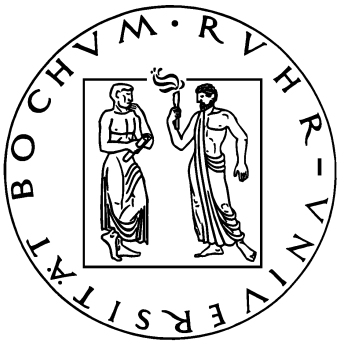
Studienarbeit v0.2

Webservice zur Synchronisation von Inventarisierungs-Tools mit verinice

Andreas Becker
<andreas.r.becker@rub.de>
Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum

20. Oktober 2009



Inhaltsverzeichnis

1 Einführung

1.1 IT-Grundschutz

Für viele Anwender von IT-Systemen ist „Sicherheit" ein Attribut des Systems, welches mit technischen Produkten und Methoden wie Firewalls und Antiviren-Software erreicht werden kann. Allerdings stellen sich, gerade in komplexen Rechnernetzen mit einer heterogenen IT-Landschaft und steigender Mitarbeiterzahl Fragen, ob eine solche technische Sicherung ausreicht.

Ein Angreifer könnte möglicherweise *von innen* unerlaubten Zugriff auf vertrauliche Daten nehmen, Rechner können im Brandfall schaden nehmen oder entwendet werden – Daher ist nicht nur die IT-Sicherheit (*Security*) im engeren Sinne zu gewährleisten, sondern auch die physikalische Sicherheit (*Safety*). Des Weiteren müssen gerade in größeren Unternehmen Zuständigkeiten genau festgelegt sein und von den Mitarbeitern eingehalten werden. Fühlt sich niemand verantwortlich dafür, dass Türen verschlossen oder der neueste Patch auf Clients und Servern eingespielt ist, bringt auch die ausgeklügelste Sicherheitsstragie nichts.

Alles in allem kommt man zu dem Schluss, dass die Sicherheit eines Rechnernetzes auf einer höheren Ebene ansetzen muss, als auf der Ebene technischer Maßnahmen. Vielmehr muss es Aufgabe des Managements sein, klare Vorgaben in Form einer Security Policy zu machen und deren Einhaltung nachhaltig zu überprüfen.

Das Bundesamt für Sicherheit in der Informationstechnik schlägt mit dem IT-Grundschutz ein standardisiertes Verfahren vor, um ein Mindestmaß an IT-Sicherheit im Unternehmen sicherzustellen. Seit der letzten Überarbeitung im Jahr 2006 orientiert sich das Grundschutz-Handbuch [4] an internationalen Standards wie ISO 27001 [5] und CobiT [6]. Unternehmen können ihre IT-Verbünde gemäß Grundschutz zertifizieren lassen; gleichzeitig ist auch eine Zertifizierung gemäß des ISO-Standards 27001 möglich, der ebenfalls vom Grundschutz abgedeckt wird.

Ein wichtiger Bestandteil des Standards sind die Grundschutzkataloge. Dabei handelt es sich um eine äußerst umfangreiche Sammlung sogenannter Bausteine, die einer personellen oder technischen Ressource (bspw. Server unter Unix) eine Reihe technischer und organisatorischer Maßnahmen zuordnen. Als Grundlage für die Zuordnung solcher Bausteine (*Modellierung*) auf Ressourcen im IT-Verbund dient die sogenannte *Strukturanalyse*. Hierbei können ggf. vorhandene Netzpläne und andere Aufzeichnungen herangezogen werden, um die IT-Landschaft des Unternehmens abzubilden, gefolgt von der *Schutzbedarfsfeststellung*.

Schon in der frühen Phase der Strukturanalyse, spätestens jedoch bei der Modellierung mit den über 4000 Seiten starken Grundschutzkatalogen stellt man fest, dass eine ma-

nuelle Durchführung, häufig mit Hilfe einiger Excel-Tabellen schnell die Grenzen des Machbaren erreicht.

Daher bietet das BSI das GSTOOL [3] als Referenzwerkzeug für den IT-Grundschutz an. Hierbei handelt es sich um ein Closed Source Projekt, das für die Plattform Microsoft Windows entwickelt wurde. Auf der anderen Seite steht mit verinice [2] ein plattformunabhängiges, auf Java und der Eclipse RCP basierendes Werkzeug unter der GNU GPLv3 zur Verfügung.

1.2 Grundschutz-Tool verinice

Mit verinice steht ein von der Göttinger SerNet GmbH entwickeltes Grundschutz-Tool zur Verfügung. Dieses Werkzeug leitet den Verantwortlichen der Unternehmensführung durch alle Phasen des Grundschutzes, angefangen bei der Strukturanalyse, über die Modellierung bis hin zur Zertifizierung durch einen unabhängigen Auditor.

Zur Modellierung des IT-Verbundes nach IT-Grundschutz werden in verinice die importierten Bausteine des Grundschutz-Katalogs[1] durch einfaches Drap & Drop einem Element des IT-Verbundes zugeordnet.

Auch in den weiteren Phasen unterstützt verinice den Anwender mit klar strukturierten Formularen, aussagekräftigen Diagrammen und einer insgesamt äußerst aufgeräumten Benutzeroberfläche. Eine Besonderheit stellt das sogenannte Hitro-UI Framework dar, das ein eingeschränktes dynamisches Objektmodell [1] implementiert. Während sämtliche Objekttypen (Mitarbeiter, Server, Client, ...) fest in verinice integriert ist, können vom Benutzer zusätzliche Attribute hinzugefügt oder nicht benötigte Attribute entfernt werden. Diese Einstellungen werden in einer einzelnen XML-Datei (SNCA.xml) vorgenommen. Änderungen in der SNCA.xml werden bei Programmstart unmittelbar im GUI nachgezogen, indem entsprechende Eingabefelder in den Formularen angezeigt werden; auch die Datenbankstruktur wird automatisch angepasst.

Ein bislang fehlendes Feature in verininice ist allerdings eine einfache Möglichkeit, Objekte im IT-Verbund entweder automatisch zu erfassen oder einen bereits vorhandenen Datenbestand aus einer externen Quelle zu importieren – Bislang muss jedes einzelne Objekt händisch in verinice eingepflegt werden, was die Strukturanalyse unter Umständen äußerst langatmig erscheinen lässt.

Allerdings ist verinice weder ein Netzwerkscanner, noch ein Werkzeug zur Inventarisierung. Zu Bestreben in diese Richtung distanzieren sich die Entwickler ganz klar. Anders sieht es mit einer Importfunktion aus oder, wie im Rahmen dieser Studienarbeit umge-

[1] Hierbei kommt eine *Web Scraping* genannte Technik zum Einsatz, bei der der aktuelle Stand der Webseiten des BSI von einem Parser interpretiert wird, der auch mit nicht ganz validen (X)HTML-Dateien umgehen kann.

setzt, einer generischen Schnittstelle zur Synchronisation von Inventardaten mit verinice.

1.3 Inventarisierung

Werkzeuge zur Inventarisierung verfolgen ein klar definiertes Ziel: Sogenannte Knoten (*nodes*) eines Rechnernetzes zu erfassen und eine ganze Reihe technischer Basisdaten, wie die Hardware-Plattform, Mainboard-Typ, verfügbarer RAM und Festplattenspeicher, Betriebssystem, usw. strukturiert abzuspeichern.

Es existiert eine ganze Reihe unterschiedlicher Tools für verschiedene Plattformen und unter verschiedenen Lizenzen. Der Trend scheint aber auch in diesem Segment zu Webapplikationen zu gehen, die auf eine serverseitige Skriptsprache und ein relationales Datenbanksystem (häufig MySQL) setzen, während auf Clientseite ein aktueller Webbrowser als Thin Client dient. In einigen Tools werden zudem Techniken wie das Simple Network Management Protocol (SNMP) oder dedizierte *Agent*-Programme eingesetzt, um die gewünschten Inventardaten und Änderungen davon automatisch zu erfassen.

All dies führt zu dem Schluss, dass Werkzeuge zur Inventarisierung, wie auch andere Datenquellen in Form von relationalen Datenbanken, Spreadsheet- oder CSV-Dateien überaus nützliche Quellen für die Strukturanalyse in verinice darstellen.

1.4 Motivation: Synchronisations-Schnittstelle für verinice

Um die Inhalte verschiedener Datenquellen in verinice verfügbar zu machen, musste zunächst eine generische Schnittstelle geschaffen werden, die nach Möglichkeit auch von Herstellern anderer Tools über ein API angesprochen werden kann, welches aus Sicht des Benutzers unveränderlich und somit robust gegenüber internen Änderungen von verinice sein sollte.

An dieser Stelle sei angemerkt, dass bereits von der Client/Server-Version von verinice ausgegangen wird. Bei der Entwicklung wurde die Entwicklerversion 0.8 aus den verinice-Quellen verwendet; das erste offizielle Release, welches auch den Applikationsserver beinhaltet, wird die Version 1.0 sein, die sich zum Zeitpunkt der Erstellung dieser Arbeit in einer späten Phase der Qualitätssicherung befindet.

Zur Remote-Kommunikation zwischen Client und Applikationsserver (der auf Apache Tomcat aufsetzt) wird der sogenannte HTTP-Invoker des Spring-Frameworks verwendet. Dabei werden serialisierte Java-Objekte, also binäre Daten, über HTTP versendet und auf der Gegenseite wieder deserialisiert. Das ist zum einen Entwickler-freundlich, da dieser Vorgang transparent stattfindet, zum anderen aber auch Firewall-freundlich, weil lediglich Port 80 zur Kommunikation benötigt wird.

Für den geplanten Synchronisationsdienst kam der Einsatz des HTTP-Invokers hingegen nicht in Frage. Die Landschaft der Inventarisierungs-Tools bzw. denkbarer Synchronisationslösungen ist beiweitem zu heterogen. Beim Einsatz des HTTP-Invokers muss man sich, entgegen aller Vorzüge, die dieser mit sich bringt, im Klaren darüber sein, dass man sich auf Java als Programmiersprache und zusätzlich auf das Spring-Framework festlegt.

Da ein generisches API mit einer exakten Schnittstellenbeschreibung geschaffen werden sollte und zudem der verinice-Server bereits auf Apache Tomcat aufsetzt, lag es nahe, einen Webservice zu entwickeln. Für diesen Dienst konnten folgende Anforderungen festgehalten werden:

- Unidirektionale Synchronisation von Inventardaten mit verinice
- Objekte mit ihren Attributen, keine Verknüpfungen
- Dienst im Kontext des verinice-Servers, der XML verarbeitet
- Authentifikation und Autorisierung des Benutzers
- Unterscheidung verschiedener Synchronisationsquellen
- Spezifikation eines offenen APIs mit XML-Schemas

Es soll eine unidirektionale Synchronisation von der Datenquelle zu verinice implementiert werden. Die Idee dahinter ist, dass bei einer geeigneten Gewaltenteilung zwischen der Datenquelle und verinice, die Datenquelle für die Bereitstellung der Basisdaten für den IT-Verbund zuständig ist, während verinice weiterhin seine Stärken als Grundschutz-Tool ausspielt. Hierzu ist für die Zukunft angedacht worden, dass „importierte" Datenfelder eventuell ausgegraut auf dem GUI dargestellt werden. Dieser Schritt wird von Seiten der SerNet jedoch vorab geprüft und ggf. in Rücksprache mit der Anwender-Community vollzogen.

Dass der Benutzer vorab zunächst geeignet authentifiziert und anschließend anhand eines Rollenkonzepts überprüft werden muss, ob dieser Benutzer überhaupt dazu berechtigt ist, Daten zu synchronisieren, sollte selbstverständlich sein. Derzeit findet in verinice bereits mit Hilfe von Spring Security eine HTTP Digest Authentification statt. In der Version 1.0 soll zudem ein vollständiges Rollenkonzept umgesetzt werden. Es wäre daher von Vorteil, die entsprechenden Mechanismen von verinice zu verwenden, um die gesamte Codekomplexität in diesem kritischen Bereich einzugrenzen.

Die Unterscheidung verschiedener Synchronisationsquellen erfolgt, indem man im einfachsten Fall bei jedem Synchronisationsvorgang einen eindeutigen Bezeichner (von nun an mit `syncId` bezeichnet) mitsendet. Die Serveranwendung kann dann alle Anfragen mit der gleichen `syncId` der gleichen Quelle zuordnen. Dabei kann der Benutzer des Dienstes frei über die Vergabepolitik solcher IDs verfügen. Dies könnte zum Beispiel dazu genutzt werden, verschiedene Instanzen eines Inventarisierungs-Tools zu unterscheiden, oder Imports aus verschiedenen Subnetzen oder Unternehmensteilen vorzunehmen.

Im Rahmen dieser Studienarbeit wurde ein Webservice entwickelt, der die gestellten Anforderungen erfüllt. Die nächsten Kapitel sollen Anwendern des Dienstes bzw. Administratoren als Handreiche dienen, die gegen das API des Webservices programmieren wollen. Zunächst wird, ausgehend von zwei wichtigen Anwendungsfällen, die Schnittstelle des Dienstes beschrieben, was durch beispielhafte XML-Dateien und eine etwas ausführlichere Fallstudie ergänzt wird.

Im dritten Abschnitt wird die Implementierung dieses APIs dokumentiert. Dieser Abschnitt richtet sich an die Entwickler von verinice, die den Webservice in das offizielle Release integrieren, warten und auf dem aktuellsten Stand halten wollen.

2 Anwender-Dokumentation

Dieser Abschnitt richtet sich an Benutzer des implementierten Webservices und somit an alle diejenigen, die unmittelbar auf das Synchronisations-API zugreifen möchten.

2.1 Verinice-Architektur

In Zukunft steht ein Webservice zur Synchronisation von Inventardaten im Kontext des verinice-Servers zur Verfügung (im Folgenden mit Sync-WS abgekürzt). Beim verinice-Server handelt es sich um einen auf Apache Tomcat basierenden Applikationsserver, der erstmals in der Version 1.0 verfügbar ist.

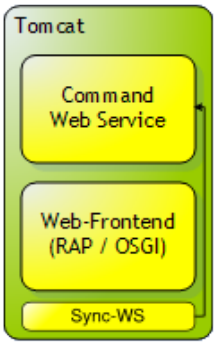


Abbildung 2.1: Sync-WS im verinice-Server, Originalbild: SerNet GmbH

Ursprünglich war verinice als Einzelplatz-Programm vorgesehen, welches standardmäßig zur Datenhaltung die integrierte Derby-Datenbank verwendet. Das bedeutet für den Benutzer einen sehr geringen Aufwand bei der Installation, die sich im Wesentlichen auf das Entpacken von verinice in ein beliebiges Verzeichnis reduziert und eine Java Virtual Machine vorraussetzt. Alternativ konnte man auch hier bereits auf ein „echtes" Datenbanksystem wie MySQL oder PostgreSQL umsatteln. Schnell stieß diese Lösung jedoch an ihre Grenzen. Bessere Skalierbarkeit, Mehrbenutzerbetrieb, eine Rollenverwaltung und eine bessere Performance waren zentrale Wünsche an die neue Version. All diese Dinge werden nun in der Version 1.0 berücksichtigt werden.

Der verinice-Server dient, wie bereits mehrfach erwähnt, als Applikationsserver, der die Geschäftslogik, sowie übergreifende Aspekte wie die Benutzerverwaltung und Sicherheit zentral zur Verfügung stellt. Dabei können mehrere Clients entfernt per HTTP(S) auf den Server zugreifen. Dazu muss der Benutzer im Menü `Einstellungen` lediglich auf den Mehrbenutzerbetrieb umstellen, die URL des Servers angeben und fortan bei jedem Start des verinice-Clients seine Credentials (Benutzername und Passwort) eingeben.

Der sogenannte Command Service muss an dieser Stelle etwas ausführlicher erläutert

werden. Intern verwendet verinice ein generisches API, welches an das Command Pattern angelehnt ist. Dieses Entwurfsmuster der Softwaretechnik legt eine sehr generische Schnittstelle fest, welches von allen Kommandoklassen implementiert wird. Damit lässt sich das Verhalten des Programms zur Laufzeit anpassen, indem Kommandoobjekte dynamisch ausgetauscht werden.

Des weiteren laufen solche Kommandos in verinice stehts transaktionssicher ab – entweder ein Kommando wird vollständig ausgeführt, oder gar nicht. Außerdem sind Kommandos auch ineinander schachtelbar. Sämtliche Fachlogik ist in verinice nun in solche Kommando-Klassen gekapselt. Des Weiteren bietet der verinice-Server entfernten Zugriff über HTTP auf seinen Command Service.

Dieser Command Service wird nun auch vom Synchronisationsdienst genutzt. D.h. jede Anfrage an den Sync-WS hat zur Folge, dass dieser, nach einer Authentifikation des Benutzers mittels HTTP Digest Authentication, eines oder mehrere Kommandos anstößt, um Objekte mit der verinice-Datenbank zu synchronisieren.

Nach diesen theoretischen Vorbemerkungen folgt nun eine Beschreibung der Synchronisations-Schnittstelle und eine Erläuterung anhand von Beispielen, wie diese Schnittstelle konkret benutzt werden kann.

2.2 Synchronisations-Webservice

In diesem Abschnitt wird zunächst die Funktionalität des Webservice auf einer etwas abstrakten Ebene erklärt. Zunächst wird die Frage geklärt werden, *was* der Webservice tut. Die Frage, *wie* man den Webservice konkret benutzt, wird hingegen in den beiden folgenden Abschnitten beantwortet.

