenger chatbot consisting of entities, contexts and intents.

Data Cleansing was a very challenging task. The data source had a certain schema, but in the fields were numerous difficulties to overcome, such as in line HTML tags as well as an inconsistent encoding of information. We cleaned the data set as much as possible and restructured into our own Topic-Objective-Intent Tree, a hierarchical data structure. Saved in a Firebase Database the restructured JSON data now served as our main database.

The tree structure was used for imagening possible chat dialogues so we could start building the chatbot with the needed intents, contexts and entities. At this point we had to make a design decision whether to build many intents with single entities or if we would build generic intents. For better maintainability we decided to go with the generic intent solution.

Transmitting intents, objectives and entities to API.AI is possible via the provided web interface and via an REST Api. Data parsing and uploading via HTTP requests was a special challenge. There is a security mechanism, preventing you from sending more than 10 requests per minute, which was no where explicitly documented. Also the limitations of API in the amount of intents, entities and synonyms per entities could only be found on some online forums.

For entity synonyms we initially invoked the Open Thesaurus API, which provided us with lots of synonyms. For composite entities we concatenated every possible combinations of synonyms, which led us to too many and very bad quality entries. Even more we had problems with entities conflicting one other. If API.AI decides to map a certain recognized word to an entity and there are several possibilities, we have no chance of influencing that decision. This decreased the quality of answers we got dramatically.

We solved the problem with colliding synonyms for different intents by grouping the provided keywords and making them pairwise different. We further narrowed the possibilities to answer to questions asked by the bot

down to predefined answers, showed as quick reply in the Facebook Messenger to prevent *non-existing* branches.

In the end we can conclude that it is a very challenging task to handle a huge amount of information and long dialogues in API.AI. Also looking to given use cases we recognized that the framework is primary built for smaller and more specific domains, such as ordering a specific product or reservations.

Bibliography

[How01] Mark Howard. “E-government across the globe: how e will change government”. In: *e-Government* 90 (2001), p. 80.

[Leo16] Matt Leonard. *Top 100 Services in local german administration*. `https://gcn.com/articles/2016/10/13/nc-chatbots.aspx`. [Online; accessed 5-June-2017]. 2016.