# **Business Rules Engine Implementation**

Table of Contents

[**Business Rules Engine Implementation** 1](#_Toc160553112)

[**Business Rules Engine (BRE)?** 1](#_Toc160553113)

[**What is JDM?** 1](#_Toc160553114)

[**What are the different kinds of nodes available?** 1](#_Toc160553115)

[**What is Zen Expression Language?** 2](#_Toc160553116)

[**How to implement BRE?** 4](#_Toc160553117)

It is important to know a few concepts before implementing the rules engine.

## **Business Rules Engine (BRE)?**

BRE short for Business Rules Engine is a software system that enables the definition, creation and execution of business rules in an application.

Its main purpose is to separate business logic from code making it easy for companies to manage their business logic without changing the core application.

## **What is JDM?**

JDM or JSON Decision Model is a modelling framework designed by GoRules to make it easier to visualize business rules.

The decision models are represented as graphs with each rule / decision being represented as a node. Each node is connected to another with an edge that works as a pathway.

The graph progresses from left to right with the Input node being the first node and the output node being the last. Data enters the input node, flows through each individual node where the data is evaluated and is sent to the output node which is the result of the decision-making process.

## **What are the different kinds of nodes available?**

Apart from input and output nodes, there are 5 node types in GoRules BRE.

1. **Decision Table Node**

It is one of the easiest ways to maintain complex rules. Inputs and outputs are laid out in a spreadsheet-like format. The inputs use Zen Expression Language which provides a wide range of operations.

Each input column value represents an “AND” operation. Each row of the table is evaluated from left to right. The output depends on the hitPolicy and rule matching.

|  |  |  |
| --- | --- | --- |
| **hitPolicy** | **Does any rule match?** | **Output** |
| First | Yes | Object |
| First | No | Null / undefined |
| Collect | Yes | Array of objects |
| Collect | No | Empty Array |

1. **Switch Node**

This node comes in handy when the decision logic is depended on inputs and when we need to separate the graph based on conditions.

The conditions should be written in Zen Expression Language.

The branching is dependent on the hitPolicy.

|  |  |
| --- | --- |
| **HitPolicy** | **Branching Strategy** |
| First | Graph extends to the first branch that matches condition. |
| Collect | Graph extends to all branches matching condition. |

1. **Function Node**

This node can be used to debug logic, modify input data or to remap existing rules.

They are JavaScript function snippets where input nodes are the function’s arguments. The node times out after 50ms.

1. **Expression Node**

Expression nodes convert input objects to alternate objects while using ZEN Expression Language. It requires 2 fields – Key and Value.

Key – the name of output property.

Value – the desired output in ZEN Expression Language.

1. **Decision Node**

Decision Node extends the existing decision models providing reusability and reduced complexity.

## **What is Zen Expression Language?**

Zen Expression Language is the expression language used in GoRules decision tables.

It supports many data types such as string, Boolean, number, array, date and time and context – a special datatype similar to JSON structure.

The data expressed in this language is evaluated in either one of these two modes: unary test and expression.

* **Unary Test Mode**

The expressions are evaluated to a Boolean value.

For example, in a decision table, if the value of a cell (usually represented by $) is 1 and the expression is 1,2,50 – the result is true which is a Boolean value. It has resulted in true because each comma separator is considered as an OR operator and the Zen Expression Language evaluates if value of cell i.e. 1 is equal to 1 OR 2 OR 50. As value of cell satisfied the unary expression, it resulted in true.

Other examples:

A screenshot of a computer code

Description automatically generated

* **Expression Mode**

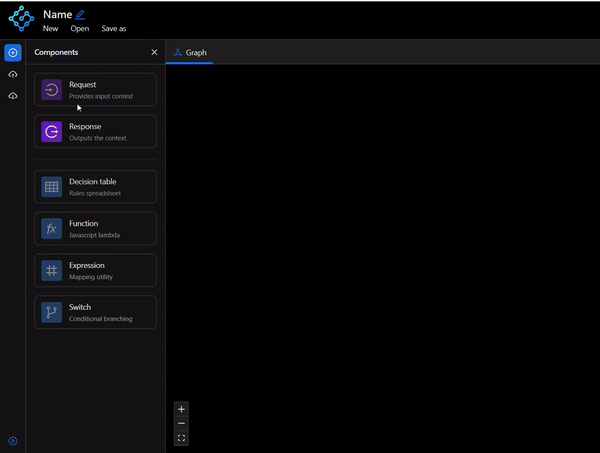
It is generally used to evaluate outputs or columns. All in built functionalities and operators of data types can be used.

