Attribute Management System

Author: @Ayush Jain
Status: Draft

Summary

The insurance product platform that we have developed till now is based on 3 main things: product components, attributes and rules. Most of these have been defined and managed by the dev team currently. We eventually want to handover some functionalities to product-team.

- Launching new products from scratch.
- Enhancing a product:
  - Launching new plans
  - Launching new functionalities
  - Enhancing existing functionalities
- Content changes
- Running experiments
- Defining concepts and complex conditional behaviours

Problem Statement

We need a dynamic context definition and rule based system to define product's business and other contextual info. The info can include various product components and their attributes. These attributes can be static or dynamic in nature i.e. they may have a fixed value or they may depend on other attributes through symbolic logic defined using constructs and expressions. This logic will be evaluated inside a context instance having the necessary attribute values from pre-evaluations or user input.

Component Driven Design

Component Based Development (CBD) is a sister technology to SOA; they are orthogonal, but sympathetic concepts; it is possible to have one without the other, but they best go hand-in-hand to produce component based systems that provide business services. For all practical purposes, service oriented architecture depends on component based development. Component based design is aimed, like object-orientation, at improving productivity by offering a better chance of reuse through better modularity. If it is done well, we can build large, complex systems from relatively small components. If combined with an agile development process and if there has been sufficient investment in components, this can lead to a faster development cycle.

A software component is an object that is defined by an interface and a specification. An object is something with a name (identity) and responsibilities of three kinds: responsibilities for remembering values (attributes); responsibilities for carrying out actions (operations/methods); and responsibilities for enforcing rules concerning its attributes and operations (often referred to as constraints). An interface is a list of the services that an object offers. A type is such a list plus the rules that the object must obey (its specification). Contrast this with the notion of a class. A class is an interface with an implementation. A class has instances; a component has implementations.

Why AMS?

In the end everything is a play over attributes which collectively define the context. The need is to give product team a tool to configure any kind of rule driven insurance product. In the v2 model we introduced components including rules which were required to model the basic product and compute the premium, next phase is to focus on attributes and their management in a way that enables generic product creation and upgradation in steps such that its easy and simple to configure for the tech functions required.

Assertions

- Attribute evaluation will never have cyclic (inter-attribute) dependencies.
- Product definitions will have a flat component structure.
- Each component will have a definition of required attributes for component/product functionalities.
- The rules used to manage attribute evaluation and logic constructs, will have templates and can evaluate multiple attributes at once. The output attributes must of the same component.
- Any dynamic attribute will be defined/evaluated only by a single rule.

Attributes and Functionalities

Attributes are basic building block of the product. A lot of these attributes will be defined by the product-team. The tech functionalities also depend on many of these attributes like premium-calculation depends on premiumAmounts defined for covers, discount, loading and tax coefficients defined by attributes.

The requirements from insurance-tech side are:

Tech implementation is based on above 3 things currently. When handing over configuration to product-team, we need to make sure whatever changes they do are well within their intentions and don't break the functional integrity of the systems. Hence welcome *functionalities*, which is how we'll constraint the scopes and intentions of changes.

Type of changes:

- Basic product definition (product, covers, discounts, loadings, taxes, plans)
- Add premium attributes and rules
- Add plans, covers, cover options, discounts (add component)
- Update app related content
- Add claims related attributes
- Add functionality
- Add functionality related attributes
- Add new features for only new client versions (content change for ex)
- Delete attribute/s

Data Model

Legend for below tables:
* - primary key/s
* - unique constraint/s
* - index/es
* - note/s

product:

property	type	desc
id	string	product id
status	string	DRAFT/PENDING_APPROVAL/ACTIVE/DISCONTINUED
effective_from	datetime	effective datetime
expiry_at	datetime	expiry datetime
template_type	string	INSURANCE
parent_product_id	string	parent product id (optional, if cloned)
description	string	description (optional)

product_approval:

property	type	desc
product_id	string	product id
approved_by	string	approved by person
discontinued_product_id	string	discontinued product id (optional)
change_description	string	description

product_functionality:

property	type	desc
name	string	function name like PREMIUM_CALCULATION
product_id	string	product id
immutable	boolean	TRUE/FALSE
description	string	description (optional)

^ *different products can have different required functionalities*

product_functionality_required_attribute:

property	type	desc
abstract_path	string	<component_type>(.<component_id>).<path.name>
functionality	string	functionality (references functionalities table)
product_id	string	product id
description	string	description

abstract_attribute:

property	type	desc
abstract_path	string	<product_id>.attribute.<component>.<(<component_id>).path.name>
product_id*	string	product id
display_name*	string[]	ex: path.name (to keep user defined attribute name)
component_type*	string	component_type (reference to component_type table)
component_id**	string	component id
tag***	string[]	add tags to search/group/organise attributes
datatype	string	datatype ex: object, array, string, int, bool, enum
enum	string	required when datatype is enum
references_attribute	string[]	list of attribute abstract_paths which are the possible value it can hold
constraint_expression	string	expression
immutable	boolean	TRUE/FALSE
description	string	description (optional)

