Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

Rule-based Evaluations for Mobile Data Collection Applications

Bachelor's thesis at Universität Ulm

Submitted by:
Daniel Rollenmiller
daniel.rollenmiller@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Johannes Schobel

2018

Version from October 9, 2018

Composition: PDF-LaTeX 2_ε

Abstract

Paper-based data collection is often directly associated with high cost and high delays. Collected data is often shipped to domain experts in order to be analyzed. This, in turn, leads to waiting times before results are available. However, in many clinical or psychological cases, e.g., with the purpose to detect potential risk factors, real time analysis would be a major improvement for all stakeholders. To achieve the latter an electronic data collection approach is needed instead of a paper-based one. A framework was developed to allow domain experts to create mobile data collection applications by themselves. This framework is shortly introduced in the beginning of this thesis. The framework makes the creation of questionnaires much more efficient. Domain experts are able to create mobile applications without needing IT expertise. Furthermore the framework allows the definition of so-called rules. Rules provide the possibility to analyze the answers of a questionnaire. To increase the efficiency of this framework further, this thesis will create a tool providing the functionality to automatically evaluate these rules with the result of the questionnaire in order to provide real time analysis and feedback. This tool is meant to be integrated into the existing framework and executed on a participant's mobile device or browser. Since rules may contain malicious code, they have to be executed in some sort of a secured context to provide safe evaluation. After comparing several different approaches, the tool was implemented with the use of the best fitting solution. In order to evaluate rules, necessary information is extracted from the questionnaire result and model and a sandbox with variables and functions build. Then a single rule is evaluated without access to any data outside its own context. Altogether this thesis implements a tool as an addition to the framework to provide safe real time analysis.

Contents

1 Introduction

Interviews and questionnaires are widespread instruments for collecting data. In several domains, for example, in clinical environments they also help to detect psychological risk factors, e.g., during pregnancy. Therefore they are an important instrument for prevention and intervention. As mentioned in [1], the interviewers are often not psychologically trained so they cannot analyze the results properly or there may not be enough personnel to allow local analysis. Paper-based data collection methods are costly and need a lot of time, for example, if the paper-based data has to be delivered to a psychologist to analyze the latter. This is obviously highly inefficient and the detection of psychological risk factors is delayed [1]. Fully automatic analysis with the use of electronic data collection approaches could provide real-time analysis of the data and faster intervention for those affected.

Electronic data collection, according to [2], can save approximately 50-60% of cost compared to paper-based data collection. However, creating electronic questionnaire needs besides the psychological or clinical knowledge, IT expertise as well. This entails some issues as stated in Section 2.1. [3] introduces an approach that allows domain experts, e.g., psychologists or study directors to simply create an electronic data collection instrument running on mobile devices without needing IT expertise. Furthermore, it allows to easily deploy the questionnaire to participants via a web service and also to analyze the answers by defining so-called rules.

This thesis will be concerned with the analysis of the results. A tool, extending the framework, will be developed that automatically evaluates conditions to analyze a questionnaire and its result. In order to meet safety requirements the evaluation of a rule will take place in some sort of a secured context. Furthermore, a predefined set of functions

and the possibility to add custom functions to the evaluator tool allows deep analysis of the result due to complex rules.
In the beginning of this thesis, Chapter 2 provides details on the underlying approach of mobile data collection applications. Furthermore, the relevant data of the questionnaire's model and result is depicted. Next the functional and non-functional requirements of the rule evaluator tool are analyzed in Chapter 3. Chapter 4 presents different approaches for evaluating condition strings, as given within a rule definition. These approaches are compared amongst each other. The one fitting the requirements the most is selected for the further development of the rule evaluator.
Details on the implementation of the tool are provided in Chapter 6. First the components of the evaluator and its structure are depicted. After that the class diagram is presented and the operating principle of the predefined and custom function sets explained. At the end of this chapter details on the creation of a safe context, as well as the safe evaluation, are provided. In the end an outlook for future development of this tool and the underlying framework itself, is presented in Chapter 7.

2 Background and Fundamentals

This chapter explains the background of this thesis, i.e, the idea of using smart mobile devices for data collection. In particular the overall framework is shortly introduced and afterwards details on the data received, provided.

2.1 Mobile Data Collection Applications

Smart mobile devices are part of almost everyone's daily life and with that in mind it makes sense to use those devices for mobile data collection, e.g., online surveys. There are a lot of reasonable benefits, like high-quality data and less paperwork that needs to be evaluated and archived manually. With the usage of mobile devices, surveys can be done automatically. However, a study director for example, has no or only little IT expertise and is most likely unable to create a mobile application himself (cf. [4]). Therefore, there is the need for IT experts to implement and maintain a mobile application for exactly that purpose. On the implementation side new challenges, like privacy issues or issues due to different operating systems arise. Additionally there may also be communication issues between a study director and IT experts because of them being experts in very different domains, having their own language.
These problems were addressed in [3], which is the basis for this thesis. The researchers experienced high maintenance cost in some of their projects due to domain experts never being fully satisfied. They came to the conclusion that it would be best if domain experts are able to create mobile data collection applications themselves without needing IT expertise.
Therefore, their aim was to create a model-driven generic framework, named Question-

Sys, that empowers domain experts to simply create mobile data collection applications by describing a questionnaire as a process model. The latter is then run on a smart mobile device by a lightweight process engine (cf. [5]).
For their architecture they took an process-driven approach relying on the ADEPT2 process model, because a process model can be mapped easily to a questionnaire, e.g., a page can be mapped to a process activity and a question to a process data element.
First collection instruments can be created with a configurator where for example the flow logic, pages and data elements are specified. Data elements represent questions, which are connected to pages, which structure the questionnaire and represent single screens on a mobile device. The configurator also offers the possibility to skip questions or pages by navigation operations for example if the answer to a previous questions makes a future question unnecessary. Also rules can be defined, whose evaluation will be the main part of this thesis.
Second after modeling the questionnaire, it should be executed and for that purpose there was the need of a process engine for mobile devices, which is able to render the model locally and allows for flexible applications.
Their third goal was to relieve IT experts. Therefore all data is exchanged and stored as XML documents and automatically managed by a Web Service. This service autonomously performs the process of creating a questionnaire by taking the questionnaire model created with the configurator by a domain expert and automatically deploying it to smart mobile devices. After that the process engine, running on the device, renders and executes the questionnaire. Then the service automatically captures and stores information about the execution in log files for evaluation and analysis purposes. However, when the Web Service provides the result back to an analyst they must be anonymized and encrypted due to privacy regulations. This procedure is illustrated in Figure 2.1 (cf. [3]). The usage of a standard document format and a Web Service that does everything automatically makes, for example, the deployment of a new version possible without the need of IT experts.
However, times have changed since 2016 and today not XML is used as a document format, but JSON.

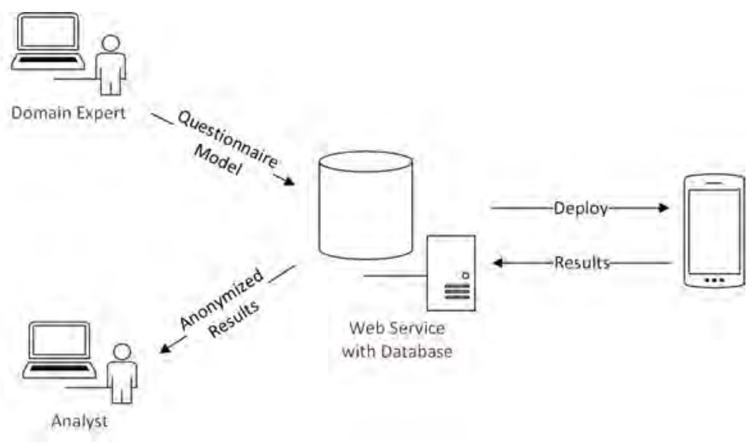


Figure 2.1: Workflow of the automatic Web Service

2.2 The Questionnaire Model

This section describes the structure of the JSON files that are created by the configurator and contain the model that is then rendered on the client-side on mobile devices as well as rules. In Listing A.1 you can see the basic structure of the overall model. It starts with some meta information for example contact information of the author, name of the questionnaire, available languages, etc. However, this information is not necessary for this thesis and therefore not further explained here. Next in the model file is the model itself which is of course the most important part.
Listing A.2 provides an example of how the model object could look like. It consists of a *nodeDataArray* where the questions are defined. In the given listing there is an example for a *FreeDate* question where the user simply has to enter his or her birth date. A question is uniquely identified by its key and contains a lot of information, such as the question itself that may be given in multiple languages within the element object, a name, the question type, etc. However, the only information needed for rule evaluation are the types of questions. Section 2.3 shows more details on the existing question types.
The most important part of the model is located within the *rules* array (cf. Listing A.3). These rules are those to be evaluated after finishing the questionnaire and, therefore, the main element of this thesis. A rule, in turn, is uniquely identified by a unique *key*, a *name*, a *conditionString*, a *variablesMapping* and a *positive* and *negative* result. The

conditionString, as its name implies, is a string that represents a Boolean condition, which can consist of variables, numbers, strings, function calls, mathematical operators, logical operators and comparison operators as used in Javascript. Variables somehow need to be resolved to constant values in order to evaluate the condition and for this purpose the *variablesMapping* is needed. The latter is an array that consists of multiple objects that contain the *variableName* and the *questionId* that identifies the question containing the associated value. However, with the information given in the variables mapping it is possible to retrieve the value of the variable from the result. This allows to evaluate the condition which results in a Boolean value or an error.
Positive and *Negative* contain a headline and a description that can be given in multiple languages and used after the evaluation, whether it was true or false, to give feedback to the user.