Der implementierte Webservice dient dazu, Inventardaten mit verinice zu synchronisieren. Was mit Inventardaten gemeint ist, wurde bereits sehr ausführlich im Einführungskapitel motiviert, nämlich die „Stammdaten" eines IT-Verbundes: Clients, Server, Mitarbeiter, usw. Hier wird im Folgenden der Begriff der **Synchronisation** mit Leben gefüllt, es handelt sich nämlich nicht nur um eine Schnittstelle für einen einmaligen Import von Daten, denn es besteht zudem die Möglichkeit, auch *Änderungen* an einem vormals synchronisierten IT-Verbund mit verinice abzugleichen. Dazu werden drei grundlegende Methoden unterschieden, nämlich das Einfügen, Aktualisieren und Löschen von Objekten.

2.2.1 Einfügen neuer Objekte (INSERT)

Mit der INSERT-Methode können Objekte in verinice neu angelegt werden, die bislang noch nicht in der verinice-Datenbank enthalten sind:

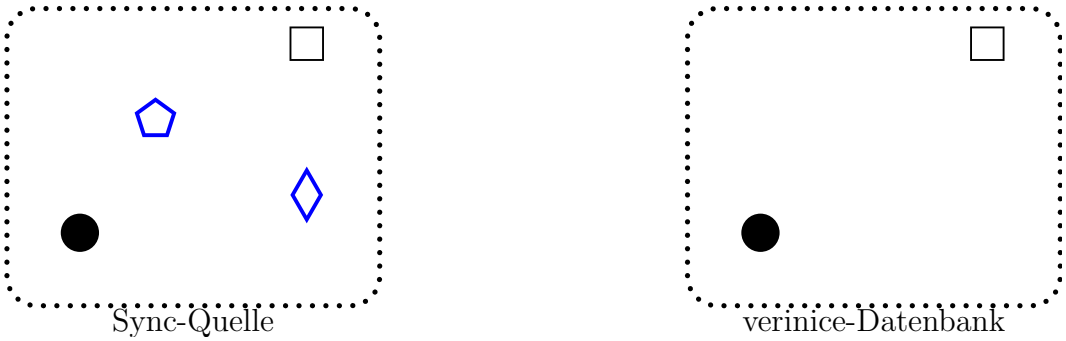


Abbildung 2.2: Datenbestand *vor* dem Einfügen

Die beiden blau markierten Objekte sind offenbar als Einzige noch nicht in der verinice-Datenbank enthalten. Bei einem INSERT aus dieser Quelle werden die Objekte daher neu angelegt:

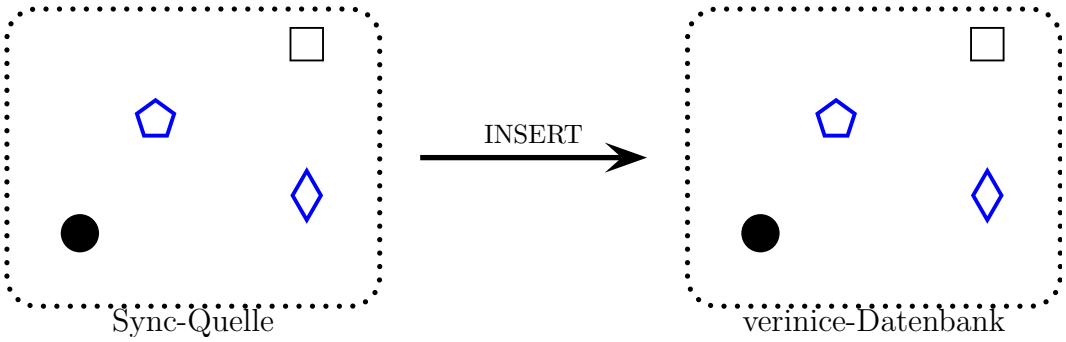


Abbildung 2.3: Datenbestand *nach* dem Einfügen

2.2.2 Aktualisieren vorhandener Objekte (UPDATE)

Mit der UPDATE-Methode können bereits in der verinice-Datenbank enthaltene Objekte aktualisiert werden; das heißt, dass sämtliche Attribute aller bereits vorhandenen Objekte mit den neuen Werten aus der Quelle überschrieben werden:

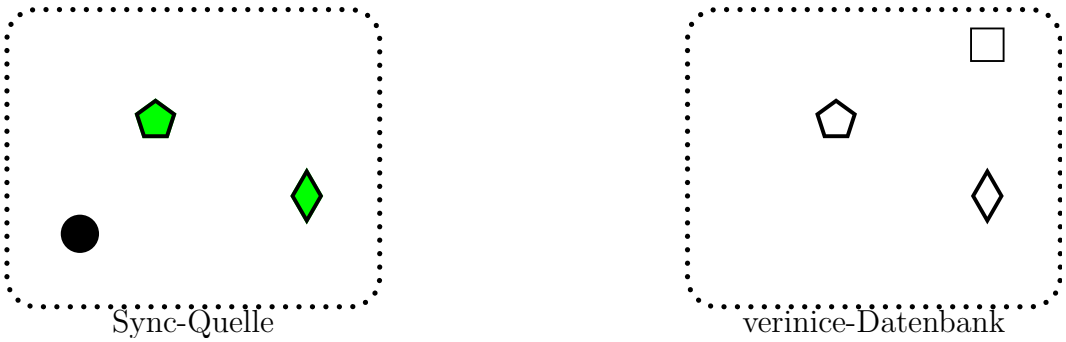


Abbildung 2.4: Datenbestand *vor* dem Aktualisieren

Bei einem UPDATE werden die geänderten Attribute aller Objekte, die bereits in verinice vorhanden sind, mit den neuen Werten aus der Quelle überschrieben. Beides trifft lediglich auf die beiden grün ausgefüllten Objekte zu. Nicht betroffen sind Objekte, die ausschließlich in der Sync-Quelle *oder* (gemeint ist hier ein exklusives oder) in verinice enthalten sind.

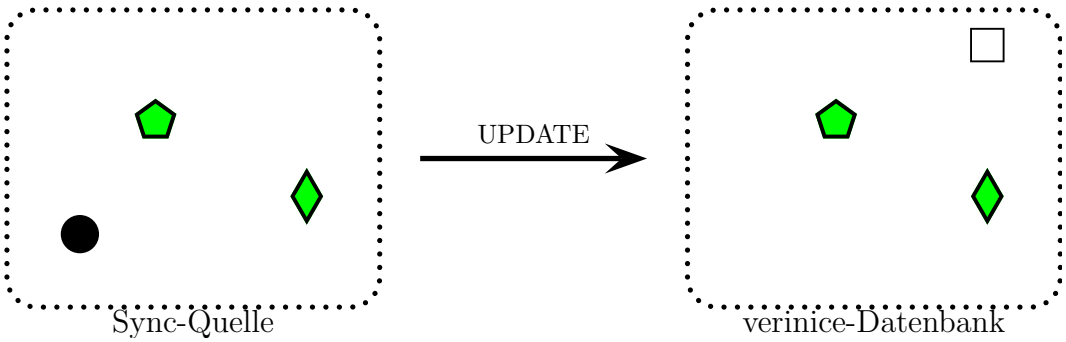


Abbildung 2.5: Datenbestand *nach* dem Aktualisieren

2.2.3 Entfernen veralteter Objekte (DELETE)

Mit der DELETE-Methode können alle Objekte, die in der verinice-Datenbank vorhanden sind, sich aber nicht (mehr) in der Quelle befinden, gelöscht werden:

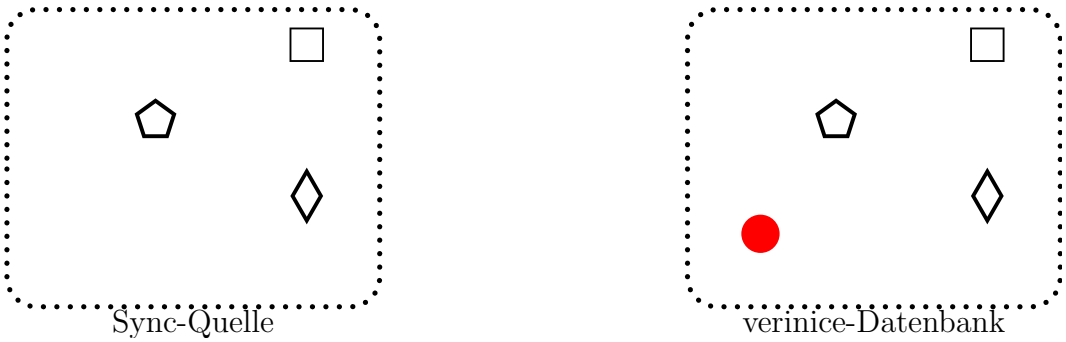


Abbildung 2.6: Datenbestand *vor* dem Löschen

Bei einem DELETE wird nun das einzige Objekt (rot markiert), das zwar in der verinice-Datenbank enthalten ist, nicht aber in der Quelle, aus der verinice-Datenbank gelöscht:

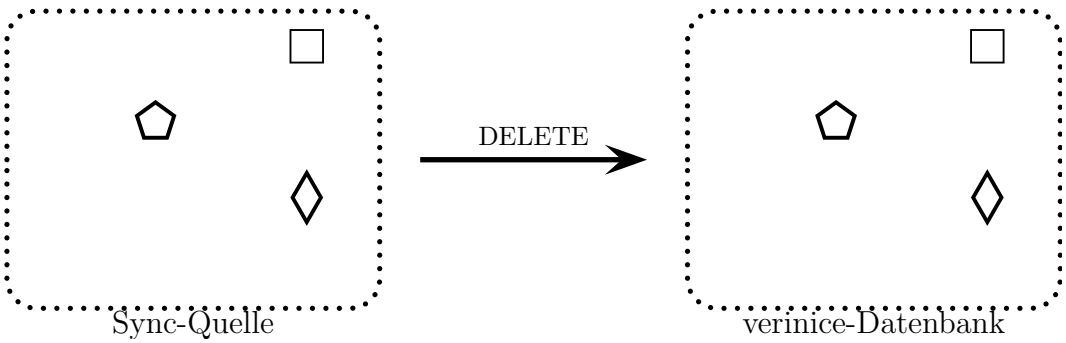


Abbildung 2.7: Datenbestand *nach* dem Löschen

Alle drei Methoden werden vom Webservice bereits unterstützt. Dabei müssen jedoch nicht separate Anfragen an den Webservice gerichtet werden. Vielmehr ist bei jeder Anfrage je Methode ein Boolsches Flag (true oder false) anzugeben, um dem Webservice anzuzeigen, ob Objekte eingefügt, aktualisiert, und/oder gelöscht werden sollen.

Der Webservice ist über SOAP ansprechbar und verarbeitet XML-Dateien. Daher werden die Flags als Attribute eines `<syncRequest>`-Elements angegeben. Wie solche Anfragen auszusehen haben, wird ausführlich im Abschnitt *API-Dokumentation und Datenkontrakt* beschrieben. Für den Moment soll uns ein grundlegendes Verständnis für die drei o.g. Methoden ausreichen.

Vorab ist noch die Frage zu klären, wie Objekttypen mit ihren Attributen aus einer Datenquelle auf die in verinice vorhandenen Objekttypen abgebildet werden.

2.2.4 Abbildung der Objekttypen mit ihren Attributen

Eine grundlegende Frage, um den Webservice zur Synchronisation verwenden zu können, ist noch zu klären: Objekttypen wie Clients, Server oder Mitarbeiter sind in verinice fest vorgegeben und tragen interne Bezeichner („IDs") wie *client*, *server* oder *person*. Wie löst man nun das Problem, dass auf der Seite der Synchronisations-Quelle ein anderes Vokabular benutzt wird, wie bspw. *Node* (häufig keine Trennung zwischen Servern und Clients), *Person*, usw.?

Zunächst ist festzustellen, dass es dafür kein Patentrezept gibt, weil das verwendete Vokabular z.B. für Tabellenbezeichnungen im Falle eines relationalen Datenbanksystems zu unterschiedlich ausfällt. Analog gilt das Gleiche für die jeweiligen Attribute eines Objekttyps bzw. die Spaltenbezeichnungen einer Tabelle.

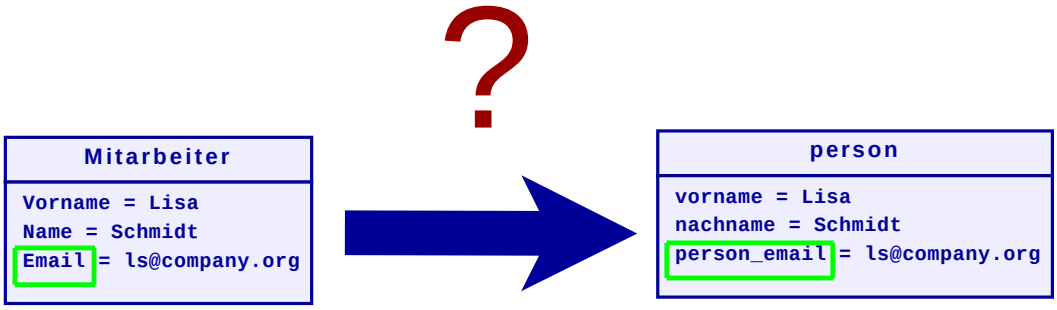


Abbildung 2.8: Abbildung der Objekttypen und ihrer Attribute

Ein Ziel des Webservices war es jedoch, eine generische Schnittstelle zur Verfügung zu stellen, die möglichst flexibel zu benutzen ist. Daher war es naheliegend, die Abbildung der Objekttypen mit ihren Attributen, ebenso wie die Inventardaten selbst, in XML zu formulieren.

Da die Abbildungsvorschrift natürlich für jede Quelle unterschiedlich aussieht, müssen diese sogenannten **Mappingdaten** individuell für jede Quelle verfasst werden, die einen Export ihrer Daten im XML-Format bereitstellt. Dazu muss vom Benutzer des Webservices in einer SOAP-Anfrage unterhalb des `<syncRequest>`-Elements zusätzlich zum `<syncData>`-Element ein `<syncMapping>`-Element mitgesendet werden, welches die konkrete Abbildungsvorschrift enthält.

Hier sei noch einmal darauf hingewiesen, dass die exakte Schnittstellenbeschreibung mit dem Datenformat einer SOAP-Nachricht im Abschnitt *API-Dokumentation und Datenkontrakt* erfolgt. Vorher wollen wir die bisher gesammelten Puzzlestücke zu unserem Sync-WS zusammensetzen und zwei wichtige Anwendungsfälle beschreiben, von denen der Grundlegendere bereits vollständig implementiert ist.

2.3 Anwendungsfälle

Im vorhergehenden Abschnitt haben wir bereits alle wesentlichen Punkte zusammengetragen, die zur Nutzung des Webservices erforderlich sind. Inventardaten und Mappingdaten müssen gemeinsam in ein `<syncRequest>` verpackt an den Webservice geschickt werden. Der naheliegenste Anwendungsfall besteht darin, dass die Mappingdaten vorab z.B. vom Hersteller eines Inventarisierungs-Tools, zusammen mit einer Exportschnittstelle, die XML-Daten liefert, bereitgestellt werden. Dies führt unmittelbar zum ersten Anwendungsfall.

2.3.1 Synchronisation – Manuell erzeugte Mappingdaten

Vorbedingung für diesen Anwendungsfall ist, dass bspw. der Hersteller eines Inventarisierungs-Tools sowohl Mappingdaten, als auch die Exportschnittstelle, welche die XML-Daten exportiert, zur Verfügung stellt. In diesem Fall bedient sich der Benutzer der angebotenen Exportschnittstelle und sendet Inventardaten und Mappingdaten in einem SOAP-Request an den Webservice:

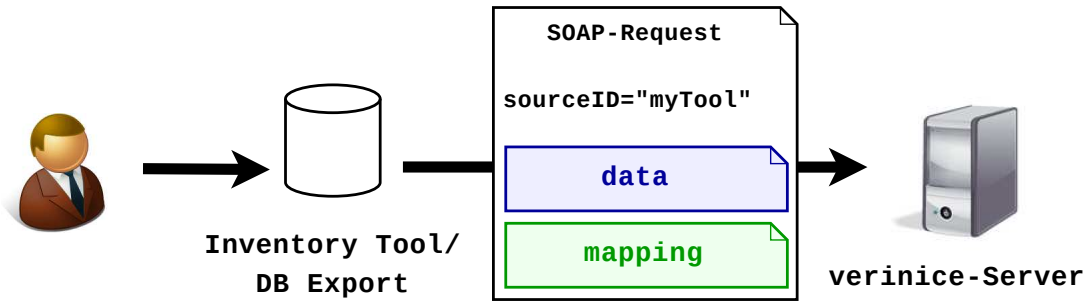


Abbildung 2.9: Synchronisation, manuell erzeugte Mappingdaten

Der SOAP-Request selber ist noch mit einer frei gewählten `sourceId` zu versehen. Für weitere Synchronisationsvorgänge aus der selben Quelle sollte allerdings die gleiche `syncId` wieder verwendet werden. In diesem Fall ist für jede Quelle im verinice-Datenbestand genau ein IT-Verbund vorgesehen.

Neu angelegte Objekte landen zunächst in diesem IT-Verbund mit dem Namen *sourceId*-Verbund. Da synchronisierte Objekte aber jederzeit eindeutig über das Tupel (sourceId, extId) identifiziert werden können, ist es auch möglich, diese Objekte aus dem automatisch angelegten IT-Verbund heraus zu verschieben. Diese stehen dem Webservice trotzdem für weitere Synchronisationsvorgänge zur Verfügung.

