Universidade Federal do Rio de Janeiro
Escola Politécnica

PROCESS-AWARE CONVERSATIONAL AGENTS

Luis Fernando Lins dos Santos

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Prof. Dr. Toacy Cavalcante de Oliveira
Co-orientadora: Gláucia Melo dos Santos

Rio de Janeiro
Março de 2021

PROCESS-AWARE CONVERSATIONAL AGENTS

Luis Fernando Lins dos Santos

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO

Autor:

Luis Fernando Lins

Luis Fernando Lins dos Santos

Orientador:

[illegible]

Prof. Dr. Toacy Cavalcante de Oliveira

Examinador:

[illegible]

Prof. Dr. Fernando Gil Vianna Resende Junior

Examinador:

[illegible]

Dr. Ulisses Telemaco Neto

Rio de Janeiro

Março de 2021

Declaração de Autoria e de Direitos

Eu, *Luis Fernando Lins dos Santos* CPF *151.182.607-01*, autor da monografia *Process-Aware Conversational Agents*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.

2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.

3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.

4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.

5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.

6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.

7. Por ser verdade, firmo a presente declaração.

Luis Fernando Lins

Luis Fernando Lins dos Santos

AGRADECIMENTO

Agradeço primeiramente à minha mãe Maria Lúcia, heroína que sempre vê o lado positivo da vida e me ajuda a passar por todas as dificuldades com suas sábias palavras. À minha avó Lúcia, carinhosa e atenciosa, e que esteve do meu lado a todo momento, ajudando na minha educação e crescimento pessoal.

Às minhas tias Simone e Silvia, que sempre torceram por mim e acreditaram no meu potencial, e meus avós Margarida e Ozanir, que me ensinaram desde pequeno a importância do estudo para o meu desenvolvimento.

Ao meu orientador Prof. Dr. Toacy, que não só vislumbrou o potencial deste trabalho e me ajudou quando fiquei perdido, mas também me acolheu tão bem quando fui visitá-lo no Canadá para realizar parte da minha pesquisa.

À Gláucia, co-orientadora e amiga, por sua enorme paciência e disposição em ajudar, e também pelos momentos de descontração quando tudo parecia caótico demais.

Agradeço a todos os professores da UFRJ que contribuíram para a minha formação, e também aos professores Paulo e Don, da Universidade de Waterloo no Canadá, pelo incentivo e suporte no tempo limitado que passei lá.

Aos meus amigos, agradeço pelo carinho, pelas risadas e pela força que me deram durante todo o curso.

Por fim, agradeço ao universo, que, em sua linda e incompreensível loucura, me permitiu a oportunidade e privilégio de estudar nesta universidade, ter experiências incríveis e conhecer pessoas especiais que levarei comigo para o resto da vida.

RESUMO

Processos são essenciais na sociedade atual, já que os mesmos promovem a padronização, documentação e controle das interações entre empresas, governos, indivíduos e outras organizações. Processos também são usados em fluxos de trabalho humano, em que os participantes atuam sozinhos ou em grupo para cumprir os seus objetivos. Presentes na vida e na rotina da maioria das pessoas, processos podem ser tão simples como seguir uma receita culinária ou tão complexos como prover tratamento para uma doença grave. Portanto, é essencial fornecer suporte aos participantes do processo durante sua execução. No entanto, a maioria das soluções atuais para gerenciamento de processos, além de serem proprietárias, carecem de clareza ao tentar ajudar os usuários na execução de processos diários. Neste trabalho, apresentamos um guia para a construção de um agente conversacional - também conhecido como *chatbot* ou simplesmente *bot* - que utiliza o padrão Business Process Model and Notation (BPMN) para auxiliar na execução de processos. Descrevemos como os conceitos das áreas de processo e de agentes conversacionais podem ser integrados, e também apresentamos um caso de uso em que um *bot* foi implementado seguindo o guia, com o uso do *framework* Rasa - um *framework* de chatbot - e também Camunda Engine - uma ferramenta de gerenciamento de processos, também conhecida como sistema de gerenciamento de fluxo de trabalho ou *workflow management system* (WFMS).

Palavras-Chave: processo de negócio, agente conversacional, fluxo de trabalho, chatbot, Camunda, Rasa.

ABSTRACT

Processes are an essential concept in current society, promoting the standardization, documentation and control of the interactions between businesses, governments, individuals and other organizations. Processes are also used in human workflows, where participants act solo or in collaboration with other participants to fulfill the processes' goals. Being present in most people's lives and routines, processes can be as simple as following a cooking recipe, or as complex as pursuing treatment for a serious disease. Therefore, it is essential to provide support to process participants during the execution of these processes. However, current solutions for process guiding are usually proprietary and lack clarity when trying to help the general public execute daily processes. In this work, we present a guide for building a conversation agent – also known as a chatbot – that leverages the Business Process Model and Notation (BPMN) standard to assist in the execution of processes. We describe how concepts from the process and conversational agent domains can be integrated, and also present a use case in which a bot was implemented according to the guide, with the use of Rasa – a chatbot framework – and Camunda Engine – a business process management tool, also known as a workflow management system.

Keywords: business process, conversational agent, workflow, chatbot, Camunda, Rasa.

SIGLAS

UFRJ	Universidade Federal do Rio de Janeiro
CA	*Conversational Agent*
PACA	*Process-Aware Conversational Agent*
BPMN	*Business Process Model and Notation*
WFMS	*Workflow Management System*
HCI	*Human-Computer Interaction*
BPMI	*Business Process Management Initiative*
REST	*Representational State Transfer*
API	*Application Programming Interface*
JAX-RS	*Jakarta RESTful Web Services*
GUI	*Graphical User Interface*
IPA	*Intelligent Personal Assistants*
IoT	*Internet of Things*
NLU	*Natural Language Understanding*
SDK	*Software Development Kit*
BPM	*Business Process Management*
JSON	*JavaScript Object Notation*

XML - *Extensible Markup Language*

Contents

List of Figures

Chapter 1

Introduction

Millions of people execute processes every day. Processes are intrinsic to our personal and professional lives, from planning a party or scheduling a trip, [1], to hiring new employees or admitting students into a University program [2]. Processes are essential for several reasons:

- Regulating a set of procedures in an attempt to unify the execution of a task among members of a large team;
- Facilitating the identification of the most critical tasks on a list;
- Clarifying the dependencies between tasks so that users can execute them in the right order;
- Improving efficiency in task execution, both individually and for an entire team, if applicable.

In this work, our primary focus is on business processes, i.e., processes that are executed within an organization, and more specifically, processes that require human action, also known as human workflows. According to Dang et al. [3], human workflows are dynamic sets of tasks performed by human participants to reach a shared goal. However, performing tasks in a process permeated with human decisions can be more complicated than it seems. Firstly, tasks may either have a strict or loose order, and this uniformity might leave participants disoriented in complex scenarios. Secondly, dependencies between tasks can be challenging to track in real life, in which case, people might not even be aware of the required information *a priori* and start executing tasks they cannot finish properly. Thirdly, processes that rely heavily on humans may differ according to the context and require extra documentation to avoid errors. [4]

In summary, participating in human workflows can be complex due to varying execution sequences, dependencies between tasks, and multiple contexts. As a result, process participants need a solution to guide them through process instances. In recent years, a new technology has been gaining increased attention: a type of system called "conversational agents" (CAs) [5]. CAs are programs that help develop human-computer interactions (HCI) [6] through speech and written language. More specifically, when a CA relies only on written communication, it can be called a "chatbot". Another fairly common nomenclature for CAs is the more generic term "bot". A CA has the goal of understanding user requests and responding to them while maintaining a natural conversation flow. As will be shown in this work, CAs can be especially helpful because they can capture variables and perform actions while interacting with users in real-time.

Although CAs are flexible enough to foster several HCI scenarios, they lack concepts from the business process domain, i.e., they do not consider the context of a process instance when interacting with users. This means that CAs can miss crucial process information, such as task ordering.

Therefore, our goal in this work is to investigate different methods of integrating process concepts into conversational agents, and ultimately, propose an approach that would enable this integration successfully. For that matter, we present a guide for building a Process-Aware Conversational Agent, also referred to as Process-Aware CA, Process-Aware Bot, or simply PACA, that can help users execute their daily processes.

With a CA interface capable of understanding user requests from natural language and acknowledging sequence flows and dependencies between tasks, the user can recognize all of the available activities for execution at any given moment. Moreover, the CA can follow users throughout the process to avoid ambiguities and mistakes while also collecting the current user's context.

The main contributions of this work are:

- A preliminary discussion about different methods of integrating process context into conversational agents;

- A Process-Aware CA (PACA) Generation Guide that helps a developer create a Rasa chatbot that is connected to a process diagram deployed on Camunda Engine;

- A conceptual model that illustrates the connection between business process and conversational agent concepts;

- Examples of practical processes converted from BPMN diagrams to fully-functioning PACAs.

The methodology used in this work to accomplish the goals and contributions listed above is presented as follows:

1. Selection of the work subject, based on the research group's focus on business processes and the increase in interest in conversational agents in the last few years

2. *Ad-hoc* literature review, both on business processes and conversational agents

3. Implementation of an early-stage "process-inspired" CA using only Rasa, as explained in Section 4.1

4. Research about Camunda Engine, a Workflow Management System (WFMS)

5. Implementation of an actual Process-Aware CA (PACA), now integrated with Camunda

6. Development of a PACA Generation Guide

7. Implementation of a use case using the PACA Generation Guide as reference

8. Project conclusion

The remainder of this work is organized as follows: Chapter 2 lays down the foundations of this study and explains significant concepts that are present in the business process and CA fields. Chapter 3 presents related work that has been published in the past. Chapter 4 proposes two solutions for closing the distance between business processes and CAs and explains why only one of the proposed solutions was chosen for this work. Then, Chapter 5 presents a guide to help a developer build a Process-Aware Conversational Agent using Rasa and Camunda Engine, while Chapter 6 presents a use case of the guide being employed to build a conversational agent. Chapter 7 discusses key aspects of the implementations presented in the guide and the use case, and lastly, Chapter 8 brings a conclusion to this work.

Chapter 2

Background

In this chapter, we briefly explain business process concepts and the modern notations used for process representation. Then, we give an overview of Conversational Agents and show some examples of highly-capable CAs found in the industry and the academy.

2.1 Business Processes

Every business has to execute several activities. To structure these activities, a process analyst may need to create a workflow, which will be relied on over the project's life cycle. This workflow will represent strategic decisions, tasks, and technological aspects of the project. Thus, it is crucial that this workflow is built using an effective methodology. Works such as [1] and [2] help create a shared understanding of business process concepts and technologies that serves both business administrators and computer scientists, who usually have disparate scopes of knowledge.

For quite some time, businesses used flowcharts to represent these processes. Flowcharts can graphically represent different characteristics of a system and have emerged from a notation system developed in the 1920s with loose standards. In 2004, however, the Business Process Management Initiative (BPMI) developed a notation called Business Process Model and Notation (BPMN) in an attempt to make processes readily understandable by all sorts of users, including business analysts, technical developers, and business users. BPMN is currently in version 2.0, and it is a structured notation for developing graphic representations of business or corporate operations.

One great example that demonstrates the importance of process modelling is present in the chemical industry. When mixing chemicals to create a chemical reactor, or even controlling the already working reactor, processes must be followed thoroughly to avoid disasters [7].

To aid in the execution of these processes, tools like Workflow Management Systems (WFMS) began to rise. These tools help manage and track the state of activities inside a workflow, facilitating the flow of tasks, information, and events. Since most of these systems use the BPMN standard, they also share a significant number of process concepts. Subsection 2.1.1 will introduce these general concepts. Then, in Subsection 2.1.2, there will be presented some of the concepts that are specific to Camunda Engine. Camunda was the WFMS we decided to use for this work because it is open-source and frequently used by our research group.

2.1.1 General Business Process Concepts

Each business process is unique in that each of them has its own purpose and intended result. However, when following the BPMN standard, some characteristics are common to most processes, and these properties can help us understand the business process domain a little more. In this subsection, we will go through each of these concepts following the order in which they might appear in a standard process flow. To better explain each concept, throughout this subsection, we will follow the example of a Restaurant Reservation process, represented in Fig. 2.1.

In this Restaurant Reservation scenario, after the start event, the user can execute the five following tasks in any order. Each task represents the piece of information that the user has to provide to the bot so that it can make the actual reservation. The user needs to inform the desired cuisine, the number of people for the table, the placement preference (indoors or outdoors), additional preferences, and, finally, some feedback about their experience with the Restaurant Reservation system itself.

We will start by analyzing a concept that every typical process – including this one – necessarily has, which is *events*. An *event* is when something relevant to

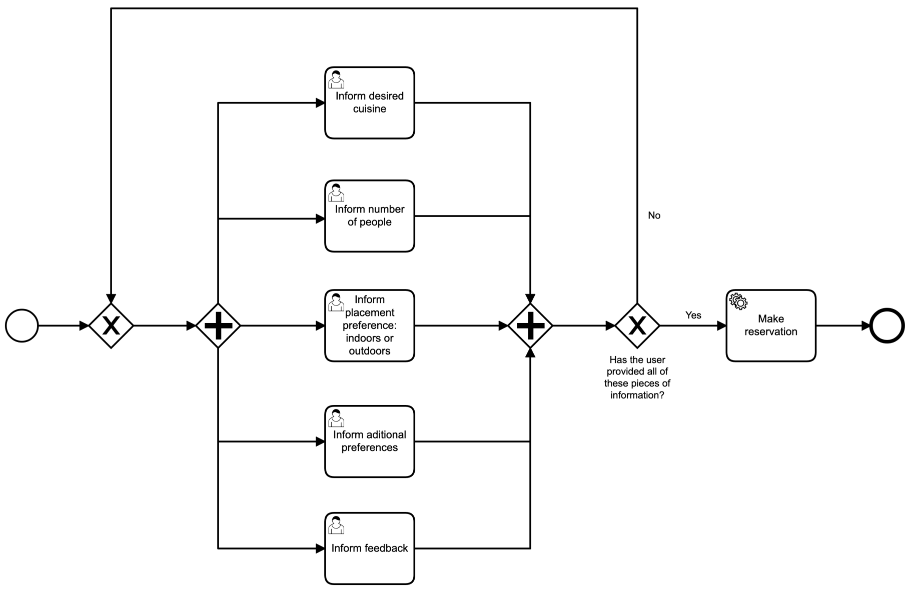

Inform desired cuisine
Inform number of people
Inform placement preference: indoors or outdoors
Inform aditional preferences
Inform feedback
No
Yes
Has the user provided all of these pieces of information?
Make reservation

Figure 2.1: BPMN diagram for the Restaurant Reservation process.

the process happens transiently, i.e, with no duration in time, [2]. The *start and end events* are two examples of those that will most frequently exist and that always occur atomically. For example, in the Restaurant Reservation system, whenever a user is ready to reserve a restaurant and informs that to the system, that is a quick circumstance, i.e., an *event*, and it initiates the reservation process, triggering a series of tasks.

Tasks, for their part, are pieces of work that take a certain amount of time. For example, gathering and storing each user preference for the reservation can take a while, and therefore, these actions are considered *tasks.*

While a *task* refers to an indivisible piece of work, another relevant term that appears in our work is *activity*, which can describe small or larger pieces of work, including sub-processes, and thus, it is a more generic term. According to the official BPMN documentation [8], "a task is an atomic activity within a process flow. A task is used when the work in the process cannot be broken down to a finer level of detail," while "An activity can be atomic or non-atomic (compound)."

In this work, we will most often use the term *task.* However, if the term *activity* comes up, it can be taken as a synonym to *task* since we will not be using the term *activity* to describe any sub-processes or larger operations.

A few sub-types of *tasks* are *User Tasks*, *Manual Tasks*, and *Service Tasks. User tasks* are the ones in which a user has to perform a task with the aid of a software tool, such as shipping a purchase order. They can also require information to be provided by the user, for example, the tracking number related to the approved order. In contrast, *Manual tasks* are the ones that rely solely on physical human execution - without the use of software -, such as filling out a paper document. Finally, a *Service task* is executed automatically by an application program, without human intervention.

In the example of reserving a restaurant table, five *User tasks* are representing each of the five necessary user inputs (desired cuisine, number of people, placement preference, additional preferences, and feedback). Together, these five tasks can be viewed as one single activity, such as "gathering information for reserving a restaurant table".

Besides events and activities, a typical process can also include *gateways*, which are spots where the process flows can diverge or converge, i.e., where the process flow can go in more than one direction. *Gateways* can be of many different types. However, for this work, the most relevant *gateway* types, and the ones we will cover, are *Parallel*, *Exclusive* and *Inclusive gateways*. *Parallel gateways* are used to create parallel flows when all of the outgoing process flows are available, and the process execution needs to go through every one of them. In contrast, in an *Exclusive gateway*, some outgoing flows might not be available. That availability will depend on the evaluation of a condition, and even when multiple paths are available, the execution can only follow one of them. Finally, *Inclusive gateways* also use evaluated conditions to determine path availability. However, in an Inclusive gateway, the process participant can take multiple paths if available, unlike what happens at an Exclusive gateway.

In the BPMN model presented in Fig. 2.1, we have an example of *Parallel* and *Exclusive gateways*. Both of them are represented as diamonds. The difference is that the *Parallel gateway* has a "+" sign in the middle, while the *Exclusive gateway* contains an "X". In that example, there is a *Parallel gateway* indicating that the user can execute any of the five User tasks. However, a little after taking the chosen path, the process execution encounters an *Exclusive gateway* that will check if the user has already provided the necessary information. If not, the process will take the user back to the start, and they will have to take a path on the *Parallel gateway* that is related to one of the missing pieces of information.

Finally, we have *sequence flows*, which are represented as a solid line with an arrowhead. They are used to connect elements in the BPMN diagram – e.g., connecting two tasks, or connecting a task to a gateway – and they show the order in which the elements should be gone through.

Although the range of business process concepts is fairly larger than these select concepts we have covered here, we have decided to limit our review only to the applicable concepts. Thus, inspired by Dumas et al. [2], who illustrated the ingredients of a business process, we decided to select only the process concepts that are relevant to our work and make a diagram portraying their connection (Fig. 2.2).

As shown in the diagram, a business process is mainly composed of Gateways, Events, Activities, and Tasks. Except for the Events, which are mandatory in all business processes [2], all other concepts are optional and dependent on the domain, which is why the diagram portrays them as having "zero or many" cardinality. In the caption box, the relationships between concepts are further explained.

