Expression Evaluator

Problem Statement

- Problem Statement

How to Start

- expression_evaluator.sh execute this script.

```
$ ./expression_evaluator.sh
```

- It will build the project and start the spring boot application.

How to access the rest API - Swagger-UI

- After starting the application Click on Swagger-home
- Sample Request is given below. Please use that for testing.

Solution

- Spring Boot based micro service.
- Rest end point /expression. Its a HTTP POST request.
- `API Request`
  - "expression" : Provide Array of expressions to be executed on the user.
  - "user" : User entity.

```
{
  "expression": [
["OR", ["IN", "event.category", ["infant", "child", "teen"]], ["LT", "user.age",
18]],
["AND", ["EQ", "user.address.city", "Los Angeles"], ["GT", "user.age", 35]],
["OR", ["EQ", "user.address.city", "San Francisco"], ["GT", "user.age", 35]],
["OR", ["EQ", "user.address.city", "Los Angeles"], ["EQ", "user.age", 35]]

]
,
  "user": {
    "address": {
      "address-line": "XYZ Street",
      "city": "San Francisco",
      "state": "CA",
      "zipcode": "94150"
    },
    "age": 35,
    "event": {
      "category": "infant"
    },
    "first-name": "Jhon",
    "last-name": "Marley"
```

```
    }
}
```

- `API Response`

```
{
  "returnCode": 200,
  "message": "Expression evaluated successfully.",
  "response_code": 200,
  "result_body": {
    "[OR, [IN, event.category, [infant, child, teen]], [LT, user.age, 18]]": true,
    "[OR, [EQ, user.address.city, Los Angeles], [EQ, user.age, 35]]": true,
    "[OR, [EQ, user.address.city, San Francisco], [GT, user.age, 35]]": true,
    "[AND, [EQ, user.address.city, Los Angeles], [GT, user.age, 35]]": false
  }
}
```

- Operators are implemented using `Command Deisgn Pattern`

```
/**
 * @author rohitkumar
 * creation date 22/07/18
 * project name expression-evaluator
 */
public interface Operator<L, R> {

    /**
     *
     * @param left operand
     * @param right operand or value
     * @return true or false
     */
    boolean execute(L left, R right);
}
```

- OR operator implementation. Rest of the other operators also implemented in the same way.

```
/**
 * @author rohitkumar
 * creation date 22/07/18
 * project name expression-evaluator
 */
public class Or implements Operator<Boolean, Boolean> {

    @Override
    public boolean execute(Boolean left, Boolean right) {
        return (left || right);
    }
}
```

- Operator Instantiation is handled using Factory Design Pattern. OperatorFactory
- `TREE` data structure is used for expression tree. Class - Tree

- `Pre-Order` traversal is used for building the tree from expression. Class - ExpressionEvaluatorEngine `buildTree()` method.

```
/**
     * @implNote Build Binary Tree of {@link
com.expression.evaluator.operator.Operator} and {@link
com.expression.evaluator.tree.Operand} using Pre order Traversal.
     * Format - [ OPERATOR, OPERAND, COMPARISON_VALUE(S) ]
     *          ["AND", ["EQ", "user.address.city", "Los Angeles"], ["GT",
"user.age", 35]]
     *          ["OR", ["IN", "event.category", ["infant", "child", "teen"]],
["LT", "user.age", 18]]
     * @param user
     * @param expression
     * @return root node of the tree.
     */
    private static Node buildTree(User user, ArrayList<Object> expression) throws
EvaluatorExpressionException {

        Node root = null;
        if (expression != null) {

            /**
             * Check whether its operator or not
             */
            Object value = expression.get(0);
            if (OperatorNames.isOperator(value.toString())) {

                Operator operator = OperatorFactory.getOperator(value.toString());
                if (Objects.isNull(operator)) {
                    throw new EvaluatorExpressionException("Invalid Operator
Name,"+value.toString());
                }
                root = new Node(operator);
            }

            if ( (expression.get(1) instanceof ArrayList) &&
isExpression((ArrayList<Object>) expression.get(1))) {
                /**
                 * sub-expression, recursive call
                 */
                root.setLeft(buildTree(user, (ArrayList<Object>)
expression.get(1)));

            } else {

                /**
                 * "user.address.city" , "event.category"
                 */
                Operand operand = buildOperandFromExpression(user,
expression.get(1));
                Node leftNode = new Node(operand);
                root.setLeft(leftNode);
            }

            if ( (expression.get(2) instanceof ArrayList) &&
```

```
isExpression((ArrayList<Object>) expression.get(2))) {
                /**
                 * sub-expression, recursive call
                 */
                root.setRight(buildTree(user, (ArrayList<Object>)
expression.get(2)));

            } else {

                /**
                 * Substitution-Values
                 */
                Node rightNode = new Node(new Operand(expression.get(2)));
                root.setRight(rightNode);
            }

        }
        return root;
    }
```

