ServerAttribute code

Description

This page expose the ServerAttribute and ServerAttributeImpl code.

ServerAttribute Interface

```
<apache license here>
import java.io.Serializable;
import java.util.Iterator;

import javax.naming.NamingException;

import org.apache.directory.shared.asn1.primitives.OID;
import org.apache.directory.shared.ldap.schema.Normalizer;

/**
 * This interface defines the valid operations on a particular attribute of a
 * directory entry.
 * <p>
 * An attribute can have zero or more values. The value may be null. Values are
 * not ordered
 * </p>
 * <p>
 * The indexed operations work as if the values
 * added previously to the attribute had been done using ordered semantics. For
 * example, if the values "a", "b" and "c" were previously added to an unordered
 * attribute using "<code>add("a"); add("b"); add("c");</code>", it is
 * equivalent to adding the same objects to an ordered attribute using "<code>add(0,"a"); add(1,"b"); add(2,"
c");</code>".
 * In this case, if we do "<code>remove(1)</code>" on the unordered list,
 * the value "b" is removed, changing the index of "c" to 1.
 * </p>
 * <p>
 * Multiple null values can be added to an attribute. It is not the same as
 * having no values on an attribute. If a null value is added to an unordered
 * attribute which already has a null value, the <code>add</code> method has
 * no effect.
 * </p>
 * <p>
 * Note that updates to the attribute via this interface do not affect the
 * directory directly.
 *
 * </p>
 * This interface represents an attribute used internally by the
 * server. It's a subset of the javax.naming.directory.Attribute, where
 * some methods have been removed, and which manipulates Value instead
 * of Object
 *
 * @author <a href="mailto:dev@directory.apache.org"> Apache Directory Project</a>
 * @version $Rev: 499013 $
 */
public interface ServerAttribute extends Cloneable, Serializable
{
    /**
     * Adds a value to this attribute. If the new value is already present in
     * the attribute values, the method has no effect.
     * <p>
     * The new value is added at the end of list of values.
     * </p>
     * <p>
     * This method returns true or false to indicate whether a value was added.
     * </p>
     *
     * @param val a new value to be added which may be null
     * @return true if a value was added, otherwise false
```

```
 */
boolean add( Value<?> val );


/**
 * Adds a value to this attribute. If the new value is already present in
 * the attribute values, the method has no effect.
 * <p>
 * The new value is added at the end of list of values.
 * </p>
 * <p>
 * This method returns true or false to indicate whether a value was added.
 * </p>
 *
 * @param val a new value to be added which may be null
 * @return true if a value was added, otherwise false
 */
boolean add( String val );


/**
 * Adds a value to this attribute. If the new value is already present in
 * the attribute values, the method has no effect.
 * <p>
 * The new value is added at the end of list of values.
 * </p>
 * <p>
 * This method returns true or false to indicate whether a value was added.
 * </p>
 *
 * @param val a new value to be added which may be null
 * @return true if a value was added, otherwise false
 */
boolean add( byte[] val );


/**
 * Removes all values of this attribute.
 */
void clear();


/**
 * Returns a deep copy of the attribute containing all the same values. The
 * values <b>are</b> cloned.
 *
 * @return a deep clone of this attribute
 */
ServerAttribute clone();


/**
 * Indicates whether the specified value is one of the attribute's values.
 *
 * @param val the value which may be null
 * @return true if this attribute contains the value, otherwise false
 */
boolean contains( Value<?> val );


/**
 * Indicates whether the specified value is one of the attribute's values.
 *
 * @param val the value which may be null
 * @return true if this attribute contains the value, otherwise false
 */
boolean contains( String val );


/**
 * Indicates whether the specified value is one of the attribute's values.
```

```
     *
     * @param val the value which may be null
     * @return true if this attribute contains the value, otherwise false
     */
    boolean contains( byte[] val );


    /**
     * Gets a value of this attribute. Returns the first one, if many.
     * <code>null</code> is a valid value.