Wollte man diesen Anwendungsfall als einzige Möglichkeit anbieten, den Webservice zu benutzen, hinterließe dies sicherlich einen bitteren Nachgeschmack. Schließlich müssen auf Seite des Benutzers eine XML Export-Schnittstelle gemäß einem vorgegebenen Schema geschrieben werden, sowie eine XML-Datei mit den Mappingdaten. Zudem muss ein Webservice über SOAP angesprochen werden. Will man sich das händische Jonglieren mit XML-Dateien ersparen, müsste für den Anstoß des gesamten Prozesses optimalerweise noch ein GUI geschrieben werden, welches die eigene Exportfunktion anstößt, resultierende Inventardaten und die Mappingdaten zu einem SOAP-Request zusammenfasst und an den Webservice schickt.

Das ist so natürlich völlig unzureichend. Es mag durchaus Szenarien geben, in denen sowieso bereits eine eigene Exportfunktion in XML vorhanden und ein Modul, welches den gesamten Prozess (vielleicht sogar per Cron-Job automatisch) anstößt schnell geschrieben ist. Im Allgemeinen möchte man den Benutzer, aber davor bewahren, sich allzu tiefgreifend mit Dingen wie XML-Schemata und Programmierung von Export-Schnittstellen auseinandersetzen zu müssen.

Daher ist der im Vorfeld genannte „Anwendungsfall" auch eher als Kernfunktionalität zu verstehen, auf dem weitere komfortablere Funktionen aufsetzen, wie z.B. der zweite, noch nicht implementierte Anwendungsfall, der in einem kommenden Release von verinice integriert werden soll.

2.3.2 Synchronisation – Interaktion mit einem Wizard

Die Integration dieses Anwendungsfalls ist erst für ein kommendes Release von verinice angedacht. Um die händische Erstellung einer XML-Datei mit Mappingdaten zu umgehen, soll dabei ein Wizard in den verinice-Client integriert werden, der mit dem Benutzer interagiert und im Hintergrund die Mappingdaten erstellt.

Da heutzutage jedes Datenbanksystem und jede Spreadsheet-Software dazu in der Lage ist, Daten im CSV-Format (**C**omma **S**eparated **V**alue) zu exportieren, wäre es sicherlich sinnvoll, auch dieses Format als Datenquelle verfügbar zu machen. Angedacht ist daher, dass solche CSV-Dateien mit Hilfe des o.g. Wizards synchronisiert werden können. Einzige Vorbedingung dafür ist, dass eine CSV-Datei vorliegt, die entweder nur Inventardaten *eines* Objekttyps enthält (bspw. ausschließlich Clients) oder Inventardaten mehrerer Objekttypen enthält, wobei eine der Spalten als sogenannter Diskriminator dient, d.h. angibt, von welchem Typ ein Objekt ist.

Alle Objekte desselben Typs müssen natürlich von der gleichen Form sein, d.h. die Ausprägungen aller Spalten von Objekten des gleichen Typs müssen auch die gleiche Semantik haben.

Im ersten Schritt exportiert der Benutzer also eine Spreadsheet-Datei in CSV bzw. stößt

den Export mit der CSV-Schnittstelle einer Datenbank an:

Abbildung 2.10: Export der Inventardaten im CSV-Format

Anschließend öffnet der Benutzer seinen verinice-Client, der im Mehrbenutzermodus konfiguriert ist. Hier startet er den Synchronisations-Wizard und gibt die soeben exportierte CSV-Datei an. Im weiteren Verlauf wird interaktiv mit Hilfe des Benutzers die Semantik jeder Spalte ermittelt, wobei ggf., wie oben erläutert, eine Diskriminatorspalte angegeben werden kann.

Abbildung 2.11: Interaktion mit dem Synchronisations-Wizard

Der Wizard generiert im Hintergrund nun die Mappingdaten und transformiert die In-

ventardaten, die im CSV-Format vorliegen, in XML. Beides führt er zu einem validen SOAP-Request mit einer vom Benutzer angegebenen sourceId zusammen und schickt diese Anfrage an den verinice-Server. Dort wird die Anfrage wie gewohnt vom Webservice verarbeitet.

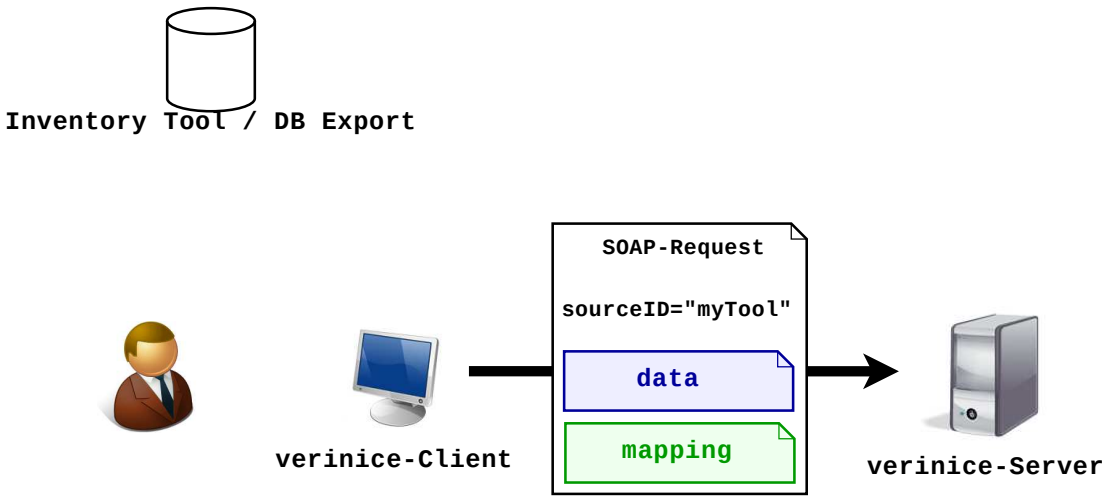


Abbildung 2.12: Wizard sendet den generierten SOAP-Request an den Webservice

Für weitere Synchronisationsvorgänge mit dem Wizard sollen auch die Wizard-Einstellungen (`sourceId`, Semantik der Spalten, Diskriminatorspalte, usw.) gespeichert werden können, beispielsweise um diese wiederverwenden zu können. Theoretisch könnte man sich mit einem solchen einmalig erstellten Satz an Wizard-Einstellungen auch den Weg über das GUI sparen und den Vorgang wiederum über die Konsole (bspw. mit Hilfe eines Cron-Jobs auch automatisch) anstoßen.

2.4 API-Dokumentation und Datenkontrakt

Nun wird die Frage geklärt, wie der entwickelte Webservice konkret zu benutzen ist. Es wird das Nachrichtenformat für die SOAP-Anfragen festgelegt und anhand von Beispielen erläutert, wie die benötigten XML-Elemente im Einzelnen auszusehen haben.

Der Webservice steht im Kontext des verinice-Servers zur Verfügung, d.h. zunächst benötigt man eine Instanz des Servers. Wir nehmen im Folgenden an, dass die verinice-Serveranwendung unter

`http://example.org/veriniceserver`

erreichbar ist. Diese URL wollen wir im Folgenden auch `APP_ROOT` nennen. Der Webser-

vice ist dann erreichbar unter `APP_ROOT/sync/syncService`. In unserem Beispiel also unter der Adresse

`http://example.org/veriniceserver/sync/syncService`

Die WSDL-Datei, die die Funktionsweise des Webservices beschreibt, ist unter `APP_ROOT/sync/sync.wsdl` zu finden. In unserem Beispiel also unter der Adresse

`http://example.org/veriniceserver/sync/sync.wsdl`

Eigentlich wäre an dieser Stelle schon alles nötige gesagt, um den Webservice bedienen zu können, da die WSDL-Datei alle nötigen Informationen, wie die Grammatik der SOAP-Nachrichten, URL des Endpunkts, usw. enthält. Im Folgenden geben wir dennoch einen intuitiveren Leitfaden zu Bedienung an.

2.4.1 Struktur des SOAP-Requests

Eine SOAP-Nachricht für unseren Webservice muss grob wiefolgt aussehen:

```
<soapenv:Envelope xmlns:soapenv="{SOAP}">
    <soapenv:Header/>
    <soapenv:Body>
        <syncRequest xmlns="{SYNC}"
            sourceId="mySource"
            insert="true"
            update="true"
            delete="false">

            <syncData xmlns="{DATA}">
                [<syncObject/>]*
            </syncData>

            <syncMapping xmlns="{MAPPING}">
                [<mapObjectType/>]*
            </syncMapping>

        </syncRequest>
    </soapenv:Body>
</soapenv:Envelope>
```

Zur übersichtlicheren Darstellung wurden dabei Kürzel der Form `{Kürzelname}` anstelle vollständiger Namespace-URIs verwendet. Die vollständigen URIs sind dem Anhang A zu entnehmen.

Der eigentliche Inhalt unseres Requests befindet sich im Body eines SOAP-Envelopes. Hier befindet sich exakt ein `<syncRequest>`-Element, das als obligatorisches Attribut die `sourceId` trägt, also einen eindeutigen Bezeichner der Synchronisationsquelle. Die booleschen Flags `insert`, `update` und `delete` geben an, welche der gleichnamigen Synchronisationsmethoden (vgl. Abschnitt 2.2) der Webservice anwenden soll. Diese Flags müssen nicht explizit angegeben werden; die Standardwerte sind `true`, `true` und `false`, um ein versehentliches Löschen von Objekten zu vermeiden.

Als nächstes schauen wir uns an, wie die `<syncObject>`-Elemente unter `<syncData>` aufgebaut sind. Diese Elemente repräsentieren die einzelnen Objekte der Inventardaten.

2.4.2 Struktur des `<syncObject>`-Elements

Mit einem `<syncObject>`-Element beschreibt man ein Inventarobjekt, z.B. einen Client, Server oder Mitarbeiter. Die Struktur dieses Elements ist sehr einfach. Es besitzt zwei obligatorische Attribute `extId` und `extObjectType` vom Typ einer Zeichenkette:

```
1 <syncObject extId="Server4711" extObjectType="Server">
```

Mit dem `extId`-Attribut gibt man die externe ID des Objektes an, also die ID des Objektes innerhalb der Datenquelle. Dabei ist zu beachten, dass die externe ID das Objekt applikationsweit eindeutig identifizieren muss. Überträgt man diese Forderung auf ein relationales Datenbanksystem, findet man häufig eine Spalte `ID` in einer Tabelle vor. Nun ist die ID 4711 aber nicht applikationsweit eindeutig, weil die gleiche ID offenbar in verschiedenen Tabellen für völlig unterschiedliche Tupel verwendet werden kann. Daher reicht es i.d.R. nicht, den Inhalt der Spalte ID als `extId` zu verwenden. Eine sehr einfache Lösung dieses Problems könnte darin bestehen, den Tabellennamen als Präfix zu verwenden, was die ID wieder eindeutig macht. So hätte man bspw. Objekte mit den IDs „Client4711” oder „Server4711”.

Mit dem `extObjectType`-Attribut gibt man die Bezeichnung des Objekttypen innerhalb der Datenquelle an. Wiederum übertragen auf eine relationale Datenbank könnte dies den Tabellennamen entsprechen. Dieser Ansatz funktioniert natürlich nicht bei komplexeren Joins, bei denen keine 1:1-Zuordnung zwischen einer einzigen Tabelle und einem Objekttypen besteht.

Als Kindelemente von `<syncObject>` können nun beliebig viele `<syncAttribute>`-Elemente angegeben werden. Hierbei handelt es sich um einfache Name/Wert-Paare mit den Attributen `name` und `value`. Die Namensattribute müssen lediglich innerhalb ihres umgebenden Objekts eindeutig sein.

Ein einfaches Objekt mit einigen Attributen könnte beispielsweise wie folgt aussehen:

```
<syncObject extId="Person4711" extObjectType="Person">
    <syncAttribute name="Name" value="Mustermann"/>
    <syncAttribute name="Vorname" value="Hans-Hubertus"/>
    <syncAttribute name="Email" value="hhm@example.org"/>
</syncObject>
```

Damit der Webservice nun weiß, wie er dieses Objekt vom Typ Person auf das verinice-Datenmodell abbilden soll, wird zudem ein entsprechendes <mapObjectType>-Element benötigt, dessen Struktur im Folgenden beschrieben wird.

2.4.3 Struktur des <mapObjectType>-Elements

Ein <mapObjectType> beschreibt, wie sämtliche Objekte eines bestimmten Typs, wie bspw. alle Personen mit ihren Attributen auf Verinice-Objekttypen abgebildet werden. Auch dieses Element ist wieder sehr einfach gehalten. Es besitzt zwei Attribute `extId` und `intId`:

```
<mapObjectType extId="Server" intId="server">
```

Die beiden Attribute stehen für genau das, was ihre Bezeichnungen (hoffentlich) bereits suggerieren – nämlich die IDs der Objekttypen innerhalb der Quelle (`extId`) bzw. innerhalb von verinice (`intId`), eine simple 1:1-Abbildung. Verinnerlichen sollte man an dieser Stelle, dass mit „ID" nun kein eindeutiger Bezeichner eines einzelnen Objektes gemeint ist, sondern die Bezeichnung eines Objekttypen oder einer Klasse.

Als Kindelemente von <mapObjectType> können beliebig viele <mapAttributeType>-Elemente spezifiziert werden. Auch diese Elemente verfügen wieder über eine externe und eine interne ID und beschreiben, wie die jeweiligen Attribute ihres Vaterelements abgebildet werden.

Um unser obiges Beispiel mit dem Objekttypen „Person" wieder aufzugreifen - die entsprechenden Mappingdaten sähen wiefolgt aus:

```
<mapObjectType extId="Person" intId="person">
    <mapAttributeType extId="Name" intId="nachname"/>
    <mapAttributeType extId="Vorname" intId="vorname"/>
    <mapAttributeType extId="Email" intId="person_email"/>
</mapObjectType>
```

So ein <mapObjectType>-Element legt man nun idealerweise für jeden Datentypen in der Synchronisationsquelle an. Alle in verinice definierten Objekttypen mit ihren Attributen

können in der SNCA.xml nachgeschlagen werden. Dies sind *raum*, *server*, *tkkomponente*, *netzkomponente*, *itverbund*, *sonstit*, *person* und *gebaeude*, wobei der gesamte IT-Verbund nicht explizit synchronisiert werden muss – Alle synchronisierten Objekte aus einer Quelle werden immer unterhalb eines automatisch erzeugten IT-Verbundes angelegt.

Die SNCA.xml kann außerdem für eigene Anpassungen verwendet werden, um etwa Attribute hinzuzufügen oder zu entfernen. Damit sind durchaus flexible Lösungen möglich, um beispielsweise auch eine Reihe technischer Attribute von Clients und Servern, wie die verbaute Hardware, Betriebssystem, usw. in verinice zu erfassen.

An dieser Stelle sei noch erwähnt, dass die exakten Grammatiken der Inventar- und Mappingdaten in den Schemadateien `data.xsd` bzw. `mapping.xsd` nachzuschlagen sind. Sobald der Webservice in verinice integriert ist, sollten diese im `WebContent`-Unterverzeichnis des verinice-Servers zu finden sein.

