Apache DS SchemaManager

> ⚠ **Work in progress**
>
> This site is in the process of being reviewed and updated.

Introduction

Apache DS has to keep a lot of internal structures available from all the parts of the server. This is done through what we call the **SchemaManager**. It hides all the internal structure from the users.

SchemaManager

The **SchemaManager** stores **Registries**, which are hives where each **SchemaObjects** are stored. We also store some dedicated data structures :

- factory : The object responsible for the **SchemaObject** instance creation, given an Entry containing a **SchemaObject** description
- namingContext : The partition this* SchemaManager* is associated with. In the future, we want to associate a **SchemaManager** to a Partition, allowing the server to have more than one **SchemaManager**
- registries : The container for all the **SchemaObject** Registries
- schemaLoader : The loader for this instance. **SchemaObjects** may be stored on disk, in a database... The loader is responsible for their retrieval
- errors : The list of errors we got when we have updated the schema. It should be empty before we can use this SchemaManager in the server

Registries

This is the internal container for all the **SchemaObject** registries. When modifying the schema, this object will be cloned, modified, chekced, and if thee is no error, we will apply those modifications to the real registries.

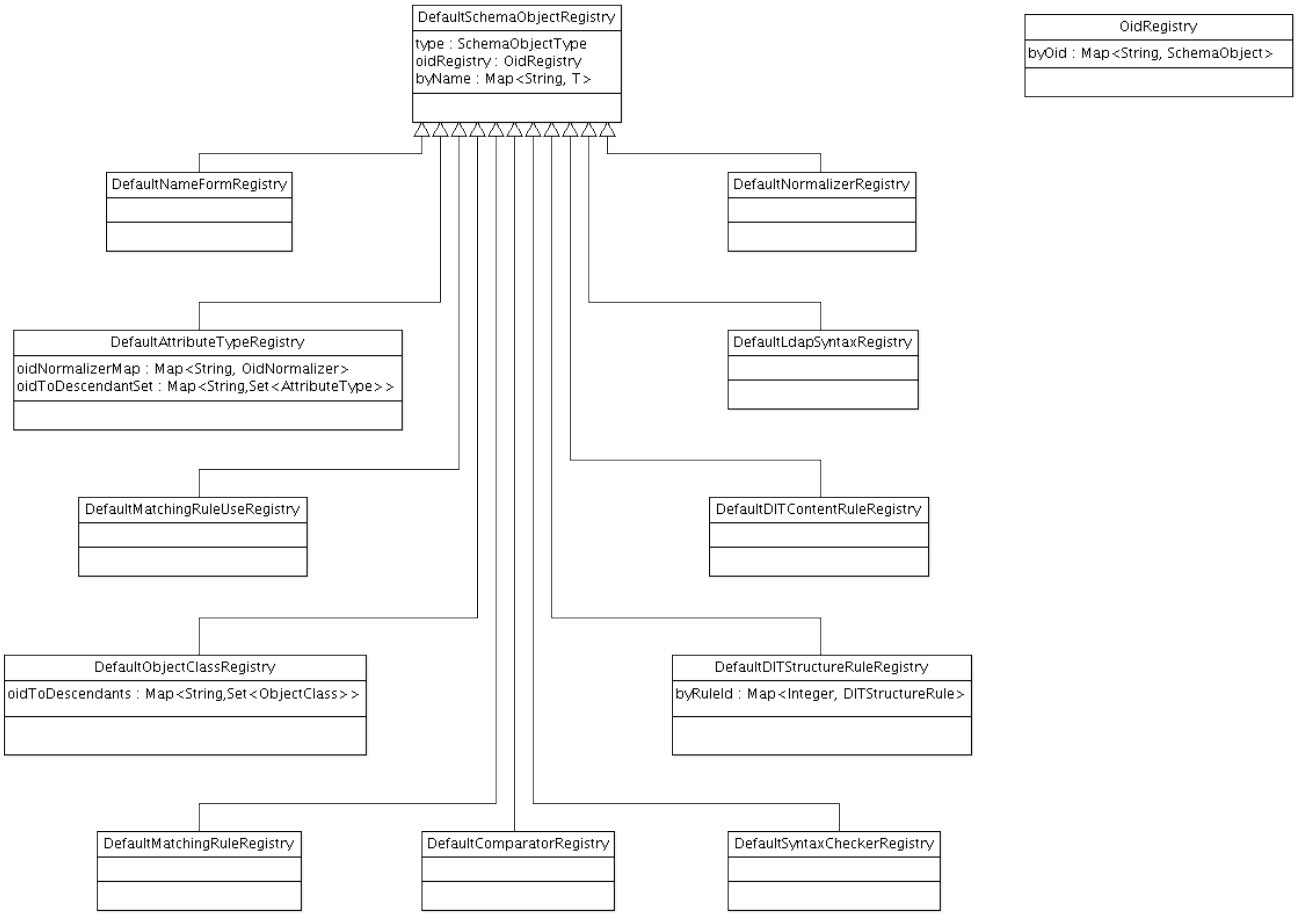
It contains a set of fields used to manage the schema :

