XML Security: Manage identities more effectively with SPML

The objectives, architecture, and basic concepts of Service Provisioning Markup Language

Manish Verma

05 January 2005

Gain a basic understanding of what Service Provisioning Markup Language (SPML) is and how it works. After an explanation of SPML's role in the management of the identity lifecycle, this article guides you through an actual working SPML scenario, using OpenSPML. Along the way, the author explains the architecture and design of SPML. Ultimately, you learn to appreciate the usefulness of this technology, and are equipped to participate in the implementation of the standard.

View more content in this series

The past couple of years have seen an increased interest in identity management. Managing identities effectively and efficiently is a critical issue for businesses, and various standards have been proposed to handle different aspects of identity management (see What is identity management?). One such standard is Service Provisioning Markup Language (SPML), which deals with resource provisioning for these identities. It brings standardization to the mundane but error prone job of preparing IT and support infrastructure to accomplish business activity. For example, with SPML it is possible to automate the provisioning workflow that results when an organization hires a new employee.

Provisioning workflow can include activities that are either digital or physical. As an example, when a new employee is hired, digital activities can include the creation of a user account in various systems and applications, while physical activities can include procurement of a new laptop for that individual.

In this article, I will explore the objectives and importance of the SPML standard. I take you through the creation of some sample programs that demonstrate how the standard helps you automate provisioning activities. For the sample code I will use openSPML, an open source implementation of SPML.

Why it's useful

What is identity management?

In the digital world, the user of a particular application, network, system, and so on is nothing but a user ID, which after authentication gets a unique identity. It is this digital identity that needs to be managed.

With the ever-increasing number and complexity of systems and networks, managing digital identities is now a major challenge. **Identity management** refers to the management of the entire lifecycle of one or more identities, from creation to destruction, and the things that happen in between -- such as managing permissions, privileges, and modifications.

Does the world really need a standard for provisioning? Well, just imagine big corporations that have hundreds of thousands of employees and huge numbers of IT systems, applications, and external partner systems. Think of the IT provisioning issues that arise when corporations merge or when new applications are rolled into production or when new employees join a company. IT provisioning issues can include things that are in the digital domain -- like giving access to computer systems, files, directories, or buildings -- as well as the physical domain -- like obtaining laptops, cell phones, or seating space for new employees. Now, consider all of the things that must be done promptly when an employee leaves, or when businesses terminate their relationships, to ensure that corporate assets are not used illegally by former employees or business partners.

So far so good -- you understand why provisioning is important. But, why is a *standard* to be used consistently across organizations needed? In this connected world, organizations have linked their supply chains and increasingly use each other's systems to extract maximum efficiencies and productivity from the combined resources. A standard that is used uniformly helps achieve the management of provisioning across different organizations and disparate systems.

What it aims to achieve

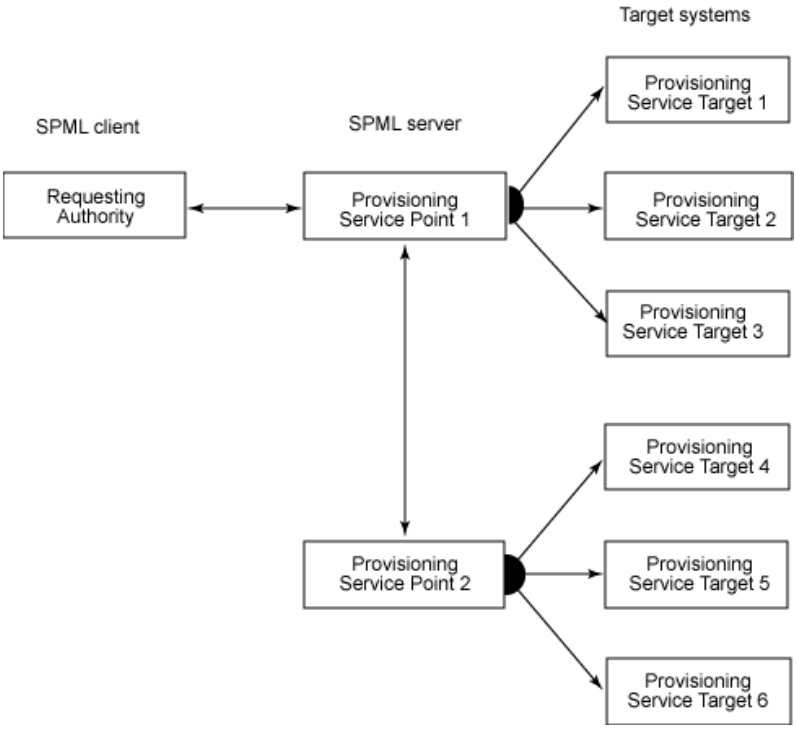
SPML aims to achieve a couple of things:

