ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering Sciences, Computer Science and Psychology
Institute of Databases and Informationsystems

Developing an Extendable Process Engine using Cross-Platform Technologies

Bachelor Thesis at Ulm University

Submitted by:
Dimitrios Kamargiannis
dimitrios.kamargiannis@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Johannes Schobel

2018

Version February 26, 2018

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Despite the increasing digitization in everyday work and industry, data collection is still often based on paper-based questionnaires. One of the areas of application where the disadvantages come to bear are large-scale studies, such as clinical trials. In such studies, an enormous amount of paper and staff is needed for transcription, which leads to logistical problems as well as error susceptibility. The reasons why paper-based questionnaires are still used are often a lack of IT knowledge of the involved, difficult to use existing software, as well as high costs for the development of new customized software. The QuestionSys framework aims to solve these problems. It supports all steps of data collection from the creation of a questionnaire, through its execution on mobile devices, to the analysis of the collected data. In order to ensure a high degree of flexibility when creating questionnaires, questionnaires are mapped to process models which can then be executed by mobile devices.

In the context of this thesis, a lightweight mobile process engine has been developed that allows to execute the process models of the QuestionSys framework. The focus was on process execution, support for several operating systems and easy extensibility. For this purpose, this thesis discusses related work, before the architecture of the engine is presented on the basis of defined requirements. In the following chapter, parts of the implementation are explained, which ultimately leads to an outlook.

Acknowledgement

At this point, I would like to thank all those who supported and motivated me during the preparation of this thesis.

First of all I would like to thank my parents who made it possible for me to study at all. In addition to my friends and my girlfriend, I would like to thank my two fellow students Philipp and Robin, who have spent the entire course of their studies with me.

Special thanks also go to my supervisor Johannes Schobel, who was always there to help me with questions during the project as well as this thesis.

Contents

1

Introduction

Mobile devices such as smartphones or tablets are increasingly represented in society and indispensable for many [1]. These products are in fact so popular that even companies like Amazon have begun to sell their own products, even though this was not part of their market. According to a survey, 48% of internet usage in Germany in 2016 is made up of smartphones and tablets [2]. In 2013, the same devices accounted for only 18% of internet usage in Germany [2]. Considering these huge increases, it is no surprise that more and more companies are beginning to integrate mobile devices into their daily work. Many companies even have their own application and more than half of the employees surveyed in [3] work at least partially on mobile devices.

Companies often use mobile devices to collect required data directly from the customer and then initiate immediate business processes. A very common example is, waiters in restaurants or coffee shops, who use their smartphone to take orders. This allows the responsible staff to process the order immediately, without having to wait for the return of the waiter. The collection of data is an important activity for many companies from various fields and is even relevant within a company (job satisfaction surveys, etc.).

Although mobile applications already support a number of business activities, paper-based questionnaires are often used in areas where more complex data collections methods are required (e.g., clinical trials) [4]. In this type of questionnaire, a quick evaluation as well as the correct completion of the questionnaire is of great importance [5]. Futhermore, such questionnaires are frequently changed and must therefore remain adaptable [4]. However, paper-based questionnaires can seldom meet these criteria, and may even lead to further disadvantages. This kind of questionnaires can require enormous resources and space, especially in large-scale studies [6]. The collected

data must then often be transferred to a computer for analysis [7], which can lead to transcription errors and, in the worst case, jeopardizes the validity of the results of a study.

Alternatively, there are some questionnaire systems on the market, that often do not support complex execution logic or require experts to make appropriate adjustments. To solve this and the problems mentioned above, the University of Ulm is currently developing the QuestionSys framework. This framework should make it possible to create, deploy, execute, evaluate, and archive questionnaires. The peculiarity of it is that questionnaires are mapped to process models. Therefore, a process engine is needed to execute them. The questionnaires should mainly be executed on mobile devices. Additionally, because operating systems of mobile devices differ [8], a cross-platform solution is desirable.

1.1 Objective

The goal of this bachelor thesis is the planning and development of a lightweight process engine using cross-platform technologies, so that the engine can be embedded in an application designed for mobile devices. The developed engine should be able to execute every well-formed process model created by the QuestionSys framework. In addition, questionnaires with complex execution rules are to be supported, with which the further course can be treated individually, based on the previous answers of the user. The collected data during runtime should be organized and communicated to the rest of the application via an API (Application Programming Interface) upon completion of a questionnaire. Additionally, as the QuestionSys framework is constantly evolving, certain parts of the developed engine should be easily extendable and exchangeable.

1.2 Outline

Since this thesis is about the development of the process engine, it primarily deals with the conception, implementation and challenging aspects of the process engine.

First, Chapter 2 covers the basics of understanding this thesis. The QuestionSys framework is explained in more detail in Section 2.1, before discussing process models in Section 2.2. In Chapter 3, related work is presented and discussed. Subsequently, Chapter 4 defines the requirements for the engine to be developed, which are divided into `functional` (Section 4.1) and `non-functional` (Section 4.2) requirements. Next, the developed concept and architecture are presented in Chapter 5, beginning with the overall concept of the developed engine (Section 5.1). Then, Section 5.6 discusses the concept of logging and Section 5.7 the results, that reflect the data collected at runtime. In the following Chapter 6, the used framework is discussed and essential parts of the implementation are presented. Finally, the findings of this thesis are briefly summarized, followed by an outlook on future meaningful extensions of the engine developed (Chapter 7).

2

Fundamentals

2.1 QuestionSys Framework

The QuestionSys project was launched in 2013 at Ulm University [9]. The aim of this project is to eliminate the problems mentioned in Chapter 1, and to allow experts from different fields (e.g., medicine, psychology), without programming skills, to create complex digital questionnaires. The QuestionSys framework is designed to support the entire lifecycle of a digital questionnaire, seen in Figure 2.1 [10].

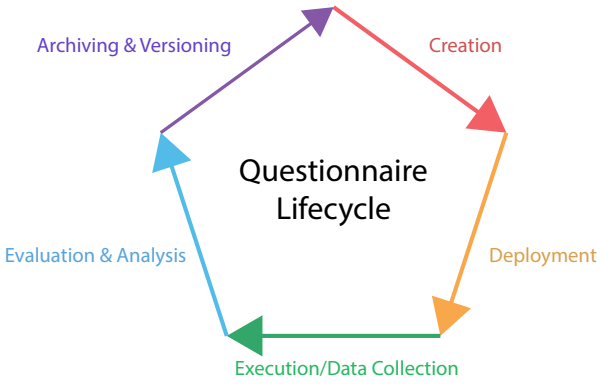

Figure 2.1: Lifecycle of a Digital Questionnaire

A questionnaire must first be created, before it can be deployed to different devices (e.g., mobile devices). There, the questionnaire can then be instantiated and executed. Subsequently, the data collected by the questionnaire can be evaluated and analyzed. Finally, the collected data can be versioned and archived [10].

The QuestionSys framework uses a process-oriented approach , which maps a question-

naire to a process model [11]. This allows the questionnaires to be executed on different devices by a process engine. Using a lightweight mobile engine, like the one developed in this thesis, questionnaires can also be executed on mobile devices.
The different phases that a questionnaire goes trough, are supported by the QuestionSys framework through three different components (Configurator, Server, Client). How these components work together is shown in Figure 2.2. The individual components are explained in more detail below.

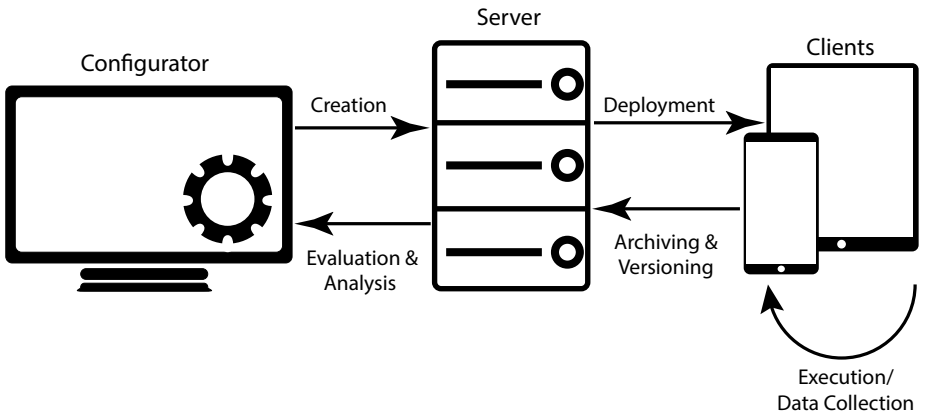

Figure 2.2: Overview of the QuestionSys framework

2.1.1 Configurator

The configurator is responsible for the creation of the questionnaires. Questionnaires can be created by linking available elements via drag and drop. The way they are linked, defines the way the questionnaire is executed.
So far, questionnaires typically consist of several pages, containing different types of questions as well as headlines and texts. Certain elements (`gateways`) can be used to alter the execution logic, depending on already given answers.

The questionnaires created with the configurator, can then be exported as a process model and sent to the server.

2.1.2 Server

The server stores the questionnaires created by the configurator. These questionnaires can then be downloaded by clients. Additionally, the collected data can be uploaded from the clients to the server for archiving and further analysis.

2.1.3 Client

The client is responsible for the execution of the questionnaires generated by the configurator, and thus for the data collection. Questionnaires can be obtained from the server, and collected data can be uploaded to the server using an API. A mobile application has been developed, which uses the engine developed in the context of this thesis, to execute the obtained questionnaires. The presentation of the questionnaires as well as the interpretation of the execution logic is done completely by the engine. The collected data can then be transferred from the application to the server.

2.2 Process Model

Every service from the preparation of a meal, to knowledge-intensive tasks such as the treatment of a patient can be a business process. A business process can consist of pure online or real services or, as usually, a combination of both [12]. Business processes can be defined in a modeling language (e.g., BPMN 2.0) and implemented, so that they can be executed by a business process engine [12]. Modeling languages usually allow a variety of ways to design a business process. Since it is not necessary to know every design possibility of such a model, only necessary parts are discussed below.

