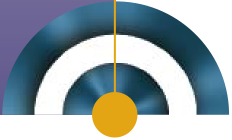


Service Data Objects

cross-language data access API

Het is tegenwoordig misschien moeilijk voor te stellen maar er zijn nog steeds situaties waarbij je geen netwerkverbinding hebt. Hier ben ik de afgelopen tijd verscheidene malen tegenaan gelopen. Zo blijken UMTS en GPRS niet goed te werken in operatiekamers van ziekenhuizen. Ook in fabrieken in derdewereldlanden blijk je niet altijd een goede netwerkverbinding te hebben.

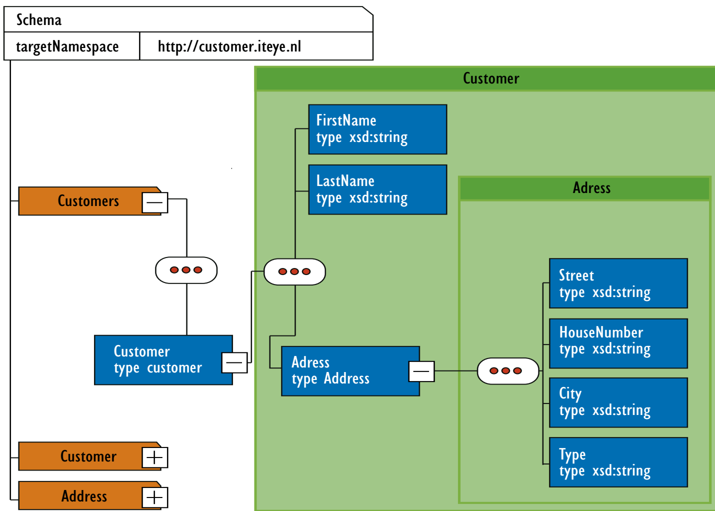
De applicaties in kwestie maken gebruik van een centrale serverapplicatie met database- en client-applicaties op laptops. Deze client-applicaties moeten dus zonder netwerkverbinding gebruikt kunnen worden. Je kunt dit op verschillende manieren aanpakken, bijvoorbeeld door op iedere client een database te draaien die met de centrale database gesynchroniseerd wordt. Oracle Lite biedt hiervoor een aardige oplossing. Een andere aanpak, die mooi past binnen een SOA-visie, is om gebruik te maken van XML-documenten en webservices. De client haalt met behulp van een webservice alle data in een XML-document binnen. Deze kan offline bewerkt worden. Zodra de gebruiker klaar is, en er weer connectie gemaakt kan worden met de centrale server, wordt het XML-document teruggestuurd en worden de wijzigingen in de centrale database aangebracht.
Je kunt het teruggestuurde XML-document in zijn geheel opslaan in de database, bijvoorbeeld met behulp van Oracle XMLDB, maar meestal zie je dat er gebruik gemaakt wordt van een relationeel datamodel. Dit kan ook met behulp van XMLDB, maar vaker zie je dat het XML-document eerst ingelezen wordt in Java, en dat vervolgens de database wordt bijgewerkt. Meestal heb je dus te maken met twee onafhankelijke frameworks: een XML-framework en een objectrelationeel framework. Bijkomende uitdagingen zijn: hoe vertaal je een relationeel model naar een hierarchisch model, hoe bepaal je welke data gewijzigd zijn, en hoe ga je om met gelijktijdige wijzigingen op verschillende clients? De data zullen namelijk meestal langer detached zijn dan in online scenario's. Al met al niet onoplosbaar, zelfs geen rocket science, maar ook niet heel simpel. De huidige focus op SOA zal het gebruik van detached XML-documenten waarschijnlijk alleen maar doen toenemen. Mijn interesse was dus snel gewekt toen ik zag dat het een van de doelen van SDO is om voor bovenstaande situatie een gestandaardiseerde oplossing te bieden.

Wat is SDO?

