Kim Möller

Developing a graphical user interface for creating chatbot configurations

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

28 April 2018

Metropolia

Author	Kim Möller
Title	Developing a graphical user interface for modifying chatbot configurations
Number of Pages	33 pages
Date	28 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Petri Pellinen, Chief Technology Officer Peter Hjort, Senior Lecturer

The purpose of this study was to design and develop a graphical chatbot flow editor application that would be capable of transforming a visual graph into a text represented configuration for a Finnish ICT company. The chatbot is one of their products being sold to customers to increase their customer service profitability and make their customer experience better over the web.

The application was developed to solve the problem of developers using large numbers of hours writing the configuration files by hand due to it being slow, tedious and highly error prone. Other benefits of the application are allowing customers to customize the chatbots themselves and possibly increasing chatbot sales to new customers or existing customers who are not using the chatbot.

The application was developed by using modern JavaScript technologies such as ECMAScript 6 and TypeScript along with Rappid Diagramming framework as the main JS framework. The applications back-end server was developed using Java with several libraries including Spring and Hibernate. All data persisted by the application were stored in a MySQL database. The application was developed into a state where it could locally work on a developer machine to create visual chatbot flows that can be exported into a TOML string, used as the configuration language, and saved to a file on a chat-server.

When the application was in a state that it worked as intended at the end of the thesis project, a user study was conducted to find possible issues with the application when in the hands of non-technical people. The study was successful in the way that it brought out several issues that should be fixed before giving the application to customers.

The future plans regarding the development of the application are to first deploy it into a test environment where it can be tested by the company and one if its customer companies. After this the application should be deployed into production.

Keywords	Chatbot, Rappid, Java, JavaScript

Tekijä Otsikko Sivumäärä Aika	Kim Möller Keskustelurobotin konfiguraatioita luova graafinen käyttöliittymä 33 sivua 28.4.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	tieto- ja viestintätekniikka
Ammatillinen pääaine	Software Engineering
Ohjaajat	teknologiajohtaja Petri Pellinen lehtori Peter Hjort

Insinöörityön tarkoituksena oli graafisen, chatbot-polkuja luovan ohjelman suunnittelu ja kehittäminen. Kehitystyö tehtiin suomalaiselle tietotekniikkayritykselle, joka muun muassa myy chatboteja eli keskustelurobotteja asiakasyrityksilleen tarkoituksena parantaa niiden asiakaspalvelun tuottoisuutta ja asiakaskokemusta.

Ohjelmisto kehitettiin vähentämään yrityksen ohjelmistokehittäjien käyttämiä tuntimääriä chatbotien luomiseen. Chatbot-tekstikonfiguraatiot luotiin ennen projektia käsin, mikä oli työlästä, hidasta ja virhealtista. Ohjelmiston muihin hyötyihin kuuluu asiakkaiden mahdollisuus muokata chatbotin polkuja itse ohjelman avulla ja parantaa chatbotin myyntiä uusille asiakkaille sekä vanhoille asiakkaille, joilla chatbot ei vielä ole käytössä.

Ohjelma kehitettiin käyttämällä nykyaikaisia JavaScript-teknologioita, kuten ECMAScript 6 ja TypeScript. Rappid-diagrammikirjasto valittiin ohjelman pääasialliseksi JavaScript-ohjelmistokehykseksi. Sovellukseen liittyvä palvelu luotiin käyttämällä Java-ohjelmistokieltä ja sen useita kirjastoja, kuten Spring ja Hibernate. Kaikki sovelluksen tallentama data otettiin talteen MySQL-tietokantaan. Ohjelma kehitettiin siihen pisteeseen, että se toimisi lokaalisti ohjelmistokehittäjän kehitysympäristössä ja sillä pystyisi luomaan TOML-konfiguraatiota visuaalisesta graafista. TOML on chatbotin käyttämä konfiguraatiokieli.

Kun sovellus oli edellä mainitussa toimivuustilassa, sille tehtiin käyttäjäkokemustestaus yrityksen työntekijän kanssa. Testauksen tarkoituksena oli löytää mahdollisia ongelmakohtia, jotka heikentäisivät ohjelman käyttäjäkokemusta. Testaus onnistui, sillä sen aikana löytyi useita ongelmia, jotka tulisi korjata tulevaisuudessa.

Sovelluksen jatkokehitysvaiheena on ensin ottaa sovellus käyttöön pilvipalvelussa toimivalla testialustalla. Tässä vaiheessa sovellusta testaa sekä kehittäjäyritys että yksi sen asiakasyrityksistä, jolla on chatbot käytössä. Tämän jälkeen seuraava vaihe on ottaa sovellus käyttöön tuotannossa.

Avainsanat	Chatbot, Rappid, Java, JavaScript, Keskustelurobotti

Metropolia

Contents

List of Abbreviations

BPMN	Business Process Model and Notation
TOML	Tom’s Obvious Minimal Language
BFS	Breadth-first-search
DFS	Depth-first-search
API	Application Programming Interface
JPA	Java Persistence API
DFS	Depth-first-search

1 Introduction

The objective of this thesis project was to design and create a user friendly graphical editor application used to design chatbot flows. The company that ordered the project was Lekane Ltd, a company that specializes in improving the customer experience of websites by providing for instance chat and callback services for online shops etc. The company had already developed the chatbot before this project started, so this thesis will not go into great details about the implementation.

The problem that initiated the need for this editor application was that the previous method of creating chatbots was slow and prone to errors. Beforehand the chatbot flows were written by hand into a text-based configuration file. The purpose of the application is to allow for easier creation and editing of the chatbot flows. One of the key aspects of the application was to allow the customers, who are using the chatbot, to modify the flow themselves. This should decrease the number of hours used inside the company to edit the chatbot flows. The editor application will hopefully have a positive impact on the chatbot sales numbers.

The project started by going through technologies used for the applications development. The technologies were mainly software libraries that would be used as part of the editor application. The libraries were being compared based on a set of criteria, the number of features required by the application, the activity and support of the libraries, their documentation and price. The goal was to choose stable and relatively new libraries that would be well maintainable in the future by various developers of the company. The applications first version was considered finished when it would work and could be deployed to production but would not include all the additional planned features. This was done to ensure that the application could be taken into use as soon as possible.

Working on the application included creating the front-end application with JavaScript, the back-end server with Java and having a MySQL database to preserve data into. Working on the front-end part required learning how the chosen library, Rappid, worked. The back-end server runs within an embedded Jetty server and mainly includes a REST-service and a hibernate configuration to deal with data persistence. Outside of technical development, the project kept user experience in an important role.

2 Application backgrounds

The company that the project was made for offers a possibility to use a chatbot on their client's web page amongst other services. The chatbot is constructed out of predefined message replies and buttons for the user to navigate through the chatbot flow. The reason client's use the chatbot is to lessen the need for human interaction on problems that could be solved by guiding the user onto the right part of the website. Behind the chatbot is a Java back-end with the logic of the chatbot and configuration files that describe the flow of the chatbot. The configuration files were written by developers before the project started. Creating a new chatbot like this was a very tedious and long project prone to many errors. The configuration file was often filled with typos during development and finding errors took a long time.

The chatbot-editor project was started to remove these problems by creating a new way of developing chatbot flows. The editor application is a drag-and-drop style application that uses shapes to visualize the flow. The editor will also give the possibility for clients to directly modify their bots to their liking. This was not possible with the configuration files and thus clients had to send requests to the development team every time they wanted any changes to be made to the chatbot. The project has multiples goals that will not only benefit the company and its development team, but also the customers using or thinking of buying the chatbot. It will save time and money for everyone involved and possibly attracts more customers into buying the chatbot service.

The information part of the report will be going through various technologies and ideas behind the project. The decisions made before starting to build the chatbot-editor will be outlined during the upcoming subchapters.

2.1 Diagramming Libraries

The project started off by searching for a diagramming tool to use for the graphical user interface. Searching was done by finding as many libraries suited for the project as possible and evaluating them based on a list of criteria. The list was formed from how well the library was able to tackle the requirements of the project, the price of the library, the health of the library; how often is it updated and how active the developers are with the community and the quality of the libraries documentation. Most of the libraries found

were meant to create complete diagrams for webpages instead of creating tools for editing diagrams. These included for instance JGraph and GoJS. In the end, three libraries were found that might suit the project's needs. These were Draw2D touch, JsPlumb toolkit and Rappid Diagramming Framework.

Draw2D-library

Draw2D was the first diagramming library to be evaluated for the chatbot-editor -project. The library works around shapes drawn on a canvas and modified when necessary. Searching through the library documentations made it clear that it had features required for the chatbot-editor, for example the possibility of nested shapes. These nested shape structures could be used to create singular views with parent-child-shapes. [1.] An example taken from the Draw2D library page is shown in Figure 1.