A process model is a graph-based representation of a business process. The QuestionSys framework is using a block structured approach to map the questionnaires on

(cf. Figure 2.3). The presented process model has not been designed with a particular modeling language and contains only elements that are relevant in the context of this thesis, except for the cigarettes and alcohol elements that represent collected data.

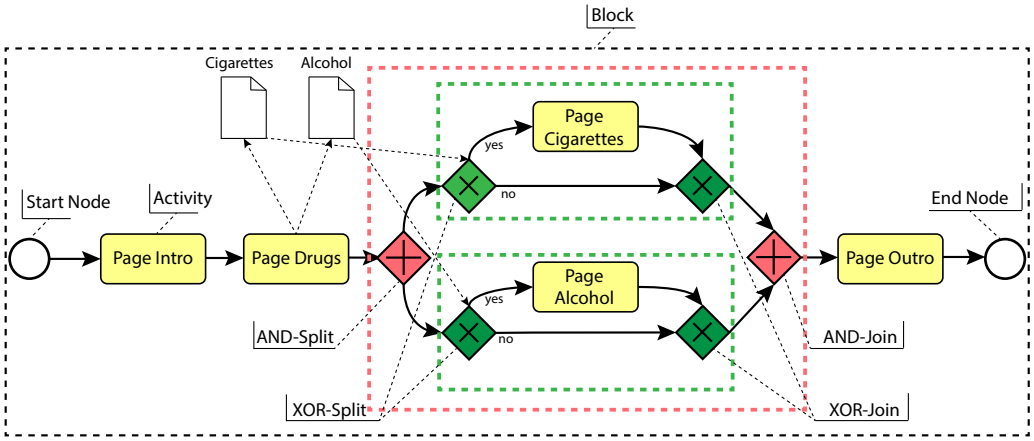

Figure 2.3: Questionnaire as a Process Model after [4]

Block Structured:

Every process model of the QuestionSys framework is block structured. This means that every process model has exactly one starting and one ending node. In addition, each block must be free of overlaps. That means, a block that begins, for example, with a `XOR-split` must also end with a `XOR-join` (i.e., well-formed).

Nodes:

There are three different types of nodes. `Start` nodes that designate the beginning of every process model and `End` nodes that describe the end accordingly. Every other node represents some kind of activity that needs execution. In the context of this thesis, this nodes contain data representing a question that has to be answered or a text to be presented. `Start` nodes have always only one outgoing edge, while `End` nodes have only one incoming edge. Nodes representing activities have one outgoing as well as one incoming edge.

Edges:

In addition to nodes, there are edges that are responsible for the control flow. Edges are directed and always connect two elements together. This way, an execution order can be set. This also means that if an edge connects node A to node B, node B can not be

executed until node A has finished.

Gateways:

`Gateways` are elements of the process model that influence the execution logic, by making the further course dependent on certain conditions. Three different types of `gateways` are used in the context of this thesis (`AND`, `XOR` and `LOOP`). Every `gateway` consists of two parts, the `split` element and the corresponding `join` element. Every `split` element starts a new block while every `join` element closes the last started block, as seen in Figure 2.3.

The above elements define a tool kit, that makes it possible to design an infinite number of different process models. In the context of this thesis the different process models represent different questionnaires. Using this paradigm when creating a questionnaire allows a process engine to execute it.

3

Related Work

In this chapter related work is presented, which also deals with mobile process execution. Section 3.1 deals with a process engine implemented in Java, with which various Process Model Notations can be integrated. MARPLE (Section 3.2) provides a Process Management System that enables the execution of process models on mobile devices. Finally, in Section 3.3 WOtAN is presented, which is able to execute and analyze ADEPT-based process models.

3.1 jBPM

jBPM is a flexible Business Process Management Suite designed to bridge the gap between business analysts and developers [13]. jBPM's core is a lightweight, extendable workflow engine written in pure Java, which allows execution of process models using the latest BPMN 2.0 specification [13]. In this case, extendable means that the engine provides a generic Process Execution component that enables other Process Model Notations (e.g., BPEL) to be integrated [14]. On top of the engine, this suite offers support for graphical creation of business processes, as well as monitoring and management of the process models and instances.

To interact with the process engine, it is necessary to set up a session as shown in Figure 3.1. Sessions need a reference to a *Knowledge Base*, which contains all the relevant process definitions. Therefore, a *Knowledge Base* has to be created first, before setting up a session. This session can then be used to start executing processes, creating a new process instance each time a process is started.

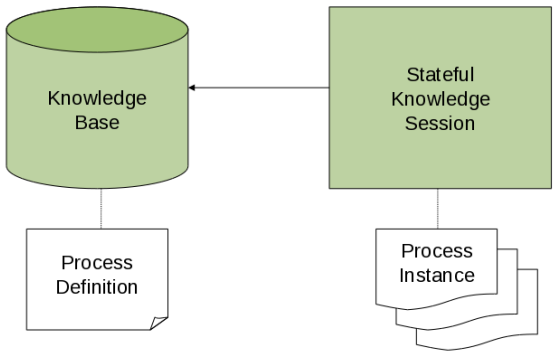

Figure 3.1: Overview of jBPM's Core Engine API [15]

In addition, it is possible to create several sessions as well as different sessions that share the same *Knowledge Base* [15].

3.2 MARPLE

The *MARPLE* [16] project has been developed since 2009 at the Ulm University. The goal of this project is the development of a lightweight process engine that allows the mobile execution of centrally stored process models, by using a client-server architecture. *MARPLE* consists of two main components, the *MARPLE Mediation Center* and the *MARPLE Mobile Engine*, shown in Figure 3.2.

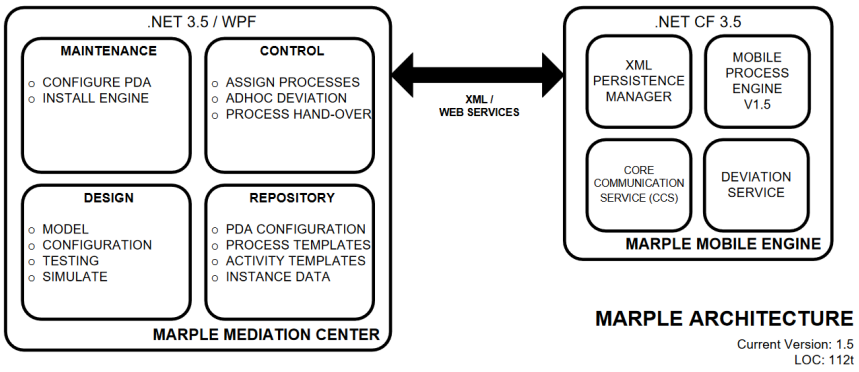

Figure 3.2: *MARPLE* Architecture [16]

The *MARPLE Mediation Center* itself, consist of four different components. These components are responsible for the administration of process models and the deployment of the *MARPLE Mobile Engine* on mobile devices. It is possible to create, configure, and test process models using the *Mediation Center*, as well as executing a process model. It is also possible to make Ad-hoc changes to the ongoing process.

The *MARPLE Mobile Engine* consists also of four components. The *Mobile Process Engine* is responsible for the execution of the processes, while the *Deviation Service* is used to manage dynamic customizations of a running process instance. Furthermore, the *Core Communication Service* is used to communicate with the *MARPLE Mediation Center* and the *XML Persistence Manager* manages the downloaded process and activity models.

3.3 WOtAN

The framework *WOtAN* [17] (Workflows on Android) was developed in 2016 as part of a master's thesis. The focus in the development of *WOtAN* was on process execution and process analysis. *WOtAN* is able to run process models designed with ADEPT.

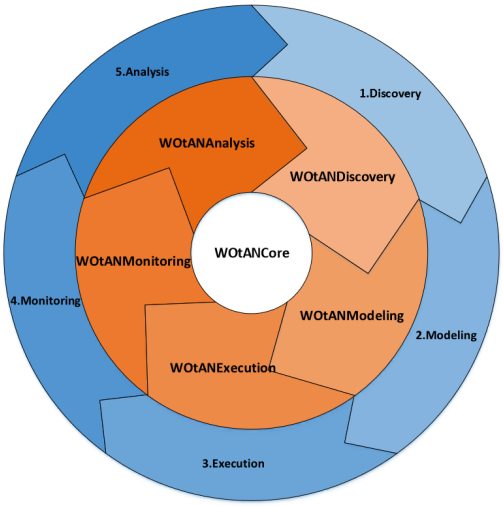

Figure 3.3: *WOtAN* Modules Integrated in BPM Lifecycle [17]

WOtAN consists of 5 modules connected by one common core, as seen in Figure 3.3. All modules can be used individually or in any combination as long as the core is included, although some combinations may not be useful. The different modules have different functionalities. Accordingly, *WOtANDiscovery* is used for finding existing process models in companies. *WOtANModeling*, another module, is not only responsible for the creation of new process models, Ad-hoc changes can also be made to existing ones. As the name implies, *WOtanExecution* is responsible for the process execution, while *WOtANMonitoring* is able to live monitor running process instances. Finally, the *WOtANanalysis* can be used to analyze executed process models.

3.4 Discussion

In this chapter, three process engines were presented, which are discussed in this section. First, jBPM was introduced, a powerful lightweight engine that can execute process models based on BPMN 2.0. It is even possible to integrate further Process Model Notations. Furthermore, it allows to create process models via drag & drop. But jBPM also has its disadvantages. Since this engine is implemented entirely in Java, it can only operate on devices that support Java. In addition, although the engine can be operated on mobile devices, it is not optimized for it.
MARPLE was next introduced, and unlike jBPM, it was designed for mobile devices. Another difference is that *MARPLE* is based on ADEPT. *MARPLE* is a project that has been continuously evolving for almost 20 years but is based on a client-server architecture.
Last but not least, *WOtAN* was presented, which concentrates on process execution and analysis, but also offers further possibilities (e.g., modeling). Just like *MARPLE*, *WOtAN* is also based on ADEPT.