- **Automated IT provisioning tasks:** By standardizing the job of provisioning and making it easier to encapsulate the security and auditing requirements of provisioning systems, SPML pushes provisioning towards as much automation as possible.
- **Interoperability between different provisioning systems:** Different provisioning systems can now expose standard SPML interfaces to each other and interoperate with each other.

The components of the provisioning system and how they fit together

A provisioning system is made up of three essential components: The Requesting Authority (RA), the Provisioning Service Point (PSP), and the Provisioning Service Target (PST). They are illustrated in Figure 1:

Figure 1. SPML architecture



- **Requesting Authority (RA):** This is the client in the SPML scheme of things. It creates well-formed SPML documents and sends them as requests to the SPML service point. These requests describe an operation to be performed at specific service points. For an RA to issue a request to an SPML service point, a trust relationship must exist between the RA and the SPML service point. Even an SPML service point can act as an RA when it issues an SPML request to another service point.
- **Provisioning Service Point (PSP):** This is the component that listens to the request from the RA, processes it, and returns a response to the RA. Any component that listens and processes well-formed SPML documents is called a Provisioning Service Point.
- **Provisioning Service Target (PST):** This is the actual software on which the action is taken. For example, it could be an LDAP directory that stores all of an organization's user accounts, or it could be an IT ticketing system that is used to log IT requests (such as obtaining a laptop for a new employee).

So as you can see, the architecture is essentially a client (RA), a server (PSP), and resources (PSTs) that SPML manages. In addition, a server can act as a client to another server and so on.

SPML explored

Now I'll take you into the guts of the SPML standard. Keep in mind that this is just version 1.0 of the spec, so do not expect it to be rich in functionality. Version 2.0 is in process and expected to have more cool features.

In SPML, you will see three things:

- A core set of operations

- Request response protocol
- A service schema definition

SPML core operations

In the SPML 1.0 specifications, the following operations are defined as core: Add, Modify, Delete, and Search. This essentially means that all PSTs make these four operations available for manipulation as SPML-defined interfaces.

To illustrate this further, assume that you are a system administrator who is adding a new employee. This employee must get access to a trouble ticket system that someone else manages. One way for you to get the new employee registered for this trouble ticket system is to e-mail or call the owner of that system, and wait for a response. SPML provides a structured and automated means of dealing with such requests. Using the SPML approach, the four core operations are available on the trouble ticket system, so you simply use the Add operation to add the new employee to that system.

One more thing before I leave the topic of core operations: SPML 1.0 does allow you to define operations for services other than the four core operations listed here. These are called **extended operations** and allow clients to make requests and receive responses with predefined syntaxes and semantics.

SPML request response protocol

SPML request response protocol covers how various SPML components talk to each other -- primarily the client (RA) and the server (PSP). This protocol dictates that the request issued by the client describes one or more operations to be performed on a specific service point. You can submit these requests individually or in a batch, and select either synchronous or asynchronous submission for both request types.

To complete your understanding of the request response protocol, you need to dig deeper into two different aspects of it:

- Request execution model (synchronous or asynchronous)
- Grouping of requests (individual or batch)

Request execution model

Let me first explain how SPML dictates support for synchronous and asynchronous execution of requests. Requests processed synchronously are blocking calls. An RA keeps its execution call open until it receives a response on that call. In the case of asynchronously executed requests, the execution model gets a little complex. To begin with, the RA issues each request with a unique request ID and does not wait for a response on the same execution call. The request ID is maintained by the PSP for the duration of the request execution, and is returned to the client along with the response when the response is ready. While the request is processed, the client may be interested in knowing the status of the request. This is where the request ID comes in handy. The request ID is also used to control and manage the pending and executing SPML requests at the server.

