# Union VMS - Rules

* [Presentation](#UnionVMS-Rules-Presentation)
	+ [How RULES works for FLUX FMC?](#UnionVMS-Rules-HowRULESworksforFLUXFMC?)
	+ [How to create new rules?](#UnionVMS-Rules-Howtocreatenewrules?)
	+ [Management of Rules](#UnionVMS-Rules-ManagementofRules)
		- [Explanation of the current system](#UnionVMS-Rules-Explanationofthecurrents)
		- [Requirements related to this contract](#UnionVMS-Rules-Requirementsrelatedtothi)
		- [Elements to be developed](#UnionVMS-Rules-Elementstobedeveloped)
* [Examples of Fact, Rule and Template Objects](#UnionVMS-Rules-ExamplesofFact,RuleandTe)
	+ [The Rules entity :](#UnionVMS-Rules-TheRulesentity:)
	+ [The Template entity :](#UnionVMS-Rules-TheTemplateentity:)
	+ [Fact Object (StructuredAddressFact in this case) :](#UnionVMS-Rules-FactObject(StructuredAdd)
	+ [Related to the xPath expressions :](#UnionVMS-Rules-RelatedtothexPathexpress)
* [Documents](#UnionVMS-Rules-Documents)

# Presentation

## How RULES works for FLUX FMC?

1. Rules module receive message from Exchange
2. Rules persists ID (message, reference, trip for further usage)
3. Rules enrich the message to apply the rules. Enrichment create "fact" objects.
	1. Enrich the Asset (must exist)
	2. Enrich the message ID (must be unique)
	3. Enrich with MDR (activity, gear, ... must be valid)
4. Rules check reception authorization in Subscription
5. Rules applies each rules one by one (rule engine is DROOLS)
	1. Validation is based on a template (a ".dtr" file in the template folder (Java resources)
	2. A template instantiates a fact (only one fact) and doesn't contain any business logic.
	3. Each rule is linked to a template
	4. Each rule is an expression (stored in the rules DB schema).
	Warning: the expression in the DB must be compatible with the rules Java library!
	5. Expression are evaluated at runtime and are based on a library of function available in the rules modules
6. If the message pass the rules, Rules send the message to Activity for persistence
7. Rules created the response and send it to Exchange

How to update a rule?

1. Update the rule values:
	1. expression (field "rules.rule.expression")
	Warning: the expression must use existing standard Java or customized functions with available "fact" properties
	2. message: information message (sync with MDM)
	3. error\_type: error code (sync with MDM)
	4. disabled: true or false
2. Refresh the rules using a REST endpoint "/unionvms/rules/rest/rules/reload"

## How to create new rules?

1. Add a row in the table "rules.rule" with value
	1. id: unique integer
	2. br\_id: rule code (ex: FA-L00-00-0001)
	3. expression: expression to evaluate. Expression must return true when the rule fails and false when the rule succeed.
	4. note: internal usage
	5. message: message sent back to the third party when rule fails
	6. level: not used information (ex: L00)
	7. error\_type: ERROR or WARNING. Warning is not blocking the message while error does.
	8. template\_id: reference id to the table "template"
	9. property\_names: key to retrieve the XPATH in the failing rules
	10. disabled: true or false
2. Refresh the rules using the REST endpoint

## Management of Rules

### Explanation of the current system

Each incoming or outgoing message is sent to the Rules modules for validation. About 700 business rules are implemented for FA and SALES.

For messages of type “FLUXFAReportMessage” or "FLUXFAQueryMessage", "FLUXSalesReportMessage", "FLUXSalesQueryMessage" after validation the system sends a “FLUXResponseMessage” to the third party with the result of business rules validation. Three kinds of responses are possible:

1. “OK”: All rules are “SUCCESSFUL”. All reports contained in the message are persisted.
2. “WOK”: At least one rule has a warning. All reports contained in the message are persisted.
3. “NOK”: At least one rule has an error. None of the reports contained in the message is persisted.

The process is similar for other types of messages. The Queries and the Response are not persisted as entity. Only the raw XML is persisted in the Exchange module.