In conclusion, all the featured engines have their own advantages and disadvantages, but none of them has been designed with cross-platform technologies. This means that new implementation will be needed to support more mobile operating systems.

4

Requirements

This chapter outlines the requirements for the engine developed in this work. These requirements are divided into `functional` and `non-functional` requirements. The former describe what the system should be capable of. `Non-functional` requirements describe how a system is meant to be.

4.1 Functional Requirement

In this section the `functional` requirements are introduced.

FR1 **(Execute a Process Model):** The lightweight engine to be developed, should be able to process the process model defined by the configurator of the QuestionSys framework. The engine should be able to execute tasks that are manual and require the user's participation, as well as tasks that are automated.

FR2 **(Pause and Resume/Continue Session):** Every running instance of a process model should be able to be paused. Of course, that also implies the procedure of loading and continuing interrupted instances. This, in turn, allows the user to process just a part of the model at a time, or start another instance before completing the running one.

FR3 **(Control Flow):** The predefined process model of the QuestionSys framework is using `gateways` in order to control the flow of an instance. The engine has to process them correctly. Additionally, as the process model is constantly evolving, the engine should provide an interface that allows the implemented `gateways` to be exchanged or expanded.

FR4 **(Collect Data):** Data collected during runtime should be stored and returned. In the context of this thesis, every given answer should be recorded. The recorded answers should be clearly assigned to a question and contain additional information like a timestamp or an iteration counter.

FR5 **(Logging):** The engine to be developed should support extensive logging. Different levels of logging as well as different logs should be possible. At least one log should cover the procedures inside the core part of the engine. In addition, exchangeable components of the engine, should be able to use a different log. In the context of this thesis, the process of answering a question should also be logged. The format used to log should allow process mining,hence a format that does not require much transformation to be used by such an algorithm.

4.2 Nonfunctional Requirements

This section introduces the `non-functional` requirements.

NFR1 **(Lightweight):** The engine to be developed is designed for mobile devices, and should ,therefore, be particularly resource-efficient.

NFR2 **(Maintainability):** Besides a good documentation, the engine to be developed must be easy to maintain. This also means that changes in the execution logic of certain parts should only affect specific classes. To accomplish this, different functionalities have to be divided into independent components.

NFR3 **(Extendable):** The engine should consist of a core part and some exchangeable parts. As the QuestionSys framework is constantly evolving, new requirements can emerge or old ones can change. To account for these changes, parts of the engine should be exchangeable without having to change the core part. This is also important as the engine to be developed might also be used for other scenarios beyond the scope of this thesis.

NFR4 **(Stability):** A stable operation of the engine should be ensured. To achieve this, two things are needed. First, sources of error must be minimized by examining and preventing potential errors. Secondly unexpected errors must be `catched` and treated respectively. The engine should never fail in a way that requires a restart of the entire application.

NFR5 **(Multiple Instances):** The engine should be able to allow for processing several different process models simultaneously.

Based on the defined requirements, the architecture of the engine was developed. Subsequently, the implementation could take place, with which then the requirements could be fulfilled. The resulting engine was then used in the application that was created in the context of this thesis.

5

Concept & Architecture

Taking the requirements defined in Chapter 4 into account, a lightweight engine was developed. This chapter, in turn, explains the developed process model that is used by the engine. Next, the architecture of the engine and all its components are presented, before discussing each one in more detail. Finally, the concept of logging and the generated results is described.

5.1 Overall Concept

The overall concept consists of two parts. First, what the nodes of the engine's process model look like, and secondly, the architecture of the engine.
The process model the engine uses is influenced by the process model the configurator of the QuestionSys framework is generating. The process model of the configurator consists of two arrays. One of the arrays contains the nodes that are identified with a unique key, along with some other individual properties. The other array contains edges that define, which nodes are connected, by using keys of the nodes respectively.
Using this format directly requires frequent and recurring searches of both arrays to process the model. But since the developed engine is required to be resource-efficient (**NFR1**), the described model is transformed. The newly developed process model is designed similar to the model of a linked list. Every node contains some data, and a pointer to the next element of the list, as shown in Figure 5.1. In contrast to an array, it is more time-consuming to find individual nodes at certain positions, but at the same time easier to go through the process model in one direction. Since the process model of the configurator contains `gateways` that are connected to more than one node, the property

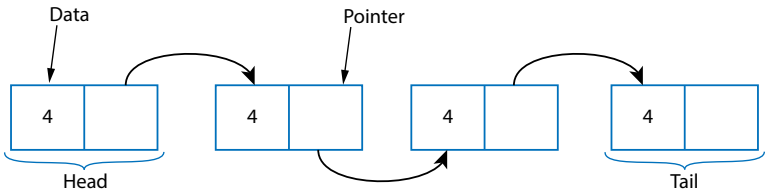

Figure 5.1: How a linked list is connected

containing the pointer to the next node, is an array that can contain multiple pointers. To use this model, the configurator's process model must be transformed into this new process model, which adds additional computation. However, since the transformed process model can be stored, this procedure must be performed only once.

JavaScript Object Notation (JSON) was chosen as the format of the process model's nodes, as it is a format that can be parsed by many programming languages and can also be read by humans. The format consists of `key/value`-pairs, which are grouped to an object. Values can contain any common data type (e.g., `string`, `boolean`, `array`, etc.) as well as other JSON objects [18].

The model consists of two different types, so-called `Node` and `Gateway`. Type `Node` represents a node containing one or more activities (e.g., a question or an information message to be presented), while a `Gateway` contains data responsible for control flow. Thus, a `Gateway` type node determines, which node will be processed next, depending on given answers or other conditions. For this reason, the properties of the two node types differ as shown in Figure 5.2.

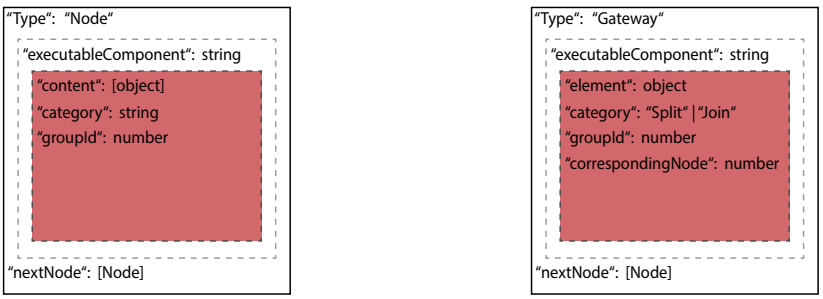

Figure 5.2: The two Node Types the Process Model Consists of

Both types have the properties shown in the white box in common. The reason for this is that this properties are needed in the core part of the engine that was designed according to **NFR3**. The properties shown in the red box and ,especially, their values can differ according to the context the engine is used in. For example, in the context of this thesis, the `content` property contains several objects, each of them representing an activity, like a question or a headline (cf. Figure 5.3).
Considering the resulting process model, the concept of the engine was developed.

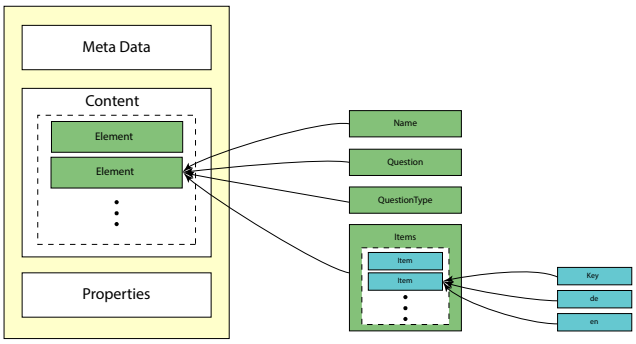

Figure 5.3: Example of a Multiple Choice Question

The engine itself, however, is divided into two main parts: the core and the exchangeable part. Each of them, in turn, consists of several components. The architecture is shown in Figure 5.4. The `Main Unit` is the only component that communicates with other applications or external components. Therefore, every process model that needs execution gets passed to it. Furthermore, it is responsible for deciding, whether the currently processed node is a `Gateway` or a `Node`, by using the `type` property. Depending on this, either the node is passed to the `Runtime Manager` (RTM) or a `GatewayComponent` (GWC) is generated, based on the `executableComponent` property, and the node is passed to it. The GWC will then decide, which node to continue with and return it to the `Main Unit`, restarting the same procedure on the returned node.
When the node is forwarded to the RTM, the RTM generates the responsible `ExecutableComponent` (EXC), based on the data the node contains. The node is then forwarded to the generated EXC.