- globalOidRegistry : It stores the list of all the **SchemaObject*s' OIDs. A *SchemaObject** has a unique OID in a **SchemaManager**
- <SchemaObject>Registry : A registry per **SchemaObject** type.
- loadedSchemas : The list of all the loaded schemas
- schemaObjectsBySchemaName : A list of all **SchemaObjects** per schema
- usedBy : a map containing the list of **SchemaObject** referencing a given **SchemaObject**
- using : a map containing the list of **SchemaObject** used by a given **SchemaObject**

SchemaObject Registries

For each different type of **SchemaObject**, we have a dedicated Registry.

The following diagram shows the class diagram for those registries :



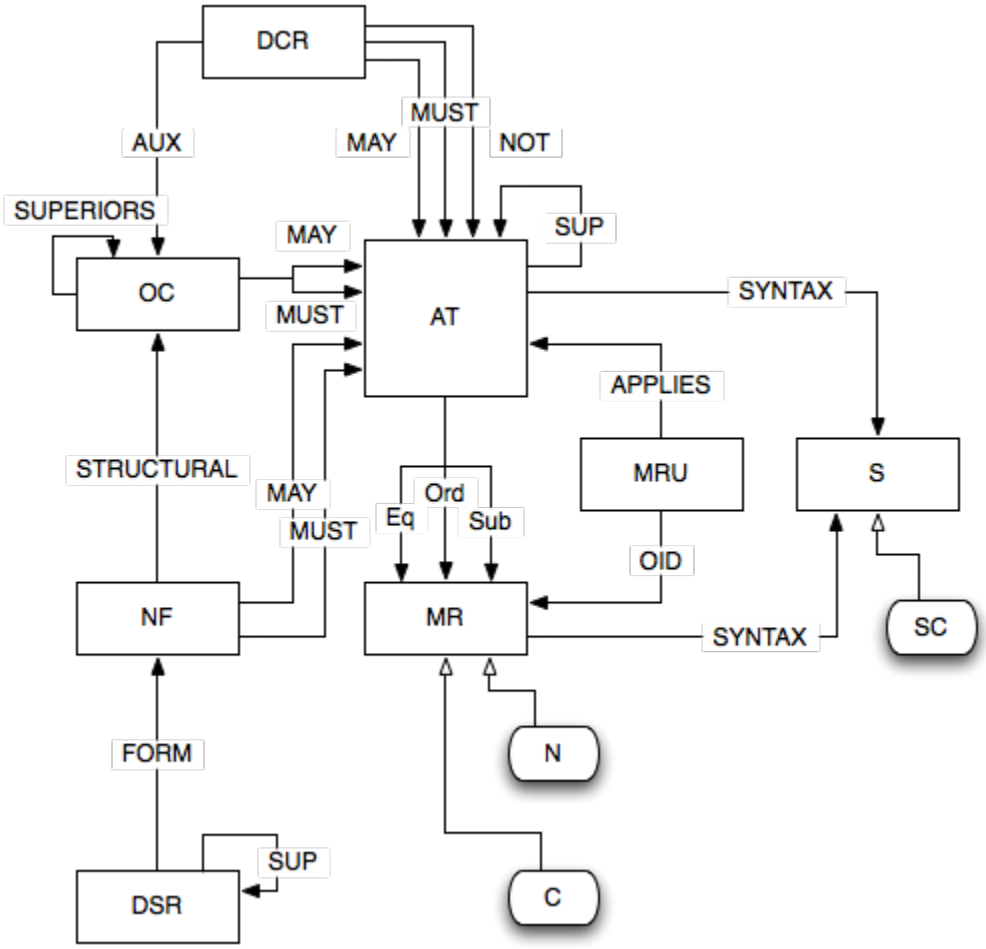
SchemaObjects

We have 11 different **SchemaObjects**, 3 of them are Apache DS specific :

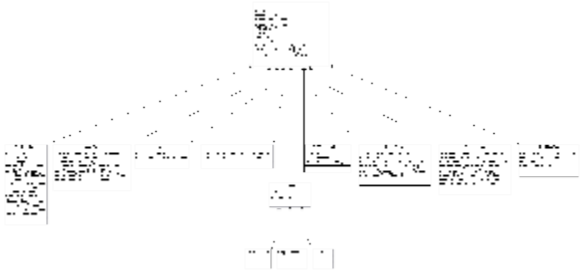
- (AT) AttributeType
- (C) Comparator (specific)
- (DCR) DITContentRule
- (DSR) DITStructureRule
- (MR) MatchingRule
- (MRU) MatchingRuleUse
- (NF) NameForm
- (N) Normalizer (specific)
- (OC) ObjectClass
- (S) Syntax
- (SC) SyntaxChecker (specific)

The specific SchemaObjects are those we can load into the server dynamically : they are compiled Java classes.

All the **SchemaObjects** are related with each other. The following schema shows all the existing relations :



This class diagram exposes the relations between all those classes :



All the Registries contain two data structure :

- byOid : stores all the associated SchemaObject by their OID
- byName : stores all the associated SchemaObject by their name (slao contains the oid)

Note that the ObjectClassRegistry and the AttributeTypeRegistry contain a specific data structure to keep a track of the descendants for each AT or OC.

AttributeType

Here are the fields stored in a AttributeType instance :

Name	default
description	N/A