     * <p>
     *
     * If the attribute has no values this method throws
     * <code>NoSuchElementException</code>.
     * </p>
     *
     * @return a value of this attribute
     * @throws NamingException If the attribute has no value.
     */
    Value<?> get() throws NamingException;


    /**
     * Returns an enumeration of all the attribute's values.
     * <p>
     * The effect on the returned enumeration of adding or removing values of
     * the attribute is not specified.
     * </p>
     * <p>
     * This method will throw any <code>NamingException</code> that occurs.
     * </p>
     *
     * @return an enumeration of all values of the attribute
     * @throws NamingException If any <code>NamingException</code> occurs.
     */
    Iterator<Value<?>> getAll() throws NamingException;


    /**
     * Returns the identity of this attribute. This method is not expected to
     * return null.
     *
     * @return The id of this attribute
     */
    String getID();


    /**
     * Returns the OID of this attribute. This method is not expected to
     * return null.
     *
     * @return The OID of this attribute
     */
    OID getOid();


   /**
      * Retrieves the number of values in this attribute.
      *
      * @return The number of values in this attribute, including the null value
      * if there is one.
      */
    int size();


    /**
     * Removes a value that is equal to the given value.
     * <p>
     * Returns true if a value is removed. If there is no value equal to <code>
     * val</code> this method simply returns false.
     * </p>
```

```
     *
     * @param val the value to be removed
     * @return true if the value is removed, otherwise false
     */
    boolean remove( Value<?> val );


    /**
     * Removes a value that is equal to the given value.
     * <p>
     * Returns true if a value is removed. If there is no value equal to <code>
     * val</code> this method simply returns false.
     * </p>
     *
     * @param val the value to be removed
     * @return true if the value is removed, otherwise false
     */
    boolean remove( byte[] val );


    /**
     * Removes a value that is equal to the given value.
     * <p>
     * Returns true if a value is removed. If there is no value equal to <code>
     * val</code> this method simply returns false.
     * </p>
     *
     * @param val the value to be removed
     * @return true if the value is removed, otherwise false
     */
    boolean remove( String val );


    /**
     * Normalize the attribute, setting the OID and normalizing the values
     *
     * @param oid The attribute OID
     * @param normalizer The normalizer
     * @throws NamingException when normalization fails
     */
    void normalize( OID oid, Normalizer normalizer ) throws NamingException;
}
```

ServerAttributeImpl Implemented class

```
<apache license here>
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.BasicAttribute;

import org.apache.directory.shared.asn1.primitives.OID;
import org.apache.directory.shared.ldap.schema.Normalizer;
import org.apache.directory.shared.ldap.util.AttributeUtils;
import org.apache.directory.shared.ldap.util.StringTools;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;


/**
 * Attribute implementation.
 *
```

```
 * @author <a href="mailto:dev@directory.apache.org"> Apache Directory Project</a>
 * @version $Rev: 499013 $
 */
public class ServerAttributeImpl implements ServerAttribute, Serializable, Cloneable
{
    private static final Logger LOG = LoggerFactory.getLogger( ServerAttributeImpl.class );

    /** For serialization */
    private static final long serialVersionUID = 2L;

    /** the name of the attribute, case sensitive */
    private final String upId;

    /** the OID of the attribute */
    private OID oid;

    /** In case we have only one value, just use this container */
    private Value<?> value;

    /** the list of attribute values, if unordered */
    private List<Value<?>> values;

    /** A size to handle the number of values, as one of them can be null */
    private int size;


    // ------------------------------------------------------------------------
    // Constructors
    // ------------------------------------------------------------------------


    /**
     * Creates a ServerAttribute with an id
     *
     * @param id the id or name of this attribute.
     * @throws NamingException if the id is null
     */
    public ServerAttributeImpl( String id ) throws NamingException
    {
        if ( StringTools.isEmpty( id ) )
        {
            LOG.error( "Attributes with an empty ID or OID are not allowed" );
            throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
        }

        upId = id;
        value = null;
        values = null;
        oid = null;
        size = 0;
    }


    /**
     * Creates a ServerAttribute with an oid
     *
     * @param oid the oid of this attribute.