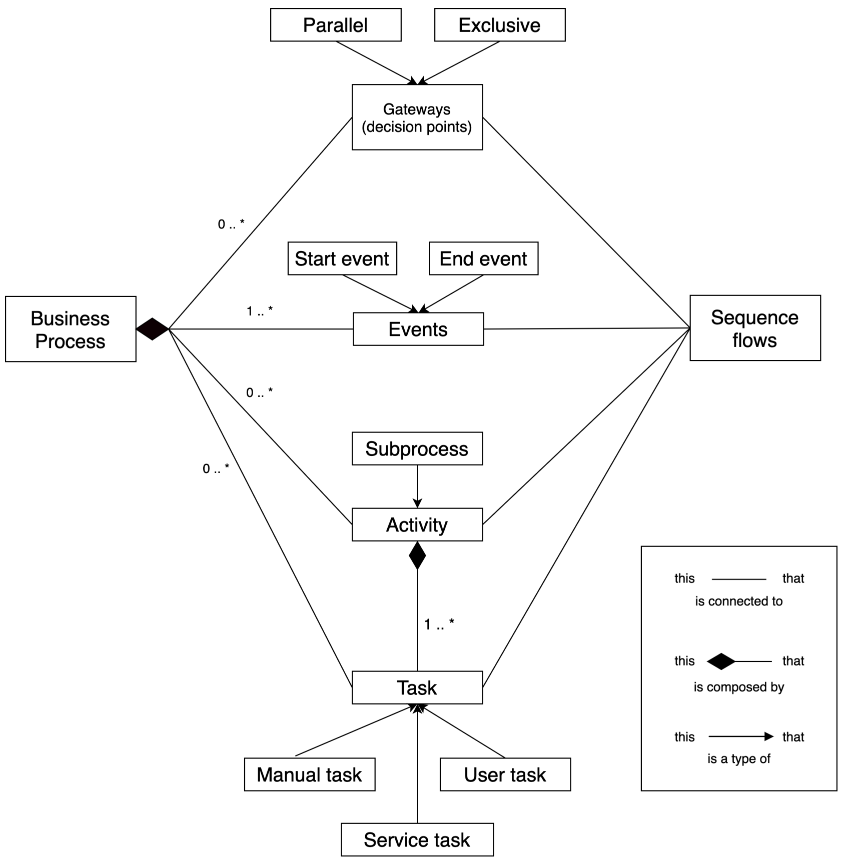


Figure 2.2: Components of a business process.

2.1.2 Camunda Concepts

In Subsection 2.1.1, we have presented a few general process concepts used in the BPMN standard. Now, we will go over the specifics of Camunda Engine, which is the Workflow Management System (WFMS) we will be using throughout this work, and that has its own assortment of features.

A process representation is nothing but a portrayal of how users think that process should be executed. For this representation to be materialized, there is the need to use a WFMS, which can set up the infrastructure for running the process and also monitor its execution. Camunda Engine, the WFMS we decided to use in this work, will be responsible for tracking the process's current state at any given time. It also ensures that the process is executed in the right order.

The first relevant Camunda concept for this purpose is *variables*. *Variables* are a way of adding data to the process so that they can be used in Java classes or for evaluating sequence flows at gateways. *Variables* are usually name-value pairs, and the value can be of various types, such as string, number, boolean, byte array, etc. In the Restaurant Reservation process from Fig. 2.1, the "Inform desired cuisine" task could ask for a `cuisine` variable, which would be a `string`, while "Inform number of people" task could ask for a `num_people` variable, which could be a `number`.

One way of filling in these *variables* is by using *User Task Forms*, also known as *Forms*. This feature allows Camunda to ask for the necessary information right before a specific User task is completed. For that to work, whoever is modelling the process diagram should register a *form* inside the User task, and add a form key and form fields, each field with its own ID, type, and label. Section 5.1 will further explain these steps.

After the BPMN file is finished and ready to be executed, Camunda provides a few options for users to run and manage their processes. One of these options is Camunda Tasklist, which is a front-end application program for executing User tasks. When executing a process, Camunda Engine will automatically add User tasks to a list in Camunda Tasklist. Users can then see each task's related information, assign a task to a specific user, and complete tasks. When a BPMN file has a *User Task Form*, for example, and the user is running the process instance in Camunda Tasklist, a *form* will pop up asking the user to fill in the form fields right as they are completing that task.

However, Camunda Tasklist is not the only method of executing a process instance with Camunda. Camunda Engine provides a Representational State Transfer Application Programming Interface (REST API) based on Jakarta RESTful Web Services (JAX-RS), which allows any kind of software to use its process engine services. The back end of our chatbot will make calls to Camunda's REST API to start the process, execute individual tasks and fill in variables without bringing up Camunda's Graphical User Interface (GUI). There is a broad range of REST endpoints provided by Camunda. However, in this work, we will only need the following endpoints:

- `POST /process-definition/key/{key}/start` – used for starting a process instance, initializing variables and storing the process instance id
- `GET /task?processInstanceId={processInstanceId}` – used for getting the list of all tasks available in a specific process instance
- `GET /process-instance/{id}` – to find out if a process instance is still running or it is already finished
- `POST /task/{id}/complete` – to complete a task, either sending variable updates or not

2.2 Conversational Agents

Conversational Agents have been on the rise since the release of Intelligent Personal Assistants (IPA) such as Siri, Alexa, Google Assistant, and Cortana, and even more so with the increasing popularity of IPA devices, e.g., Amazon Echo and Google Home [9]. These personal assistants can fulfill multiple user requests, such as checking the weather, setting alarms, playing music, and also interacting with Internet of Things (IoT) devices, including smart lamps, smart locks, and more [10].

As defined by Gnewuch et al. [5], conversational agents can be classified regarding either their primary mode of communication (voice-based, text-based, or embodied) [11], or their context, i.e., whether they have a general or task-oriented goal. One way of evaluating a CA is by considering its usability, i.e., its efficiency, effectiveness, and user satisfaction [12]. For example, one trait that makes a CA rank higher in these three quality attributes is maintaining convoluted conversations. Recent works, such as a chatbot named Iris [13], have shown that CAs can be used even in highly-complex mathematical applications. For that to work, though, the CA needs to have a deeper understanding of what the user is trying to say and achieve. Fast et al. [13] have created a highly-intelligent text-based CA by using automata that allow for function composition inside a conversation. This function composition makes it possible for users to make two different requests in a single sentence and have the result of one request be input into the other.

Apart from text-based solutions, a voice-based CA example is presented in Devy [14]. Bradley and colleagues have considered supporting software engineers with a voice-based conversational assistant that utilizes the context elements needed to support software development workflows. Another valuable use of CAs is in healthcare. Miner et al. [15] have developed a study using CAs that leverage user sentiment analysis for mental health treatments.

Although the Iris [13] and Devy [14] chatbots were built from scratch, when building a CA, it is useful to utilize a chatbot framework to avoid recreating all of the CA logic and interface from the ground up. In the industry, there is a number of conversational agent frameworks available, such as Google's Dialogflow[1], IBM Watson[2], Facebook's Wit.ai[3], and also a few open-source frameworks, such as Rasa[4] and Botpress[5].

For this work, the framework we have chosen was Rasa, and towards the end of Subsection 2.2.1, we will explain this choice. Before that, however, we will go over a few CA concepts that are relevant to our work. Some of them are shared between most frameworks, and we will introduce these general concepts in Subsection 2.2.1. Then, in Subsection 2.2.2, we will present some of the Rasa-specific concepts used in the development of this work.

[1]https://dialogflow.com/

[2]https://www.ibm.com/watson/

[3]https://wit.ai/

[4]https://rasa.com/

[5]https://botpress.com/

2.2.1 General CA Concepts

The majority of CA frameworks are composed of two parts: a Natural Language Understanding (NLU) component, which interprets user input, and another component that decides how the bot should act and reply, which could be viewed as the chatbot "back end". The latter is framework-specific and will be explained in Subsection 2.2.2. In this subsection, we will present some of the concepts that are not specific, but rather, used by multiple frameworks. Most of these general concepts from the CA domain are related to the NLU piece since this component is framework-independent. In the NLU scope, the most significant concepts are *intents* and *entities*, because they have a direct influence on the bot's understanding of user input.

To better explain these concepts, it is interesting to use an example. Suppose there is a CA for assisting in restaurant reservations. If the user has a specific restaurant in mind, the bot will book a table at that particular establishment. However, if the user is not sure where to eat and wants to request restaurant suggestions, the bot will search for restaurants based on some provided criteria and then send the results back to the user.

To begin with, an *intent* is the objective of each user message. So, for example, suppose this CA supports two different intents: `request_restaurant`, which is triggered when the user wants to search for restaurants and see all the results, and `book_table`, triggered when the user wants to book a table at a specific restaurant. In that case, whenever the user says "I want to book a table at restaurant X", the CA should classify this message as a `book_table` intent, because the objective of this message is to reserve a table at an already chosen restaurant, instead of trying to discover new restaurants.

Moving on, for the CA to be able to capture precise details of the user's input and, thus, respond appropriately, we must prepare it to receive *entities*. *Entities* are nothing more than essential information that must be extracted from the user's message for the intention to make sense. For example, if the user wants restaurant suggestions, it would be hard to find relevant places without knowing which cuisine they are looking for. With that in mind, `cuisine` could be one of the *entities* in this case, and the bot can extract it from the message.

Finally, after the user's message is understood and processed, the bot needs to determine how to respond to it. In this phase, there is another shared concept between frameworks that is *slots* – the only shared concept we will see in this subsection that is not from the NLU component. A *slot* is a variable in the CA "back end" that will store captured entities for later use. Regarding the previous example, when the user's desired cuisine is captured as an *entity*, this information can be stored in a *slot* so that the bot can use it to make a successful reservation in a later step.

Regarding the actual bot responses, each framework has its own technique. Some of the frameworks are rule-based, which means that every intent has a predefined response mapped out. However, this technique may fall short in certain cases. Firstly, because it requires a certain amount of prep data for the bot to start working at all. Secondly, when going into a conversation edge case, the bot can easily get lost if the developer has not added at least hundreds of conversation paths [16].

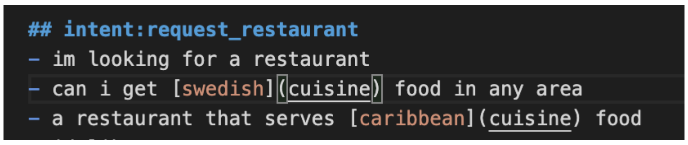
To mitigate those issues, we chose to work with the Rasa framework. Rasa is an open-source CA framework that would allow us to understand its internal code and modify it if needed. Furthermore, Rasa provides a custom Software Development Kit (SDK) for its chatbots that allows developers to fully customize the business logic behind the bot's actions. This will be vital for making our chatbots more suitable to our process-aware scenarios.

2.2.2 Rasa-specific Concepts

Since we decided to use the Rasa framework, we will now introduce an overview of the specific concepts present in this framework. Rasa consists of two components: the previously-mentioned NLU and also Rasa Core, the back-end component of a Rasa bot. As mentioned in Subsection 2.2.1, the NLU component is responsible for making sense of the user's input, while the Core is in charge of deciding what the bot should say or do next.

We will now explore how some of the common CA concepts that were previously introduced are effectively used in Rasa, and also describe a few of its particular features.

As previously explained, the NLU component is responsible for understanding user input and transforming it into a structured output with its respective intents and entities. In Rasa, the text samples that the bot should use to understand each intent and its related entities are stored in the `nlu.md` file. A snippet of the `nlu.md` file concerning the previously mentioned restaurant bot example is depicted in Fig. 2.3.



```
## intent:request_restaurant
- im looking for a restaurant
- can i get [swedish](cuisine) food in any area
- a restaurant that serves [caribbean](cuisine) food
```

Figure 2.3: Fragment of the `nlu.md` file from a Restaurant Reservation bot.

The three sentences demonstrated in Fig. 2.3 represent the `request_restaurant` intent, and they will be used as a training example for when the NLU tries to understand user input. In sentences that include *entities*, the name of the entity - cuisine - is enclosed in parentheses, while the word that symbolizes that entity's value - Swedish or Caribbean, for example - is enclosed in square brackets. However, a single sentence may not convey all the necessary information for a task. To book a flight, for example, only knowing the destination is not enough. Other details are also required, such as departure date (and return date, if applicable), number of passengers, etc. In our restaurant example, we require the desired cuisine, number of people, preferred seating area, additional preferences, and feedback.

To fill in these details, Rasa uses the previously mentioned concept of *slots*, which is how Rasa Core stores the required information. While *entities* are used to extract information from messages, a *slot* is a kind of variable where that information will be stored so that Rasa Core can use it whenever needed. For example, the *slots* containing user preferences can be used at the very end of the process to finally make a reservation. Alternatively, the bot could gradually add more filters to the restaurant search as each slot is filled. If we have a list of restaurants containing different cuisines, right after the user chooses a specific cuisine, it could display only the restaurants that are relevant to their choice.

Regarding the techniques used for slot filling, slots can be filled automatically from extracted entities, or they can be set through *custom actions*. In fact, *actions* are not only responsible for filling slots, but they actually determine what the bot will say or do after a specific intent.

The actions that are solely responsible for sending a specific message to the user can be set up as simple *utter actions*. Due to their simplicity, these actions do not require coding in Python or any other language, unlike regular *custom actions*. Utter actions need to start with the `utter_` prefix. Whenever an action has this prefix, Rasa will search for its template under the `responses` section of the `domain.yml` file, and that template will be the message that is sent to the user.

Custom actions, in contrast, can perform extremely complex operations. One action may be, for instance, responsible for sending the flight reservation request to the airline or travel agency server. Since *actions* are the main part of Rasa Core, i.e., the bot's back end, they can be written in any server language, such as Node.js, .NET, Java, etc. However, Rasa provides a Python SDK, which greatly simplifies the development effort, and thus, Python is the language we have chosen for our actions. During development, it is possible to make *actions* access slots and their values, so, for example, one can create a condition that sending a reservation request can only be carried out if all the necessary slots have been filled. That is, even if the user tries to finalize the booking every time they fill a slot, the reservation will only be made after all required slots are filled.

An essential type of *custom actions* is *form actions* – also known as *forms* –, which are responsible for filling slots and verifying if all of the required fields are loaded. In Rasa, *forms* create a sequence of interactions to ask the user for required information in an orderly fashion and without the need of specific intents. As we have seen before, outside the scope of a *form*, every time the bot receives a message from the user, it tries to fit it into an intent. That would raise the problem of needing to know *a priori* all the possible answers the user could give and then map each of these to different intents. Moreover, what would happen if the bot requested the number of people for the table and got back a single number without further context? How would it be mapped into an intent? In such cases, *forms* are a great feature to be used.

When configuring a *form* action, we have to specify its required slots in the action's code and then add to the `domain.yml` file the questions that the bot should ask when requiring each slot. After sending each request, the bot will expect to receive a value for that particular slot, and will not try to map the user's response to any intents. However, if necessary, the bot can transform the response into a type of data that the slot is expecting. For instance, if it is a boolean slot, it is possible to transform a “yes” or “no” response into “true” or “false”, or if it is a number slot, it is possible to make “five” turn into “5”. Later, after all the required slots are filled, the form can then run some code to submit the collected information to whichever service.

Finally, the last major concept left to introduce is the concept of *stories*, which are conversation examples used to train the Machine Learning model for the bot’s responses. Rasa utilizes the multiple conversation examples - *stories* - to decide the best response in a particular moment by considering its context. They help the bot decide, for example, which action should be executed after receiving a certain intent from the user.

Throughout this work, we will edit specific Rasa files, and the reader needs to have an overview of what these files are and what they do. The following list describes each of these relevant files:

- `nlu.md` – provides examples of phrases that should be mapped to each intent so that the NLU component can properly identify the intent corresponding to the user input;
- `stories.md` – delineates conversation paths, including which actions follow which intents, so that Rasa Core can learn from them and identify the next action to take at any point in time;
- `actions.py` – carries the code to be used for each action. As previously mentioned, actions are usually written in Python due to Rasa providing a Python SDK, but they can be written in any other server-side language;
- `domain.yml` – contains a summary of all of the information that is relevant to the bot, such as intents, entities, actions, etc., so that the bot can understand what to expect from the other files and identify is any information is missing ;
- `config.yml` – provides basic configuration for the bot.

Chapter 3

Related Work

This chapter will present a few task-assisting conversational agents developed in the academy and a few process management tools available in the industry. To the best of our knowledge, there is not much literature connecting the formal concepts of business processes and conversational agents. However, there have been a handful of studies towards the implementation of process-guided CAs in the past.

In [17], Toxtli et al. created a tool called TaskBot that aimed to help teams complete assignments. The authors intended to reduce the onus of context switching by using the bot to assign tasks to team members and remind them to start or finish their tasks. This bot would reduce the need for team members to simultaneously check multiple tools such as Trello, Slack, e-mail, and GitHub. Despite this tool not using the concept of business process as a workflow, it does utilize the concept of tasks, even if these tasks are disconnected.

Iris, itself [13], as mentioned in Section 2.2, is also an example of a process-guided CA, and it supports complex processes by using function composition. After evaluating their solution, Fast et al. discovered that users executed data science tasks approximately 2.6 times faster using the Iris chatbot than sklearn and Jupyter. One significant advantage to their solution, according to participants, is the fact that, by using a bot, they did not have to remember which functions to call, as Iris would walk them through the process execution depending on which procedure they wanted to perform.

In [18], Cranshaw et al. have built and evaluated an app called Calendar.help whose purpose is assisting in scheduling and rescheduling meetings. The whole process of setting up meetings is divided into microtasks and macro tasks. Microtasks are simpler functions that a CA can do, such as sending e-mails and checking invitees' availability. Meanwhile, macro tasks are more complex, such as rescheduling a meeting with many invitees. In the latter case, a human agent would step in to help with the task. This study is closer to ours because their proposed CA also uses the concept of process workflows – in this case, for meeting scheduling. Cranshaw et al. developed a process representation that depicts most of the micro and macro tasks involved in setting up meetings between coworkers, including conflict solving. Unlike our approach, they opted not to use BPMN for their representation. However, their diagram is still a perfectly valid portrayal of their process, including tasks and gateways.

In recent years, there has been a rise in the appearance of business process management (BPM) tools such as Kissflow[1], Process Street[2] and Pipefy[3] that aim to help businesses and their employees track the execution of essential and recurring processes. Camunda itself is also a BPM tool. These kinds of tools usually use features such as checklists, forms, and workflows to ensure all team members are executing the right task at any given time. Several companies use these tools to support the execution of their processes.

Nevertheless, when business processes become too large, employees can get lost in the middle, unsure about which tasks are available to them, or even make mistakes during the execution [4]. In these cases, these process management tools, including Camunda, due to their complexity may not help these workers at all. Therefore, the tool that we will present in this work aims to complement BPM tools – in this case, Camunda – to make them more user-friendly by leveraging natural language understanding.

[1] `https://kissflow.com/`

[2] `https://www.process.st/`

[3] `https://www.pipefy.com/`

Chapter 4

Integration Rationale

This chapter aims to explain the rationale behind our work and the implementation we developed throughout our studies. Workflow Management Systems (WFMS), as described in Section 2.1, are instrumental in controlling the process execution; however, they typically lack a flexible user experience. On the other hand, CAs provide a better user experience, but they lack awareness of the process state. CAs also need an initial setup with intentions, entities, etc. For these reasons, combining business process concepts and CA concepts would allow the creation of PACAs and serve as a building block to enhance WFMS. For this purpose, in this section, we leverage the Theoretical Background and Related Work presented, respectively, in Chapter 2 and Chapter 3, to implement a solution that combines the business process and CA domains. This solution will allow us to generate a bot that could decide which tasks are available in a process instance and help guide the user throughout its execution.