extensions	N/A

isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	ATTRIBUTE_TYPE
oid	AT OID
schemaName	N/A
specification	not modifiable
canUserModify	TRUE
equality	MR reference
equalityOid	MR OID
isCollective	FALSE
isSingleValued	FALSE
ordering	MR reference
orderingOid	MR OID
substring	MR reference
substringOid	MR OID
superior	AT reference
superiorOid	MR OID
syntax	Syntax reference
syntaxLength	0
syntaxOid	Syntax OID
usage	userApplications

Adding an AttributeType

Register the AttributeType into the AttributeTypeRegistry :
update the AttributeTypeRegistry.byOid : add <oid, AttributeType>
for all the attributeType names, plus the oid,
update the AttributeTypeRegistry.byName : add <name, AttributeType>
update the AttributeTypeRegistry.byName : add <oid, AttributeType>
associate the AttributeType with the schema : <schema, set<SchemaObject>> += AttributeType
update the Registries.globalOidRegistry : add <oid, AttributeType>

We also have a special data structure to update :

When all the schema modifications will be done, do the additional updates :

update the AttributeType.syntax : SyntaxRegistry.lookup(syntaxOid)
update the AttributeType.equality : MatchingRuleRegistry.lookup(equalityOid)
update the AttributeType.ordering : MatchingRuleRegistry.lookup(orderingOid)
update the AttributeType.substring : MatchingRuleRegistry.lookup(substringOid)
update the AttributeType.sup : AttributeTypeRegistry.lookup(superiorOid)
update the Registries.using map : <AttributeType.oid, set<SchemaObject>> +=
syntax, equality, ordering, substring, superior
update the Registries.usedBy map :
<Syntax.oid, set<SchemaObject>> += AttributeType
<equalityOid, set<SchemaObject>> += AttributeType
<orderingOid, set<SchemaObject>> += AttributeType
<substringOid, set<SchemaObject>> += AttributeType
<superiorOid, set<SchemaObject>> += AttributeType

