midPoint による
実用的なアイデンティティ管理

著者：RADOVAN SEMANČÍK ほか

第 1.1 版
2018 年 4 月

midPoint による
実用的なアイデンティティ管理

Radovan Semančík ほか
Evolveum 社

改訂版：1.1
発行日：2018 年 4 月
本書に該当する midPoint のバージョン：3.7.1

本書の主たるスポンサー：

目次

第 1 章：　はじめに

家を出ていくのは危険だぞ、フロド。外へ踏み出しても、足元を定めなければどこへ連れて行かれるか知れたものじゃない。

ービルボ・バギンズ

（J.R.R.トールキン『指輪物語』）

数年前、我々はあるプロジェクトを立ち上げた。そうしなければならなかったのだ。当時は市場やビジネスといったことはさほど分かっておらず、ただひたすら技術に力を注いだ。そしてプロジェクトを実直に進めていった。山あり谷ありではあったが、そこには常に純粋なエンジニアとしての情熱があった。やがて努力は実を結び、今や他に類のない製品となった。それが midPoint である。

midPoint はアイデンティティの管理およびガバナンスのシステムである。我々は本システムを一から構築した。midPointは豊富な機能を備えた包括的システムであり、アイデンティティのライフサイクル全体と管理を扱うほか、アイデンティティのガバナンスおよびコンプライアンスの一部も扱っている。本システムにより新入社員のアカウント作成プロセスをスピードアップさせることができる。従業員の契約終了後は自動でアカウントを無効にする。従業員、パートナー、エージェント、受託業者、顧客、学生など、ほぼすべてのタイプのユーザーに対するロール（役割）および特権のアサインを管理する。ポリシーが常に保守されるよう監視を行う。アクセスレビューのプロセスを管理する。アイデンティティデータをもとに監査および報告を行う。さらに最近のバージョンではコンプライアンス分野に取り組んでいる。つまり新ポリシーおよび変更後ポリシーの管理、設定内容に関するポリシー遵守状況の判断、円滑なポリシー導入の調整である。

midPoint はその包括さゆえ競合製品はごくわずかであるが、それらに勝る決定的なメリットが一つある。それは midPoint が完全にオープンソースだということである。オープンソースは midPoint の基本姿勢である。現代の高品質ソフトウェア開発においてオープンソースは一つの重要な側面である。しかし midPoint コミュニティ、すなわちパートナーや貢献者や支援者、そして実際に midPoint で作業するすべてのエンジニアにとって、オープンソース方式は絶対に必要である。オープンソース戦略であることはすなわち、どのエンジニアも midPoint を完全に理解できるということである。また、必要に応じて変更が可能であり、特に midPoint 開発の持続性を確保できるよう迅速に問題を修正できるということでもある。midPoint と共に何年も過ごした今となっては、オープンソースでないアイデンティティ技術を使用することなどとても想像できない。

midPoint を構築したチームの大半が、2000 年代初頭からアイデンティティ管理のデプロイに携わってきた。当時は「アイデンティティ管理とアクセス管理（Identity and Access Management：IAM）」という言葉すらなかった。我々はキャリアを通じて数多くの IAM ソリューションを目にしてきたが、その大半がアイデンティティ管理（Identity Management：

IDM）システムを中心としたものであった。それが我々の視点によるものか、一般的なルールなのかははっきりとはわからない。その判断は読者にお任せする。とにかく我々が確信をもっていることは、midPoint は真に役立つツールだということである。正しく使用すれば奇跡を起こしてくれる。これこそが本書で伝えたいことにほかならない。つまり midPoint を正しく活用してアイデンティティ管理における実用的なソリューションを構築するということである。本書では実用的な IDM ソリューションのデプロイ*方法*について説明している。しかしそもそも*なぜ*それを行うのかということも説明する。また、機能や構成の種類だけではなく、モチベーションや根本原則についても説明する。少なくとも根本原則を理解することは、IDM 製品の構造を学ぶことと同じくらい重要なことである。本来、本書には製品が機能するとどう動作するのかということが記載されるものだろう。しかし我々は制限事項、デメリット、落とし穴についても述べようと思う。特に未開発の分野で新たなソリューションを設計するようなときは、しばしば機能よりも制限事項の方がはるかに重要だからだ。

第 1 章では IAM の基本的な考え方について説明する。ごく一般的な内容で midPoint には一切触れていないため、すでに IAM に詳しいのであれば当章は飛ばしてもらっても構わない。また、じっと待ってなどいられない、すぐにでも midPoint にとりかかりたいという人も同様である（いずれにせよそうするだろう？）。しかしもしそうなら、のちほど時間を見つけて第 1 章に戻ってきてほしい。当章にはより幅広い視点で midPoint を捉えるための重要な情報が盛り込まれているからだ。完全な IAM ソリューションを構築するのであればそのような情報が必要となろう。

第 2 章では midPoint の全体像について説明する。外側から midPoint がどう見えるかを示す。一般に midPoint がどのように使用され、どんな挙動を行うかについて説明する。当章の目的は読者に midPoint の働きおよび基本原則を知っていただくことである。midPoint の*使用*方法について説明する。

第 3 章では midPoint のデプロイと構成に関する基本的な考えを説明する。midPoint のインストールについて案内する。また、特定のデプロイのニーズに合わせて midPoint を*カスタマイズ*する方法も述べる。ただし midPoint のカスタマイズは非常に複雑であり、当章で説明する内容は基本的な原則のみである。詳細まで説明すれば本書のページの大部分を割かなければならないだろう。

第 4 章では*リソース*および*マッピング*の概念を説明する。これはアイデンティティ管理になくてはならない重要な要素である。当章では非常に基本的な midPoint デプロイの作成方法、ターゲットシステムの接続方法、データのマッピングおよび変換方法について説明する。

第 5 章は*同期*について取り上げる。同期を行う主な目的は人事システム等のリソースから midPoint にデータを取得することである。しかし midPoint の同期はそれよりもはるかに強力である。また、マッピングやデルタ等、midPoint の根底にある原則についてもさらに詳しく説明する。

第 6 章では midPoint の開発プロセスと全体的な手法についておおまかに説明する。また、

midPoint 開発への資金援助の仕組みと、midPoint サブスクリプション方式の仕組みについても説明する。

これ以降の章はまだ執筆されていない。つまりロールベースのアクセスコントロールやポリシーといった高度な内容はまだ掲載されていない。本書はまだ初期バージョンにすぎず、midPoint と同様少しずつ加筆修正されていく。すぐれた書物を執筆することはそれ自体が大作業であり、かなりの時間がかかる。しかし midPoint コミュニティにはこのような書物が明らかに必要である。だからこそ我々は本書の完成まで待たないことにした。つまり、十分完成された章から継続的に発行していこうと決めたのである。何もないよりはあったほうがよいというのが一般的である。どうかご辛抱いただきたい。いつかは完成版ができあがるだろう。そしていつものごとく、あなた方からの支援、貢献、援助によってこれらの作業はかなりスピードアップするだろう。

midPoint の開発者、貢献者、支援者の全員に感謝申し上げる。この数年間、実に多くの人々が midPoint に携わってくれた。そして midPoint を推し進めてくれた。しかしとりわけ、midPoint が未熟だった頃から支えてくれていた人たち、そして今日現在も支えていてくれる人たちに感謝したい。Katka Stanovská 氏、Katka Valaliková 氏、Igor Farinič 氏、Ivan Noris 氏、Vilo Repáň 氏、Pavol Mederly 氏、Radovan Semančík 氏に感謝申し上げる。彼らは当時から支えてくれたことはもちろん、今も midPoint を未来へと導く力となってくれている。

本書の記載内容はすべて執筆陣の意見である。我々は IDM ベンダーでもある Evolveum 社の社員である。客観性を保とうと懸命に努力してきた。しかしどんなに努力しようと、やはりどうしても変えられない考え方もある。そのため我々の意見は少し偏った部分もあるかも知れない。ただ、一切の偏見をなくしエンジニアリングの正しい慣習に従おうと誠実に努力している。この件に関しては、本書の読者こそが審判員であり陪審団である。我々が成功したかどうかは、あなた方読者にご判断いただきたい。そのために必要な情報はすべて無料で入手できる。midPoint のすべての文書およびソースコードと同様、本書もまた自由に閲覧できるようになっている。我々は一切隠し事をしていない。他のベンダーと違い、隠したいこともなければその必要もないからだ。

ようなことはあってはならない。また商業研修にて本書を使用することもできない。どんな形であれ、利益を生むような形で本書を使用することは認められていない。もしそのような形で本書を利用する必要があれば、Evolveum までご連絡いただきたい。特別なライセンスを供与する。そこで得られたライセンス料は midPoint、および、とりわけ midPoint の文書強化にあてられる。これが必要であることはご理解いただけるだろう。

本書の文章や図の作成に取り組んできたのは以下の人物である。

- Radovan Semančík（著者およびメンテナー）
- Veronika Kolpaščiková（イラスト、修正担当）

しかし他にも大勢の人々の力があってこそ、本著作物を実現させることができた。midPoint の開発者、貢献者、アナリスト、デプロイエンジニア、スペシャリスト、ゼネラリスト、理論科学者、実地技術者、技術スタッフ、ビジネス関係者、Evolveum の人員、パートナー会社で働く人々、家族、友人、現在と過去の何世代にもわたるすべてのエンジニアおよび科学者たちである。我々はまさに巨人の肩の上に立っている。

第 2 章： アイデンティティ管理とアクセス管理を理解する

自分が理解できないことの発見が知識の始まりとなる。

– フランク・ハーバート

アイデンティティ管理とアクセス管理とは何か？その答えは簡単でもあり実に複雑でもある。簡単に答えるなら、アイデンティティ管理とアクセス管理（Identity and access management：IAM）とはサイバースペースでアイデンティティを扱う一連の情報技術のことである。一方複雑な部分を答えたものが、本書の残りの部分である。

本書は主に*エンタープライズアイデンティティ管理とアクセス管理*について取り扱う。これはエンタープライズ（企業）、金融機関、政府機関、大学、医療など、より大規模な組織を対象としたアイデンティティ管理とアクセス管理であり、その重点は従業員、受託業者、顧客、パートナー、学生、および組織に協力するその他の人々を管理することにある。ただ、その仕組みや原則の多くは非エンタープライズ環境にも適用することができる。

アイデンティティ管理とアクセス管理の話は情報セキュリティから始まる。セキュリティ要件とはユーザーの認証および認可の必要性を定めたものである。認証とは、ユーザーが本当に当該個人であると主張する人物であるかどうかをコンピュータが確認する仕組みのことである。そして認可とは、これに関連してユーザーに特定のアクションを許可するか否かをコンピュータが判断する仕組みのことである。ほぼすべてのコンピュータシステムが、何らかの認証手段および認可手段を備えている。

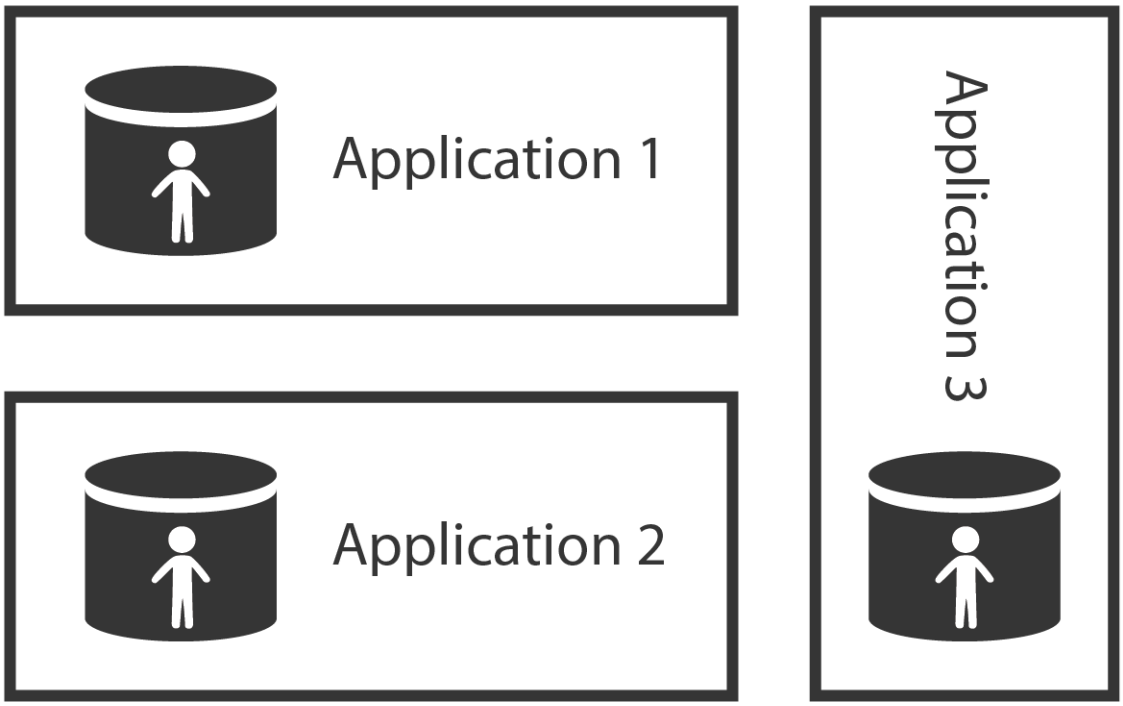
おそらくもっとも広く利用されている認証方式がパスワードをベースとした「ログイン」手続きであろう。ユーザーは自分の ID とパスワードを入力する。するとコンピュータはそのパスワードが有効か確認する。この手続きを行うため、コンピュータは有効なすべてのユーザーおよびパスワードが格納されたデータベースにアクセスする必要がある。初期のスタンドアロン型の情報システムにはサイバースペースから切り離された独自のデータベースがあり、データのメンテナンスは手動で行われていた。しかしコンピュータネットワーキングの到来によってすべてが変わった。ユーザーは多くのシステムにアクセスできるようになり、システムそのものも相互に接続されるようになった。各システム内の孤立したユーザーデータベースを管理することはもはやほとんど無意味となった。そしてここから本当のデジタルアイデンティティのストーリーが始まるのである。