For FLUX Fishing Activities messages, 2 types of messages can be received by the system. Here are the main flow's steps:

* **FLUXFAReportMessage** (data).
The system sends back a FLUXResponseMessage:
	1. Receive report (plugin)
	2. Log report (exchange)
	3. Validate report (rules)
		1. Standard validation rules
		2. **Execute Subscription** to validate authorisation (currently always "Yes")
	4. Persist report - if validation is OK (activity)Send response
		1. Generate response - with validation result (rules)
		2. Validate response (rules)
		3. Log response (exchange)
		4. Send response (plugin)
	5. **Execute Subscription** to transmit Report(s)
* **FLUXFAQueryMessage** (request for data).
The system sends back a FLUXResponseMessage and - if the query is OK - a FLUXFAReportMessage:
	1. Receive query (plugin)
	2. Log query (exchange)
	3. Validate query (rules)
		1. Standard validation rules
		2. **Execute Subscription** to validate authorisation. If no subscription exists, the query is rejected and the response is negative.
		3. **Execute Subscription** to fetch Report(s) data - if validation is OK
	4. Send response
		1. Generate response - with validation result (rules)
		2. Validate response (rules)
		3. Log response (exchange)
		4. Send response (plugin)
	5. **Execute Subscription** and Send report(s)
		1. Generate report (activity)
		2. Validate report (rules)
		3. Log report (exchange)
		4. Send report (plugin)

### Requirements related to this contract

The purpose of editing the rule is not to create new rules or to update the rule’s expression. This would be too risky and could break the system.

With the edition of Rules, an authorised user can:

* List all rules
* Filter rules
* Edit rule's properties
* Manage Sets of Rules
* In addition, the system must allow synchronising the rules with the EU's Master Data Register. Changes made in the "local" MDR will be over-written when synchronising with "central" MDM.
* When modifications are applied to the business rules or synchronisation with MDM has taken place, the system must take into account the updated data, so that changes are effective immediately to incoming messages.
* Any messages incoming while the rules are being reloaded must be held until it can be processed by rules.

**1.       List of Rules**

Under the menu Admin > Configuration, go to "RULES" tab.

Rules properties are:

1. reference: ex.: FA-L03-00-0038
2. active (or not): Yes/No
3. type: warning or error
4. level: the rule level (currently L00, L01, L02, L03 are in use)
5. sub-level: the rule sub-level (currently only 00 is in use)
6. message: message sent in the response
7. note: additional information
8. start/end date; validity period for a rule to be applied
9. dataflow(s) (see Set of Rules)

**2.      Filter rules**

A filter on the list enables to filter rules on all columns. Multiple columns can be filtered at the same time.

**3.      Edit rule's properties**

A user can change all properties of a specific rule.

Rules' expression must be adapted to take into account this new feature (see hereunder).

**4.      Manage Set of Rules**

A user can link a rule to one or more data-flow(s). From the list of rules, the data-flow column is editable as a "select multiple" widget that contains the list of possible dataflows. If multiple data-flows are registered in MDR, this information shall also be integrated in FLUX FMC as part of the synchronisation process.

Data-flows determine the set of rules that apply to the incoming/outgoing messages. In international agreements different rules and data exchange formats may apply. If no data flow is provided for a rule, it always applies.

The list of the different data-flows is defined in USM module in the "Communication Channel". Rules' expression must be adapted to take into account this new feature.

The current data model is one table with the rules' metadata (coming from MDR) and the Drools expression. The proposal is to split the table in two separate tables:

* Rules\_MDR (metadata): this table is editable in Flux FMC and overwritten on each synchronisation with the central MDM of DG MARE.
* Rules\_Expression: Drools expression that evaluate the rules.

|  |
| --- |
| **Rules\_MDR** |
| **Dataflow** | **Rule ID** | **Message** | **Type** | **Start** | **End** | **Active** | **Level** |
| FA | 001 | AAA | ERR | 01/01/2010 | 31/12/2020 | Y | 01 |
| NEAFC | 001 | BBB | ERR | 01/01/2010 | 31/12/2020 | Y | 01 |
| NEAFC | 002 | CCC | ERR | 01/01/2010 | 31/12/2020 | Y | 01 |
|  | 002 | DDD | ERR | 01/01/2010 | 31/12/2020 | Y | 01 |
| NAF | 002 | EEE | WAR | 01/01/2010 | 31/12/2020 | N | 01 |

|  |
| --- |
| **Rules\_Expression** |
| **Dataflow** | **Rule ID** | **Expression** |
|  | 001 | aaa() |
| NEAFC | 002            | bbb() |
|  | 002 | ccc() |
|  |  |  |

The system should apply each rule only once depending of the message's dataflow. The expression and returned message are both applied with priority if the dataflow is specified in the respective tables (Rules\_MDR and Rules\_Expression). In both tables the combination dataflow/Rule ID is unique with the possibility for the dataflow to be null. If a rule has different versions with different start/end date, it must have a different identifier.