Depending on the EXC's implementation several scenarios are possible. The EXC could

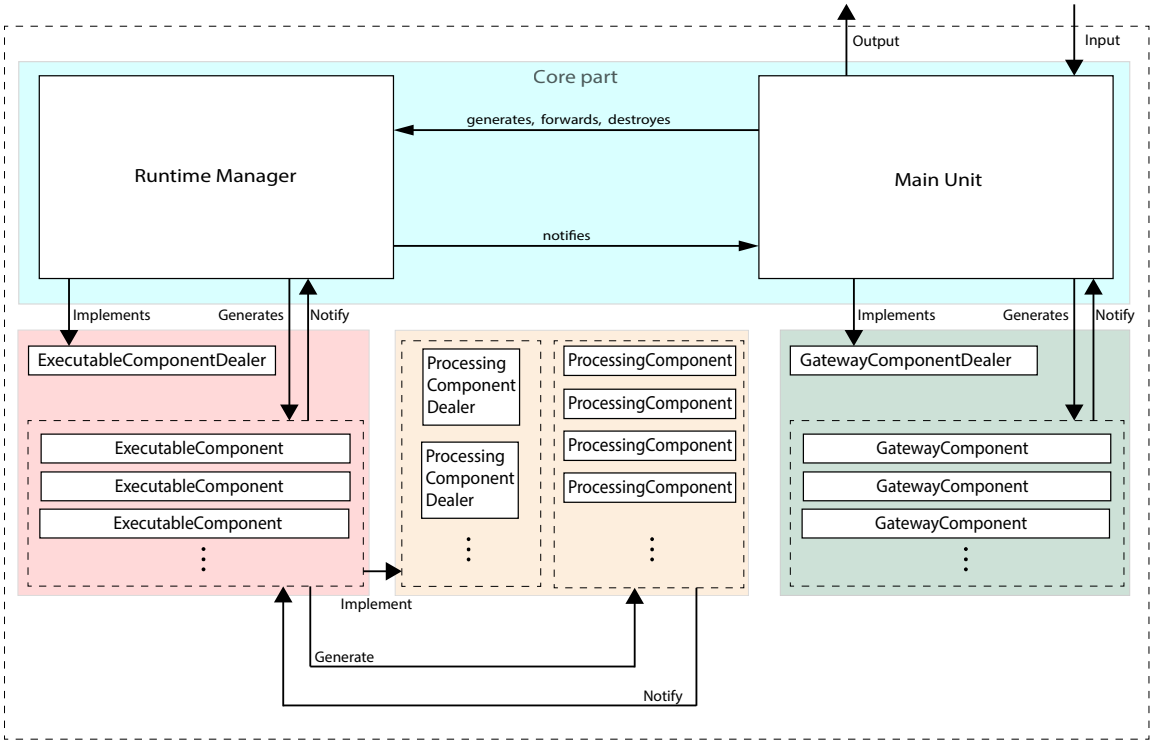

Figure 5.4: Architecture of the developed engine

either use the information inside the node to do the intended task, or create as many `ProcessingComponent` (PCC) as needed to split the task. Either way, when the EXC has finished, it will notify the RTM about it. The RTM will then destroy the generated EXC and notify the `Main Unit` about completion. The `Main Unit` will then proceed to the next node in the process model by using the `nextNode` property. The whole procedure will continue until the model is completely processed or the engine is interrupted.

5.2 Main Unit

The primary goal of the `Main Unit` is to decide where to pass the momentary processed node, using the `type` property mentioned in Section 5.1. But since it is necessary to generate GWCs in case of a `gateway` node, GWCs need to be integrated somehow into the `Main Unit`. To achieve this, the `GatewayComponentDealer` was introduced,

which is a class with a list containing references to available `GatewayComponents`. This class is imported by the `Main Unit`, allowing to add and remove custom GWCs, by listing the components there, without altering the core part. The previously mentioned `executableComponent` property of the node determines, which GWC needs to be created.

Since the `Main Unit` is the only component communicating with external components, it initiates every procedure. As a result, all components have a reference to the `Main Unit`. Therefore, all globally relevant data (e.g., results or logs) is stored there. The results collected during runtime are broadcast app-wide, at each completion or pause by the `Main Unit`. This way collected results can be stored by other parts of the application.

5.3 Runtime Manager

The RTM manages everything about the EXCs. The manager generates, executes, and destroys the required EXCs. To accomplish this, the RTM imports the `ExecutableComponentDealer` that works like the `GatewayComponentDealer`, but contains EXCs instead. The RTM is generating EXCs the same way, the `Main Unit` is generating GWC. In addition, the view (if any) of the EXCs is displayed in a container offered by the RTM. Upon completion, the RTM will notify the `Main Unit` about it.

5.4 Gateway Components

Every GWC needs to implement an interface that is not very restricting. It just defines two methods, which will be called by the `Main Unit`. The concept is intended to allow as much freedom as possible in the implementation of GWCs. Consequently, a variety of `gateways` can be implemented, as well as different versions of one and the same.

5.5 Executable Components

In order to process the activities within the nodes, EXCs are needed. They belong to the exchangeable part of the engine. Every EXC must implement an interface, just like the GWCs, which is also not very restrictive. This interface is needed to ensure the communication between RTM and EXC. Any task required to complete the current activity can be performed here. If tasks have to be split, an EXC can perform a similar function as the RTM and manage PCCs. This can be accomplished by importing custom `ProcessingComponentDealers` that contain PCCs. These components can then be used as desired. There are no restrictions for them, although it is recommended to define some kind of interface.

5.6 Logging

Logging is an essential part of the whole engine, especially of the core part. Supporting different levels of logging is an important requirement (**FR5**). By comparing logging techniques of other applications, four logging levels were defined (`info, debug, error, warn`). In addition, three different logger classes are set, to allow different parts of the application to write their own logs. Every part of the application has access to the loggers and is able to use them. This allows easy integration of logging into custom components.

The first logger, called `ProcessLogger`, is documenting all procedures in the core part. Therefore, this logger can not be altered without making changes in the specific classes. The other two loggers can be altered and differ according to the context. In the context of this thesis, the second logger (`ComponentsLogger`) is used to document all procedures in the EXCs and PCCs, while the `DataHistoryLogger`, which is the third one, is used by the PCCs to log every action taken by the user while answering a question.

Each logger class implements a predefined interface to ensure they can always be used the same way. The difference between the logs is the data stored each time, otherwise they are the same. The format in which is logged should allow process mining but also

be readable by humans. Additionally, basic operations, like sorting or filtering, should be easy to perform. For this reason, JSON was chosen as the logging format, organized in a list where every entry is a single JSON object. This provides readability while retaining the benefits that objects offer.

5.7 Results

The overall intention of the developed lightweight engine is to collect data or produce some kind of results, depending on the context in which the engine is used in. Since the desired results may differ, the generation of results must take place in the exchangeable part of the engine.
Considering that every component has access to the `Main Unit`, which stores the results, the procedure of generating results has been moved to the EXCs and PCCs. Both component categories can read and write results depending on the implementation. The only limitation to the results is that they must be stored as a JSON object. Otherwise the results can be of any structure and are exclusively generated by the mentioned, exchangeable components. Consequently, the structure of the results can be arbitrarily changed to suit many different scenarios.

In the QuestionSys application the engine is used in, only the PCCs generate and store results, although it is possible to add EXCs that do. In this context, results are representing answers given to certain questions. The final result structure is seen in Figure 5.5. Every question has a unique id, identifying the question. The easiest way to store them is using this id as the `key` and the answer to this question as the corresponding `value`. However, since meta data (such as timestamp, iteration) should also be stored, the answers to the questions are wrapped in an object that contains the meta data, called `resultObject`. This object has a property called `payload`, which contains the given answer.

The next important thing is, there are two types of questions in this context. There are questions that have one specific answer (e.g., name, age), and questions that allow multiple answer options (Multiple Choice). Therefore, the `payload` can either contain

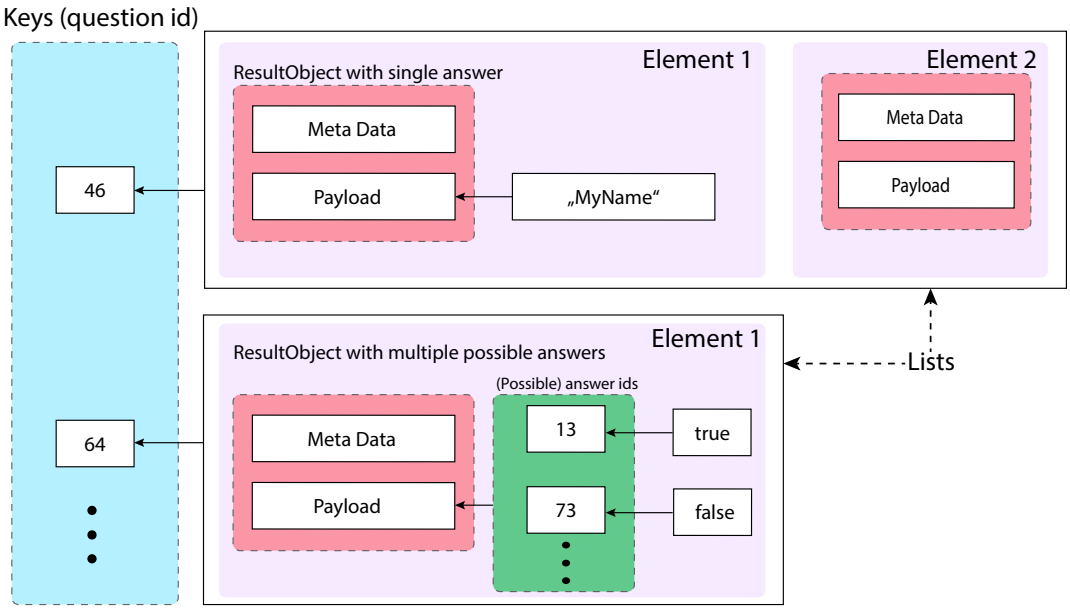

Figure 5.5: Structure of the generated results

one value (e.g., string) or an object containing further `keys` (the possible answers). The id of each possible answer is used as `key` and `true` or `false` as `value`, depending on the user's answer. There are a few exceptions that use values other than `true` or `false`, but do not affect the structure of the results.

The last adaptation that had to be made was due to possible `LOOPS`. A `LOOP` is a possible `gateway`, which can be used to cycle trough specific paths of the process model, processing the same questions (with same id) multiple times. To make this possible, the `resultObject` that contains the answer (or answers) to a question is inside a list. Each time a question is answered, a new `resultObject` is added to the end of the list. This last adjustment leads to the final structure and meets all the requirements of the process model.

6

Implementation

This chapter first introduces the Ionic 2 Framework as the lightweight mobile engine, developed as a part of this thesis, is embedded in an application created with this framework. It further introduces fundamental parts of the implementation that demonstrate how the framework was used.