ディレクトリサービスおよびその他のユーザーデータベース

アイデンティティ管理では、個人に関する情報を含むデータレコードが中心的な概念である。この概念にはユーザープロファイル、ペルソナ、ユーザーレコード、デジタルアイデンティティなど多くの呼称がある。アイデンティティ管理においては*ユーザーアカウント*が最も一般的な呼称である。一般にアカウントは氏名など一連の属性データを使用して実世界の個人を示す情報を有している。しかしおそらくもっとも重要なのはアカウントが作

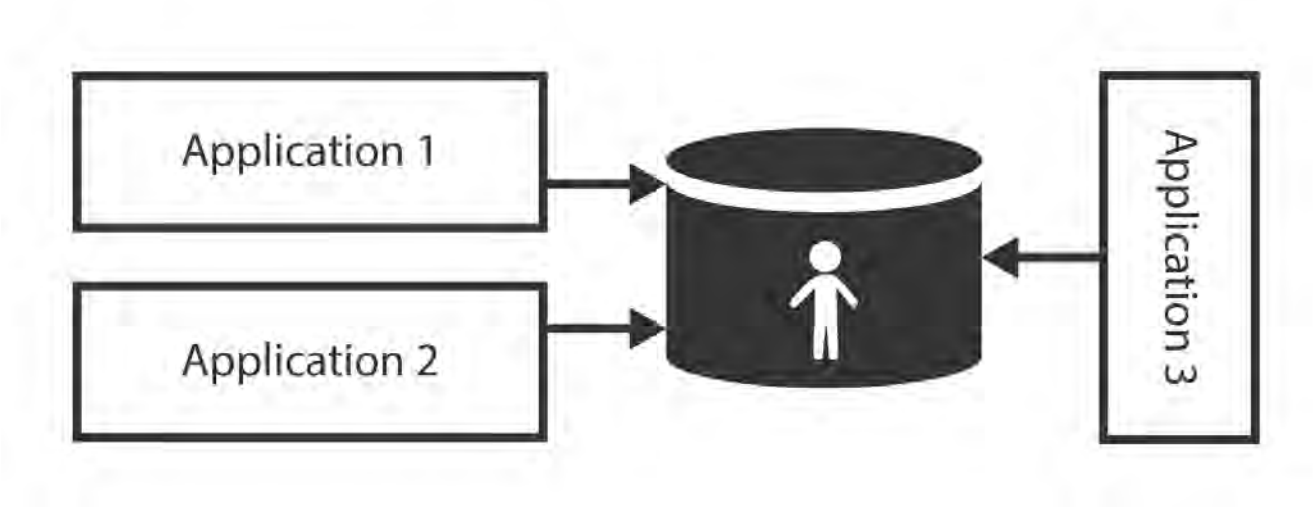
成された情報システムの操作に関する技術情報であろう。例えば、ユーザーのホームディレクトリの場所等の操作パラメータ、グループやロールメンバーシップ等の様々なアクセス権限情報、システムリソース制限といった情報である。ユーザーアカウントは関係データベースレコードから構造化データベース、そして半構造化テキストファイルに至るまで、その形式は多種多様である。しかしそのレコードの具体的な保存方法および処理方法の如何にかかわらず、*アカウント*は IAM 分野における最も重要な概念の一つであることは確かである。そしてアカウントを格納するデータベースも同様である。なぜなら、アカウント、すなわちデータレコードは必ずどこかに保存しなければならないからだ。

アカウントデータベースはユーザーレコードと同様に多種多様である。かつてはほとんどのユーザーデータベースが、システム自体で使用されているものと同じデータベース技術を使用した、モノリシックな情報システムに絶対必要な要素として実装されていた。これは当然の選択であり、今でもたいへん普及し続けている。そのため多くのアカウントが関係データベーステーブルや似たようなアプリケーションデータストアに保存されている。



一般にアプリケーションデータストアはアプリケーションと強く結びついているため、そのようなデータベースに格納されたアカウントを他のアプリケーションと共有するのは難しい。しかし組織内でのアカウントデータの共有は大いに望まれている。データベースごとに切り離してアカウントデータを管理してもほとんど意味はない。それらのほとんどが複数のアプリケーションで重複しているならなおさらである。そこで、多くのアプリケーションで共有できるアカウントデータベースをデプロイしたいという強いモチベーションが出てくる。

ディレクトリサーバーの目的はデータストレージを共有することである。通常、アプリケーションデータベースは独自のプロトコルを使用するが、ディレクトリサーバーには標準化されたプロトコルが実装される。データベースが完全にカスタムデータモデル向けに構築されるのに対し、ディレクトリサーバーは一般に標準化データモデルを拡張して相互運用性を向上させる。多くの場合データベースは重量があり性能向上には費用もかかるが、ディレクトリサーバーは軽量かつ大規模な拡張性を持つ設計となっている。こうしたことからディレクトリサーバーは共有アカウントデータベースとして申し分ない選択肢となっている。



共有アイデンティティストアによってユーザー管理は容易になってきている。アカウントはーか所で作成、管理すればよい。認証はアプリケーションごとに独立して行われる。しかしアプリケーションは共有ストアから同じクレデンシャルを使用するため、ユーザーは接続するアプリケーションすべてに同じパスワードを使用してもよい。

ライトウェイト ディレクトリ アクセス プロトコル（Lightweight Directory Access Protocol：LDAP）はディレクトリサービスにアクセスするための標準プロトコルである。インターネット標準による古いプロトコルであり、そのルーツははるか1980年代までさかのぼる。ただ古いとはいえ、廃れるどころかむしろその逆である。これは分散型共有データベースを強力にサポートすべく設計された、非常に有効なバイナリープロトコルである。シンプルな小さな一連の操作が明確に定義されている。このプロトコルが示す操作およびデータモデルによって、データレプリケーションとディレクトリサーバーの水平方向の拡張性は常に効率的になる。また読み取り操作における低レイテンシおよび高スループットを実現する。ディレクトリサーバーインスタンスの水平方向の拡張性および相対的な独立性によってディレクトリシステムの可用性が向上する。こうしたメリットは書き込み操作が遅くなるという犠牲のもとに得られるのが一般的である。しかしアイデンティティデータはしばしば読み取られることはあっても変更されることはめったにないため、大いに許容可能なトレードオフである。LDAPベースのディレクトリサーバーが、昔も今もアイデンティティデータのデータベースとして最も普及しているのはそのためである。

LDAPはIAM分野において確立された数少ない貴重な標準の一つである。そしてほぼすべての組織がLDAP対応のデータストアにアイデンティティを保存している。したがって本

書では何度も LDAP プロトコルに触れることがあるだろう。

共有ディレクトリサーバーにもとづくアイデンティティ管理ソリューションは、シンプルで実に費用効果も高い。我々は長年同じことを助言してきた。「もしすべてのアプリケーションを一台の LDAP サーバーに接続できるなら、あまり考えすぎないでとにかくやりなさい」と。問題は、これが一般には非常にシンプルなシステムにしか通用しないということである。

ディレクトリサーバーはデータベース

ディレクトリサーバーはまさに情報を格納するデータベースであり、それ以上のものではない。ディレクトリサーバーへのアクセスに使用するプロトコルおよび API はデータベースのインターフェイスとして設計される。つまり、ディレクトリサーバーはデータの格納、検索、取得に最適ということである。アカウント内のデータにはエンタイトルメント情報（アクセス権限、グループ、ロール等）が含まれることはあっても、アイデンティティストアはそれらの評価にはあまり向いていない。つまり、ディレクトリサーバーはアカウントがどんなアクセス権限を持っているかという情報は提供できても、特定の操作を許可あるいは拒否する*判断*ができるようには作られていないのである。他にも問題はある。ディレクトリサーバーには*ユーザーセッション*に関する情報は含まれない。つまり、ユーザーが現在ログイン中かどうかわからないということである。多くのディレクトリサーバーがベーシック認証、さらには認可にも使用されている。しかしディレクトリはそのように設計されなかったため、非常に基本的な機能しか提供しない。認証および認可に対応する機能を部分的に実現するようなプラグインや拡張機能はある。しかしそれは根本的な設計原則を変えるものではない。ディレクトリサーバーはデータベースであり、認証サーバーでもなければ認可サーバーでもないのである。

単一ディレクトリサーバーの神話

共有ディレクトリサーバーがあればユーザー管理はより簡単になる。しかしこれは完全なソリューションではなく、この手法には重大な限界がある。情報システムがそれぞれ異質であるため、必要なすべてのデータを単一のディレクトリシステムへ格納することはほとんど不可能だということである。

単一で整合性のとれた情報ソースがないことは明らかに問題である。一般的には単一ユーザーに対し複数の情報ソースが存在している。例えば、人事システムはユーザーをその企業に存在させ、従業員 ID をアサインする権威あるシステムである。経営情報システムにはユーザーロールを判断する責任がある（例：プロジェクト型組織構造にて）。在庫管理システムにはユーザーに電話番号をアサインする責任がある。グループウェアシステムはユーザーのＥメールアドレスといった電子的な連絡方法のデータのソースとして権威あるシステムである。一般に、単一ユーザーについて権威のある情報を提供するシステムは 2〜20 種類存在する。そのため、簡単にデータをディレクトリシステムにフィードし管理できる方法はない。

複雑なアプリケーションはほぼすべてローカルユーザーデータベースが必要である。効率よくオペレーションを行うにはユーザーレコードのコピーを各アプリケーションのデータベースに格納しなければならない。例えば大規模な請求システムでは外部データとの効率

的な連携ができない（例：関係データベース *join（結合）*のため）。よってディレクトリサーバーをデプロイしたとしても、こうしたアプリケーションにはローカルコピーの保守が必要となる。このコピーとディレクトリデータとで同期を保つことは大したタスクではないと思われるかもしれないが、実はそうではない。また、外部データに一切アクセスできないようなレガシーシステムもある（例：LDAP プロトコルに対応していないなど）

いくつかのサービスはデータベースレコードよりもステート（状態）を保つ必要がある。例えばファイルサーバーは一般にユーザー用のホームディレクトリを作成する。通常、ステートの自動作成は要求に応じて行うことができるが（例：ユーザーの初回ログオン時にユーザーディレクトリを作成）、ステートの変更や削除ははるかに難しい。ディレクトリサーバーはこれを行わない。

しかしおそらくもっとも悩ましい問題は、アクセス制御ポリシーが複雑であることだろう。ロール名やアクセス制御属性がどのシステムでも同じ意味を持つとはかぎらない。一般に各システムはそれぞれ異なる認可アルゴリズムを採用しており、それらは互換性がない。この問題はアプリケーションごとのアクセス制御属性で解決できるものの、こうした属性の管理が些細なものであることはめったにない。また、各アプリケーションが一連の属性を独自に有してアクセス制御ポリシーを管理しているため、一元化されたディレクトリによるメリットはほとんどない。それなら属性はアプリケーション自体の中に存在したほうがよいだろう。たいていのデプロイは、まさにこのような結末になる。ディレクトリサーバーには RBAC ロールを大まかに決めたグループのみが含まれる。アクセス制御ポリシーおよびきめの細かい認可はやはりアプリケーションにて管理される。

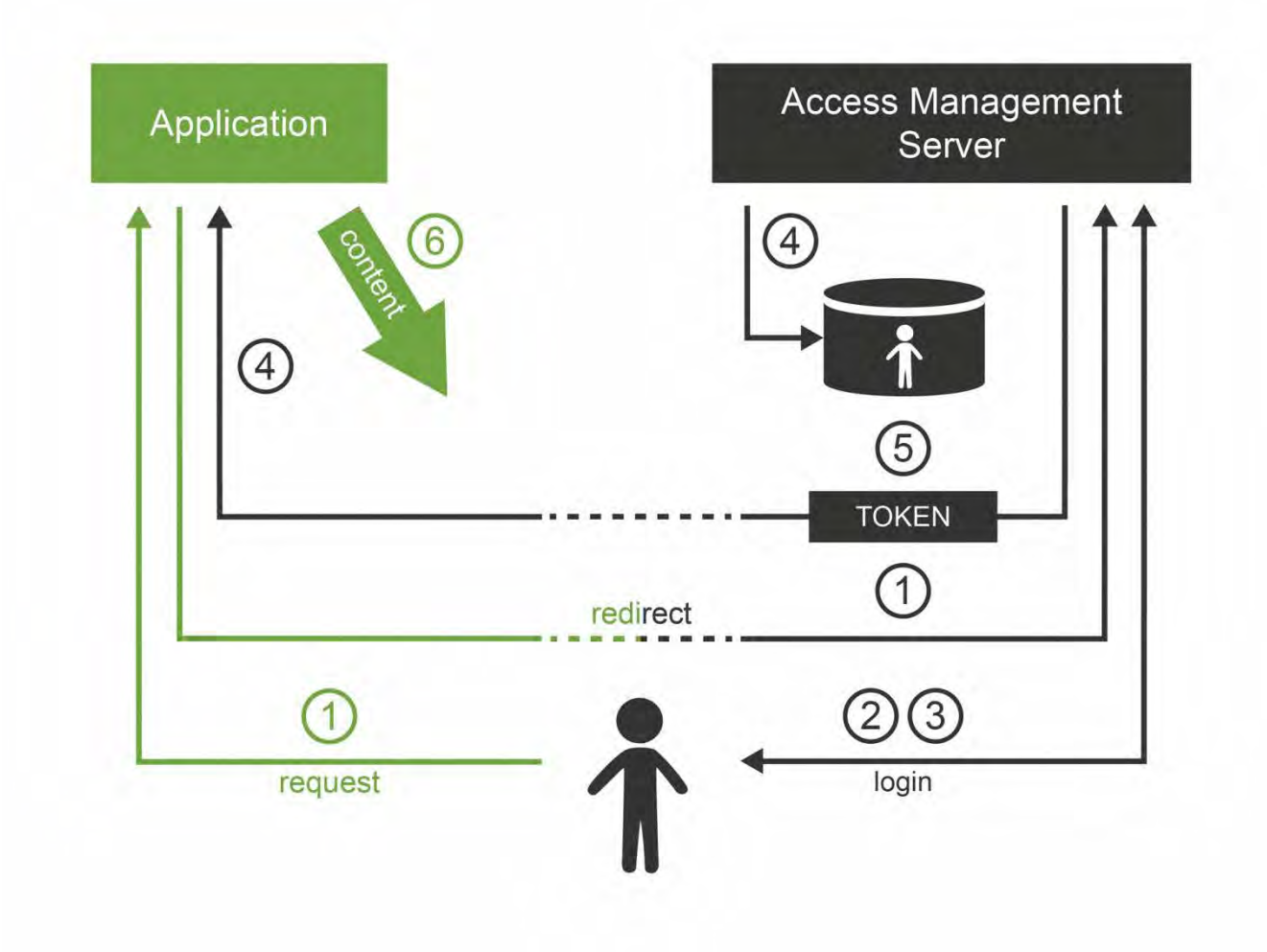
単一ディレクトリという手法は、非常にシンプルな環境またはほぼ全体が同機種の環境でしか実現できない。それ以外の環境はすべて別のアイデンティティ管理技術が必要となる。しかしこれはディレクトリサーバーが役に立たないと言っているのではない。むしろその逆で、正しく利用すれば非常に有効である。ただ単体では利用できない。完全なソリューションを構築するにはもっとコンポーネントが必要である。