2.4.4 Struktur der SOAP-Response

Haben wir uns die Mühe gemacht, eine gültige Anfragenachricht an den Webservice zu schicken, lassen wir den Webservice zunächst seine Arbeit tun und bekommen anschließend die Quittung in Form einer SOAP-Response. Diese Antwortnachricht hat den folgenden Aufbau:

```
<soapenv:Envelope xmlns:soapenv="{SOAP}">
    <soapenv:Header/>
    <soapenv:Body>
        <syncResponse xmlns="{SYNC}">
            <replyMessage>Synchronisation erfolgreich</
               replyMessage>
            <inserted>5</inserted>
            <updated>5</updated>
            <deleted>10</deleted
        </syncResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

In diesem Fall war die Synchronisation also erfolgreich, wie uns in der Textnachricht mitgeteilt wird. Außerdem wird uns in aller Kürze mitgeteilt, wieviele Objekte jeweils eingefügt, aktualisiert bzw. gelöscht wurden, als Textknoten unterhalb der Elemente `<inserted>`, `<updated>` bzw. `<deleted>`. Die verwendeten Namespace-Kürzel sind wieder dem Anhang A zu entnehmen.

Sollte wider erwarten mal etwas schief gehen, was zum Abbruch des Synchronisationsvorgangs führt, wird dies ebenfalls im `<replyMessage>`-Element mitgeteilt. Ursachen können z.B. interne Serverfehler bei der Verarbeitung oder eine ungültige Anfragenachricht sein.

An dieser Stelle sei erwähnt, dass der Webservice relativ gutmütig bei der Verarbeitung der Synchronisationsdaten bzw. speziell beim Mapping vorgeht. Stößt er auf ein Objekt oder ein Attribut, für das keine gültigen Mappingdaten gefunden werden können, verwirft er dieses Objekt oder Attribut einfach und arbeitet weiter, sodass **nicht** der gesamte Vorgang abgebrochen wird. Dennoch wird der Fehler natürlich protokolliert und in der Antwortnachricht angezeigt.

2.5 Fallstudie „Mustermann Neue Medien GmbH"

Nachdem wir nun im Detail erläutert haben, wie die Nachrichtenelemente aufgebaut sein müssen und wie der Webservice zu erreichen ist, soll zum Abschluss ein ausführlicheres Beispiel erläutert werden. Dafür werden nacheinander drei Request-Nachrichten an den Webservice geschickt.

Um den Webservice ohne einen evtl. vorhandenen Wizard oder eine grafische Oberfläche verwenden zu können, kann man sich entweder einer Skriptsprache oder einer höheren Programmiersprache bedienen, oder aber man benutzt soapUI [7]. Dabei handelt es sich um ein äußerst flexibles Open Source-Werkzeug unter der LGPL zum Testen von Webservices. Da soapUI in Java entwickelt wurde, ist es ebenfalls auf allen Plattformen lauffähig, für die eine aktuelle JVM existiert.

Um den Webservice auszuprobieren legt man in soapUI unter `File` → `New soapUI Project` ein neues Projekt an. Hier kann man die WSDL-Datei des Webservices angeben und sich direkt leere Beispiel-Requests generieren lassen. Alternativ kann man auch die beiliegenden Requestdateien request[01-03].xml importieren. Per Klick auf die Schaltfläche mit dem grünen Pfeil kann ein Request abgeschickt werden.

Noch ein Wort zur Authentifikation: Nur gültige verinice-Benutzer, die dem Server bekannt sind, dürfen eine Synchronisation durchführen. Die Authentifikation erfolgt mittels Benutzernamen und Passwort via HTTP Digest Authentication. Dazu müssen die Credentials in soapUI im Fenster `Request Properties` vorab eingetragen werden.

Selbstverständlich besteht auch die Möglichkeit, die Dienste des verinice-Applikationsservers per SSL abzusichern. Da SOAP über HTTP abgewickelt wird, entstehen clientseitig keine weiteren Maßnahmen als der aufmerksame Umgang mit Serverzertifikaten. Auch soapUI bietet die entsprechenden Möglichkeiten, Zertifikatsketten zu inspizieren.

An dieser Stelle freuen wir uns, den Leser mit der **Mustermann Neue Medien GmbH** bekannt machen zu dürfen, ein fiktives Unternehmen, das sich auf Dienstleistungen rund um das Internet und Web 2.0 spezialisiert hat. Die Mustermann GmbH hat ihre Inventarisierung bislang mit einem einfachen selbst geschriebenen Web Frontend und einem relationalen Datenbanksystem durchgeführt, dessen Tabellen wir uns zunächst ansehen wollen:

Person				
id	Name	Vorname	E-Mail	Raum_id
1	Mayer	Christian	cmayer@mustermann-gmbh.org	1
2	Schmidt	Lisa	lschmidt@mustermann-gmbh.org	1
3	Schulze	Wolfgang	wschulze@mustermann-gmbh.org	2
4	Müller	Stefan	smueller@mustermann-gmbh.org	2
5	Klein	Julia	jklein@mustermann-gmbh.org	2
6	Bauer	Hans	hbauer@mustermann-gmbh.org	3

PC				
id	Hostname	Ipv4Addr	Admin_id	Raum_id
1	pc001	10.0.0.1	6	1
2	pc002	10.0.0.2	6	1
3	pc003	10.0.0.3	6	2
4	pc004	10.0.0.4	6	2
5	pc005	10.0.0.5	6	2
6	pc006	10.0.0.6	6	3

Server					
id	Hostname	Ipv4Addr	Betriebssystem	Admin_id	Raum_id
1	srv001	10.0.10.1	Debian GNU/Linux	6	3
2	srv002	10.0.10.2	Debian GNU/Linux	6	3
3	srv003	10.0.10.3	Windows Server 2008	6	3

Raum		
id	Bezeichnung	Gebaeude_id
1	Büroraum klein	1
2	Büroraum groß	1
3	Serverraum	1

Gebaeude	
id	Bezeichnung
1	Firmengebäude

Person_nutzt_PC	
Person_id	*PC_id*
1	1
2	2
3	3
4	4
5	5
6	6

Anm.: Primärschlüssel erscheinen in Kursivschrift.

Es sind also nur vergleichsweise rudimentäre Attribute der einzelnen Objekttypen vorhanden. Es gibt nur ein Gebäude mit drei Räumen und einigen Mitarbeitern. Während es vorgesehen ist, dass mehrere Personen einen Rechner regelmäßig *nutzen* können, besteht eine 1:n-Relation zwischen PC bzw. Server und Administrator. In unserem Beispiel betreut allerdings sowieso ein einzelner Mitarbeiter sämtliche Rechner.

Zu den Relationen sei generell gesagt, dass diese bislang noch nicht vom Webservice verarbeitet werden, was hoffentlich bereits bei der Besprechung der Schnittstelle klar geworden ist. Uns interessieren hier also lediglich die Objekte mit ihren Attributen.

Herr Müller von der Firma Mustermann wurde damit beauftragt, eine Exportfunktion der Daten gemäß der Schemadatei `data.xsd`, sowie die benötigten Mappingdaten gemäß `mapping.xsd` zu schreiben. Dazu hat er zunächst aufmerksam die Abschnitte 2.3 und 2.4 studiert, dem Autor per E-Mail konstruktive Kritik zukommen gelassen und sich ans Werk gemacht.

Er hat den Rat befolgt und erzeugt applikationsweit eindeutige IDs aus seiner Tabellenstruktur, indem er den Tabellennamen der jeweiligen ID voranstellt (Tipp: Konkatenations-Operator `||` von SQL oder `CONCAT()` nutzen, je nach verwendetem Datenbanksystem!).

Wir werden uns nun das Einfügen dieser Objekte, sowie zwei weitere inkrementelle Änderungen am IT-Verbund mit Hilfe des Webservices ansehen. Da wir die Tabellenstruktur nun einmal gesehen haben, beschränken wir uns auf eine etwas intuitivere grafische Übersicht über den sich ändernden IT-Verbund. Außerdem interessieren uns natürlich die (Auszüge aus den) XML-Dateien mit den Anfragen an den Webservice und die Informationen, was der Webservice damit macht.

2.5.1 Erste Synchronisation mit verinice – Importieren des IT-Verbundes

Bei der initialen Synchronisation mit verinice handelt es sich noch um einen reinen „Import", da lediglich neue Objekte eingefügt werden müssen, aber keine Objekte zu aktualisieren oder gar zu löschen sind. Dabei wird auch automatisch ein neuer IT-Verbund in verinice erzeugt, in dem die neuen Objekte angelegt werden.

Herr Müller ist inzwischen bei einer stabilen Betaversion seines Exportskriptes angelangt und auch die Erstellung der Mappingdaten ging ihm zügig von der Hand, da nur fünf verschiedene Objekttypen in der Inventardatenbank der Mustermann GmbH vorhanden sind, nämlich *Person*, *PC*, *Server*, *Raum* und *Gebäude*.

Die korrespondierenden Objekttypen und Attribute des Hitro-UI Frameworks hat Herr Müller dazu in der `SNCA.xml` nachgeschlagen.

Dies sind die Mappingdaten für die Inventardatenbank der Mustermann GmbH:

```
<syncMapping xmlns="http://www.sernet.de/sync/mapping">

    <mapObjectType extId="Raum" intId="raum">
        <mapAttributeType extId="Bezeichnung" intId="raum_name"/>
    </mapObjectType>
    <mapObjectType extId="Server" intId="server">
        <mapAttributeType extId="Hostname" intId="server_name"/>
        <mapAttributeType extId="IPv4Addr" intId="server_netadr"/>
        <mapAttributeType extId="Betriebssystem" intId="
            server_plattform"/>
    </mapObjectType>
    <mapObjectType extId="PC" intId="client">
        <mapAttributeType extId="Hostname" intId="client_name"/>
        <mapAttributeType extId="IPv4Addr" intId="client_netadr"/>
    </mapObjectType>
    <mapObjectType extId="Person" intId="person">
        <mapAttributeType extId="Name" intId="nachname"/>
        <mapAttributeType extId="Vorname" intId="vorname"/>
        <mapAttributeType extId="Email" intId="person_email"/>
    </mapObjectType>
    <mapObjectType extId="Gebaeude" intId="gebaeude">
        <mapAttributeType extId="Bezeichnung" intId="gebaeude_name
            "/>
    </mapObjectType>

</syncMapping>
```

Die Mappingdaten werden von seiner Exportfunktion bei der Erzeugung des SOAP-

Requests automatisch eingefügt. Da sich diese Daten von nun an nicht mehr ändern, wollen wir sie aus den folgenden Code-Beispielen ausschließen. Wann immer ein „eingeklapptes" `<syncMapping/>` auftaucht, ist dieses Element durch den obigen XML-Schnipsel zu ersetzen.

Zur Visualisierung, wie der IT-Verbund der Mustermann GmbH für die erste Synchronisation aussieht, hier eine grafische Übersicht, ähnlich einem groben Netzwerkplan, der um die Mitarbeiter und wichtige Attribute ergänzt wurde:

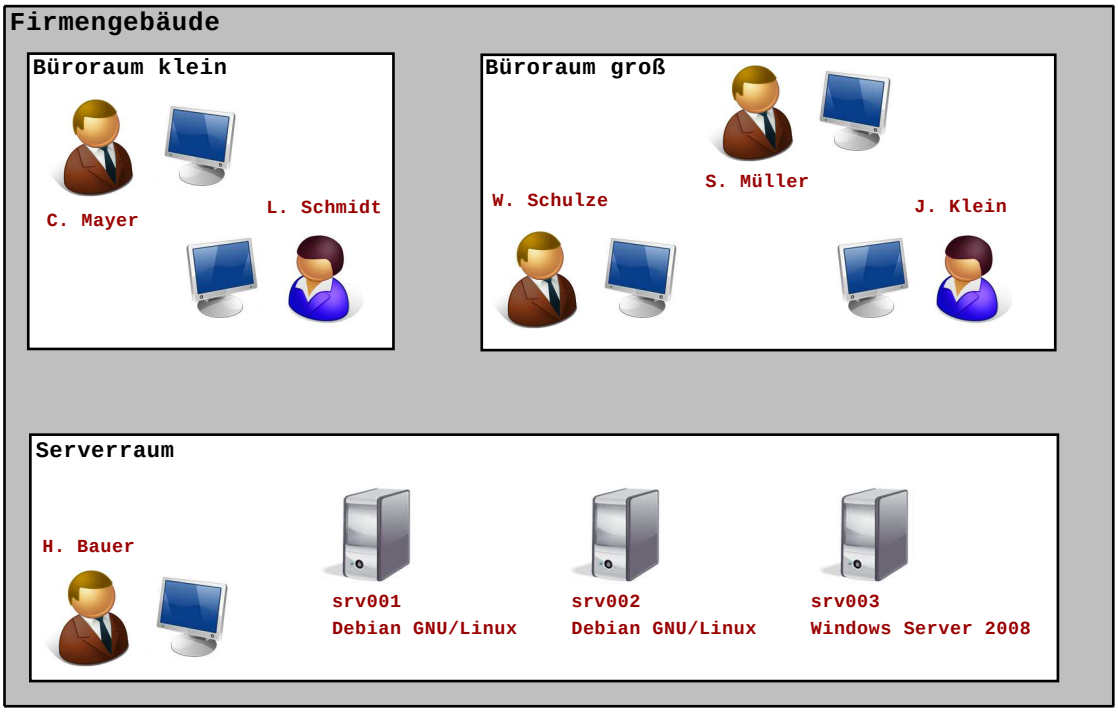


Abbildung 2.13: Schnappschuss des IT-Verbundes vor der ersten Synchronisation

Nun wird es spannend! Herr Müller hat die Mappingdaten in sein Exportskript integriert und stößt über das GUI die erste Synchronisation an. Dazu hat er als Adresse des Endpunkts

```
http://srv001.mustermann.local/veriniceserver/sync/syncService
```

und als `sourceId` den Bezeichner „Mustermann-Inventar" eingetragen. Zu Testzwecken lässt er sich alle ein- und ausgehenden SOAP-Nachrichten in eine Textdatei schreiben.

Der folgende SOAP-Request, aus Platzgründen im Querformat dargestellt, wurde nun an den Webservice gesendet:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="{SOAP}">
   <soapenv:Header/>
   <soapenv:Body>
      <syncRequest xsi:schemaLocation="{SYNC} sync.xsd"
                   xmlns="{SYNC}"
                   xmlns:xsi="{XSI}"
                   sourceId="Mustermann-Inventar" insert="true" update="false" delete="false"
                   >
         <syncData xmlns="{DATA}">
            <syncObject extId="Person1" extObjectType="Person">
               <syncAttribute name="Name" value="Mayer"/>
               <syncAttribute name="Vorname" value="Christian"/>
               <syncAttribute name="Email" value="cmayer@mustermann-gmbh.org"/>
            </syncObject>
            <syncObject extId="Person2" extObjectType="Person">
               <syncAttribute name="Name" value="Schmidt"/>
               <syncAttribute name="Vorname" value="Lisa"/>
               <syncAttribute name="Email" value="lschmidt@mustermann-gmbh.org"/>
            </syncObject>
            <syncObject extId="Person3" extObjectType="Person">
               <syncAttribute name="Name" value="Schulze"/>
               <syncAttribute name="Vorname" value="Wolfgang"/>
               <syncAttribute name="Email" value="wschulze@mustermann-gmbh.org"/>
            </syncObject>
            <syncObject extId="Person4" extObjectType="Person">
               <syncAttribute name="Name" value="Mueller"/>
               <syncAttribute name="Vorname" value="Stefan"/>
               <syncAttribute name="Email" value="smueller@mustermann-gmbh.org"/>
            </syncObject>
            <syncObject extId="Person5" extObjectType="Person">
               <syncAttribute name="Name" value="Klein"/>
```

```
34      <syncAttribute name="Vorname" value="Julia"/>
35        <syncAttribute name="Email" value="jklein@mustermann-gmbh.org"/>
36      </syncObject>
37      <syncObject extId="Person6" extObjectType="Person">
38        <syncAttribute name="Name" value="Bauer"/>
39        <syncAttribute name="Vorname" value="Hans"/>
40        <syncAttribute name="Email" value="hbauer@mustermann-gmbh.org"/>
41      </syncObject>
42      <syncObject extId="PC1" extObjectType="PC">
43        <syncAttribute name="Hostname" value="pc001"/>
44        <syncAttribute name="IPv4Addr" value="10.0.0.1"/>
45      </syncObject>
46      <syncObject extId="PC2" extObjectType="PC">
47        <syncAttribute name="Hostname" value="pc002"/>
48        <syncAttribute name="IPv4Addr" value="10.0.0.2"/>
49      </syncObject>
50      <syncObject extId="PC3" extObjectType="PC">
51        <syncAttribute name="Hostname" value="pc003"/>
52        <syncAttribute name="IPv4Addr" value="10.0.0.3"/>
53      </syncObject>
54      <syncObject extId="PC4" extObjectType="PC">
55        <syncAttribute name="Hostname" value="pc004"/>
56        <syncAttribute name="IPv4Addr" value="10.0.0.4"/>
57      </syncObject>
58      <syncObject extId="PC5" extObjectType="PC">
59        <syncAttribute name="Hostname" value="pc005"/>
60        <syncAttribute name="IPv4Addr" value="10.0.0.5"/>
61      </syncObject>
62      <syncObject extId="PC6" extObjectType="PC">
63        <syncAttribute name="Hostname" value="pc006"/>
64        <syncAttribute name="IPv4Addr" value="10.0.0.6"/>
65      </syncObject>
66      <syncObject extId="Server1" extObjectType="Server">
67        <syncAttribute name="Hostname" value="srv001"/>
```

```
68      <syncAttribute name="IPv4Addr" value="10.0.10.1"/>
69        <syncAttribute name="Betriebssystem" value="Debian GNU/Linux"/>
70      </syncObject>
71      <syncObject extId="Server2" extObjectType="Server">
72        <syncAttribute name="Hostname" value="srv002"/>
73        <syncAttribute name="IPv4Addr" value="10.0.10.2"/>
74        <syncAttribute name="Betriebssystem" value="Debian GNU/Linux"/>
75      </syncObject>
76      <syncObject extId="Server3" extObjectType="Server">
77        <syncAttribute name="Hostname" value="srv003"/>
78        <syncAttribute name="IPv4Addr" value="10.0.10.3"/>
79        <syncAttribute name="Betriebssystem" value="Windows Server 2008"/>
80      </syncObject>
81      <syncObject extId="Raum1" extObjectType="Raum">
82        <syncAttribute name="Bezeichnung" value="Bueroraum ßgro"/>
83      </syncObject>
84      <syncObject extId="Raum2" extObjectType="Raum">
85        <syncAttribute name="Bezeichnung" value="Bueroraum klein"/>
86      </syncObject>
87      <syncObject extId="Raum3" extObjectType="Raum">
88        <syncAttribute name="Bezeichnung" value="Serverraum"/>
89      </syncObject>
90      <syncObject extId="Gebaeude1" extObjectType="Gebaeude">
91        <syncAttribute name="Bezeichnung" value="Firmengebaeude"/>
92      </syncObject>
93
94    </syncData>
95
96    <syncMapping xmlns="{MAPPING}">
97
98      <mapObjectType extId="Raum" intId="raum">
99        <mapAttributeType extId="Bezeichnung" intId="raum_name"/>
100     </mapObjectType>
101     <mapObjectType extId="Server" intId="server">
```

```
102          <mapAttributeType extId="Hostname" intId="server_name"/>
103          <mapAttributeType extId="IPv4Addr" intId="server_netadr"/>
104          <mapAttributeType extId="Betriebssystem" intId="server_plattform"/>
105       </mapObjectType>
106       <mapObjectType extId="PC" intId="client">
107          <mapAttributeType extId="Hostname" intId="client_name"/>
108          <mapAttributeType extId="IPv4Addr" intId="client_netadr"/>
109       </mapObjectType>
110       <mapObjectType extId="Person" intId="person">
111          <mapAttributeType extId="Name" intId="nachname"/>
112          <mapAttributeType extId="Vorname" intId="vorname"/>
113          <mapAttributeType extId="Email" intId="person_email"/>
114       </mapObjectType>
115       <mapObjectType extId="Gebaeude" intId="gebaeude">
116          <mapAttributeType extId="Bezeichnung" intId="gebaeude_name"/>
117       </mapObjectType>
118
119    </syncMapping>
120
121 </syncRequest>
122 </soapenv:Body>
123 </soapenv:Envelope>
```

Auch hier wurden wieder sämtliche Namespace-URIs abgekürzt, die vollständigen URIs sind dem Anhang A zu entnehmen.