6.1 Ionic 2 Framework

The Ionic 2 framework is a open-source project for creating mobile web-based hybrid applications [19]. For graphical representation, the web technologies HTML5 and SASS (a superset of CSS syntax [20]), are used. TypeScript, which is a “typed superset of JavaScript that compiles to plain JavaScript” [21] is used for developing the application logic. Ionic 2 builds on top of two additional frameworks called Angular 4 and Apache Cordova .The overall setup is shown in Figure 6.1

Ionic 2 focuses on the front-end of a mobile application, especially on the appearance and interaction of the user interface. For example, it provides pre-build and partially adjustable, user interface components like lists or buttons. This allows for a quick and easy implementation of frequently used components. These components, in turn, have a different appearance depending on the underlying platform (e.g., Android or iOS), offering a native look and feel. Ionic Native, an enhancement of the framework itself, is a set of TypeScript wrappers for Cordova plugins, allowing access to native functionality (e.g., camera, file system) in Ionic applications. It additionally wraps plugin callbacks in a way that allows Angular 4 to react properly to them [22]

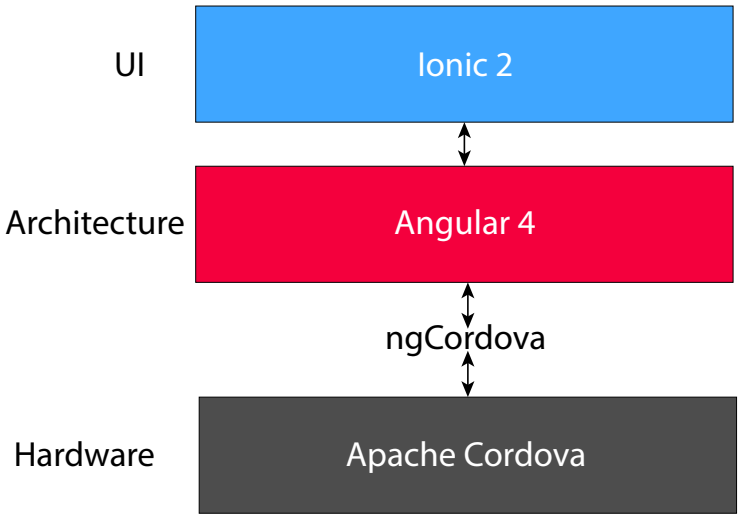

Figure 6.1: Stucture of the Ionic 2 framework

Apache Cordova is an open-source mobile development framework using standard web technologies (HTML5, CSS3 and JavaScript). Applications run within wrappers that target every platform, relying on standards-compliant API bindings to access the capabilities of each device, such as sensors and network status [23]. For each device function, a separate plugin must be added to the project to use its API. However, this also has the advantage that no unnecessary code inflates the project. Since this framework offers completely different functionalities than the Ionic 2 framework, it complements it particularly well.

The last framework contained is Angular 4. “Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript” [24]. It is, therefore, a set of several libraries with some of them core and some optional. Angular applications consist of HTML templates with angularized markup, that allow advanced features such as property binding. These templates, in turn, can then be managed by component classes, written in TypeScript. One can add application logic by implementing services, and bundle components and services to modules. According to the official documentation, “a service is typically a class with a narrow, well-defined purpose” [24]. Services typically provide specific functions that are

used by various components in the application. As services can be implemented as singleton, meaning every component using this service will work on the same instance, it is a great way to exchange information across different components [24].

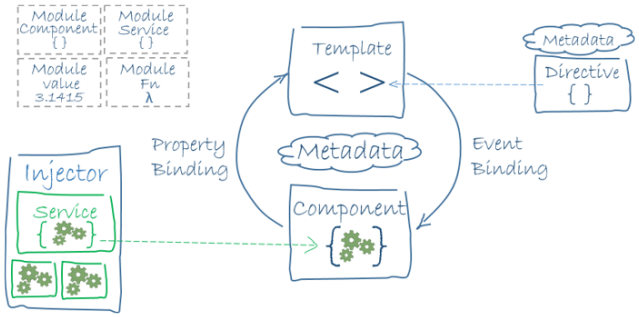

Figure 6.2: Structure of Angular 4 [24]

Another feature that needs to be mentioned, as it has been used frequently in this thesis, are lifecycle hooks. Every component has a lifecycle managed by Angular. The latter creates, renders and destroys every component and offers lifecycle hooks that provide the ability to respond to specific key life moments. Developers can implement this hooks to tap into these key moments. The implemented hook methods are then automatically called for the respective key moment [24].

6.2 Implementation of Selected Components

This section introduces selected components of the implementation, especially covering the engine. It explains core functions, how they work and the reasons why they were implemented like this using the Ionic 2 framework.

6.2.1 Main Unit

The `Main Unit` is implemented as a service, which allows easy integration into other components, while still being able to provide multiple instances, as required by **NFR5** [24]. In addition, great importance was attached to ensuring good error handling and

catching unexpected errors as required in **NFR4**. First, the relevant properties of the `Main Unit` are discussed.

_history: Completed nodes are pushed on this array.

results: An empty JSON object. Can be used to store results/collected data.

_currentNode: A reference to the currently processed node.

_splitStack: This is an array containing the `split` elements. Every time a `gateway` is processed, which is a `split`, the `gateway` is pushed on the top of this array. This way, whenever a `join` is processed, it can be assigned to the corresponding `split` element, which is then dropped from the stack. This is possible because of the block structure mentioned in Section 2.2.

When the `Main Unit` is instantiated several things happen. First, the logger classes are set up. In addition, the `Main Unit` is subscribing to the app-wide `Event` channel called “interrupt”. `Events` can be used to publish data on a specific channel across the whole application. Each component of the application can subscribe to such a channel. A subscription enables the component to react individually and immediately, as soon as something is published on this channel. The „interrupt“ channel is used by the RTM. In case the user manually pauses the execution of a process model, the RTM publishes an object on this channel, containing information about the corresponding `Main Unit`. This `Main Unit` then publishes (on another channel) an object with which the interrupted instance can be continued later, and shuts down.

Initialize:

The engine must first be initialized, before it can be started. To start a new instance of a process model, the `initialize` method is used, which takes the following parameters.

navCtrl:	The *NavigationController* on which the RTM will later be pushed by the `Main Unit`.
model:	The process model to be executed.
options:	An array that gets passed down to the PCCs, which can differ, depending on the use case (e.g., selecting a specific language).
logOptions:	(*Optional*) Can be used to log only on a specific level (e.g., only `info`).

This method initializes the `Main Unit` and sets default values. Additionally, it sets the `_currentNode` to the passed `model` and the other parameters to their respective properties. In case `logOptions` are not passed, they will be an empty object. If everything worked properly, the property `_isInitialized` is set to `true` and the method will return `true`. Else this method will return `false`.

LoadInstance:

If a paused instance is to be continued, the `loadInstance` method must be called instead. The parameters of this function are also explained below.

navCtrl:	The *NavigationController* on which the RTM will later be pushed by the `Main Unit`.
engineData:	The object the `Main Unit` published when it was interrupted.
instanceId:	(Optional) Can be used to identify the instance again. Has not type restrictions and is `null` by default. Is contained in the `resultObject`.

This method also starts by initializing the `Main Unit` and sets default values. After that, the `Main Unit's` properties are overwritten with the values contained in the `engineData` object. This way, the engine is reset to the state it had when it was interrupted. This is possible, because the `engineData` object is a serialized[1] copy of the engine at the time of interruption.

However, this creates a problem. The `splitStack` property contains objects (the `split` GWCs), which have their own methods. The drawback of serialization is that only properties but no methods can be saved. Keeping the restored GWCs like this, would

[1] A text representation of a JSON object. Often used to store an object permanently

result in an error, as soon as a method of it is called. This would eventually render the whole procedure of loading an instance useless.

To solve this, every previously generated GWC stores an identifier, called `gatewayCompName`, with which it was created during runtime. This identifier is used by the `Main Unit` to create the correct GWC during execution. The property can then be used to create the same component, which was created during runtime, again and pass it the data of the serialized one. This procedure can be seen in Listing 6.1. When loading an instance, the method loops through a temporary stored array called `tmpSplitStack`, containing the serialized GWCs. For every element inside this array, a new GWC is created, depending on the identifier. This newly generated GWC contains the necessary methods. Subsequently, this newly created GWC is then set to the state it had at the time of interruption, by transferring the properties of the serialized GWC to the new one. Finally, the object is pushed on the `splitStack`.

```
1  loadInstance(navCtrl, engineData, instanceId?):boolean{
2   for(let comp of tmpSplitStack){
3   let gatewayComponent;
4           ...
5     }
6     //Generate new GatewayComponent depending on gatewayCompName
7    } else {
8     gatewayComponent = new GatewayComponentDealer.comps[comp.gatewayCompName];
9    }
10   //Now pass all relevant data to the newly generated GatewayComponent
11   for(var key in comp){
12    if(comp.hasOwnProperty(key)) {
13     gatewayComponent[key] = comp[key];
14    }
15   }
16   //Add engine reference after loop
17   gatewayComponent.engine = this;
18   //Finally push element on SplitStack
19   this._splitStack.push(gatewayComponent);
20  }
```

Listing 6.1: Excerpt of the `loadInstance` method from the `Main Unit`

Several `catch`-blocks were used to be able to distinguish between an error of the `engineData` object, and a missing GWC that is not accessible or installed right now, but was used in this instance. In both of these cases the method will return `false`. If no errors occured the `_isInitialized` property is set to `true` and the method returns `true` as well.

Start

To start the processing of an instance and generate the corresponding view via the RTM, the `start` method needs to be called.

```
1  start(optionalParams?){
2   //Can be done with logical OR as only objects or null is valid
3   optionalParams = optionalParams || null;
4   //Sets loggers depending on the logOptions
5   ...
6   if(this._isInitialized == true){
7    this._navCtrl.push(RuntimeManager, {engine: this, options: this._options,
         optional: optionalParams, procLogger: this.log})
8    .then(()=>{
9          this._rtmViewController = this._navCtrl.last();
10         this._rtmComponent = this._rtmViewController._cmp.instance;
11         this._process();
12   })
13   ...
14  }
15 }
```

Listing 6.2: Excerpt of the `start` method of the `Main Unit`

optionalParams: (Optional) This is a JSON object that can be used to alter the appearance of the RTM. But as this contributes only to the visual appearance this will not be discussed further in this thesis.