アクセス管理

ディレクトリシステムは複雑な認証を扱えるような作りにはなっていないが、*アクセス管理（access management：AM）*システムはまさにそれを扱うべく構築されたものである。AM システムはあらゆる種類の認証およびアクセスを扱うほか、認可も一部扱う。基本的にどの AM システムも原則は同じである。

1. AM システムはユーザーとターゲットシステムの間に位置する。これは様々な仕組みで実現できるが、もっとも一般的なのは既存のセッションがない場合、アプリケーション自体でユーザーを AM システムへとリダイレクトする仕組みである。
2. AM システムはユーザーにユーザー名とパスワードの入力を求め、証明書を要求し課題を作成してレスポンスを求めるか、あるいはそれ以外の方法で認証手続きを開始する。
3. ユーザーがクレデンシャルを入力する。

4. AMシステムは入力されたクレデンシャルの妥当性を確認しアクセスポリシーを評価する。
5. アクセスが許可されると、AM システムはユーザーをアプリケーションにリダイレクトする。多くの場合、このリダイレクトにはアクセストークン（ユーザーが認証されたことをアプリケーションに知らせる小さな情報）が含まれている。
6. アプリケーションはトークンを検証したのち、ローカルセッションを作成しアクセスを許可する。



この手続きが終わると、ユーザーは通常どおりそのアプリケーションを使用して作業を行う。つまり最初のアクセスのみ AM サーバーを経由する。これは AM システムのパフォーマンスおよびサイズ変更において重要である。

アプリケーションは AM システムと統合するコードを提供するだけでよい。それ以外にアプリケーションが提供しなければならない認証コードは一切ない。パスワード入力を求めるのは AM システムであってアプリケーションではない。この点が LDAP ベースの認証との違いである。LDAP ではアプリケーションがパスワードの入力を求める。AM システムの場合においては AM サーバーがすべてを行う。ユーザーがどうやって認証されたのかすらわからないアプリケーションも多い。アプリケーションにとって必要な情報は、ユーザーが認証されたということ、そしてその認証は十分なものであったということだけである。

この特長がシステム全体に非常に望ましい柔軟性をもたらしてくれる。認証の仕組みはアプリケーションを中断させることなくいつでも変更できる。我々は今、パスワードの仕組みがより強固な認証の仕組みへと移行する時代に生きている。AM ベースの手法がもたらす柔軟性は、この移行に際し重要な役割を果たすであろう。

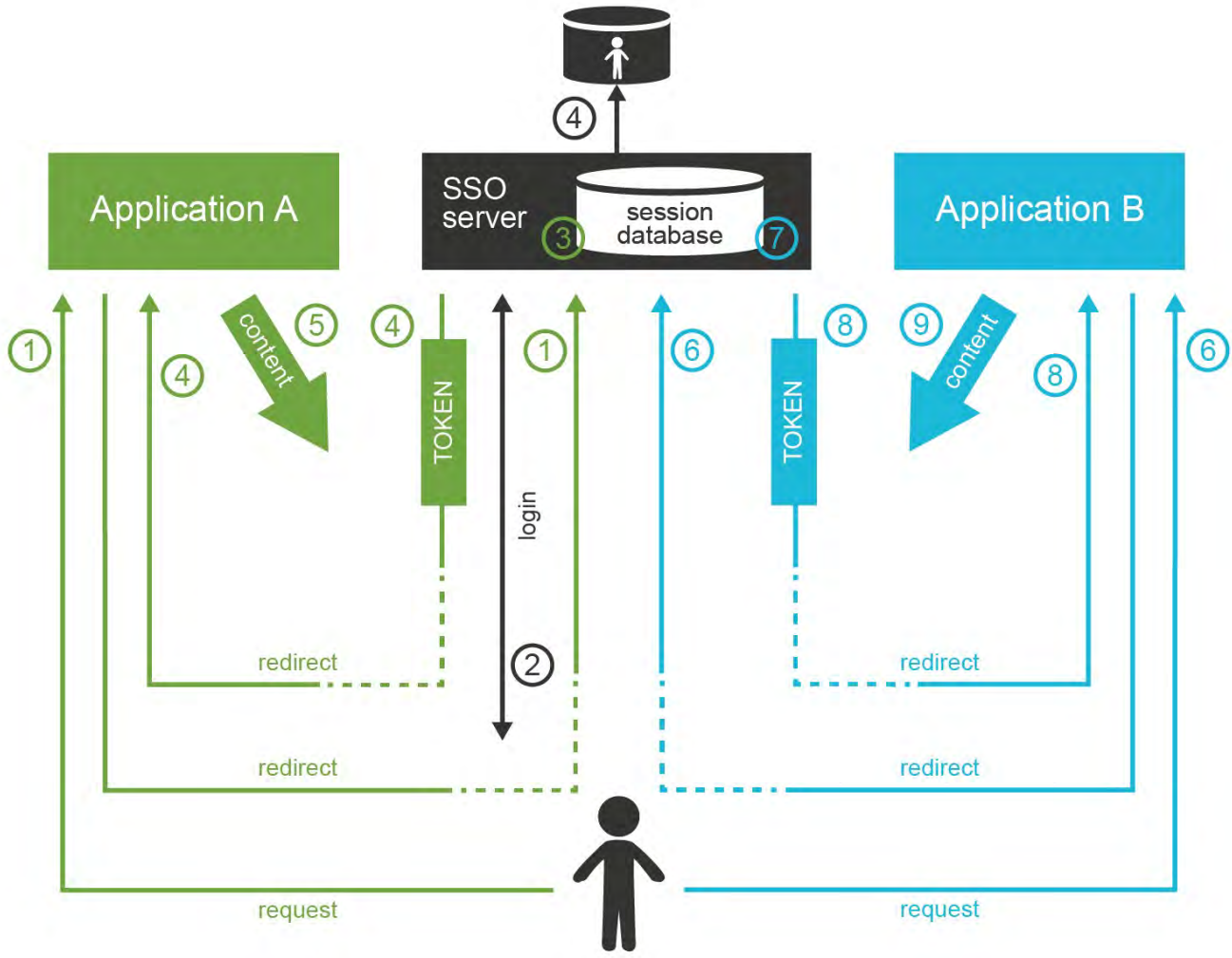
Web シングルサインオン

シングルサインオン（Single Sign-On：SSO）システムは、ユーザーが一度認証を受けるだけで複数のシステムへアクセスできる仕組みである。通常、認証はユーザーとの直接的なやりとりに依存するため、さまざまな仕組みを活用した SSO 機能を実装しているシステムが多い。

また Web アプリケーション用の SSO システムも数多くあるが、こうしたシステムはすべて同じ基本原則を使用しているようだ。Web SSO は上記の一般的なアクセス管理原則にもとづいている。

1. アプリケーション A がユーザーを AM サーバー（SSO サーバー）へとリダイレクトする。
2. AM サーバーがユーザーを認証する。
3. AM サーバーがユーザーブラウザとセッション（SSO セッション）を確立する。これが SSO の仕組みで要となる部分である。
4. ユーザーはアプリケーション A へとリダイレクトされる。通常はアプリケーション A がユーザーとローカルセッションを確立する。
5. ユーザーがアプリケーション A とやりとりする。
6. ユーザーがアプリケーション B にアクセスしようとすると、アプリケーション B はユーザーを AM サーバーへとリダイレクトする。
7. AM サーバーが SSO セッションの存在を確認する。ユーザーはすでに AM サーバーによって認証されているため、有効な SSO セッションが存在している。
8. AM サーバーはユーザーの再認証を行う必要はなく、速やかに当ユーザーをアプリケーション B へとリダイレクトする。
9. アプリケーション B はユーザーとローカルセッションを確立し通常通り続行される。

アプリケーション B へのアクセス時、ユーザーはたいてい自身がリダイレクトされたことにすら気付かない。リダイレクトの往復の間に対話がないうえ、AM サーバーの処理が非常に高速だからだ。それはまるでユーザーがアプリケーション B にずっとログインしていたかのようである。



アクセス管理における認可

ユーザーのアプリケーションへのアクセス要求は直接または間接的にアクセス管理サーバーを経由する。よって理論上はアクセス管理サーバーが当要求を分析し、ユーザー要求を認可するか否か評価できる。しかし実態はもっと複雑である。

AM サーバーが受け取るのはアプリケーションにアクセスしたいという最初の要求だけである。すべての要求を受け取っていてはパフォーマンスに影響が出るためだ。最初の要求ののち、アプリケーションがローカルセッションを確立し、AM サーバーとのコミュニケーションなしに操作を続行する。よって AM サーバーは最初の要求時に認可を評価できるだけである。つまり、非常にきめの粗い認可判断を評価することしかできないということである。実際のところ、AM サーバーが行える認可判断は、特定のユーザーが特定のアプリケーションのすべての部分にアクセスできるか、またはそのアプリケーションには一切アクセスできないか、であることが一般的だ。AM サーバー自体はそれ以上きめの細かい判断はできない。

AM システムの中には、アプリケーションにデプロイでき、よりきめ細やかな認可判断を実行するエージェントを提供するものもある。このようなエージェントはしばしば HTTP 通信に依存しユーザーがアクセスしている URL をもとに判断を行っている。この手法は 1990 年代であればうまく機能するだろう。しかし AJAX、シングルページ Web アプリケーション、モバイルアプリケーションの時代においては適用できる対象が非常に限られてしまう。

高度なアプリケーションはしばしば、要求またはユーザープロファイルには一切利用できないコンテキストをもとに認可判断を行う必要がある。たとえばe-バンキングのアプリケーションの場合、同日に実施された過去の一連のトランザクションをもとにして、新たなトランザクションを許可もしくは拒否することはできない。一般にはすべての認可情報をユーザープロファイルに同期させることができるものの、たいていそれは望まれていない。そのような情報を常に更新し整合性を維持することは大きな負担になるからだ。

AM システムにはしばしば、すべてのアプリケーションの認可を統合するとともに組織全体のセキュリティ方針を一元管理するという約束が付いてくる。しかし残念ながらそのような約束はほぼ例外なく事実と違う。理論上は、AM システムは認可ステートメントの一部を評価し実行する。これはデモ中や、非常にシンプルなデプロイならばうまくいくだろう。しかし複雑で実用的なデプロイ環境の場合、この機能は極端に制限されてしまう。認可判断のほとんどは個々のアプリケーションでそれぞれ実行されており、完全に AM システムの手の届かないところだからだ。

SAML および OpenID Connect

多くのアクセス管理システムが、アプリケーションおよびエージェントと通信を行うため独自のプロトコルを使用している。このことは、特にインターネット環境で AM 原則が使用される場合には、明らかに相互運用性の問題となる。そしてこの分野に標準化をもたらすきっかけとなったのがインターネットである。