Fig. 4.1 portrays the expected operation of our solution. Whenever a user sends a message to the bot requesting advice on how to start a process, the bot will reply with all of the available tasks. The user will, then, choose to execute any task that is available, and as soon a task is completed, the bot should send the next available tasks until the process is finished. In case the user tries to execute a task that is not available, the bot should prevent this execution, by informing to the user that the task is not available.

The first step in building such a bot was to take a subset of concepts from the business process domain and analyze how each of these concepts is related to the terms used in conversational agents.

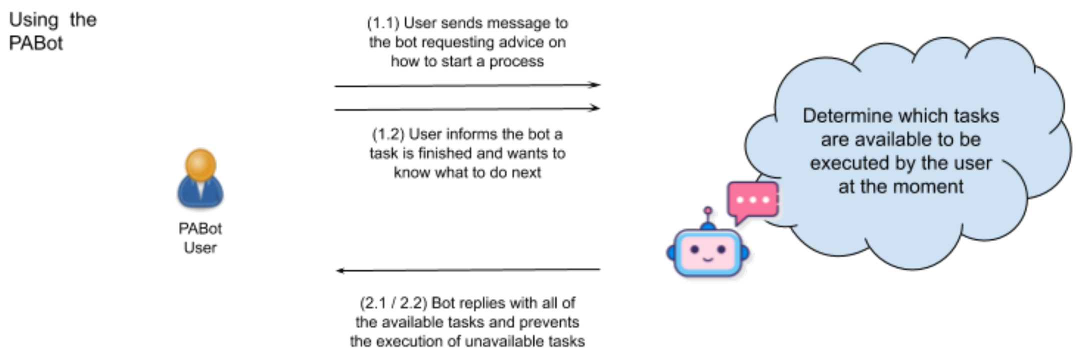

Using the PABot
(1.1) User sends message to the bot requesting advice on how to start a process
(1.2) User informs the bot a task is finished and wants to know what to do next
PABot User
Determine which tasks are available to be executed by the user at the moment
(2.1 / 2.2) Bot replies with all of the available tasks and prevents the execution of unavailable tasks

Figure 4.1: Overview of the expected operation of the Process-Aware Conversational Agent.

The business process concepts we decided to examine for now are:

- task
- gateway
- sequence flow
- start event
- end event
- process variable (Camunda)
- user task form (Camunda)

Similarly, the CA concepts that were chosen:

- intent
- action
- slot
- story

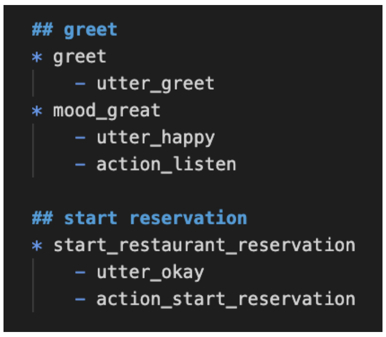
Under business process concepts, as explained in Section 2.1, there are both general and Camunda-specific concepts, the latter identified with "Camunda" inside parenthesis. Meanwhile, regarding CA concepts, it is worth noting that each framework has its own characteristics and feature. Hence, the CA terms listed above are primarily valid for the Rasa framework. Other environments, however, can have analogous attributes using other terms.

To properly formulate this connection between business process and CA concepts, we have gone through two phases. First, we tried building a process-inspired bot with only Rasa, without using any Workflow Management System (WFMS). Although this technique did not provide us with a substantial connection between business process and CA concepts, it was a necessary step towards understanding the differences between both domains and how they could be connected. This first phase will be described in Section 4.1.

Then, in the second phase, we successfully integrated Rasa with Camunda, a WFMS. This integration has allowed us to form a more solid connection between the business process and CA domains and, thus, create an effective Process-Aware Conversational Agent. This approach will be the primary base for our work, and it will be described in detail throughout Section 4.2 and Chapter 5.

4.1 Manual Integration

In this section, we start a preliminary discussion about how a process-inspired bot could be built with only a CA framework - in this case, Rasa -, without using any Workflow Management Systems. Therefore, in this section, we will not be using any Camunda-exclusive concepts. The goal here is to manually associate concepts from the business process and CA domains in a way that, after taking a BPMN file, looking at its components, and writing the bot's code, Rasa will be solely responsible for the process execution, without any aid of workflow engines or specialized process-guiding tools. Since this first bot will not use any of these tools, we cannot call it process-aware. That is why we refer to this first implementation as being process-inspired. Even though this is not a scalable solution, it was part of this work's evolution and a necessary step towards an actual Process-Aware Conversational Agent. We will take a simple look at how one can execute a manual integration. However, for simplicity reasons, in this work, we will not go deeply into explaining all



```
## greet
* greet
  - utter_greet
* mood_great
  - utter_happy
  - action_listen

## start reservation
* start_restaurant_reservation
  - utter_okay
  - action_start_reservation
```

Figure 4.2: Fragment of the `stories.yml` file from a Restaurant Reservation bot.

the steps to making this manual integration. Instead, we encourage the integration of Rasa with a Workflow Engine, for which you can find detailed instructions in our PACA Generation Guide, in Chapter 5.

To begin a manual integration, the first relevant process concept to be examined is the start event, which should be represented as an intent. In this work, we are covering processes that are initiated by a user, and an intent is needed to help the bot understand when a user asks to start a process. Going back to the Restaurant Reservation process represented in Fig. 2.1, we could call it a `start_restaurant_reservation` intent. It would be triggered when a user said something like "I would like to book a restaurant", and this intent would provoke the start of the process.

After identifying the first intent, the bot will try to match it to a set of applicable stories. Fig. 4.2 demonstrates a fragment of the `stories.md` file for a Restaurant Reservation bot. In the figure, there are two stories. Each story has its beginning marked by double hash signs (`##`), followed by that story's name. Then, each intent is preceded by an asterisk (`*`), and each action is preceded by a hyphen (`-`). The first of our two stories begins with the `greet` intent, and the other, with the `start_reservation` intent. If the user says "hi", it will go into the `greet` story, which is not directly related to the process but still needs to exist to enable the bot to carry out more natural conversations. In contrast, if the user already starts the conversation saying they want to book a restaurant, the conversation flow will automatically go into the second story.

The process's sequence flows, in this manual integration method, are contained in the stories. In Fig. 4.2, the fact that `utter_okay` is followed by `action_`⌋

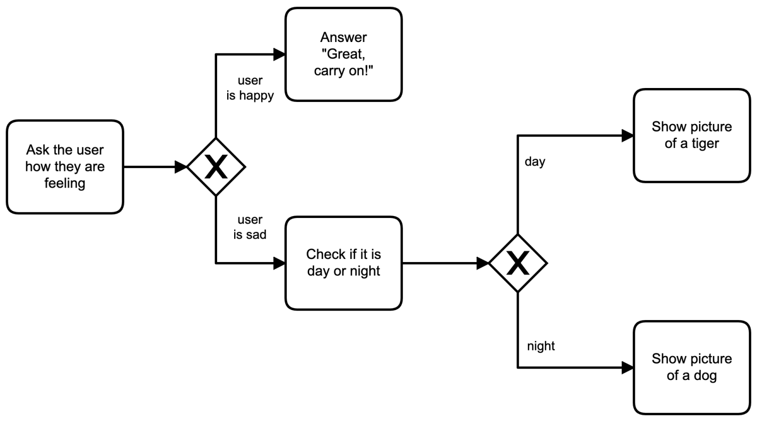

Ask the user how they are feeling
user is happy
Answer "Great, carry on!"
user is sad
Check if it is day or night
day
Show picture of a tiger
night
Show picture of a dog

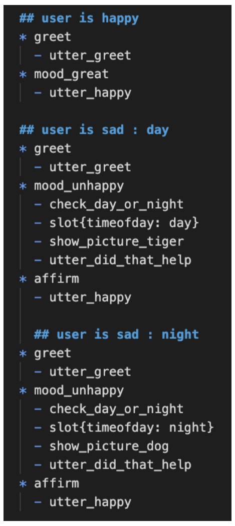
Figure 4.3: Fragment of the BPMN diagram for the "User Mood" process.

`start_reservation` means that that latter should happen right after the former, just as a sequence flow would represent the movement from one task to the next.

Regarding process gateways, the way they can be constituted in Rasa is by using multiple stories, with slots being used to differentiate them if needed. To illustrate this scenario, we will take a look at a different process from the Restaurant Reservation one we have been examining so far. Even though the Restaurant Reservation BPMN contains gateways, they are merely used to check if the reservation can be made. Now, we are going to see how gateways can actually create a branch in the process flow.

For this purpose, we will examine a "User Mood" process, in which the bot asks the user how they are feeling, checks if they are happy or sad, and then replies accordingly. If the user is happy, the bot should then reply with a message saying "Great, carry on!", whereas if the user is sad, in an attempt to "cheer the user up", it should either show a picture of a tiger or a dog, depending on whether it is day or night. Fig. 4.3 represents a small fragment of this "User Mood" process, while Fig. 4.4 shows its respective Rasa model.

According to the diagram shown in Fig. 4.3, if there were only two possible responses – one for when the user is happy and another for when the user is sad –, there would only be one gateway (to check on the user's mood), and consequently, only two stories would be necessary. However, when the user is sad, the bot needs to check the time of day and reply adequately. Hence, there comes another gateway, and therefore one more story, to check the current time and show the appropriate picture.



```
## user is happy
* greet
  - utter_greet
* mood_great
  - utter_happy

## user is sad : day
* greet
  - utter_greet
* mood_unhappy
  - check_day_or_night
  - slot{timeofday: day}
  - show_picture_tiger
  - utter_did_that_help
* affirm
  - utter_happy

  ## user is sad : night
* greet
  - utter_greet
* mood_unhappy
  - check_day_or_night
  - slot{timeofday: night}
  - show_picture_dog
  - utter_did_that_help
* affirm
  - utter_happy
```

Figure 4.4: Fragment of the `stories.yml` file from a “User Mood” bot.

In this manual integration scenario, the act of checking the gateway and deciding on one path or another relies solely on Rasa features. In Rasa, the decision can be made based on different intents or based on slots set by custom actions. For example, the decision at the “user mood” gateway, as can be seen in Fig. 4.4, depends on the intent that follows the `utter_greet` action, which sends a message saying “Hey! How are you?” to the user. If the user replies saying they are happy, then the computed intent will be `mood_great`, and the bot will fall into the first story. If the user replies saying they are sad or something similar, then the gauged intent will be `mood_unhappy`, leading the bot to the second or third story.

In the case of an evaluated `mood_unhappy` intent, what comes next is a custom action. The `check_day_or_night` action will get in contact with the system's clock to check for the current time. However, the only way to leverage the acquired information to decide which action to take is by using slots. As explained in Section 2.2, slots are variables that store crucial information required by the bot. In this case, there is a slot called `timeofday`, which can be set to either `day` or `night`, and its current value will determine which of the stories the bot will choose to follow. Each story that accesses the `timeofday` slot value must have a line containing either `- slot{timeofday: day}` or `- slot{timeofday: night}`, as that will represent which value is related to that specific story. For example, in Fig. 4.4, `user is sad : day` should be followed when `timeofday` slot value is `day`, while `user is sad : night` should be followed when the value is `night`. This verification must be made right after the `check_day_or_night` action because that is the action that will set the slot value, which will be crucial for determining the appropriate story.

In this particular process fragment, there is no end event. However, if there were such an event and the tasks preceding it were required, this evaluation could be done with slots and custom actions. In that case, every task would need a boolean slot indicating its execution status. They could all start as `false`, and each task execution would set its respective slot to `true`. After each task, a custom action would run to check which slots are `true` and which are `false`. Lastly, when all slots were `true`, that would mean that all tasks had been executed, and thus, the process execution has reached the end event. The bot could then inform the user that they have finished the process.

4.2 Integration with Workflow Engine

In this section, we demonstrate how we have used a Workflow Management System to control the general execution of a BPMN process, more specifically, delegating the decisions inside the process, such as the availability of certain tasks after a gateway, to the WFMS. Although in this work we have decided to use Camunda Engine, any other WFMS could technically replace it. However, to use a system other than Camunda, some changes would have to be made since we use concepts and tools that are Camunda-exclusive, as explained in Subsection 2.1.2.

According to Section 4.1, for the manual integration, we used only Rasa features, such as slots and stories, to guide the conversation, and hence, the process execution. Although that did work for that specific scenario, that form of implementation is neither process-aware nor scalable. It is not process-aware because, even though that technique uses the process to help guide the bot's implementation, Rasa does not use the process itself during the conversation, which is why we referred to that bot as process-inspired. Also, the manual approach is not scalable because, in that scenario, it was not possible to generalize the code generation for the bot. Each scenario would need a completely different implementation, with not much re-utilizable code. Therefore, an extensive process would increase the complexity of generating an appropriate bot, taking a considerable amount of time and effort.

By integrating Camunda, a WFMS, with Rasa, it becomes easier to write the bot's code. Since Camunda will be responsible for evaluating all significant decisions, this chore is removed from Rasa. This paradigm shift is only logical as well since a WFMS is specialized in following and managing business processes and workflows, whereas Rasa is not.

Our proposed solution is to be executed in two phases: configuration and usage. The former is shown in the upper section of Fig. 4.5, while the latter is in the image's lower section. During the configuration phase - upper section -, the process analyst has to create a process model in Camunda Modeler (a desktop application for modelling BPMN diagrams) and make necessary adaptations to the model according to the PACA Generation Guide presented in Chapter 5 (Fig. 4.5.1). Then, the process analyst can deploy the process model to Camunda Engine (Fig. 4.5.2). Later, the developer will read this BPMN and use the PACA Generation Guide to convert each process feature that is present in the BPMN file, such as tasks and events, into Rasa files (Fig. 4.5.3). Finally, with the process deployed to Camunda Engine and Rasa fully configured, as soon as the bot is up and running, Rasa will be able to communicate with Camunda Engine, forming a fully functioning Process-Aware Conversational Agent (Fig. 4.5.4).

With a working bot, the user can then utilize it to navigate the process. As presented in the lower section of Fig. 4.5, whenever a user sends a message asking to start a process and requesting advice on how to start it (Fig. 4.5.5), the bot will list all of the tasks that are available to the user.

Then, the user can start executing an available task. It is worth noting that, even though in Camunda a user can "claim"a task to start its execution and then "complete"it after they are done, in this work, we will not support the claiming of tasks, only completion. This means that when a user informs they are "booking a flight" or "have booked a flight", both of these sentences will mean that the user is trying to complete (not claim) a task in Camunda.

Since a task might not be available at that moment, Rasa can prevent the completion of unavailable tasks (Fig. 4.5.6) since it has access to the list of available tasks at any given moment. In fact, Rasa will prevent the completion of that task

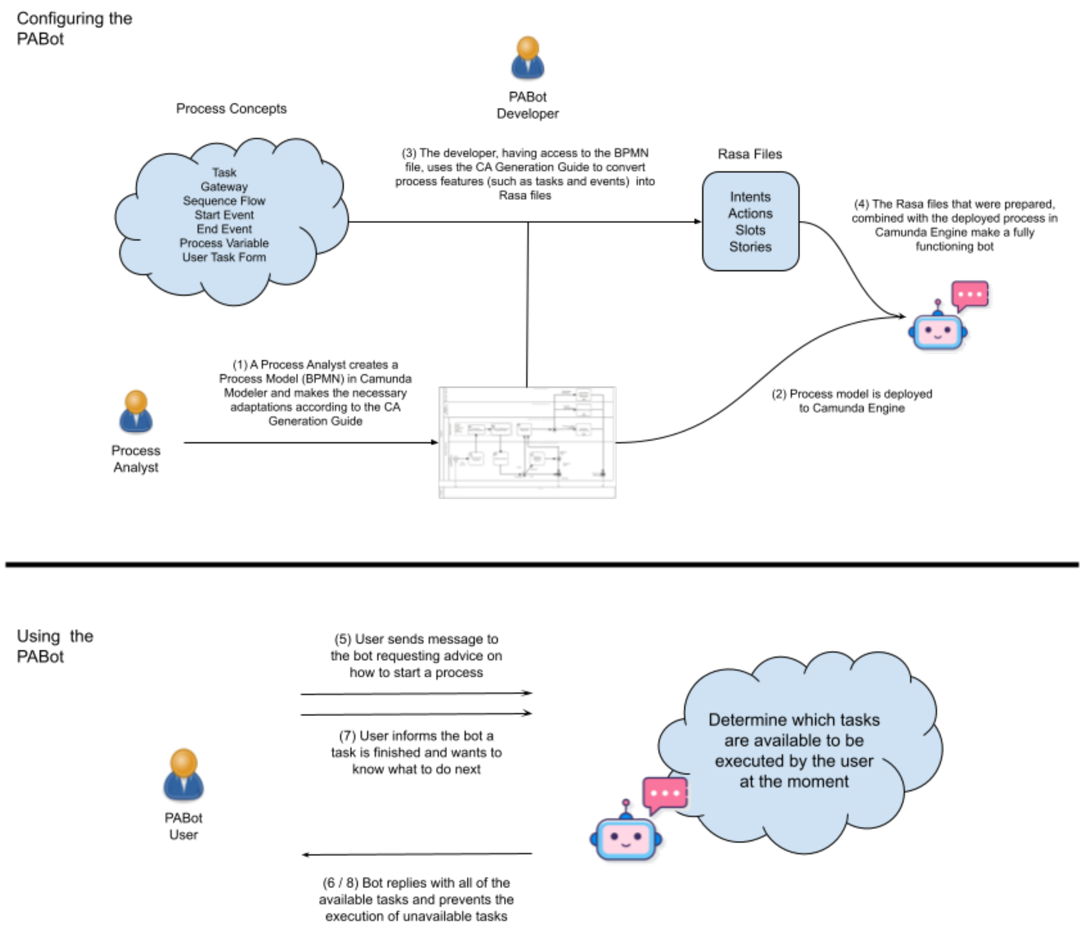

Configuring the PABot
Process Concepts
Task
Gateway
Sequence Flow
Start Event
End Event
Process Variable
User Task Form
PABot Developer
(3) The developer, having access to the BPMN file, uses the CA Generation Guide to convert process features (such as tasks and events) into Rasa files
Rasa Files
Intents
Actions
Slots
Stories
(4) The Rasa files that were prepared, combined with the deployed process in Camunda Engine make a fully functioning bot
(1) A Process Analyst creates a Process Model (BPMN) in Camunda Modeler and makes the necessary adaptations according to the CA Generation Guide
Process Analyst
(2) Process model is deployed to Camunda Engine
Using the PABot
(5) User sends message to the bot requesting advice on how to start a process
(7) User informs the bot a task is finished and wants to know what to do next
PABot User
(6 / 8) Bot replies with all of the available tasks and prevents the execution of unavailable tasks
Determine which tasks are available to be executed by the user at the moment

Figure 4.5: Overview of the necessary steps to configure and use the Process-Aware Conversational Agent.

even if the user insists. That is an essential feature for rigid processes, as it might be necessary to follow a specific order of task execution. Although one might argue that Camunda should be the one responsible for blocking the completion of an unavailable task, in this case, Rasa is doing so because it might need to ask the user for task-related information before sending the task completion request to Camunda. This procedure for completing tasks will be further explained in the PACA Generation Guide, in Section 5.3.

