# Approach to support implementation specific schemes

A DSS-X server may support a set different profiles. In contrast to the DSS core version 1.0 it’s a design goal of DSS-X core version 2 to maximize type safety and developer support by avoiding the xs:any construct. The xs:any places the burden of handling incoming generic requests against an implicit scheme upon the developer and also shifts the fall out of incompatibilities to the runtime. No static type safety and IDE support is available.

Therefore, a DSS-X endpoint should provide a specific WSDL representing the structure its requires explicitly. This requires endpoints to provide a WSDL and schema specific to a supported sets of profiles and introduces a restriction of the ability of a single server endpoint to serve an arbitrary set of profiles at once. But in a real world scenarios DSS endpoints are usually bound to specific tasks reflected by access rights, audit requirements, visibility and the supported profile set. It is unlikely that one endpoint supports the creation of personalized qualified signatures, code signatures and ebXML signatures by just selecting different profiles. So it seems to be a valid assumption that the vast majority of endpoints are single-purpose endpoints. An example of a set of endpoints may look this:

|  |  |  |
| --- | --- | --- |
| **endpoint** | **profiles** | **accessable for** |
| /dss/signDocument | xades, eseal, async | accounting |
| /dss/signJar | j2me, async | development |
| /dss/verify | verificationReport, async | whole intranet |

The published set of profiles should be interpreted as the maximum offer of an endpoint. Not requesting a profile (e.g. not requesting the async profile while it is supported) does not violate the comprehensive schema declared. For example, the endpoint ‘/dss/signJar’ (of the table above) may be called by a build tool unaware of the asynchronous profile without any problems.

## Schema handling using ‘xs:redefine’

Schema support for simple cases where one single profile extends the DSS-X core the resulting schema can be defined by using the ‘xs:redefine’ element is used:

<xs:redefine schemaLocation="core-schema.xsd">

<xs:complexType name="dss:OptionalOutputsVerifyType">

<xs:complexContent>

<xs:extension base="dss:OptionalOutputsVerifyType">

<xs:group ref="prf:optionalOutputGroup"/>

</xs:extension>

</xs:complexContent>

</xs:complexType >

</xs:redefine>

The snippet above extends the set of sub-components of OptionalOutputsVerifyType with the group of elements from a given profile.

With this mechanism it is possible to extend the core schema to specific requirements while preserving the advantage of type safety and tool / IDE support. This sample illustrates the use of ‘extension’. in the same way ‘restriction’ can be applied. In more complex scenarios (e.g. multiple profiles apply, need for extending **and** restriction the core schema) the use of other techniques (e.g. XSLT) may be required.

## Schema handling using XSLT

A solution using XSLT may look like this:

1. Use the core schema and parameters selecting the active profiles as input for the stylesheet
2. Add schema’s prefix within the root element
3. Add ‘Imports’ for schema files
4. Append OptionalInputs and –Outputs to the existing sequence/choice of elements:

<xsl:template match="//xs:complexType[@name = 'OptionalOutputsVerifyType']/xs:complexContent/xs:extension/xs:sequence/xs:choice">

<xs:copy>

<xs:apply-templates select="@\*|node()"/>

<xs:if test="$includeVerificationProfile = 'true'">

<xs:group ref="vr:optionalOutputGroup"/>

</xs:if>

</xs:copy>

</xs:template>

The sample, straight forward stylesheet (enableProfileSpecifics.xsl) can easily be expanded to a bigger set of profiles. Using the XSLT technology it’s also possible to rewrite existing elements of the core. It would be easy to setup a web service that creates the schema files for a given set of profiles. In contrast to the ‘xs:redefine’ variant using XSLT has no limitation in the number of elements / element groups to be inserted.

## Multiple syntaxes

The creation of a JSON schema is more tricky. The current approach uses the XML schema as the source, builds a Java object model from it, compiles the model, executes it, derives a raw JSON schema and performs some post processing. Delegating this build pipeline to a developer is not recommended while manual modifications are tedious and error prone.

## Providing pre-packaged profile sets

A more promising approach would be the provisioning of a pre-processed set of XML and JSON schemes and their generated set of Java classes. This would help an interested developer to have a quick start implementing DSS-X components using the generated object model without digging into details of the syntax dependent definitions. To leverage further development support the explanatory text fragments managed with the Word specification should be fed back as JavaDoc comments into the generated classes.

A proposed set of pre-packed profiles may include

* Plain core
* Verification centric set: verificationReport, evidence record, async
* Signing set: XAdES, PAdES, async
* Preservation Service
* ChipGateway