SDO staat voor Service Data Objects. SDO is een cross-language data access API. Dit betekent dat je, ongeacht de programmeertaal, op een gestandaardiseerde wijze met data kunt omgaan in applicaties. SDO biedt, onder andere via DAS (Data Access Service), ondersteuning voor verschillende databronnen, zoals XML, relationele databases, ldap, enzovoort. De SDO-specificatie wordt door een groot aantal bedrijven binnen het OSOA-samenwerkingsverband opgesteld. OSOA staat voor Open Service Oriented Architecture. Ook SCA (Service Component Architecture) wordt door deze club ontwikkeld. De reden dat SDO en SCA niet als JSR-standaard ontwikkeld worden, is dat SDO en SCA platformonafhankelijke specificaties zijn. Behalve een implementatie in Java is er bijvoorbeeld ook een PHP-implementatie. Hieronder zal ik aan de hand van een aantal voorbeelden de belangrijkste kenmerken van SDO toelichten. Het is namelijk mijn ervaring dat je een nieuwe technologie vaak het snelst oppakt via een aantal voorbeelden.

Dynamic data API

Data worden in SDO gerepresenteerd in de vorm van een Data Object. Elk dataobject bevat één of meer properties. Je kunt deze dataobjecten dynamisch of statisch gebruiken. In het statische geval genereer je Java-klassen op basis van een XML Schema. Je kunt een XML Schema-document echter ook direct in je code gebruiken om datatypen te definiëren. Dit wordt in het eerste voorbeeld getoond. Als uitgangspunt wordt de volgende XSD gebruikt:



De eerste stap is het definiëren van de datatypen die je gaat gebruiken. Met behulp van een XSDHelper kun je de Complex Typen in een XSD-document als dataobjecttypen definiëren. Vervolgens worden de dynamische SDO API's gebruikt om een datatype te instantieeren en op te slaan in een XML-document.

```
// definieer data typen
URL url = getClass().getResource("/nl/iteye/sdoexamples/Customer.xsd");
HelperContext scope = SDOUtil.createHelperContext();
scope.getXSDHelper().define(url.openStream(), url.toString());

// instantieer customer object
DataObject customer =
    DataFactory.INSTANCE.create("http://customer.iteye.nl/",
        "Customer");
customer.set("FirstName", "Victor");
customer.set("LastName", "van Dort");

// instantieer customer address object
DataObject address1 = customer.createDataObject("Address");
address1.set("Street", "darkalley");
address1.set("Type", "home");

// save customer in xml document
scope.getXMLHelper().save(customer, "http://customer.iteye.nl/",
    "Customer", System.out);
```

Zoals in dit voorbeeld getoond wordt, ondersteunt SDO geneste dataobjecten. Binnen een scope bestaat één definitie van een datatype. Wil je meer definities hanteren, dan kun je meer scopes creëren. Dit kan handig zijn als je bijvoorbeeld een dynamische en een statische definitie van customer wilt gebruiken.

Dynamic data API zonder XSD

In het vorengenoemde voorbeeld werd een XML Schema-document gebruikt om datatypen te definiëren. Datatypen kunnen echter ook programmatisch gedefinieerd worden zoals het volgende voorbeeld toont. Met behulp van een DataFactory wordt eerst een type gedefinieerd. Vervolgens wordt gedefinieerd welke properties dit type heeft.

```
HelperContext scope = SDOUtil.createHelperContext();

// definieer Customer type
DataObject customerType =
    scope.getDataFactory().create("commonj.sdo", "Type");
customerType.set("uri", "http://customer.iteye.nl/");
customerType.set("name", "Customer");

// definieer properties van Customer
DataObject firstNameProperty = customerType.
createDataObject("property");
firstNameProperty.set("name", "FirstName");
firstNameProperty.set("type",
    scope.getTypeHelper().getType("commonj.sdo", "String"));
DataObject lastNameProperty = customerType.createDataObject("property");
lastNameProperty.set("name", "LastName");
lastNameProperty.set("type",
    scope.getTypeHelper().getType("commonj.sdo", "String"));

// maak customer type bekend binnen scope
scope.getTypeHelper().define(customerType);
```