*abstract_path: The advantage of using abstract path is that component templates can be defined. Say an insurance product has a component called plan which has associated attributes like name, covers, discounts etc. Now whenever have to create a new plan. We just need to create attribute instances using the abstract template. It'll also help in validation of values and associations.

attribute:

property	type	desc
path	string	<component_type>.<component_id>.<path.name> ex: PRODUCT.<id>.nested.path.attribute
abstract_path	string	<component_type>(.<component_id>).<path.name> (references abstract_attributes table)
rule_id	string	rule id (references rule table)
type	string	STATIC/DYNAMIC
value	jsonb	attribute value if static (optional)
product_id	string	product id

product_template_enum:

property	type	desc
name	string	name ex: CoverOptionValueType
product_template_type	string	product template type
value	string[]	values ex: ["DAY","CURRENCY","YEAR"…]
description	string	description

datatype:

property	type	desc
name	string	name ex: object, int, enum, array, bool, string, identifier, attribute_reference
	string	type ex: object, int, enum, array, bool, string
description	string	description

rule:

property	type	desc
id	string	id
type	string	rule function name (internal field) ex: GET_PREMIUM (grouping by functionality)
input_attribute	string[]	list of input attribute.paths
output_attribute	string[]	list of output attribute.paths
display_expression	string	input expression
expression	string	compiled expression
description	string	description (optional)

Why is there an abstract attributes table. To templatize and manage common attributes across a component type this table will help when you create new component instance as well as functionality attribute constraints.

Reserved Attributes

There would also be a list of reserved attributes, defined and used by tech either directly using the above model or in the functionality specific local contexts.

APIs and Data Flows

Create Functionalities:

```
------------------------------- datatype -------------------------------

request:
PUT /ams/v1/datatype
{
  "name": "<name>",
  "type": "<type>",
  "description": "<description>"
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
GET /ams/v1/datatype/<name>

response:
{
  "statusCode": 201/400,
  "data": {
    "name": "<name>",
    "type": "<type>",
    "description": "<description>"
  }
}

------------------------------- product_template -------------------------------

request:
PUT /ams/v1/productTemplate/{productTemplateType}/enum
{
  "name": "<name>",
  "value": ["<value1>", "<value2>", ...],
  "description": "<description>"
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
```

```
GET /ams/v1/productTemplate/{productTemplateType}/enum/{name}

response:
{
  "statusCode": 201/400,
  "data": {
    "name": "<name>",
    "value": ["<value1>", "<value2>", ...],
    "description": "<description>"
  }
}

------------------------------- products ------------------------------

request:
PUT /ams/v1/product
{
  "productId": "<productId>",
  "description": "<description>"
  "effectiveFrom": "<effective_from>",
  "expiryAt": "<expiry_at>",
  "templateType": "<template_type>"
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
PUT /ams/v1/product/{parentProductId}/clone
{
  "productId": "<productId>",
  "description": "<description>"
  "effectiveFrom": "<effective_from>",
  "expiryAt": "<expiry_at>"
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
GET /ams/v1/product/{productId}

response:
{
  "statusCode": 201/400,
  "data": {
    "productId": "<productId>",
    "description": "<description>"
    "effectiveFrom": "<effective_from>",
    "expiryAt": "<expiry_at>",
    "templateType": "<template_type>"
    "status": "<status>"
  }
}

------------------------------

request:
POST /ams/v1/product/{productId}/submit

response:
{
  "statusCode": 201/400,
  "data": {
    "productId": "<productId>",
    "status": "<status>",
    "description": "<description>"
  }
}

errors:
- product does not exist
- product already active
- missing required attributes

------------------------------

request:
POST /ams/v1/product/{productId}/approve
{
  "approvedBy": "<approved_by>",
  "discontinuedProductId": "<discontinued_product_id>",
  "changeDescription": "<change_description>"
}

response:
{
  "statusCode": 201/400,
  "data": {
    "productId": "<productId>",
    "effectiveFrom": "<effective_from>",
    "expiryAt": "<expiry_at>",
    "status": "<status>",
    "approvedBy": "<approved_by>",
    "discontinuedProductId": "<discontinued_product_id>",
    "changeDescription": "<change_description>"
  }
}

errors:
- product does not exist
- product already active or draft

------------------------------ fucntionalities ------------------------------
```

```

request:
PUT /ams/v1/product/{productId}/functionality
{
  "name": "<name>",
  "description": "<description>",
  "immutable": true/false,
  "requiredAttributes": [
    {
      "abstractPath": "<abstract_path>",
      "description": "<description>"
    },
    ...
  ]
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

errors:
- functionality already exists
- attribute does not exist
- product does not exist

-------------------------------

request:
GET /ams/v1/product/{productId}/functionality/{name}

response:
{
  "statusCode": 201/400,
  "data": {
    "name": "<name>",
    "description": "<description>",
    "immutable": true/false,
    "status": "<status>",
    "requiredAttributes": [
      {
        "abstractPath": "<abstract_path>",
        "description": "<description>"
      },
      ...
    ]
  }
}

errors:
- functionality does not exist
- product does not exist

------------------------------- abstract_attributes -------------------------------

request:
```

```
PUT /ams/v1/product/{productId}/abstractAttribute
{
  "displayName": "<display_name>",
  "tag": ["<tag1>", ...],
  "datatype": "<datatype>",
  "enum": "<enum>",
  "referencesAttribute": "<references_attribute>",
  "constraintExpression": "<constraint_expression>",
  "immutable": "<immutable>",
  "description": "<description>"
}

response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
GET /ams/v1/product/{productId}/abstractAttribute/{displayName}

response:
{
  "statusCode": 201/400,
  "data": {
    "abstractPath": "<abstract_path>",
    "componentType": "<component_type>",
    "componentId": "<component_id>",
    "displayName": "<display_name>",
    "tag": ["<tag1>", ...],
    "datatype": "<datatype>",
    "enum": "<enum>",
    "referencesAttribute": "<references_attribute>",
    "constraintExpression": "<constraint_expression>",
    "immutable": "<immutable>",
    "description": "<description>"
  }
}

------------------------------- attributes -------------------------------

request:
PUT /ams/v1/product/{productId}/attribute
{
  "displayName": "<display_name>",
  "value": <value>,
  "rule": {
    "type": "<rule_type>",
    "inputAttribute": [...],
    "outputAttribute": [...],
    "displayExpression": "<display_expression>",
    "description": "<description>"
  }
}