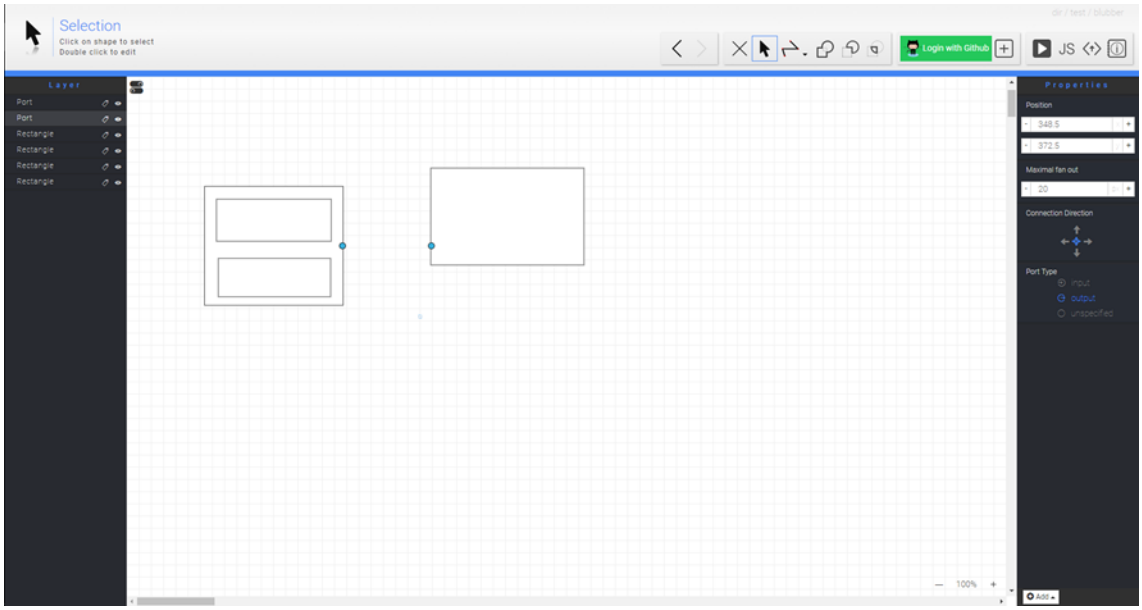

Figure 1. Draw2D touch demo [2].

The main functionality for creating shapes in Draw2D is to pick a tool and click and drag the wanted shape on the canvas. Considering the complexity of the shapes required to illustrate the chatbot, this method would be slow and cumbersome. The structure of the bot is illustrated in Figure 2. The lower price of the library was one of the big reasons it was originally picked to be evaluated. It also comes with clear documentation and easily understandable source code. [3.] The evaluation was cut short due to problems regarding the health of the library. The library forums did not have ongoing discussions, there

had been no updates in a while and the developer of the project did not respond to queries in time.

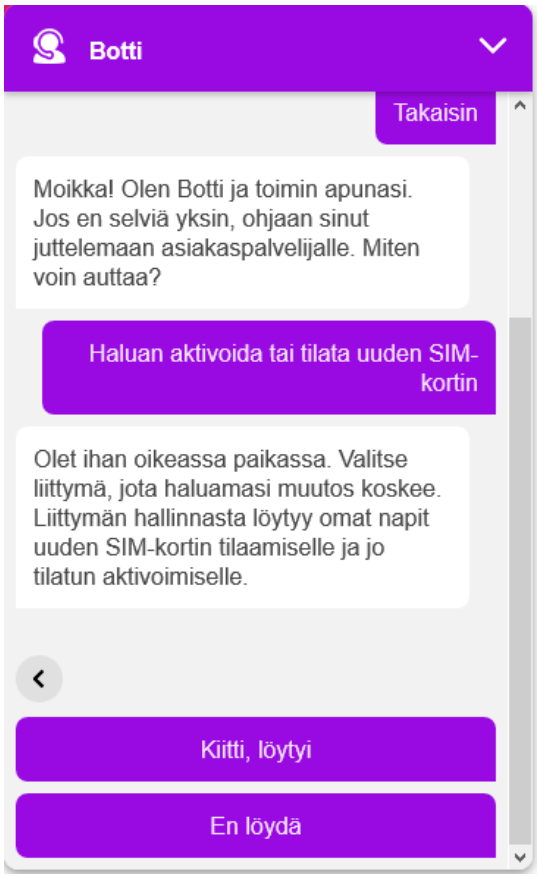

Figure 2. Chatbot example [4].

JsPlumb toolkit

JsPlumb toolkit was the second JavaScript library to be evaluated for the project. The price of the library stood out first as a negative aspect. The license costs 3 500 euros and gives access to 1-5 concurrent developers with no technical support included. The base license does not include updates, instead these are bundled in with technical support with an annual price of 3 000 euros. This additional bundle seems mandatory if the library will be in use for several years. [5.] The documentation is not well organized and is complicated, making it difficult to start working with the library.

The source code and data structures used are clear and understandable. The elements are constructed out of JS-objects and the library functions are made using basic JS and

the libraries included, for example jQuery. [6.] Figure 2 shows an example app done with JsPlumb.

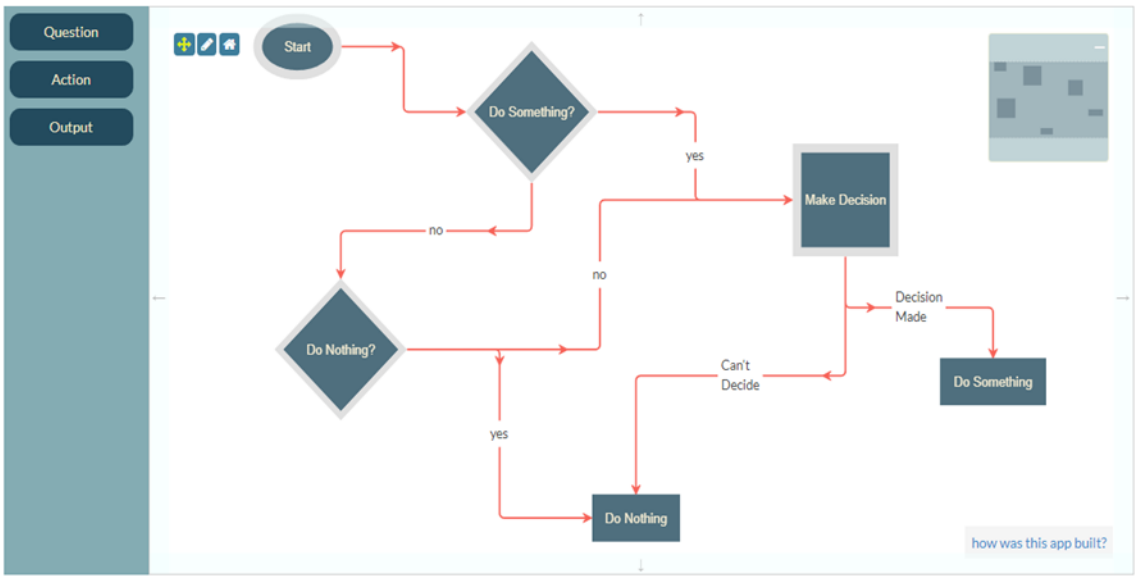

Figure 3. JsPlumb toolkit demo [7].

JsPlumb can create nested elements, thus passing one of the requirements set for the chosen library. The example however does not showcase a way for the user to modify the existing diagram. This is a fundamental feature of the project thus making JsPlumb unusable. The library is mainly meant to be used to create multiple different diagram types, like the question-answer-diagram in Figure 3 or a database model diagram. [7.]

Rappid Diagramming Framework

Rappid was the last library to be evaluated. The library comes packed with many features and methods to be used to build your own diagram-editing-application. On Rappid's homepage there are multiple pre-build demos that show of different features of the library. These include, but are not limited to, a BPMN editing app, a shortest-path-finder app, a finite-state-machine editing demo and a diagramming toolkit, that functioned as the base for the chatbot-editor application, shown in Figure 4. [8.]

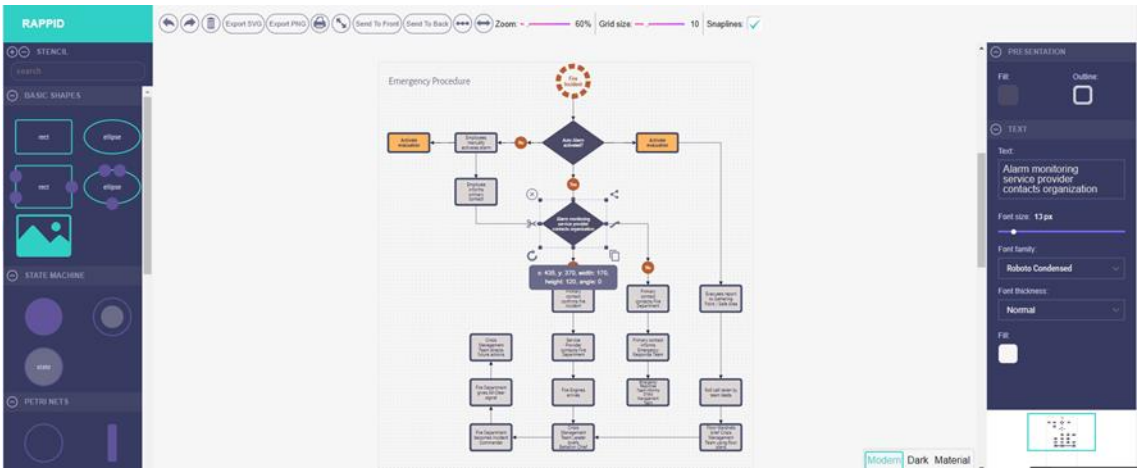

Figure 4. Rappid Diagramming Framework demo [8].