last, not least, update the AttributeTypeRegistry oidNormalizerMap and oidToDescendant data structures :
AttributeTypeRegistry.oidNormalizerMap += new OidNormalizer(oid, NormalizerRegistry.lookup(equality.getNormalizer())
AttributeTypeRegistry.oidToDescendant : <superiorOid, Set<AttributeType>> += attributeType

There are special cases to deal with :

- if the Syntax is null, then inherit it from the superior, whcih must not be null
- if the equality is null, and if we have a superior, inherit the equality from it
- if there is a superior, the Usage must be the same than its superior's

- if the superior is COLLECTIVE, the it must also be COLLECTIVE
- if it's COLLECTIVE, then the Usage must be operational

NOTE : The OidNormalizerMap is probably useless

Modifying an AttributeType

We won't create a new object, but will update the existing one. The OID can't be changed

If we have changed one of the MRs, or the S, or the superior AT, when all the schema
modifications will be done, do the additional updates :
update the AttributeType.<MR/S/sup> : <MR/S/AT>Registry.lookup(oid)
update the Registries.using map : <AttributeType.oid, set<SchemaObject>> -= old <MR/S/AT>
update the Registries.using map : <AttributeType.oid, set<SchemzObject>> += new <MR/S/AT>
update the Registries.usedBy map : <old <MR/S/AT>.oid, set<SchemaObject>> -= AttributeType
update the Registries.usedBy map : <new <MR/S/AT>.oid, set<SchemaObject>> += AttributeType
update the oidNormalizerMap : remove the Oidnormalizer, add the new one
update the oidToDescendant Map : remove the AT from the old superior, add it to the new superior

We will also have to check that the modified AT is still valid

Deleting an AttributeType

We can't delete an AttributeType if it's refered by an ObjectClass, or another AttributeType.
This is checked by verifying inthe usedBy table :

if usedBy.get(attributeType.oid) is not empty, generate an error.

remove the oid from the AttributeTypeRegistry.byOid
for each AttributeType's name,
remove the AttributeType.name from the AttributeTypeRegistry.byName
+ remove the AttributeType.oid from the AttributeTypeRegistry.byName
remove all the names and the oid from the AttributeTypeRegistry.byName
remove the AttributeType from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= AttributeType
remove the AttributeType.oid from the globalOidRegistry : remove <oid, AttributeType>
remove the AttributeType's OidNormalizer from the oidNormalizerMap

When all the schema modifications will be done, do the additional updates :

update the Registries.using map : remove the relation <AttributeType.oid, set<SchemaObject>>
update the Registries.usedBy map for each AT/MR/S referenced in the AttributeType :
<<AT/MR/S>.oid, set<SchemaObject>> -= AttributeType
remove the AttributeType from the oidToDescendant Map

Comparator

Here are the fields stored in a Comparator instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	COMPARATOR
oid	MR OID
schemaName	N/A
specification	not modifiable
bytecode	not modifiable
fqcn	not modifiable

Adding a comparator

Register the comparator into the ComparatorRegistry :
update the ComparatorRegistry.byOid : add <oid, comparator>
update the ComparatorRegistry.byName : add <oid, comparator>
associate the Comparator with the schema : <schema, set<SchemaObject>> += comparator

Modifying a comparator

Nothing to do but update the comparator in place (replacing all the fields of the original comparator).

Note that all the fields are not modifiables.

Deleting a comparator

We can't delete a comparator if it is used by a MatchingRule. This is checked by verifying in
the usedBy table :

if usedBy.get(comparator.oid) is not empty, generate an error.