2.3 The Questionnaire's Result

The result contains a lot of logging information as well as the evaluated results for the questions. The most important part of the result object is the *payload* object (cf. Listing A.4). The *payload* comprises several arrays: The *processLog* and *componentsLog* contain logging information with timestamps for any actions happening during the execution of the questionnaire.
The results object contains the final answers given by the user. The id of a question references to the question definition within the model. The relations between the model and result is illustrated in Figure 2.2. Within the result the question id is mapped to an array that contains an object for every iteration. In the example there is only one iteration and, therefore, only one object within the array. This object consists of a timestamp, the *name* and *exportname* as well as the count of the *iteration* and most important a *value* field. The *value* field can be a single value or a object that maps keys to value or keys to keys depending on the question types.
In Listing A.5 some examples for question results are shown. In the first result there is a simple free float question. In this case the question id is enough to identify a result. The second result is of a single choice question where keys from every possible choice

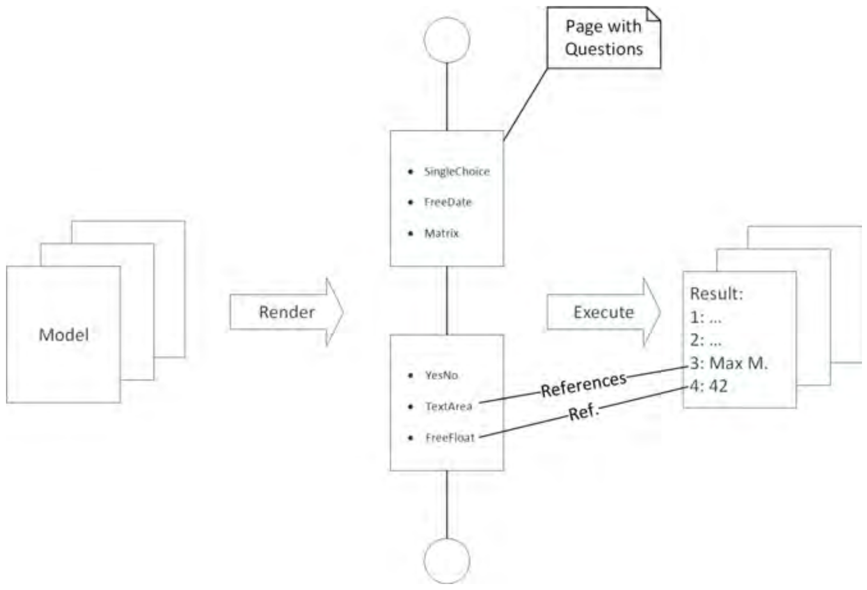


Figure 2.2: Relation between Model and Result

are mapped to either true or false according to what answer was checked or not. Due to multiple input options the question id is not enough to retrieve a single value from this kind of question. There are also other question types that map keys to other datatypes as shown in other examples, however the key-value principle stays the same. In the third example the result is a key mapped to another key, which can only result from a matrix question. The first key identifies the row of the matrix and the second key identifies the column, which means that at this row and column in the matrix the participant checked an answer. The examples below the matrix question slightly differ from the examples mentioned above, but the principles remain the same. How these different values are resolved when evaluating a rule will be explained in 6.4.
While the *results* contain the final answers for each question, the *dataHistory* includes the steps the participant took towards answering the question. For every question it contains a *history* array where any change to the answer options, like checking a check-box or entering some text into a textfield are saved with the (partial) value and a time stamp. With that it is possible to understand the history of a question and may even allow analyzing the steps the user did while answering the question.

3 Requirements Analysis

The following chapter describes requirements in the context of the rule evaluator. The rule evaluator is a tool that allows to evaluate the rules defined in the model within the context of a result. The evaluator tool extends an engine that is part of another project about mobile data collection applications. This helps with analyzing the results by evaluating rules and allows the user to receive feedback immediately. The exact functional requirements for the rule evaluator as well as non-functional requirements are explained below.

3.1 Functional Requirements

Req.#	Requirement Description	Priority
1	The tool should analyze the results of the questionnaire according to the defined rules.	High
2	It must be possible to integrate and run the tool in a web application.	Medium
3	The evaluator should primarily be available in a mobile application written with angular/ionic.	High
4	Evaluation of malicious code should not endanger the application. Therefore there should be no access to the global context when evaluating condition strings.	High

5	Human mistakes while defining rules or answering the questionnaire, e.g., wrong syntax should not result in a failure of the application.	High
6	The evaluator needs to take and parse the questionnaire results to a simpler result context with only necessary information.	High
7	The evaluator needs to receive the questionnaire model and extract the rules and all other needed information from it.	High
8	From the result and model the evaluator needs to create a context for every rule that contains only the variable mappings needed for evaluating this rule.	High
9	From a context of a rule, it should not be possible to access the context of another rule or any information not directly related.	High
10	The variable mappings defined for each rule need to be resolved. A variable is defined by the question id, that refers to one question and a value array that identifies the value if more than one are available. How the value array looks like is determined by the question types: • Zero Elements: *FreeFloat*, *FreeDate*, *SliderSingle* and *FreeTextArea* questions • One Element: *Single-* and *MultipeChoice*, *YesNo*, *Distribution*, *Ranking*, *SliderRange* and *ButtonGrid* questions • Two Elements: *Matrix* question	High
11	The tool should be able to evaluate the condition string, which, if no error exists, results in a Boolean value.	High
12	After evaluation the evaluator should return a evaluation result for every rule with its id, whether it was true or false as well as the positive and negative fields.	High

13	If an error occurs or some data is missing an appropriate result with the cause of the error should be returned instead of true or false.	Medium
14	A predefined function set with common functions for math, dates, string and shapes should be available by default.	High
15	It should be possible to inject further custom functions. It should be possible to override predefined functions.	Low
16	A timeout should be available if the evaluation takes too long (i.e., if a function contains an infinite loop).	Low
17	The predefined function set should contain important mathematical functions, such as computing the sum, maximum value or minimum value of a series of numbers as well as the mean or median.	High
18	Functions for strings, e.g., if a string contains a certain substring or a sentence contains a certain word, etc. should be predefined.	Medium
19	The moment library for date comparison and arithmetic should be available as well as some predefined functions to make the usage of moment more intuitive.	Low
20	To allow working with shapes, e.g., checking if a point is within a bounding box the function set should contain a simple function to check whether a point is within a polygon or not.	Low

3.2 Non-Functional Requirements

Data security

The tool needs to ensure that the data is handled confidentially, which specifically means that information is only available when needed and cannot be accessed from outside the specific rule context.

Maintainability

Since the main project of the configurator is still in development, adjustments may also be required for the evaluator, e.g., when the model or result changes. Therefore, the code must be easy to read, to understand and to maintain in order to implement new requirements. To ensure consistency the *Airbnb JavaScript Style Guide*[1] is adapted.

Platform compatibility

The tool must be available on modern mobile platforms, such as iOS 8 or newer and Android 4.4.X or newer, as well as in modern web browsers, such as Safari, Chrome, Firefox and Edge.

Robustness

The tool must be robust to human errors. In case of errors such as wrong rule definitions with wrong syntax or functions and variables being undefined the evaluator must not crash or stall the application.

[1]`https://github.com/airbnb/javascript`

Test coverage

All components of the evaluator must be covered by test cases to ensure that the code is working as intended. As a minimal guideline the test cases should cover about 80% of written code.

4 Related Approaches

A rule, thereby, consists of a condition string, which is a JavaScript expression that evaluates to true or false, and variables that map some identifier to the value of a question. This chapter deals with the evaluation of boolean conditions given as a string. Several different approaches will be introduced and differentiated into categories.

4.1 Expression Parser

This section contains different approaches that use the JavaScript compiler or an own parser and evaluator for evaluation of a string.