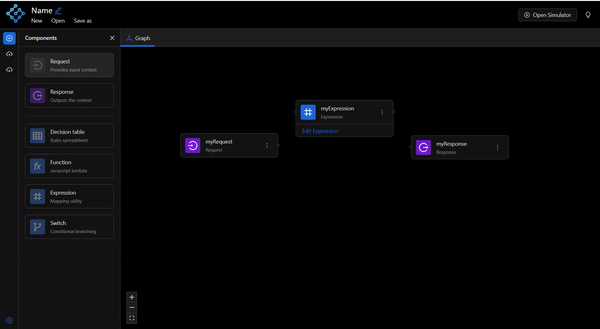
For example, if the rule is to verify if the number 3 exists in the array [88,44,5,1,9,3,7,6], the expression would be 3 **in** [88,44,5,1,9,3,7,6]

This expression uses the inbuilt **in** operator of the array datatype which evaluates to true if an element exists in an operator and false if it doesn’t.

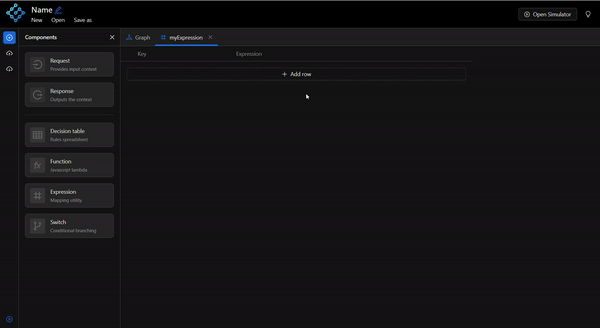
## **How to implement BRE?**

(Throughout this step, it is assumed that the rules engine is to be built on a Node application.)

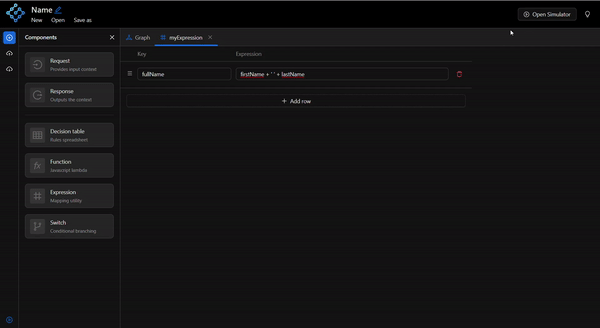
1. Create JDM graph using the free editor - <https://editor.gorules.io/>
   1. Drag and drop the required components
   2. Add edges for data to flow and click on the component to open a separate tab.



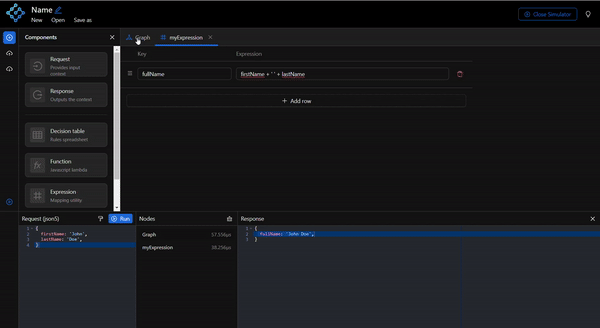
* 1. Add rows / input as per the requirement.



* 1. Open Simulator to test rule with sample input data.



* 1. Verify the data flow in graph tab (If node turns green, it symbolizes that the data has flown)



* 1. Download JDM graph in JSON format.

A screenshot of a computer

Description automatically generated

1. Install Zen Engine using your preferred package manager in your application.

NPM: npm i @gorules/zen-engine

Yarn: yarn add @gorules/zen-engine

1. Use the newly downloaded JDM JSON file and add it to your app folder.
2. Use the following code to execute the above decision.

const content = await fs.readFile("./graph.json");

const engine = new ZenEngine();

const decision = engine.createDecision(content);

const result = await decision.evaluate({

firstName: 'John',

lastName: 'Doe',

}

);

console.log(result);

Find detailed code and examples for each node in the repository – [Click here.](https://github.com/sarvani-b/gorules-bre)

In the demonstrated example, as it was a simple decision involving a single node, we have first created the decision using Zen Engine and then evaluated it. This approach might not be feasible for complex decisions involving multiple nodes. In such scenarios, we use a loader that expects a path to the JDM JSON file and a key which would be the input to the decision.

The JDM JSON file we have downloaded from the editor can be stored in the file system (as shown in the example) or in an AWS S3 bucket or could be a response from REST API or directly from database.