Example of incoming Report and applied expression / MDR metadata:

|  |  |  |  |
| --- | --- | --- | --- |
| Dataflow | Rule ID | Expression | MDR (Message) |
| FA | 001002 | aaa()ccc() | AAADDD |
| NEAFC | 001002 | aaa()bbb() | BBBCCC |
| NAF | ~~001~~002 | ccc() | EEE |
|  |  |  |  |

**5.      New rules’ expression**

The idea is to let Drools evaluating all rules' expressions and adding in all expressions a function that returns the applicability (or not) of the rule.  The new rules’ expression would the like: isRuleApplicable(creationDateTimeString) && (old\_expression).

The expression checks that

1. The expression is compatible with the message's dataflow
2. The rule exist in MDR for the message's dataflow (rule is applied by priority if the dataflow is specified)
3. The rule is active in MDR
4. The report’s “creationDateTimeString” is between the rule’s start and end date in MDR

Diagram isRuleApplicable()

|  |  |
| --- | --- |
|

|  |
| --- |
| C:\caa7f3efafe6368eaa7cf8f2e787f7ff |

 |

In the above example, here is the expected result for different messages (all expressions are evaluated):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dataflow** | **Msg Date** | **Rule**  | **New Expression** | **isRuleApplicable() should returns** |
| FA | 01/01/2019 | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | Y - if failing error message AAA is sentN - DF is NEAFC for that expressionY - if failing error message DDD is sent |
| FA | 01/01/**2021** | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | N - rule 001 has expired for FA (see MDR)N - DF is NEAFC for that expressionN - rule has expired for FA (in MDR)-> Message is always "OK" |
| NEAFC | 01/01/2019 | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | Y - if failing error message BBB is sentY - if failing error message CCC is sentN - rule 002 exists for NEAFC specifically |
| NEAFC | 01/01/2021 | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | N - rule 001 has expired for NEAFC (see MDR)N - rule 002 has expired for NEAFC (see MDR)N - rule 002  exists for NEAFC specifically-> Message is always "OK" |
| NAF | 01/01/2019 | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | N - rule 001 is N/A for NAF (no correspondence in MDR)N - DF is NEAFC for that expressionN - rule 002 is not active (in MDM)-> Message is always "OK" |
| NAF | 01/01/2021 | 001002002 | isRuleApplicable()  && aaa()isRuleApplicable()  && bbb()isRuleApplicable()  && ccc() | N - rule 001 is N/A for NAF (no correspondence in MDR)N - DF is NEAFC for that expressionN - rule 002 is not active (and has expired) for NEAFC-> Message is always "OK" |

The technical implementation could cache data to prevent slower performance.

### Elements to be developed

This list of elements to be developed is indicative and not exhaustive.

1. New feature: Set of Rules
	1. Add an entity (and rule's property) to link a rule to one or many data-flows.
	2. Adapt back-end services
2. New feature: Start / End date
	1. Add 2 properties for a rule: start date and end date
	2. Add the start and end date in the existing MDR synchronization mechanism
	3. Adapt back-end services
3. Create REST API
	1. list rules (+ filter)
	2. activate a rule
	3. de-activate a rule
	4. update data-flows for a rule
	5. protect API with USM
	6. update all rule properties (type, message, note, start/end, dataflow, level, sub-level)
	7. synchronization with MDR, the local "cache" of MDM.
4. Create Web interface
	1. List of rules
	2. Filter
	3. Edit data-flow (select multiple)
	4. Edit rule's properties
5. Adapt Rules' expression. Add a new function in all expressions to take into account the new constraints:
	1. Rule's validity period based on the report's creation date
	2. Rule's active status
	3. Rule's data-flow

# Examples of Fact, Rule and Template Objects

It is the LIQUIBASE that populates the rule.rule and rule.template tables. Those are the 2 needed entities which will contain the rules definitions. Those will be needed during the deployment so that the whole drools engine can construct the needed artifacts.