SPML provides two operations to manage and control asynchronous request execution:

- `StatusRequest` -- This operation allows clients to get the current status of the request that's being executed asynchronously. A variant of the `StatusRequest` operation allows the clients to get not only the current results, but also the current result set, if available.
- `CancelRequest` -- This operation allows the RA to cancel the execution of a pending asynchronous request.

Grouping of requests
The RA can issue a request either individually or in a batch. In either case, requests can be executed synchronously or asynchronously. Individual requests are handled in a standard manner, but the batch requests have a kink that you need to understand.

As you probably know, a batch request is multiple requests grouped together. With batch processing, it's important to understand three things:

- How results are sent back for each of the requests in a batch
- How multiple requests are processed (in parallel or sequentially)
- How errors are handled when some batch requests fail

I will explain these three issues within the context of SPML.

Result processing: Multiple SPML operations are collected together and issued as a single `BatchRequest`. The results are processed by the PSP and sent as a `BatchResponse` based on positional correspondence -- which essentially means that the first response in a `BatchResponse` corresponds to the first request of the `BatchRequest`, the second response corresponds to the second request, and so on.

Processing types: SPML supports both sequential and parallel processing of requests. The RA can determine how the processing is done. In the case of sequential processing, requests are processed in the order in which they are mentioned in the batch request. In case of parallel processing, the server can execute the batched requests in any order. In both the cases, however, the response is determined by the positional correspondence.

Error handling: SPML provides two options -- resume and exit -- for handling cases where some of the requests in a batch fail. The RA specifies one of these as the preferred option at the time a batch request is issued. When the instruction to the server is to resume, the failure of an individual operation does not affect execution of the remaining requests, and the positional correspondence is maintained. When the instruction is to exit, the server terminates the execution of the remaining requests as soon as an error is encountered, and all the requests that do not execute are marked as failed.

Object class and attribute-sharing model

An **object class** is a convenient mechanism for putting attributes together in a group and referencing them together by a name. Object classes are used in making requests and responses.

With an **attribute-sharing model**, SPML allows object classes to be referenced by other classes, thus allowing sharing of the attributes across multiple linked classes.

SPML service schema definition

Service schema allows clients to interrogate the SPML server for the details of the operations that it supports.

SPML service schema is based on W3C XML Schema, and adds to it the object class definition and attribute-sharing model (see Object class and attribute-sharing model). In the sample service schema in Listing 1, note how attributes are defined in XML Schema style. You can also see how attributes are grouped together under the `ObjectClassDefinition`. I use this same service schema in the SPML code samples later in the article.

Listing 1: Service schema

```
<schema majorVersion="1" minorVersion="0">
<providerIdentifier providerIDType="urn:oasis:names:tc:SPML:1:0#URN">
<providerID>urn:oasis:names:tc:SecF</providerID>
</providerIdentifier>
<schemaIdentifier schemaIDType="urn:oasis:names:tc:SPML:1:0#GenericString">
<schemaID>PersonSchema</schemaID>
</schemaIdentifier>
<attributeDefinition name="FullName"
                     description="Full name of the employee joining."/>
<attributeDefinition name="email" description="E-mail address."/>
<attributeDefinition name="description" description="Description."/>
<attributeDefinition name="project" description="Project assigned to."/>
<objectclassDefinition name="employee" description="Sample employee.">
<memberAttributes>
<attributeDefinitionReference name="FullName" required="true"/>
<attributeDefinitionReference name="email" required="true"/>
<attributeDefinitionReference name="description"/>
<attributeDefinitionReference name="project" required="true"/>
</memberAttributes>
</objectclassDefinition>
</schema>
```

Clients use the same request response protocol described above to get the schema information. The operation provided by SPML for obtaining schema information is called `SchemaRequest`. `SchemaRequest` is used to retrieve specific provisioning schemas. These schemas are retrieved using identifiers -- specifically, the **provider identifier** and the **schema identifier**. If no schema identifier is provided, all the schemas supported by the provider are returned. The provider identifier is the unique identifier of a provider, and can be either an organization or an individual who is responsible for a set of operations. A schema identifier is the unique identifier of a schema that a provider offers. Its uniqueness is guaranteed within the context of the provider.