Es wurde nur das Flag `insert=true` gesetzt, da wir lediglich neue Objekte anlegen wollten. Außerdem ist die korrekte `sourceId` als Attribut von `<syncRequest>` angegeben. Unterhalb dieses Elements sehen wir eine ganze Reihe von `<syncObject>`s. Für jedes exportierte Objekt ist ein solches Element zu finden.

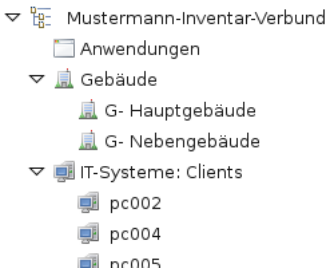
Herr Müller schickt die obige Anfrage an den Webservice und wartet auf die Antwort des Servers. Nach kurzer Wartezeit bekommt er eine positive Rückmeldung über die synchronisierten Inventardaten.

Die Antwort des Webservices sieht wiefolgt aus:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="{SOAP}">
   <SOAP-ENV:Header/>
   <SOAP-ENV:Body>
      <sync:syncResponse xmlns:sync="{SYNC}">
         <sync:replyMessage>Synchronisation erfolgreich.</sync:
            replyMessage>
         <sync:inserted>19</sync:inserted>
         <sync:updated>0</sync:updated>
         <sync:deleted>0</sync:deleted>
      </sync:syncResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Offenbar wurden 19 Inventarobjekte erfolgreich in die verinice-Datenbank eingefügt. Wie nicht anders zu erwarten war, wurden jedoch keine Objekte aktualisiert oder gelöscht.

Herr Müller ist zunächst kritisch und startet seinen verinice-Client. Er meldet sich mit seinen Benutzerdaten am Server an und findet einen neu angelegten IT-Verbund vor:



Der IT-Verbund wurde gemäß seiner Angabe der sourceId als „Mustermann-Inventar-Verbund" bezeichnet. Er kann alle importierten Objekte in ihren entsprechenden Kategorien vorfinden, das Mapping hat also funktioniert.

2.5.2 Zweite Synchronisation mit verinice – Aktualisieren von Objekten

Wir wollen eine zweite Anfrage an den Webservice stellen, um ein paar Änderungen am IT-Verbund mit verinice abzugleichen. Dazu muss selbstverständlich neben dem `insert`-Flag auch `update` gesetzt sein, ansonsten würden geänderte Attribute vom Webservice ignoriert werden.

Abbildung 2.14 zeigt die grafische Übersicht über den veränderten IT-Verbund.

Bei der Mustermann GmbH hat sich einiges getan. Weil sich das Unternehmen langfristig vergrößern will, wurde ein Nebengebäude angebaut, in das der Serverraum umgezogen ist. Zwischenzeitig wurde der alte, nun leerstehende Serverraum untervermietet, fällt also aus der Inventarisierung heraus.

Das alte Hauptgebäude wurde sinnvollerweise nun in „Hauptgebäude" umbenannt. Außerdem wurde auf einem der Server ein neues Betriebssystem, Ubuntu GNU/Linux aufgespielt. Herr Müller stößt in seinem Export-Werkzeug ein weiteres Mal die Synchronisation an und setzt diesmal das Häkchen bei „Update (Daten aktualisieren)".

Unser `<syncRequest>`-Element muss, wie bereits angesprochen, diesmal die Attribute `insert="true"` und `update="true"` aufweisen:

```
1 <syncRequest ...
2             sourceId="Mustermann-Inventar"
3             insert="true" update="true" delete="false">
```

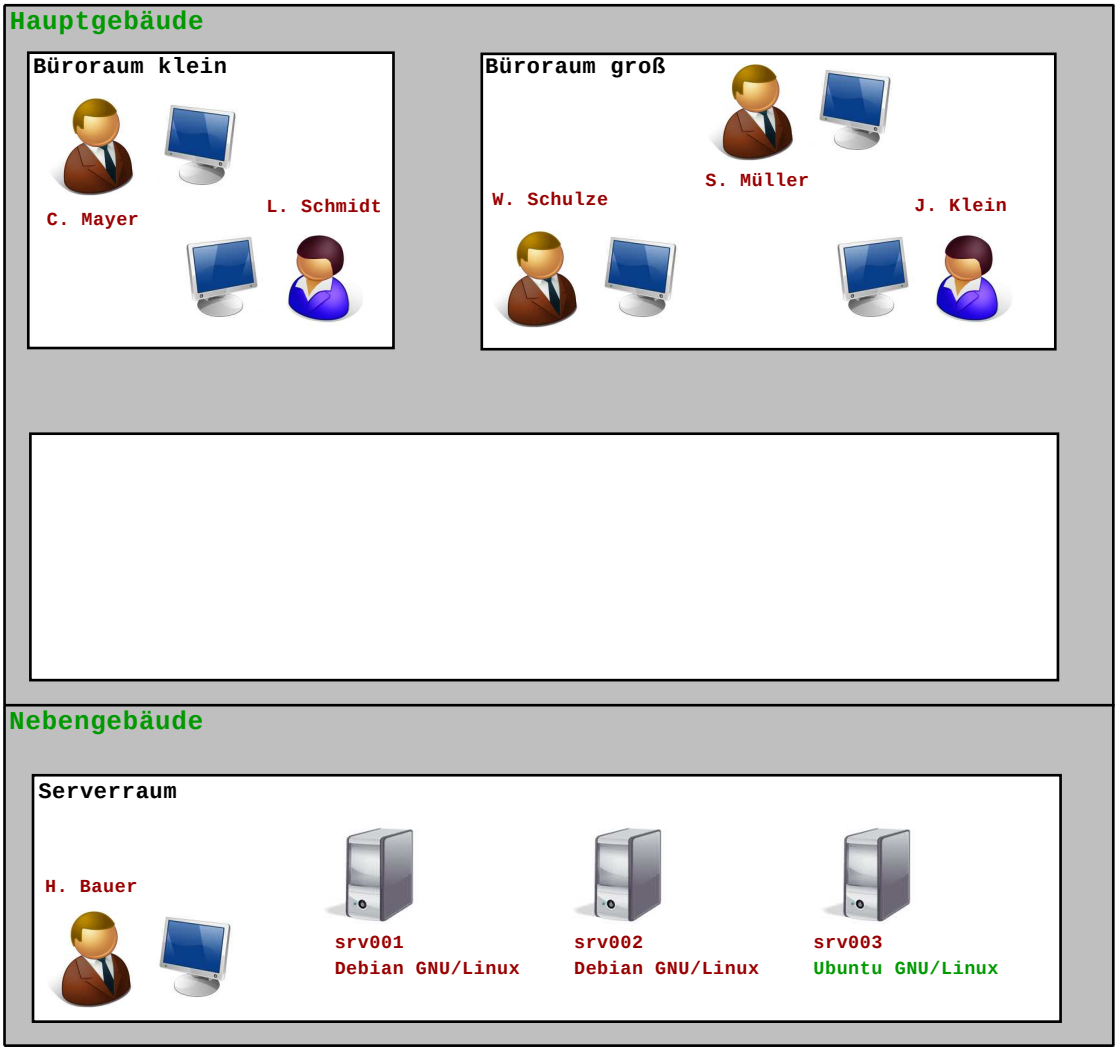


Abbildung 2.14: Schnappschuss des IT-Verbundes vor der zweiten Synchronisation

Anstatt hier wieder ein längliches Listing der gesamten Synchronisationsdaten zu zeigen, werden lediglich die geänderten bzw. neu angelegten <syncObject>-Elemente dargestellt. Zum Einen wurde auf dem Server #3 Ubuntu aufgespielt:

```
1 <syncObject extId="Server3" extObjectType="Server">
2     <syncAttribute name="Hostname" value="srv003" />
3     <syncAttribute name="IPv4Addr" value="10.0.10.3" />
4     <syncAttribute name="Betriebssystem" value="Ubuntu GNU/Linux"
        />
5 </syncObject>
```

Zum Anderen ist ein Gebäude neu hinzugekommen, während das Firmengebäude umbenannt wurde:

```
<syncObject extId="Gebaeude1" extObjectType="Gebaeude">
    <syncAttribute name="Bezeichnung" value="Hauptgebaeude" />
</syncObject>
<syncObject extId="Gebaeude2" extObjectType="Gebaeude">
    <syncAttribute name="Bezeichnung" value="Nebengebaeude" />
</syncObject>
```

Die Antwort des Servers ist wenig überraschend:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="{SOAP}">
   <SOAP-ENV:Header/>
   <SOAP-ENV:Body>
      <sync:syncResponse xmlns:sync="{SYNC}">
         <sync:replyMessage>Synchronisation erfolgreich.</sync:
            replyMessage>
         <sync:inserted>1</sync:inserted>
         <sync:updated>19</sync:updated>
         <sync:deleted>0</sync:deleted>
      </sync:syncResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.5.3 Dritte Synchronisation mit verinice – Löschen von Objekten

Abschließend wollen wir uns noch anschauen, wie das Löschen veralteter Objekte funktioniert. Zur Erinnerung: Setzt man das `delete`-Flag auf `true`, so sammelt der Server sämtliche Objekte aus der angegebenen Quelle, die ihm bekannt sind und verwirft unwiderruflich alle Objekte, die in den neuen Synchronisationsdaten nicht mehr enthalten sind.

Leider hat sich die Mustermann GmbH mit dem Neubau des Nebengebäudes finanziell etwas übernommen. Aufgrund der derzeit schwierigen Wirtschaftslage sieht sich das Management gezwungen, sich von den Mitarbeitern Schulze und Mayer zu trennen[I]. Auch bei der IT-Infrastruktur mussten einige Kürzungen vorgenommen werden.

Um den geänderten IT-Verbund mit verinice zu synchronisieren, aktiviert Herr Müller nun auch die Checkbox „Veraltete Objekte löschen" und sendet die geänderten Inventardaten an den Webservice, was wiederum mit einer Erfolgsmeldung bestätigt wird.

[I] Beide Mitarbeiter konnten jedoch zwischenzeitig führende Positionen in großen deutschen Softwarehäusern einnehmen.

Abbildung 2.15 zeigt die aktuelle Übersicht über den IT-Verbund. Insgesamt wurden zwei Mitarbeiter mit ihren Clients und ein Server aus dem IT-Verbund entfernt.

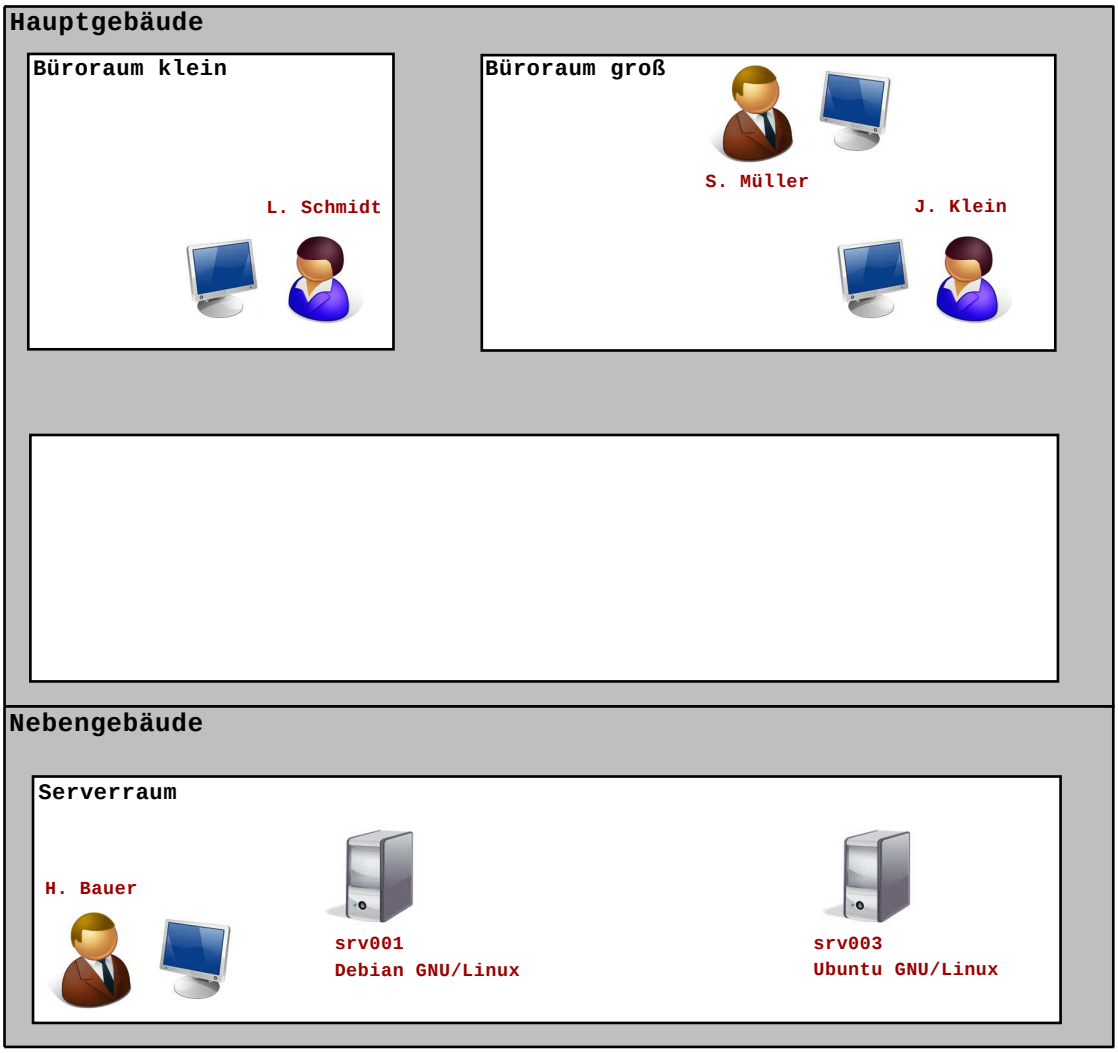


Abbildung 2.15: Schnappschuss des IT-Verbundes vor der dritten Synchronisation

Der SOAP-Request ändert sich nur in sofern, dass zum Einen das `delete`-Flag gesetzt ist und zum Anderen die insgesamt fünf Objekte, die aus dem Verbund entfernt wurden, nicht mehr in den Inventardaten enthalten sind (aus dem einfachen Grund, dass sie aus der Inventardatenbank gelöscht wurden). An dieser Stelle wird dem Leser zugetraut, sich die geänderte XML-Struktur ohne weiteres Codebeispiel vorstellen zu können.

Der Webservice antwortet auch bei diesem dritten Request mit einer Erfolgsmeldung:

```
1 <SOAP-ENV:Envelope xmlns:SOAP-ENV="{SOAP}">
```

```
2     <SOAP-ENV:Header/>
3     <SOAP-ENV:Body>
4        <sync:syncResponse xmlns:sync="{SYNC}">
5           <sync:replyMessage>Synchronisation erfolgreich.</sync:
               replyMessage>
6           <sync:inserted>0</sync:inserted>
7           <sync:updated>15</sync:updated>
8           <sync:deleted>5</sync:deleted>
9        </sync:syncResponse>
10    </SOAP-ENV:Body>
11 </SOAP-ENV:Envelope>
```

Hoffentlich konnte der Leser anhand dieser Fallstudie etwas mitnehmen, wenn es sich dabei auch um ein recht hypothetisches Beispiel handelt und die eigentliche Schwierigkeit, nämlich die Programmierung einer XML Export-Schnittstelle, unterschlagen wurde.

Zumindest die Nachrichtenstruktur der SOAP-Nachrichten sollte etwas klarer geworden sein, für den unwahrscheinlichen Fall, dass der Leser einmal in die Verlegenheit kommen sollte, unmittelbar mit dem Webservice interagieren zu müssen.

Hier sei noch einmal darauf hingewiesen, dass zukünftig ein Wizard in verinice integriert werden soll, der den Benutzer vor der unmittelbaren Berührung mit dem Webservice bewahrt. Dennoch steht natürlich auch in Zukunft der Webservice zur Verfügung, um direkt gegen ein Synchronisations-API programmieren zu können, ohne einen verinice-Client zu benötigen.

3 Technische Dokumentation

Dieser Abschnitt richtet sich an die Entwickler von verinice, die den Webservice zur Synchronisation einpflegen und warten, sowie alle diejenigen, die sich mit der Implementierung des Dienstes befassen wollen.

Der Webservice läuft im Kontext des verinice Applikationsservers. Bei der Implementierung des Dienstes wurde der Ansatz des *Contract First* Webservices verfolgt. Es wurde also, ausgehend von einer Beschreibung des Datenkontrakts in Form von XML-Schemata, eine Endpunktklasse implementiert, die valide Anfragen verarbeitet und die Synchronisation vornimmt.