#### The Rules entity :

@Entity
@Table(name = **"rule"**)
@ToString
**public class** Rule **implements** Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.**AUTO**)
    @Column(name = **"rule\_id"**)
    **private** Long **id**;
    @AttributeOverrides({
            @AttributeOverride(name = **"createdOn"**,
                    column = @Column(name = **"rule\_created\_on"**)),
    })
    @Embedded
    **private** Audit **audit**;
    @Column(name = **"br\_id"**, nullable = **false**, unique = **true**)
    **private** String **brId**;

    @Column(name = **"expression"**, nullable = **false**, columnDefinition = **"text"**)
    **private** String **expression**;
    @Column(name = **"note"**, nullable = **false**, columnDefinition = **"text"**)
    **private** String **note**;
    @Column(name = **"error\_type"**, nullable = **false**)
    @Enumerated(EnumType.**STRING**)
    **private** ErrorType **errorType**;
    @Column(name = **"message"**, nullable = **false**, columnDefinition = **"text"**)
    **private** String **message**;
    @ManyToOne(fetch = FetchType.**LAZY**)
    @JoinColumn(name = **"template\_id"**)
    **private** Template **template**;
    @Column(name = **"level"**, nullable = **false**)
    **private** String **level**;
    @Column(name = **"property\_names"**, nullable = **false**)
    **private** String **propertyNames**;
    **private** Boolean **disabled**;

#### The Template entity :

@Entity
@Table(name = **"template"**)
@ToString(exclude = {**"factRules"**})
@NamedQueries({
        @NamedQuery(name = **LIST\_ALL\_ENABLED**, query = **"FROM Template t LEFT JOIN FETCH t.factRules f WHERE f.disabled IS NULL OR f.disabled <> true"**)
})
**public class** Template **implements** Serializable {
    **public static final** String **LIST\_ALL\_ENABLED** = **"template.listAllEnabled"**;
    @Id
    @GeneratedValue(strategy = GenerationType.**AUTO**)
    @Column(name = **"template\_id"**)
    **private** Long **id**;
    @Column(name = **"template\_name"**, nullable = **false**, unique = **true**)
    **private** String **templateName**;
    @Column(name = **"fact\_template"**, nullable = **false**, unique = **true**)
    @Enumerated(EnumType.**STRING**)
    **private** FactType **type**;
    @OneToMany(mappedBy = **"template"**, cascade = CascadeType.**ALL**)
    **private** Set<Rule> **factRules**;

Each template can contain many rules. In this sense**, each template relates to one entity from the implementation document**, and each rule definition (from the implementation doc) relates to one Rule row (in the rules table)!

#### Fact Object (StructuredAddressFact in this case) :

**public class** StructuredAddressFact **extends** AbstractFact {

    **private** String **postcodeCode**;
    **private** String **postalArea**;
    **private** String **streetName**;
    **private** String **cityName**;
    **private** IdType **countryID**;
    **private** String **plotIdentification**;
    **private** String **postOfficeBox**;

    @Override
    **public void** setFactType() {
        **this**.**factType** = FactType.**FA\_QUERY**;
    }

}

When we receive the xml message we map it to the appropriate fact object. In this case we map it to an **StructuredAddressFact** object. This object is related to a particular template through the common field **factType** that both the classes have.