4.1.1 Javascript's Eval

When thinking about running an expression given as a string the first that comes to mind is javascript's built-in function *eval()*. Eval is a function that takes a string, calls the JavaScript compiler and evaluates the string as an expression. Even if this sounds good so far, eval is often referred to being "evil" and its usage must be considered closely. *Eval* isn't necessarily evil. It is totally fine for evaluating a string whose input is known and can be trusted. But that is exactly the problem. The condition string of a rule, as well as the answers to questions are inputs given by random users mostly without IT expertise. So there might be mistakes in, either the rule definition or an answer that leads to errors while evaluating the latter. However, this wouldn't be a serious problem unless poor implementation without any error handling techniques. But another obvious scenario would be malicious input done by the user himself with a bad purpose in mind.

The problem is that the malicious expression a user enters is not only run on his client, where it couldn't do more harm than the developer console of any browser, but on any other device. For example, a study director could write malicious rule definitions that are then run on the study participant's mobile device or if the tool is used on the server both the study director and the participant could inject malicious user inputs. Obviously *eval* is safe if the user input can be fully trusted or validated. However, in the given scenario this may not be the case, therefore, using *eval* is a security risk that cannot be taken and another approach may be needed here.

4.1.2 JavaScript Expression Language

The *JavaScript Expression Language*[1] (further referred as *Jexl*) provides an eval function that comes along with its own expression parser and evaluator. Due to having an own grammar that is different from JavaScript and therefore an own parser and evaluator it is evaluated without direct access to the JavaScript compiler or objects outside the function. *Jexl* supports all standard operators for arithmetic, logic and comparisons providing a good alternative at first sight. Another advantage of *Jexl*, due to not having access to objects and variables outside the function, is the possibility of handing over a context that are simple JavaScript objects and can be used within the expression. So for every rule a dedicated context can be created that contains the needed variables. Listing A.6 shows a simple example of *Jexl's* usage, including error handling. However, there are also some downsides of *Jexl* as it does not support custom functions. It does, however, support transforms, which act almost the same as functions (cf. Listing A.7). Although the latter almost work like functions they bear a huge disadvantage by having their own specific syntax. It may be requested to import external modules to use their functions. But since the syntax of calling transforms differs from the syntax of a function call in JavaScript every function has to be rewritten as a transform. Therefore *Jexl* is not the perfect solution and another approach is desired.

[1] `https://github.com/TomFrost/Jexl`

4.1.3 Notevil

The *notevil*[2] package offers a function called *safeEval* that evaluates JavaScript expressions similar to the built-in *eval* function but in a safe context. It parses the expression to the JavaScript Abstract Syntax Tree and then evaluates it and returns the evaluated result. As Jexl, *safeEval* is also context-based, which means that it is possible to pass a context with objects and variables for every rule. Contrary to built-in eval, *safeEval* does not have access to global objects, only to the context that is passed when calling the function, which makes it robust for evaluating untrusted user inputs. Listing A.8 shows a brief example with a simple context in Line 3 and 4, containing a simple variable *x* and a function *max* that computes the maximum of a series of numbers. As this example illustrates it is possible to simply pass functions with the context, which makes it also easy to allow other libraries, e.g., for working with dates without having to rewrite any function.
Another upside of *safeEval* is that it contains a technique that prevents infinite execution and interrupts the evaluation when a maximum amount of iterations is reached. However there is a small drawback, because the amount of iterations is predefined within the code and, therefore, cannot be set by the user individually. Still, this feature is nice to have, as it prevents an application from stalling. All in all, the notevil module provides a valuable function and depicts a good choice for evaluating rules.

4.2 Sandboxes

Another approach to executing code in a safe manner is running it within a sandbox. The following section introduces two specific sandboxed approaches.

4.2.1 NodeJS' VM

NodeJS offers a module, namely *vm* that provides the feature of a sandbox. *VM* allows to execute code without the access to global objects and the standard node library in

[2] `https://github.com/mmckegg/notevil`

a new context with its *runInNewContext* function. A short example presents a simple context with a variable and a function which is then used for evaluating a function call (cf. Listing A.9). Although standard libraries such as *console* are by default not available they can be injected into the context as well as external modules. This allows not only every rule to have its own context but also the usage of complex modules. In addition, VM's *runInNewContext* function comes with the possibility of setting options with a third parameter. One of these options is a timeout in order to terminate the execution. This helps to protect against infinite loops and, unlike notevil, it can be individually set by the user.
Although VM is considered safe so far as it cannot access global objects, this is not quite the truth. Listing A.10 shows a valid exploit that is available when the context contains not only primitives (cf. [6]). Since for this thesis it is not affordable to only use primitives, due to the need of functions, VM might not be as safe as required even if it fits perfectly otherwise.

4.2.2 VM2 - An enhancement to Node's VM

The *vm2*[3] module comes with two different sandboxes one called *VM* and the other called *NodeVM*. Both are based on NodeJS' VM module. However, they rely on the use of proxies in order to provide extra security checks to prevent code from "escaping" the sandbox, which is to some extent a problem with NodeJS' module.
As already mentioned there are two variations available. While *VM* is a simple sandbox with only the possibility of setting a context, timeout and, if necessary, a compiler like coffeescript, *NodeVM* offers a lot more options. For example, it has the possibility for allowing built-in or external modules. Even if *NodeVM* has more available options than *VM*, it misses out a timeout feature. The latter is definitely a feature that has proven its usefulness when working with untrusted user inputs. Therefore, the focus of this section will be on the simpler *VM* further on.
As the example in Listing A.11 presents, *VM* works similar as previously discussed approaches. In Line 5-8 the initialization of the sandbox is achieved with the context and

[3] `https://github.com/patriksimek/vm2`

the timeout of one second. Line 10 shows the evaluation of an expression.
Required functions from external modules can be injected into the sandbox via the context. All in all, *VM* does not change the functionality of NodeJS' VM, but due to extended security layers this module is a good choice for evaluating untrusted JavaScript expressions, in general.

4.3 Comparison

As different approaches on how to evaluate rules more or less safely were presented previously, now the pros and cons have to be weighed up. Recap that the given expressions are user inputs that cannot be trusted. Therefore JavaScript's default *eval* function is not sufficient, as access to JavaScript's compiler should not be granted.
However, there are a lot of alternatives available that need to be compared amongst each others regarding different aspects. As already mentioned, security issues with *eval* exist. *Jexl* due to having its own parser and evaluator is safer to any kind of attack with plain JavaScript than *eval*. However, if an attacker manages to inject transforms that contain malicious functions the attacker could inject infinite loops or something similar. But he will not be able to access global objects similar to Listing A.10. Therefore *Jexl* in case of safety would be a good choice.
NodesJS' *VM* has a well known vulnerability, which allows to escape the sandbox. Not even the timeout option may be able to compensate for it. This weakness is solved with the *vm2* module that offers the same functionality and adds further security layers that makes it, at least very hard, if not impossible to escape the sandbox. This solution, in turn, is not perfect as the timeout does not work for asynchronous calls, which makes it vulnerable for infinite loops. Note that an infinite loop is far less dangerous than escaping the sandbox would be. Therefore this would still be a good solution in case of security.
Last but not least *notevil* needs to be discussed in more details. It is safe to accessing global objects unless injected via context, but if JavaScript's *process* object is not injected it is safe to the weakness of NodeJS' VM. In addition, every loop calls an infinity checker, which makes it robust for any kind of infinite loop. However, there is a vulnerability known when using regular expressions as can be see in Listing A.12. This regular expression

searches for all ways to arrange the nested groups which leads to an exponential amount of time needed. So according to safety only *Jexl*, *notevil* and *vm2's VM* are the possibilities left.
Looking at the syntax, *Jexl* significantly differs from the others. Having the first parameter in front of the function separated by a pipe, seems not to be very intuitive, especially when not even having arguments to pass or a function contains another function as an argument. Relying on plain JavaScript syntax, notevil and vm2 have the advantage here. However, if thinking of making an external module available, e.g., for working with dates the major downside of *Jexl* becomes obvious. Every function would have to be rewritten and added as a transform instead of just easily injecting it with the context as it would be possible with *notevil* and *vm2*. Therefore, *Jexl* is also not considered anymore.
The following Table provides a quick comparison of the different approaches as a short reminder what was discussed so far.

	Eval	**Jexl**	**Node's vm**	**Notevil**	**Vm2's vm**
Context	Global	Context-based			
Syntax	Javascript	Own Syntax	Javascript		
Module import	With global objects	Rewritten as transforms	via context		
Protection for infinite loops	No	No	Yes	Yes	Yes
Access to global objects	Yes	Only when injected into the context[4]			

Table 4.1: Comparison of different approaches

The choice for the following realization is now between the *notevil* and *vm2* modules, since they are the safest approaches and importing functions or even whole modules is easy with injecting them into their contexts. In addition, both are very simple in their usage and both have some minor vulnerabilities regarding infinite loops. The latter, however, are not considered a big threat, since they cannot cause a lot of harm.

