Frontend Code walkthrough

First of all, the home folder of the frontend is UML/ngUML.editor. Furthermore, react is used for the frontend with yarn as the package manager, in order to run you therefore need to download yarn. After navigating to the correct folder you can run the command “yarn” from your terminal and the app will compile. Once complete you can run the app with “yarn start” and a new window will open.

For this project most files were not created by us as our project builds for on another project. There are a few files that we changed/created:

-src/shell/NodeMenu/index.tsx
In this file we only added a navigation link so our menu can pop up, if you click on this link a class gets toggled that changes the width of our menu from 0 to a certain width with a sliding animation.

-src/shell/RuleMenu/index.tsx
This is one of the files that we created and is where most of our written code in the frontend is. This file could be understood best when you see it as four different parts.

The imports speak for themselves so first off we will focus on the RuleObject class. In order to keep the code maintainable and suitable for additional features without getting cluttered we decided to create a class for the rules in the frontend. This class is basically a struct with only a few fields and one method to get a string for debugging. Notice that this class also supports variables for type and python but these are not currently used.

Secondly there is the state, which is a variable that contains all the rules as javascript elements. If this state changes, all the elements on screen that use this variable will get refreshed, so in this way we can update our application without refreshing. After this line there is a line for the effects (useEffect), all the functions in this line will get called as soon as the application loads.

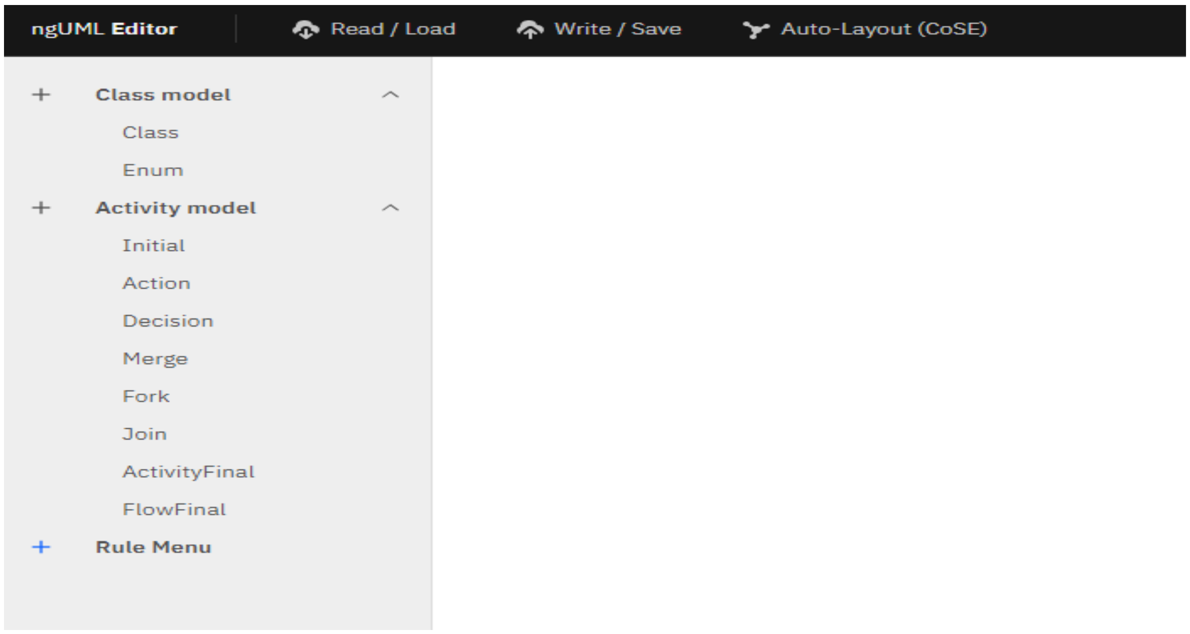
Thirdly you will see the functions. These functions all have self explanatory names so probably do not need much explanation. Most of the functions are basic calls to our api. Only the rulesToComponents is different. In this function we take an array of rules (of type RuleObject) and we translate these into javascript elements with buttons and a textfield, then finally we set our state to these elements.