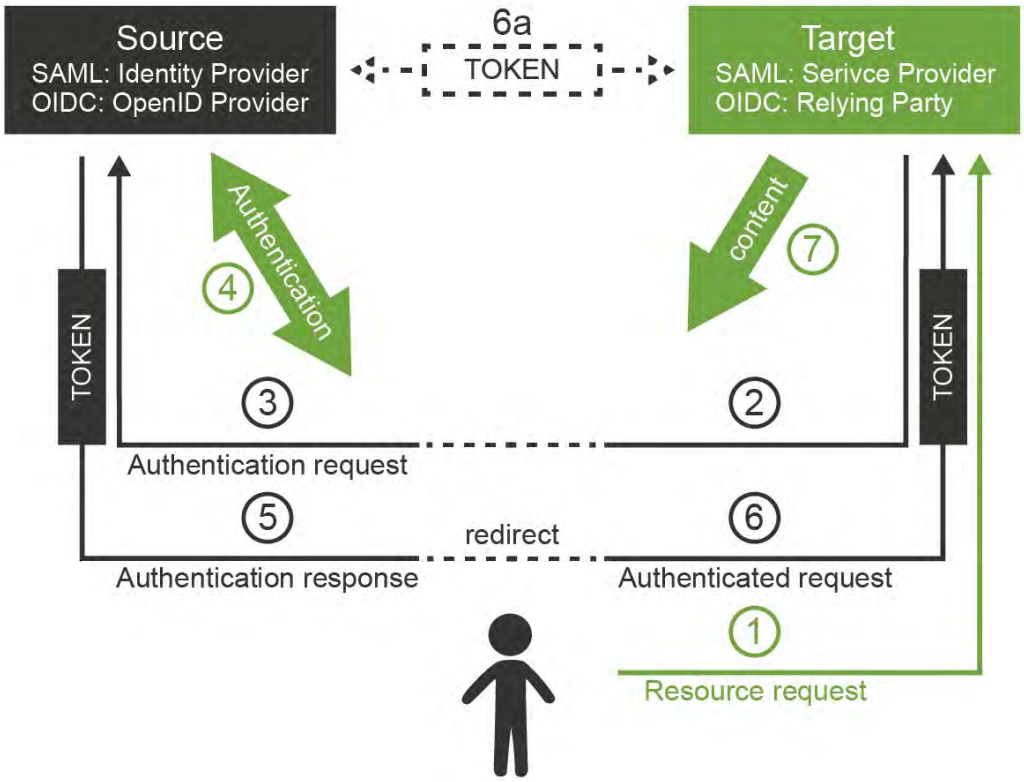
この分野に最初に普及した標準化プロトコルが、セキュリティ アサーション マークアップ ランゲージ（Security Assertion Markup Language：SAML）である。SAML の本来の目的は、クロスドメインサインオン、およびインターネット上で組織間でのアイデンティティデータの共有を可能にすることであった。これはきわめて複雑なプロトコルであり、XML 標準に大きくもとづいたセキュリティトークン形式である。SAML の理念および設計は SOAP ベースの Web サービスに非常に近く、実際 Web サービス世界に容易に溶け込むよう設計された。SAML の仕様は長く、複数のプロファイルに分かれており、オプション要素や機能も多く、SAML 全体は非常にリッチで柔軟性のある仕組みがまとまったものである。

SAML の主な目的はアイデンティティ情報の組織間転送である。SAML のトークンおよびプロトコルをベースとした数十または数百の組織から成る大規模なフェデレーションがある。電子政府のソリューションの多くは SAML をベースとしており、SAML で稼働している大規模なパートナーネットワークがあり、全体的にみると SAML はうまくいっているように見える。しかし SAML はみずからの柔軟性および複雑性の犠牲となった。そして最新の流行りの傾向は SAML にとってあまり好ましいものではない。XML および SOAP

ベースによる web サービスの仕組みは時代遅れになりつつある。おそらくこうしたすべてのことが他のプロトコルを生み出すきっかけとなったのだろう。

最新の流行では RESTful サービスやこれに似たよりシンプルな構築手法が支持されている。おそらくそのすべてが OpenID Connect（OIDC）プロトコルの開発に貢献したのだろう。OpenID Connect がベースとする概念は SAML よりもはるかにシンプルであるが、基本原則は同じものを再利用している。OpenID Connectの歴史は実に波乱に満ちている。LID や SXIP 等、いまとなってはそのほとんどが忘れ去られてしまったものの、数多くの自作プロトコルがすべての始まりであった。その後 OpenID プロトコルの開発へと続いたが、それはまだ非常にシンプルなものであった。OpenID は特にインターネットサービスプロバイダーの間である程度の注目を集めた。しかし、そのシンプルさとは裏腹に OpenID のエンジニアリングはあまりうまくいかず、すぐに限界へと達してしまった。OpenID を大幅に向上させる必要があることは明らかだった。当時、ほぼ無関係のプロトコルとして OAuth というプロトコルがあった。これはドメイン間の認可を管理する目的で設計されたものであった。そしてそのプロトコルは最初の姿からはまったくとはいわないが、ほぼ違うものへと発展した。OAuth2 と呼ばれるものだ。実際のところそれはプロトコルというよりも他のプロトコルを構築するために漠然と定義された枠組みに近い。OAuth2 の枠組みはドメイン間の認可およびユーザープロファイルプロトコルの構築に使用された。この新プロトコルは本来の OpenID よりも SAML にはるかに似ているため、当然ながら OpenID Connect と呼ばれるようになった。世の中には守る価値のある伝統もある。

今、同じ原則を使用し、ほぼ同じことを行う 2 つのプロトコルがある。その原則を示したのが下図である。



この図では次のようなやりとりが行われる。

1. ユーザーがリソースにアクセスしようとしている。アクセス先はターゲットサイトの Web ページまたは Web アプリケーションでもよい。
2. ターゲットサイトにはこのユーザーに有効なセッションがない。そのためユーザーブラウザをソースサイトへとリダイレクトする。リダイレクトには認証要求も追加する。
3. ブラウザがソースサイトへのリダイレクトに従う。ソースサイトが認証要求を受け取りこれを解析する。
4. ソースサイトがまだ認証していないユーザーであれば、ここで認証が発生する。ソースサイトはユーザー名、パスワード、証明書、ワンタイムパスワードのほか、ポリシーで必要とされるすべてのクレデンシャルの入力を求める。順調にいけば認証に成功する。
5. ソースサイトはブラウザをターゲットサイトへとリダイレクトする。ソースサイトはリダイレクトに認証レスポンスを追加する。このレスポンスの要となるのがトークンである。トークンによって直接または間接的にユーザーのアイデンティティが肯定される。

6. ターゲットサイトは認証レスポンスを解析しトークンを処理する。このトークンは単に真のトークン（SAML アーティファクト）への参照のこともあれば、アイデンティティ（OIDC UserInfo）を提供する別サービスへのアクセスキーのこともある。この場合、ターゲットサイトは新たに要求を行う（6a）。たいていこの要求は直接的なもので、ブラウザリダイレクトは行われない。いずれにせよ、こうしてターゲットサイトはユーザーアイデンティティに関するクレームを取得する。
7. ターゲットサイトはアイデンティティを評価し、認可等の処理を行う。通常、ユーザーとのローカルセッションはこの段階で確立され、次回要求時より認証リダイレクトは省略される。ターゲットサイトがようやくコンテンツを提供する。

下表はSAMLとOIDCでそれぞれ使用されている用語および技術を比較したものである。

	SAML	**OpenID Connect**
ソースサイト	アイデンティティプロバイダー（Identity Provider：IdP）	アイデンティティプロバイダー（Identity Provider：IDP）またはOpenID プロバイダー（OpenID Provider：OP）
ターゲットサイト	サービスプロバイダー（Service Provider：SP）	リライングパーティ（依拠当事者、Relying party：RP）
トークン	SAML アサーション（またはアーティファクト）	ID トークン、アクセストークン
対象	Web アプリケーション、Web サービス（SOAP）	Web アプリケーション、モバイルアプリケーション、REST サービス
ベースとしているプロトコル	該当なし	OAuth2
データ表現	XML	JSON
暗号化規約	XMLenc、XMLdsig	JOSE
トークン形式	SAML	JWT

注意深い読者は、Web ベースのアクセス管理の仕組みに似ていることに気付くだろう。その通りである。これは同種のものが何度も繰り返し発明されているのである。しかし白状すると、実はこの説明はWeb ブラウザのフローに限定したものである。SAML も OIDC も適用範囲はもっと広く、それだと違いはもっと明らかになる。ただ Web ブラウザを例に挙げれば、SAML、OpenID Connect、そしてシンプルな web-SSO システムの原則と類似性をうまく説明できる。

おそらく SAML、OpenID Connect、web-SSO システム間で最も重要な違いはその用途にあるだろう。

- SAML はWeb アプリケーションおよびWeb サービス向けに設計されたもので、集中型（シングル-IDP）のシナリオの取り扱いに優れるが、分散型のフェデレーションであっても機能できる。もしSOAP およびWS-セキュリティを使用しているか、大規模な分散型フェデレーションを構築する予定であれば SAML がよいだろう。

- OpenID Connect はほぼソーシャルネットワークや同じようなインターネットサービス向けに設計されたものである。その理念はいまだにいくぶん集中型である。一つの強力なアイデンティティプロバイダーとリライングパーティが多数いる場合にはうまくいくだろう。技術面でいえば、OpenID Connect は SAML よりも RESTful の環境の方がはるかにうまく適合するだろう。現在の流行の傾向は OIDC に好意的である。
- Web-SSO システムは単一組織内での使用向けに設計されている。複数の顧客向けアプリケーションの間に SSO を実装するならこれは理想的である。顧客はたった一つではなく多くのアプリケーションと対話をしていることなどまったく気付かずにすむからだ。Web-SSO システムは組織を超えて機能するような作りにはなっていない。

SAML も OIDC も主にドメインを超えて使用するために設計されているとはいえ、単一組織内でのみ使用されたとしても大した驚きではない。独自の仕組みではなく、標準化されたオープンなプロトコルを使用するのには明確なメリットがあるからだ。しかし SAML または OIDC をもとにした SSO システムは、シンプルな Web-SSO システムよりも若干複雑な設定になることを覚悟しなくてはならない。

ケルベロス、エンタープライズ SSO、フレンズ

多くの人々は、現在は何でも Web 技術にもとづいており、非 Web の仕組みは過去のことと考えがちである。しかし今でも Web ベースでないケースや、Web ベースの SSO や AM の仕組みがあまりうまくいっていない場合がある。特にエンタープライズ環境では今もレガシーアプリケーションが多く利用されている。リッチクライアントにもとづくアプリケーションや、文字ベース端末を使用したやりとりさえ目にするのは珍しくない。そして Windows や多数の UNIX 系といったネットワーク OS があり、VPN や 802.1X 等のネットワークアクセス技術がある。Web ベースのアクセス管理や SSO が全く機能しないケースは今も多い。

たいてい、こうした技術は Web よりも先行する。そして正直に言ってしまうと、集中型の認証や SSO は必ずしも新しいアイデアではない。だから非 Web アプリケーション向けに認証や SSO ソリューションがあってもさほど驚きではないのだ。

エンタープライズ SSO システムの代表的な例がケルベロス（Kerberos）である。これは 1980 年代に MIT により開発されたプロトコルであり、共通鍵暗号方式にもとづく OS およびリッチクライアント対応のシングルサインオンプロトコルである。暗号プロトコルとはいえさほど複雑ではなく、時という試練をしっかりとくぐり抜けてきた。特に認証やネットワーク OS の SSO にてまだ現役で使用されている。ケルベロスは Windows ネットワークの一部であり、UNIX サーバーの認証にはしばしば第一候補のソリューションとなる。しかしその共通鍵暗号方式を使用することはケルベロスに最も深刻な限界を与えてしまう。ケルベロスのレルムが大規模になると鍵管理が非常に難しくなってしまうためだ。そのためクロスドメインのシナリオでケルベロスを使用することはあまり現実的ではない。しかしクローズドな一組織の中であれば今でも非常に有効なソリューションである。

ケルベロスを使用する上で大きな欠点となるのは、各アプリケーションおよびクライアン

トを「ケルベロス対応（kerberized）」にする必要があるということだ。別の言い方をすれば、ケルベロス認証に参加したい人はそのソフトウェアをケルベロス対応にする必要がある。ケルベロス対応バージョンのネットワークユーティリティはたくさんあるため、UNIX ベースのネットワークであれば通常は問題はない。しかし一般的なアプリケーションには問題である。一般的な Web ブラウザにもケルベロス対応サポートがある。これはしばしば「SPNEGO」と呼ばれる。しかし、このサポートは通常 Windows ドメインとの相互運用に限定されている。そのためケルベロスは OS の SSO には有効とはいっても、すべてのアプリケーションに当てはまるような網羅的なソリューションではない。

多くのネットワークデバイスで、いわゆる認証（Authentication）、認可（Authorization）、アカウンティング（Accounting）（AAA）に RADIUS プロトコルが使用されている。しかし RADIUS はバックエンドプロトコルであり、クライアントとのやりとりには関与しない。RADIUS の目的は、ネットワークデバイス（Wi-Fi アクセスポイント、ルーター、VPN ゲートウェイ等）が他プロトコルの一部として受け取ったユーザークレデンシャルを検証できることにある。VPN または Wi-Fi に接続しているクライアントは RADIUS について何も知らない。よって RADIUS は LDAP プロトコルに似ており、必ずしもアクセス管理技術ではない。

すべてのエンタープライズアプリケーションに SSO を提供できるような、シンプルかつエレガントなソリューションなどないことは明らかである。しかし 1990 年代から 2000 年初めにかけて、「エンタープライズシングルサインオン（Enterprise Single Sign-On：ESSO）」と呼ばれる技術が登場した。ESSO の手法は各クライアントデバイスにインストールされたエージェントを使用することであった。画面上にログインダイアログが表示されると、エージェントはこれを検出し、ユーザー名とパスワードを入力し、ダイアログを提出する。このエージェントが実に高速であればユーザーはダイアログすら気付かないため、シングルサインオンの印象を受ける。しかし明白な欠点がある。エージェントはクリアテキスト形式ですべてのパスワードを把握しておく必要があるのだ。ESSO にはランダムに生成されたパスワードや、ワンタイムパスワードさえあるようなバリエーションがあり、この問題を部分的には解決している。しかし ESSO は全アプリケーションのパスワード管理との統合も必要であり、これは必ずしも簡単なことではない。しかし最も深刻な欠点はエージェントである。エージェントはエンタープライズの厳しい管理下にあるワークステーション上でのみ機能する。しかし世界は変化しており、エンタープライズの境界線は効率よく消えていったが、エンタープライズはクライアントデバイスを必ずしも管理できているわけではない。そのため ESSO も今となってはほぼ遺物である。

アクセス管理およびデータ

アクセス管理サーバーとアイデンティティプロバイダーが適切に機能するにはユーザーに関するデータが必要である。しかしこれは複雑である。アクセス管理システムの目的はユーザーによるアプリケーションへのアクセスを管理することである。通常これはアクセスを認証、認可（一部）、監査することを意味する。これが機能するには、ユーザーデータが格納されているデータベースに AM システムがアクセスできる必要がある。ユーザー名、パスワードなどのクレデンシャルのほか、認可ポリシー、属性などへのアクセスが必要と