A partial list of shapes used in the demo can be found in the left part of Figure 4. The shapes can be dragged and dropped onto the paper located in the middle of the application. After placing the shapes on the paper their size, shape and content can be edited giving the user freedom in how the diagram should look. Most of the shape-editing happens in the right-hand-side column that is populated with options once a user clicks a shape on the paper. These options include for instance modifying the text content of the shape, if present, changing font, background and border color and choosing a font and modifying its size. The top part of the demo application is filled with various tools to be used while editing the diagram. These include redo/undo buttons, zoom, clear paper etc. All application parts can be modified with instructions given in the library documentation [9].

The demo shown in Figure 4 gave a good starting point into exploring the possibilities of the library. It did not, however, give enough insight into the possible shapes that could be created by the library and whether they could be complex enough to fill the needs of the chatbot-editor. Rappid does provide a month-long trial period to be used to evaluate different use-cases and potentials of the library before buying the full license. With the license comes the full source code of Rappid and all the demos presented on the Demo page. The trial version was used to understand how the library works and if it could be used for the project.

Rappid is not the cheapest library costing 1 500 euros per developer with an optional package of library updates costing from 800 euros to 1 200 euros depending on the length of the package. The options are one, two or three years of additional updates.

The first year of updates is included in the base price of the license. The license package can also be expanded by a personal support personnel. This high price does transfer into a high-quality product with great documentation and regular updates. [10.]

Conclusion

After the project's evaluation Rappid Diagramming Framework was chosen as the library to use. The large selection of features and demos included in the library had the biggest impact in the decision-making process. The quality and size of the documentation affected the choice positively. The higher price of the library was overshadowed by the overall quality of the product. Draw2D was dropped because of the issues regarding its health discussed earlier. Having a healthy library with ongoing updates was important for the project. JsPlumb was not selected because of its high price and missing editing features. These two flaws made it unnecessary to dig deeper into the features of the library since much of the same could be found in Rappid with a lower price point.

2.2 Chatbot data types

The project's application will be transferring visual graphs into a configuration language that the chatbot can understand so it was important to know how the configuration language-in-use worked. This chapter will go over the basics of Tom's Obvious Minimal Language (TOML) used with the chatbot.

TOML is a configuration language similar to YAML or property lists. TOML as a language is richer than its predecessors XML and JSON. Unlike the predecessor languages, TOML supports datatypes that XML does not support at all and JSON incompletely. For instance, all numbers in JSON are floats and it does not support date types. Both XML and JSON are hard for humans to read and write whereas YAML is easier to read but it's precise indentations make it hard to produce. TOML strives to be easily read and written and it supports comments unlike the predecessor languages. Indentation is not required in TOML, but it can be used to enhance the reading experience. The basic structure of TOML is key-value pairs. [11.] TOML and its datatypes are showcased in Listing 1.

```
# Comments start with hash
```

```
foo = "strings are in quotes and are always UTF8 with escape codes: \n \u00E9"
bar = """multi-line strings
use three quotes"""

baz = 'literal\strings\use\single\quotes'

bat = '''multiline\literals\use
three\quotes'''

int = 5 # integers are just numbers
float = 5.0 # floats have a decimal point with numbers on both sides

date = 2006-05-27T07:32:00Z # dates are ISO 8601 full zulu form

bool = true # good old true and false
```

Listing 1. TOML-language datatypes [11].

Unlike JSON, TOML uses tables instead of objects. These tables can be nested. In TOML all nested tables must have their parent mentioned in their name because it does not require additional spaces between tables or indentations. All tables must be named accurately for the language does not use ordering to parse them. [11.]

```
[[comments]]
author = "Nate"
text = "Great Article!"

[[comments]]
author = "Anonymous"
text = "Love it!"
```

Listing 2. List of tables in TOML [11].

TOML tables can be formed into lists as shown in the above Listing 2. This resembles the JSON-object given in Listing 3. This comparison brings to light how much easier TOML is to read and write compared to JSON.

```
 {
      "comments" : [
            {
                  "author" : "Nate",
                  "text" : Great Article!"
            },
            {
                  "author" : "Anonymous",
                  "text" : Love It!"
            }
      ]
}
```

Listing 3. JSON-object [11].

2.3 Parsing algorithm

A parsing algorithm will be used to gather data from the diagram sent onto the server and saved into a database. For this, Rappid supplies two algorithm options to be used. The Breadth-first-search (BFS) and Depth-first-search (DFS) algorithms. BFS traverses given elements one by one in layers from left to right until the specified element is found or all elements have been checked. While traversing layers, the algorithm requires knowledge of which elements to check when going to the next layer. For this purpose, the algorithm uses a first-in-first-out type queue to save pointers to the child elements of the current layer. This way of traversing requires memory where the queue will be stored on runtime. Figure 5 visualizes the order of the BFS-algorithm. [12.]

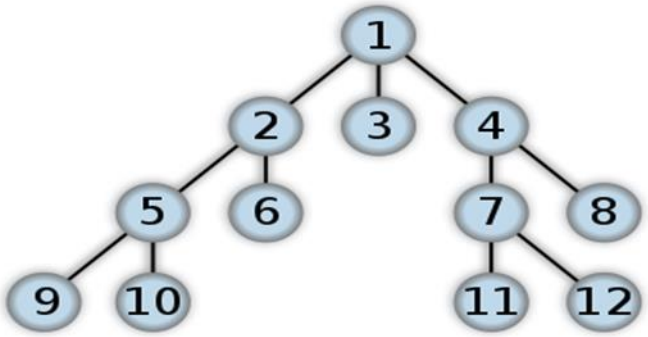

Figure 5. Breadth-first-search algorithm order [12].

DFS iterates through elements starting from the left and moving downwards until it hits an element with no child element. After this it moves back to the last ancestor with an unchecked child and repeats the previous process to the child element. This goes on like with BFS until the specified element is hit, or all elements have been checked. With the way DFS iterates through elements, there is no need to use memory to store pointers to child elements. This makes it more memory efficient than BFS. [12.]

Using a family tree as an example the differences between the two ways of iterating can easily be compared. For instance, if the person being searched for, is still alive, they are closer to the bottom of the tree making DFS most likely the faster algorithm to find them. In comparison if the person has died a long time ago, BFS has a higher chance of finding them first while going down layer by layer. The algorithm used for the project is DFS due

to it being more memory efficient. As the order does not matter, only the memory usage influenced the choice. Figure 6 gives an example of the iteration order of DFS. [12.]

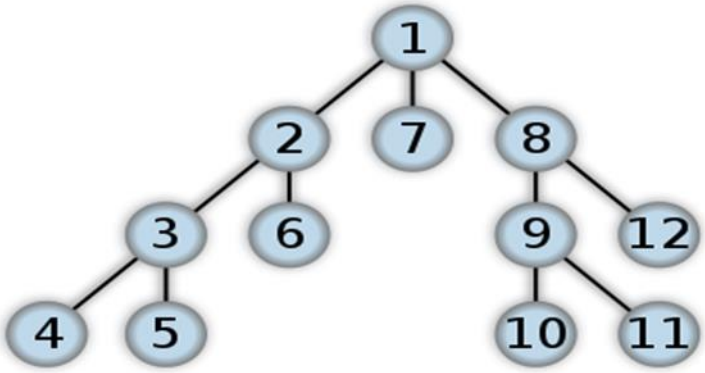

Figure 6. Depth-first-search algorithm order [12].

2.4 Chatbot

Unlike many of the chatbots found around the internet, the chatbot the projects company uses is not created with machine nor deep learning. It is a traditional FSM (finite-state-machine) [13] that was common practice when chatbots first started appearing in chat applications like IRC (Internet Relay Chat). The chatbot can be used for instance in minimizing the amount of people asking for help for subjects that can be found from the website they are visiting or to increase sales in web shops. This is done by giving the customer predefined paths that will guide them towards the information they are looking for without the need of a chat agent's help. This on its own will bring the company profits by lowering the amount of chat agents that need to be available during any given time. The chatbot is done in the traditional FSM way because it is easier and faster to deploy without running into the risk of the bot giving customers questionable or wrong answers.

When talking about self-thinking autonomous bots creating them into a controlled environment with a clear goal in mind gives the highest chance of success. Having a predefined format of messages and a possible maximum length limits the amount of information the bot needs to handle at once. Having an accurate image of the possible topics and objectives that the bot will be dealing with is necessary for an accurate performance. Sorting out the phrases and words send to the bot by relevance will increase the

correctness of the answers it gives. Defining multiple answers for the same keywords or phrases will increase the quality and human-feel of the bot. [14.]