[4]Node's VM, however, has a known exploit to escaping the sandbox

Basically the timeout and infinity checker are solid features to make the system more robust against unwanted error from non-IT-experts. Since *vm2* is in use far more often in other projects and is still maintained, this is the module of choice for the further course of this thesis.

5 The Evaluation Data

This chapter provides details on the data required for evaluating a rule and how the latter is extracted from the given model and result (cf. Chapter 2). The questionnaire's model and result are simplified so that only necessary information are shown. Additionally, details on the final result of the evaluation are depicted.

5.1 Rule Context

For every rule associated with a questionnaire, a specific rule context is build during evaluation. This context contains only required information for evaluating a specific rule. Keep in mind the structure of rule definitions as presented in Listing A.3. For evaluation purposes only the rule identifier, its key, as well as the condition string and the variables mapping are needed. The negative and positive objects as well as the rule's name are not relevant for the evaluation itself. The complete model of the rule context is presented in Listing A.13.
This context is built for every rule and stored in an array. In every iteration a single rule is evaluated. How the sandbox is build and the condition string executed, will be explained in section 6.4. For now it is enough to know the required information from rule definitions.

5.2 Evaluation Result

The information left out for the rule context, because they were not needed for the evaluation itself, however, are necessary for the evaluation result. The positive and

negative objects contain descriptions in one or more languages that provide additional information for the participant with respect to his given answers. For example, if someone participates in a medical study and due to his answers he could suffer an addiction, information about addiction counseling can be shown. Therefore, the evaluation result contains the key and name to identify a rule as well as the positive and negative objects from the rule definitions. In addition to that a result field is added.
Listing A.14 shows an enumeration with the different results that are possible. *Evaluation.True* and *Evaluation.False* map to the corresponding Boolean values. *Evaluation.MissingData* appears as a result if some data, be it a variable or a function that are not defined, is missing. *Evaluation.Error* appears on any other errors, for example, if the condition string is not a valid JavaScript expression, or if the rule does not evaluate to a Boolean value. In order to make debugging in case of an error easier, an optional property *error* is added to the result that if the execution of a rule failed contains the error message thrown.
Listing A.15 shows the entire evaluation result. The individual evaluation results are altogether stored and returned as an array.

5.3 Result Context

The questionnaire's *results* object (cf. Listing A.5) contains vital information about the answers, which are transformed to the *result context*. A question is, first of all, defined by its unique identifier. Furthermore, a question may have one specific type, which determines how the result value looks like. For simpler questions the answer are a single value, e.g., a number or string, for more complex questions the result looks different. Details on the question types will be provided later on.
However, the result of a question can contain multiple iterations, each with different values possible. Since iterations are not yet implemented in the main application, this issues is not discussed in detail, but further mentioned in Chapter 7. However, because iterations don't make the result context more complicated, they are already represented in the context, but not used during evaluation. Listing A.16 presents the result context. The *content* array contains the value object for every iteration, which can be extracted

from the questionnaire result. Information about the timestamp, the name, etc. are not necessary for rule evaluation and, therefore, omitted.
To determine the structure of the result the type of the question is missing, which is given within the questionnaires model. Recap Listing A.2 about the model object and its *nodeDataArray*, which contains the definitions of the questions. From the element object of a question definition, the *questionType* is retrieved. Summarizing, first of all the necessary properties are extracted from the *results* object and then additional information about the type is retrieved from the model. A single result context represents the answer to a single question. These contexts are altogether stored in an array, which represents the entirety of the result.
The following Figure 5.1 illustrates the flow of the collected information. Merging the rule context and result context creates the sandbox context. After executing a rule the evaluation result is provided with the result.

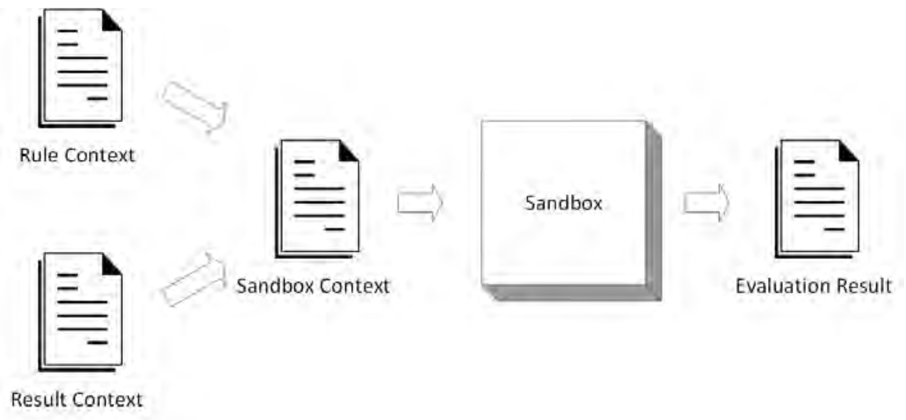


Figure 5.1: Information Flow within the Evaluator

6 Architecture and Implementation

The following chapter analyzes the components and classes to describe the architecture. Also it depicts certain implementation details as well as required dependencies for certain modules and libraries. After showing the relationship between different components and the structure of the system's classes, details on the sandbox context are provided.

6.1 The Components

The *RuleManager*, developed in the context of this thesis, can be divided into multiple components, each presenting different functions. The *QuestionnaireParser* receives the questionnaire model and result from the client, e.g., a mobile app or a web browser. Furthermore it offers functions for parsing the rule and result context as well as the evaluation result, described in the previous section. These contexts are then provided to the *RuleEvaluator* itself. Furthermore there is a predefined set of functions that are available on evaluation, since they are injected into the sandbox by default. In addition there exists the possibility to define a custom function set outside the *RuleManager* scope, that may override predefined functions, and inject it to the evaluator, as well.
The *RuleEvaluator* component resolves the variable mappings, given in the rule context and builds the sandbox context. The latter consists of variables, functions and a timeout. Next, the *Sandbox* is initialized with its context. Subsequently, the rule condition is evaluated within the sandbox in a safe manner. An overview of the basic components and their relations is shown in Figure 6.1