After the user informs they are executing or have executed a task (Fig. 4.5.7), if that task is available, the bot will, in turn, once again reply with the next available task(s) (Fig. 4.5.8) until the process is finished as a whole.

In what follows, we shall see a concrete connection between business process and CA concepts. We will describe the step-by-step towards building a Process-Aware CA, connecting each part of the BPMN model in Camunda to the Rasa framework.

Chapter 5

PACA Generation Guide

The purpose of this chapter is to showcase step-by-step instructions on how to generate your own Process-Aware Conversational Agent, given you have some knowledge in Python and BPMN. This guide also considers that the process has already been modelled into a BPMN diagram. For this guide, we will be using Rasa 1.8.2, Rasa SDK 1.8.0, Camunda BPM Platform 7.15.0, and Camunda Modeler 3.4.1.

This guide consists of the following steps:

1. Preparing the BPMN
2. Building the default Rasa bot
3. Adding process-specific pieces to the Rasa bot
4. Running Camunda and the bot

After these four steps, there is one last section in this chapter (Section 5.5), which describes the conceptual connection between the business process and CA domains. That last section aims to give the reader a clearer understanding of how concepts from each domain are connected and facilitate the adaptation of the bot's code if needed.

Throughout this guide, we will be exploring a "Trip Planning" scenario we have elaborated to explain each step of our PACA implementation. "Trip Planning" serves as an excellent example for our studies because it is well known and has dependencies between tasks. The code for the "Trip Planning" bot that will be created throughout the guide is available on GitHub[1]. The guide we are about to present is already comprehensive; however, it is helpful to have the code in hand since it has a lot of boilerplate code that can be copied when creating your own PACA.

Before we dive into more technical matters, we first need to explain the process, with its tasks and dependencies. In this process, the conceptualized tasks are the following:

- booking a flight
- booking a hotel
- booking a transfer
- booking a tour

Fig. 5.1 presents the diagram for this process, which includes the four designated tasks, the process gateways and sequence flows.

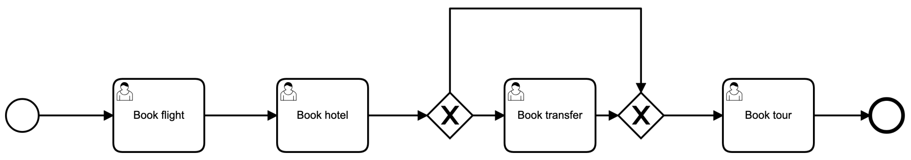


Figure 5.1: BPMN diagram for the "Trip Planning" process.

When elaborating this scenario, the team has decided to enforce the dependencies between particular tasks. For example, it is risky to book a hotel before booking a flight from our point of view. That is because, before purchasing the flight ticket, you do not have 100% certainty of the day on which you will be arriving at your destination. For instance, a good deal could appear and induce you to book a flight that would arrive one day earlier or later than you originally intended, and if you had already booked the hotel, that could be a problem. That is why we presumed that one should only book the hotel after purchasing the flight ticket, and we wanted to portray this dependency in the process.

Regarding transfers, one may or may not want to book a transfer between the airport and the hotel, so we made this task optional, which is why there is a path

[1]https://github.com/luis-f-lins/process-aware-conversational-agent

circumventing this task. However, if the user does want to book a transfer, they first need to have already booked both the flight and the hotel, since the transfer company would need both the arrival and departure dates and also the hotel address. Thus, we added a requirement that the transfer should only be booked after booking the flight and hotel.

Finally, we have tour booking, which is the last part of the process and can be executed right after booking the hotel or the transfer, depending on whether the user chose to book a transfer or not. In this scenario, for simplicity reasons, we decided to also portray the tour booking as required. Then, after booking the tour, the user reaches the end event, finishing the process. The possible execution paths can be seen as:

```
1. FLIGHT -> HOTEL -> TRANSFER -> TOUR
2. FLIGHT -> HOTEL -> TOUR
```

Now that the "Trip Planning" process is explained and its BPMN diagram has already been modelled (Fig. 5.1), for it to work on our PACA, there are some minor adjustments that need to be made to its BPMN file. The following section will further explain these necessary modifications. When creating your own PACA with a different process scenario, you will also need to execute the following steps on your BPMN file after your process has been modelled.

5.1 Preparing the BPMN

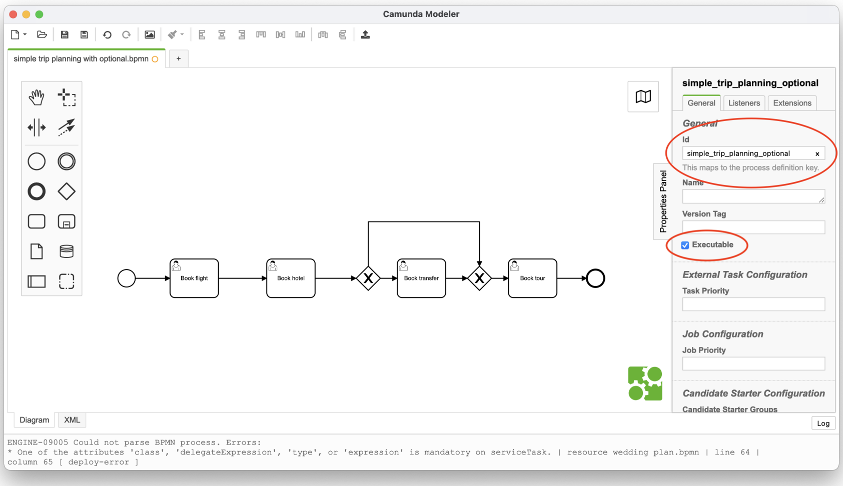
This section will go through some steps for preparing the BPMN file for the Process-Aware Conversational Agent. These initial steps will be executed in Camunda Modeler, a desktop application program for modelling BPMN process diagrams. It allows a user to not only visually model a BPMN workflow but also edit different properties that are needed for process execution. Camunda Modeler can be downloaded straight from Camunda's website[2], and this guide presumes that this program is already installed.

[2] https://camunda.com/products/camunda-platform/modeler/

5.1.1 Setting general process properties

After going into Camunda Modeler and loading your BPMN file, the first step is making sure your process has a "process definition key". When you click on a blank space in the canvas, it should show on the right sidebar an Id field that maps to the "process definition key" (Fig. 5.2). You need to set this to an Id with no spaces, and it will be used later while writing Rasa files. In this case, the process Id will be `simple_trip_planning_optional`.

You also need to make sure the process is set as executable, otherwise Camunda Engine, the process engine we will later use, will not be able to run any instances of this process definition.



simple_trip_planning_optional

General

Id: simple_trip_planning_optional

This maps to the process definition key.

Name:

Version Tag:

[✓ Executable]

External Task Configuration

Task Priority:

Job Configuration

Job Priority:

Candidate Starter Configuration

Candidate Starter Groups

```
ENGINE-09005 Could not parse BPMN process. Errors:
* One of the attributes 'class', 'delegateExpression', 'type', or 'expression' is mandatory on serviceTask. | resource wedding plan.bpmn | line 64 |
column 65 [ deploy-error ]
```

Figure 5.2: Process details shown in the sidebar of Camunda Modeler.

5.1.2 Setting each task's type

Then, you should set a type for each task, making each of them either a User or Service Task, which are the types of tasks that this guide supports. You can choose the task type by clicking on a task and then on the wrench icon that appears, as shown in Fig. 5.3. The rule is: if it is a task that can be done automatically by the system, it should be a Service task.

However, it is worth noting that all Service tasks need to have the "Implementation" field filled out, as shown in Fig. 5.4a. This requirement means that the developer needs to implement a service that will be connected to that Service task and have it running through an expression, Java Class[3] or another supported implementation method. If there is a Service task with no implementation, the process execution will fail when reaching that task.

In this "Trip Planning" process, all tasks are User tasks, which means they all require user participation – the user will have to explicitly inform the bot that they have completed a task.

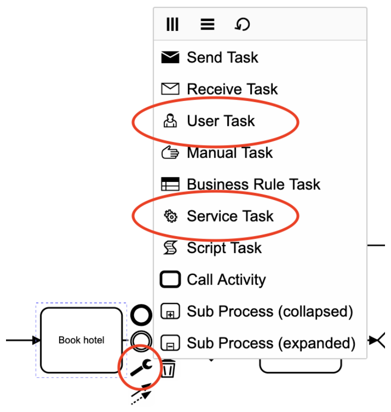


Figure 5.3: Action Menu shown in Camunda Modeler.

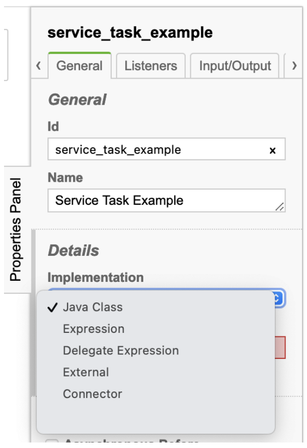
5.1.3 Configuring User tasks

For each User task, you need to click on it and, on the sidebar (Fig. 5.4b), make sure each of them has an Id (Fig. 5.4b.1), with no spaces, and a name (Fig. 5.4b.2), which can and should contain spaces because it will be used when presenting the available tasks to the user. The names of all available tasks will be listed whenever the user completes a task or asks something like "What is still left to do?" (Fig. 5.5).

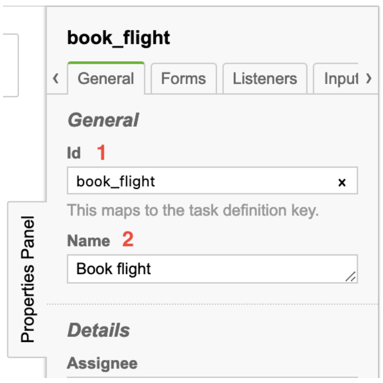
5.1.4 Requesting information with User tasks (optional)

A User task might require information or not. In this "Trip Planning" example, when the user completes the "Book flight" task, the bot asks what the flight date is, so that it can store the date in a process variable in Camunda (Fig. 5.6a). In contrast, when the user completes the "Book tour" task, no further information is requested, and the process is finished (Fig. 5.6b).

[3] https://docs.camunda.org/get-started/java-process-app/service-task/

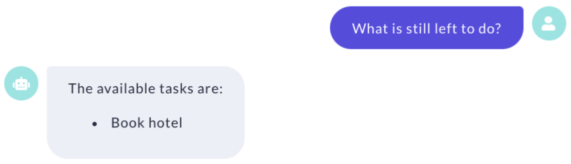


(a) Service task details.



(b) User task details.

Figure 5.4: Task details shown in the sidebar of Camunda Modeler.



What is still left to do?

The available tasks are:

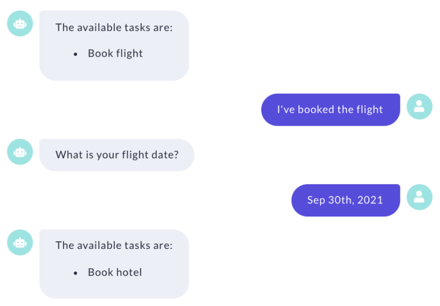
- Book hotel

Figure 5.5: Fragment of a conversation showing the `now_what` intent.

To make the bot request information in a User task, you need to add a form to the task. Do not mistake Camunda forms, examined in this subsection, with Rasa forms, presented in Section 2.2.2. This guide will come back to Rasa forms later, in Section 5.2, but for now, all mentioned forms are Camunda's User Task Forms.

This process needs two of these forms, one for the "Book flight" task and another for the "Book hotel" task. The "Book flight" task, as previously mentioned and presented in Fig. 5.7a, should ask for the flight date, which will be stored in a variable called `flightDate`. This process only stores the flight date without using it later, but it could be used for evaluating gateway sequence flows or in a service task if needed.

The "Book hotel" task (Fig. 5.7b), for instance, will require some information that will indeed be used. However, unlike the "Book flight" task, which requires the flight date, the "Book hotel" task will not ask for the hotel name. This task will need to ask for a piece of information that is more essential, which is whether the user wants to book a transfer or not. According to Fig. 5.1, the "Book transfer" task should be optional, and thus, there should be a variable called `userWantsToBookATransfer`, that can be evaluated at the proper gateway. First,



The available tasks are:

- Book flight

I've booked the flight

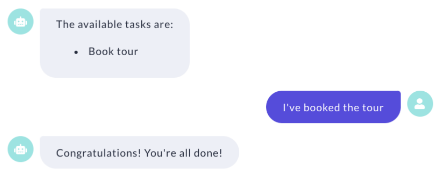
What is your flight date?

Sep 30th, 2021

The available tasks are:

- Book hotel

(a) Booking the flight, which requires information.



The available tasks are:

- Book tour

I've booked the tour

Congratulations! You're all done!

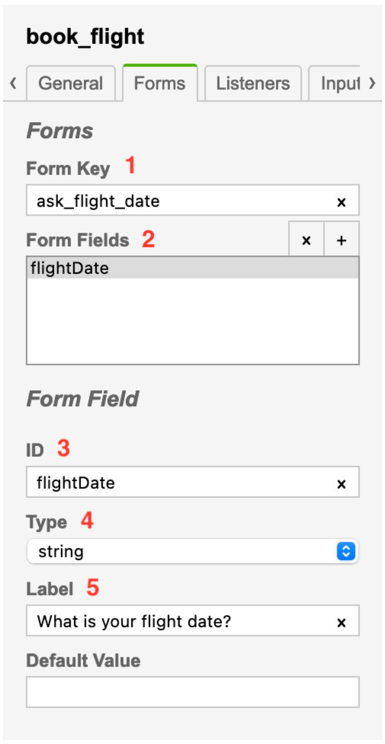
(b) Booking the tour, which does not require information.

Figure 5.6: Fragments of conversations.

the bot will ask "Do you want to book a transfer?" as soon as the user completes the "Book hotel" task. Then, based on the user's response, this variable will be filled with either `true` or `false`, and Camunda Engine will use its value to decide whether to present the "Book transfer" task or not.

With some extra work, the "Book hotel" task could even request two pieces of information, for example, the `userWantsToBookATransfer` variable and also the hotel name. However, for simplicity reasons, this guide will cover the scenario of requesting only the most relevant piece: the `userWantsToBookATransfer` variable.

To create forms for requesting information, you need to click on the respective task and go to the Forms tab on the sidebar, presented in the two sub-figures of Fig. 5.7. The form key (Fig. 5.7.1) is a name that should represent what this form will request, and it will be used as an action's name in Rasa, so it is vital to choose a name with no spaces. This name should be something like `ask_X`, with X being the required information, such as `ask_flight_date`. Then, for each information requested, add a form field (Fig. 5.7.2) by clicking on the button with the plus sign (+) next to Form Fields. Each form field needs an ID (Fig. 5.7.3), which will be the name of the process variable, a type (Fig. 5.7.4), and a label (Fig. 5.7.5), which is essentially the question that the bot will ask to request that information from the user. If you want a task to request more than one piece of information, you can click on the button with the plus sign (+) again to add more fields, filling in the details for each new variable. In contrast, if it is not necessary to request information in any User task, you do not need to create any forms.



book_flight

‹ General | Forms | Listeners | Input ›

Forms

Form Key 1: ask_flight_date

Form Fields 2: flightDate

Form Field

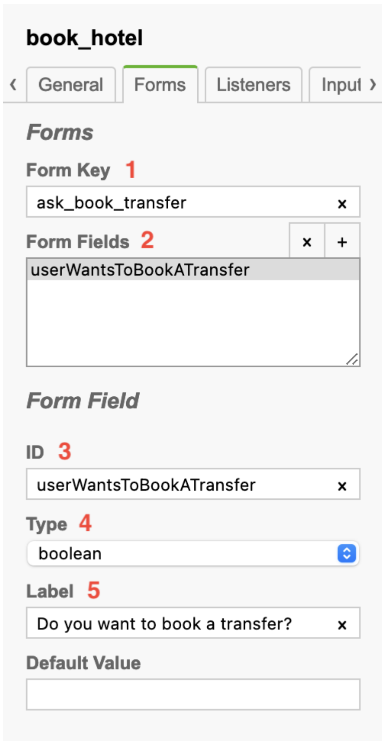
ID 3: flightDate

Type 4: string

Label 5: What is your flight date?

Default Value:

(a) For the "Book flight" task.



book_hotel

‹ General | Forms | Listeners | Input ›

Forms

Form Key 1: ask_book_transfer

Form Fields 2: userWantsToBookATransfer

Form Field

ID 3: userWantsToBookATransfer

Type 4: boolean

Label 5: Do you want to book a transfer?

Default Value:

(b) For the "Book hotel" task.

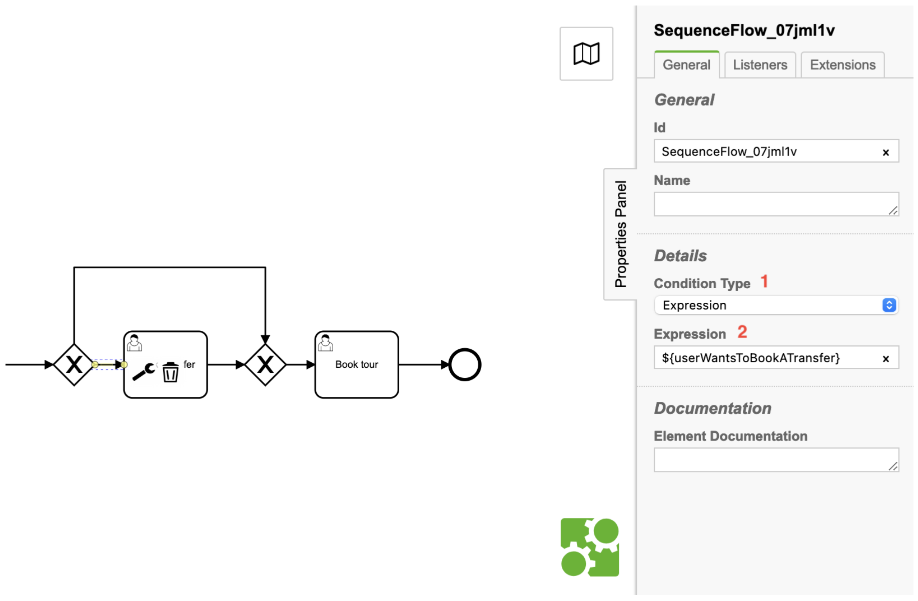
Figure 5.7: Form tab shown in the sidebar of Camunda Modeler.