Drools understands **.drl** files. During deployment those files are created dynamically starting from template files that look like the following (Ex. **StructuredAddress.drt**) :

template header
tname
expression
brid
rulemsg
type
level
propertyNames

package eu.europa.ec.fisheries.uvms.rules.service.business.activity;
import eu.europa.ec.fisheries.uvms.rules.service.business.fact.StructuredAddressFact;
import java.util.Arrays;
global eu.europa.ec.fisheries.uvms.rules.service.SalesRulesService salesService;
global eu.europa.ec.fisheries.uvms.rules.service.MDRCacheRuleService mdrService;
template "@{tname}"
rule "Vessel Transport means @{tname} - @{brid}"
when
    $fact : StructuredAddressFact(@{expression})
then
    $fact.setOk(false);
    $fact.addWarningOrError("@{type}", "@{rulemsg}", "@{brid}", "@{level}", "@{propertyNames}");
end
end template

The following is the piece of code that starting from a template file and the rules related to it constructs a .drl file to be deployed in the Drooles engine (for each rule of the template 1 .drl file is created!) :

**private** Map<String, String> generateRulesFromTemplate(String templateName, String templateFile, List<RuleType> rules) {
    **if** (CollectionUtils.isEmpty(rules)) {
        **return** Collections.emptyMap();
    }
    InputStream templateStream = **this**.getClass().getResourceAsStream(templateFile);
    TemplateContainer tc = **new** DefaultTemplateContainer(templateStream);
    Map<String, String> drlsAndBrId = **new** HashMap<>();
    TemplateDataListener listener = **new** TemplateDataListener(tc);
    **int** rowNum = 0;
    **for** (RuleType ruleDto : rules) {
        listener.newRow(rowNum, 0);
        listener.newCell(rowNum, 0, templateName, 0);
        listener.newCell(rowNum, 1, ruleDto.getExpression(), 0);
        listener.newCell(rowNum, 2, ruleDto.getBrId(), 0);
        listener.newCell(rowNum, 3, ruleDto.getMessage(), 0);
        listener.newCell(rowNum, 4, ruleDto.getErrorType().value(), 0);
        listener.newCell(rowNum, 5, ruleDto.getLevel(), 0);
        listener.newCell(rowNum, 6, ruleDto.getPropertyNames(), 0);
        rowNum++;
    }
    listener.finishSheet();
    String drl = listener.renderDRL();
    **log**.debug(drl);
    drlsAndBrId.put(drl, templateName);
    **return** drlsAndBrId;
}

This explains the basics of how it all relates and how we construct the needed drools files for the engine to fire rules against facts.

#### Related to the xPath expressions :

There is an XPathRepository which basically can be imagined as a table in which each row contains one xPath entry. The following is that class :

**public class** XPathRepository {
    **public static final** XPathRepository **INSTANCE** = **new** XPathRepository();
    **private** Map<Integer, Map<String, String>> **xpathsMap** = **new** ConcurrentHashMap<>();
    **private** Integer **sequence** = 10000;
    **private** XPathRepository(){
        **super**();
    }
    **public void** addToMap(Integer sequence, String propName, String xpath){
        Map<String, String> xpathSingleMap = **xpathsMap**.get(sequence);
        **if**(MapUtils.isNotEmpty(xpathSingleMap)){
            xpathSingleMap.put(propName.toLowerCase(), xpath);
        } **else** {
            Map<String, String> newXpathMap = **new** HashMap<>();
            newXpathMap.put(propName.toLowerCase(), xpath);
            **xpathsMap**.put(sequence, newXpathMap);
        }
    }
    **public void** clear(Collection<AbstractFact> facts){
        **if**(CollectionUtils.isNotEmpty(facts)){
            **for**(AbstractFact fact : facts){
                **xpathsMap**.remove(fact.getSequence());
            }
        }
        **sequence** = 10000;
    }
    **public** String getMapForSequence(Integer sequence, String propName) {
        String xpath = **null**;
        Map<String, String> propsMap = **INSTANCE**.getXpathsMap().get(sequence);
        **if**(MapUtils.isNotEmpty(propsMap)){
            xpath = propsMap.get(propName.toLowerCase());
        }
        **return** xpath;
    }
    **public** Map<String, String> getMapForSequence(Integer sequence){
        **return xpathsMap**.get(sequence);
    }
    **public** Integer getNewSequence(){
        **sequence**++;
        **return  sequence**;
    }
    **public** Map<Integer, Map<String, String>> getXpathsMap() {
        **return xpathsMap**;
    }

Basically for each property of the fact object an entry is stored in the repository (above class) during the mapping (from xml to fact objects).

To have a unique way to get that xPath back when needed each fact object while being created has a unique **sequence** assigned to it (declared in the AbstractFact class)!