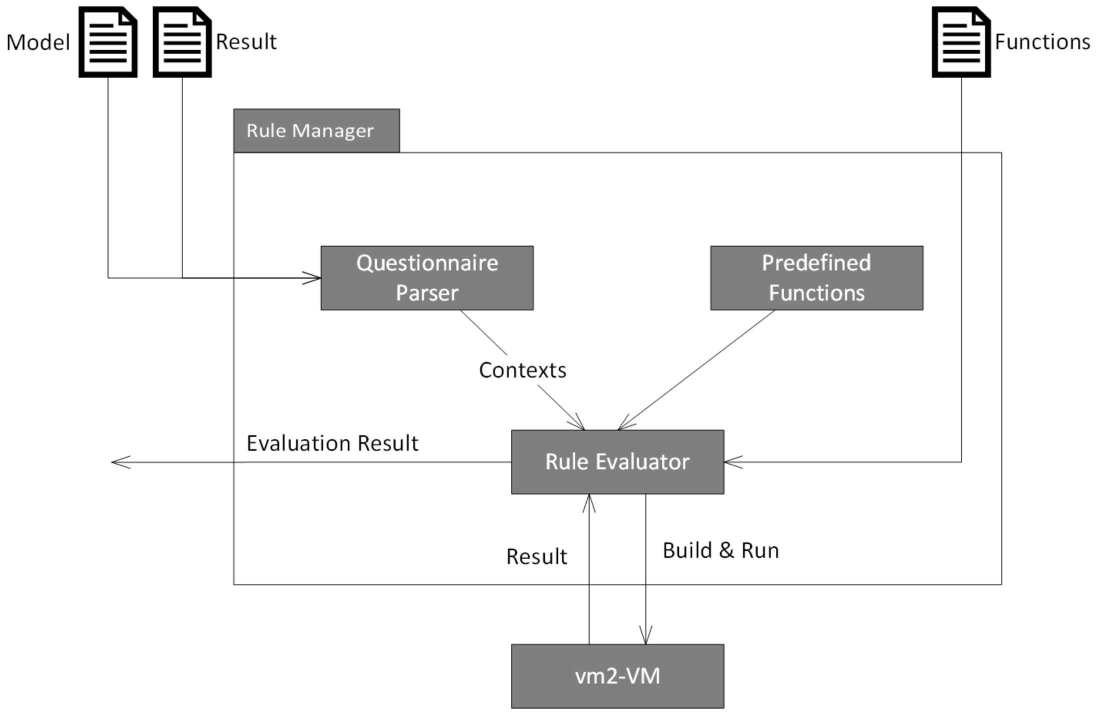


Figure 6.1: The Components of the Rule Evaluator

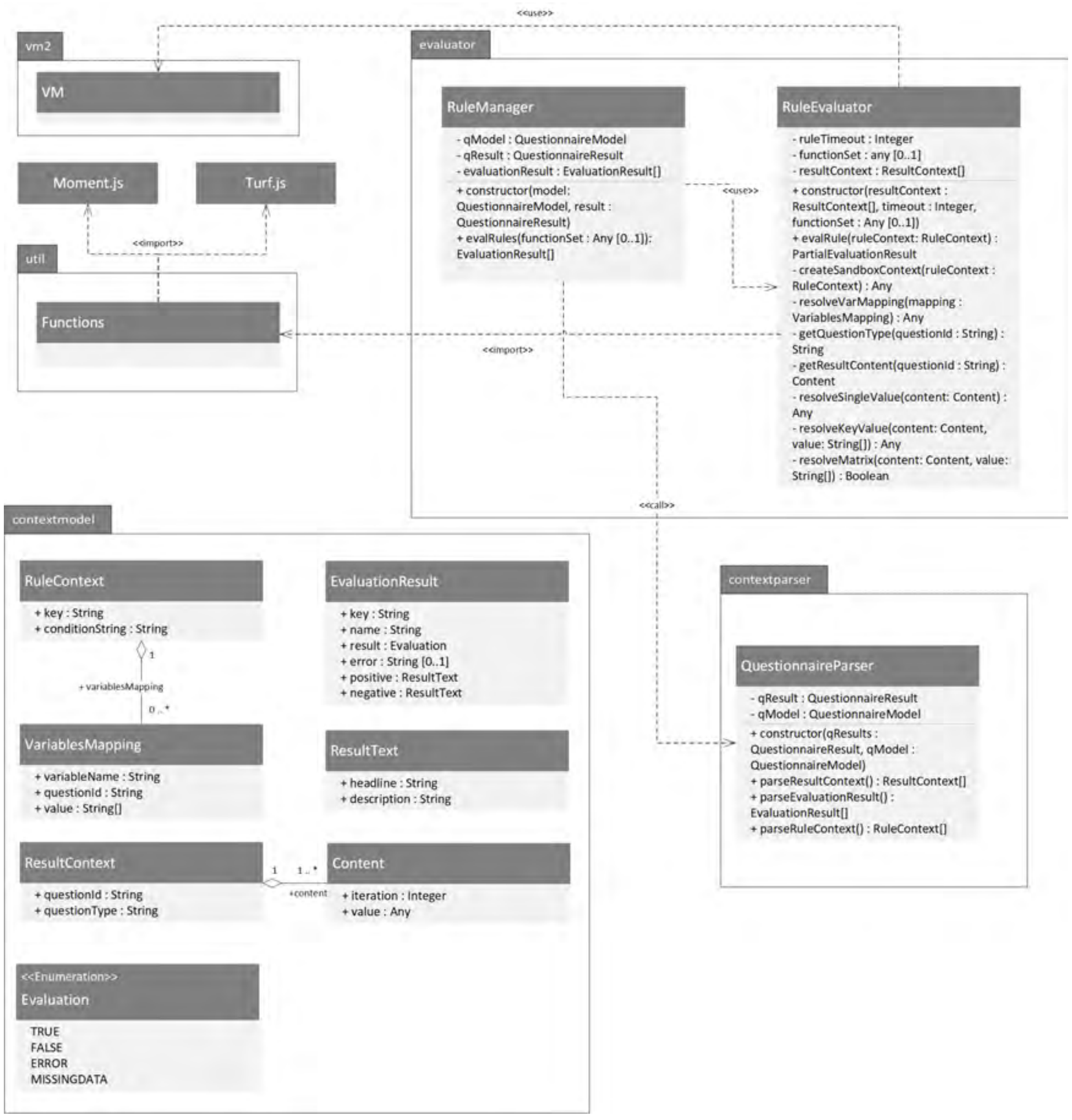
6.2 Class Diagram

The following section provides details on the implementation and discusses important prerequisites. The tool will extend an existing *Ionic* project. *Ionic* is a framework for creating mobile applications for multiple mobile platforms, such as iOS and Android, using web technologies. It is build upon Google's *Angular* framework [7]. *Angular*, however, is a framework for creating web applications and therefore based on the web technologies HTML, CSS and JavaScript [8]. Instead of JavaScript, *Angular* recommends *TypeScript* as the programming language, which implements features of the ECMAScript 6 standard [9]. Since the Angular framework supports Typescript and the tool is likely to be extended from time to time, it makes sense to write the tool in a clean and object-oriented way, using TypeScript as well.
Hence the classes and their structure as well as dependencies are analyzed first. The

tool has three major dependencies with the *moment.js* [1] library , the *vm2* [2] module and *turf.js* [3]. *Moment* is a library for date manipulation and comparison and already used in the main application. *Vm2* as described in Section 4 provides the *VM* sandbox for evaluating the condition in a safe context. *Turf.js* provides features to check whether a point is inside a polygon or not. Figure 6.2 illustrates the class structure of the tool.

The package *contextmodel* contains the classes that were introduced in the previous Chapter 5 and represents all vital information, which is extracted by the *QuestionnaireParser*. The latter is initialized with the model and result and provides functions for parsing the specific contexts and result.
The parser is called by the *RuleManager* class which is the entry point for the tool and manages the evaluation procedure. Upon initialization the model and result as well as the timeout for the sandbox are injected. Invoking the evaluation with the function *evalRules*, also provides the possibility to inject a user-defined object, for example, a set of functions. Within *evalRules* the parser is called and after the evaluation is finished the result as well as the error message that may occur are inserted into the *EvaluationResult*. However, the evaluation itself is conducted to the *RuleEvaluator*.
The latter is initiated with the complete result context, the timeout for the sandbox, as well as the optional object. Calling the function *evalRule* invokes the evaluation of a single rule. The VM is initiated with the given timeout and context, which is provided by the *createSandboxContext* function that resolves the variables mapping.
After resolving all variables the condition is evaluated and its result analyzed and returned. The *RuleManager* then inserts the result into the corresponding *EvaluationResult* object. Figure 6.3 illustrates the message flow within and outside the tool.

[1]`http://momentjs.com/`
[2]`https://github.com/patriksimek/vm2`
[3]`http://turfjs.org/`



<<use>>
vm2
VM
evaluator
RuleManager
- qModel : QuestionnaireModel
- qResult : QuestionnaireResult
- evaluationResult : EvaluationResult[]
+ constructor(model: QuestionnaireModel, result : QuestionnaireResult)
+ evalRules(functionSet : Any [0..1]): EvaluationResult[]
RuleEvaluator
- ruleTimeout : Integer
- functionSet : any [0..1]
- resultContext : ResultContext[]
+ constructor(resultContext : ResultContext[], timeout : Integer, functionSet : Any [0..1])
+ evalRule(ruleContext: RuleContext) : PartialEvaluationResult
- createSandboxContext(ruleContext : RuleContext) : Any
- resolveVarMapping(mapping : VariablesMapping) : Any
- getQuestionType(questionId : String) : String
- getResultContent(questionId : String) : Content
- resolveSingleValue(content: Content) : Any
- resolveKeyValue(content: Content, value: String[]) : Any
- resolveMatrix(content: Content, value: String[]) : Boolean
<<use>>
Moment.js
Turf.js
<<import>>
util
Functions
<<import>>
<<call>>
contextmodel
RuleContext
+ key : String
+ conditionString : String
1
+ variablesMapping
0 .. *
VariablesMapping
+ variableName : String
+ questionId : String
+ value : String[]
EvaluationResult
+ key : String
+ name : String
+ result : Evaluation
+ error : String [0..1]
+ positive : ResultText
+ negative : ResultText
ResultText
+ headline : String
+ description : String
ResultContext
+ questionId : String
+ questionType : String
1
1 .. *
+content
Content
+ iteration : Integer
+ value : Any
<<Enumeration>>
Evaluation
TRUE
FALSE
ERROR
MISSINGDATA
contextparser
QuestionnaireParser
- qResult : QuestionnaireResult
- qModel : QuestionnaireModel
+ constructor(qResults : QuestionnaireResult, qModel : QuestionnaireModel)
+ parseResultContext() : ResultContext[]
+ parseEvaluationResult() : EvaluationResult[]
+ parseRuleContext() : RuleContext[]