Last but not least is our html part of the file. This is just basic HTML. There is one thing to notice and this is where our state gets used. Our state contains the javascript elements which are AccordionItems, in our HTML we then put this state between the Accordion tag. These Accordion items are imported from a library and this takes care of the design for us.

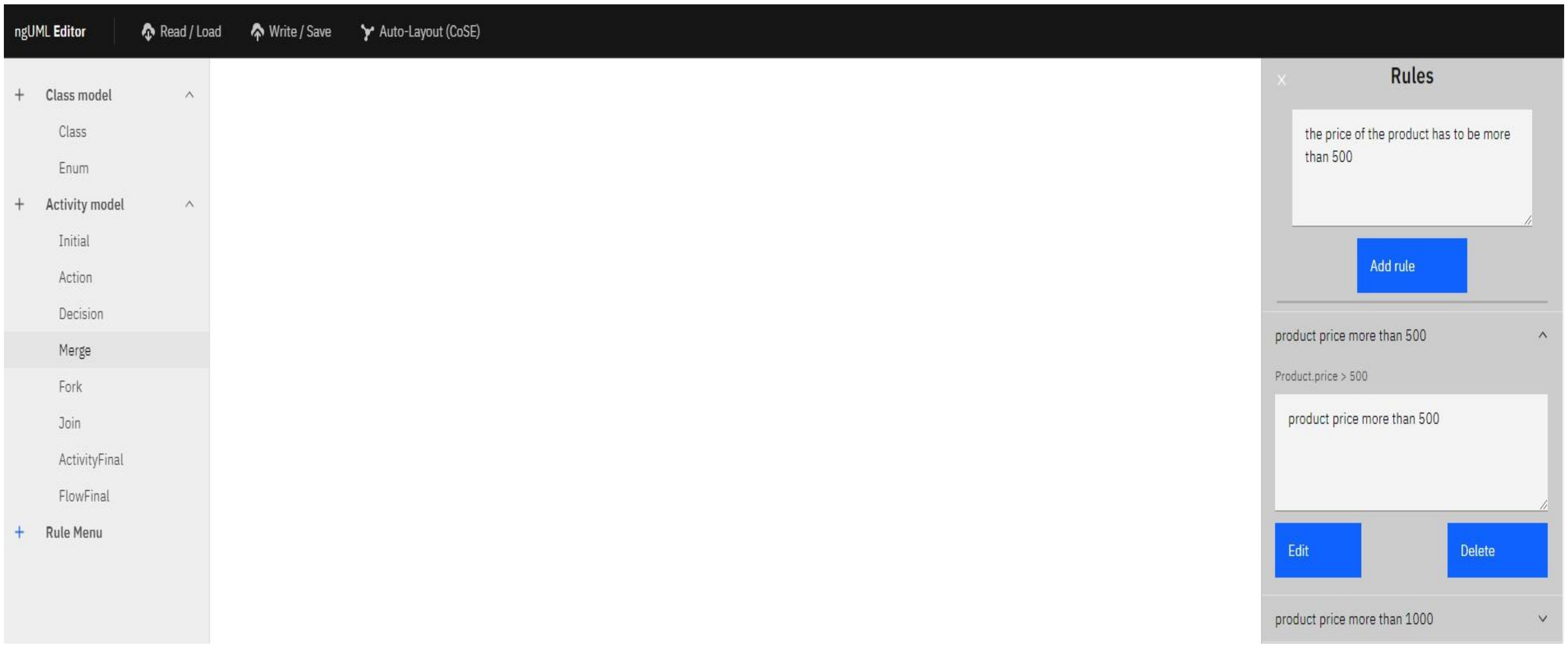
-src/shell/RuleMenu/_rulemenu.css
This file is also created by us and only takes care of some basic styling and element placement.

The graphical user interface

The first addition to the existing interface was the “rule menu” (see image). Which has been added in the left pop up menu underneath “class model” and “Activity model”. This means that you can add a certain rule to a relation within the UML model.