In verinice wurde vorher bereits das Spring-Framework eingesetzt. Hierbei handelt es sich in erster Linie um ein Framework zur Dependency Injection (DI) und aspektorientierten Programmierung (AOP). Beide Techniken führen zu lose gekoppelten Code. Dependency Injection sorgt dafür, dass Abhängigkeiten gegenüber konkreten Klassen bspw. zur Datenbankanbindung zentral an einer Stelle konfiguriert werden können (im Falle von Spring in einer XML-Konfigurationsdatei). Hierdurch finden sich nicht mehr über den gesamten Code verteilt Auszüge wie der folgende:

```
1 IBusinessObjectFactory myFactory = new BusinessObjectFactory();
```

Stattdessen reicht es aus, an dieser Stelle ein schreibbares Attribut

```
1 IBusinessObjectFactory myFactory;
```

zu deklarieren und das Attribut durch ein DI-Framework wie Spring von außen zu *injizieren*. Dadurch ist es dem Entwickler möglich, in seinen Klassen ausschließlich gegen Schnittstellen, statt gegen konkrete Klassen zu programmieren.

Mit Hilfe der aspektorientierten Programmierung können übergreifende Aspekte wie die Benutzerverwaltung und Sicherheitsaspekte zentral implementiert werden. An allen Stellen, an denen beispielsweise die Authentifikation eines Benutzers nötig ist, sorgt nun ein AOP-Framework wie Spring dafür, dass der entsprechende *Aspekt* an solchen *Pointcuts* „eingewebt" wird. Somit verteilt sich übergreifender Code nicht über den gesamten Quelltext, was zu Problemen führen würde, falls z.B. die Benutzerauthentifikation umgestellt werden muss.

Um nun den roten Faden zu unserem Synchronisations-Webservice (im Folgenden kurz Sync-WS) wieder aufzunehmen – Spring ist viel mehr als ein DI- und AOP-Framework. Die Kernfunktionalität, welche die Verschaltung von POJOs (Plain Old Java Objects) zur DI und AOP in einem Container erlaubt, wird ergänzt durch unzählige Module

zur Webentwicklung, objektrelationalen Abbildung und vielen mehr. Darunter auch das recht neue Modul *Spring Web Services* (im Folgenden kurz Spring-WS). Dieses Modul erlaubt es, ein in Spring konfiguriertes POJO (im Folgenden Spring-Bean genannt), als Webservice anzubieten. Die WSDL-Datei die den Webservice beschreibt, wird dabei dynamisch zur Laufzeit aus den Angaben der Spring-Konfigurationsdatei und dem zuvor erstellten Datenkontrakt generiert.

Diesen Ansatz nennt man auch *Contract First*, weil er von der Schemabeschreibung der ausgetauschten Nachrichten ausgeht. Man legt sich also zunächst (im Idealfall einmalig) auf den Datenkontrakt fest, unabhängig von der Implementierung. Der Vorteil liegt auf der Hand: Änderungen des APIs der internen Implementierung haben keinen Einfluss auf das API des Webservices nach außen. Haben erst einmal mehrere Anbieter den Webservice in ihren Implementierungen verwendet, wirken sich selbst kleinste Änderungen mitunter dramatisch auf die Funktionalität dieser Produkte aus, die nun auf das geänderte API angepasst werden müssten.

Entsprechend der vorgeschlagenen Vorgehensweise von Craig Walls in [8] wird die Implementierung des Webservices in drei Schritten vorgestellt. Zunächst werden die Schemadateien für den Datenkontrakt kurz erläutert. Für eine intuitivere Einführung empfiehlt es sich, vorher den Abschnitt 2.4 zu lesen. Anschließend wird die Implementierung des Endpunkts vorgestellt, der von `AbstractJDOMPayloadEndpoint` abgeleitet ist und Anfragen mit Hilfe von DOM und XPath verarbeitet. Schließlich fügen wir die Puzzlestücke zu einem Ganzen zusammen. Die Einbindung des Endpunkts und einiger weiterer Klassen in den Spring-Container, sowie die marginalen Änderungen in der Konfiguration des verwendeten OR-Mappers Hibernate bilden den Abschluss.

Im Anhang B befindet sich eine Übersicht aller im Rahmen der Entwicklung des Webservices neuen und geänderten Pakete und Dateien. Dies bezieht sich auf die Entwicklerversion v0.8 aus den verinice-Quellen.

3.1 Datenkontrakt

In diesem Abschnitt wird der Datenkontrakt des Webservices anhand der XML-Schemata erläutert. Vorab ist es hilfreich, den Abschnitt 2.4 gelesen zu haben, in dem die Nachrichtenformate anhand anschaulicher Beispiele eingeführt werden. Die Schemadateien sind im `WebContent`-Verzeichnisses des verinice-Servers zu finden.

Wie bereits im Abschnitt *Anwender-Dokumentation* gesehen, enthält der SOAP-Body einer Anfrage an den Webservice genau ein `<syncRequest>`-Element. Die Struktur dieses Elements ist in der Schemadatei `sync.xsd` festgelegt. Diese Datei, wie auch die anderen beiden Schemadateien werden im Folgenden auszugsweise erläutert.

3.1.1 Schemadatei sync.xsd

Die Definition des <syncRequest>-Elements in der Datei `sync.xsd` ist äußerst übersichtlich gehalten:

sync.xsd

```
14 <xsd:element name="syncRequest">
15     <xsd:complexType>
16         <xsd:sequence>
17             <xsd:element ref="data:syncData" minOccurs="1"
                   maxOccurs="1"/>
18             <xsd:element ref="map:syncMapping" minOccurs="1"
                   maxOccurs="1"/>
19         </xsd:sequence>
20         <xsd:attribute name="sourceId" type="xsd:string" use="
               required"/>
21         <xsd:attribute name="insert" type="xsd:boolean" default="
               true"/>
22         <xsd:attribute name="update" type="xsd:boolean" default="
               true"/>
23         <xsd:attribute name="delete" type="xsd:boolean" default="
               false"/>
24     </xsd:complexType>
25 </xsd:element>
```

Das Element <syncRequest> hat vier Attribute. Neben dem Pflichtattribut `sourceId` können die boolschen Flags für die drei Synchronisationsmethoden angegeben werden. Ist dies nicht der Fall, werden diese mit sinnvollen Vorgabewerten belegt. Standardmäßig werden Objekte eingefügt und aktualisiert, aber nicht gelöscht.

Die beiden Unterelemente werden in den separaten Schemadateien `data.xsd` und `mapping.xsd` definiert, auf die noch später eingegangen wird. Diese Schemadateien werden, unter Beachtung ihrer jeweiligen Namespaces `{DATA}` und `{MAPPING}` in `sync.xsd` eingebunden:

sync.xsd

```
8  <xsd:import namespace="{DATA}"
9              schemaLocation="data.xsd"/>
10
11 <xsd:import namespace="{MAPPING}"
12             schemaLocation="mapping.xsd"/>
```

Auch hier wurden und werden weiterhin die Namespace-Kürzel aus Anhang A verwendet. Das Nachrichtenformat für die Antwortnachrichten des Webservices ist ebenfalls in dieser Schemadatei festgelegt:

sync.xsd

```
34 <xsd:element name="syncResponse">
35     <xsd:complexType>
36         <xsd:sequence>
37             <xsd:element name="replyMessage" type="xsd:string"
                   minOccurs="1" maxOccurs="1"/>
38             <xsd:element name="inserted" type="xsd:int" minOccurs=
                   "1" maxOccurs="1"/>
39             <xsd:element name="updated" type="xsd:int" minOccurs="
                   1" maxOccurs="1"/>
40             <xsd:element name="deleted" type="xsd:int" minOccurs="
                   1" maxOccurs="1"/>
41         </xsd:sequence>
42     </xsd:complexType>
43 </xsd:element>
```

Das <syncResponse>-Element enthält vier Unterelemente mit Inhalten einfacher Typen. Angegeben werden eine Textnachricht, die entweder eine kurze Meldung „Synchronisation erfolgreich", oder alternativ eine Liste von Fehlern enthalten kann. Die anderen drei Elemente enthalten die numerische Angabe, wieviele Objekte bei der Synchronisation neu angelegt, aktualisiert oder gelöscht wurden.

3.1.2 Schemadatei data.xsd

In der Schemadatei `data.xsd` wird die Struktur des <syncData>-Elements festgelegt. Die Definition des Elements in der Schemadatei sieht wie folgt aus:

data.xsd

```
6  <xsd:element name="syncData">
7      <xsd:complexType>
8          <xsd:sequence>
9
10             <xsd:element name="syncObject" maxOccurs="unbounded">
11                 <xsd:complexType>
12                     <xsd:sequence>
13                         <xsd:element name="syncAttribute"
                               maxOccurs="unbounded">
14                             <xsd:complexType>
15
16                                 <xsd:attribute name="name" type="
                                       xsd:string" use="required"/>
17                                 <xsd:attribute name="value" type="
                                       xsd:string" use="required"/>
18
19                             </xsd:complexType>
```

```
20                          </xsd:element>
21                      </xsd:sequence>
22
23                      <xsd:attribute name="extId" type="xsd:string"
                            use="required"/>
24                      <xsd:attribute name="extObjectType" type="
                            xsd:string" use="required"/>
25
26                  </xsd:complexType>
27              </xsd:element>
28
29          </xsd:sequence>
30
31      </xsd:complexType>
32  </xsd:element>
```

Offenbar darf das <syncData>-Element beliebig viele <syncObject>-Elemente beinhalten, die jeweils ein Objekt des IT-Verbundes repräsentieren. Ein <syncObject>-Element trägt zwei obligatorische Attribute, `extId` und `extObjectType`. Ersteres ist die applikationsweit eindeutige ID des Objektes in der Synchronisationsquelle, z.B. `Server4711`, während letzteres der Bezeichnung des Objekttypen entspricht, z.B. `Server`. Außerdem darf ein <syncObject> beliebig viele Unterelemente vom Typ <syncAttribute> enthalten. Diese leeren Elemente enthalten lediglich zwei Attribute `name` und `value` vom Typ einer Zeichenkette, um Namen und Wert eines Attributes des umgebenden <syncObject>-Elements aufnehmen zu können.

3.1.3 Schemadatei mapping.xsd

In der Schemadatei `mapping.xsd` wird die Struktur des <syncMapping>-Elements festgelegt. Dieses Element beschreibt die Abbildung eines Objekttypen der Synchronisationsquelle mit seinen Attributen auf einen Hitro-UI Entitätstypen in verinice:

mapping.xsd

```
14      <xsd:element name="syncMapping">
15          <xsd:complexType>
16              <xsd:sequence>
17
18                  <xsd:element name="mapObjectType" minOccurs="0"
                        maxOccurs="unbounded">
19                      <xsd:complexType>
20                          <xsd:sequence>
21
22                              <xsd:element name="mapAttributeType"
                                    minOccurs="0" maxOccurs="unbounded"
                                    >
```

```
23                                  <xsd:complexType>
24                                      <xsd:attribute name="extId"
                                            type="xsd:string"/>
25                                      <xsd:attribute name="intId"
                                            type="xsd:string"/>
26                                  </xsd:complexType>
27                              </xsd:element>
28
29                          </xsd:sequence>
30                          <xsd:attribute name="extId" type="xsd:ID"
                                use="required"/>
31                          <xsd:attribute name="intId" type="
                                xsd:string" use="required"/>
32                      </xsd:complexType>
33                  </xsd:element>
34
35              </xsd:sequence>
36          </xsd:complexType>
37      </xsd:element>
```

Das <syncMapping>-Element enthält offenbar beliebig viele <mapObjectType>-Elemente. Jedes dieser Unterelemente beschreibt die Abbildung eines Objekttypen aus der Synchronisationsquelle mit seinen Attributen auf einen Hitro-UI Entitätstypen in verinice.

Ein <mapObjectType>-Element enthält die minimale Menge von Angaben, die dazu nötig sind, eine solche Abbildung durchzuführen. Benötigt werden zwei Bezeichner von Objekttypen in der Quelle (`extId`) bzw. in verinice (`intId`). Für die externe ID wurde der Typ „ID" gewählt, damit die Schemadatei dazu verwendet werden kann beispielsweise in DOM von der Methode `GetElementById()` Gebrauch zu machen. So kann auf einfache Weise für jedes Objekt, insofern vorhanden, das passende Mapping gefunden werden.

Jedes <mapObjectType> enthält wiederum beliebig viele <mapAttributeType>-Elemente, um auch die zugehörigen Attribute aufeinander abzubilden. Hier ist nur zu beachten, dass die externe ID des Attributes nur innerhalb des zugehörigen Objekttypen eindeutig sein sollte, es wird aber keine applikationsweit eindeutige Bezeichnung verlangt. So kann z.B. sowohl eine Person, als auch ein Gebäude jeweils ein Attribut *Name* tragen.

3.2 Dienstendpunkt

Der Endpunkt, der eingehende Anfragen an unseren Synchronisationsdienst verarbeitet, liegt in der Klasse `SyncEndpoint` im Paket `sernet.gs.server.sync` des verinice-Servers. Diese Klasse erbt von `AbstractJDOMPayloadEndpoint`, einer Klasse des Spring-WS Moduls. Die zentrale Methode `invokeInternal()` wurde überschrieben, um die Synchronisation durchzuführen. Diese Methode wird immer dann von Spring aufgerufen, wenn ein Re-

quest an den Webservice eingeht. Wie genau dieser Mechanismus konfiguriert wurde, ist dem nächsten Abschnitt zu entnehmen.

Die Methode erhält als einzigen Parameter ein JDOM-Element, welches die Nutzdaten der SOAP-Nachricht enthält, in unserem Fall also ein `<syncMapping>`-Element mit seinen Kindelementen (vgl. Abbildung 3.1).

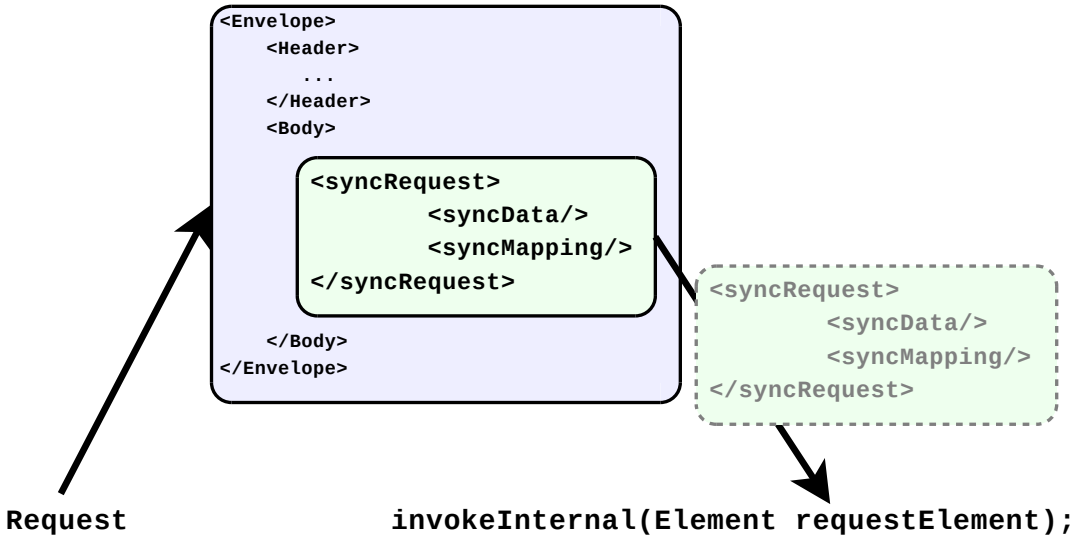


Abbildung 3.1: Übermittlung des Nachrichten-Payloads an den Endpunkt

Zunächst werden die Attribute `syncId`, `insert`, `update` und `delete` mit Hilfe der JDOM-Methode `getAttributeValue()` aus dem Wurzelelement ausgelesen. Hier werden außerdem die Default-Werte gemäß der Schemadatei `sync.xsd` berücksichtigt[1]. Auch die Kindelemente `<syncData>` und `<syncMapping>` werden dem Wurzelelement entnommen.