Contrary to the previous chapter, a chat will not have a predefined maximum length or format for the messages that the chatbot would receive. This would make figuring out the important parts of the message more complicated. While a web page usually has a limited number of topics that the user can ask about they can differ from each other greatly. Having predefined paths and options given to the user guarantees that they will be guided more accurately to the section of the webpage that holds their answer. A state-machine can be used to create a very controlled environment that is modifiable by the company providing it. One benefit of a traditional chatbot is that it is ready to go the instant it is deployed to the web without having to use time to train and modify it before giving it out to customer use.

3 Chatbot-editor application

The chatbot-editor application is the core part of the project. All the end-user's modifications to the chatbots will be done on it. The core features of the application and the main development phases that were done during the project will be discussed in this chapter.

3.1 Update to ECMAScript 6 and TypeScript

Rappid source code and demo applications were all written in standard ES5 JavaScript during the project. The company had moved onto ES6 and TypeScript, so the chatbot-editor application was also made using these newer technologies. After the evaluation process, before any real work was put into the project, the editor code had to be refactored to use these technologies. As browsers do not natively support ES6 or TypeScript, any code using their features need to be built and transferred over to ES5. For this, a builder library had to be chosen. Possible options already used inside the company were either Grunt with Babel and Browserify or Webpack.

Browserify is a simple tool to handle npm packages but that simplicity also comes with a negative side effect. Browserify does not have a way to bundle, minify or lint JavaScript. This requires the use of a build automation tool such as Grunt with a list of attached

transforms and plugins. Grunt works great but requires a hefty number of configurations to work properly. [15.] This requirement for multiples tools pushed away from using them and shifted the decision towards Webpack, a single tool to do all the before mentioned build processes. Webpack uses a single configuration file to describe what the build process should accomplish.

Webpack can not only handle all the required JavaScript files, but it can also work with CSS and image files. Webpack gives the opportunity to include CSS and image files like you would with JavaScript files. It will dynamically inline stylesheets when they are small enough and otherwise minify the file. This can also be done for images using the URL loader plugin. The building process can also split resources into bundles to lower the size of served files on webpages. This is useful in larger Single Page Applications (SPA) were only a part of the entire JavaScript is required for the page to function. This more complete package can reduce the time required to set-up the build process. Webpack configurations are also relevantly short meaning less room for error and configuration debugging. [15.] These were the main reasons to pick Webpack over Grunt and Browserify in the project.

After the build automation tool was chosen, the code refactoring could be started. ES6 uses object class structures unlike ES5 so the JS code had to be transformed from one monolith file into multiple classes separating the existing features up. The components of the demo application were split into separate classes with their original functions transformed into class methods. JointJs is the core library that Rappid has been built on and all methods used by Rappid require JointJs to work. The original code held JointJs in the global namespace “joint” and splitting the code into parts broke this functionality. Using global variables goes against the common methods of ES6 so this created an issue but since at the time Rappid did not natively support TypeScript, the best option was to declare joint as a variable for each class that needed it and assign the global value for it. This enabled keeping all the core functionalities built into the demo app while still refactoring the code into a newer format.

In the original demo application, the beginning state was initialized entirely when the application was started, and the state was held in different components inside the joint global namespace. All components were initialized one by one in a specific order to prevent any dependencies between components from breaking. To preserve this structure of dependent components, all ES6 classes were made singleton to assure that only one

instance was made and that the same instance was bound to all components that depended on it. The original initialization functions were called inside each class' initialize method that is called in index.js on application start-up. Once the classes were refactored into their own files, the Webpack build configuration was updated to include the new JS files as one bundle served on the application web page.

The application uses configuration files to list its properties used during component initialization. The properties are built from JS objects and were tied to the joint namespace. During the change to ES6 these property objects were tied to constants and bound to components during start-up. The shapes used by the application were all listed in one file and were all tied to the joint namespace. Originally during the refactor these were all split into separate files for clarity and removed from the namespace to be served from constants. This created errors on runtime because the application could not find the shape definitions anymore. While trying to figure this problem out, a forum post was found that discussed this same issue that was answered with the information that the shapes must reside inside the joint.shapes namespace or the applications graph and paper components will not find them. [16.]

3.2 Configuring chatbot-editor shapes

The shapes that are used to create the visual flow of the chatbot are the most important part of the entire project. These are the core feature of the editor and all the features around them are there to make it from the visual representation of the chatbot flow to the TOML configuration and from there to the actual chatbot on clients' webpages. The main principle when designing and developing the shapes was to manage the line between shapes that are easy for users to understand and simultaneously relatively easy to transfer into the required TOML format. This lead to the conclusion of trying to mimic the look and format of the actual chatbot closely while designing the way the shapes look and work. Following this decision, the singular view of the bot shown in Figure 2 was copied to the way the shapes would be combined. One shape would be used to represent the container in which the text and buttons would be nested inside. Then the singular shapes representing the different features of the bot would be placed inside the container in the order they should be shown in the chatbot. Figure 7 gives a visual example of the container with nested shapes.

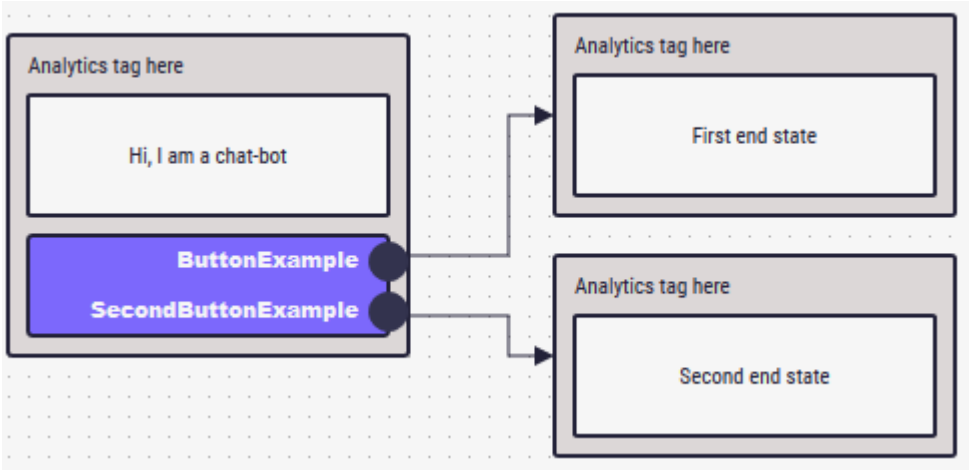

Figure 7. Chatbot-editor shape container example.

The shape example given in Figure 7 will create a chatbot response with one message and two buttons leading to new responses with singular messages. All buttons inside one view will be placed inside a singular button shape with multiple ports that use text labels to configure the actual buttons text. Connections will be drawn between the ports and their targets to indicate what state will be transitioned into after clicking the button. Once the bot reaches a target with no connection forward, this being a shape with no ports, the chatbot reaches its end state. Figure 8 showcases how a branch action could be constructed in the editor. The main functionality of a branch action is to monitor a given service and route the user forward based on the state of the service. For example, this can be used to route customers into a call back form if the chat service is closed.

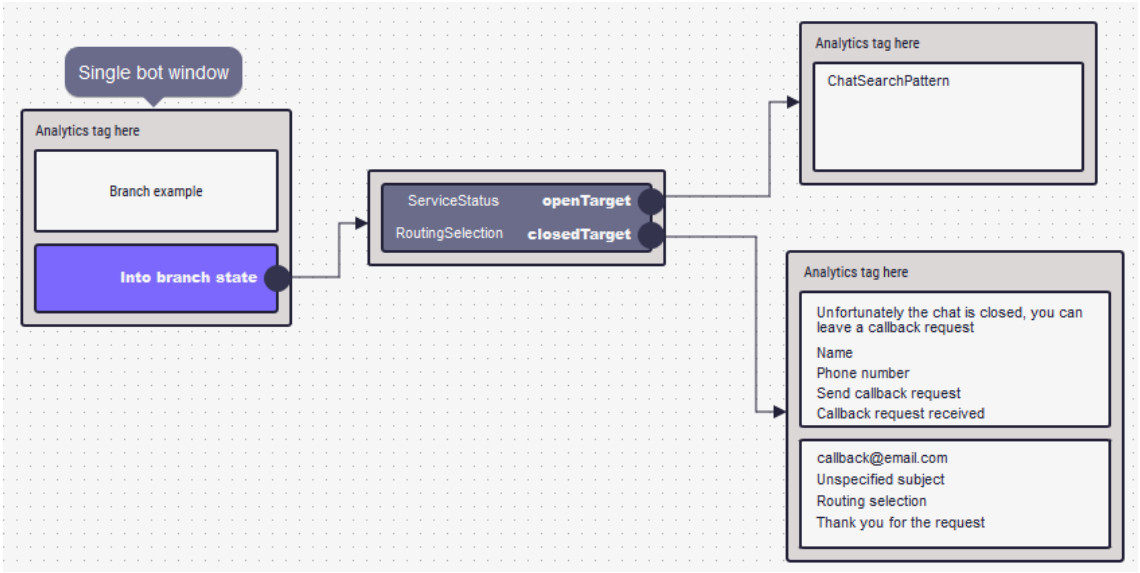

Figure 8. Branch action example.