Nu het customer-type bekend is, kan het op dezelfde wijze gebruikt worden als in het vorige voorbeeld. SDO biedt de mogelijkheid om voor typen die op bovenstaande manier gedefinieerd zijn een XSD te genereren.

```
// print xsd voor dynamisch gedefinieerde typen
Type[] types = new Type[] {
  scope.getTypeHelper().getType("http://customer.iteye.nl/", "Customer")
  };
System.out.println(scope.getXSDHelper().generate(Arrays.asList(types)));
```

Het resultaat hiervan is, zoals verwacht mag worden, het volgende:

```
<?xml version="1.0"?>
<xs:schema xmlns:sdo="commonj.sdo"
  xmlns:sdoJava="commonj.sdo/java"
  xmlns:stn_1="http://customer.iteye.nl/"
  xmlns:xs="http://www.w3. org/2001/XMLSchema"
  attributeFormDefault="qualified" elementFormDefault="qualified"
  targetNamespace="http://customer.iteye.nl/">
  <xs:complexType abstract="false" name="Customer">
    <xs:sequence />
    <xs:attribute name="FirstName" type="xs:string" />
    <xs:attribute name="LastName" type="xs:string" />
  </xs:complexType>
  <xs:element name="customer" type="stn_1:Customer" />
</xs:schema>
```

Statisch data API

Op basis van een XSD of databaseobjecten kunnen statische SDO dataobjecten gegenereerd worden. Hiervoor is onder andere een Maven-plugin beschikbaar. Naast interface en implementatieklassen voor alle complexe typen in een XML Schema

wordt ook een factory-class gegenereerd. Deze moet eerst geregisteerd worden voordat je de bijbehorende typen kunt gebruiken in Java. Daarna laten de classen zich grotendeels gebruiken als normale pojos.

```
// registreer sdo factor voor statische typen
HelperContext scope = SDOUtil.createHelperContext();
StaticSdoFactory.INSTANCE.register(scope);

// instantieer statische sdo type
Customer customer = StaticSdoFactory.INSTANCE.createCustomer();
customer.setFirstName("Barkis");
customer.setLastName("Bittern");
```

SDO biedt ook Xpath-ondersteuning. Je kunt bijvoorbeeld een dataobject selecteren op basis van een property-waarde zoals in het volgende voorbeeld getoond wordt.

```
// selecteer klant mbv xpath expressie
Customer victoria = (Customer)((DataObject)customers).getDataObject
    ("//Customer[FirstName='Victoria']");
```

Change summary

Een SDO datagraph is een container waarbinnen data en wijzigingen worden bijgehouden. Een datagraph bevat één root dataobject. Wijzigingen worden in een datagraph bijgehouden in een Change Summary. Het volgende voorbeeld toont hoe een datagraph geïnstantieerd wordt met daarin één customer-object.

```
// definieer typen mbv xsd document
URL url = getClass().getResource("/nl/iteye/sdoexamples/Customer.xsd");
HelperContext scope = SDOUtil.createHelperContext();
scope.getXSDHelper().define(url.openStream(), url.toString());

// instantieer een datagraph
DataGraph dg = SDOUtil.createDataGraph();
DataObject customer =
   scope.getDataFactory().create("http://customer.iteye.nl/",
"Customer");
DataObject rootObject = dg.createRootObject(customer.getType());
customer = rootObject;
customer.set("FirstName", "Victor");
customer.set("LastName", "van Dort");

// datagraph opslaan
SDOUtil.saveDataGraph(customer.getDataGraph(), System.out, null);
```

Het resultaat hiervan is het volgende XML-document.

```
<?xml version="1.0" encoding="ascii"?>
<sdo:datagraph xmlns:customer="http://customer.iteye.nl/"
xmlns:sdo="commonj.sdo">
  <customer:Customer>
    <customer:FirstName>Victor</customer:FirstName>
    <customer:LastName>van Dort</customer:LastName>
  </customer:Customer>
</sdo:datagraph>
```

Om ervoor te zorgen dat ook wijzigingen worden opgeslagen in een datagraph kun je de beginLogging-methode gebruiken van het ChangeSummary-object. Na aanroep van deze methode worden alle wijzigingen als onderdeel van het ChangeSummary-object bijgehouden. In het volgende voorbeeld wordt één property van waarde gewijzigd en wordt vervolgens de datagraph weer als XML-document opgeslagen.

```
// start logging
customer.getChangeSummary().beginLogging();

// wijzig een property van waarde
customer.set("FirstName", "Pieter");

// datagraph opslaan als xml document
ChangeSummary.SDOUtil.saveDataGraph(customer.getDataGraph(), System.out,
null);
```