なる。一般に AM システム自体がこれらのデータを格納しておくことはなく、あくまでも外部のデータベースに依存している。そしてそのデータベースはディレクトリサービスに格納されていることがほとんどである。ディレクトリサービスは軽量で可用性が高く、拡張性に富んでいることからして、これは当然である。一般に AM システムが必要とする属性はシンプルなため、ディレクトリーデータモデルの機能に制限があっても、ここでは制約要因にはならない。アクセス管理とディレクトリサービスが実に賢明な組み合わせであることは明らかだ。

しかし、ここに一つ重大な問題がある。特に AM システムをシングルサインオンサーバーとしても使用する場合、ディレクトリサービス内のデータとアプリケーション内のデータを統合し、必ず整合性を確保しなければならない。たとえば、一人のユーザーがアプリケーションごとに違うユーザー名を使用していると大問題となる。ログインに使用するユーザー名はどれなのか？どのユーザー名をアプリケーションに送信したらよいのか？このような状況への対処法はいくつかあるが、非常に厄介でコストもかかってしまう。AM システムをデプロイする前にデータを統合するほうがずっと簡単である。

AM の「M」が管理（Management）を表すとはいえ、標準的な AM システムのデータ管理機能は非常に限られたものにすぎない。その基礎となるディレクトリシステムがすでに適切に管理されていることが前提となっている。例えば、典型的な AM システムが有するユーザーインターフェイスは、ユーザーレコードの作成、修正、削除というごく限られたミニマリズム的なものにすぎない。中にはセルフサービス機能（パスワードリセット等）などを備えたものもあるかも知れないが、そのような機能もかなり制限されているのが一般的だ。AM システムは AM ディレクトリサービス内のデータとアプリケーション内のデータの整合性がとれているという事実に依存しているにもかかわらず、AM システムそのものを使用してデータを完全に同期させる手段はないのがほとんどである。オンデマンドや何らかの機会に乗じてデータ更新を行う方法はあるかもしれない。例えばユーザーの初回ログイン時にデータベース内のユーザーレコードを作成する、といったことである。しかしレコード削除やアクティブでないユーザーのレコード更新についてはソリューションが一切ない。

よって AM システムを単独でデプロイすることは通常不可能である。AM 機能が最も程度の低いものだとしても、基礎となるディレクトリサービスはほぼ必ず厳しい要件となる。しかし AM システムが本当に適切に機能するためには、データを管理・同期してくれるものが必要となる。そこでよく使用されるのがアイデンティティ管理（Identity Management: IDM）システムである。そして実際に、たいていの場合は、ディレクトリおよび IDM システムを AM システムより先にデプロイするよう強く推奨されている。AM システムはデータがなければ機能しない。そしてもし適切に管理されていないデータで稼働すれば、すぐにうまくいかなくなるだろう。

アクセス管理システムのメリットとデメリット

アクセス管理およびWeb SSOシステムには大きなメリットがある。その特徴の大半は集中型認証というアクセス管理の原則によるものである。認証は中央アクセス管理サーバーが行うため、その管理も監査も容易である。このような集約化を利用することで、認証ポリシーを着実に適用し簡単に変更できる。また認証技術への投資をより適切に活用できる。たとえば、多要素認証やアダプティブ認証をアプリケーションごとに実装しようとするとコストが非常にかかる。しかしAMサーバーに実装すれば、追加投資がなくてもすべてのアプリケーションがこれを再利用できる。

一方で欠点もある。アクセス管理が一か所に集中しているため、明らかにそこが単一障害点になってしまうことだ。万が一AMサーバーがダウンしてしまえばだれもログインできなくなる。そしてたいていこれはシステム機能に大きな影響を与えることとなる。よってAMサーバーには高い可用性が必要なのだが、そう簡単にはいかない。AMサーバーはいとも簡単にパフォーマンスのボトルネックになりかねないため、そのサイズ設定には細心の注意が必要なのだ。しかし最も深刻な欠点はトータルコストであろう。AMサーバー自体のコストは大した問題ではない。ただ、サーバーは各アプリケーションと統合する必要がある。この分野にはいくつか標準があるものの、それでも統合はかなり骨の折れる作業である。アプリケーションでのAM標準とプロトコルのサポートは完成には程遠いものである。特に旧式のエンタープライズアプリケーションは、これまでの認証サブシステムをAMサーバーに切り替えるための変更が必要となる。これはしばしば多大な費用を要し、AMテクノロジーの採用がほんの一部のエンタープライズアプリケーションに限られてしまうほどである。

多くの企業がIAMプロジェクトの第一歩としてAMシステムのデプロイを計画しているものの、この一歩がほとんどうまくいかない。一般にプロジェクトでは5割〜8割のアプリケーションをAMソリューションへ統合しようと計画される。しかしAMシステムと統合されるアプリケーションはほんの一握りになってしまうのが現実だ。残りのアプリケーションは、あとから慌ててプロジェクトに追加されるIDMシステムを利用して統合される。だからこそ前もって計画したほうがよい。AM統合にかかる工数を分析し、AMソリューションに向けて現実的な計画を作成するということだ。アクセス管理によって約束されたメリットが確実にもたらされるようにする。IDMからスタートし、AM部分はあとから追加する戦略のほうがはるかに合理的であることが多い。

単一のアクセス管理神話

アクセス管理システムのリダイレクションによるアプローチは、認証プロンプトを表示しユーザーとの対話を実行できる何らかのものをユーザーがもっていることが前提となっている。多くはWebブラウザである。そのためこの手法は従来のWebベースアプリケーションに適用されることがほとんどである。また、この手法が変化したものもシングルページWebアプリケーションに適用されている。ただ、一般に、リッチクライアントやOS認証を使用したアプリケーション、モバイルアプリケーションには直接適用できない。これらは認証実行に使用できる主環境がWebブラウザではないからだ。埋め込みブラウザに依存したソリューションならばいくつかあるが、これらのアプリケーションに対してAMテクノロジーは完全には適合しないという根本的な事実をくつがえすものではない。こう

したアプリケーションは SSO システムとしてケルベロスに依存するか、SSO システムとは一切統合しない。

典型的なエンタープライズ環境はさまざまな技術が組み合わさって成り立っており、それらがすべて Web ベースとは限らない。そのため、企業でデプロイされているすべてに対して、単一の AM システムを適用できる可能性は低い。認証はユーザーとの対話と深く関連しているため、ユーザーがアプリケーションと対話する方法に依存する。ユーザーは Web アプリケーション、モバイルアプリケーション、OS との対話でそれぞれ違う技術を使用しているため、これらのシステムにおける認証方法および SSO 方法も異なることは明らかである。

それゆえ企業には複数の AM システムまたは SSO システムがあり、それぞれが自身の技術領域を担当していると期待される。そして、それぞれの領域は管理されている必要がある。

実用的なアクセス管理

統合アクセス管理システム、シングルサインオン、クロスドメインアイデンティティフェデレーション、普遍的に適用できる 2 要素認証—これらはアイデンティティ管理とアクセス管理（Identity and Access Management：IAM）を検討する際に多くの人々が望むものである。そしてこれらはすべて完全に有効な要件である。ただしどれもコストがかかる。そして特にアクセス管理分野においては、コストの大半は AM ソフトウェアではないため見積もりも非常に難しい。トータルコストの大部分が既存のアプリケーションやサービス、クライアントの中に潜んでいるのだ。AM プロジェクトを計画する際はこれらをすべて考慮しなくてはならない。

たとえ人々の望むものが AM だとしても、第一歩として AM から着手するのは賢明**ではない**。AM デプロイメントは多くのものと依存関係にあるからだ。たとえば一元化されたユーザーデータベース、管理され常に同期のとれているデータ、十分な柔軟性で統合が可能なアプリケーションなどである。IT インフラストラクチャーが高度に均質かつシンプルでないかぎり、これらの依存関係が満たされている可能性はほぼない。従って、IAM プログラムの初期段階で AM プロジェクトを計画したとしても、その目標は未達となるか、完全に失敗に終わることはほぼ確実である。しかし、AM プロジェクトの範囲を適切に定め、計画し、現実的な目標を設定するのであれば、成功する確率は高く十分な価値がもたらされるだろう。

おそらく次のようなことを自身に問いかけてみることが AM プロジェクトを評価する上で最良の方法であろう。

- 本当にすべてのアプリケーションにアクセス管理が必要なのか？100%対象とする必要があるか？すべてのコストを賄えるか？最も大変な統合対象である、2、3 のアプリケーションだけでも十分ではないか。そうしたアプリケーションはどれか把握しているか？業務中にユーザーが本当に使用するのはどれか把握しているか？ユーザーが必要とするものを把握しているか？
- AM と SSO の効果がほぼユーザーの利便性であることをしっかり認識しているか？

AM デプロイメントによるセキュリティ上の真のメリットは何か？アプリケーションへのネイティブ認証を無効にするか？システム管理者に対してもか？管理上の緊急時はどうするか（例：システム回復）？システム管理者は AM システムを回避できるのか？できる場合、セキュリティ上の本当のメリットは何か？できない場合、万が一 AM システムがダウンした場合の回復処理は？

- 旧式かつあまり使われていないアプリケーションにも本当に SSO が必要か？そこにある本当の問題は何か？問題はユーザーが1日に数回パスワードを入力することなのか？または、ユーザーが複数のアプリケーションにそれぞれ異なるユーザー名やパスワードを入力しなければいけないことやクレデンシャルを何度も忘れてしまうことなのか？おそらく簡単なデータクリーンアップやパスワード管理によって最も大変な問題も解消され、AM プロジェクトにかかる費用をかなり抑えられるのではないか？

AM テクノロジーは IAM プログラムの中で最も目に見える部分である。しかし同時に最もコストのかかる部分であり、セットアップと維持が最も難しい部分でもある。したがって他の IAM 技術を軽視したりせず、また何でもかんでも AM という金のハンマーで解決しようとしてはならない。各状況でそのジョブに相応しいツールを使うことが適切な対応である。ただ、IAM プログラムを成功させるには AM テクノロジーが必要不可欠である。

アイデンティティ管理

アイデンティティ管理とアクセス管理（IAM）分野全体の中で、おそらくもっとも見過ごされ軽視されてしまうのが *アイデンティティ管理*（Identity management：IDM）であろう。しかし IDM はほぼすべての IAM ソリューションにおいて重要な部分である。そしてほぼすべての組織に相当なメリットをもたらせるのも IDM である。ではこの得体の知れない IDM とは一体何なのか？

IDM とはまさにその名のとおり、アイデンティティを管理することにほかならない。新入社員のアクティブディレクトリとメールボックスを作成するプロセスのことである。セキュリティインシデントの発生時には疑わしいユーザーへの全アクセスをただちに無効にすることができる。組織変更の際はユーザーへの新たな特権の追加や古い特権の削除にも対応する。退職した従業員のすべてのアカウントが適切に無効になっているかを確認する。派遣社員、パートナー、サポートエンジニアなど、人事システムで管理されていないすべてのサードパーティのアイデンディティのアクセス権限を記録する。ロールの要求と承認プロセスを自動化する。ユーザー権限の変更をすべて監査証跡に記録する。ロールおよびアクセス権限の年次レビューを管理する。アプリケーション内に保持されるユーザーデータのコピーを確実に同期させ、適切に管理する。そのほかにも、各組織が効率的かつ安全な方法で運用するために欠かせないさまざまなことを行うのが IDM だ。

IDM はまるで比類なきもののようだ。では何か落とし穴があるのではないか？そう、確かに落とし穴はある。というより落とし穴はあった、といったほうが良いかも知れない。かつて IDM システムは高価だった。非常にコストのかかるものだった。せっかくすばらしく明白なメリットがあっても、あまりに高すぎるがゆえそのコストの正当化に非常に苦慮するほどであった。しかしそのような時代はもはや終わった。

注：用語について。「*アイデンティティ管理*（Identity Management）」という用語はアイデンティティ管理とアクセス管理（IAM）分野全体に関して頻繁に用いられている。これは何やら紛らわしいのは、シングルサインオンやアクセス管理といった技術が実際にはアイデンティティを管理していないためだ。このような技術が管理するのはアプリケーションへのアクセスである。ディレクトリサーバーでさえ、正確にはアイデンティティを*管理*していない。ディレクトリサーバーはアイデンティティを格納しそれらにアクセス権を与えている。実際に、アイデンティティを管理する一つの完全な技術領域がある。これらのシステムはアイデンディティの作成と管理の責任を担っており、*アイデンティティプロビジョニング*システム、*アイデンティティライフサイクル管理*システム、*アイデンティティ管理*システムとも呼ばれている。しかしその技術の現状を考えれば、こうした呼び方はたしかに控えめだ。これらができることはアイデンティティのプロビジョニングやアイデンティティライフサイクルの管理にとどまらないからだ。我々はこれらのシステムをシンプルに*アイデンティティ管理*（IDM）システムと呼ぶこととする。一方、アクセス管理、ディレクトリサービス、アイデンティティ管理およびガバナンスを含む分野全体を指すときは、*アイデンティティ管理とアクセス管理*（identity and access management：IAM）と呼ぶことにする。

では最初から話そう。1990 年代はまだ明確に「アイデンティティ管理」とされた技術はなかった。もちろん現代コンピュータの時代がはじまってほぼすぐ、これまで述べてきたすべての問題が存在していた。そしてそれに対する何らかのソリューションも常にあった。ただ、そうしたソリューションの大半はペーパーワークやスクリプティングをもとにしたものであった。状況が深刻になったのは、1990 年代から 2000 年代にかけて業界全体にシステム統合の大きな波が押し寄せてからである。各アプリケーションのデータやプロセスが統合されると共に、IDM の問題はますます顕著になってきた。情報スーパーハイウェイの時代に、手動によるペーパーベースのプロセスはあまりに遅すぎた。数週間おきに新たなアプリケーションがデプロイされるような世界でスクリプトを保守することは実に面倒であった。そこで当時の最新のアイデンティティ技術を活用したアイデンティティ統合の取り組みがはじまったのは自然な流れであった。その技術がディレクトリサービスである。しかしすでに説明したとおり、ディレクトリは統合に最適なものではなかった。LDAP をある種の異国病と考えるような環境や、アプリケーション管理チームによってユーザー名および識別子がまったく無作為にアサインされた環境、各アプリケーションが、自身のデータベースに格納されるデータだけが権威あるデータであると主張する環境には、ディレクトリは必ずしも適さなかったのである。