This is a more complex operation, as we may have some MatchingRules pointing on this object.
We have to :
remove the oid from the ComparatorRegistry.byOid
remove the oid from the ComparatorRegistry.byName
remove the Comparator from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= comparator

DITContentRule

Not Yet Implemented

DITStructureRule

Not Yet Implemented

MatchingRule

Here are the fields stored in a MatchingRule instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	ATTRIBUTE_TYPE
oid	AT OID
schemaName	N/A
ldapComparator	Comparator reference
ldapSyntax	Syntax reference
ldapSyntaxOid	The Syntax OID
normalizer	Normalizer reference

Adding a MatchingRule

Register the MatchingRule into the MatchingRuleRegistry :
update the MatchingRuleRegistry.byOid : add <oid, MatchingRule>
update the MatchingRuleRegistry.byName : add <oid, MatchingRule>
associate the MatchingRule with the schema : <schema, set<SchemaObject>> += MatchingRule
update the Registries.globalOidRegistry : add <oid, MatchingRule>

When all the schema modifications will be done, do the additional updates :

update the MatchingRule.syntax : SyntaxRegistry.lookup(syntaxoid)
update the MatchingRule.normalizer : NormalizerRegistry.lookup(matchingRule.oid)
update the MatchingRule.comparator : ComparatorRegistry.lookup(matchingRule.oid)
update the Registries.using map : <MatchingRule.oid, set<SchemaObject>> += syntax, normalizer, comparator
update the Registries.usedBy map :
<Syntax.oid, set<SchemaObject>> += MatchingRule
<Comparator.oid, set<SchemaObject>> += MatchingRule
<Normalizer.oid, set<SchemaObject>> += MatchingRule

Modifying a MatchingRule

We won't create a new object, but will update the existing one. The OID can't be changed

If we have changed the (N)ormalizer, the (S)yntax or the (C)omparator, when all the schema
modifications will be done, do the additional updates :
update the MatchingRule.<C/N/S> : <C/S/N>Registry.lookup(oid)
update the Registries.using map : <MatchingRule.oid, set<SchemaObject>> -= old <C/S/N>
update the Registries.using map : <MatchingRule.oid, set<SchemzObject>> += new <C/S/N>
update the Registries.usedBy map : <old <C/S/N>.oid, set<SchemaObject>> -= MatchingRule
update the Registries.usedBy map : <new <C/S/N>.oid, set<SchemaObject>> += MatchingRule

Deleting a MatchingRule

We can't delete a MatchingRule which is used by an AttributeType. This is checked by verifying in
the usedBy table :

if usedBy.get(MatchingRule.oid) is not empty, generate an error.

Otherwise, here are the operations we have to conduct :
remove the MatchingRule.oid from the MatchingRuleRegistry.byOid
for each MatchingRule's name,
remove the MatchingRule.name from the MatchingRuleRegistry.byName
+ remove the MatchingRule.oid from the MatchingRuleRegistry.byName
remove the MatchingRule from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= MatchingRule
remove the Registries.globalOidRegistry : remove <oid, MatchingRule>

When all the schema modifications will be done, do the additional updates :

update the Registries.using map : remove the relation <MatchingRule.oid, set<SchemaObject>>
update the Registries.usedBy map for each C/S/N referenced in the MatchingRule :
<<C/S/N>.oid, set<SchemaObject>> -= MatchingRule

MatchingRuleUse

Not Yet Implemented

NameForm

Not Yet Implemented

Normalizer (specific)

Here are the fields stored in a Normalizer instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	NORMALIZER
oid	MR OID
schemaName	N/A

specification	not modifiable
bytecode	not modifiable
fqcn	not modifiable

Adding a Normalizer

Register the Normalizer into the NormalizerRegistry :
update the NormalizerRegistry.byOid : add <oid, Normalizer>
update the NormalizerRegistry.byName : add <oid, Normalizer>
associate the Normalizer with the schema : <schema, set<SchemaObject>> += Normalizer

Modifying a Normalizer

Nothing to do but update the Normalizer in place (replacing all the fields of the original Normalizer).

Note that all the fields are not modifiables.

Deleting a Normalizer

We can't delete a Normalizer if it is used by a MatchingRule. This is checked by verifying in
the usedBy table :

if usedBy.get(Normalizer.oid) is not empty, generate an error.