So, when storing an entry in this repository 3 elements are present :

1. The fact id.
2. The property name.
3. The xPath value.

       An example piece of code mapping each property of the xml java object (**StructuredAddress** to **StructuredAddressFact**) to a fact object + storing the xPath entries for each property :

**public** StructuredAddressFact generateFactsForStructureAddress(StructuredAddress structuredAddress) {
    **if** (structuredAddress == **null**) {
        **xPathUtil**.clear();
        **return null**;
    }
    String partialXpath = **xPathUtil**.getValue();
    StructuredAddressFact structuredAddressFact = **new** StructuredAddressFact();
    structuredAddressFact.setPostalArea(getValueFromTextType(structuredAddress.getPostalArea()));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**POSTAL\_AREA**).storeInRepo(structuredAddressFact, **"postalArea"**);
    structuredAddressFact.setCountryID(structuredAddressCountryIDValue(structuredAddress));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**COUNTRY\_ID**).storeInRepo(structuredAddressFact, **"countryID"**);
    structuredAddressFact.setCityName(getValueFromTextType(structuredAddress.getCityName()));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**CITY\_NAME**).storeInRepo(structuredAddressFact, **"cityName"**);
    structuredAddressFact.setStreetName(getValueFromTextType(structuredAddress.getStreetName()));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**STREET\_NAME**).storeInRepo(structuredAddressFact, **"streetName"**);
    structuredAddressFact.setPlotIdentification(getValueFromTextType(structuredAddress.getPlotIdentification()));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**PLOT\_IDENTIFICATION**).storeInRepo(structuredAddressFact, **"plotIdentification"**);
    structuredAddressFact.setPostOfficeBox(getValueFromTextType(structuredAddress.getPostOfficeBox()));
    **xPathUtil**.appendWithoutWrapping(partialXpath).append(**POST\_OFFICE\_BOX**).storeInRepo(structuredAddressFact, **"postOfficeBox"**);
    **return** structuredAddressFact;
}

As you can notice the xPath entries are built manually (because is orders of magnitude faster than Sax Parsers or other alternatives)!

When a rule fails the method **addWarningOrError(..)** is called. This method is implemented as follows (in AbstractFact.java) :

**public void** addWarningOrError(String type, String msg, String brId, String level, String propertyNames) {
    **final** List<String> xpathsForProps = getXpathsForProps(propertyNames);
    **if** (type.equalsIgnoreCase(ErrorType.**ERROR**.value())) {
        RuleError ruleError = **new** RuleError(brId, msg, level, xpathsForProps);
        **errors**.add(ruleError);
    } **else** {
        RuleWarning ruleWarning = **new** RuleWarning(brId, msg, level, xpathsForProps);
        **warnings**.add(ruleWarning);
    }
}

**private** List<String> getXpathsForProps(String propertyNames) {
    List<String> xpathsList = **new** ArrayList<>();
    **if** (StringUtils.isNotEmpty(propertyNames)) {
        String propNamesTrimmed = StringUtils.deleteWhitespace(propertyNames);
        String[] propNames = propNamesTrimmed.split(**","**);
        **for** (String propName : propNames) {
            xpathsList.add(XPathRepository.**INSTANCE**.getMapForSequence(**this**.getSequence(), propName));
        }
    }
    **return** xpathsList;
}

Basically when this method is called we have all the parameters needed to extract an xPath entry from the repository.

Namely we have the **property names** (since there could be more then one), and the **unique fact sequence**.

When the evaluation process has finished evaluating this message the XPathRepository is cleared.

The domain objects (rule/template among other entities):

<https://github.com/UnionVMS/UVMS-RulesModule-APP/tree/dev/domain/src/main/java/eu/europa/ec/fisheries/uvms/rules/entity>

The .drt definitions:

<https://github.com/UnionVMS/UVMS-RulesModule-APP/tree/dev/service/src/main/resources/templates>

The fact objects definition

<https://github.com/UnionVMS/UVMS-RulesModule-APP/tree/dev/service/src/main/java/eu/europa/ec/fisheries/uvms/rules/service/business/fact>

The xPath repository:

<https://github.com/UnionVMS/UVMS-RulesModule-APP/tree/dev/service/src/main/java/eu/europa/ec/fisheries/uvms/rules/service/mapper/xpath/util>