5.1.5 Configuring gateways

In regards to gateways, this guide has been tested with Exclusive, Inclusive, and Parallel gateways. After choosing the right type for each gateway, if there are any Exclusive or Inclusive gateways in the process, i.e., gateways whose outgoing flows are supposed to have conditions, you need to configure these conditions. To do that, click on each outgoing sequence flow, and then, as demonstrated in Fig. 5.8, under Condition Type (Fig. 5.8.1), you should choose Expression. Inside the Expression field (Fig. 5.8.2), type in something like `${variable}` or `${!variable}`. The $ sign and curly braces are necessary. Unless you are configuring variables outside of Camunda Modeler, such as in a Java class, you need to use variables that are present in a previously created form. It is also worth noting that you do not need to give a semantic Id to each sequence flow; you can leave them as they are.

5.1.6 Setting up start and end events

Finally, you need to make sure that a start event is connected to the first task of the process and the last task of the process is connected to an end event so



SequenceFlow_07jml1v

General

Id: SequenceFlow_07jml1v

Name:

Details

Condition Type 1: Expression

Expression 2: ${userWantsToBookATransfer}

Documentation

Element Documentation:

Figure 5.8: Sequence Flow details shown in the sidebar of Camunda Modeler.

that the process instance can end right after the user completes the last task.

5.2 Building the default Rasa bot

From this section on, it is assumed that your BPMN file already satisfies the following conditions:

- the process has a start and end event
- the process has an ID (process definition key)
- each task has an Id (task definition key) and a name
- each task that requires information has a form, with a form key and form fields, each form field with an ID, type, and label
- all tasks that require user action are configured as User tasks
- all service tasks have an implementation configured
- if there are any Exclusive or Inclusive gateways, each sequence flow coming out of those gateways has an expression that evaluates the appropriate variable(s)

The guide will now go over the steps for building the base of a Process-Aware bot. Every bot built with this guide will need to have the following intents and actions, regardless of the process scenario. These pieces are necessary for the bot to carry out a more natural conversation and better understand the user's input.

5.2.1 Planning default intents

For this guide, the default bot needs to have at least three intents:

- `greet`
- `affirm`
- `deny`

The first one will be responsible for welcoming the user when they say "hello" or any other greeting. The other two will be responsible for understanding when the user says "yes" or "no" and mapping their response to the `affirm` or `deny` intents. These last two intents will be especially useful when requiring boolean information.

5.2.2 Planning default actions

Unlike the `affirm` and `deny` intents, which are merely for the bot's understanding, the `greet` intent needs to be followed by a bot's response. Since responses are sent by actions, you need to create a simple utter action called `utter_greet`. Then, another action that needs to be written in this initial moment is the `utter_default_fallback` action, which will be the bot's default response when it does not understand what the user means.

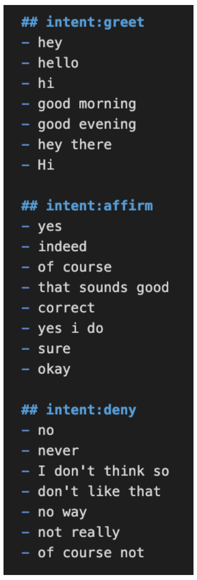
Therefore, for this initial part of the "Trip Planning" bot, you should have two actions:

- `utter_default_fallback`
- `utter_greet`

To implement the three intents and two actions mentioned, you will need to edit three files: `nlu.md`, `domain.yml` and `stories.md`. The initial editing of these files will be covered in subsections 5.2.3, 5.2.4 and 5.2.5.

5.2.3 Adding default intents to the `nlu.md` file

The `nlu.md` file will be responsible for providing examples of phrases that should be associated with each intent. As shown in Fig. 5.9, each intent should be followed by phrases that represent it. For this step, the more phrases you can come up with that could portray an intent, the better.

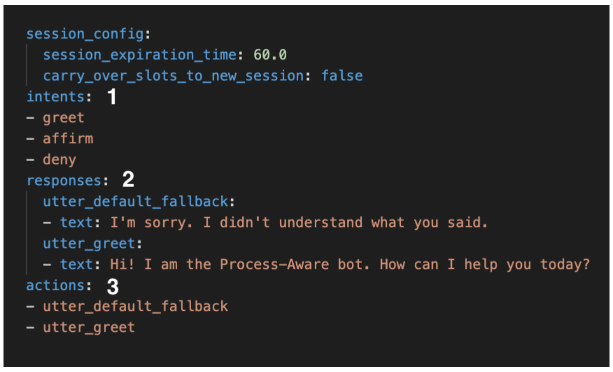


```
## intent:greet
- hey
- hello
- hi
- good morning
- good evening
- hey there
- Hi

## intent:affirm
- yes
- indeed
- of course
- that sounds good
- correct
- yes i do
- sure
- okay

## intent:deny
- no
- never
- I don't think so
- don't like that
- no way
- not really
- of course not
```

Figure 5.9: Fragment of the `nlu.md` file from the "Trip Planning" bot with only default intents.

5.2.4 Adding default intents and actions to the `domain.yml` file

In the `domain.yml` file, as shown in Fig. 5.10, you have to write the intents listed above under the `intents` section of the file (Fig. 5.10.1). Meanwhile, the actions have to be listed under `actions` (Fig. 5.10.3). Since these first actions are going to be used only for sending a response to the user and nothing else, they can be simple utter actions, as explained in Section 2.2.2. Their names need to begin with the `utter_` prefix, and their respective templates, i.e., the messages that will be sent by the bot during these actions, need to be added under `responses` (Fig. 5.10.2). The domain file containing these default intents and actions is shown in Fig. 5.10.



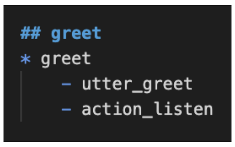
```
session_config:
  session_expiration_time: 60.0
  carry_over_slots_to_new_session: false
intents:
- greet
- affirm
- deny
responses:
  utter_default_fallback:
  - text: I'm sorry. I didn't understand what you said.
  utter_greet:
  - text: Hi! I am the Process-Aware bot. How can I help you today?
actions:
- utter_default_fallback
- utter_greet
```

Figure 5.10: Fragment of the `domain.yml` file from the "Trip Planning" bot with only default intents and actions.

5.2.5 Configuring default stories in the `stories.md` file

The `stories.md` file, represented in Fig.5.11, will be responsible for providing conversation paths, i.e., which actions should follow which intents. In this initial step, you only need to add a story called `greet`. Here, in the stories file, as previously explained, the story name is preceded by double hash signs (`##`). The bot will not use these names; they are only used to improve code understanding. Then, under the story name, each intent is preceded by an asterisk (`*`), and each action is preceded by a hyphen (`-`). So, the picture shows that the `greet` intent should be followed by the `utter_greet` action and then `action_listen`, which means that the bot will stop and wait for new messages coming from the user.

The `affirm` and `deny` intents do not appear in this `stories.md` file because they are only used for mapping values inside custom actions, as will be shown later in the guide. The `utter_default_fallback` action will also be left out of this file because, as the name indicates, it is used by default whenever the bot cannot determine the next step.



```
## greet
* greet
    - utter_greet
    - action_listen
```

Figure 5.11: Fragment of the initial `stories.md` file from the "Trip Planning" bot.

5.3 Adding process-specific pieces to the Rasa bot

Now that default intents and actions have been covered, the guide can now go over more process-specific segments. Some of the intents and actions presented from here on out will vary greatly from process to process. Nevertheless, there will still be some intents (such as `now_what`) and actions (such as `start_process` and `whats_next`) that will need to be added to every bot, just like the default intents and actions.

There is a reason why these seemingly general intents and actions are being classified as "process-specific" instead of "default". While default pieces are completely unrelated to the process itself, the intents and actions covered by this section are considerably more connected to the business process domain. They will sometimes even use process variables, whereas the default ones will never use process-specific information.

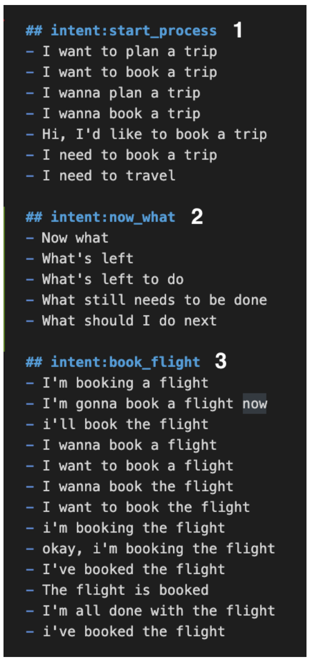
To start adding these process-specific pieces to the bot, you need to examine the User tasks and events that the process contains. Going back to the process model, Fig. 5.1 shows that the "Trip Planning" process has four User tasks:

- Book flight
- Book hotel
- Book transfer
- Book tour

The "Book flight" task comes after a *start event*, which means that it is the first task in the process. Meanwhile, the "Book tour" task is followed by an *end event*, which means it is the last. The guide will now demonstrate how these tasks and events are translated into process-specific intents and actions.

5.3.1 Planning process-specific intents and adding them to the `nlu.md` file

For the start event, the intent that should be created is called `start_process`. It will be responsible for understanding when the user says a sentence like "I want to plan a trip", which should, in turn, start the "Trip Planning" process. This intent needs to be added to any Process-Aware Bot created using this guide, not just to this "Trip Planning" bot. However, the phrases you will want to represent this intent are more context-specific, so they will greatly vary depending on the process scenario. For this example, the chosen sentences for this `start_process` intent are shown in Fig. 5.12.1, which is a screenshot of the updated `nlu.md` file, providing sentence examples for each intent.



```
## intent:start_process  1
- I want to plan a trip
- I want to book a trip
- I wanna plan a trip
- I wanna book a trip
- Hi, I'd like to book a trip
- I need to book a trip
- I need to travel

## intent:now_what  2
- Now what
- What's left
- What's left to do
- What still needs to be done
- What should I do next

## intent:book_flight  3
- I'm booking a flight
- I'm gonna book a flight now
- i'll book the flight
- I wanna book a flight
- I want to book a flight
- I wanna book the flight
- I want to book the flight
- i'm booking the flight
- okay, i'm booking the flight
- I've booked the flight
- The flight is booked
- I'm all done with the flight
- i've booked the flight
```

Figure 5.12: Fragment of the `nlu.md` file from the "Trip Planning" bot showing the `start_process`, `now_what` and `book_flight` intents.

Then, you need an intent for when the user is "lost" in the process or unsure about what tasks are available at that moment. You should call this intent `now_what`, and it will be triggered with phrases shown in Fig. 5.12.2. This intent, just like the default ones, needs to be added to every bot. However, it is not categorized as a default intent because it is more connected to the business process domain.

Each task should have its respective intent, for when the user indicates that they want to start it. For example, "I want to book a flight" for starting the flight task, and "I want to book a tour" for starting the tour task. In Fig. 5.12.3, it is only being shown one task intent with its respective sentences, which is `book_flight`, but there are three more pieces not shown in this picture that are analogous to this last one, one for each of the other tasks (`book_hotel`, `book_transfer`, `book_tour`).

It is relevant to highlight that, even when a user says they want to start a specific task, it does not mean that the task will be started. That is because it may not be available at that moment, and the bot needs to block its execution until it is available. The guide will demonstrate how this is done when describing the creation of task actions (Section 5.3.6).

So far, the NLU component should have the following intents:

- `start_process`, to represent the user asking to start the "Trip Planning" process
- `now_what`, to represent the user asking what is left on their list after they have completed some tasks
- a set of four intents representing the user saying the task they want to start: `book_flight`, `book_hotel`, `book_transfer` and `book_tour`.

5.3.2 Planning process-specific actions

For each process-specific intent, you also have to create their respective response actions, i.e., what the bot should say or do after each of these intents is recognized. Actions can be either "utter actions", such as the `utter_greet` action that has already been covered, or "custom actions", which are fragments of code that can perform more complex tasks, such as API requests, conditionals, etc. Actions are utilized by Rasa Core, which is the bot's back end, and the creation of these actions will be described in what follows.

For a PACA, the bot should use custom actions to collect information from the user and make calls to Camunda API. The API, in turn, will help the bot start and navigate the process. At the outset, two actions have to be added to every Process-Aware Bot created using the guide. They are:

- `start_process`, a custom action that will make a call to the Camunda API to start an instance of the deployed process.
- `whats_next`, a custom action that puts together a message containing all the currently available tasks, or, if the process is finished, informs the user they

are done.

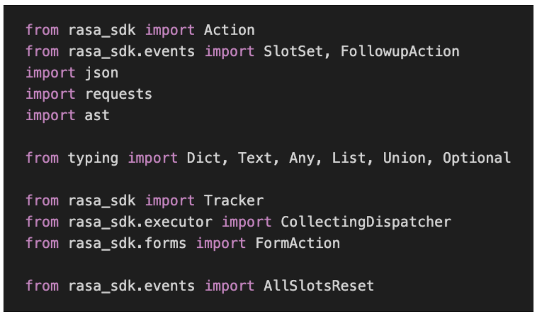
Then, there are actions that heavily depend on the specific process that is being examined. For this "Trip Planning" scenario, the following actions should be created:

- a set of four custom actions, with the same name as their corresponding intents (`book_flight`, `book_hotel`, `book_transfer` and `book_tour`), each meant to be executed right after its respective intent – these actions will depend on the specific User tasks in the process.
- two form actions (`ask_flight_date` and `ask_book_transfer`) that will ask the user for information and store it for future use – the number of form actions will depend on how many tasks require information.

First, this guide will cover the essential parts of each action in the `actions.py` file, and then it will demonstrate how to add them to the `domain.yml` and `stories.md` files.

5.3.3 Initial configuration of the `actions.py` file

In `actions.py`, you need to start by importing required Python packages and setting global variables. Fig. 5.13 illustrates these imports.



```python
from rasa_sdk import Action
from rasa_sdk.events import SlotSet, FollowupAction
import json
import requests
import ast

from typing import Dict, Text, Any, List, Union, Optional

from rasa_sdk import Tracker
from rasa_sdk.executor import CollectingDispatcher
from rasa_sdk.forms import FormAction

from rasa_sdk.events import AllSlotsReset
```

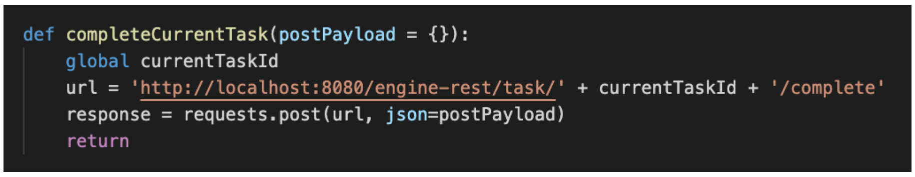
Figure 5.13: Fragment of the `actions.py` file from the "Trip Planning" bot showing the imports.

In the Process-Aware bot, there should be some global variables, and they will be set in different places of the code:

- `processKey` – will be set in the global scope and identifies the process definition independently of its running instances;

- taskGetUrl – will be set in the start_process action, and it is the endpoint used to retrieve all the available tasks at a given moment;
- processInstanceGetUrl – will be set in the start_process action, and it is the URL that will be used to see if the process instance is still running or is already finished;
- processInstanceId – will be set in the start_process action and stores the Id of the process instance after it is started;
- currentTaskId – will be set in each task action right after the user starts that task, and it will store the Id that Camunda attributes to the task after it has begun

In this file, there also needs to be an auxiliary function called completeCurrentTask (Fig. 5.14). This function will send an API request to complete the current task. The function knows what the current task is by accessing the global variable currentTaskId, which will be set after the user informs they are executing or have executed a specific task. This will be further explained in the "Creating task actions" step. If any process variables need to be updated while completing a task, this function will send the updated variable value in the POST payload. Otherwise, it will send an empty object, meaning there are no variables to be updated.



```python
def completeCurrentTask(postPayload = {}):
    global currentTaskId
    url = 'http://localhost:8080/engine-rest/task/' + currentTaskId + '/complete'
    response = requests.post(url, json=postPayload)
    return
```

Figure 5.14: Fragment of the actions.py file from the "Trip Planning" bot showing the completeCurrentTask auxiliary function.

5.3.4 Creating a start_process action in the actions.py file

For the start_process action, Fig. 5.15 will illustrate each step that needs to be done.

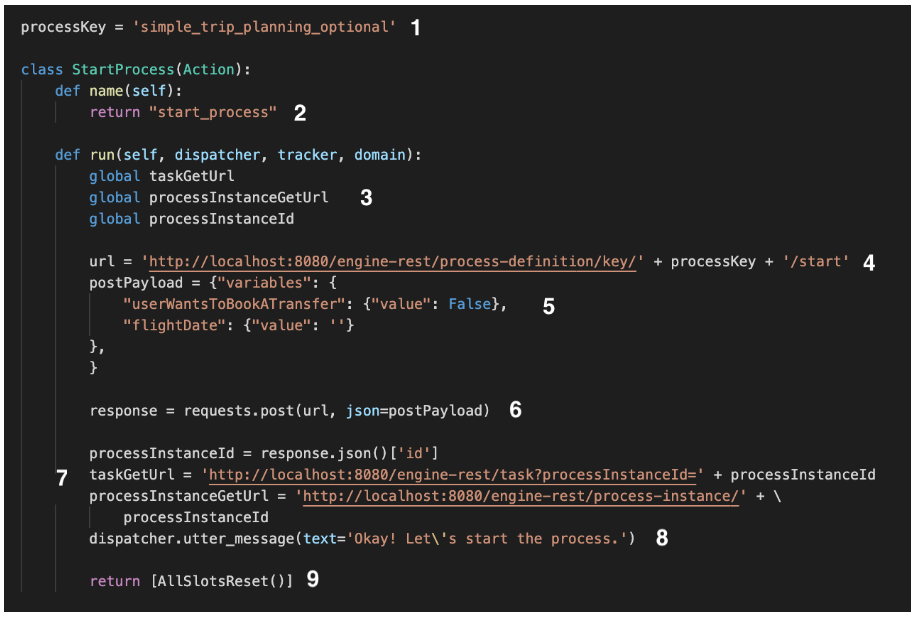
In Fig. 5.15.1, located right under the imports, the global variable processKey should be defined. It does not need the global keyword because it is already located in the global scope and will only be read and not modified. This variable will need to carry the value of the process definition key that was set in Camunda, in Fig. ??.

Then, a `StartProcess` class is created. It inherits from the `Action` class. This class should have a method called `name` (Fig. 5.15.2) that returns the name of this action. It is through this name that this action is going to be referenced in the `domain.yml` and `stories.md` files. Meanwhile, the `run` method is the one that will truly implement what the action will do. In this `start_process` action, three global variables need to be set, as they will be later read by other actions. They are first declared as global in Fig. 5.15.3, but will only be defined later in Fig. 5.15.6. They can only be defined later because they will need to know the process instance ID, which is only obtained after the process has been started. To better explain this, it is useful to follow the process life cycle in Camunda Engine: With the process key set in Fig. 5.15.1, the bot can start a process instance of the deployed BPMN. Each instance will have a specific ID, and this ID will be stored throughout the entire bot execution, so that it is always referring to the same instance since the beginning and not any other instance that might be running.