- `Post-Order` traversal is used for evaluation of expression tree. Class - ExpressionEvaluatorEngine `evaluateExpressionTree()` method.

```
 private static Object evaluateExpressionTree(Node root) {

        if (root != null) {

            /**
             * leaf node will contains the operand and substitution values..
             */
            if (root.getLeft() == null && root.getRight() == null) {
                return root.getOperand();
            }

            /**
             * Evaluate left subtree.
             */
            Object left = evaluateExpressionTree(root.getLeft());

            /**
             * Evaluate right subtree.
             */
            Object right = evaluateExpressionTree(root.getRight());

            /**
             * Since All non leaf nodes are Operator, Now evaluate the operator.
             */
            if (left instanceof Operand && right instanceof Operand) {

                return root.getOperator().execute(((Operand)left).getValue(),
((Operand)right).getValue());

            } else {

                return root.getOperator().execute(left, right);
```

```
            }


        }
        return false;
    }
```

- `ExpressionEvaluatorEngine` is util class which provides all the tree traversal, expression evaluation methods.
- `Swagger-UI` is used for rest documentation and testing.
- Spring Junit and Mockito is used for Junt.

```
@RunWith(SpringRunner.class)
public class APITest {

    @InjectMocks
    private API api;

    private MockMvc mockMvc;
    private ExpressionEvaluatorService expressionEvaluatorService;

    @Before
    public void setUp() {

        MockitoAnnotations.initMocks(this);
        expressionEvaluatorService =
Mockito.mock(ExpressionEvaluatorServiceImpl.class);

        Field field = ReflectionUtils.findField(API.class,
"expressionEvaluatorService");
        ReflectionUtils.makeAccessible(field);
        ReflectionUtils.setField(field, api, expressionEvaluatorService);

        this.mockMvc = MockMvcBuilders.standaloneSetup(api).build();

    }

    @Test
    public void testEvaluateExpression() throws Exception {

        String apiRequestJson = FileUtils.readFileIntoJson(APIRequest.class,
"api_request.json");
        Map<Object, Boolean> map = new HashMap<>();
        map.put("[OR, [IN, event.category, [infant, child, teen]], [LT, user.age,
18]]", true);
        map.put("[OR, [EQ, user.address.city, Los Angeles], [EQ, user.age,
35]]",true);

        String message = "Expression evaluated successfully.";
        APIResponse<Map<Object, Boolean>>  apiResponse = new APIResponse(message,
HttpStatus.OK.value(), map);
        ResponseEntity expectedResponse = new
ResponseEntity<APIResponse>(apiResponse, HttpStatus.OK);

Mockito.when(expressionEvaluatorService.evaluateExpression(any())).thenReturn(map);
```

```
        this.mockMvc.perform(post("/expression")
                .contentType(MediaType.APPLICATION_JSON_VALUE)
                .content(apiRequestJson))
                .andExpect(status().is(200))
                .equals(expectedResponse);
    }
}
```

How to use swagger-

1. Open Swagger Home
2. Execute the post request using request json.
3. Server will return json response.

Cases not supported for now.

- Not operator not supported.
- Expression - ["AND", ["IN", "event.category", ["infant", "child", "teen"]], ["LT1", "user.age", 18]] is failing.

Thanks for your time.