     * @throws NamingException if the oid is null
     */
    public ServerAttributeImpl( OID oid ) throws NamingException
    {
        if ( oid == null )
        {
            LOG.error( "Attributes with an empty ID or OID are not allowed" );
            throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
        }

        upId = oid.toString();
        value = null;
        values = null;
        this.oid = oid;
```

```
    size = 0;
}


/**
 * Creates a ServerAttribute with an id and a value
 *
 * @param id the id or name of this attribute.
 * @param val the attribute's value
 * @throws NamingException if the id is invalid
 */
public ServerAttributeImpl( String id, Value<?> val ) throws NamingException
{
    if ( StringTools.isEmpty( id ) )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = id;
    value = val;
    values = null;
    oid = null;
    size = 1;
}


/**
 * Creates a ServerAttribute with an oid and a value
 *
 * @param oid the oid of this attribute.
 * @param val the attribute's value
 * @throws NamingException if the oid is invalid
 */
public ServerAttributeImpl( OID oid, Value<?> val ) throws NamingException
{
    if ( oid == null )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = oid.toString();
    value = val;
    values = null;
    this.oid = oid;
    size = 1;
}


/**
 * Creates a ServerAttribute with an id and a byte[] value
 *
 * @param id the id or name of this attribute.
 * @param val a value for the attribute
 * @throws NamingException if the id string is invalid
 */
public ServerAttributeImpl( String id, byte[] val ) throws NamingException
{
    if ( StringTools.isEmpty( id ) )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = id;
    values = null;
    value = new BinaryValue( val );
    oid = null;
    size = 1;
}
```

```
/**
 * Creates a ServerAttribute with an oid and a byte[] value
 *
 * @param oid the oid of this attribute.
 * @param val a value for the attribute
 * @throws NamingException if the oid is invalid
 */
public ServerAttributeImpl( OID oid, byte[] val ) throws NamingException
{
    if ( oid == null )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = oid.toString();
    values = null;
    value = new BinaryValue( val );
    this.oid = oid;
    size = 1;
}


/**
 * Creates a ServerAttribute with an id and a String value
 *
 * @param id the id or name of this attribute.
 * @param val a value for the attribute
 * @throws NamingException if the id string is invalid
 */
public ServerAttributeImpl( String id, String val ) throws NamingException
{
    if ( StringTools.isEmpty( id ) )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = id;
    values = null;
    value = new StringValue( val );
    oid = null;
    size = 1;
}


/**
 * Creates a ServerAttribute with an oid and a String value
 *
 * @param oid the oid of this attribute.