To start the process, first, you need to put together the URL that the bot needs to call to start the process (Fig. 5.15.4). Before making a POST request to it, you need to create the POST request payload (Fig. 5.15.5). This payload will initialize every process variable you will need during process execution; otherwise, it will not work when you later try to set values to these variables. It is good to keep in mind that each process variable – in this case, `userWantsToBookATransfer` and `flightDate` – will also need to be added as Rasa slots. However, this addition will be covered later on.

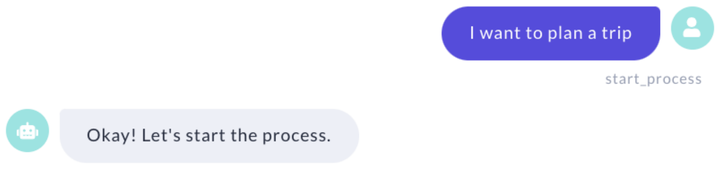
With the URL and payload ready, you make the request using the `requests` package (Fig. 5.15.6). In the JSON object that comes in the response body, there will be a key called `id`, whose value is the process instance ID, and this will be used to set the global variable `processInstanceId`. Both the `taskGetUrl` and `processInstanceGetUrl` global variables will use `processInstanceId` in them (Fig. 5.15.7). The `taskGetUrl` variable will later be used in other actions to see if a specific task is available or to get all the available tasks, while `processInstanceGetUrl` will be used to check if the instance is still running or has already been finished.

Subsequently, in Fig. 5.15.8, the bot needs to utter a simple message to the user, saying the process is being started, and finally, in Fig. 5.15.9, the action ends with the reset of all slots. This reset is necessary for when the user wants to start a new process instance in the middle of the execution of another instance, so that previous Rasa slots are not carried to the new execution. Fig. 5.16 shows an excerpt of the conversation from when a user asks to start the process.



```
processKey = 'simple_trip_planning_optional'  1

class StartProcess(Action):
    def name(self):
        return "start_process"  2

    def run(self, dispatcher, tracker, domain):
        global taskGetUrl
        global processInstanceGetUrl    3
        global processInstanceId

        url = 'http://localhost:8080/engine-rest/process-definition/key/' + processKey + '/start'  4
        postPayload = {"variables": {
            "userWantsToBookATransfer": {"value": False},    5
            "flightDate": {"value": ''}
        },
        }

        response = requests.post(url, json=postPayload)  6

        processInstanceId = response.json()['id']
  7     taskGetUrl = 'http://localhost:8080/engine-rest/task?processInstanceId=' + processInstanceId
        processInstanceGetUrl = 'http://localhost:8080/engine-rest/process-instance/' + \
            processInstanceId
        dispatcher.utter_message(text='Okay! Let\'s start the process.')  8

        return [AllSlotsReset()]  9
```

Figure 5.15: Fragment of the `actions.py` file from the "Trip Planning" bot showing the `start_process` action and global variables.



I want to plan a trip

start_process

Okay! Let's start the process.

Figure 5.16: Fragment of a conversation as a user asks to plan a trip.

5.3.5 Creating a `whats_next` action in the `actions.py` file

After finishing the `start_process` action, the next step is creating the `whats_next` action, which will put together a list with the available tasks and send it to the user.

This action will run in three scenarios:

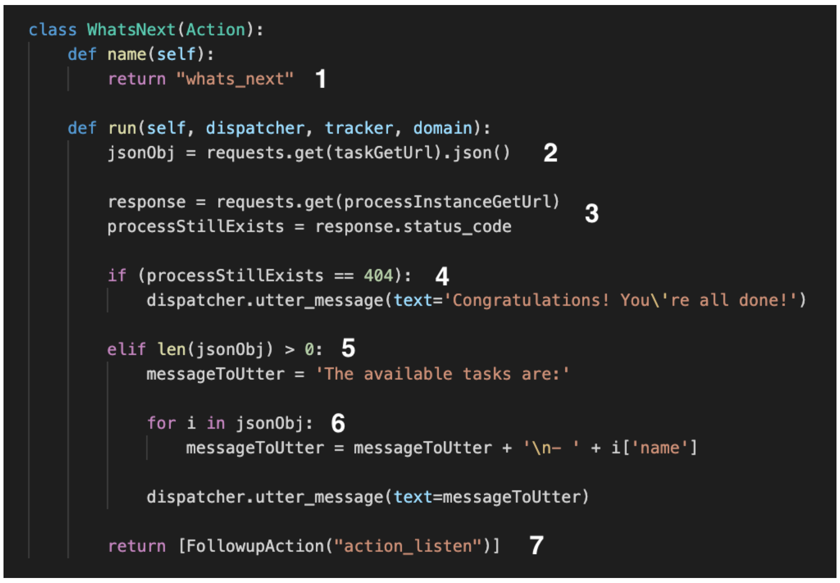
- after the `start_process` action
- after each task is finished
- whenever the bot identifies the `now_what` intent coming from the user (The sentences that trigger this intent can be seen in Fig. 5.12)

The code for the `whats_next` action can be seen in Fig. 5.17. The image is numbered for a clearer understanding of each part of the code.

In Fig. 5.17.1, just like in the `start_process` action, there should be the action's name. Then, in Fig. 5.17.2, you need to make a GET request to the `taskGetUrl` variable to retrieve all the available tasks at that moment. Another GET request should be made (Fig. 5.17.3), but now to the `processInstanceGetUrl` endpoint, which will provide information about the process instance – more specifically, if it is still running.

If this GET request responds with status code 404, then that means the process instance has already been finished (Fig. 5.17.4). In that case, the bot shall inform the user so, with the message "Congratulations! You're all done!". This type of exchange can be seen in Fig. 5.6b.

Conversely, if the response status code is not 404 and the JSON object in the response body has at least one item (Fig. 5.17.5), this means that the process is still running and has at least one available task. In that case, the bot should utter a message indicating it will send a list of available tasks, and in Fig. 5.17.6, it should iterate over these tasks, listing them one by one. The resulting list will contain the "names" of each task, which have been defined in Camunda, as shown in Fig. 5.5. That is why it is crucial that all task names are absolutely clear and understandable by the user. Finally, in Fig. 5.17.7, the action ends after returning `action_listen` as a follow-up action. This means the bot will stop and wait for the user to say which task they want to execute next. An example of this conversation path can be seen in Fig. 5.6a.



```
class WhatsNext(Action):
    def name(self):
        return "whats_next"  1

    def run(self, dispatcher, tracker, domain):
        jsonObj = requests.get(taskGetUrl).json()   2

        response = requests.get(processInstanceGetUrl)
        processStillExists = response.status_code      3

        if (processStillExists == 404):  4
            dispatcher.utter_message(text='Congratulations! You\'re all done!')

        elif len(jsonObj) > 0:  5
            messageToUtter = 'The available tasks are:'

            for i in jsonObj:  6
                messageToUtter = messageToUtter + '\n- ' + i['name']

            dispatcher.utter_message(text=messageToUtter)

        return [FollowupAction("action_listen")]  7
```

Figure 5.17: Fragment of the `actions.py` file from the "Trip Planning" bot showing the `whats_next` action.

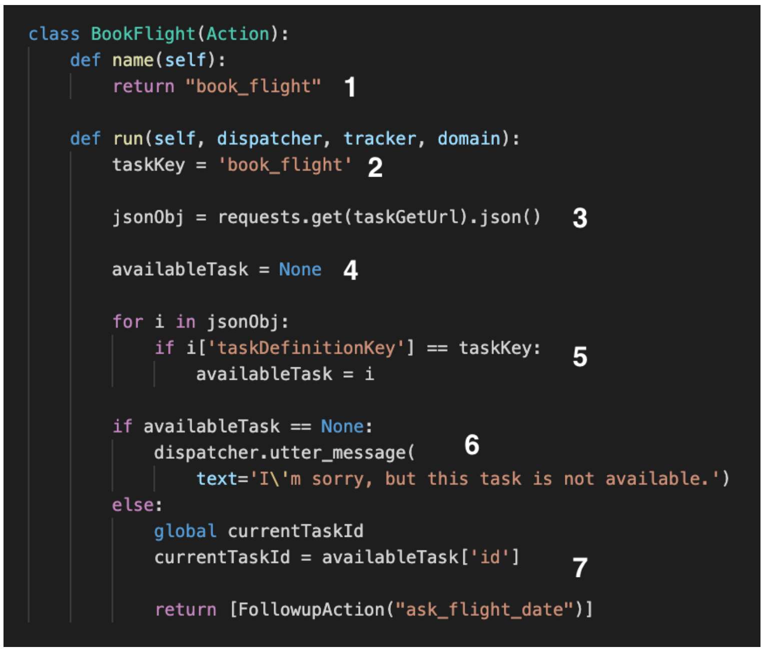
After the list of tasks has been presented and the bot is listening, the user has to pick one task to execute. For each task in the process, there will need to be an action to determine what will happen after the task's execution. Remember that, in the "Trip Planning" process, there are a few tasks that require information, such as "Book flight", and others that do not, such as "Book tour". The creation of both of these types of task actions will be covered in what follows.

5.3.6 Creating task actions in the `actions.py` file

Let us start with the "Book flight" task because it is the first one in the "Trip Planning" process. This is a task that requires information, so it will originate two custom actions: one *task action* (Fig. 5.18), which will be covered in this step, and also a *form action*, for gathering the information and completing the task itself (Fig. 5.20). The latter will be covered in Section 5.3.7.

As shown in Fig. 5.18.1, the name of this first task action will be `book_flight`. The action's name must be the same as the intent's name, and all of them equal to the task definition key that was defined in Camunda in Fig. 5.4b. The task key also needs to be added as a local variable called `taskKey` inside the `run` method, as shown in Fig. 5.18.2, because it will be used for checking if the task is present in the available tasks object.

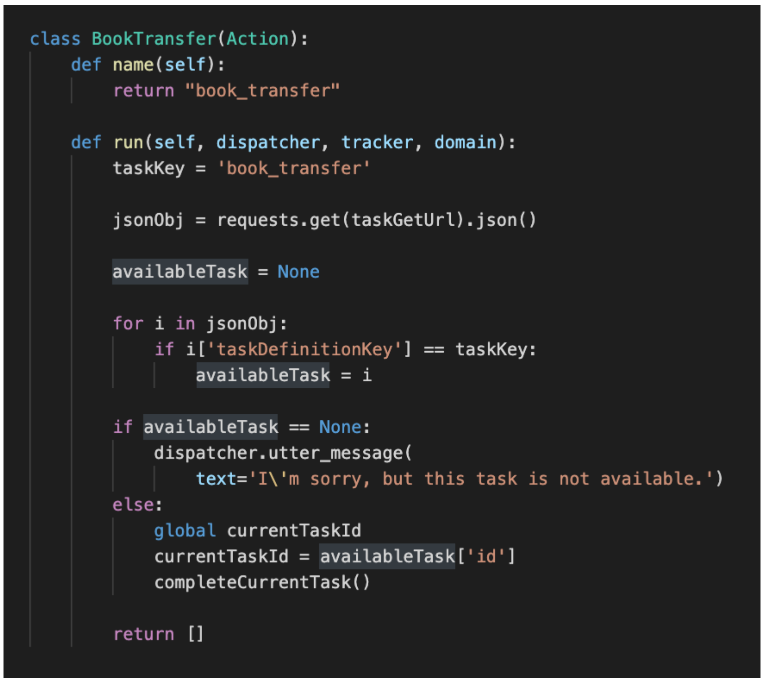
To make this check, first, the bot needs to make a GET request to retrieve all of the available tasks (Fig. 5.18.3) and initialize an `availableTask` variable as `None` (Fig. 5.18.4). The bot will then go over each available task in the retrieved object, and if one of them is the sought task, the bot will store it in the `availableTask` variable (Fig. 5.18.5). Otherwise, the `availableTask` variable will remain as `None` and that will mean the requested task is not available, and the bot should let the user know (Fig. 5.18.6). Finally, if the task was, in fact, available, the bot will store its ID in the global variable `currentTaskId`, which will later be used for completing the task (Fig. 5.14). Before this "Book flight" task can be completed, though, the bot needs to gather some information. In this case, it needs to ask the user what their flight date is. To do that, this `book_flight` action should be followed by a *form action*, which can be called `ask_flight_date` (Fig. 5.18.7).



```
class BookFlight(Action):
    def name(self):
        return "book_flight"   1

    def run(self, dispatcher, tracker, domain):
        taskKey = 'book_flight'  2

        jsonObj = requests.get(taskGetUrl).json()    3

        availableTask = None   4

        for i in jsonObj:
            if i['taskDefinitionKey'] == taskKey:    5
                availableTask = i

        if availableTask == None:
            dispatcher.utter_message(     6
                text='I\'m sorry, but this task is not available.')
        else:
            global currentTaskId
            currentTaskId = availableTask['id']      7

            return [FollowupAction("ask_flight_date")]
```

Figure 5.18: Fragment of the `actions.py` file from the "Trip Planning" bot showing the `book_flight` action.

If this task did not require any information, the task action would have to call the `completeCurrentTask` function (Fig. 5.14) before the method's "return" statement. When calling this function, it would have to pass an empty object as argument since no variables need to be updated. Also, the method would return an empty array, which would mean that the bot could follow the regular story path instead of being redirected to another action. This case is exemplified by the "Book transfer" action, shown in Fig. 5.19, which does not require any information.



```python
class BookTransfer(Action):
    def name(self):
        return "book_transfer"

    def run(self, dispatcher, tracker, domain):
        taskKey = 'book_transfer'

        jsonObj = requests.get(taskGetUrl).json()

        availableTask = None

        for i in jsonObj:
            if i['taskDefinitionKey'] == taskKey:
                availableTask = i

        if availableTask == None:
            dispatcher.utter_message(
                text='I\'m sorry, but this task is not available.')
        else:
            global currentTaskId
            currentTaskId = availableTask['id']
            completeCurrentTask()

        return []
```

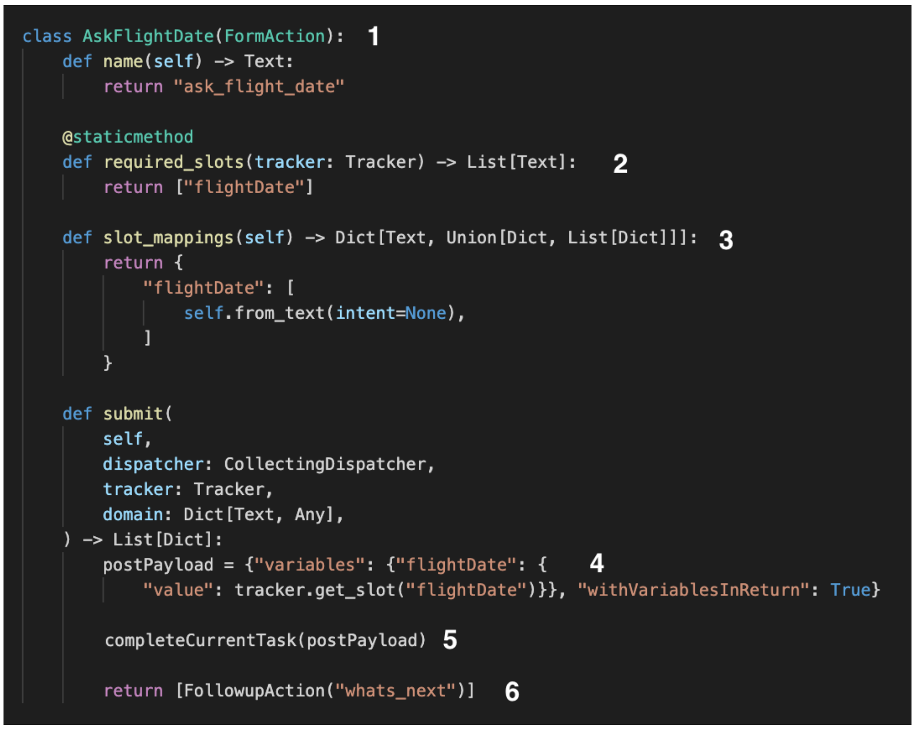
Figure 5.19: Fragment of the `actions.py` file from the "Trip Planning" bot showing the "Book transfer" action.

5.3.7 Creating form actions in the `actions.py` file

As previously explained, since the "Book flight" action requires information, at the end of the `book_flight` task action, the story path needs to be redirected to its respective form action. Since the `ask_flight_date` action (Fig. 5.20) will be a *form action*, its class needs to inherit from the `FormAction` class (Fig. 5.20.1). Every form action also needs three methods, besides "name". They are: `requred_slots` (Fig. 5.20.2), `slot_mappings` (Fig. 5.20.3) and `submit` (Fig. 5.20.4, Fig. 5.20.5 and Fig. 5.20.6).

The `required_slots` method (Fig. 5.20.2) is a static method and should return an array with all of the variables required by this task. This specific "Book flight" task only requires one variable, as shown in Fig. 5.7a, and that is `flightDate`. Thus, the `flightDate` slot should be added to the `required_slots` method.

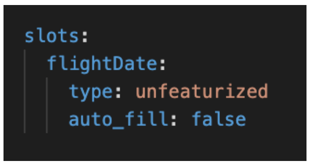
Each slot also needs to be added to the `domain.yml` file in the `slots` section.



```python
class AskFlightDate(FormAction):  1
    def name(self) -> Text:
        return "ask_flight_date"

    @staticmethod
    def required_slots(tracker: Tracker) -> List[Text]:  2
        return ["flightDate"]

    def slot_mappings(self) -> Dict[Text, Union[Dict, List[Dict]]]:  3
        return {
            "flightDate": [
                self.from_text(intent=None),
            ]
        }

    def submit(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any],
    ) -> List[Dict]:
        postPayload = {"variables": {"flightDate": {    4
            "value": tracker.get_slot("flightDate")}}, "withVariablesInReturn": True}

        completeCurrentTask(postPayload)  5

        return [FollowupAction("whats_next")]   6
```

Figure 5.20: Fragment of the `actions.py` file from the "Trip Planning" bot showing the `ask_flight_date` action.

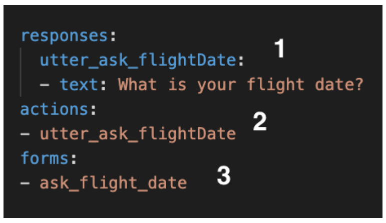
As seen in Fig. 5.21, you should add the `flightDate` slot under `slots`, setting its type to `unfeaturized`, which means that its value should not affect the dialogue flow (it is an independent variable that does not influence the conversation). You should also set `auto_fill` to false because these slots should not be automatically mapped from entities – they are going to be filled by form actions.



```yaml
slots:
  flightDate:
    type: unfeaturized
    auto_fill: false
```

Figure 5.21: Fragment of the `domain.yml` file from the "Trip Planning" bot showing the slot section.