As shown in Listing 6.2 this method just stores some necessary properties to start processing. It sets the loggers depending on the previously passed `logOptions`, before generating the RTM and pushing it onto the previously passed *NavigationController*. Storing the `_rtmViewController` is necessary to be able to close the page later. The `_rtmComponent` is the direct reference to the codebase of the RTM and allows access

to it. Finally the method _process is called.

Process

The following information is shown in Figure 6.3.

The _process method is the main method of the Main Unit. It is responsible for the forwarding, mentioned in Section 5.1. The method first checks the _isTerminated property. If it is true it will call the method _populateResults which will then generate and publish the results. After that, the _shutDown method is called that will remove the RTM from the *NavigationController* by using the previously set property _rtmViewController. Finally the Main Unit's state is set to default. Further explanation to the _populateResults and shutDown methods can be found later on.

```
1 _process(){
2   if(this._isTerminated === true){
3     //Doing some logging stuff here
4     ...
5     this._populateResults(true, false);
6     this._shutDown();
```

Listing 6.3: Excerpt of the process method of the Main Unit

Next is a similar check to the one in Listing 6.3. This time, however, the _currentNode property is checked. In case it is null or undefined, the same happens as in Listing 6.3, except that the _populateResults method is called with different parameters. This is done to determine later that an error occured during execution of this instance. Nothing happens if the _currentNode property contains an object. This and different behaviour of the loggers in both cases led to separating this checks from each other.

The next step is also the main functionality of this method. Depending on the type property of the _currentNode, there are different possible behaviours. There are only two valid types, Gateways and Nodes, all other types are ignored and treated as invalid.

In case of a Gateway the procedure is simple. The method tries to generate a GWC, relative to the information of the executableComponent property contained in the _currentNode object. The generated GWC has, according to the defined interface

(cf. Section 5.4), a method called `evalNode`. This method is called after creation. The `_currentNode` reference, as well as some additional information is passed to the GWC this way. The generated GWC then asynchronously returns the next node to proceed. This has been done to give developers of new GWCs the freedom to do asynchronous operations, although this was not needed in the context of this thesis. The `_currentNode` is then set to the node returned by the GWC, as it can be seen in Listing 6.4, Line 10. Finally, the method is called recursively.

```
1  _process(){
2   ...
3   case "Gateway":
4    let splitStackLength = this._splitStack.length;
5    try{
6     //Code relevant for logging is left out
7     let comp = new GatewayComponentDealer.comps[this._currentNode.
          executableComponent];
8     comp.evalNode(this._currentNode, this, this._currentNode.
          executableComponent, this._options)
9     .then(nextNode =>{
10     this.currentNode = nextNode;
11     this._process();
12   }
13  ...
14 }
```

Listing 6.4: Excerpt of the `process` method of the `Main Unit`

Special attention was paid to error management. In case of an error, various actions are carried out. First, it must be determined if the error occurred while trying to generate the GWC or during the method call. To do this the `_splitStack` array length has been stored temporarily as seen in Listing 6.4, Line 4. Since the GWCs push themselves on the `_splitStack` array upon creation, it is possible to check the momentary length of the array and compare it to the previously stored one. If the length has increased, the error, therefore, occured during the method call. The generated GWC is then removed from the array, to prevent further errors later on.

The next step is to try to replace the failed component. This can result in process models

being executed correctly despite an error. Nevertheless, every error is logged and can be viewed later. First it must be determined whether the failed GWC is a `split` or a `join`. This is done by some logical checks, for example by checking if there is any element on the `_splitStack`. If not, due to the process model's block structure, it can be assumed that the failed GWC was a `split`. These checks are arranged in such a way, that either a `split` GWC or a `join` GWC is assumed at the end.

If a `join` is assumed, a default `join` GWC is created and executed, just like any other GWC, which will return a node to proceed, using the corresponding `split` GWC on the `_splitStack`. However, if a `split` is assumed, a corresponding default `split` GWC is created and pushed on the `_splitStack`. This is done to avoid errors that would occur if later on the associated `join` GWC is created and executed. In this case the node to proceed with, is the first object of the `_currentNode's nextNode` array (cf. Figure 5.2), hereafter referred to as skipping the node. In case that additional errors occur during the replacement, the previously mentioned methods `_populateResults` and `shutDown` will be called and this method terminates.

The other possibility of a valid node type is the `Node`. In this case the method checks if the `content` of this is empty. This `content`, thereby, is the array that contains the objects that have to be processed by custom EXCs. In case the `content` array is empty, the `_currentNode` is skipped. Otherwise the objects inside the `content` array need to be processed. In order to do this, the `_currentNode` has to be passed to the RTM. This can be done by simply calling the correct method called `execute` of the previously saved reference `_rtmComponent` to the RTM. After calling the RTM's method `execute`, this method of the `Main Unit` is finished and will get called again as soon as the RTM terminates.
In case the `_currentNode` type is neither `Gateway` nor `Node`, this method will try to skip the `_currentNode` or shut down if this is not possible.

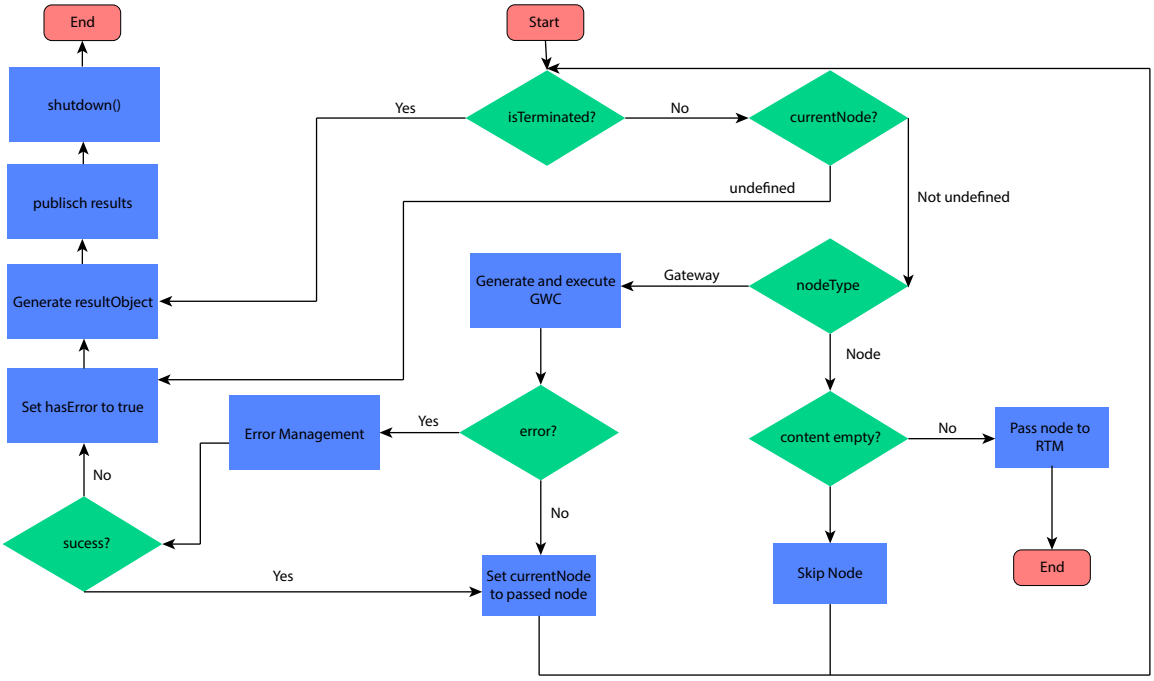

Figure 6.3: Flow Chart of the `_process` method

RtmFinished:

In order to get notified when the RTM finished, and continue processing the model with the `process` method, the `_rtmFinished` method is called by the RTM upon completion.

```
1  _rtmFinished(rtmStatus){
2   //rtm status =
3   // { finished: boolean, processedNodeType: string}
4   ...
5   //logging is left out as always
6   if(rtmStatus.processedNodeType == "End"){
7    this._isTerminated = true;
8   }
9   if(this._currentNode.type == "Node" && !(this._currentNode.category == "
        dontAddToHistory")){
10   this._history,push(this._currentNode);
11  }
12  ...
13 }
```

Listing 6.5: Excerpt of the `_rtmFinished` method of the `Main Unit`

The method starts of, by checking if the last processed node `type` was an “End”. If this is the case the property `_isTerminated` is set to `true`. The next thing it does is to check if the processed node should be added to the history. This needs to be done because it is possible to use pseudo elements that are generated by GWCs. Of course, these do not belong to the actual original model and, therefore, should not be added to the history. This for example is used on the `AND` GWC, where the user needs a graphical interface to decide, which path he wants to run through next. So in order to allow this, a pseudo node is generated and returned by the component to the `Main Unit`. This newly generated node’s category is marked as „dontAddToHistory“.
After both checks are completed the `_currentNode` is skipped and this method calls `_process` again.

Shutdown:

This method sets every property of the `Main Unit` to default values and destroys the RTM.

```
1 _shutDown(){
2  try{
3   this._rtmComponent.killedByEngine = true;
4   this._navCtrl.removeView(this._rtmViewController);
5   this.clear()
6  } catch(error)
7  ...
8 }
```

Listing 6.6: Excerpt of the `_shutDown` method of the `Main Unit`

The RTM has a closure prevention that is explained further in Subsection 6.2.2. In order to close the RTM in a controlled manner, without bothering the user, the RTM's `killedByEngine` property has to be set to `true` first. After that the RTM can be destroyed by using the build-in method `removeView`.