Het resulterende XML-document bevat nu naast de actuele waarden van het Customer-object ook een apart onderdeel waarin de wijzigingen staan genoteerd.

```
<?xml version="1.0" encoding="ascii"?>
<sdo:datagraph xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:customer="http://customer.iteye.nl/" xmlns:sdo="commonj.sdo"
    xmlns:sdo_1="http://www.apache.org/tuscany/2005/SDO">
  <changeSummary xmlns="" logging="true">
    <objectChanges key="#//@eRootObject">
      <value xsi:type="sdo_1:ChangeSummarySetting"
             featureName="FirstName" dataValue="Victor" />
    </objectChanges>
    <resourceChanges resourceURI="root.xml" />
  </changeSummary>
  <customer:Customer>
    <customer:FirstName>Pieter</customer:FirstName>
    <customer:LastName>van Dort</customer:LastName>
  </customer:Customer>
</sdo:datagraph>
```

Data Access Service

Dataobjecten bevatten meestal data uit een databron, bijvoorbeeld een database. Voor een aantal veelgebruikte databronnen bevat SDO standaard oplossingen. Deze zogenaamde Data Access Services, ook wel DAS genoemd, instantiëren datagraphs aan de hand van data uit een databron. Ook zorgen ze er voor dat wijzigingen in een datagraph teruggeschreven worden naar een databron.

Het volgende voorbeeld toont hoe DAS-RDB, Data Access Service voor Relationele Databases, gebruikt kan worden om data te selecteren en te persisteren. Allereerst wordt in een XML-configuratiebestand beschreven welke tabellen er zijn, en hoe deze mappen op de SDO Data Objecten. Voor iedere tabel kan aangegeven worden welke SDO datatype gebruikt moet worden, en ook kunnen kolommen op properties gemapt worden. Indien je gebruik maakt van foreign keys worden databaserelaties ook in dit configuratiebestand beschreven. Daarnaast bevat het configuratiebestand ook SQL queries. Het configuratiebestand is dus in grote lijnen vergelijkbaar met wat je gewend bent bij bijvoorbeeld Hibernate of Ibatis.

```
<?xml version="1.0" encoding="windows-1252" ?>
<Config xmlns="http:///org.apache.tuscany.das.rdb/config.xsd"
        dataObjectModel="http://customer.iteye.nl/rdb/">
 <!--
   == map table to data object
   -->
 <Table tableName="CUSTOMERS" typeName="Customer">
  <Column columnName="ID" primaryKey="true" propertyName="Id"/>
  <Column columnName="FIRSTNAME" primaryKey="false"
propertyName="FirstName"/>
  <Column columnName="LASTNAME" primaryKey="false"
propertyName="LastName"/>
 </Table>
 <!--
   == definieer commando voor het selecteren van klanten
   -->
<Command name="AllCustomers"
          SQL="select * from CUSTOMERS"
          kind="Select">
 </Command>
</Config>
```

Het is nu vrij eenvoudig om data uit een database te selecteren en weg te schrijven in een XML-document. We beginnen weer met een XML Schema-document dat alle typen definieert. Daarna wordt een Data Access Service geïnstantieerd. Vervolgens wordt het select-statement uit de configuratie uitgevoerd en het resultaat in dataobjecten gezet. Deze zijn nu eenvoudig in een XML-document te schrijven.

```
// definieer alle data typen
URL url = getClass().getResource("/nl/iteye/sdoexamples/ex5/Customer-
RDB.xsd");
HelperContext scope = HelperProvider.getDefaultContext();


scope.getXSDHelper().define(url.openStream(), url.toString());

// instantieer DAS mbv configuratie
InputStream is = getClass().getClassLoader()
    .getResourceAsStream("nl/iteye/sdoexamples/ex5/customer-das-config.
xml");
```

```
DAS das = DAS.FACTORY.createDAS(is, getConnection());

// selecteer alle klanten
Command readAll = das.getCommand("AllCustomers");
DataObject customers = readAll.executeQuery();

// save customer data in xml document
XMLHelper.INSTANCE.save(customers, "http://customer.iteye.nl/rdb/",
                                "Customers", System.out);
```