第一世代

統合問題が契機となって 2000 年代初頭には IDM 技術が始まった。初期の IDM システムは、ユーザーおよびアカウントについて稼働するようにと、どうにかハードコードされたデータ同期エンジンにすぎなかった。少し後になって、シンプルなロールベースアクセス制御（Role-Based Access Control：RBAC）エンジンと管理インターフェイスが追加された。2000 年代半ばには一応完成された IDM システムが姿を現した。これが第一世代の IDM システムである。これらのシステムはアプリケーション間のアイデンティティデータを同期させ、基本的な管理機能を提供することもできた。単純な機能ではあったが、それさえ大成功であった。IDM システムはアプリケーションに大幅な変更を加えることなくデータを同期させることができたため、統合コストを妥当なレベルに抑えられた。問題は IDM

システム自体のコストが非常にかかることであった。当時のシステムはやはりいくぶんか未熟であったため、その構成やカスタマイズには専門性に優れたエンジニアが必要であった。そのため中小規模の組織にとっては IDM ソリューションのデプロイは費用がかかりすぎてとても手が出せなかった。大規模な組織でさえ、許容できるコストに抑えるため機能を大きく制限した IDM ソリューションをデプロイすることが多かった。こうした IDM システムがのちに進化し強化されていったのである。そしてその機能を拡張するガバナンスおよびコンプライアンス用のコンパニオン製品があった。しかし多くの場合、元来のアーキテクチャや製品を変えることはほぼ不可能であった。ゆえに多くの第一世代の製品は初期の製品設計時に発生した制限事項に今もなお苦しめられている。

第一世代の IDM システムはすべてソース非開示（クローズドソース）の商用ソフトウェアであった。これらの多くは今も市場にあり、リーダーとすらみなされている。しかし IDM 製品がクローズドソースであることはそもそも大問題である。どの IDM ソリューションも多かれ少なかれカスタマイズしなければならない。適応するのは IDM システムであるべきで、アプリケーションではない。各アプリケーションを IDM 標準インターフェイスに適応させるということは、あらゆる場所、プラットフォーム、言語に多くの変更を必要とする。そのトータルコストたるや積みあがって莫大な数字になる。これはときどき試みられているものの、ほぼ例外なく失敗している。実用的な方法ではない。IT インフラストラクチャーには多くのアプリケーションがある一方、IDM システムは一つのみだ。IDM システムがアプリケーションや業務プロセスに適応すれば、変更は少なくて済み、一か所にすべてが揃い、単一プラットフォームに実装される。だから IDM システムが適応できなくてはならないのだ。相当に、しかも容易に適応できなくてはならない。クローズドソースのソフトウェアは、予測困難な要件への適応が苦手なことでよく知られている。つまりは実際のところ、第一世代の製品をベースとした IDM プロジェクトは困難かつ高コストであったということだ。また、クローズドソースのソフトウェアはベンダーロックインに非常に陥りやすい。一度 IDM システムをデプロイし統合してしまったら、競合システムに置き換えることは非常に難しい。当システムを変更できるのは担当したクローズドソースベンダーだけとなり、システムを効率よく置き換えられないため、顧客はとても交渉できるような立場ではないのだ。つまりメンテナンスコストが高いということである。そのおかげで第一世代の IDM システムは商業的には成功した。ベンダーにとってのみだが。この世代の IDM ソリューションが本当にうまくいったのはほんの一部の顧客のみだったが、こうした顧客についてさえ、本当に長期にわたるメリットがあったかは疑わしい場合が多い。

第二世代

理由は何だったのか、それは推測の域を出ないが、2009 年から 2011 年頃にかけて新たに実に興味深い IDM 製品が市場に登場したことは事実である。興味深いことの一つは、それらがすべて多少なりともオープンソースであることだ。しかしそれだけではない。どの製品も IDM 分野をターゲットにしているにもかかわらず、それぞれがかなり異なっていることだ。その理念も統治する原則も違う。プログラムしてうんざりするほど拡張できる単純なフレームワークとしての製品もあれば、ガバナンス機能、コンプライアンス機能に対応した本格的な IDM 製品もある。また、ある製品はアーキテクチャが非常に重厚でサービス指向アーキテクチャ（Service-Oriented Architecture：SOA）の原則にもとづいている。そして実際にシステム内に完全なエンタープライズサービスバス（Enterprise Service

Bus：ESB）が組み込まれている。一方、JavaScript でつなぎ合わせられる疎結合の一連のコンポーネントをアーキテクチャとする製品もある。他にもいくつかあるが、ほとんどは実績があり効率もよく軽量な Java アーキテクチャをベースとしたものである。優れたオープンソース標準に従い、ユーザーコミュニティを構築している製品もあれば、ソースコードライセンスの下でのみオープンソースにしている製品もあり、後者は明らかにあまりコミュニティ志向ではない。このように大きなばらつきがあるが、つまりは選択肢が多いということである。

正直に言ってしまえば、第二世代の IDM 製品は全部が全部、実用化の準備が整っているわけではない。製品が成熟するにはある程度時間がかかる。しかし何の問題もなく一般的な IDM シナリオすべてに対処できる製品はほとんどない。ただし、すべての機能を網羅した包括的な IDM システムが少なくとも１つはある。現在、第二世代の製品の大半は活発な開発段階にあり、近い将来すばらしい改善がみられると期待できる。

第二世代 IDM 製品の最も際立った強みは、オープンソースという特色である。見過ごされがちな特色ではあるが、この特色には非常に大きな価値がある。IDM エンジニアなら誰もが知っているとおり、IDM 製品を理解すること、そして IDM 製品の適応力、この２つはどの IDM プロジェクトにおいても非常に重要である。オープンソースはこの理解と柔軟性の両方に対応する最良の方法である。そして３つ目の重要な強みは、オープンソース製品であればベンダーロックインの状況を作り出すことはほぼ不可能ということである。オープンソース製品はすべて、専門的なサポートサービスを提供する企業によって支えられている。彼らのサービスは IDM の商用製品が提供するサービスと同等である。よってそれらのプロジェクトにあるオープンソースという特色からは何らデメリットは発生しない。しかも、この手法を真に革新的にする大きなメリットがある。

新しいIDMデプロイプロジェクトには第二世代のIDM製品を選ぶべきなのは明白である。その長所は実に明白で今後の発展にも大きな可能性を秘めている。しかし、例によって賢く選択しなければならない。

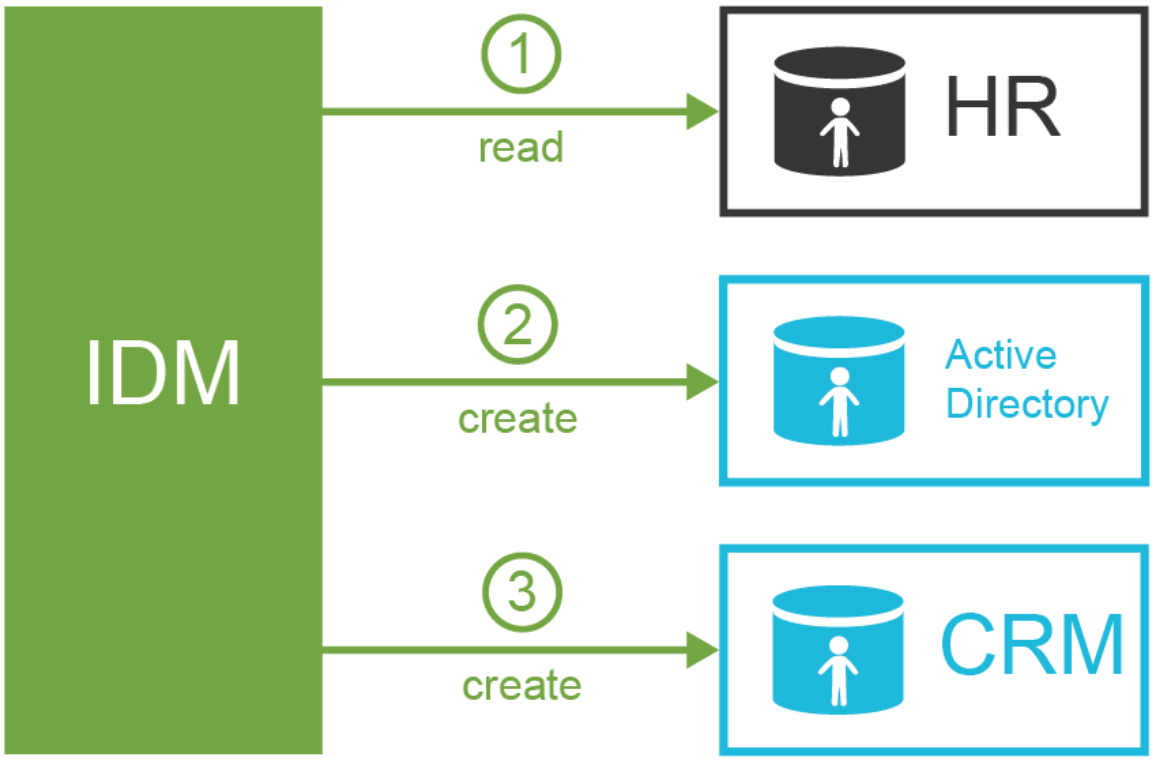
アイデンティティ管理とは、結局のところ何なのか？

*アイデンティティ管理*は、非常に豊富で包括的な機能を網羅するシンプルな用語である。その機能とは、アイデンティティプロビジョニング（および再プロビジョニング、プロビジョニング解除）、同期、組織管理、ロールベースのアクセス制御、データの整合性、ワークフロー、監査、その他何十もの機能のことである。これらはすべて、ちょっとしたスクリプトと味付けを加えて配合され混合され、やがて滑らかな IDM ソリューションができあがる。そのためアイデンティティ管理とは何かを辞書のように定義して説明することは実に難しい。むしろアイデンティティ管理とは何か、典型的な使用例をいくつか用いて説明したい。ここに ExAmPLE, Inc.という架空の企業があるとしよう。この企業は、数千人の従業員、適切なパートナーネットワーク、顧客、供給業者など現実世界の企業とすべて同じものを有している。そして同社の IT インフラストラクチャーでは IDM システムが稼働している。

ExAmPLE 社はアリスという新入社員を雇用する。彼女は就業開始の数日前に雇用契約を結ぶ。ExAmPLE 社の人事スタッフが当契約を人事システムに入力する。IDM システムは定期的に人事レコードをスキャンしており、新入社員のレコードを検出する。するとこのレコードを取り込み、分析を行う。さらに人事レコードからユーザーの氏名と従業員番号

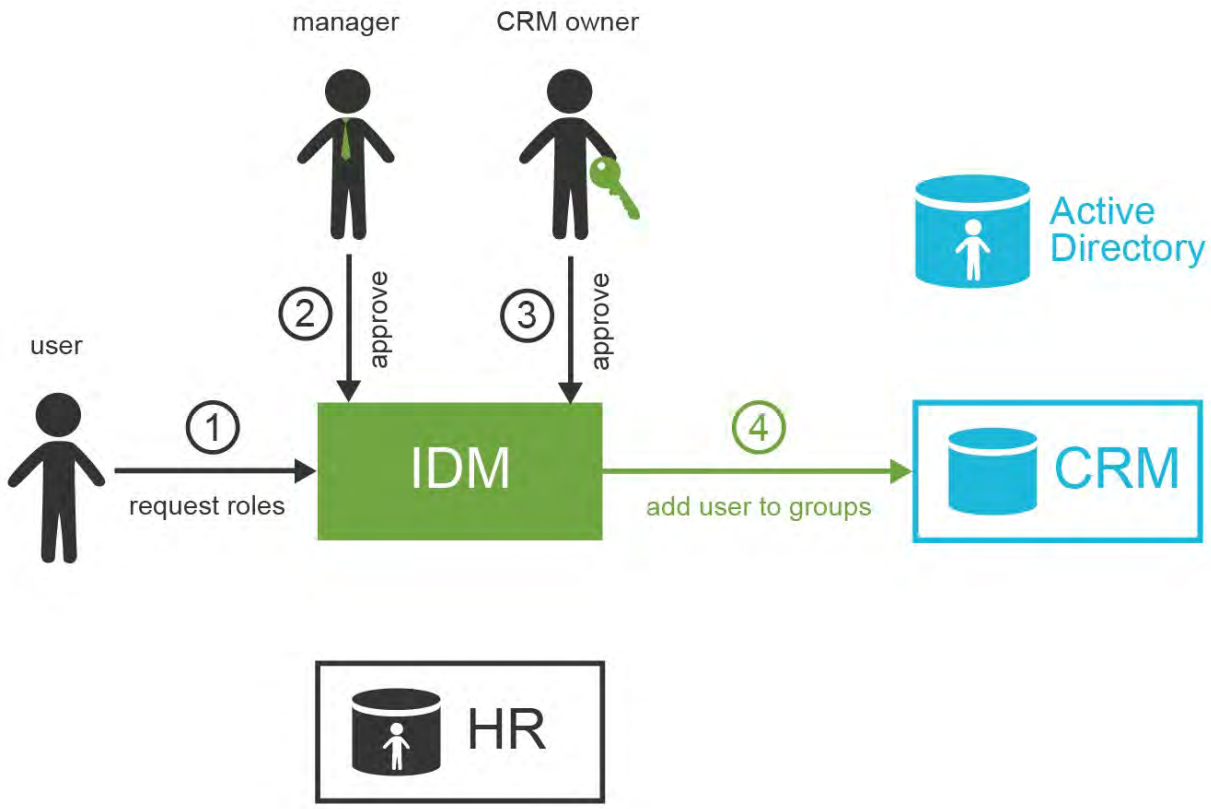
を取得して一意のユーザー名を生成し、その情報をもとにIDMシステムにユーザーレコードを作成する。さらに人事レコードから組織コード（10010）も取得する。