This is a more complex operation, as we may have some MatchingRules pointing on this object.
We have to :
remove the oid from the NormalizerRegistry.byOid
remove the oid from the NormalizerRegistry.byName
remove the Normalizer from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= Normalizer

ObjectClass

Here are the fields stored in a ObjectClass instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	NORMALIZER
oid	MR OID
schemaName	N/A
specification	not modifiable
mayAttributeTypes	all the referenced MAY attributeType
mayAttributeTypeOids	all the referenced MAY attributeType's oid
mustAttributeTypes	all the referenced MUST attributeType
mustAttributeTypeOids	all the referenced MUST attributeType's oid
objectClassType	STRUCTURAL
superiors	all the inherited SUPERIOR
superiorOids	all the inherited SUPERIOR's oid

Adding an ObjectClass

Register the ObjectClass into the ObjectClassRegistry :
update the ObjectClassRegistry.byOid : add <oid, ObjectClass>
for all the ObjectClass names, plus the oid,
update the ObjectClassRegistry.byName : add <name, ObjectClass>
update the ObjectClassRegistry.byName : add <oid, ObjectClass>
associate the ObjectClassType with the schema : <schema, set<SchemaObject>> += ObjectClass
update the Registries.globalOidRegistry : add <oid, ObjectClass>

When all the schema modifications will be done, do the additional updates :

update all the ObjectClass.may : AttributeTypeRegistry.lookup(mayOid)
update all the ObjectClassType.must : AttributeTypeRegistry.lookup(mustOid)
update all the ObjectClass.superiors : ObjectClassRegistry.lookup(superiorOid)
update the Registries.using map : <ObjectClass.oid, set<SchemaObject>> +=
all the may, all the must, all the superiors
update the Registries.usedBy map for each may, must and superior :
<may.oid, set<SchemaObject>> += ObjectClass
<must.Oid, set<SchemaObject>> += ObjectClassuteType
<superior.oid, set<SchemaObject>> += ObjectClass

If the added ObjectClass inherits from one ore more ObjectClass, update the oidToDescendant map :
for each superiorOid :
<superiorOid, Set<ObjectClass>> += ObjectClass

There are special cases to deal with :

- If the ObjectClass is ABSTRACT, then it can inherit from a STRUCTURAL or AUXILIARY ObjectClass
- If the ObjectClass is AUXILIARY, then it can't inherit from a STRUCTURAL ObjectClass
- If the ObjectClass is STRUCTURAL, then it can't inherit from a AUXILIARY ObjectClass

Modifying an ObjectClass

We won't create a new object, but will update the existing one. The OID can't be changed

We will also have to check that the modified AT is still valid

Deleting an ObjectClass

We can't delete an Objectlass if it's refered by another ObjectClass.
This is checked by verifying inthe usedBy table :

if usedBy.get(objectClass.oid) is not empty, generate an error.

remove the oid from the ObjectClassRegistry.byOid
for each ObjectClass's name,
remove the ObjectClass.name from the ObjectClassRegistry.byName
+ remove the ObjectClass.oid from the ObjectClassRegistry.byName
remove the ObjectClass from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= ObjectClass
remove the ObjectClass.oid from the globalOidRegistry : remove <oid, ObjectClass>

When all the schema modifications will be done, do the additional updates :

update the Registries.using map : remove the relation <ObjectClass.oid, set<SchemaObject>>

Syntax

Here are the fields stored in a Syntax instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	ATTRIBUTE_TYPE
oid	AT OID
schemaName	N/A

isHumanReadable	FALSE
syntaxChecker	SC reference

Adding a Syntax

Register the Syntax into the SyntaxRegistry :
update the SyntaxRegistry.byOid : add <oid, Syntax>
update the SyntaxRegistry.byName : add <oid, Syntax>
associate the Syntax with the schema : <schema, set<SchemaObject>> += Syntax
update the Registries.globalOidRegistry : add <oid, Syntax>