Figure 6.2: Class Diagram

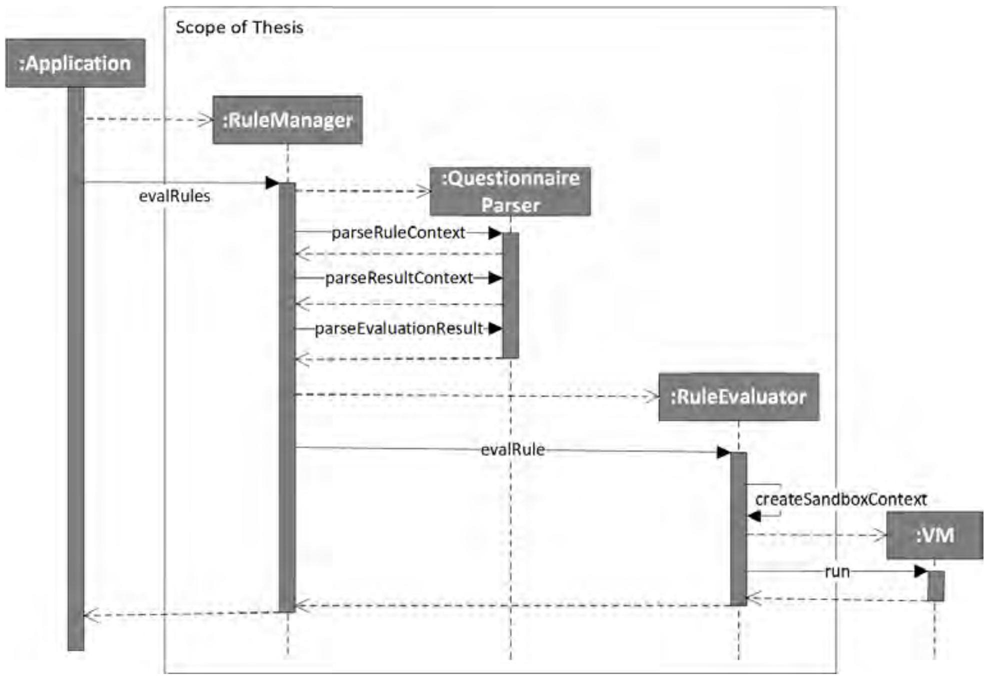


Figure 6.3: Sequence Diagram

6.3 Predefined and Custom Functions

This section provides details on how the function sets are handled. The tool comes with a predefined set that contains functions for Math, Date, Shapes and Strings. The mathematical function, for example, allow to calculate the maximum and minimum of a series of numbers and some more. The date functions allow to manipulate and compare dates as desired and is based on the moment.js library. Furthermore functions to check, for example, if a string is contained within another string or sentence are provided. Using turf.js allows for checking, if a point is within a polygon.
Because other functions may be necessary in future, the tool implements the possibility to inject a custom set of functions, when invoking the rule evaluation. These functions are then passed to the evaluator and finally into the sandbox as illustrated in Figure 6.4. If functions with the same name exist in both sets, the custom function will override the predefined function.

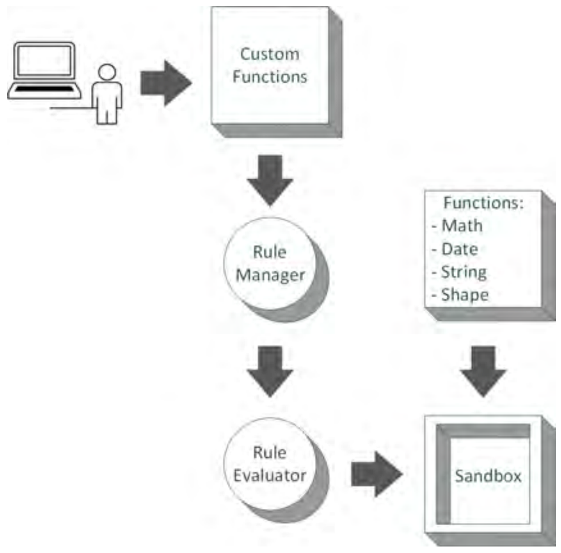


Figure 6.4: Function

6.4 Creating the Sandbox and Evaluation

This section provides details on building and running the sandbox, which is used for evaluating rules. To build the sandbox the variable mappings need to be resolved and, therefore, a closer look at the question types is needed. Recap the existing question types. These question types can be clustered because some of them act similar and are resolved in the same or a similar way.
The questions of the first group contain only a single value in its result and, therefore, need only the id of the question to retrieve the value. Listing A.17 shows an example how the result looks like for these kind of questions and Listing A.18 a corresponding variable mapping that retrieves the value and stores it in the variable. In order to receive the value the evaluator simply has to get the result context for the given question id and retrieve the content object and its value property. This works in a similar way for *FreeFloat, FreeDate*, *FreeTextArea*, as well as *SliderSingle* questions. However, due to JSON parsing the handling of the *FreeDate* question slightly differs since the value may be converted to a Moment object before passing it into the sandbox.
The next group of questions contain multiple values mapped to unique keys as their result. Listing A.19 shows an example result for these types of questions with multiple

key-value pairs. The variables mapping obviously has to contain not only the id of the question, but also the key of the requested value (cf. Listing A.20). Since JavaScript objects act like associative arrays the value is retrieved by first getting the content object through the question id. Then the value property is extracted and the value of the variable retrieved by accessing the property identified by the key. These two approaches fit to all other question types, but the *Matrix* question.

A *Matrix* question contains pairs of keys and in contrast to the other questions, no direct value. In order to identify a cell within a matrix, two indexes are necessary (cf. Listing A.21). Together they identify a selected answer, which is represented as a Boolean true. A corresponding variables mapping is presented in Listing A.22.

The first string given in the value array represents the row and the second one the column. The value is resolved by verifying, if for a given row the column key in the mapping is the same as the key within the result. In the given example the variable would have the Boolean value false, because the column keys are different.

As the variables can be resolved now the sandbox can finally be build. Listing A.23 shows a simplified example how the sandbox is created and run. First of all, in Line 1 the context is build by injecting the predefined set of functions. The latter may be overridden in Line 3-5 by a user defined set of functions. In Line 7 to 9 all variables are resolved by iterating through each mapping and adding it to the context. This is then, together with the timeout option, passed to the VM's constructor.

In Line 14 the sandbox is build and the rule, respectively its condition, is evaluated. To meet the requirement that a rule is not allowed to access any information that is not directly related, every rule has its own separate sandbox and context.

The final result, however, is not only a boolean value. Furthermore, it is represented by a enumeration consisting of the values *True*, *False*, *Error* and *MissingData*. The latter also reports an error, but indicates that the error occurred due to missing data. This may be the case when a *ReferenceError*[4] appears. *ReferenceError* occurs if a object, i.e., a variable or function, is called, but not defined. *Error* contains every other error that may be thrown. Figure 6.5 illustrates how the different results come about.

[4]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Not_defined`

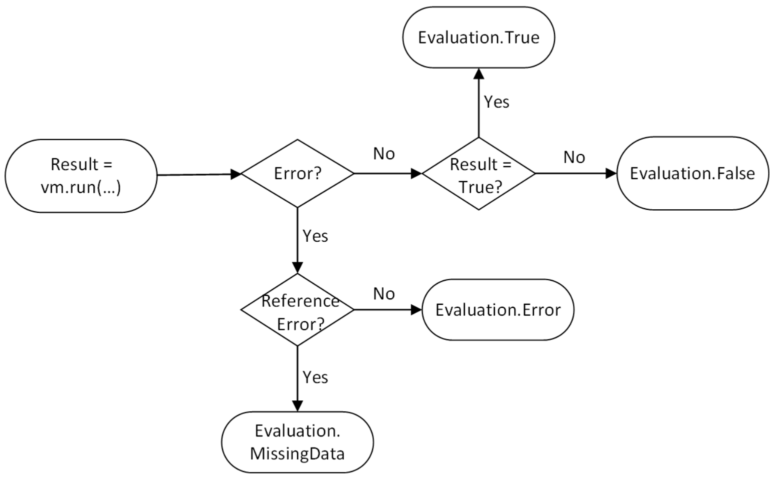


Figure 6.5: The Different Results

7 Summary and Future Work

The framework for mobile data collection applications has been introduced, the given data analyzed and an evaluation tool implemented. For evaluation purpose the *results* object contains question ids mapped to the corresponding answers. The model contains the *rules* array with rule definitions consisting of a condition string and variable mappings. The latter are resolved with the id of a question and specific keys, retrieving a certain value from the result. To prevent the application from errors or malicious code the *vm2* module is used to execute a rule within its own and safe context.
The evaluator tool offers a predefined function set as well as the possibility to inject a custom function set. These allow domain experts to create complex rules to deeply analyze the questionnaire result. The automatic analysis with the evaluator tool can lead to further time and cost savings compared to paper-based data collection. All in all this thesis offers a tool that extends the underlying framework with the functionality to evaluate defined rules in a safe manner.
However, with the development of the main application this tool may need further development, as well. For example, the iterations of the result were omitted within this thesis. Iterations appear if a question has to be answered multiple times, e.g., if a participant is asked for his siblings. During evaluation the number of siblings may be interesting. In other cases, answers of all iterations may be vital for rule evaluation. Therefore, the answers need to be stored in some sort of iterable list. This would require changes within the main application and its rule definitions, respectively the variable mappings as well as the evaluator tool. At the current stage a variable resolved from its mapping does only contain a single value. Allowing multiple values, e.g., by using TypeScript arrays and appending additional array functions, could solve this problem. This would need some