IDMシステムは組織系統図を調べ、コード10010が営業部に属していることを発見する。そこで当ユーザーを営業部にアサインする。また、人事レコード内にある職務コード（007）の処理も行う。IDM ポリシーによるとコード 007 とはセールスエージェントのことであり、このコードを有する人員はみな自動的に「セールスエージェント」というロールを受け取ることになる。よってこのロールをアサインする。セールスエージェントは営業部に属するコア従業員であるため、 IDM は自動で当ユーザー用に会社のメールボックスとともにアクティブディレクトリのアカウントを作成する。作成したアカウントが営業部の組織単位に配置される。「セールスエージェント」ロールは当ユーザーにさらに特権を与える。よってアクティブディレクトリの当アカウントが自動で営業グループおよび配布リストにアサインされる。また、このロールはCRMシステムへのアクセス権も与える。よってCRMアカウントも自動作成され適切なグループにアサインされる。今述べたことはすべて、新しい人事レコードの検出後わずか2〜3秒の間に起こる。しかもすべて自動で。

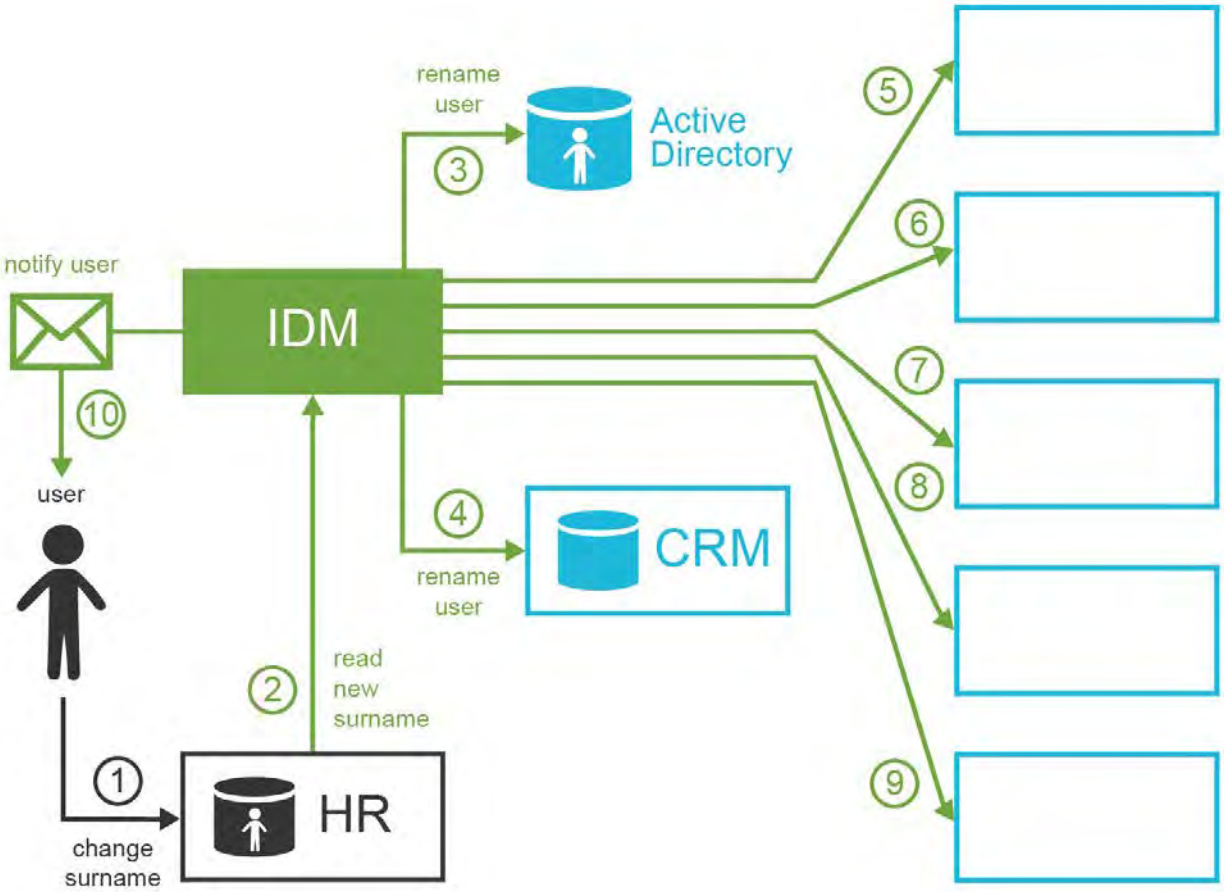


アリスはキャリアを築きはじめ、実に有能な従業員となっている。そのためさらに責任を任されるようになる。彼女は当該分野の経験的データをもとに専門的な市場分析にとりかかろうとしている。ExAmPLE社はとても柔軟性のある企業であり、さらなる業務効率化に向け常に新しい方法を創り出している。今回も、彼女のスキルを有効活用するためとし

てこの職務を創り出した。つまりアリスの新しい職務に該当する職務コードはまだない。しかし効率的に業務を行うには CRM システムに新たな特権が設定される必要がある。しかも今すぐに。幸いなことに、ExAmPLE 社は柔軟性のある IDM システムを整備している。アリスは IDM システムにログインし、必要な特権を選択しそれを要求する。その要求にはアリスの上司と CRM システムオーナーの承認が必要である。要求通知を受取った両者は、IDM システムで簡単に承認または拒否を行うことができる。承認されれば、アリスの CRM アカウントが適切な CRM グループへ自動でアサインされる。こうしてアリスは、特権を要求してから数分あるいは数時間後には分析業務にとりかかることができる。

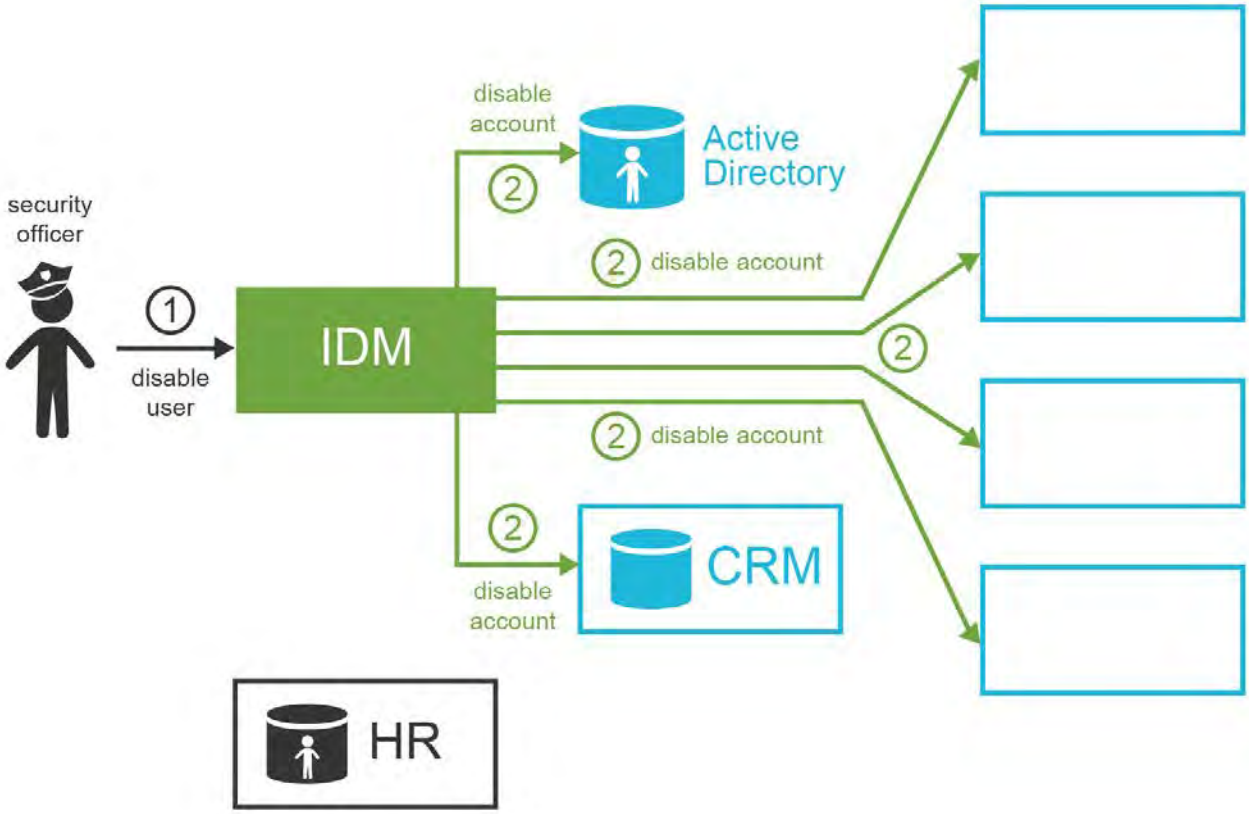


アリスはその後幸せな日々を送っている。ある日、彼女は結婚を決める。他の多くの女性と同様、彼女も結婚後は名字を変えるという変わった慣習の下で生きている。しかし彼女は非常に責任のある職務についており、多くの情報システムにアカウントをもっている。それらすべての名前を変えるのは簡単ではないとお思いだろう？しかし実際はとても簡単なのだ。ExAmPLE 社には IDM システムがあるのだから。アリスは人事部を訪れる。そして人事スタッフは人事システムに登録されている彼女の名字を変更する。IDM システムがこの変更をピックアップし、関係するシステムすべてに伝達する。アリスは自動的に新姓で新メールアドレスも取得する（旧アドレスはエイリアスとして保持される）。アリスのもとに新メールアドレスが使用できることを知らせる通知が届く。変更は素早くすっきりと、そして簡単に済む。



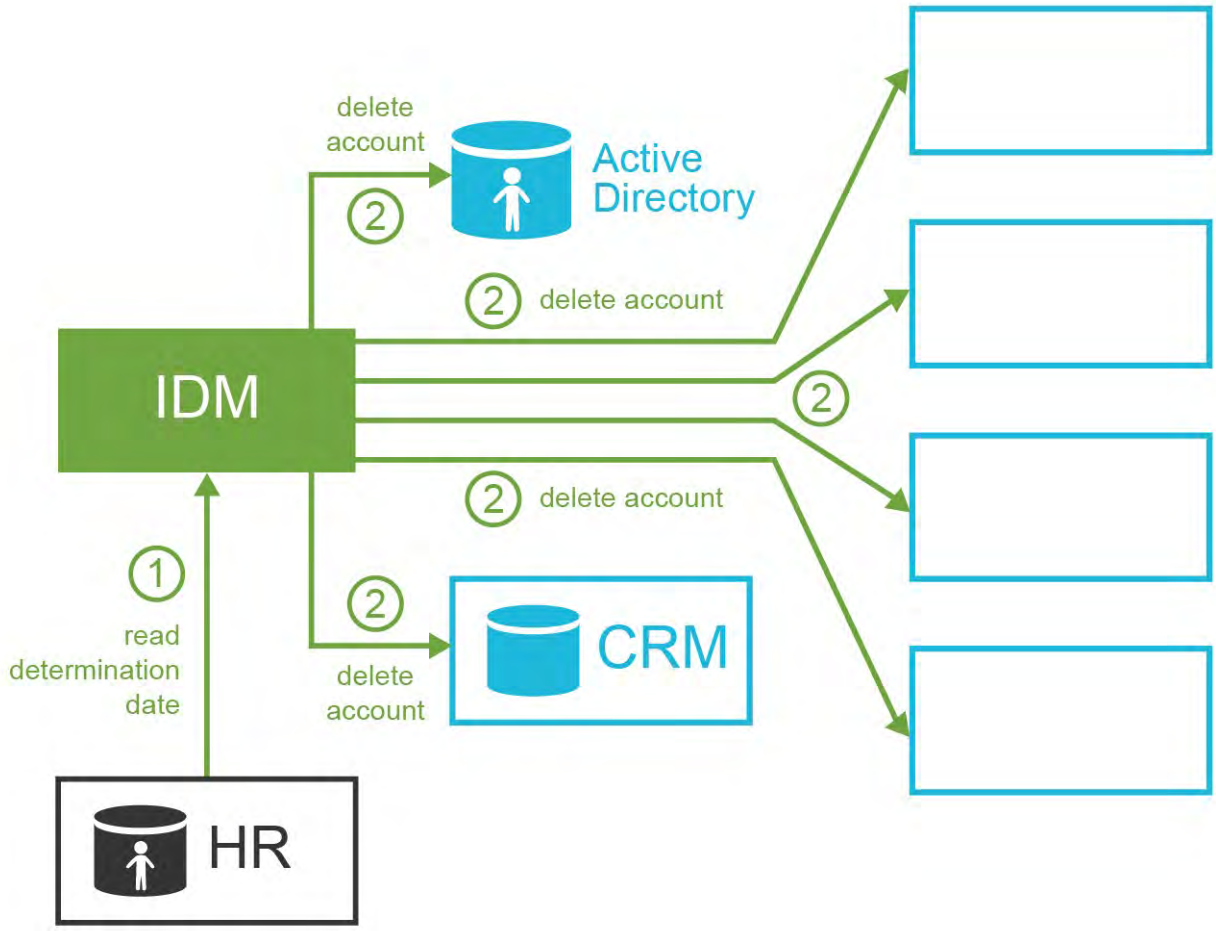
その日の午後遅く、アリスは自分のパスワードがまもなく期限切れであることに気づく。アプリケーションすべてのパスワードを変更するのはさぞ大変な作業だろう。しかし彼女は何をしたらよいかわかっている。IDM システムにログインしてパスワードを変更する。IT セキュリティオフィスが設定したポリシーに従い、このパスワード変更が関係する各システムに自動で伝達される。