Rasa has a brilliant way of knowing how to ask the user for these slots and fill them with what it receives. In the `domain.yml` file, you need to add two more elements for each variable and one more for each form. Fig. 5.22 shows the necessary information for each variable and each form. Initially, all of the forms have to be listed under `forms` (5.22.3). So far, there is only the `ask_flight_date` form. Then, for each required slot, Rasa needs the developer to configure a simple "utter action" that will utter the question asking the user for that particular variable. The name of this "utter action" has to follow the pattern `utter_ask_{variable}`, and it should be listed under `actions` (5.22.2), while the message that represents it should be under `responses` (5.22.1). In this case, since the variable is called `flightDate`, the action will be named `utter_ask_flightDate` and the bot's response for this action should be "What is your flight date?" For consistency with the BPMN model, this question should be the one set in this variable's "Label" field in the Camunda form for this task (Fig. 5.7a).



```
responses:
  utter_ask_flightDate:
  - text: What is your flight date?
actions:
- utter_ask_flightDate
forms:
- ask_flight_date
```

Figure 5.22: Fragment of the `domain.yml` file from the "Trip Planning" bot showing the necessary information for a form.

Then, going back to Fig. 5.20, for each slot, you need to describe in the `slot_mappings` method (Fig. 5.20.3) how the bot should map the response. That is, if it is a string, should it be converted to a number? If it is a "yes" or "no" response, should it be converted to boolean? In this case, since a flight date can be anything, the bot should store it as regular text without trying to map it to any intent (that is why `intent=None`).

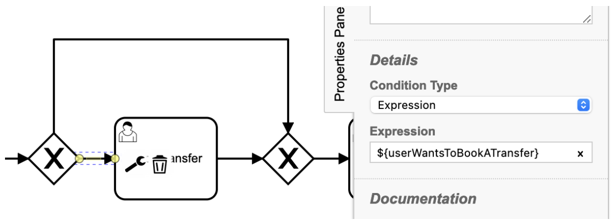
Finally, there is the `submit` method, where the variables will be sent to Camunda and the task will be completed. Fig. 5.20.4 shows the POST payload, which should contain each variable's value. The `tracker.get_slot("flightDate")` command is used to get the value of the `flightDate` slot in Rasa.

In Fig. 5.20.5, the `completeCurrentTask` auxiliary function (Fig. 5.14) is called, and it will effectively send the POST request to complete the current task. Since the `ask_flight_date` action needs to change the value of a process variable, the `completeCurrentTask` function will send the updated variable value in the POST payload.

Finally, in Fig. 5.20.6, the `ask_flight_date` action is finished by calling the `whats_next` action, which will send to the user the next available tasks.

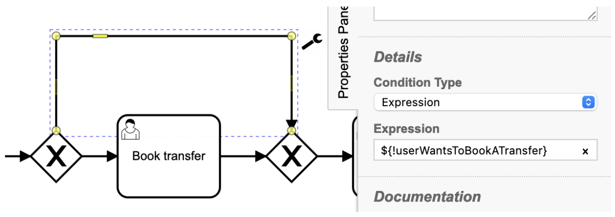
This concludes the creation of the "Book flight" form action, which was necessary because the “Book flight” task requires information. The next task, “Book hotel”, is also one that requires information. This guide will not go deeply into its code since it is quite analogous to the “Book flight”. The only difference between “Book hotel” and “Book flight” is that in “Book flight”, the requested information is not used in the process itself, while the one requested in “Book hotel” is.

“Book hotel” asks for a variable called `userWantsToBookATransfer`. As previously explained, the “Book transfer” task is optional. That is represented in Fig. 5.1 by an exclusive gateway being used to either go to this task or go around it. The choice of path at this point is decided by a sequence flow condition, which will evaluate whether the Camunda process variable `userWantsToBookATransfer`, requested at the end of the “Book hotel” task, is `true` (Fig. 5.23) or `false` (Fig. 5.24).



Condition Type: Expression
Expression: ${userWantsToBookATransfer}

Figure 5.23: Gateway sequence flow in Camunda when the `userWantsToBookATransfer` variable is `true`.



Condition Type: Expression
Expression: ${!userWantsToBookATransfer}

Figure 5.24: Gateway sequence flow in Camunda when the `userWantsToBookATransfer` variable is `false`.

After “Book hotel”, there are two more tasks: “Book transfer” and “Book tour”. These tasks do not require any information, and thus, do not have to be followed by any form action. As exemplified by the “Book transfer” task in Fig. 5.19, actions that do not require information only return an empty array, which means the bot can follow the regular story path.

5.3.8 Configuring process-specific stories in the `stories.md` file

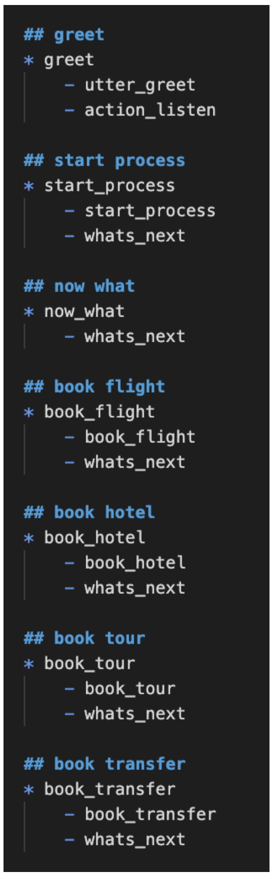
As previously explained, “stories” are mainly used as a reference by the bot to how conversations should go, for example, which actions come after which intents. Unlike the manual integration described in Section 4.1, for this guide, stories will not be responsible for following the process’s sequence flows, as Camunda will be the one responsible for tracking the process. Therefore, the PACA will use stories only to make sure that each action follows its corresponding intent. The actions will then call Camunda API, which will, in turn, monitor the process. Hence, as shown in the finished “stories” file in Fig. 5.25, every intent is followed by an action with the same name, and then comes the `whats_next` action, to inform the user what the next available tasks are. These are the bot’s regular story paths, and they will be followed when there is no follow-up action set up, such as in Fig. 5.19.

Another example of when the bot follows the conventional story path is in the “Book flight” action shown in Fig. 5.18. In Fig. 5.18.7, if the task is available, the `ask_flight_date` action is called, as previously shown. However, that return is conditional. If the task is not available, the execution will stop at Fig. 5.18.6, and the method will return nothing. An empty return statement, just like the return of an empty array, means that the bot can follow the regular story path, calling the `whats_next` action.

5.3.9 Configuring the `domain.yml` file

Finally, after the “stories” file is finished, the `domain.yml` file must be completed. You first need to add the names of all of your recently written actions to the “actions” section. Then, you should check if every form action has its name listed under “forms” and, for every slot requested in a form, if there is a configured response. It is also good to check if all of the intents present in the `nlu.md` file and used in the `stories.md` file are also listed here under “intents”.

The complete `domain.yml` file is shown in Fig. 5.26, including all of the intents, slots, responses, actions, and forms that have been previously covered.



```
## greet
* greet
    - utter_greet
    - action_listen

## start process
* start_process
    - start_process
    - whats_next

## now what
* now_what
    - whats_next

## book flight
* book_flight
    - book_flight
    - whats_next

## book hotel
* book_hotel
    - book_hotel
    - whats_next

## book tour
* book_tour
    - book_tour
    - whats_next

## book transfer
* book_transfer
    - book_transfer
    - whats_next
```

Figure 5.25: The complete `stories.md` file from the "Trip Planning" bot.

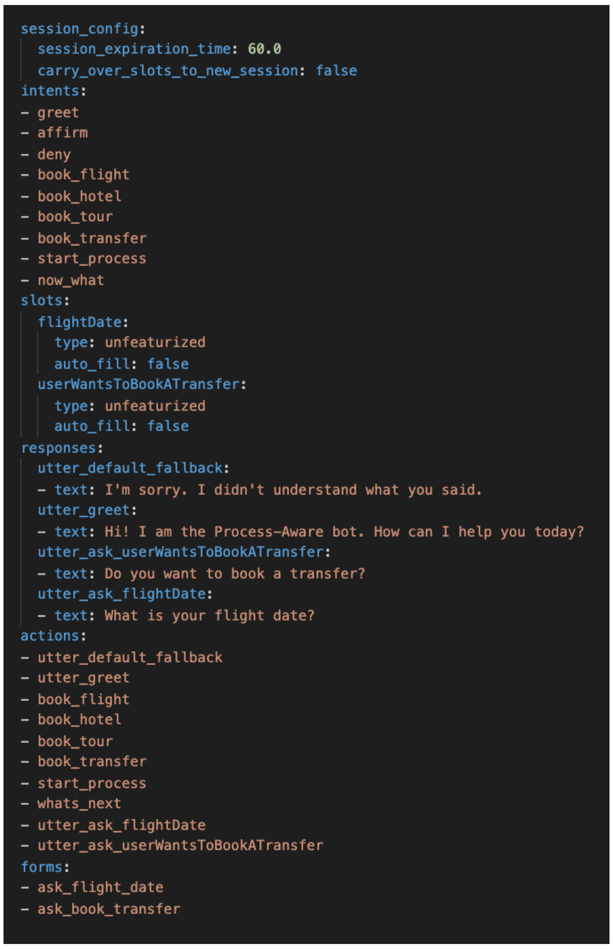

```yaml
session_config:
  session_expiration_time: 60.0
  carry_over_slots_to_new_session: false
intents:
- greet
- affirm
- deny
- book_flight
- book_hotel
- book_tour
- book_transfer
- start_process
- now_what
slots:
  flightDate:
    type: unfeaturized
    auto_fill: false
  userWantsToBookATransfer:
    type: unfeaturized
    auto_fill: false
responses:
  utter_default_fallback:
  - text: I'm sorry. I didn't understand what you said.
  utter_greet:
  - text: Hi! I am the Process-Aware bot. How can I help you today?
  utter_ask_userWantsToBookATransfer:
  - text: Do you want to book a transfer?
  utter_ask_flightDate:
  - text: What is your flight date?
actions:
- utter_default_fallback
- utter_greet
- book_flight
- book_hotel
- book_tour
- book_transfer
- start_process
- whats_next
- utter_ask_flightDate
- utter_ask_userWantsToBookATransfer
forms:
- ask_flight_date
- ask_book_transfer
```

Figure 5.26: The complete `domain.yml` file from the "Trip Planning" bot.

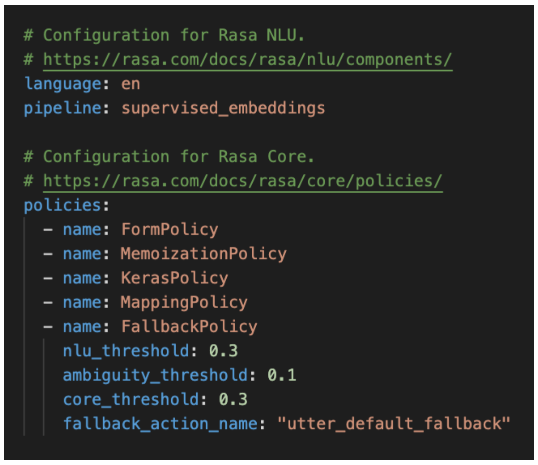
5.3.10 Configuring the `config.yml` file

There is also one more file that is useful to check, which is `config.yml`, shown in Fig. 5.27. This file provides the basic configuration for the bot, both for the NLU and Core components. For the NLU, you should set the language and the NLU pipeline, which contains all of the steps that the NLU will execute to try to understand user input. Here, we are using a pipeline template (`supervised_embeddings`), which is a predefined set of pipeline components. However, you can choose your own components, and the NLU accuracy will vary according to the components used. To learn more about all of the available NLU pipeline components, you can visit the Pipeline section[4] of Rasa docs.

Then, for the Core component, you have to choose which policies are going to be used. The policy set is what Rasa uses to combine all of the available information, such as stories, actions, and the last identified intent, to decide what the bot should do next. The most crucial policies to be listed here are `FormPolicy`, which is necessary in this case because the "Trip Planning" scenario requires Forms, and also `FallbackPolicy`, which will make sure that the `utter_default_fallback` action, included in the `domain.yml` file, is executed when the bot does not understand user input. The other policies listed in this file (`MemoizationPolicy`, `KerasPolicy` and `MappingPolicy`) are used to generate the probabilities for each possible action after an intent, and then choose which action to take. All of the available policies, and also the arguments for the `FallbackPolicy` that were used in Fig. 5.27, are further explained in the Policies section[5] of Rasa documentation. It is wise to check it out and experiment with different policies and arguments if there is enough time. That is because different policy combinations might generate a more stable or unstable bot, and the only way to figure out which combination is the best is by trial and error.