When creating a branch shape, the editor gives the user freedom to customize the service to be monitored, the routing that should be used with the service and where the bot leads to, based on the state of the service. The service can be any service with states but for this example a chat service will be used. In a chat service, the routing selection can be used for instance to configure a group of agents who will be offered the incoming chat request. The service given to the branch action could also route users to the call back form when all chat agents are busy. The call back shape allows modifying all the texts shown to the user as well as the email configurations for the address the call back will be sent to. While the branch action is a more complex combination of shapes, the page change action illustrated in Figure 9 is a less complex set of shapes to create an advanced chatbot feature.

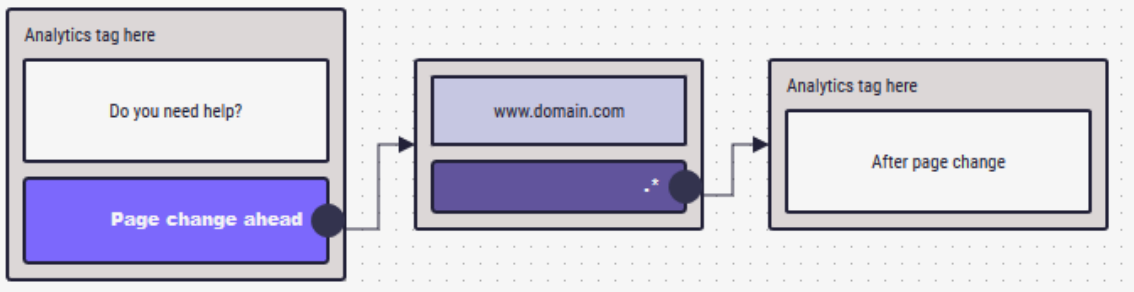

Figure 9. Page change action example.

A page change action does what its name indicates, it redirects the user to the page on the shape. After the redirect is complete the visitor will be shown the chat with the current state moved to the one targeted by the page change action.

The shapes went through multiple iterations during the project of how they should be created and used by the user. It started by creating singular shapes that make up all the combinations shown above. After creating the first container and a couple of singular shapes, the shape nesting logic was constructed. The core functionality of nesting shapes is to create events that trigger when a new shape is dropped on to the paper. These events watch for different shape types and act accordingly based on what shape was placed and where. The chatbot applies certain rules between shape-types that will be discussed in a future chapter. When a container was drawn on the paper, no nesting was necessary since it is the base shape that will contain child shapes. All shapes placed on top of the container will be nested as children for that container. During this phase, all shapes had to be repositioned and resized by hand to create a tidy looking diagram.

During the development of the project, this by-hand resizing, and repositioning was quickly found to be too cumbersome for the end-user. Shape templates were created that would transform into the most commonly needed shape combinations when dragged onto the paper. All the shapes shown above are ready templates that can easily be dragged onto the paper. The mechanism behind this is again multiple events that each catch their own shape and instead of the template shape being drawn on the paper, it is removed, and the event programmatically creates and resizes the shape combination intended for that template. To make the templates even easier and faster for the end-user to use, the singular shapes are resized and fitted nicely inside the container removing the need for additional user interaction.

Certain containers need the possibility to be expanded beyond just the shapes that come along with the template. Currently this is necessary only for the container with messages and a menu in it and the chat action. The chat action has an optional shape that can be added inside the container that holds information about a queue on the chat if it has been configured for the chatbot. If the user has dragged a container with just one message in it, they might want to add a menu action later as well. These additions created again the need to resize the existing shapes to fit them into the containers. This was an unwanted feature and events were created that would resize and refit the entire container when a shape was added into it. Now the end-user could drag and drop shapes onto the paper without having to modify their size while keeping the diagram tidy.

The chatbot comes along with a strict set of rules that need to be followed for it to work properly. The end-user using the application is not required to know these rules and thus the application must make sure that these rules are being enforced onto the set of shapes created by the user.

3.3 Chatbot rules

The chatbot actions follow a set of rules that dictate how it works and what actions can be combined with other actions. The very basic set of actions is the message-menu actions that will only work together inside one container. The branch-action cannot be combined with other actions and so the application will disallow placing other shapes inside its container. The page change action is technically one action that does one thing but as it is constructed of two parts, it was separated into two shapes for clarity. Bot of the

actions inside the page change container are mandatory. The chat action can either consist of only the core shape or it can be combined with an optional shape for added configurations. The call back action is split into two sections, one detailing the visual side while the other controls how the call back is handled. Both shapes are mandatory. Nesting sets of actions (containers) inside each other is not allowed.

As mentioned in the previous chapter, the shapes are created on the paper using events that either modify templates into correct shapes or place basic shapes onto existing containers. These events are handled by Rappid's jointjs paper and graph components. The graph catches the events of adding and removing shapes to and from the paper. It also catches all modifications done on the paper [17]. The paper mostly catches events created by the user, for example if the user clicks on an existing element on the paper [18]. Because the add event of the graph is one event that catches all additions to the paper, there is no way within the library to tell the event to only listen to the addition of a specific shape or element. This means that all functions listening to the add event will trigger when there is a new addition. Checking for what shape was added had to be done inside the function itself to prevent it from executing incorrectly for all additions. An example of this check is shown in Listing 4.

```
Paper.graph.on('add', (cell: any) => {
     if(cell.get('type') == 'app.MessageAction' && cell.get('parent') != unde-
fined){
          … execute code on add here
          … for example, check if any rules are being violated
     }
))
```

Listing 4. Example code of catching a message action add event.

The code snippet above shows how the add event is caught and how the check for a specific shape type is being done. The event is being given the added cell as a parameter and this can be used to figure out its properties like the type or whether it has a parent or not. As shapes are not allowed to be placed outside of containers, all add event functions check that they have a parent after being added on the paper. Placing shapes against the rules is being checked in their own functions that will remove any incorrectly placed shapes and give the user an alert letting them know that their previous action was not allowed and the reason why. Listing 5 shows an example of this functionality.

```
Paper.graph.on('add'. (cell) => {
     const allowedContainerType = shapeRules[cell.get('type')].canBeInsideCon-
tainerType;
```

```
      if(allowedContainerType){
            if(cell.get('parent') == undefined) || allowedContainerType != Pa-
per.graph.getCell(cell.get('parent')).get('containerType')) {
                  Paper.graph.removeCell(cell);
                  incorrectActionWarning.open():
            }
      }
})
```

Listing 5. Removing incorrectly placed shape example.

The function in Listing 5 first checks if the added cell was one that can be placed inside a container. These were the message shape, the menu shape and the optional chat shape. If it was one of these shapes, then it will make sure that it was placed inside a container instead of on the paper and that it was placed inside the right container. The constant allowedContainerType will hold a string with a container type as its value. Currently the possible container types for these shapes are the basic and chat container. These container types along with other properties set for all the shapes are described in the shapeRules JS-object. This object holds properties for shapes that describe the rules bound to each shape and container. Moreover, to the allowed container type, these properties describe whether a shape can be embedded, if they are unique shapes inside one container or the type of a container.

The shape rules within the application went through multiple iterations before ending up in their current form. They started off by being hard coded into the rule validation functions where shape and container types were being checked. All the possible allowed shapes were typed out into the if statements. This was fine during the first tests of the rule validations when there were only a few different shapes and rules to be handled. As more shapes were added and with the knowledge in mind that there can be more shapes added in the future, it became evident that keeping the shape types hard coded would be impossible. The first refactor for this hard-coded system was to create arrays for shapes that shared the same rules. For instance, embeddable shapes were inside one array, unique shapes were held in their own array etc. This solution worked out for a while but as more shapes got added into multiple arrays it got difficult to maintain as more shapes and rules were added. The next iteration was the current shape rule object.

3.4 Persisting the graph and robot

Saving and storing the different robot versions went through a few iterations before settling to its current form. The starting point was to write the robots TOML content into a file resembling the way it was done with the chatbot before the editor project started. First tests of the client graph parsing and persistence were done with no real integration with the back-end server. The graph is parsed through using Rappid's DFS-algorithm and turned into one JSON-object that can then be transformed into a TOML string by the server. This JSON-object was logged into the browsers console and copy-pasted from there to a file on the server that would be used in the TOML-writer. The TOML-writer functionality will be discussed in detail in chapter 4. The parser goes through each container and each nested shape within the container step by step to build the JSON-object with key values matching the ones needed in the TOML string. The shapes are iterated through in the order in which the user has placed them on the paper. To make sure that they will come into the JSON-object in the order they are arranged on the paper, sort functions were used to order them by their y-axis coordinates.