 * @param val a value for the attribute
 * @throws NamingException if there's no OID provided
 */
public ServerAttributeImpl( OID oid, String val ) throws NamingException
{
    if ( oid == null )
    {
        LOG.error( "Attributes with an empty ID or OID are not allowed" );
        throw new NamingException( "Attributes with an empty ID or OID are not allowed" );
    }

    upId = oid.toString();
    values = null;
    value = new StringValue( val );
    this.oid = oid;
    size = 1;
}
```

```
/**
 * Create a copy of an Attribute, be it an ServerAttributeImpl
 * instance of a BasicAttribute instance
 *
 * @param attribute the Attribute instace to copy
 * @throws NamingException if the attribute is not an instance of BasicAttribute
 * or ServerAttributeImpl or is null
 */
public ServerAttributeImpl( Attribute attribute ) throws NamingException
{
    if ( attribute == null )
    {
        LOG.error(  "Null attribute is not allowed" );
        throw new NamingException( "Null attribute is not allowed" );
    }
    else if ( attribute instanceof ServerAttributeImpl )
    {
        ServerAttributeImpl copy = (ServerAttributeImpl)attribute;

        upId  = copy.upId;
        oid = copy.oid;

        switch ( copy.size() )
        {
            case 0:
                values = null;
                value = null;
                size = 0;
                break;

            case 1:
                value = getClonedValue( copy.get() );
                values = null;
                size = 1;
                break;

            default :
                value = null;
                values = new ArrayList<Value<?>>( copy.size() );

                Iterator<Value<?>> vals = copy.getAll();

                while ( vals.hasNext() )
                {
                    Value<?> val = vals.next();
                    values.add( val );
                }

                size = copy.size();

                break;
        }

        oid = null;
    }
    else if ( attribute instanceof BasicAttribute )
    {
        upId = attribute.getID();
        oid = null;

        switch ( attribute.size() )
        {
            case 0 :
                value = null;
                values = null;
                size = 0;
                break;

            case 1 :
                Object val = attribute.get();
```

```
                if ( val instanceof String )
                {
                    value = new StringValue( (String)val );
                }
                else if ( val instanceof byte[] )
                {
                    value = new BinaryValue( (byte[])val );
                }
                else
                {
                    LOG.error( "The value's type is not String or byte[]" );
                    throw new NamingException( "The value's type is not String or byte[]" );
                }

                values = null;
                size = 1;

                break;

            default :
                NamingEnumeration<?> vals = attribute.getAll();

                while ( vals.hasMoreElements() )
                {
                    val = vals.nextElement();

                    if ( val instanceof String )
                    {
                        value = new StringValue( (String)val );
                    }
                    else if ( val instanceof byte[] )
                    {
                        value = new BinaryValue( (byte[])val );
                    }
                    else
                    {
                        LOG.error( "The value's type is not String or byte[]" );
                        throw new NamingException( "The value's type is not String or byte[]" );
                    }
                }

                values = null;
                size = attribute.size();

                break;
        }
    }
    else
    {
        LOG.error( "Attribute must be an instance of BasicAttribute or AttributeImpl" );
        throw new NamingException( "Attribute must be an instance of BasicAttribute or AttributeImpl" );
    }
}


/**
 * Clone a value. This private message is used to avoid adding try--catch
 * all over the code.
 * @param value the value to clone
 * @return the value that was cloned
 */
private Value<?> getClonedValue( Value<?> value )
{
    if ( value == null )
    {
        return new StringValue( null );
    }
    try
    {
        return value.clone();
    }
```

```
    catch ( CloneNotSupportedException csne )
    {
        return new StringValue( null );
    }
}

// ----------------------------------------------------------------------
// ServerAttribute Interface Method Implementations
// ----------------------------------------------------------------------

/**
 * Gets a Iterator wrapped around the iterator of the value list.
 *
 * @return the Iterator wrapped as a NamingEnumberation.
 */
public Iterator<Value<?>> getAll()
{
    if ( size < 2 )
    {
        return new Iterator<Value<?>>()
        {
            private boolean more = (value != null);

            public boolean hasNext()
            {
                return more;
            }

            public Value<?> next()
            {
                more = false;
                return value;
            }

            public void remove()
            {
                value = null;
                more = true;
            }
        };
    }
    else
    {
        return values.iterator();
    }
}


/**
 * Gets the first value of the list or null if no values exist.
 *
 * @return the first value or null.
 */
public Value<?> get()
{
    switch ( size )
    {
        case 0 :
            return null;

        case 1 :
            return value;

        default :
            return values.get( 0 );
    }
}


/**
 * Gets the size of the value list.
```

```
     *
     * @return size of the value list.
     */
    public int size()
    {
        return size;
    }


    /**
     * Gets the id or name of this Attribute.
     *
     * @return the identifier for this Attribute.
     */
    public String getID()
    {
        return upId;
    }


    /**
     * Gets the OID of this Attribute.
     *
     * @return the OID for this Attribute.
     */
    public OID getOid()
    {
        return oid;
    }


    /**
     * Checks to see if this Attribute contains val in the list.