After many alterations we eventually got the following result. We added a text box in which natural language can be added. The natural language is processed in a shorter sentence.This rule can be edited with the left button, “edit”. The original rule will then be changed to the new rule in the textbox. If the added rule was a mistake or if you just want to get rid of the added rule you push the right button, “delete”.



the price of the product has to be more than 500

product price more than 500

Product.price > 500

product price more than 500

product price more than 1000

Backend Code Walkthrough

The processors file contains two files: ‘TextProcessor’ and ‘ValidatorProcessor’, Textprocessor translates natural language into a class called ‘Termlist’. Textprocessor first changes every word into a basic word, so for example ‘seventeen’ is translated to ‘17’, ‘products’ is translated to ‘product’. It also does a basic spell correction. After that it loops through every word and tries to see if the word matches a classifier, a property, a numeric value or an operator (these are called ‘terms’). If it matches any of these it creates the matching class and appends that class to the term list. Operators have a number of aliases, for example ‘<’ has the aliases ‘less than’, ‘fewer’ etc. If it scans an alias it replaces this alias with the operator.

The termlist is then passed to a function that tries to match any syntax to the term list. For example the syntax of a numerical rule is one property, one classifier, one value and one operator. If the operator is also in the list of allowed operators the function returns a rule object from the numerical rule class with the termlist. Every rule class implements a number of methods defined in the class baserules. For more information see the code comments in the ‘BaseRule.py’ file. The methods implement basic functionality such as adding validators, removing validators and returning a more structured, streamlined version of the original input.

The RulesManager class implements functionality such as adding and removing rules from both the generated application and the database. The views in ‘views.py’ call these functions. There are four views:

- Index:
    - Returns all the rules as dictionaries.
- Add:
    - Add a rule to the database and the application. The body of the POST request should contain a field called ‘rule’ and the key of that field should be the rule text.
- Remove:
    - Remove a rule from the database and the application. The body of the POST request should contain a field called ‘pk’ and the key of that field should be the primary key of the rule you want to remove.
- Description:
    - Returns documentation of every implemented rule class. It takes the doc strings from implemented functions and variables to generate a dictionary with all the information about every class that inherits from the BaseRule. The rule class should also be located in the folder ‘RuleClasses’. This

view function could in the future be used for a help window to inform the user of the syntax of the implemented rules.

The database of the rules application only contains a single model called Rule. The model has a single property called ‘original_input’ containing the original text of the rule. When retrieving a rule from the database the developer should use the function ‘get_rule’ from RulesManager, as that function automatically converts the database object to an instantion of the appropriate rule class.

Short description of the code outside of the rules folder

A warning for future new developers: the code in this project generally lacks comments and explanations about the content in complicated variables such as a dictionary of lists. It takes a lot of time to debug and reverse engineer the code to find out what exactly is going on. Sometimes parameters are passed to functions that never use them or only use a very tiny part of the variable. For example variables that contain entire page requests are passed only for the receiver to use the name of a classifier that is contained somewhere in the request data. An improvement would be to pass only the classifier and rename the variable to more precisely represent the data it contains. This limits the confusion that could arise when an unfamiliar developer tries to figure out what a function does, and what data it should pass to the function.

Important for a new developer on this project is to know that the *model* application within the project provides a mapping of the UML diagram. Every single element of the diagram is saved there. For a lot of logic such as tree traversals the project uses this mapping instead of using methods or functions provided by django. Generally this allows for more possibilities but a new developer should also realize that this means that he will not change the generated models file in shared in the usual way. The developer should change the objects in the mapping instead of the text in the generated models file. It took us a while to realize this.

The ‘shared’ folder contains generated files such as the generated models file and the generated views. The ‘extractor’ folder translates the natural language into a UML diagram.

StanfordcoreNLP

The stanfordcorenlp part of the backend uses a lot of ram and slows down the developer’s computer a lot when running it without enough ram. Some of our team members’ computers couldn’t handle the software. To avoid this if someone does not need to run the stanfordcore you could comment out the stanfordcorenlp stuff from the docker-compose file. This will allow the developer to debug without being slowed down when the developer does not need to use the stanfordcore.