Originally the application had two different save buttons, one for the TOML and one for the JSON representing the Rappid graph. For the graph saving part, Rappid already had a ready function to do this. The idea of having two different save buttons was to give the user the option to save their work for the graph without always saving the TOML too. This was later removed and turned into one button that would do both. The idea was that the graph and TOML version should always be in sync. The application will have a separate function to apply a created and saved robot into production so having possible incomplete versions is not an issue. Only having one button will also prevent the user from forgetting to save one of their version and in general removes the need to know about the two types.

Further down the development mode, a decision was made inside the company to start saving the graph and TOML versions into a database instead of using files for them. As the TOML string was not human written anymore there was no need to keep it in a human accessible file but instead could be stored in a database. This also simplified saving the graph JSON too. The big advantage of having the robot saved in a database is that saving related important information like the robot's name and version, who created it and when was it created, was made much easier. Saving multiple version while

persisting the old ones is more effective when it is additional rows in a database instead of additional files on our servers.

3.5 Future of the application

Currently the application works in a localhost development environment. The first upgrade to be done in the future is to get it to a live test server for our customers to be able to use it too. After those tests have been done, the next step will be to take it into production. Before it can be taken into production, the companies chat server needs to be modified by adding a feature that allows loading chatbot-robots from the database instead of a file. Having a possibility to preview the robot being currently edited in the application would also make the creation project more convenient. There is a high chance that before all these steps have been made, the chatbot has acquired more features that need to be implemented into the editor.

For now, the editor can only be used by one person per customer at a time. Depending on the customer needs, a concurrent editing possibility could be added later into the editor. This functionality is directly supported by Rappid which makes implementing it much simpler. This feature is low on the priority list as getting the editor into production is the highest priority and concurrent editing is not a requirement for that.

4 Java backend server

The chatbot-editor client application will be talking to a Java backend server that will handle client requests, data persistence and user authentication. The server is constructed of three main parts, a rest service, a hibernate connection to a database and the actual robot logic. The server runs an embedded Jetty server that contains all the components.

4.1 Robot

Robot entity is an object describing the content of the chatbot and holds information related to the editor graph. Robots will be created by the client application and stored into the database to be used with the chatbots. The Robot entity holds all the necessary

information required by the client application and the chatbot. The important ones are the name, version and TOML-content that will be used by the chatbot to load the correct robot-content based on the values contained in the webpage users chat request. The client stores information about the graph's details used to render saved graphs later. The user who saved the robot and the date it was saved is stored as it may be needed for problem solving. The robots are persisted into the database with the servers hibernate configuration. The request path going from the client application into the database is illustrated in Figure 10.

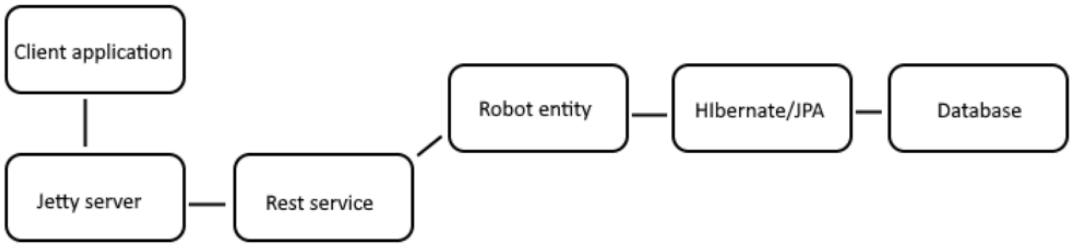

Figure 10. Request path from client to database and back.

The Java Persistence API (JPA) can be used to create repositories for database queries. Creating these repositories can be a tedious and long process with lots of boilerplate code. Even if an abstract base repository that provides CRUD operations for our entities and is created along with concrete repository classes for the entities the code for creating database queries must still be written for every query. The Spring Data JPA projects alleviates this problem by removing all this boilerplate code. Instead of being a JPA repository, the Spring Data JPA is a library adding an extra layer of abstraction on top of a JPA provider. It gives a way of creating JPA repositories by extending the repository interface. The library's repository layer contains three components, the Spring Data JPA, Spring Data Commons and the JPA Provider. [19.]

Creating a Crud Repository using Spring Data JPA can be done in two different ways, creating an interface that extends the CrudRepository interface or creating an interface that extends the Repository interface and adding the required methods to the interface [20]. The project uses the first methodology, creating an interface and extending the CrudRepository. This removes the requirement of writing any boilerplate code for the project's database queries. Before the CrudRepository interface can be used, the project needs a class representing an entity in the database. After this is done, the type of the

class and the type of the entity's id field can be given as parameters for the interface. [20.] An example of a JPA repository for the project is illustrated in Listing 6. This JPA repository is then configured to work together with the projects hibernate configuration that handles the connection between the server and the database in use.

```
public interface RobotDAO extends CrudRepository<Robot, String> {

    public List<Robot> findByDomainAndNameAndFormat( final String domain, fi-
nal String name, final RobotFormat format );

    public List<Robot> findByDomainAndNameAndFormatOrderByVersionDesc( final
String domain, final String name,
            final RobotFormat format, Pageable pageable );

}
```

Listing 6. JPA-repository request methods.

The servers spring framework is entirely configured with Java instead of an XML-configuration. The hibernate connection gets its connection details, database URL, credentials, etc., from a property file. These property files are environment specific to allow for separate database connections on production and test environments. An EntityManagerFactory is created as a bean into Spring that will set up the JPA repository using the created hibernate connection and entity classes from an appointed Java package. The JPA spring configuration and entity classes are in a project shared by the chat-server and the chatbot-editor server to avoid having to configure Spring JPA twice. Most of the requests to the database are done by the rest service working as an end-point for the client application.

4.2 RESTful service

The server's RESTful HTTP service is built with the aid of the Apache CFX service framework and its JAX-RS implementation. Apache CFX is an open source services framework that is used to make service development easier. It supports frontend programming APIs like JAX-WS and the earlier mentioned JAX-RS. The services support multiple protocols including SOAP, XML/HTTP, RESTful HTTP and Cobra. [21.] The CFX servlet is added to the Jetty Servers Web App Contexts (WAC) servlets with a servlet holder. The servlet is bound to */rest/** to handle any requests that are sent to that part of the server's context. This means that all requests by client applications to the rest service will be sent to the /rest context path. The CFX configuration is demonstrated in Listing 7.

```
WebAppContext wac = new WebAppContext();
ServletHolder cfxServlet = new ServletHolder( CFXServlet.class );
wac.addServlet( cfxServlet, "/rest/*" );
```

Listing 7. CFX servlet configuration.

After the CFX service is configured and the JAX-RS application API is tied to the CFX service through the projects Spring configuration. The JAX-RS server is given an application path inside the CFX servlet and Spring is informed that the server depends on CFX. The server's custom REST service class is then given to the JAX-RS server as a service bean. The REST service class implements methods that are annotated using javax.ws.rs annotations such as POST and Path. Listing 8 shows the servers JAX-RS configuration.

```
@ApplicationPath( "/robotbuilder" )
public class JaxRsApiApplication extends Application {}

@Bean
@DependsOn( "cfx" )
public Server jaxRsServer( ApplicationContext context ) {
     JAXRSServerFactoryBean factory = RuntimeDelegate.getInstance().cre-
ateEndpoint( jaxRsApiApplication(), JAXRSServerFactoryBean.class);
     List<Object> providers = (List<Object>) factory.getProviders();
     factory.setProviders( providers );
     factory.setServiceBean( new ApplicationRestService() );
     return factory.create();
}

@Bean
public JaxRsApiApplication jaxRsApiApplication() {
     return new JaxRsApiApplication();
}
```

Listing 8. JAX-RS server configuration.

4.3 Jetty server

Eclipse Jetty is used to run the project web server and the web application within it. There are two ways to configure a Jetty server. It can either be configured as a container for the web server and application to run in or it can be run embedded inside the web server. The chatbot-editor project will be using the embedded version of Jetty. Embedded Jetty is built inside the web server as a Java class that holds all the components and definitions of the server. The components of the project are the JAX-RS RESTful web service, a Jetty security authentication servlet and the web application context. All the servlets in

the server are added as handlers for the web application context. The rest service is bound onto a CFX servlet and strapped to /rest/* for the web app to use.