adjustments in the configurator as well as the evaluator tool. The amount of changes needed, is dependent on the degree of changes on the rule definitions, respectively the variables mapping. Allowing, only to access the same key within every iteration may need a lot less changes than allowing different keys for different iterations.
Another approach that would need more changes to the evaluator than the configurator, is to extend the variables mapping with additional properties that define if iterations and how many should be used for evaluation. The evaluator tool then reads this properties and executes a condition with all possible combinations of variables. This would need some specification or more properties to define how to treat the different results. Therefore, the first approach may be more practical.
However, the evaluator only analyzes a small amount of the given data, namely the final answers. Remember, that the main application logs different information, such as the data history (cf. Section 2.3). The latter contains information about how the question was answered and, therefore, it may be utilized to draw conclusions on the participant's behavior. Analyzing this data may help detecting any kind of participant bias, such as social desirability that leads to participants not answering truthfully [10].

Bibliography

[1] Ruf-Leuschner, M., Brunnemann, N., Schauer, M., Pryss, R., Barnewitz, E., Liebrecht, M., Kratzer, W., Reichert, M., Elbert, T.: Die KINDEX-App - ein Instrument zur Erfassung und unmittelbaren Auswertung von psychosozialen Belastungen bei Schwangeren in der täglichen Praxis bei Gynäkologinnen, Hebammen und in Frauenkliniken. Verhaltenstherapie 2016, Volume 26, 171-181 (2016)

[2] Ivan Pavlović, Tomaž Kern, D.M.: Comparison of paper-based and electronic data collection process in clinical trials: costs simulation study. Contemporary Clinical Trials, Volume 30, Issue 4, 300-316 (2009)

[3] 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: 5th IEEE International Conference on Mobile Services (MS 2016), San Francisco, USA, IEEE Computer Society Press (2016) 1–8

[4] Schobel, J., Pryss, R., Schickler, M., Reichert, M.: A Configurator Component for End-User Defined Mobile Data Collection Processes. In: Demo Track of the 14th International Conference on Service Oriented Computing (ICSOC 2016), Baff, Alberta, Canada (2016)

[5] Schobel, J., Pryss, R., Schickler, M., Schlee, W., Probst, T., Gebhardt, D., Reichert, M.: Development of Mobile Data Collection Applications by Domain Experts: Experimental Results from a Usability Study. In: 29th International Conference on Advanced Information Systems Engineering (CAiSE 2017), Essen, Germany, Springer (2017) 60–75

[6] Nadalin, A.: Eval no more: a journey through nodejs' vm module, vm2 and expression language. `https://odino.org/eval-no-more-understanding-vm-vm2-nodejs/` (2017) , visited 2018-09-29.

[7] Griffith, C.: Mobile App Development with Ionic, Revised Edition: Cross-Platform Apps with Ionic, Angular, and Cordova. O'Reilly media (2017)

[8] Hussain, A.: Angular 5: From Theory to Practice: Build the web applications of tomorrow using the new Angular web framework from Google. CodeCraft (2017)

[9] Syed, B.A.: TypeScript Deep Dive. Samuari Media Limited (2017)

[10] Farnsworth, B.: What is participant bias? (and how to defeat it). `https://imotions.com/blog/participant-bias/` (2016) , visited 2018-09-12.

A

Sources

```
{
  "meta": {...},
  "model": {...},
  "labels": [...],
  "rules": [...],
  "media" : [...]
}
```

Listing A.1: The basic structure of the model

```
"model": {
    "nodeDataArray": [
      {
        "key": -1,
        "isGroup": true,
        "category": "Structure"
      },
      {
        "text": "FreeDate",
        "category": "Question",
        "group": -9,
        "element": {
          "name": "FreeDate",
          "placeholder": {"de": ""},
          "question": {
            "de": "Geben Sie ihr Geburtsdatum ein"
          },
          "questionType": "FreeDate",
          "exportName": "",
          "isMandatory": true,
          "instruction": {
            "de": ""
          },
          "minDate": "1920-01-01",
          "maxDate": "2017-01-01"
        },
        "key": -23
      }
    ],
    "linkDataArray":[{
        "from": 1,
        "to": -10
      },
      {
        "from": -10,
        "to": -12
      }
    ]
}
```

Listing A.2: Small example of the model object

```
"rules": [
    {
      "key": "rule1",
      "name": "Rule 1",
      "conditionString": "$a > 5",
      "variablesMapping":[
        {
          "variableName": "$a",
          "questionId" : "-100",
          "value": ["1232142"]
        }
      ],
      "positive":
  {
    "headline" : {"de" : "lorem ipsum"},
    "description" : {"de" : "Ut enim ad minim"}
  },
      "negative":
  {
    "headline" : {"de" : "Mauris commodo"},
    "description" : {"de" : "ac tortor dignissim"}
  }
    },
    {
      "key": "rule2",
      "name": "Rule2",
      "conditionString": "$a > 5 && $b > 15",
      "variablesMapping":[
        {
          "variableName": "$a",
          "questionId" : "-100",
          "value": ["1232142"]
        },
        {
          "variableName": "$b",
          "questionId" : "-99",
          "value": ["1232"]
        }
      ],
      "positive": {
        "headline" : {"de" : "Fermentum iaculis"},
        "description" : {"de" : "Orci eu lobortis elementum"}
      },
      "negative": {
        "headline" : {"de" : "Facilis magna"},
        "description" : {"de" : "Venenatis a condimentum vitae"}
      }
    }
]
```

Listing A.3: Rule definitions

```
{
  "data": {
    "type": "results",
    "attributes": {
      "instance": "2017-11-29T12:42:40.127Z",
      "identifier": "Max Mustermann",
      "description": "Lorem ipsum",
      "locale": "de_DE",
      "payload": {
        "processLog": [{
          "level": "INFO",
          "source": "ENGINE",
          "message": "Engine started...",
          "timestamp": 1511959158753,
          "id": "",
          "action": ""
        }],
        "componentsLog": [{
          "level": "INFO",
          "source": "FREEFLOAT",
          "message": "State:",
          "timestamp": 1511959294354,
          "id": -22,
          "action": "FINISHED"
        }],
        "dataHistory": {
          "-23": [{
            "iteration": 0,
            "history": [{
              "timestamp": 1511959302375,
              "value": {
                "1505481288327966": 100
              }
            }]
          }]
        },
        "results": {
          "-23": [{
            "timestamp": "2017-11-29T12:42:00.770Z",
            "exportname": "Distribution",
            "name": "Distribution",
            "iteration": 0,
            "value": {
              "1505481288327966": 100,
              "1505481761347780": 0,
              "1505482088301257": 0,
              "1505482119317299": 0,
              "150548215163666": 0
            }
          }]
        }
      },
      "flags": [],
      "colors": [],
      "collected_at": 1511959320,
      "client": {
        "os": "Android 6.0",
        "device": "3574788e0e4fc2",
        "application": "Questionnaire v.0.0.1 b.1"
      }
    },
    "relationships": {
      "purchase": {
        "data": {
          "type": "purchases",
          "id": "wdmvbn8yw6y35a7r"
        }
      },
      "product": {
        "data": {
          "type": "products",
          "id": "vazd7x6gqrer4wq9"
        }
      },
      "productversion": {
        "data": {
          "type": "productversions",
          "id": "mqwlpk64kney39zo"
        }
      }
    }
  }
}
```