[4] `https://legacy-docs-v1.rasa.com/nlu/choosing-a-pipeline/`

[5] `https://legacy-docs-v1.rasa.com/core/policies/`



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

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  - name: FormPolicy
  - name: MemoizationPolicy
  - name: KerasPolicy
  - name: MappingPolicy
  - name: FallbackPolicy
    nlu_threshold: 0.3
    ambiguity_threshold: 0.1
    core_threshold: 0.3
    fallback_action_name: "utter_default_fallback"
```

Figure 5.27: The `config.yml` file from the "Trip Planning" bot.

5.4 Running Camunda and the Bot

After you have finished preparing the BPMN and all Rasa files, you can finally execute the Process-Aware Bot. For that, you need to make sure you have installed Camunda Modeler (in this guide, version 3.4.0 was used), Rasa 1.8.2, and Rasa SDK 1.8.0.

5.4.1 Launching Camunda Engine

The first step is starting Camunda Engine. On Camunda's website, there are step-by-step instructions on how to run Camunda using Docker[6] or from scratch[7]. After it is up and running, you need to check `http://localhost:8080/`, to see if the Camunda web page is showing. If it asks for a login, the default user and password combination is demo/demo. If you go to Cockpit and have never used Camunda in your current machine, you should probably see no running process instances, and below Process Definitions, there should be a total of zero.

Now, you need to go to Camunda Modeler, and, after loading the BPMN file, click on the last icon in the toolbar, the "Deploy current diagram" icon. In the dialog that opens up, the REST endpoint should be pointing to `http://localhost:8080/engine-rest`, and Authentication should be set to None. The name of the deployed diagram can be whichever is preferred. Finally, click on the Deploy

[6]`https://docs.camunda.org/manual/7.14/installation/docker/`

[7]`https://docs.camunda.org/manual/latest/installation/camunda-bpm-run/`

button. When you go back to Camunda Cockpit, you should now see there is 1 Process Definition, and when you click on the number, you should see your process definition key, with 0 running instances. There are no running instances yet because the first one will start running when the bot is asked to start the process.

5.4.2 Launching Rasa

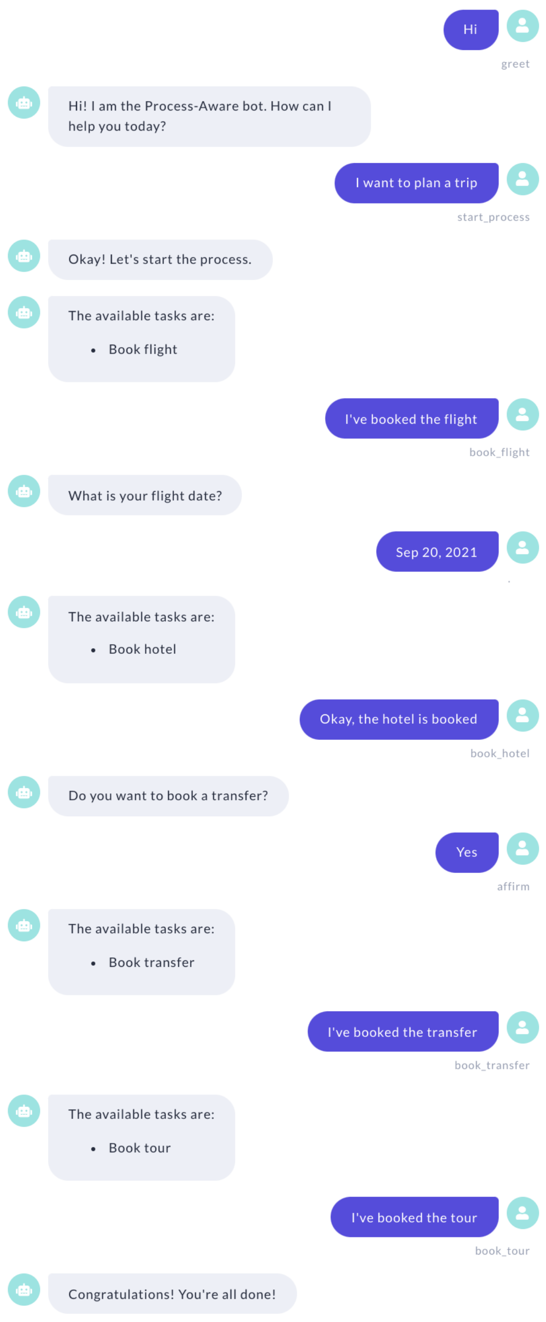
For the Rasa part, you need to open two Terminal windows in the directory where the bot files are located. In the first one, you should run `rasa run actions`, so it will start the action server. In the other one, you should run `rasa train` and then `rasa shell`, which will start the bot in the command shell. It is worth noting that, instead of `rasa shell`, you can use the `rasa x` command to execute the bot in Rasa X – Rasa's default GUI – if this tool is configured. There is also the `rasa interactive` command, which will give you a more in-depth look at why the bot is responding a certain way if it is behaving unexpectedly.

Finally, when the bot is up and running, as soon as you type in a sentence corresponding to the `start_process` intent, it will start a process instance, and you can follow the process execution inside Camunda Cockpit.

This is the end of the PACA Generation Guide. This guide has presented step-by-step instructions for building a Process-Aware bot. A "Trip Planning" scenario has been used for materializing each step; however, the introduced steps can be successfully extrapolated to other scenarios, as will be shown in Chapter 6. A conversation example with the Process-Aware "Trip Planning" bot is shown in Fig. 5.28.

5.5 The conceptual connection between Business Processes and CAs

Ultimately, the connection between the business process and CA domains is represented in Fig. 5.29. Almost all of the concepts listed in Chapter 4 are portrayed here. On the left, we have business process concepts, including Camunda-specific ones, and on the right, we have Rasa concepts, some of them split into more specific blocks.



Hi

greet

Hi! I am the Process-Aware bot. How can I help you today?

I want to plan a trip

start_process

Okay! Let's start the process.

The available tasks are:

- Book flight

I've booked the flight

book_flight

What is your flight date?

Sep 20, 2021

The available tasks are:

- Book hotel

Okay, the hotel is booked

book_hotel

Do you want to book a transfer?

Yes

affirm

The available tasks are:

- Book transfer

I've booked the transfer

book_transfer

The available tasks are:

- Book tour

I've booked the tour

book_tour

Congratulations! You're all done!

Figure 5.28: A complete conversation with the “Trip Planning” bot.

There are four types of actions:

- `start_process` action
- task actions
- `whats_next` action
- form actions

and two types of intent that were portrayed:

- `start_process` intent
- task intents

The four basic intents covered in the beginning of Section 5.2 (`greet`, `affirm`, `deny` and `now_what`) are not shown in the picture because they are not connected to any business process concept. The same goes for the "stories" concept. They only exist to facilitate the conversation flow.

Now, we will examine each connection portrayed in Fig. 5.29. The process's start event will be represented as a `start_process` intent and a `start_process` action (Fig. 5.29.1). After that, each process task will produce a task intent and a task action (Fig. 5.29.2). The end event and all sequence flows are represented in the `whats_next` action (Fig. 5.29.3). This action, as previously explained, will make a call to Camunda to see what the next available tasks are, and if the process has reached the end event, it will utter a message informing the user the process is finished. The concept of "Gateway" is not in the picture because gateways are evaluated exclusively inside Camunda, and therefore, not connected directly to any CA concept. Then, for each User Task Form in Camunda, there should be a form action in Rasa (Fig. 5.29.4) to ask the user for specific information and then send it back to Camunda. Lastly, this requested information is stored in Camunda as a Process Variable, while in Rasa, it is stored as a slot (Fig. 5.29.5).

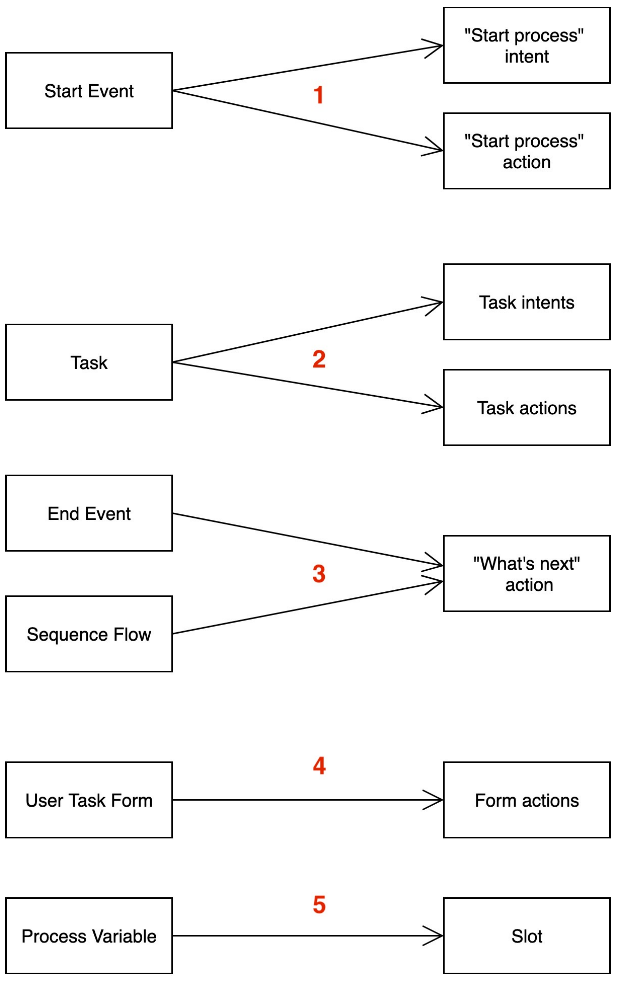


Figure 5.29: Diagram portraying the connection between Process and CA Concepts.

Chapter 6

Process-Aware Conversational Agent: A Use Case

To assess the applicability of our approach, we present an example in which the PACA Generation Guide was used to build a Process-Aware Conversational Agent, testing the connection between business process and CA components. For this use case, we wanted to analyze a looser scenario, i.e., a process in which tasks are not strictly dependent on each other. Doing so, we intended to evaluate if the connection between business process and CA concepts remained even without rigid dependencies between tasks. From this perspective, we chose a "Wedding Planning" process. In this example, we established five main tasks:

- picking a date
- booking a venue
- booking a band
- booking a caterer
- booking a photographer

Whenever we seek to translate a real-life set of activities into a process, there are always many assumptions we have to make to successfully represent it using a widespread notation. We supposed that, once the couple has decided on a specific date, the other four tasks listed above could be executed in any particular order. For example, you do not necessarily need to have previously booked a venue to book a band, a caterer, or a photographer. These four activities are arguably independent of each other.

After drawing a model that considered this flexibility, the resulting process representation is displayed in Fig. 6.1. This diagram shows that the first task to be executed is picking the date for the wedding. After that, the four main tasks can be executed in any particular order, which is represented by the fork and join with Parallel gateways. This fork and join also means that all four tasks must be executed for the process flow to move on.

Then, there is a Service Task called "Finish Booking" right after the gateway following the four booking tasks. In this specific scenario, this task is not executing any code. It is only there to show that it would be possible to create a Java class to effectively make the booking, and it would automatically run after the four booking tasks had been completed. However, in our use case, it simply evaluates an expression that will always return `true`, so this Service task will be passed through without further implications.

Finally, the only way of adequately ending the process execution is by finishing all bookings. If the user were to exit the software mid-execution, the process instance would not be properly terminated, continuing to run in Camunda Engine until it was manually stopped through Camunda Cockpit or the API.

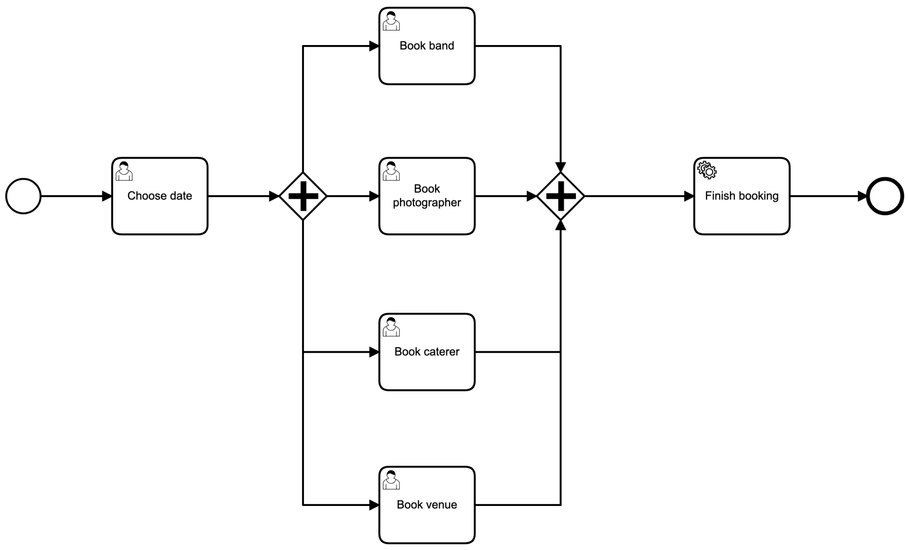


Figure 6.1: BPMN diagram for the "Wedding Planning" process.

First, we had to create the default Rasa bot according to step 5.2 of the guide. Then, we moved on to process-specific intents and actions. For the process-specific part, in accordance to step 5.3.1 of the guide, we planned the following intents, which were then added to the `nlu.md` file:

- `start_process`, for when the user asks to start the "Wedding Planning" process
- `now_what`, for when the user what is left on their list after they have completed some tasks
- `choose_date`, for when the user has picked the date and is ready to inform it to the bot
- a set of four intents for when the user wants to pick the task they want to start: `book_band`, `book_photographer`, `book_caterer` and `book_venue`.

Then, according to step 5.3.2 of the Guide, for each intent, we needed to plan their respective response actions, i.e., what the bot should say or do after each of these intents is recognized. Each action would be used to make calls to the Camunda API so that the bot could start and navigate the process.

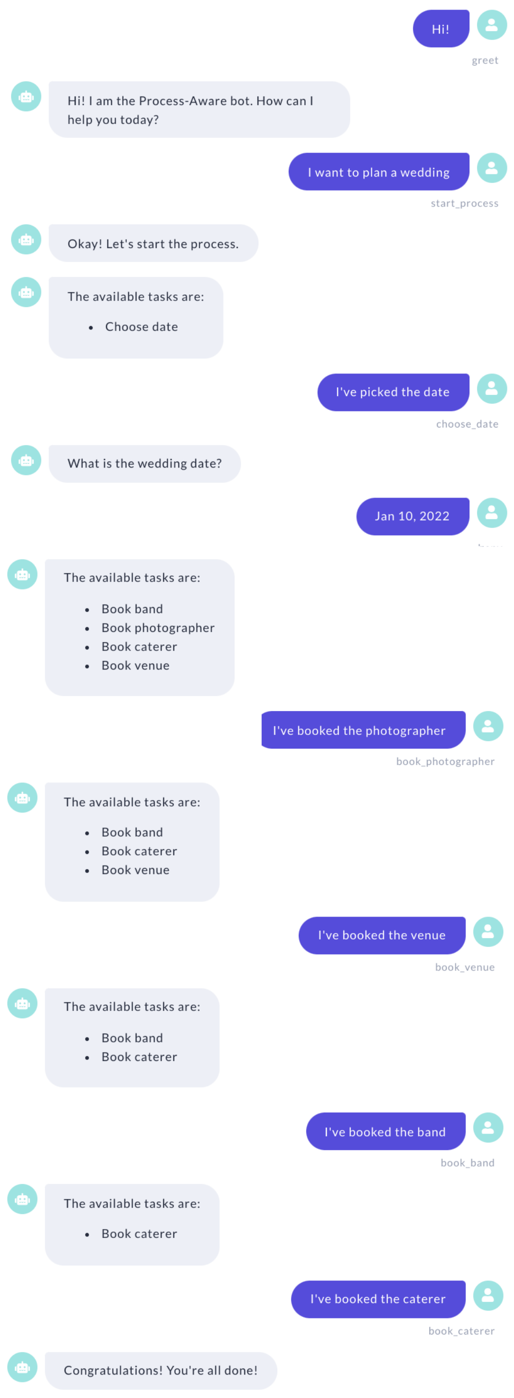
The following actions were planned:

- `start_process`, which will start an instance of the deployed process.
- `whats_next`, which lists the currently available tasks and also informs the user when the process is finished
- five custom actions, with the same name as their corresponding intents (`choose_date`, `book_band`, `book_photographer`, `book_caterer` and `book_venue`), each meant to be executed right after its respective intent.
- one form action (`ask_wedding_date`) that will ask the user for the wedding date so it can be stored and possibly used later.

After the initial configuration of the `actions.py` file (step 5.3.3 of the guide), each of these actions had to be added to the `actions.py` file. The `start_process` action was created according to step 5.3.4, and the `whats_next` action followed step 5.3.5. The five custom task actions were built by following step 5.3.6, and the only form action for this process (`ask_wedding_date`) was created according to step 5.3.7.

The process will be finished once all of the four booking tasks have been executed, and at that moment, the `whats_next` action will inform the user they are done.

Finally, we finished the bot's configuration by completing the `stories.md` file (step 5.3.8), `domain.yml` file (step 5.3.9) and the `config.yml` file (step 5.3.10). A conversation example with the "Wedding Planning" bot is shown in Fig. 6.2.



Hi!

greet

Hi! I am the Process-Aware bot. How can I help you today?

I want to plan a wedding

start_process

Okay! Let's start the process.

The available tasks are:

- Choose date

I've picked the date

choose_date

What is the wedding date?

Jan 10, 2022

The available tasks are:

- Book band
- Book photographer
- Book caterer
- Book venue

I've booked the photographer

book_photographer

The available tasks are:

- Book band
- Book caterer
- Book venue

I've booked the venue

book_venue

The available tasks are:

- Book band
- Book caterer

I've booked the band

book_band

The available tasks are:

- Book caterer

I've booked the caterer

book_caterer

Congratulations! You're all done!

Figure 6.2: A complete conversation with the "Wedding Planning" bot.

Chapter 7

Discussion

In this chapter, we discuss some aspects of the implementation, including differences between the “Trip Planning” and “Wedding Planning” scenarios that we have handled.

Initially, both of these scenarios were attempted to be implemented without the integration with Camunda, using only the tools that Rasa provides. This method of manual integration was briefly explained in Section 4.1. The “Trip Planning” scenario was not particularly challenging to implement using this method because of its stricter design. We found the suggestions were highly accurate, mainly because the user was only allowed to execute one, or at most, two actions at any point in time.

However, this method was shortly perceived as unscalable and difficult to replicate. When we tried to adapt the “Trip Planning” bot for the “Wedding Planning” scenario, it became clear that most of the original code would have to be rewritten. That was because we had had to embed too much context-specific logic in Rasa for a process to work correctly, and the “Trip Planning” scenario was already too entrenched in it. Beyond that, the fact that the “Wedding Planning” process had fewer dependencies between tasks incurred in increased implementation complexity. Since Rasa requires the developer to write stories beforehand to portray all possible conversation paths, it was challenging to establish without Camunda that the four booking tasks could be executed in any order, which led to suggestions being occasionally wrong.

By comparison, when using Camunda, both scenarios were relatively easy to implement and produced good results. Following the guide, the wedding plan bot took less than a day to implement. This time would undoubtedly increase if we were also to create Java classes for Service tasks; however, we did not need to implement any classes for our scenario.

We also found that Exclusive, Inclusive, and Parallel gateways worked naturally well out-of-the-box. Camunda was able to determine the correct paths when coming out of a gateway, and all of the available paths were adequately presented to the user.In our array of tests, we found only one gateway scenario in which the Camunda/Rasa combination did not do its best, which was when there was an Exclusive gateway with more than one outgoing sequence flow evaluated as `true`. In this case, Camunda shows only the first available path according to the XML order instead of showing all of the available paths and letting the user pick the one they want to follow. This strategy makes sense considering that, at an Exclusive gateway, the user can only pick one flow anyway. However, if more than one path is available, it would be nice to see all of the available paths before picking one.

Regarding natural language processing, the bot proved successful in understanding different sentences for the same intent, and even typos without needing to configure them explicitly in the `nlu.md file`. For example, if a user types "tirp" instead of "trip", or "filght" instead of "flight", the bot could still understand the message. This project, however, does not yet support two different intents in the same message, nor an intent combined with an information, so the user will not be understood if they send "I want to book a flight and a hotel", or "I want to book a hotel and do not want a transfer". This is planned to be supported in the future.

Despite not using an evaluation method in this work, a possible evaluation approach for a related future work would be to follow the footsteps of Cranshaw et al. [18], dividing our evaluation into four steps. The first one would be a "wizard-of-oz" for gathering initial data to populate our NLU and story files. Then, a usability study could be conducted to assess the solution's viability and to improve some aspects of it. And finally, two phases of field deployments for two different user groups.

As a whole, from the combination of theory and examples, it has been shown that business processes can be successfully used to generate bots. Although some parts of the chatbot need to be manually programmed, such as custom actions, even with that manual effort, the chatbot development demanded less effort than it would take to generate a bot from scratch, while producing a quite satisfactory result. It was also found that major concepts from the business process domain can be effectively translated into the CA domain.

Chapter 8

Conclusion

In this work, we have presented a guide that helps with the creation of Process-Aware Conversational Agents and have discussed the challenges of associating CAs with business processes. We believe that embedding process theory into conversational agents with good communication skills has a great potential to help users navigate their daily processes.

We have also realized that processes are present not only in business scenarios but also in most of our everyday activities. For instance, for a chef to prepare a specific dish - the outcome -, they have to follow several activities that are represented in the recipe. This means that cooking is a process. Processes can also be more complex, such as a scenario of treating patients, with a compound set of variables, including the patient's symptoms, insurance allowances, and integrated medical systems [19]. Interestingly enough, many of these processes can also be written using BPMN, despite being located outside of the business realm, which leads us to believe that a solution such as ours, that utilizes business process concepts, could be expanded to different areas of our lives.

Our guide has been tested with Exclusive, Inclusive, and Parallel gateways, and both User and Service tasks. It also supports a Camunda feature called User Task Forms, which allows the bot to ask for user input and save the responses into variables for later use. Other BPMN features that were not mentioned in this study might actually work, but they have not been tested in our specific scenarios.

Future work might involve, firstly, implementing more process examples as bots and a formal evaluation of the proposal. Secondly, it would be helpful to create a bot that supports multiple process definitions and automatically recognizes the process definition that the user wants to start. Another possible future development is the association of process mining theory and tools with the process and CA concepts listed in this work. Finally, an interesting extension of this work would be to implement a prototype that can write Rasa files and train its model automatically after parsing a BPMN process.

References

[1] WESKE, M., *Business process management : concepts, languages, architectures*. Berlin New York, Springer, 2012.

[2] DUMAS, M., *Fundamentals of business process management*. Berlin, Germany, Springer, 2018.

[3] DANG, J., TOKLU, C., HAMPEL, K., *et al.*, "Human workflows via document-driven process choreography". In: *2008 International MCETECH Conference on e-Technologies (mcetech 2008)*, pp. 25–33, IEEE, 2008.

[4] DIJKMAN, R., LA ROSA, M., REIJERS, H., "Managing large collections of business process models-current techniques and challenges", *Computers in Industry*, v. 63, n. 2, pp. 91–97, 2012.

[5] GNEWUCH, U., MORANA, S., MAEDCHE, A., "Towards Designing Cooperative and Social Conversational Agents for Customer Service." In: *ICIS*, 2017.

[6] HELANDER, M. G., *Handbook of human-computer interaction*. Elsevier, 2014.

[7] COUGHANOWR, D. R., KOPPEL, L. B., OTHERS, *Process systems analysis and control*, v. 2. McGraw-Hill New York, 1965.

[8] OMG, "Business Process Model and Notation (BPMN), Version 2.0.2", Jan. 2014.

[9] BENTLEY, F., LUVOGT, C., SILVERMAN, M., *et al.*, "Understanding the long-term use of smart speaker assistants", *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, v. 2, n. 3, pp. 1–24, 2018.

[10] CLARK, L., PANTIDI, N., COONEY, O., *et al.*, "What makes a good conversation? challenges in designing truly conversational agents". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2019.

[11] LEE, C., JUNG, S., KIM, S., *et al.*, "Example-based dialog modeling for practical multi-domain dialog system", *Speech Communication*, v. 51, n. 5, pp. 466–484, 2009.

[12] RADZIWILL, N. M., BENTON, M. C., "Evaluating Quality of Chatbots and Intelligent Conversational Agents", 2017.

[13] FAST, E., CHEN, B., MENDELSOHN, J., *et al.*, "Iris: A conversational agent for complex tasks". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2018.

[14] BRADLEY, N., FRITZ, T., HOLMES, R., "Context-aware conversational developer assistants". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 993–1003, IEEE, 2018.

[15] MINER, A., CHOW, A., ADLER, S., *et al.*, "Conversational agents and mental health: Theory-informed assessment of language and affect". In: *Proceedings of the fourth international conference on human agent interaction*, pp. 123–130, 2016.

[16] NICHOL, A., "A New Approach to Conversational Software", Jun 2019.

[17] TOXTLI, C., MONROY-HERNÁNDEZ, A., CRANSHAW, J., "Understanding chatbot-mediated task management". In: *Proceedings of the 2018 CHI conference on human factors in computing systems*, pp. 1–6, 2018.

[18] CRANSHAW, J., ELWANY, E., NEWMAN, T., *et al.*, "Calendar. help: Designing a workflow-based scheduling agent with humans in the loop". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 2382–2393, 2017.

[19] GORRY, G. A., "Modelling the diagnostic process", *Academic Medicine*, v. 45, n. 5, pp. 293–302, May 1970.