Anschließend delegiert unser Endpunkt die eigentliche Arbeit an zwei Commands, die von `GenericCommand` erben und im Paket `sernet.gs.ui.rcp.main.service.crudcommands` liegen:

```
85 SyncInsertUpdateCommand cmdInsertUpdate = new
      SyncInsertUpdateCommand( sourceId, syncDataElement,
      syncMappingElement, insert, update, errorList );
86 cmdInsertUpdate = ServerCommandService.getCommandService().
      executeCommand( cmdInsertUpdate );
87
88 inserted += cmdInsertUpdate.getInserted();
```

[1] An dieser Stelle wäre es sicherlich sauberer, die Schemadatei zu Rate zu ziehen, anstatt die Vorgabewerte hart zu kodieren. Auch eine Schemaprüfung wäre vorab angebracht.

```
89  updated += cmdInsertUpdate.getUpdated();
90
91  if( delete )
92  {
93      SyncDeleteCommand cmdDelete = new SyncDeleteCommand( sourceId,
            syncDataElement, errorList );
94      cmdDelete = ServerCommandService.getCommandService().
            executeCommand( cmdDelete );
95      deleted += cmdDelete.getDeleted();
96  }
```

Die `execute()`-Methoden dieser beiden Kommandoklassen werden in den folgenden Unterabschnitten skizziert. Basierend auf den Inhalten der Variablen `inserted`, `updated` und `deleted` (Zählervariablen, welche die Anzahl der eingefügten, aktualisierten und gelöschten Objekte zählen), sowie der Fehlerliste `errorList`, baut der Endpunkt nach der Ausführung der Kommandos ein weiteres JDOM-Element `responseElement` zusammen:

SyncEndpoint.java

```
106 Element responseElement = new Element( "syncResponse",
        SyncNamespaceUtil.SYNC_NS );
107
108 /* ... */
109
110 return responseElement;
```

Dieses Element wird von der `invokeInternal`-Methode zurückgegeben, von Spring in einen SOAP-Envelope verpackt und an den Client zurückgeschickt. Zur zentralen Verwaltung der konkreten Namespace-URIs wird die Hilfsklasse `SyncNamespaceUtil` aus dem Paket `sernet.gs.ui.rcp.main.sync` verwandt. Diese Klasse definiert die drei Konstanten `SYNC_NS`, `DATA_NS` und `MAPPING_NS` vom Typ `org.jdom.Namespace`. Ebenfalls aus diesem Paket sind die beiden Exceptionklassen `InternalErrorException` und `InvalidRequestException`. Diese Exceptions werden bei entsprechenden Fehlern von den Kommandoklassen geworfen. Unsere Endpunktklasse behandelt diese Exceptions jedoch nicht selbst, sondern weist ebenfalls das entsprechende `throws`-Statement auf. Stattdessen werden diese Exceptions von Spring-WS mit Hilfe eines Exception Resolvers in SOAP Faults umgesetzt, die an den Client zurückgeschickt werden. Dies geschieht in der Servlet-Konfigurationsdatei `messageDispatcher-servlet.xml` und ist im Abschnitt 3.3 näher erläutert.

3.2.1 Kommando SyncInsertUpdateCommand

Die Kommandoklasse `SyncInsertUpdateCommand` im Paket `sernet.gs.ui.rcp.main.service.crudcommands` ist dafür zuständig, abhängig von den entsprechenden Flags

(insert,update) neue Objekte einzufügen und/oder bereits vorhandene Objekte zu aktualisieren. Die dazu notwendigen Parameter werden dem Kommando im Konstruktor übergeben:

SyncInsertUpdateCommand.java

```
1  public SyncInsertUpdateCommand( String sourceId,
2                                  Element syncDataElement,
3                                  Element syncMappingElement,
4                                  boolean insert,
5                                  boolean update,
6                                  List<String> errorList )
7  {
8      this.sourceId = sourceId;
9      this.syncDataElement = syncDataElement;
10     this.syncMappingElement = syncMappingElement;
11     this.insert = insert;
12     this.update = update;
13     this.errorList = errorList;
14 }
```

Das Kommandoobjekt enthält also vor seiner Ausführung per `execute()` in jedem Fall alle notwendigen Parameter. Nun wird grob skizziert, wie das Objekt beim Aufruf der `execute()`-Methode vorgeht.

Zunächst wird das `BSIModel`-Objekt abgerufen, welches als Wurzelelement für alle derzeit in verinice enthaltenen IT-Verbünde dient. Darin sucht das Kommando als nächstes nach einem IT-Verbund mit der übergebenen `sourceId`[1]. Ist die Suche erfolgreich, werden im Folgenden neue Objekte im gefundenen IT-Verbund angelegt. Falls die Suche fehlschlägt, wird zuvor ein neuer IT-Verbund erzeugt.

Anschließend werden die `<syncObject>`-Elemente im übergebenen `<syncData>`-Element untersucht. Für jedes `<syncObject>` wird ein *passendes* `<mapObjectType>`-Element unterhalb des übergebenen `<syncMapping>`-Elements gesucht. Unter einem *passenden* Mapping-Element ist zu verstehen, dass die `extId` des Mappings mit dem `extObjectType`-Attribut des Inventarobjektes übereinstimmen muss, denn wir suchen ein Mapping-Element, welches die Abbildung des externen Objekttypen auf einen internen Hitro-UI Entitätstypen beschreibt. Diese Suche wird mit Hilfe der XPath-Unterstützung in JDOM und folgendem Ausdruck realisiert:

```
//map:mapObjectType[@extId="{{extObjectType}}"]
```

[1] Das Feld *sourceId*, wie auch das Feld *extId* wurde für den Webservice als zusätzliche Attribute der Klasse *CnATreeElement* angelegt. Dies musste auch in den Hibernate-Mappings entsprechend nachgezogen werden, was in Abschnitt 3.3 erläutert wird.

Dabei ist der Platzhalter `{{extObjectType}}` durch den jeweils gesuchten Wert zu ersetzen. Wenn ein Mapping gefunden wird, können wir die zugehörige `intId` verwenden, um den verinice-Objekttypen und die Kategorie herauszufinden, in die ein ggf. neu anzulegendes Objekt „eingehängt" wird. Beide Zuordnungen werden mittels einer Hashmap vorgenommen.

Um das gerade betrachtete Objekt neu anzulegen bzw. zu aktualisieren, wird es zunächst in der Datenbank gesucht. Hierzu wird der zusätzliche Schlüssel (`sourceId`,`extId`) verwendet. Das weitere Vorgehen zum Einfügen/Aktualisieren sieht in Pseudocode-Notation wie folgt aus:

```
boolean setAttributes = false;
CnATreeElement elementInDB = findDbElement( sourceId, extId );

if( elementInDB != null && update )
    setAttributes = true;

if( elementInDB == null && insert )
{
    elementInDB = createElement( veriniceObjectType );
    elementInDB.setSourceId( sourceId );
    elementInDB.setExtId( extId );
    setAttributes = true;
}

if( elementInDB != null && setAttributes )
{
    /* setze alle Attributwerte */
}
```

Hier wurde also das Aktualisieren der Attributwerte mit den neuen Werten aus der Quelle an den Schluss gestellt, unabhängig davon, ob zuvor ein neues Objekt angelegt wurde, oder ein bestehendes Objekt aktualisiert werden soll. Zur übersichtlicheren Darstellung wurden das Inkrementieren der Zählervariablen `inserted` und `updated`, die Zuordnung einer Kategorie, sowie sämtliche Schritte zur Fehlerbehandlung weggelassen.

Um die Attribute mit ihren neuen Werten aus der Quelle aktualisieren zu können, müssen auch die Attribute des gerade behandelten Objektes ge*mapped* werden. Dazu suchen wir für jedes `<syncAttribute>`-Kind des `<syncObject>` mit folgendem XPath-Ausdruck nach dem Mapping:

```
//mapObjectType[@extId="{{extObjectType}}"]/
mapAttributeType[@extId="{{attrExtId}}"]
```

Der Zeilenumbruch ist hier lediglich der Formatierung geschuldet. Wieder sind beide Platzhalter durch die entsprechenden Werte zu ersetzen. Funktioniert auch dieses Mapping, so wird das Attribut mit dem neuen Wert belegt:

SyncInsertUpdateCommand.java

```
288 String attrIntId = mapAttr.getAttributeValue( "intId" );
289 PropertyType propertyType = elementInDB.getEntityType().
       getPropertyType( attrIntId );
290
291 if( null != propertyType )
292     elementInDB.setSimpleProperty( attrIntId, attrValue );
```

Diese Schritte werden für jedes Objekt in den Synchronisationsdaten vollzogen. Zu beachten ist, dass Objekte zwar alle in *einem* automatisch erzeugten IT-Verbund angelegt werden, es aber kein Problem ist, wenn der Benutzer die Objekte verschiebt – anhand des Feldes `sourceId` können jederzeit unabhängig vom übergeordneten IT-Verbund, sämtliche Objekte aus einer Quelle wiedergefunden werden.

3.2.2 Kommando SyncDeleteCommand

Die Kommandoklasse `SyncDeleteCommand` im Paket `sernet.gs.ui.rcp.main.service.crudcommands` ist dafür zuständig, abhängig vom Wert des `update`-Flags, Objekte aus der verinice-Datenbank zu löschen, die in den neuen Inventardaten aus der Synchronisationsquelle nicht mehr vorhanden sind. Auch diesem Kommando werden vorab alle notwendigen Parameter im Konstruktor übergeben:

SyncDeleteCommand.java

```
55 public SyncDeleteCommand(   String sourceId,
56                             Element syncDataElement,
57                             List<String> errorList )
58 {
59     this.sourceId = sourceId;
60     this.syncDataElement = syncDataElement;
61     this.errorList = errorList;
62 }
```

Zunächst lädt die `execute()`-Methode dieses Kommandos alle in verinice enthaltenen Objekte aus der angegebenen `sourceId` mit Hilfe eines weiteren Kommandos `LoadCnAElementsBySourceID()`, dessen Konstruktor die `sourceId` als Parameter übergeben wird. Die Liste dieser Objekte wird nun mit der Liste derjenigen Objekte in den neuen Inventardaten verglichen. Sobald die Methode auf ein Objekt in verinice stößt, die in der neuen Liste nicht (mehr) enthalten ist, wird dieses Objekt in verinice gelöscht. In Pseudocode-Notation:

```
List oldElements = loadCnAElementsBySourceId( sourceId );
HashMap currentExtIds = new HashMap();

for( Element e : syncDataElement.getChildren( "syncObject" ) )
{
    currentExtIds.put( e.getAttributeValue( "extId" ), new Object
       () );
}

for( Element oldElement : oldElements )
{
    if( currentExtIds.get( oldElement.getAttributeValue( "extId" )
        ) == null )
    {
        removeElement( oldElement );
    }
}
```

Für jede vorhandene `extId` wird in der Hashtabelle `currentExtIds` ein beliebiges „Token" abgelegt. Uns interessiert hier lediglich die boolsche Information, ob ein Objekt mit einer bestimmten `extId` aus unserer Quelle bereits existiert. Daher bedienen wir uns des einfachsten in Java denkbaren Objekts für diese Information, einer Instanz der Klasse Object. Ansonsten wurden wieder technische Details wie die Schritte zur Fehlerbehandlung usw. weggelassen. Diese können dem kommentierten Quelltext entnommen werden. Statt der synchronen Funktionsaufrufe `loadCnAElementsBySourceId()` bzw. `removeElement` werden die gleichnamigen Kommandoklassen benutzt. Wichtig zu beachten ist außerdem, dass der IT-Verbund aus diesem Löschvorgang ausgeschlossen wird, da dieser niemals explizit in den Inventardaten zu finden ist und ansonsten immer gelöscht würde. Dies wird mittels `dbElement instanceof ITVerbund` abgefangen.

3.2.3 Kommandos zur Abfrage synchronisierter Objekte

Die beiden Kommandos zur Synchronisation bedienen sich zweier neuer Kommandos, die die Abfrage von Objekten aus der verinice-Datenbank anhand der neuen Schlüsselattribute (`sourceId`,`extId`) erlauben. Diese beiden Kommandoklassen liegen ebenfalls im Paket `sernet.gs.ui.rcp.main.service.crudcommands` und arbeiten ähnlich wie bspw. `LoadCnAElementById`. Zur Abfrage der jeweiligen Objekte wird die Hibernate Query Language (HQL) verwendet.

Das Kommando `LoadCnAElementByExternalID` lädt genau ein Objekt mit dem Schlüssel (`sourceId`,`extId`). Die HQL-Anfrage sieht wie folgt aus:

```
from CnATreeElement elmt where elmt.sourceId = {SOURCE_ID} and
   elmt.extId = {EXT_ID}
```

Dabei sind die Platzhalter `{SOURCE_ID}` und `{EXT_ID}` durch die jeweils gewünschten Werte zu ersetzen. Das gleiche gilt auch für das Kommando `LoadCnAElementsBySourceID`, welches eine *Menge* von Objekten aus der angegebenen Quelle sucht. Dazu wird die folgende HQL-Anfrage benutzt:

```
from CnATreeElement elmt where elmt.sourceId = {SOURCE_ID}
```

Beiden Kommandoklassen werden der oder die oben genannten Parameter, wie gewohnt, als Konstruktorargumente übergeben. Nach der Ausführung von `execute()` können die gefundenen Elemente mit der `getElements()`-Methode abgerufen werden.

3.3 Einbindung in Spring und Hibernate

Nachdem wir die Schemadateien für den Datenkontrakt, sowie die Implementierung des Endpunkts für unseren Webservice erläutert haben, müssen diese Puzzleteile zu einem Ganzen zusammengefügt werden. Es müssen einige Objekte (Spring Beans) im Spring-Container verschaltet werden, um den Webservice mit der dynamisch erzeugten WSDL-Datei publizieren zu können.

Vorab mussten jedoch noch kleinere Anpassungen der Klassen des IT-Verbundes vorgenommen werden. Alle Objekte in verinice sollten ab sofort eine `sourceId` und eine `extId` tragen[1]. Da alle Klassen von der gemeinsamen Oberklasse `CnATreeElement` aus dem Paket `sernet.gs.ui.rcp.main.common.model` erben, mussten die beiden Attribute auch nur dieser *einen* Klasse hinzugefügt werden:

CnATreeElement.java

```
189 private String sourceId;
190
191 private String extId;
192
193 public String getExtId()
194 {
195     return extId;
196 }
197
198 public void setExtId(String extId)
199 {
200     this.extId = extId;
201 }
```

[1] Tatsächlich haben wir diese beiden Attribute bereits in unserer Implementierung des Endpunkts genutzt. Dass erst jetzt erklärt wird, wie und wo diese Attribute hinzugefügt wurden, hat lediglich den Grund, dass insbesonders alle Anpassungen an bestehendem Code zentral in einem Abschnitt nachzulesen sind.

```
202
203 public String getSourceId()
204 {
205     return sourceId;
206 }
207
208 public void setSourceId(String sourceId)
209 {
210     this.sourceId = sourceId;
211 }
```

Sinnvollerweise sind beide Attribute lesbar und schreibbar. Um die Anpassungen konsistent in allen Schichten nachzuziehen, müsste nun auch die Persistenzschicht angepasst werden, im schlimmsten Fall Tabellenstrukturen und SQL-Abfragen. Allerdings bedient man sich in verinice des OR-Mappers (ein Tool, welches im Wesentlichen die objektrelationale Abbildung kapselt) Hibernate. Dieser ist so konfiguriert, dass für alle Klassen für die ein Hibernate-Mapping spezifiziert wurde, beim Programmstart entsprechende Tabellen in der Datenbank neu angelegt bzw. aktualisiert werden. Aus diesem Grund mussten die Attribute lediglich in den Mappingdateien für die Klasse `CnATreeElement`, `CnATreeElement.hbm.xml` und `CnATreeElement_derby.hbm.xml` Hibernate bekannt gemacht werden. In der Datei:

CnATreeElement.hbm.xml

```
26 <property
27     name="extId"
28 />
29
30 <property
31     name="sourceId"
32 />
```

Somit sind die beiden neuen Attribute bereits durchgängig in verinice verfügbar, werden jedoch (noch) nicht im GUI angezeigt.

Um die bereits implementierte Endpunktklasse als Webservice verfügbar zu machen, muss diese Klasse noch entsprechend in Spring konfiguriert werden. Zur Erinnerung: Wir haben uns für Spring-WS entschieden. Dabei handelt es sich um ein Teilprojekt des Spring-Frameworks, um *Contract First* Webservices in Spring-basierte Anwendungen zu integrieren.

Um eingehende Anfragen an den richtigen Adressaten weiterzuleiten bedient sich Spring-WS eines Mapping-Mechanismus', der äußerst mächtige Konfigurationsmöglichkeiten bietet. Analog zum `DispatcherServlet`, einer Servletklasse, die als Frontcontroller in Spring-MVC dient, um eingehende Anfragen anhand der URL durch einen bestimmten Controller weiter verarbeiten zu lassen, gibt es im Kontext von Webservices das

`MessageDispatcherServlet`. Dieses Servlet dient dazu, speziell SOAP-Anfragen an Endpunkte zur Verarbeitung weiterzuleiten. Diese Klasse muss, wie jedes andere Servlet auch, in der Datei `web.xml` konfiguriert werden. In unserem Fall wurde das Servlet wie folgt in der Datei `WebContent/WEB-INF/web.xml` des verinice-Servers angelegt:

web.xml

```
47 <servlet>
48     <servlet-name>messageDispatcher</servlet-name>
49     <servlet-class>org.springframework.ws.transport.http.
           MessageDispatcherServlet</servlet-class>
50     <init-param>
51         <param-name>transformWsdlLocations</param-name>
52         <param-value>true</param-value>
53     </init-param>
54 </servlet>
```

Der gesetzte Parameter `transformWsdlLocations` sorgt dafür, dass wir im Folgenden für die Angabe von Pfaden für die dynamische Erzeugung der WSDL-Datei relative Pfadangaben verwenden können. Dies ist sinnvoll, da ansonsten vor jedem Deployment der verinice-Serveranwendung Pfadangaben angepasst werden müssten.

Damit Anfragen an bestimmte URL-Muster (*Patterns*) an unser `MessageDispatcherServlet` weitergereicht werden, sind, ebenfalls in der Datei `web.xml`, zwei Servlet-Mappings hinzugefügt worden:

web.xml

```
64 <servlet-mapping>
65     <servlet-name>messageDispatcher</servlet-name>
66     <url-pattern>/sync/*</url-pattern>
67 </servlet-mapping>
68
69 <servlet-mapping>
70     <servlet-name>messageDispatcher</servlet-name>
71     <url-pattern>*.wsdl</url-pattern>
72 </servlet-mapping>
```

Dies hat zur Folge, dass alle HTTP-Requests an URLs unterhalb von `/sync/*` relativ zum Wurzelverzeichnis der Webanwendung, sämtliche Requests nach WSDL-Dateien zu unserem Message Dispatcher gelangen.