     *
     * @param val the value to test for
     * @return true if val is in the list backing store, false otherwise
     */
    public boolean contains( Value<?> val )
    {
        switch ( size )
        {
            case 0 :
                return false;

            case 1 :
                return AttributeUtils.equals( value, val );

            default :
                for ( Value<?> value:values )
                {
                    if ( AttributeUtils.equals( value, val ) )
                    {
                        return true;
                    }
                }

                return false;
        }
    }


    /**
     * Checks to see if this Attribute contains val in the list.
     *
     * @param val the value to test for
     * @return true if val is in the list backing store, false otherwise
     */
    public boolean contains( String val )
    {
        return contains( new StringValue( val) );
    }
```

```
/**
 * Checks to see if this Attribute contains val in the list.
 *
 * @param val the value to test for
 * @return true if val is in the list backing store, false otherwise
 */
public boolean contains( byte[] val )
{
    return contains( new BinaryValue( val) );
}


/**
 * Adds val into the list of this Attribute's values at the end of the
 * list.
 *
 * @param val the value to add to the end of the list.
 * @return true if val is added to the list backing store, false if it
 *         already existed there.
 */
public boolean add( Value<?> val )
{
    if ( contains( val ) )
    {
        // Do not duplicate values
        return true;
    }

    // First copy the value
    val = getClonedValue( val );

    switch ( size() )
    {
        case 0 :
            value = val;
            size++;
            return true;

        case 1 :
            // We can't store different kind of Values in the attribute
            // The null value is an exception, as it is not associated
            // with a StringValue or a BinaryValue
            if ( ( value.getValue() != null) && ( val.getValue() != null ) &&
                 ( value.getClass() != val.getClass() ) )
            {
                return false;
            }

            // value is never null.
            if ( value.equals( val ) )
            {
                // Don't add two times the same value
                return true;
            }
            else
            {
                values = new ArrayList<Value<?>>();
                values.add( value );
                values.add( val );
                value = null;
                size++;
                return true;
            }

        default :
            Value<?> firstValue = values.get( 0 );

            if ( firstValue.getValue() == null )
            {
```

```
                firstValue = values.get( 1 );
            }

            // We can't store different kind of Values in the attribute
            if ( ( val.getValue() != null ) &&
                 ( firstValue.getValue().getClass() != val.getClass() ) )
            {
                return false;
            }

            if ( values.contains( val ) )
            {
                // Don't add two times the same value
                return true;
            }
            else
            {
                values.add( val );
                size++;
                return false;
            }
    }
}


/**
 * Adds val into the list of this Attribute's values at the end of the
 * list.
 *
 * @param val the value to add to the end of the list.
 * @return true if val is added to the list backing store, false if it
 *         already existed there.
 */
public boolean add( String val )
{
    return add( new StringValue( val) );
}


/**
 * Adds val into the list of this Attribute's values at the end of the
 * list.
 *
 * @param val the value to add to the end of the list.
 * @return true if val is added to the list backing store, false if it
 *         already existed there.
 */
public boolean add( byte[] val )
{
    return add( new BinaryValue( val) );
}


/**
 * Removes val from the list of this Attribute's values.
 *
 * @param val the value to remove
 * @return true if val is remove from the list backing store, false if
 *         never existed there.
 */
public boolean remove( Value<?> val )
{
    if ( contains( val) )
    {
        switch ( size )
        {
            case 0 :
                return false;

            case 1 :
                value = null;
```

```
                size = 0;
                return true;

            case 2 :
                values.remove( val );
                value = values.get(0);
                values = null;
                size--;
                return true;

            default :
                values.remove( val );
                size--;
                return true;
        }
    }
    else
    {
        return false;
    }
}


/**
 * Removes val from the list of this Attribute's values.
 *
 * @param val the value to remove
 * @return true if val is remove from the list backing store, false if
 *         never existed there.