PopulateResults

This method is used to create an object, which contains the results and additional information about the processed instance of the model (e.g., logs, history). The parameters of this function are briefly explained below.

finished: A boolean that indicates whether the executed instance has been finished or interrupted.

hasError: A boolean indicating whether the interruption occured due to an error.

This method behaves differently depending on the `finished` parameter, storing different information inside the `resultObject`. In case `finished` is set to `true`, only information relevant to the results are stored, omitting runtime information.
This starts by setting the `nextNode` property of every entry in the `_history` to `null` in order to lower the required memory space as this information is superfluous (cf. Listing 6.7, Line 4). This is achieved by using the `forEach` method, a build-in method of TypeScript, that calls the passed callback function one time for each element present in the array, in ascending index order. The `forEach` method does not directly modify the

array, but in this case the callback function does [25]. Subsequently, the `resultObject` is generated, containing the actual payload (e.g., logs, results). The `resultObject` is then wrapped in a parent object with metadata (cf. Figure 6.4). The parent object is then published with the `Event` service, explained in the beginning of this section. In this case, the "engineClosed" channel is used. This allows any service or part of the application to subscribe to this channel and receive the published information. In this case the *SqlStorage-Service* subscribed to this channel and stores these objects automatically, depending on the information contained.

```
1  _populateResults(finished, hasError){
2   if(finished === true){
3    let finishedHistory = JSON.parse(JSON.stringify(this._history));
4    finishedHistory.forEach(entry => entry.nextNode = null);
5    let resultObject = {
6     processLog: this._processLog,
7     componentsLog: this._componentsLog,
8     ...
9    }
10   this.events.publish("engineClosed", {
11    isFinished: true,
12    hasError: hasError,
13    instanceId: this.instanceId,
14    payload: resultObject
15   }
16   ...
17  }
18 }
```

Listing 6.7: Excerpt of the `_populateResults` method of the `Main Unit`

The second option is executed when `finished` is set to `false`. This procedure is very similar to the first one, but with some exceptions. First of all, the generated `resultObject` contains all the previously mentioned informations, but in addition also runtime informations. This is necessary in order to be able to load and execute the saved instance later. The problem that occurs here is that components contained in the `_splitStack` have a reference to the `Main Unit`. This prevents this objects from getting stringified, causing a *Circular Structure* error. In order to avoid this error,

the references to the `Main Unit` are set to `null`. This object is then wrapped and published in the same parent object as before.

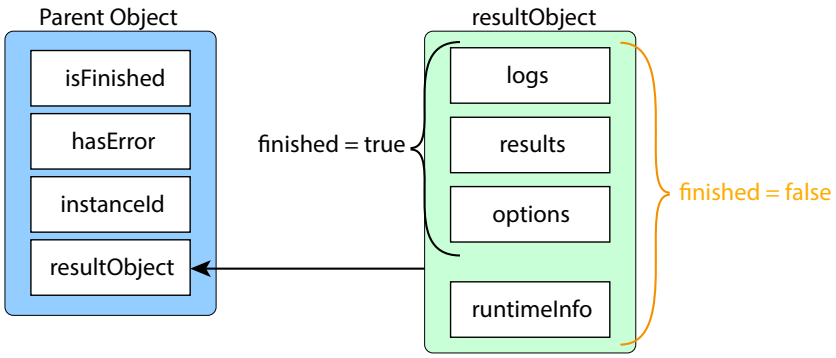

Figure 6.4: Structure of a Published Object

6.2.2 Runtime Manager

The `Runtime Manager` (RTM) is configured as a *Page*[2] component. The reason for this is that a view is required, to show the user information. First, the relevant properties of the RTM are discussed. The properties that will be set with values coming from the `Main Unit` are marked as *passed*.

processing: A boolean property. If there is no content to be displayed, this property is set to `true` and thus a graphical element is shown to the user, indicating that background processes are running.

engine: (*Passed*) The reference to the `Main Unit`. Needed for communication.

options: (*Passed*) An array that gets passed down to the PCCs, which can differ, depending on the use case (e.g., selecting a specific language).

[2] A *Page* component is a component that has its own view.

factories:	A `key/value` map. The `keys` are the identifiers of EXCs, while the value is the corresponding *ComponentFactory* that allows to dynamically generate this component.
compRefHolder:	Is an array that contains references to the dynamically generated components.
killedByEngine:	A boolean value that determines if the attempt to close this page is being performed by the `Main Unit` or by the user.
log:	(*Passed*) The passed logger class from the `Main Unit` to allow logging.
componentContainer:	This is the *ViewContainerRef*. This allows to display the dynamically generated EXCs to the user. It works much like an array that contains different view elements.

When the RTM is created, it will create a *ComponentFactory* for every `key/value` pair found in the `ExecutableComponentDealer`, and store it in the `factories` map. A *ComponentFactory* is an object provided by Angular that allows to create components dinamically, just like classes. However, each *ComponentFactory* can only create one type of component. Therefore, a separate *ComponentFactory* must be created for each EXC contained in the `ExecutableComponentDealer`. Additionally, the passed properties from the `Main Unit` will be set during the *IonViewDidLoad* cycle. This is a lifecycle provided by Ionic that runs when the page is loaded for the first time [26]. The *componentFactoryResolver*, seen in Listing 6.8, is a build-in service, which allows the creation of *ComponentFactories*.

```
1 this.factories = {};
2 for (let key in ExecutableComponentDealer.comps){
3   if(ExecutableComponentDealer.comps.hasOwnProperty(key)){
4     this.factories[key] = this.componentFactoryResolver.
          resolveComponentFactory(ExecutableComponentDealer.comps[key]);
5   }
```

Listing 6.8: Excerpt of the `constructor` method from the RTM

Execute:

The `execute` method is the only method called by the `Main Unit`. The method expects a node as parameter. This node than has a property called `executableComponent`, which contains the identifier of the EXC that can process the information of this particular node. The identifier is than used as `key`, to find the corresponding *ComponentFactory* inside the `factories` map and generate a new instance of this component, pushing it on the `compRefHolder`. In case the EXC has a user interface, `processing` is set to `false`. By doing this, the EXC's view will be rendered inside the RTM's view.
According to the implemented interface (cf. Section 5.5), every EXC has to implement a method called `execute`. After the corresponding EXC has been created, this method is called. Among other information, the currently processed node is passed to the EXC, as it can be seen in Listing 6.9, Line 8.
After this method has been called, the RTM's `execute` method is finished. The RTM's `exeCompfinished` method will be called when the EXC has finished.

```
1  execute(groupNode){
2   //logging left out as always
3   try{
4   let currentIndex = this.compRefHolder.push(this.componentContainer.
        createComponent(this.factories[groupNode.executableComponent]))-1;
5   if(this.compRefHolder[currentIndex].instance.hasUI){
6    this.processing = false;
7   }
8   this.comprefHolder[currentIndex].instance.execute(groupNode, this.engine,
        this, this.options);
9   }
10  ...
11 }
```

Listing 6.9: Excerpt of the `constructor` method from the RTM

Two possible errors can occur during this method, one during the creation of the EXC and the other one when calling its `execute` method. This can be differentiated, but only affects the logging. In both cases the RTM's `exeCompfinished` method will be called.

ExeCompfinished:

This method clears the view, sets all relevant properties to their default values, and notifies the `Main Unit` of completion, by calling the `Main Unit's _rtmFinished` method. The `exeCompfinished` method is usually called by the previously generated EXC.

```
exeCompfinished(nodeType){
 //logging left out as always
 this.processing = true;
 this.compRefHolder = [];
 this.componentContainer.clear();
 this.engine._rtmFinished({
        finished:true,
        processedNodeType: nodeType});
}
```

Listing 6.10: `exeCompfinished` method of the RTM

nodeType: A string value that defines the type of the node. Is used currently used to recognize the *End* node.

ionViewCanLeave:

This method is a so called *Guard* that can be used to check permissions before a page can leave [26]. This method is called automatically and must return a boolean value. In this case, the property `killedByEngine` is checked. If it is `true`, the page will close with no further actions. This is the case when the RTM is closed by the `Main Unit`. If the value is set to `false`, however, the user must confirm that he wants to leave the page. If the user rejects, the page will not leave. Upon proceeding, an object notifying the `Main Unit` about interruption, is published, using the previously mentioned, “interrupt” channel. Additionally, the RTM resets to its default state and `killedByEngine` is set to `true` to not trigger the `ngOnDestroy` check.

ngOnDestroy:

This is a “ lifecycle hook that is called when a directive, pipe or service is destroyed” [27]. Here the RTM is set to its default state, but additionally the `killedByEngine`

property is checked and if it's `false`, the *Event* notifying the `Main Unit` about closing, is published. This is done to avoid unpredictable errors and the loss of progress of this instance when the RTM is closed by something that can bypass the `ionViewCanLeave` guard (e.g., a crash).

6.2.3 Executable Component (PaperPage)

`Executable Components` (EXCs) are generated by the RTM. Their view (if existent) is rendered inside the RTM's view. Each EXC can be implemented according to the use case, but must meet the requirements of the defined interface (cf. Section 5.5). In the context of this thesis, the specific EXC called "PaperPage" is introduced, which happens to be very similar to the RTM.
The `contructor` method will generate factories in the same way the RTM does, but using the `ComponentDealer` instead of the `ExecutableComponentDealer`.

Execute:
The RTM just calls one method of every EXC, which is `execute`. The node to process is passed on this method, as well as a reference to the RTM, the `Main Unit`, and the `options` array that can be used for various things as previously mentioned.

Like the RTM, this method generates and embeds components in his own `componentContainer`, the same way the RTM does. It will generate a PCC for every element inside the node's `content` array. Every element possesses a property named `questionType`, determining which PCC is created. The PCCs used in the context of this thesis, have also implemented an interface. According to this interface, every created PCC has the `render` method, which gets called after creation. PCCs, which encounter errors during the method call or upon creation, are either removed or skipped. If no components could be generated, the `next` method of "PaperPage" is called, otherwise this method ends immediately.