Roll up your sleeves and get your hands dirty

In this section, I take you through some sample SPML code. The objective here is to create an e-mail account for a new employee. Now assume that you are a human resources (HR) rep and you want to create an e-mail account for the new employee. You call the mail server administrator. Your conversation with the administrator may go as follows:

You: A new employee has joined the organization. I want to get his e-mail account created. What information do you need from me to set up the e-mail account?
Admin: Please give me the employee's full name, preferred e-mail ID, a description of the job roles and responsibilities, and the project to which the employee is assigned.
You: Sure. (You provide all the details.) Please let me know when the account is created. Also let me know if any restrictions apply to the account.
Admin: I will get back to you when the mail account is created. I will also send applicable restriction details.

In the SPML world, this conversation gets reduced to a SOAP-encoded SPML message exchange between two systems -- the HR system and the mail server in which e-mail account of the new employee is to be created. The sequence of the SPML messages exchanged is as shown below.

SPML message 1: The HR system (RA) sends a `SchemaRequest` to the PSP responsible for managing the mail server (PST) to find out what the mail server needs to create a new e-mail account.
SPML message 2: The PSP sends a `SchemaResponse` back to the HR system. This `SchemaResponse` includes details of the information required by the mail server (PST) for mail account creation.

Listing 2 illustrates the creation of both `SchemaRequest` and `SchemaResponse` in synchronous execution mode. The provisioning schema that I use for this sample (SchemaResponse.xml) is available for download as part of the resource bundle (see Resources).

In the `SchemaRequest` code snippet below, I demonstrate the creation of a generic `SchemaRequest` -- that is, I neither specified a provider ID nor gave a schema ID. All of the schemas that are available with the PSP are returned to the RA. However, in this sample only one schema is returned as I only specified one schema in the PSP.

Listing 2: SchemaRequest and SchemaResponse

```
public class UseSPML {

    private static void mySchemaRequest(SpmlClient client) {
        SchemaRequest schemaRequest = new SchemaRequest();

        try {
            SchemaResponse schemaResponse = (SchemaResponse) client
                        .request(schemaRequest);
        } catch (SpmlException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();

    }

    public static void main(String[] args) {

        SpmlClient client = new SpmlClient();
        try {
            // Replace the URL with the one where your SPML server
// might be running
            client.setUrl("http://localhost:8080/spml");
        } catch (MalformedURLException e1) {
            // TODO Auto-generated catch block
```

```
            e1.printStackTrace();
        }
        client.setTrace(true);

        mySchemaRequest(client);
    }
}


// Handling of SchemaRequest by PSP and generation of SchemaResponse.
// Schema returned in SchemaResponse is read from a file named
// SchemaResponse.xml.
// This file is available for download as part of the resource bundle.


public class SPMLServer implements SpmlHandler {

    public SpmlResponse doRequest(SpmlRequest request) {

        SpmlResponse response = null;

        if (request instanceof SchemaRequest) {

            response = (SchemaResponse) ((SchemaRequest) request)
                        .createResponse();
            try {
                        ((SchemaResponse) response).addSchema(new Schema(
                            readFile("SchemaResponse.xml")));
            } catch (SpmlException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
            }

        }

    }

    private static String readFile(String fileName) {

        FileReader fr = null;
        try {
            fr = new FileReader(fileName);
        } catch (FileNotFoundException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        BufferedReader br = new BufferedReader(fr);

        StringBuffer fileBuffer = new StringBuffer();
        String rec = new String();

        try {
            while ((rec = br.readLine()) != null) {
                        fileBuffer = fileBuffer.append(rec);
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        return fileBuffer.toString();
    }

}
```

As you can see in Listing 2, the `doRequest` method of the class `SPMLServer` returns a `SchemaResponse` as read from the file SchemaResponse.xml.