 */
public boolean remove( String val )
{
    return remove( new StringValue( val ) );
}


/**
 * Removes val from the list of this Attribute's values.
 *
 * @param val the value to remove
 * @return true if val is remove from the list backing store, false if
 *         never existed there.
 */
public boolean remove( byte[] val )
{
    return remove( new BinaryValue( val ) );
}


/**
 * Removes all the values of this Attribute from the list backing store.
 */
public void clear()
{
    switch ( size )
    {
        case 0 :
            return;

        case 1 :
            value = null;
            size = 0;
            return;

        default :
            values = null;
            size = 0;
    }
}
```

```
/**
 * Not a deep clone.
 *
 * @return a copy of this attribute using the same parent lock and id
 *         containing references to all the values of the original.
 */
public ServerAttribute clone()
{
    try
    {
        ServerAttributeImpl clone = (ServerAttributeImpl)super.clone();

        // Simply copy the OID.
        clone.oid = oid;

        if ( size < 2 )
        {
            clone.value = value.clone();
        }
        else
        {
            clone.values = new ArrayList<Value<?>>( values.size() );

            for ( Value<?> value:values )
            {
                Value<?> newValue = value.clone();
                clone.values.add( newValue );
            }
        }

        return clone;
    }
    catch ( CloneNotSupportedException cnse )
    {
        return null;
    }
}


/**
 * Checks for equality between this Attribute instance and another. The
 * Attribute ID's are compared with regard to case.
 *
 * The values are supposed to have been normalized first
 *
 * @param obj the Attribute to test for equality
 * @return true if the obj is an Attribute and equals this Attribute false
 *         otherwise
 */
public boolean equals( Object obj )
{
    if ( obj == this )
    {
        return true;
    }

    if ( ( obj == null ) || !( obj instanceof ServerAttributeImpl ) )
    {
        return false;
    }

    ServerAttributeImpl attr = ( ServerAttributeImpl ) obj;

    if ( attr.oid != oid )
    {
        return false;
    }

    if ( !upId.equalsIgnoreCase( attr.getID() ) )
    {
        return false;
```

```
    }

    if ( attr.size != size )
    {
        return false;
    }

    switch ( size )
    {
        case 0 :
            return true;

        case 1 :
            return ( value.equals( attr.get() ) );

        default :
            Iterator<Value<?>> vals = getAll();

            while ( vals.hasNext() )
            {
                Value<?> val = vals.next();

                if ( !attr.contains( val ) )
                {
                    return false;
                }
            }

            return true;
    }
}


/**
 * Normalize the attribute, setting the OID and normalizing the values
 *
 * @param oid The attribute OID
 * @param normalizer The normalizer
 */
@SuppressWarnings(value="unchecked")
public void normalize( OID oid, Normalizer normalizer ) throws NamingException
{
    this.oid = oid;

    switch ( size )
    {
        case 0 :
            return;

        case 1 :
            value.normalize( normalizer );
            return;

        default :
            for ( Value value:values)
            {
                value.normalize( normalizer );
            }
    }
}


/**
 * @see Object#toString()
 */
public String toString()
{
    StringBuilder sb = new StringBuilder();

    sb.append( "ServerAttribute id : '" ).append( upId ).append( "', " );
```

```
        if ( oid != null )
        {
            sb.append( " OID : " ).append( oid ).append( '\n' );
        }

        sb.append( " Values : [");

        switch ( size )
        {
            case 0 :
                sb.append( "]\n" );
                break;

            case 1 :
                sb.append( '\'' ).append( value ).append( '\'' );
                sb.append( "]\n" );
                break;

            default :
                boolean isFirst = true;

                for ( Value<?> value:values )
                {
                    if ( !isFirst )
                    {
                        sb.append( ", " );
                    }
                    else
                    {
                        isFirst = false;
                    }

                    sb.append( '\'' ).append( value ).append( '\'' );
                }

                sb.append( "]\n" );
                break;
        }

        return sb.toString();
    }
}
```