Next:
This method will usually be triggered by a button click. The first thing it does is call the method `validatePage`, checking if every generated component is finished and ready to be closed (cf. Listing 6.11).

```
validatePage(){
 //logging left out as always
 let valid = true;
 for (let component of this.compRefHolder){
  valid = component.instance.isFinished() && valid;
 }
 return valid;
}
```

Listing 6.11: `validatePage` method of the PaperPage

If this is the case, it will remove all generated components and clear its properties in the same way the RTM does. After that, the RTM's `exeCompfinished` method is called, notifying the RTM about completion.

6.2.4 Logging

Logging is an important part of the engine, especially of the core part. It is also strongly integrated in the implemented EXCs and PCCs, but this may vary due to the interchangeability of these.
There are three logger classes, each with its own log. But since they all implement the same interface and are implemented very similarly, in this case just the "ProcessLogger" class is introduced.
The class has two properties. One is the array to log, called `logArray`, and the other is the `logOptions` object, which allows for different levels of logging. Both are set with the `setLogObject` method. Besides that, the logger offers four logging methods that also define the level to log on (`debug`, `warn`, `error`, `info`). All four methods offer the same functionality and are nearly identical. The methods can be summarized, but this way it allows the developer a better overview while working with this class.
So, as seen in Listing 6.12 the methods are very simplistic. The `if-clause` , checks if logging is wanted at this level. Otherwise the `emitLogObject` method is called in all four methods.

```
1 debug(source, msg, id, action){
2   if(this.logOptions.debug === false) return;
3   this.emitLogObject("DEBUG", source, msg, id, action);
4 }
```

Listing 6.12: `debug` method of the ProcessLogger

The `emitLogObject` method will than create an object. The object contains all the passed information as well as newly added ones like the time stamp. This object is then pushed onto the `logArray` and the method terminates.

7

Summary

The goal of this thesis was to develop an extendable lightweight process engine using cross-platform technologies. The main goal of the engine was to support the process model generated by the QuestionSys framework so that the engine could be used in the client-side application. During development, it was important that executing components are extendable and exchangeable, in order to be able to support further developments of the QuestionSys framework. This feature also allows the use of this engine for other use cases besides the QuestionSys framework. Ultimately, due to the technology used, the developed engine can be used on all major mobile operating systems (Android, iOS, Windows) without a reimplementation.

To achieve this, the requirements for the engine were first defined in Chapter 4. In the foreground was the execution of process models with complex flow logic, as well as the logging of these. The collected data should also be organized and secured. Furthermore, the engine should be extendable and able to respond to unexpected errors.

Based on the requirements of the engine, a concept could be developed in Chapter 5. For this purpose, a process model specially developed for the engine was first designed, so that questionnaire models of the QuestionSys framework could be mapped on. The engine itself was split into two parts, the core part and the exchangeable part. In addition, interfaces have been defined to allow the core part to interact with exchangeable components. Finally, a concept for logging and the collected data was defined.

After the architecture of the engine was developed, the engine could be implemented. Implementation took place with Ionic 2, Angular 4 and Cordova, all of which are very current, popular and constantly evolving cross-platform technologies. For this purpose,

Chapter 6 presented the most important aspects of the technologies used, as well as the main points of implementation.

7.1 Outlook

Up to now, the engine developed could not be tested and further developed long enough to make solid statements about necessary changes. Especially since other possible questions and node types can be added to the QuestionSys framework, such as questions that require pairing to external devices via bluetooth (e.g., blood glucose meter). Nevertheless, some points can be recognized on which further work can be done.

Currently three different `gateways` are implemented, `AND`, `XOR` and `LOOP`, where the `AND gateway` is not an `AND` in the conventional sense, i. e. all paths are executed in parallel. The implemented variant only allows a sequential execution of all paths, where the sequence is determined by the user. An implementation of a conventional `AND gateway` would be the next logical step and could also make sense in the QuestionSys context.

Furthermore, it is currently not possible to jump to a previous node unless it is in a `LOOP`. However, this could be useful in the context of QuestionSys, for example, to change a previous answer or to read a question again.

Finally, it is possible to add further modules or functions similar to the work presented in Chapter 3. A library or a module that transforms different process models (e.g., BPMN 2.0) to that of the engine would be conceivable. So far this is only available for the QuestionSys models. It would also be possible to add a component for analyzing collected data. This component could be based on previously defined rules and evaluate certain data on the basis of these. A module of this kind could, for example, quickly suggest or prohibit drugs or treatment methods based on the collected data of questionnaire that is used in a hospital.

The best way to further develop the engine, however, is to use the application in which the engine is embedded in real scenarios. This way, the users can work with the applica-

tion and the engine for a longer period of time and can give better feedback for further development.

Bibliography

[1] Boulos, M.N.K., Wheeler, S., Tavares, C., Jones, R.: How smartphones are changing the face of mobile and participatory healthcare: an overview, with example from eCAALYX. Biomedical engineering online **10** (2011) 24

[2] Monitor, T.C.: Anteile der einzelnen Gerätetypen an der Internetnutzungsdauer in Deutschland in den Jahren 2013 bis 2016. https://de.statista.com/statistik/daten/studie/455003/umfrage/anteile-der-geraetetypen-an-der-internetnutzungsdauer/ (2016) Accessed: 2018-02-20.

[3] Schmalen, K.: IDC-Studie: Deutsche Unternehmen setzten auf mobile Apps zur Verbesserung ihrer Geschäftsprozesse. http://idc.de/de/ueber-idc/press-center/56517-idc-studie-deutsche-unternehmen-setzen-auf-mobile-apps-zur-verbesserung-ihrer-geschaftsprozesse (2013) Accessed: 2018-02-20.

[4] Schobel, J., Schickler, M., Pryss, R., Maier, F., Reichert, M.: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps, Barcelona, Spain, April 3-5, 2014, pp. 371-382.

[5] Fritz, F., Balhorn, S., Riek, M., Breil, B., Dugas, M.: Qualitative and quantitative evaluation of EHR-integrated mobile patient questionnaires regarding usability and cost-efficiency. International Journal of Medical Informatics **81** (2012) 303–313

[6] Schobel, J., Pryss, R., Reichert, M.: Using Smart Mobile Devices for Collecting Structured Data in Clinical Trials: Results from a Large-Scale Case Study. In: Computer-Based Medical Systems (CBMS), 2015 IEEE 28th International Symposium on, IEEE (2015) 13–18

[7] Schobel, J., Schickler, M., Pryss, R., Reichert, M.: Process-Driven Data Collection with Smart Mobile Devices. In: International Conference on Web Information Systems and Technologies, Springer (2014) 347–362

[8] statista.com: Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/ (2017) Accessed: 2018-02-20.

[9] Institute of Databases and Information Systems, University of Ulm: QuestionSys - A Generic and Flexible Questionnaire System Enabling Process-Driven Mobile Data Collection. (https://www.uni-ulm.de/in/iui-dbis/forschung/laufende-projekte/questionsys/) Accessed: 2018-02-20.

[10] Schobel, J., Pryss, R., Schickler, M., Ruf-Leuschner, M., Elbert, T., Reichert, M.: End-User Programming of Mobile Services: Empowering Domain Experts to Implement Mobile Data Collection Applications. In: Mobile Services (MS), 2016 IEEE International Conference on, IEEE (2016) 1–8

[11] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: A Lightweight Process Engine for Enabling Advanced Mobile Applications. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer (2016) 552–569

[12] Tiemuer, A., et al.: Mobile Business Processes: Challenges, Opportunities and Effect. (2017)

[13] RedHat: jBPM. (http://www.jbpm.org) Accessed: 2018-02-20.

[14] RedHat: jBPM Overview. (https://docs.jboss.org/jbpm/v6.0/userguide/jBPMOverview.html) Accessed: 2018-02-20.

[15] RedHat: jBPM Core Engine API. (https://docs.jboss.org/jbpm/v6.0/userguide/jBPMCoreEngine.html) Accessed: 2018-02-20.

[16] Pryss, R., Tiedeken, J., Kreher, U., Reichert, M.: Towards Flexible Process Support on Mobile Devices. In: Forum at the Conference on Advanced Information Systems Engineering (CAiSE), Springer (2010) 150–165

[17] Wipp, W.: Workflows on Android: A Framework Supporting Business Process Execution and Rule-Based Analysis. PhD thesis, Ulm University (2016)

[18] International, E.: ECMA-404 The JSON Data Interchange Standard. (http://www.json.org/) Accessed: 2018-02-20.

[19] ionicframework.com: Build Amazing Native Apps and Progressive Web Apps with Ionic Framework and Angular. (https://ionicframework.com/framework) Accessed: 2018-02-20.

[20] Sass: Sass Documentation. (http://sass-lang.com/documentation/#sass_gem_version_inline_docs) Accessed: 2018-02-20.

[21] Microsoft: Typescript - Javascript that scales. (https://www.typescriptlang.org) Accessed: 2018-02-20.

[22] ionicframework.com: Ionic Native. (https://ionicframework.com/docs/native) Accessed: 2018-02-20.

[23] Apache Software Foundation: Architectural Overview of Cordova Platform - Apache Cordova. (http://cordova.apache.org/docs/en/latest/guide/overview/index.html) Accessed: 2018-02-20.

[24] Google: Angular Docs. (https://angular.io/guide) Accessed: 2018-02-20.

[25] Microsoft: Microsoft Docs. (https://docs.microsoft.com/en-us/scripting/javascript/reference/foreach-method-array-javascript) Accessed: 2018-02-20.

[26] ionicframework.com: Ionic API Docs. (https://ionicframework.com/docs/api) Accessed: 2018-02-20.

[27] Google: Angular Docs API. (https://angular.io/api) Accessed: 2018-02-20.

List of Figures

Name: Dimitrios Kamargiannis Matrikelnummer: 861700

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Dimitrios Kamargiannis