SPML message 3: Based on the schema received in `SchemaResponse`, the HR system (RA) sends an `AddRequest` to the PSP for adding a new mail account.
SPML message 4: The PSP initiates the process of creating the e-mail account by calling the mail server APIs directly, or by further passing an SPML `AddRequest` to the mail server. In Listing 3, the PSP simply records the e-mail account creation details in a file. The PSP sends an `AddResponse` message back to the HR system with the specifics of the applicable restrictions.

Listing 3 shows the `AddRequest` and `AddResponse` message exchange between the RA and the PSP. I create a mail account for a user named "Manish Verma." In response, the PSP sends the mail ID created and the restriction on the size of the mailbox.

Listing 3 AddRequest and AddResponse

```
public class UseSPML {

    private static void myAddRequest(SpmlClient client) {

        AddRequest req = new AddRequest();
        req.setObjectClass("urn:oasis:names:tc:SPML:PersonSchema:employee");
        req.setIdentifier("spmlserver");
        req.setAttribute("FullName", "Manish Verma");
        req.setAttribute("email", "mverma@secf.com");
        req.setAttribute("description", "Center Head");
        req.setAttribute("Project", "SecF India");

        try {
            AddResponse response = (AddResponse) client.request(req);
        } catch (SpmlException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

    public static void main(String[] args) {

        SpmlClient client = new SpmlClient();
        try {
            // Replace the URL with the one where your SPML server
            // might be running
            client.setUrl("http://localhost:8080/spml");
        } catch (MalformedURLException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
        client.setTrace(true);
        myAddRequest(client);

    }
}

Handling of AddRequest by  PSP and generation of AddResponse


public class SPMLServer implements SpmlHandler {

    public SpmlResponse doRequest(SpmlRequest request) {
```

```
        SpmlResponse response = null;

        if (request instanceof AddRequest) {

            List attributes = ((AddRequest) request).getAttributes();

            BufferedWriter bw = null;
            try {
                bw = new BufferedWriter(new FileWriter(
                        "C:\\work\\SPML\\addreqdoc.txt"));
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            StringBuffer addRequestString = new StringBuffer("AddRequest ");
            addRequestString.append(writeln(""));

            for (int i = 0; i < attributes.size(); i++) {
                addRequestString.append(((Attribute) attributes.get(i))
                        .getName());
                addRequestString.append(" ");
                addRequestString.append(((Attribute) attributes.get(i))
                        .getValue());
                addRequestString.append(writeln(""));
            }

            try {
                bw.write(addRequestString.toString());
                bw.close();

            } catch (IOException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }

            response = (AddResponse) ((AddRequest) request).createResponse();
            Identifier addResponseId = new Identifier();
            addResponseId.setType(Identifier.TYPE_EMailAddress);
            addResponseId.setId((String) ((AddRequest) request)
                        .getAttributeValue("email"));
            ((AddResponse) response).setIdentifier(addResponseId);
            ((AddResponse) response).setAttribute("MailBoxLimit", "500MB");

        }

        return response;
    }


    // utility method that puts line separator characters to the end
    // of the input string
    private static String writeln(String string) {
        String lineSeparator = System.getProperty("line.separator");
        return string + lineSeparator;
    }

}
```

You can download the complete code in Resources. To execute and work with the samples, first install and configure openSPML. Then replace openSPML's default PSP (`org.openspml.test.TestSpmlHandler`) with the `SpmlHandler` that's provided in the resource bundle (`org.eklavya.spml.SpmlServer`). The SPML client is in the class `org.eklavya.spml.UseSpml`. Also, change the settings in these two files to match your setup.

What you've learned

You now know where provisioning fits into the overall identity management space. You realize the benefits of automation and interoperability that accrue when SPML is adopted as a standard. Using OpenSPML, you now know how to configure an SPML server and how SPML messages are exchanged.

I encourage you to play with the technology and maintain an understanding of this standard. Whenever your organization discusses or implements identity management, you can draw upon your knowledge and understanding of SPML to add value.

Downloadable resources

Description	Name	Size
Sample code for SPML	x-secspml1_SPMLbundle.zip	6 KB