Initially all the servlets were to be added straight as handlers for the server, but this resulted in only the latest servlet being loaded onto the server. Thus, all the servlets had to be added as handlers for the web app context that was then added as a handler for the Jetty server. Spring framework was set as initialisation parameter for the web app with an event listener added for it to function from. The security handler used for authentication purposes was created with two constraints, one that blocked all access for non-authenticated users and one that allowed access for everyone. This was necessary to allow access for assets needed on the login page, such as images and CSS files. Both constraints and the login service, used for the authentication logic, was added to a security handler. The authentication configuration is demonstrated in Listing 9.

```
Constraint constraintNoAuth = new Constraint();
constraintNoAuth.setAuthenticate( false );

ConstraintMapping constraintMappingNoAuth = new ConstraintMapping();
constraintMappingNoAuth.setConstraint( constraintNoAuth );
constraintMappingNoAuth.setPathSpec( "/css/login.css,/assets/*,/heartbeat" );

Constraint constraint = new Constraint();
constraint.setName( Constraint.__FORM_AUTH );
constraint.setAuthenticate( true );

ConstraintMapping constraintMapping = new ConstraintMapping();
constraintMapping.setConstraint( constraint );
constraintMapping.setPathSpec( "/*" );

ConstraintSecurityHandler securityHandler = new ConstraintSecurityHandler();
securityHandler.addConstraintMapping( constraintMappingNoAuth );
securityHandler.addConstraintMapping( constraintMapping );
securityHandler.setLoginService( loginService );
```

Listing 9. Constraint mapping for the Jetty server.

The authentication mechanism uses a form authentication that will redirect unauthenticated users onto the specified login page and after being successfully authenticated, they are allowed access to the rest of the resources. If the authentication fails, they will be redirected to the error page. The server's authentication mechanism was created using Jetty security library. The login mechanism for users was created by studying the source code of existing login services of the library. The library included a mapped login service that functions by authenticating user credentials and saving them into a hash map after a successful authentication. This mapped login service was used as a base for the self-implemented login service used in the server. Extending the mapped login

service gave most of the required methods out of the gate leaving only the loadUser-method and user principle to be implemented. The structure of the authentication service is displayed in Figure 11.

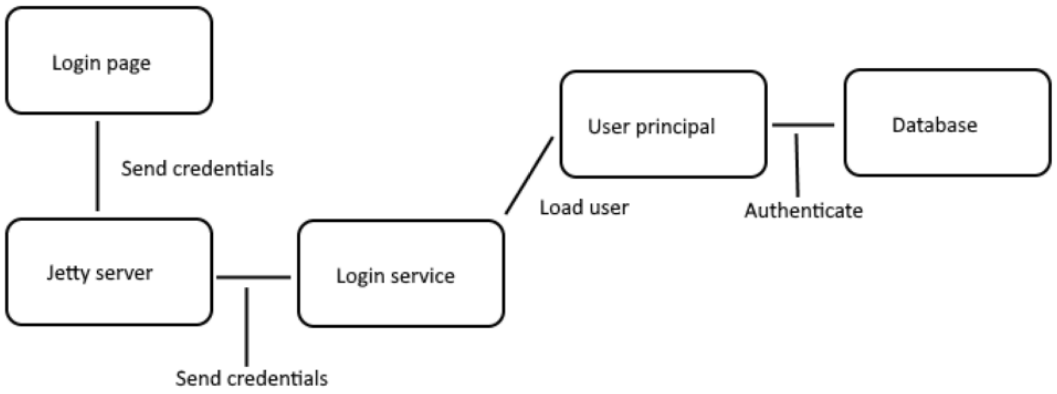

Figure 11. Jetty server login mechanism.

The loadUser-method takes the authentication credentials as parameters, this being for example a username and a password, and fetches an existing user that matches the credentials or creates a new user identity if none was found. The user identity holds all the information for users that the server might need when users are making authenticated requests. These are being held inside a user principal object given to the user identity object. The user principal object holds authentication credentials and handles user authentication. This is done by comparing encrypted credentials against known users in the database. If a matching pair of credentials was found, the user will be successfully authenticated.

4.4 TOML writer

The TOML writer on the server is using toml4j library [22]. It is a TOML parser made for Java that worked for this project. Creating a TOML object that can later be inserted into a file or a database is trivial with it. However, for this project, getting the TOML writer to work was not as simple. The writer takes a String-Object map that it then converts into TOML. The biggest issue was to figure out what the Object inside the map was supposed to hold inside for it to work. The project was originally using a TOML parser from The Electron Will [23]. Listing 10 displays an example of the map and write function used by

this library. It was later scrapped as the chatbot project already used the toml4j library, so it felt redundant to bring in a second library to do the same thing. Both libraries use the same data format making migration require only changing the write function in use.

```
Map<String, Object> data = ...//put your data in a Map
Toml.write(data, file);
```

Listing 10. Example of The Electron Will's TOML writer [23].

Learning how the data should be structured inside the Map was done by reading the source code of the Electron Will's library source code and testing the write function with a test class in Java. Trying to build the required Object in Java ended up being complicated and unnecessary in the end. The Object inside the map is a set of nested arrays and objects within each other. Once some data was successfully placed inside the map and reading the output of the map was possible, it came clear that the data format used in the write function was very similar to that of JSON objects with objects inside arrays. After figuring this out, all it took to make the write work was to transfer a JSON object to a java map. The client application parses the graph into a JSON object that is sent to the server's rest endpoint where it is turned into a map and passed onto the write function.

5 User Experience

The project was being built with the mindset of being mainly used by non-technical customers who do not understand the underlying logic of the chatbot. The user interface (UI) and client application logic had to be understandable by people who do not know how the chatbot works on the software side, but rather who understand how it should work on the client side for their customers. This meant that keeping the priority of user experience on the same level as the actual logic of the application was important.

5.1 User experience design

The first part of guiding the user to the right way of using the application, was through clear texts and visual indicators throughout the application. On top of these, a big part of making an easy-to-use application was to come up with tooltips that would clearly indicate what the different components do. After this, knowing that users would make errors

while operating the applications, clear guide lines and error messages needed to be added as dialogue boxes when the user does something unintended. For example, if the user tries to place a shape inside a wrong container, the shape would be removed to prevent the chatbot from breaking and the user would be informed that their action was not allowed. Preventing errors that are not caused by the user, such as network errors or browser crashes, from deleting the work done by the user is as important as preventing user errors from breaking the application and chatbot. To handle this, an auto-save feature that holds application snapshots, saved in the browser local storage was created, to prevent unsaved data from disappearing on errors.

A major part of the current features, made to improve the user experience, came when creating and testing the application. Every time a feature or way of using the application felt clunky or irritating, an improvement idea was added to the list of future development plans. This took the project step by step to a comfortable level of usability. Getting the opinions of other company members helped increase the variety of opinions for what would make the application nicer to work with. The big improvements to the applications UX is expected to come once the application is being tested by actual customers. To get some first impressions, before getting the application to customers, from a non-technical person, a user testing session was conducted with one of the members of the company.

5.2 User testing and feedback

At the end of the project, user testing took place in-house to get the first feedback from someone who was not part of the creation or design of the project. To get a more authentic representation of what the customers using the application would be like and how they would want to use the application, the testing was done by a non-technical person from the company. The testing revealed several key points that did not come to mind when creating the application, on how the users might want to use the application and what features they would be looking out for. The testing was done by giving the user a fresh blank start and telling them to create a simple chatbot-flow with the application. This was done to give a general idea of how the user would be interacting with all the different parts and phases of the application.

The first thing that came up from the user, was how to start working on the flow. The first steps were not clearly enough marked and guided and so the user felt lost in all the

options. This was partly due to too little visual differentiation between container shapes and shapes that would be added to these containers. The tester reported wanting to place the single message and menu options on the canvas first as they were the clearest shapes to be seen and the menu had a visible circle on it that would later work as the connector from menu to container. It became obvious that some clarification was needed between the container shapes and the shapes that would later be added inside containers. These shapes and container templates can be seen in Figure 12. Tooltips were not always clear enough on what should be placed where. An important realization was made during this, even though error messages informed the user of the mistake and how to correct this, the right way of using the application should be instantly clear from the tooltip to begin with. Because the tooltips were sometimes unclear, the user made mistakes, which lead to situations where the clarity and guidance of error messages were being heavily tested. The user gave feedback on the error messages on how they were clear enough to lead them to the correct way of working with the application.

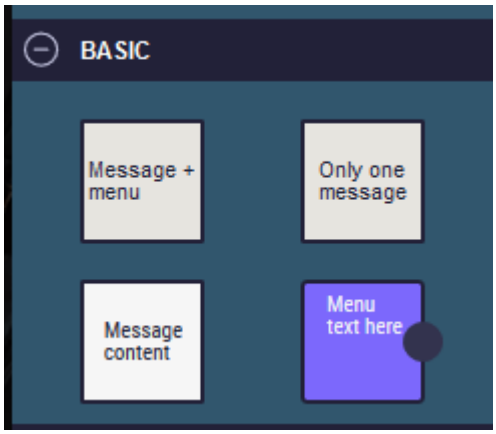

Figure 12. Basic containers and shapes commonly used with the application.

The UI in general was described as *"engineerish"*, being simple but mostly easy to understand and use. The tester reported being confused by some unnecessary tools on the applications toolbar and by being shown too many shape options from the get-go. Being only presented with the bare minimum to get going with the application would make it clearer on where to start and what are the most often used parts of the application. The suggestion from the user was to hide all other shapes but the basic ones used to create messages and menus, when the application was first launched. The toolbar has two empty dropdown menus at the start that will be later populated by chatbot name and version when the first iteration is saved. These were met with some level of confusion as to what their function was and after clarifying their purpose, the tester implied

that having some placeholders in them would clear the confusion. After the slightly slow beginning and some minor guidelines given, the tester got into making the very first parts of the chatbot flow.