```

```
response:
{
  "statusCode": 201/400,
  "message": "...",
  "errors": [...]
}

-------------------------------

request:
GET /ams/v1/product/{productId}/attribute/{displayName}

response:
{
  "statusCode": 201/400,
  "data": {
    "displayName": "<display_name>",
    "abstractPath": "<abstract_path>",
    "path": "<path>",
    "value": <value>,
    "rule": {
      "type": "<rule_type>",
      "inputAttribute": [...],
      "outputAttribute": [...],
      "displayExpression": "<display_expression>",
      "description": "<description>"
    }
  }
}

------------------------------- attributes by functionality -------------------------------

request:
GET /ams/v1/product/{productId}/functionality/{functionality}/attribute

response:
{
  "statusCode": 201/400,
  "data": {
    "attribute": [
      {
        "displayName": "<display_name>",
        "path": "<path>",
        "value": <value>,
        "rule": {
          "type": "<rule_type>",
          "inputAttribute": [...],
          "outputAttribute": [...],
          "displayExpression": "<display_expression>",
          "description": "<description>"
        },
        "constraintExpression": "<constraint_expression>"
      },
      ...
    ]
  }
}
```

```
------------------------------- attributes by tags -------------------------------

request:
GET /ams/v1/product/{productId}/attributeByTag/{tag}

response:
{
  "statusCode": 201/400,
  "data": {
    "attribute": [
      {
        "displayName": "<display_name>",
        "datatype": "<datatype>",
        "enum": "<enum>",
        "referencesAttribute": "<references_attribute>",
        "constraintExpression": "<constraint_expression>",
        "immutable": "<immutable>",
        "description": "<description>"
      },
      ...
    ]
  }
}

```

Doubts:

- As the naming will evolve for an internal attribute, its possible that the user (product-team) may want to change the display_name (naming convention) of an attribute. Do we need to and how to handle backward compatibility in that case?
  Resolution: we'll not allow such changes, name and display-name convention will be driven by a fixed algorithm.

Insurance Product Configuration steps

- Create basic product
- Create cover groups, covers, cover options
- Create discounts, loadings, taxes
- Create plans
- Create asset attributes and product input
- Create rules for product input validation, premium calculation, get applicable plans, get premium, get cover options for plan, applicable covers, get discount, get loading, get cancellation

Excel Sheets vs UI

Creating premium slabs is possible on sheet but creating some other rules which have complex conditions (sheets can only support simple condition slabs like <, >, =, OR, AND, etc) like for example product input validation rules for example will not be feasible through the sheet.

We need to have a UI in order to manage the attributes. As attributes have interdependencies and rules can be complex, we need to have an interactive UI for configuration management.

Configurations through sheets:

Product Configuration Dashboards & Sheet Generation and Parsing

Execution:

Step 1: Making sheet parsing logic and hooking it up with the current product (using adapters so that we can hookup to AMS also later)

Step 2: Ops portal dashboards for creating and viewing product/plan, uploading and managing documents and templates as sheet templates wont support doc. attachments.

Step 3: Implement AMS and hookup with it while discussions on moving some things to ui dashboards so that in future new attributes can be added (attribute mapping management)

References:

Coincidentally I also found a US patent (dated 2008) on this, which shares some of the objectives we want to achieve here:
https://patentimages.storage.googleapis.com/8a/ae/a2/fbbd530e18bee6/US7401061.pdf