When all the schema modifications will be done, do the additional updates :

update the Syntax.syntaxChecker : SyntaxCheckerRegistry.lookup(oid)
update the Registries.using map : <Syntax.oid, set<SchemaObject>> += SyntaxChecker
update the Registries.usedBy map : <SyntaxChecker.oid, set<SchemaObject>> += Syntax

Modifying a Syntax

We won't create a new object, but will update the existing one. The OID can't be changed

If we have changed the SyntaxChecker, when all the schema modifications will be done, do the additional updates :
update the Syntax.syntaxChecker : SyntaxCheckerRegistry.lookup(oid)
update the Registries.using map : <Syntax.oid, set<SchemaObject>> -= old SyntaxChecker
update the Registries.using map : <Syntax.oid, set<SchemaObject>> += new SyntaxChecker
update the Registries.usedBy map : <old SyntaxChecker.oid, set<SchemaObject>> -= Syntax
update the Registries.usedBy map : <new SyntaxChecker.oid, set<SchemaObject>> += Syntax

Deleting a Syntax

We can't delete a Syntax which is used by either a MatchingRule or an AttributeType. This is checked by verifying in the usedBy table :

if usedBy.get(Syntax.oid) is not empty, generate an error.

Otherwise, here are the operations we have to conduct :
remove the Syntax.oid from the SyntaxRegistry.byOid
for each Syntax' name,
remove the Syntax.name from the SyntaxRegistry.byName
+ remove the Syntax.oid from the SyntaxRegistry.byName
remove the Syntax from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= Syntax
remove the Registries.globalOidRegistry : remove <oid, Syntax>

When all the schema modifications will be done, do the additional updates :

update the Registries.using map : remove the relation <Syntax.oid, set<SchemaObject>>
update the Registries.usedBy map : <SyntaxChecker.oid, set<SchemaObject>> -= Syntax

SyntaxChecker (specific)

Here are the fields stored in a SyntaxChecker instance :

Name	default
description	N/A
extensions	N/A
isEnabled	TRUE
isObsolete	FALSE
isReadOnly	FALSE
names	No names
objectType	SYNTAX_CHECKER
oid	SYNTAX OID
schemaName	N/A
specification	not modifiable

bytecode	not modifiable
fqcn	not modifiable

Adding a SyntaxChecker

Register the SyntaxChecker into the SyntaxCheckerRegistry :
update the SyntaxCheckerRegistry.byOid : add <oid, SyntaxChecker>
update the SyntaxCheckerRegistry.byName : add <oid, SyntaxChecker>
associate the SyntaxChecker with the schema : <schema, set<SchemaObject>> += SyntaxChecker

Modifying a SyntaxChecker

Nothing to do but update the SyntaxChecker in place (replacing all the fields of the original SyntaxChecker).

Note that all the fields are not modifiables.

Deleting a SyntaxChecker

We can't delete a SyntaxChecker if it is used by a Syntax. This is checked by verifying in
the usedBy table :

if usedBy.get(SyntaxChecker.oid) is not empty, generate an error.

This is a more complex operation, as we may have some Syntax pointing on this object.
We have to :
remove the oid from the SyntaxCheckerRegistry.byOid
remove the oid from the SyntaxCheckerRegistry.byName
remove the SyntaxChecker from the bySchemaNameSchemaObject map : <schema, set<SchemaObject>> -= SyntaxChecker

SyntaxCheckers list

We can see that we may have many syntax checkers. The list are given in RFC 2252, RFC 4517 and RFC 4523:

RFC 2252 /22566	RFC 4517 /4523	Syntax Checker	OID	H /R
X	X	Attribute Type Description	1.3.6.1.4.1.1466.115.121.1.3	Y
X	(removed)	Binary	1.3.6.1.4.1.1466.115.121.1.5	N
X	X	Bit String	1.3.6.1.4.1.1466.115.121.1.6	Y
X	X	Boolean	1.3.6.1.4.1.1466.115.121.1.7	Y
X	(RFC 4523)	Certificate	1.3.6.1.4.1.1466.115.121.1.8	N
X	(RFC 4523)	Certificate List	1.3.6.1.4.1.1466.115.121.1.9	N
X	(RFC 4523)	Certificate Pair	1.3.6.1.4.1.1466.115.121.1.10	N
X	X	Country String	1.3.6.1.4.1.1466.115.121.1.11	Y
X	X	Delivery Method	1.3.6.1.4.1.1466.115.121.1.14	Y
X	X	Directory String	1.3.6.1.4.1.1466.115.121.1.15	Y
X	X	DIT Content Rule Description	1.3.6.1.4.1.1466.115.121.1.16	Y
X	X	DIT Structure Rule Description	1.3.6.1.4.1.1466.115.121.1.17	Y
X	X	DN	1.3.6.1.4.1.1466.115.121.1.12	Y
X	X	Enhanced Guide	1.3.6.1.4.1.1466.115.121.1.21	Y
X	X	Facsimile Telephone Number	1.3.6.1.4.1.1466.115.121.1.22	Y
X	X	Fax	1.3.6.1.4.1.1466.115.121.1.23	N
X	X	Generalized Time	1.3.6.1.4.1.1466.115.121.1.24	Y
X	X	Guide	1.3.6.1.4.1.1466.115.121.1.25	Y
X	X	IA5 String	1.3.6.1.4.1.1466.115.121.1.26	Y

X	X	Integer	1.3.6.1.4.1.1466.115.121.1.27	Y
X	X	JPEG	1.3.6.1.4.1.1466.115.121.1.28	N
X	X	LDAP Syntax Description	1.3.6.1.4.1.1466.115.121.1.54	Y
X	X	Matching Rule Description	1.3.6.1.4.1.1466.115.121.1.30	Y
X	X	Matching Rule Use Description	1.3.6.1.4.1.1466.115.121.1.31	Y
X	(removed)	MHS OR Address	1.3.6.1.4.1.1466.115.121.1.33	Y
X	X	Name and Optional UID	1.3.6.1.4.1.1466.115.121.1.34	Y
X	X	Name Form Description	1.3.6.1.4.1.1466.115.121.1.35	Y
X	X	Numeric String	1.3.6.1.4.1.1466.115.121.1.36	Y
X	X	Object Class Description	1.3.6.1.4.1.1466.115.121.1.37	Y
X	X	Octet String	1.3.6.1.4.1.1466.115.121.1.40	Y
X	X	OID	1.3.6.1.4.1.1466.115.121.1.38	Y
X	X	Other Mailbox	1.3.6.1.4.1.1466.115.121.1.39	Y
X	X	Postal Address	1.3.6.1.4.1.1466.115.121.1.41	Y
X	(removed)	Presentation Address	1.3.6.1.4.1.1466.115.121.1.43	Y
X	X	Printable String	1.3.6.1.4.1.1466.115.121.1.44	Y
-	X	Substring Assertion	1.3.6.1.4.1.1466.115.121.1.58	Y
X	(RFC 4523)	Supported Algorithm	1.3.6.1.4.1.1466.115.121.1.49	N
X	X	Telephone Number	1.3.6.1.4.1.1466.115.121.1.50	Y
X	X	Teletex Terminal Identifier	1.3.6.1.4.1.1466.115.121.1.51	Y
X	X	Telex Number	1.3.6.1.4.1.1466.115.121.1.52	Y
X	X	UTC Time	1.3.6.1.4.1.1466.115.121.1.53	Y

As we can see, each syntax should have a specific class associated with a specific **check** method used to control that the attribute value is correct. This value will be checked for every entry adsded or modified by a user. Each **DN** submitted will also be checked against those classes.

To be able to extend the server with new syntaxes, thoses checker classes will be dynamically loaded at startup. We can do that in different ways :

- defining those classes into ADS code, and load them statically during the compilation of the server : this will not be dynamic, but anyway, we will follow the specification and the server will be LDAP V3 compliant
- compiling the syntaxChecker classes and injecting the **.class** into the **cn=schema,ou=system** partition : This is possible by either injecting a LDIF file where the .class is serialized, or using a description containing the SyntaxChecker

TO BE CONTIINUED

.