Listing A.4: Basic structure of the result

```
"results": {
              "-23": [{
                  "timestamp": "2017-11-29T12:39:11.783Z",
                  "exportname": "Free Float",
                  "name": "Free Float",
                  "iteration": 0,
                  "value": 55.437
              }],
              "-22": [{
                  "timestamp": "2017-11-29T12:39:11.821Z",
                  "exportname": "SingleChoice",
                  "name": "SingleChoice",
                  "iteration": 0,
                  "value": {
                      "1505478670111650": false,
                      "1505478723948697": true,
                      "150547872484694": false,
                      "1505479884619120": false
                  }
              }],
              "-21": [{
                  "timestamp": "2017-11-29T12:39:11.791Z",
                  "exportname": "Matrix",
                  "name": "Matrix",
                  "iteration": 0,
                  "value": {
                      "1505480828722": "1505480931808",
                      "1505481091481": "1505480713371",
                      "1505490936380": "1505931253808"
                  }
              }],
              "-20": [{
                  "timestamp": "2017-11-29T12:39:11.794Z",
                "exportname": "FreeTextArea",
                "name": "FreeTextArea",
                "iteration": 0,
                "value": "Lorem ipsum"
            }],
            "-19": [{
                "timestamp": "2017-11-29T12:39:11.799Z",
                "exportname": "YesNo",
                "name": "YesNo",
                "iteration": 0,
                "value": {
                    "150548060740682": false,
                    "1505480607406600": true
                }
            }],
            "-18": [{
                "timestamp": "2017-11-29T12:39:11.804Z",
                "exportname": "SliderSingle",
                "name": "SliderSingle",
                "iteration": 0,
                "value": 10
            }],
            "-17": [{
                "timestamp": "2017-11-29T12:39:11.809Z",
                "exportname": "Ranking",
                "name": "Ranking",
                "iteration": 0,
                "value": {
                    "1505480089975517": 4,
                    "1505480191977551": 3,
                    "1505480238343170": 1,
                    "1505480291247701": 2
                }
            }],
            "-16": [{
                "timestamp": "2017-11-29T12:39:11.813Z",
                "exportname": "SliderRange",
                "name": "SliderRange",
                "iteration": 0,
                "value": {
                    "lower": 1460,
                    "upper": 5000
                }
            }],
            "-15": [{
                "timestamp": "2017-11-29T12:39:11.817Z",
                "exportname": "FreeDate",
                "name": "FreeDate",
                "iteration": 0,
                "value": "1997-01-01"
            }],
            "-14": [{
                "timestamp": "2017-11-29T12:39:11.779Z",
                "exportname": "Distribution",
                "name": "Distribution",
                "iteration": 0,
                "value": {
                    "1505481288327966": 70,
                    "1505481761347780": 0,
                    "1505482088301257": 10,
                    "1505482119317299": 20,
                    "150548215163666": 0
                }
            }],
            "-13": [{
                "timestamp": "2017-11-29T12:39:11.825Z",
                "exportname": "MultipleChoice",
                "name": "MultipleChoice",
                "iteration": 0,
                "value": {
                    "12": false,
                    "13": false,
                    "14": true,
                    "150547959606220": true,
                    "1505479804089178": false
                }
            }],
            "-12": [{
                "timestamp": "2017-11-29T12:39:11.828Z",
                "exportname": "btngrid",
                "name": "btngrid",
                "iteration": 0,
                "value": {
                          "150547821213853": false,
                          "1505478329993168": false,
                          "1505478346579512": false,
                          "1505478360820218": false,
                          "1505478374543373": false,
                          "1505478375554589": false,
                          "1505478376396437": false,
                          "150547837719325": false,
                          "1505478377826144": false,
                          "1505478378587753": false,
                          "1505478379273398": true,
                          "1505478379989920": true,
                          "1505478380760857": true,
                          "1505478381546629": true,
                          "1505478382701237": true
                }
            }]
}
```

Listing A.5: Question types and results

```
var jexl = require('jexl');

var context = { age : 15 };

jexl.eval('age < 18', context, function(err, res){
  if(err){
    console.log('Something went wrong');
  }else{
    console.log(res);
  }
});
```

Listing A.6: A simple JEXL example

```
var jexl = require('jexl');

var context = { a : 10 };

jexl.addTransform('max', function(...args){
  return Math.max(...args);
});

jexl.eval('0|max(1,2,3,4,5,a)', context).then(function(res){
  console.log(res);
});
//prints 10
```

Listing A.7: JEXL transform

```
var safeEval = require('notevil')

var context = { x : 25,
         max: function(...args){ return Math.max(...args)} };

var result = safeEval('max(5, 15, x)', context);
console.log(result); // prints 25
```

Listing A.8: Notevil safeEval

```
var vm = require('vm');

var context = { x : 25, max: (...args) => Math.max(...args)};

var result = vm.runInNewContext('max(5, 10, x)', context);
console.log(result); // prints 25
```

Listing A.9: NodeJS VM

```
var result = vm.runInNewContext(
  'this.max.constructor.constructor("return process")().exit()'
  , context);
```

```
console.log(result); // is never executed
```

Listing A.10: Exploit to VM

```
import { VM } from 'vm2';

const context = { x: 25, max: (...args) => Math.max(...args)};

const vm = new VM({
  sandbox: context,
  timeout: 1000
})

const result = vm.run('max(5, 10, x)');
console.log(result); // prints 25
```

Listing A.11: vm2 VM example

```
var safeEval = require('notevil')

safeEval('/((a+)+)b/.test("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa")');
```

Listing A.12: Malicious regular expression

```
export class RuleContext{

  key: string;
  conditionString: string;
  variablesMapping: VariablesMapping[];

  constructor(key: string, conditionString: string,
              varMapping : VariablesMapping[]) {
    this.key = key;
    this.conditionString = conditionString;
    this.variablesMapping = varMapping;
  }
}

export class VariablesMapping{

  variableName: string;
  questionId: string;
  value: string[];

  constructor(variableName: string, questionId: string,
              value: string[]) {
    this.variableName = variableName;
    this.questionId = questionId;
    this.value = value;
  }
}
```

Listing A.13: The rule context

```
export enum Evaluation{

  True,
  False,
  Error,
  MissingData,
}
```

Listing A.14: The enumeration with the possible results

```
export class EvaluationResult{

  key: string;
  name: string;
  result: Evaluation;
  error?: string;
  positive: {
    headline: {[lang: string]: string},
    description: {[lang: string]: string},
  };

  negative: {
    headline: {[lang: string]: string},
    description: {[lang: string]: string},
  };

  constructor(key: string, name: string,
              positive: { headline: {[lang:string]: string},
              description: {[lang: string]: string}},
              negative: { headline: {[lang:string]: string},
              description: {[lang: string]: string}}) {
    this.key = key;
    this.name = name;
    this.result = Evaluation.Error;
    this.positive = positive;
    this.negative = negative;
  }
}
```

Listing A.15: The evaluation result

```
export class ResultContext{

  questionId: string;
  questionType: string;
  content: Content[];

  constructor(qId: string, qType: string, content: Content[]) {
    this.questionId = qId;
    this.questionType = qType;
    this.content = content;
  }
}

export class Content{

  iteration: number;
  value: any;

  constructor(iteration: number, value: any) {
    this.iteration = iteration;
    this.value = value;
  }
}
```

Listing A.16: The result context

```
"-12": [{
    "timestamp": "2018-08-29T12:42:00.770Z",
    "exportname": "Single value question",
    "name": "Single value question",
    "iteration": 0,
    "value": 52.3
}]
```

Listing A.17: Questions with only a single value

```
"variablesMapping":[
        {
          "variableName": "$f",
          "questionId" : "-12",
          "value": []
        }
]
```

Listing A.18: A variables mapping for a question with a single value

```
"-23": [{
    "timestamp": "2017-11-29T12:42:00.770Z",
    "exportname": "Multiple value question",
    "name": "Multiple value question",
    "iteration": 0,
    "value": {
        "12345": 70,
        "12341": 0,
        "12347": 30,
        "12359": 0,
        "12350": 0
    }
}]
```

Listing A.19: Questions with multiple values

```
"variablesMapping":[
        {
          "variableName": "$d",
          "questionId" : "-23",
          "value": ["12347"]
        }
]
```

Listing A.20: A variables mapping for a question with multiple values

```
"-21": [{
    "timestamp": "2017-11-29T12:39:11.791Z",
    "exportname": "Matrix",
    "name": "Matrix",
    "iteration": 0,
    "value": {
        "15054": "15061",
        "15055": "15063",
        "15056": "15062"
    }
}]
```

Listing A.21: The result of a matrix question

```
"variablesMapping":[
        {
          "variableName": "$m",
          "questionId" : "-21",
          "value": ["15054", "15063"]
        }
]
```

Listing A.22: A variables mapping for a matrix question

```
const sandbox = Object.assign({}, predefinedFunctions);

if (this.functionSet) {
  Object.assign(sandbox, this.functionSet);
}
```

```

for (const mapping of ruleContext.variablesMapping) {
  Object.assign(sandbox, this.resolveVarMapping(mapping));
}

const sandbox = this.createSandboxContext(ruleContext);
const vm = new VM({ sandbox, timeout: this.timeout }) ;

const result = vm.run(ruleContext.conditionString);
```

Listing A.23: Building and running the sandbox

List of Figures

List of Tables

Name: Daniel Rollenmiller Matriculation number: 867350

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Daniel Rollenmiller