Zu guter Letzt wurde ein Filter-Mapping spezifiziert, welches dafür sorgt, dass die gleichen Authentifikationsmechanismen, die bereits für den Command Service des verinice-Servers zum Einsatz kommen, auch für unseren Webservice verwendet werden:

web.xml

```
25 <filter-mapping>
26     <filter-name>springSecurityFilterChain</filter-name>
27     <url-pattern>/sync/*</url-pattern>
28 </filter-mapping>
```

Die Konfiguration der Spring-Bean `springSecurityFilterChain` ist der Datei `/WebContent/WEB-INF/applicationContext-veriniceserver.xml` zu entnehmen. Hier wurde eine weitere Filterkette für das Pattern `/service/**` angelegt, welche die entsprechenden Filterungen vornimmt. Diese Kette wurde völlig analog zu der bereits bestehenden Kette für das Pattern `/service/**` definiert.

Damit ist die Konfiguration in der `web.xml` bereits abgeschlossen. Spring erwartet nun allerdings eine Servlet-Kontextdatei namens `messageDispatcher-servlet.xml` im Verzeichnis `/WebContent/WEB-INF/`, deren Dateiname sich aus dem Servletnamen ableitet. In dieser Datei können für das Servlet notwendige Spring-Beans konfiguriert werden.

Zunächst benötigen wir eine Instanz unseres Endpunkts, die wir als Spring-Bean ohne weitere Parameter in der Servletkontextdatei anlegen:

messageDispatcher-servlet.xml

```
15 <bean id="syncEndpoint"
16     class="sernet.gs.server.sync.SyncEndpoint"/>
```

Das oben beschriebene Mapping soll nun anhand des Wurzelelements des Payloads eingehender SOAP-Requests stattfinden. Konkret sollen alle SOAP-Requests (die das Dispatcher-Servlet aufgrund des relativen Pfades `/sync/*` dem Message Dispatcher übergeben hat) mit einem Wurzelelement `<syncRequest>` aus dem Namespace `{SYNC}` an unseren Endpunkt übergeben werden. Genau das leistet folgendes Payload-Mapping:

messageDispatcher-servlet.xml

```
5  <bean id="payloadMapping"
6      class="org.springframework.ws.server.endpoint.mapping.
           PayloadRootQNameEndpointMapping">
7      <property name="endpointMap">
8          <map>
9              <entry key="{http://www.sernet.de/sync/sync}
                   syncRequest"
10                 value-ref="syncEndpoint"/>
11         </map>
12     </property>
13 </bean>
```

Mit Hilfe der Map-Eigenschaft können theoretisch noch beliebig viele weitere Mapping-Einträge hinzugefügt werden. Jeder dieser Einträge besteht dabei aus der Zuordnung ei-

nes Payload-Wurzelelements zu einer Endpunkt-Bean. Zu beachten ist hier die etwas außergewöhnliche Art, wie der Namespace dem Elementnamen vorangestellt wird. Um nicht unnötig Verwirrung zu stiften wurde hier ausnahmsweise der vollständige Namespace-URI verwendet. Ansonsten gelten wieder die Namespace-Kürzel aus Anhang A.

Als nächstes wollen wir uns einer Bean ansehen, die dafür sorgt, dass vom Endpunkt geworfene Exceptions in entsprechende SOAP-Faults umgesetzt und an den Client zurückgeschickt werden. Dies erledigt die Klasse `SoapFaultMappingExceptionResolver` für uns:

messageDispatcher-servlet.xml

```
17 <bean id="endpointExceptionResolver"
18     class="org.springframework.ws.soap.server.endpoint.
          SoapFaultMappingExceptionResolver">
19     <property name="exceptionMappings">
20         <props>
21             <prop key="sernet.gs.ui.rcp.main.sync.
                  InternalErrorException">
22                 RECEIVER, Internal Server Error
23             </prop>
24             <prop key="sernet.gs.ui.rcp.main.sync.
                  InvalidRequestException">
25                 SENDER, Invalid message received
26             </prop>
27         </props>
28     </property>
29 </bean>
```

Zur Erinnerung: Die beiden Fehlerklassen waren einfache Unterklassen der Klasse `Exception`. Indem wir zwei unterschiedliche Klassen für interne Verarbeitungsfehler bzw. Fehler bei ungültigen Anfragen verwenden, können wir diese beiden Fehlertypen mit dem Exception Resolver in unterschiedliche SOAP-Faults umsetzen. Das Message Dispatcher Servlet sucht eigenständig nach Exception Resolvern, sodass keine weiteren Schritte nötig sind, um die Exceptions in SOAP-Faults umzusetzen.

Wir haben bislang eine Instanz unserer Endpunktklasse in Spring angelegt, sowie einen Exception Resolver, der die Fehlerbehandlung von Exceptions kapselt, die von unserem Endpunkt geworfen werden. Anhand des Payload Mappings werden gültige SOAP-Requests an den Endpunkt weitergereicht.

Es fehlt allerdings immer noch ein Mechanismus um die WSDL-Datei aus den Schemadateien zu generieren. Auch hierfür hält Spring-WS eine Klasse bereit. Hier kommt die Klasse `DynamicWsdl11Definition` zum Einsatz. Diese Klasse generiert, mit Hilfe einer weiteren Builder-Klasse die gewünschte WSDL-Datei. In unserer Servletkontext-Datei legen wir folgende Bean an:

messageDispatcher-servlet.xml

```
<bean id="sync"
    class="org.springframework.ws.wsdl.wsdl11.
       DynamicWsdl11Definition">
    <property name="builder">
        <bean class="org.springframework.ws.wsdl.wsdl11.builder.
           XsdBasedSoap11Wsdl4jDefinitionBuilder">
            <property name="schema" value="/sync.xsd"/>
            <property name="portTypeName" value="syncWS"/>
            <property name="locationUri" value="/sync/syncService"
               />
        </bean>
    </property>
</bean>
```

Der Builder vom einprägsamen Typ `XsdBasedSoap11Wsdl4jDefinitionBuilder` durchsucht die angegebene Schemadatei `sync.xsd` nach Elementdeklarationen, die auf `Request` bzw. `Response` enden. Elemente, die dieser Namenskonvention folgen, werden als Nachrichten einer Operation des Webservices aufgefasst. Als Name der Operation werden die Elementnamen ohne die jeweiligen Suffixes verwendet, in unserem Fall heißt die Operation also `sync`. Der frei wählbare Name des PortTypes in der WSDL-Datei ist mit `syncWS` angegeben worden. Viel wichtiger ist aber die Eigenschaft `locationUri`. Unter dieser URI wird später der Webservice verfügbar gemacht. Die URI ist nicht ganz frei wählbar, da sie zumindest zu unseren gewähltem Servlet-Mapping passen muss. Deshalb wurde hier `/sync/syncService` gewählt.

Wir wollen an dieser Stelle noch einmal zusammenfassen, welche Konfigurationen in den Dateien `web.xml` und `messageDispatcher-servlet.xml` vorgenommen wurden, um den Webservice verfügbar zu machen. Abbildung 3.2 stellt die Konfiguration in einer UML(-ähnlichen) Notation dar.

Anfragen an die URLs `APP-ROOT/sync/*` bzw. `APP-ROOT/*.wsdl` werden auf das neu definierte sync-Servlet gemappt, was entsprechend in der `web.xml` angegeben ist. Mit `APP-ROOT` ist hier das Wurzelverzeichnis der Server-Anwendung bezeichnet, bspw. `http://example.org/verinicеserver`.

Wichtig sind die Namenskonventionen der Elemente, die farblich hervorgehoben wurden. Spring erwartet eine Konfigurationsdatei, die sich aus dem Namen unseres Servlets, `messageDispatcher` und dem Suffix `-servlet.xml` zusammensetzt. Außerdem wurde `sync` als ID der WSDL-Definition-Bean angegeben. Deshalb antwortet das Message Dispatcher-Servlet bei Anfragen nach einer `sync.wsdl` mit der WSDL-Datei, die aus den Angaben des Builders und unserer Schemadatei generiert wird. Allerdings werden HTTP Requests nur dann überhaupt an dieses Servlet weitergeleitet, falls diese gemäß

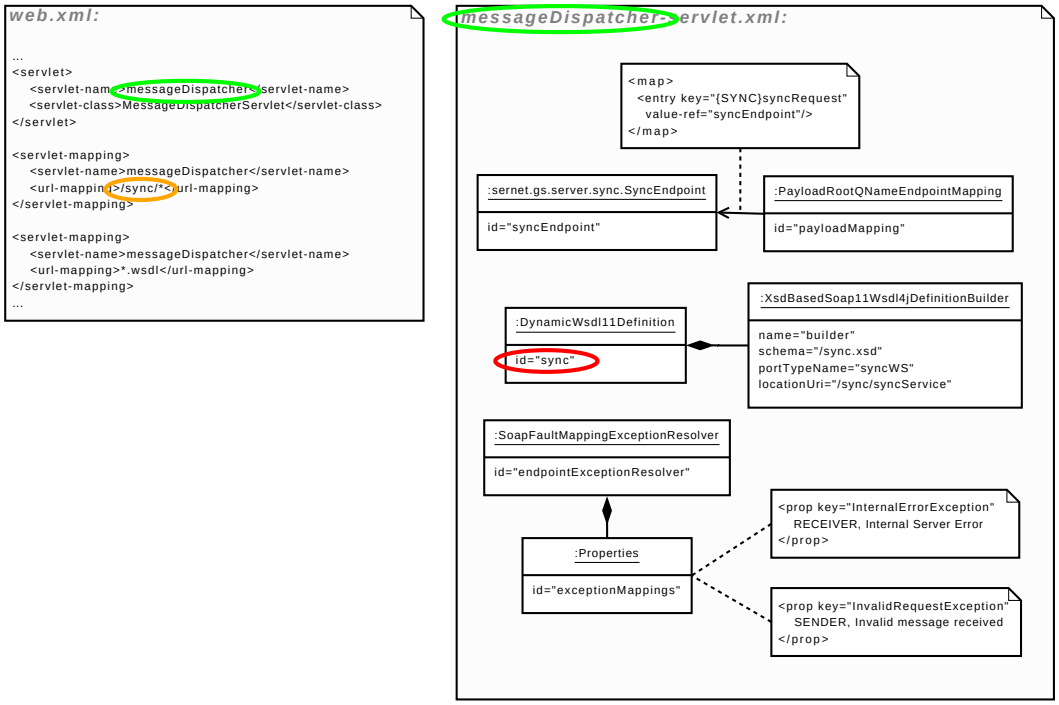


Abbildung 3.2: Verschaltung der notwendigen Spring-Beans

des Servlet-Mappings an das Unterverzeichnis `/sync/*` gerichtet sind[1]. Somit ist die generierte WSDL-Datei unter folgender URL zu finden:

http://example.org/veriniceserver/sync/sync.wsdl

Gemäß unserer Angabe der `locationUri` im WSDL-Builder steht der Webservice selbst unter der URL `APP-ROOT/sync/syncService` zur Verfügung, also, dem obigen Beispiel folgend:

http://example.org/veriniceserver/**sync/syncService**

Wie oben bereits erwähnt, musste auch bei dieser Angabe das Servlet-Mapping beachtet werden, welches HTTP-Requests an `APP-ROOT/sync/*` an unser Servlet weiterleitet.

[1] Derzeit ist es in der Tat so, dass sämtliche Anfragen nach WSDL-Dateien an das Servlet gemapped werden. Die Aussagen beziehen sich also auf den Fall, dass das zweite experimentelle Servlet-Mapping aus der web.xml entfernt wird, was sicherlich die sauberere Lösung darstellt. So können zukünftig auch weitere Servlets angelegt werden, die Anfragen nach WSDL-Dateien verarbeiten können.

Literatur

[1] Michel Tilman Dirk Riehle und Ralph Johnson. *Dynamic Object Model*. [Online, abgerufen am 25. September 2009]. URL: http://dirkriehle.com/computer-science/research/2005/plopd-5.pdf.

[2] SerNet GmbH. *verinice Grundschutztool*. [Online abgerufen am 19. September 2009]. URL: http://www.verinice.org.

[3] Bundesamt für Sicherheit in der Informationstechnik. *GSTOOL*. [Online abgerufen am 19. September 2009]. URL: https://www.bsi.bund.de/DE/Themen/weitereThemen/GSTOOL/gstool_node.html.

[4] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Kataloge*. [Online abgerufen am 25. September 2009]. URL: https://www.bsi.bund.de/gshb.

[5] International Organization for Standardization, ISO. *ISO 27001*.

[6] International Systems Audit and Control Association, ISACA. *CobiT 4.1*. [Online abgerufen am 25. September 2009. URL: http://www.isaca.org/Content/NavigationMenu/Members_and_Leaders1/COBIT6/Obtain_COBIT/Obtain_COBIT.htm.

[7] Eviware Software. *soapUI*. [Online abgerufen am 25. September 2009]. URL: http://www.soapui.org.

[8] Craig Walls und Ryan Breidenbach. *Spring in action*. Greenwich, CT, USA: Manning Publications Co., 2007. ISBN: 9781933988139.

Abbildungsverzeichnis

A Liste der Namespace-Kürzel

In den Quelltext-Beispielen wurden wiederholt Kürzel der Form `{Kürzelname}` anstelle vollständiger Namespace-URIs verwendet, um die Lesbarkeit zu erhöhen. Diese Kürzel müssen durch die folgenden URIs ersetzt werden.

{XSI} http://www.w3.org/2001/XMLSchema-instance

{SOAP} http://schemas.xmlsoap.org/soap/envelope/

{SYNC} http://www.sernet.de/sync/sync

{DATA} http://www.sernet.de/sync/data

{MAPPING} http://www.sernet.de/sync/mapping

B Neue und geänderte Pakete und Quelltextdateien

An dieser Stelle folgt eine tabellarische Auflistung aller zur Implementierung des Webservices neu angelegten und geänderten Pakete und Quelltextdateien. Dies bezieht sich auf die Version v0.8 aus den verinice-Quellen.

Dabei steht ein „+" für eine neu angelegte Datei, ein „e" für verändert (edited).

sernet.gs.server		
+	`/src/sernet/gs/server/sync`	Paket für die Implementierung des Endpunktes.
+	`/src/sernet/gs/server/sync/SyncEndpoint.java`	Klasse SyncEndpoint, die von AbstractJDomPayloadEndpoint erbt und den Webservice implementiert.
e	`/WebContent/WEB-INF/applicationContext-veriniceserver.xml`	Bean springSecurityFilterChain: Pfad /sync/** hinzugefügt.
+	`/WebContent/WEB-INF/messageDispatcher-servlet.xml`	Beans payloadMapping, syncEndpoint, endpointExceptionResolver und sync definiert.
e	`/WebContent/WEB-INF/web.xml`	Servlet „messageDispatcher" und Servlet-Mapping, sowie Filter-Mapping für /sync/* hinzugefügt.
+	`/WebContent/data.xsd`	XML-Schema für Inventardaten
+	`/WebContent/mapping.xsd`	XML-Schema für Mapping-Daten
+	`/WebContent/sync.xsd`	XML-Schema für SOAP-Nachrichten, bindet data.xsd und mapping.xsd ein

sernet.gs.ui.rcp.main		
e	`/src/CnATreeElement_derby.hbm.xml`	<hibernate-mapping> für CnATreeElement um sourceId und extId ergänzt.
e	`/src/CnATreeElement.hbm.xml`	<hibernate-mapping> für CnATreeElement um sourceId und extId ergänzt.
e	`/src/sernet/gs/ui/rcp/main/common/model/CnATreeElement.java`	Attribute sourceId und extId mit gettern und settern hinzugefügt.
e	`/src/sernet/gs/ui/rcp/main/service/crudcommands/CreateElement.java`	containerDAO.merge() statt containerDAO.reload() in Zeile 61 bzw. 62
+	`/src/sernet/gs/ui/rcp/main/service/crudcommands/SyncDeleteCommand.java`	Command zum Entfernen von Objekten gemäß delete-Flag
+	`/src/sernet/gs/ui/rcp/main/service/crudcommands/SyncInsertUpdateCommand.java`	Command zum Einfügen und Aktualisieren von Objekten gemäß update / delete Flags
+	`/src/sernet/gs/ui/rcp/main/service/crudcommands/LoadCnAElementByExternalID`	Command zur Abfrage eines verinice-Objektes anhand von `sourceId` und `extId`
+	`/src/sernet/gs/ui/rcp/main/service/crudcommands/LoadCnAElementsBySourceID`	Command zur Abfrage mehrerer Objekte anhand ihrer `sourceId`.
+	`/src/sernet/gs/ui/rcp/main/sync`	Paket für allgemeine Util-Klassen für die Synchronisation
+	`/src/sernet/gs/ui/rcp/main/sync/InternalErrorException`	Exception-Klasse für interne Serverfehler, wird vom endpointExceptionResolver gefangen.
+	`/src/sernet/gs/ui/rcp/main/sync/InvalidRequestException`	Exception-Klasse für ungültige Client-Anfragen, wird vom endpointExceptionResolver gefangen.
+	`/src/sernet/gs/ui/rcp/main/sync/SyncNamespaceUtil`	Util-Klasse, die die Namespaces sync, data und mapping zur Verfügung stellt.