Ook het opslaan van wijzigingen is vrij eenvoudig. Nadat data gewijzigd zijn, kunnen alle wijzigingen gepersisteerd worden met behulp van de methode applyChanges:

```
// wijzig data
DataObject addressHarold = customers.getDataObject("Customer[LastName='H
arold']");
addressHarold.setString("FirstName", "kerkstraat");
das.applyChanges(customers);

// close connection
closeConnection();
```

In vergelijking met andere persistency-raamwerken zoals Hibernate of Toplink is bovenstaande niet echt revolutionair. Maar in combinatie met de eerder getoonde Change Summaries kan het voor een aantal projecten wel een flinke vereenvoudiging betekenen.

SCA en SOA

SDO wordt meestal samen met SCA gepresenteerd als belangrijke onderdelen van een SOA-oplossing. SCA biedt de mogelijkheid tot het definiëren van composite applications. Dit zijn applicaties die door gebruik te maken van al bestaande services nieuwe samengestelde applicaties vormen. Een belangrijk uitgangspunt van SOA is dat je te maken hebt met heterogene omgevingen. Services kunnen op verschillende platformen draaien, en in verschillende programmeertalen gerealiseerd zijn. Dit betekent dat je te maken krijgt met data-afkomsten van verschillende platformen. XML is de meest voor de hand liggende oplossing om deze heterogeniteit te overbruggen. De rol van SDO in dit geheel is er voor te zorgen dat in alle programmeertalen datatypen op een compatible manier worden vertaald in XML en omgekeerd. In tegenstelling tot de meeste XML-parsers zorgt SDO er ook voor dat bijgehouden wordt welke data gewijzigd zijn in een XML-document. In veel SCA-diagrammen wordt SDO dan ook getoond als communicatiemiddel tussen alle componenten in een SCA-applicatie. Door het toenemende gebruik van SOA zal de behoefte aan detached documenten waarschijnlijk ook toenemen. In een traditionele JEE-applicatie blijven persistente objecten meestal binnen de Java-container en kunnen dus via een ORM Persistency

Manager gemanaged worden. In de SOA-applicatie zul je echter vaak gebruik maken van grote XML-documenten die tussen verschillende services en composite applications verplaatst worden. Je hebt dus feitelijk continue te maken met detached objecten. In deze situatie zal het dus van toegevoegde waarde zijn dat wijzigingen in een XML-document bijgehouden worden.

Soms zinvol, soms niet

SDO biedt, zeker in combinatie met de DAS voor relationele databases, een zinvolle oplossing voor een situatie die we steeds vaker tegenkomen: XRM, oftewel, XML Relational Mapping. Daarbij moet wel gezegd worden dat de huidige deeloplossingen zoals bestaande ORM-frameworks en XML-libraries meestal completere oplossingen bieden. SDO heeft ook eigenschappen die nu niet echt goed ondersteund worden door bestaande frameworks, zoals ondersteuning voor detached documenten. Met change summaries wordt het gemakkelijker deze documenten later weer te synchroniseren met de originele bron. De behoefte hieraan zal door de groeiende toepassing van SOA waarschijnlijk toenemen. Dat betekent echter niet dat SDO automatisch een groot succes gaat worden. In tegenstelling tot SCA, dat een oplossing biedt voor een probleem waarvoor nog geen goede oplossing is, zijn er tal van alternatieven voor SDO. Er zijn talloze XML-raamwerken beschikbaar in Java en ook voor database-persistency is de keuze groot. Wil je profijt kunnen halen uit het gebruik van change summaries, dan betekent dit waarschijnlijk dat alle lagen van jouw heterogene SOA-applicatie met SDO moeten gaan werken. Het ligt niet voor de hand dat dit snel gaat gebeuren. Maar voor specifieke applicaties, zoals die beschreven in het begin van dit artikel, kan SDO wel een goede oplossing bieden.

Andrej Koelewijn