This started by adding onto the canvas a container with one menu and one message. The first intuition of the user was to edit the text right where it read on the shape. This however is not yet possible due to technical limitations on the software. Instead, the texts are modified from the inspector on the right edge of the application. This was no problem once the user knew how it worked, but feedback has been given that being able to modify the texts on the shape, would be a nice addition. When modifying the texts, for the application to render the new text on the shape, the user must either click anywhere on the inspector or select another shape from the paper. This brought out the idea of adding a submit button on the inspector that would not have any function but would work as a clear place to trigger the text rendering from. After the texts were modified and it was time to move on to the next container of shapes, an interesting way of using the application emerged. The user wanted to first drag out the connecting link and then place the new container on top of this link. This does not work. Instead, the user is required to first create the second container and then connect the containers with a link. When connecting the shapes, the link cannot be placed on any of the shapes inside the container, it must be placed precisely on the underlying container shape. Even though there is a color highlight when the link is in the correct part, this still created some confusion and made the application unnecessarily complicated to use. Figure 13 illustrated this color highlight when connecting containers together.

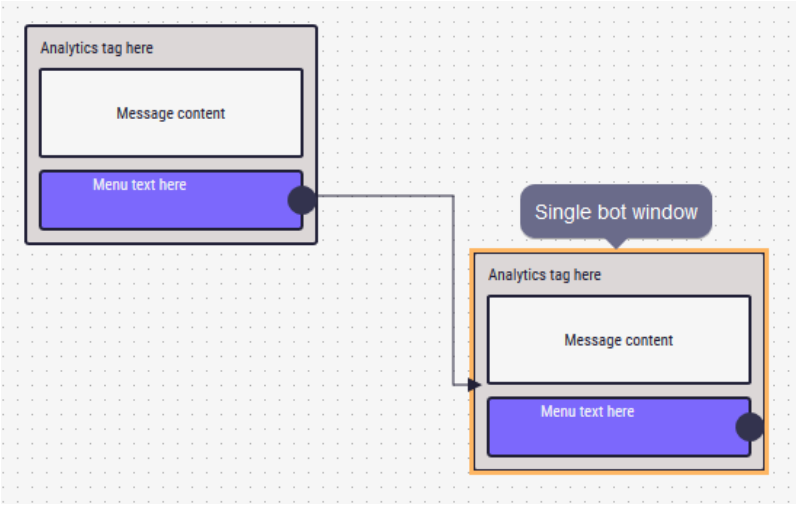

Figure 13. Connecting two shapes together with a link.

The summary for the application, after the testing, was that it is a well-designed tool in general with only minor issues that could be improved on. The application was relatively easy to start working with considering the level of complexity that can be done with it once learning how the components work. The UI is clear and for the most parts the user understood what the different parts did and were to generally start from. Small guidance was required in the beginning and based on the feedback that requirement could be removed by improving on the tooltips and clarifying some visuals. A user manual will be created upon customer request to give them guidelines to use while using the application. What comes to presenting the application to potential new customers, some simplification of the initial UI would be necessary to avoid overwhelming people who are unfamiliar with or have never seen the application.

5.3 Future-plans to improve user experience

The first steps to improve the UX in the future is to implement fixes for most of the issues found during the user testing session. Figuring out how to improve the experience of using the application for the very first time will be the first part of these additions. This will be tackled by improving the way different shapes of the chatbot will be presented to the user upon first login. Redesigning tooltips to better reflect what the shapes and components of the application do and how are they supposed to be used will come next. Modifying error messages will be given a lower priority as they seemed to be working fine during the testing. These will be looked at if customers using the application express that they are not clear enough. After this, the general usage of the application when placing and working with shapes on the paper will be modified slightly. Way of modifying texts in a shape will be changed, if possible, to be modifiable straight on the shape instead of in a toolbox on the side. The way links work and the order they should be used with the shapes will not be changed unless most customers feel negative towards how they currently work.

Improving on the design and usability of the application will really kick off only after it is being used by different customers with different opinions on how they would like to use it. At this point, a lot of feedback needs to be collected and sorted into a list of priorities and possibilities for what will be developed and when, if ever. Once the application is being tested by the first customers, a user study will be conducted with them to collect feedback. After the first version of the application is finished and deployed into

production, planning of the second big release version can be started based on customer feedback.

6 Summary and conclusions

The purpose of this thesis was to bring financial benefits to Lekane Ltd by decreasing the number of hours used creating and editing chatbot flows while also increasing the sales of the chatbot. To achieve these benefits, the editor application was designed to be user friendly and enhance the feeling that buying and using a chatbot can be straight forward and simple. The goal of the thesis was to create an application that could be used within the company as a first step towards easier chatbot configurations. This was achieved outside of some errors within the application that were left outside of the range of the thesis project and into future development. Getting the project into the hands of customers was dropped to future development of the application.

During the project, a large amount of knowledge was gained on how to plan, design and develop software projects that would be used by a company for several years after its initial release. A substantial amount of time should be used on preplanning and figuring out the requirements and implementation methods of the entire project before starting the actual development work. During the development of the project, keeping in mind that it will be developed and maintained by several people who were not part of the initial development, is vital. This will end up saving time and money for the company in the long run. Numerous practical and concrete software development methods were learned and utilized during the project. The project built a solid foundation for designing and developing future software projects.

The development of the application will continue within the company and once it is in a stable working state, it will be deployed to a test-environment for the company and customers to test it out. Once it successfully makes it through the testing phase, it will be deployed into production for customers to use.

References

1 Herz, Andreas. API Documentation. E-material. <http://www.draw2d.org/draw2d_touch/jsdoc_6/#!/api>. Read 31.10.2017.

2 Herz, Andreas. Shape Designer. E-material. <http://free-group.github.io/draw2d_js.app.shape_designer/>. Read 31.10.2017.

3 Herz, Andreas. Draw2D Commercial. E-material. <http://www.draw2d.org/draw2d/commercial.html> Read 31.10.2017.

4 www.telia.fi

5 Toolkit Edition. E-material. JsPlumb. <https://jsplumbtoolkit.com/purchase>. Read 31.10.2017.

6 Toolkit documentation. E-material. JsPlumb. <https://jsplumbtoolkit.com/community/doc/home.html>. Read 31.10.2017.

7 Build connectivity fast. E-material. JsPlumb. <https://jsplumbtoolkit.com/>. Read 31.10.2017.

8 Demos. E-material. Rappid. <http://resources.jointjs.com >. Read 30.10.2017.

9 Demos. E-material. Rappid. < http://resources.jointjs.com/docs/rappid/v2.1/ui.html>. Read 30.10.2017.

10 Product and Support Pricing. E-material. Rappid. <https://www.jointjs.com/pricing>. Read 30.10.2017.

11 Finch, Nate. 2014. Intro to TOML. E-material. <https://npf.io/2014/08/intro-to-toml/>. 16.8.2014. Read 2.11.2017.

12 What's the difference between DFS and BFS. E-material. Programmer Interview. <http://www.programmerinterview.com/index.php/data-structures/dfs-vs-bfs/>. Read 1.11.2017.

13 Finite State Machines. E-material. Brilliant.org. <https://brilliant.org/wiki/finite-state-machines/>. Read 28.4.2018.

14 Salto Martinez Rodrigo, Jacques Garcia Fausto Abraham. Development and Implementation of a Chat Bot in a Social Network. IEEE Xplore. Published 1.6.2012. Read 16.1.2018.

15 House, Cory. Browserify vs Webpack. E-material. <https://medium.freeco-decamp.org/browserify-vs-webpack-b3d7ca08a0a9>. 27.7.2015. Read 20.8.2017.

16 Steghofer Felix, dave. fromJSON problem. E-material. <https://groups.google.com/forum/#!topic/jointjs/0UMa-mKdexY>. 17.5.2014. Read 4.9.2017.

17 dia.Graph.events. E-material. <http://re-sources.jointjs.com/docs/jointjs/v2.0/joint.html#dia.Graph.events> Read 1.2.2018

18 dia.Paper.events. E-material. <http://re-sources.jointjs.com/docs/jointjs/v2.0/joint.html#dia.Paper.events>. Read 1.2.2018

19 Kainulainen, Petri. Spring Data JPA Tutorial: Introduction. E-material. <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/>. 30.11.2014. Read 6.3.2018.

20 Kainulainen, Petri. Spring Data JPA Tutorial: CRUD. E-material. <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-two-crud/> 24.1.2012. Read 6.3.2018.

21 Apache CFX. E-material. <https://cxf.apache.org/>. Read 26.2.2018.

22 mwanji. toml4j. E-material. < https://github.com/mwanji/toml4j >. Read 6.3.2018.

23 TheElectronWIll. TOML-javalib. E-material. < https://github.com/TheElectron-Will/TOML-javalib >. Read 16.8.2017