翌月、予期していなかった事が発生する。セキュリティインシデントである。セキュリティオフィスがかなり早い段階で発見し、現在調査が行われている。どうやら内部犯によるものらしい。セキュリティ担当者は IDM システムからのデータを活用しながら、影響のあったアセットへのアクセス権をもつユーザーを重点的に調査している。そして第一容疑者としてマロリーを隔離する。奇妙なことに、マロリーにはそれらのアクセス権が一切ないはずなのに、だ。幸い IDM システムはすべての特権変更に関する監査証跡も保持している。おかげでセキュリティ担当者はマロリーにそれらの特権をアサインしたのが同僚のオスカーであることが分かる。両人は尋問を受けることになる。ただ、今回のインシデントは機密性の高いアセットに影響を与えるため、この話が広まる前にいくつか予防措置を行っておく必要がある。セキュリティ担当者は IDM システムを利用して、マロリーおよびオスカーのアカウントをすべてただちに無効にする。IDM はわずか 2〜3 秒で、関係するすべてのアプリケーションに対する 2 人のアカウントを無効にできる。

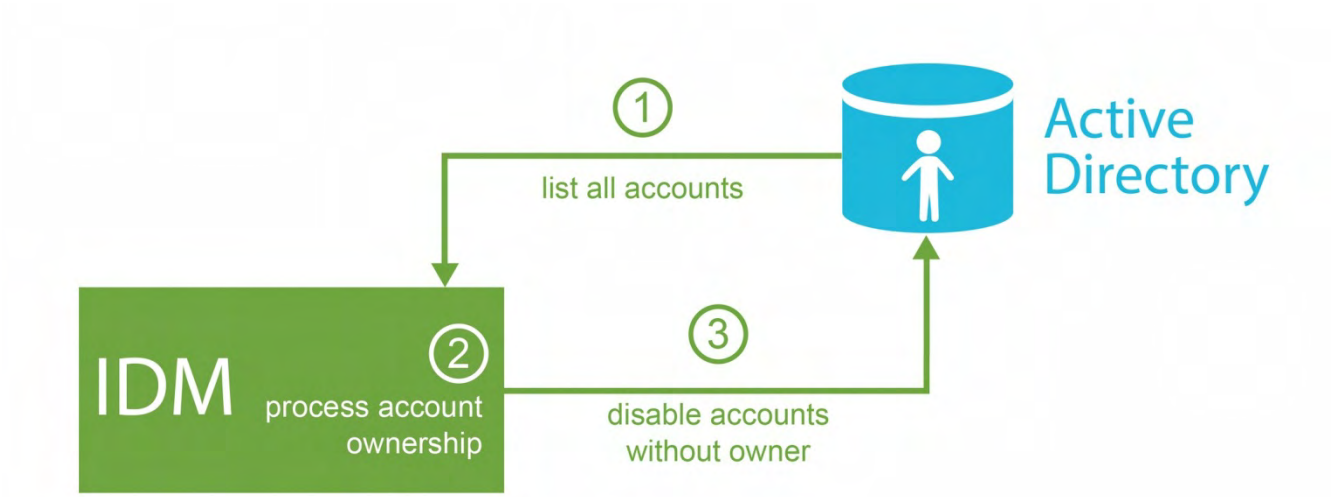


あとになって、オスカーはほぼ無実であることが調査により判明する。マロリーはオスカーからの信頼を悪用し、彼をだまして自身に特権を追加させた。その特権を不正利用して機密データを入手、これを売ろうとしていたのである。よってマロリーはただちに退職、オスカーは会社に残ってもよいという結論となる。しかしオスカーは本件で判断力不足が露呈したため、彼の責務は減らされる。IDM を利用してマロリーのアカウントをすべて永久に無効にし、オスカーのアカウントは再び使用可能にするとともに、オスカーがもつことでリスクが伴うと考えられる重要な特権は取り消される。

数か月たっても、オスカーはまだ自分の失敗を恥じている。そして彼は ExAmPLE 社との雇用契約は更新せず、これ以上トラブルを起こすことなく退職しようと決意する。彼の雇用契約はその月の末日で終了する。人事システムにその日付が記録されると IDM システムがそれを取得する。最終出勤日の夜中 12 時に IDM システムは彼のアカウントをすべて自動で削除する。オスカーはニューヨークでバーテンダーとして新たなキャリアをスタートする。彼は大きな成功を収める。



セキュリティオフィスはこのセキュリティインシデントにプロらしく対応し、IDM システムは迅速かつ効率的なセキュリティレスポンスに必要な重要データを提供した。セキュリティオフィスは取締役会から称賛を受ける。それでも当チームは常に改善に励んでいる。今回のインシデントから学び、再発の可能性を減らそうとしているのだ。そこで IDM システムからのデータを利用して各ユーザーにアサインされた特権の分析を行っている。本来、IDM システムはアプリケーションにアカウントを作成、変更を行うのが仕事である。しかし IDM システムはアプリケーションと双方向通信を行っているため、この分析はただ無意味にスプレッドシートを使用することとはわけが違う。分析は実際のアプリケーションデータプロセスにもとづいており、IDM システムによって統合される。実際のアカウントは何か、どのユーザーに属しているのか、どんなロールをもっているのか、どのグループに属しているのか、等である。IDM システムはオーナーが不明確なアカウントを検出できる。セキュリティチームは大量のテスト用アカウントがあることに気づく。それらは明らかに、約半年前に発生した直近のデータセンター停止時にて使用されたものであった。データセンター停止後に IT オペレーションスタッフがこれらのアカウントのことを忘れてしまったのは明らかだ。そこでセキュリティスタッフは IDM ツールを利用してこれらのアカウントを無効にし、今後そのようなアカウントはないか監視する自動プロセスを設定する。



IDM データをもとに、セキュリティ担当者は特権を持ちすぎているユーザーがいると疑いをもつ。これは十中八九、要求承認プロセスの結果であり、単に特権が長期間かけて蓄積されていったのだろう。しかしこれは推測にすぎない。特定のユーザーが特定の特権をもっていることが正しいかどうか、セキュリティスタッフが判断するのは常に難しい。特に ExAmPLE 社のような柔軟な組織では業務責任が増えていくことが多く、組織構造がどことなくあいまいなためそうした判断は難しい。しかし各従業員が何をすべきかを把握している人たちがいる。それは彼らの管理者である。とはいえ多くの部署に多くの管理者がいるため、彼ら一人一人と話をして特権について調べるのは骨の折れる作業である。ここで再び IDM システムが登場し助け舟をだしてくれる。セキュリティ担当者は自動アクセス再認定キャンペーンを設定する。そして IDM システムに保持されている組織構造をもとに全ユーザーを管理者ごとにソートする。各管理者は部下であるユーザーとそれぞれの特権が掲載されたインタラクティブなリストを受け取る。管理者はユーザーがリストのとおりの特権を今も必要としていることを確認（再証明）しなければならない。このキャンペーンは非常に効率的に実施される。作業が組織全体にむらなく分配されるからだ。よって 2～3 日でキャンペーンは完了する。ついにセキュリティ担当者はどの特権がもはや不要で削除してよいか分かる。こうしてアセットの露出が抑えられる。これはセキュリティの残存リスクを低減する大変効率のよい方法である。

アイデンティティ管理技術はどう機能するのか？

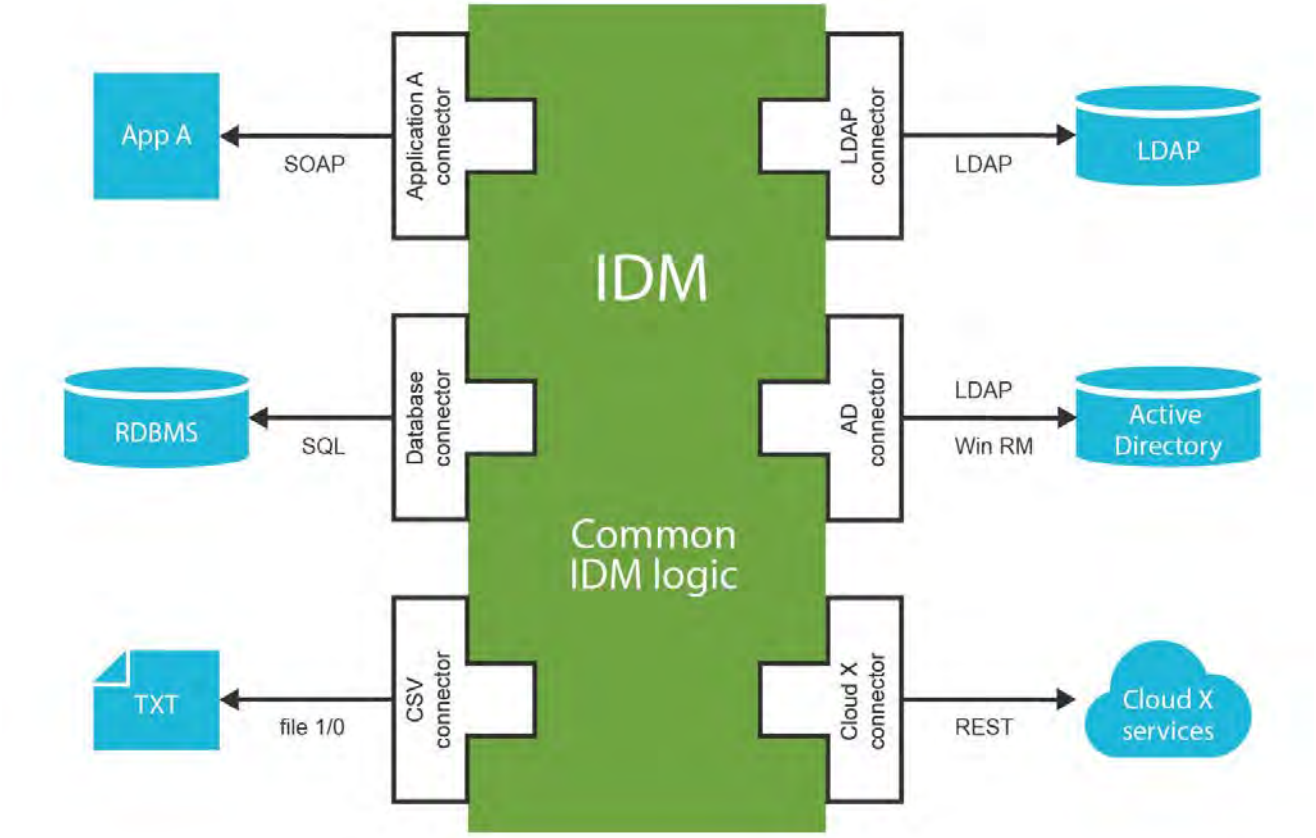
アイデンティティ管理システムが業務、プロセス、効率、その他すべてにとってあらゆるメリットがあることは明らかだ。しかし同システムは実際、技術レベルでどう機能するのか？基本原則は実にシンプルで、アイデンティティ管理システムはまさに洗練されたデータ同期エンジンである。

アイデンティティ管理システムは人事データベース等のソースシステムからデータを取得する。そして必要に応じてそのデータを処理し、値をマッピングする。どのデータが新しいか分かるだろう。IDM エンジンはレコードに何らかの（多くは実に複雑な）処理を行う。通常その処理にはロールベースのアクセス制御（RBAC）、組織ポリシー、パスワードポリシー等の処理ポリシーが含まれている。この処理による結果は、多くはアクティブディレクトリや CRM システムといった他システムのユーザーアカウントの作成もしくは変更である。つまり基本的にはデータを取得し、変更し、そして移動させることにほかならない。これでは大して画期的ではないと思われるだろう？しかし肝心なのはその詳細である。IDM システムがどうやってデータを集め、どうやってそのデータを処理し、変更を伝達しているかという点が大きく違うのである。

アイデンティティ管理コネクタ

アイデンティティ管理システムはさまざまなアプリケーション、データベース、情報システムに接続しなくてはならない。典型的な IDM デプロイではそのような接続が数十または数百にもなる。そのため、いかに簡単に IDM システムをその環境に接続できるかが必要な資質の一つとなってくる。

現行の IDM システムは*コネクタ*を使用してすべての周辺システムとの通信を行っている。これらのコネクタはデータベースドライバと同様の原則にもとづいている。片方の端には、同じ「プロトコル」を使用して全システムからデータを与える統一されたコネクタインターフェイスがある。そしてもう片方の端には、アプリケーションが対応する固有のプロトコルがある。よって、LDAP、さまざまな LDAP 派生、SQL プロトコル、SQL 言語用のコネクタ、ファイルベースのコネクタ、Web サービスあるいは REST サービスを呼び出すコネクタなどがある。ほんの少し高度な IDM システムにさえも、数十ものコネクタがある。



通常、コネクタは比較的シンプルなコードで、通信プロトコルに適応することが典型的な責務である。よってLDAPコネクタは、共通のコネクタインターフェイスを使用してLDAPプロトコルメッセージを表現されるデータへと変換する。SQLコネクタはSQLベースのプロトコルで同じことを行う。また、共通のコネクタインターフェイス上で IDM システムによって呼び出された操作もコネクタが解釈する。よってLDAPプロトコルはLDAP「追加」メッセージをLDAPサーバーに送信し応答を解析することで「作成」操作を実行する。一般にコネクタは作成（Create）、読み取り（Read）、更新（Update）、削除（Delete）という基本の操作（CRUD操作）を実施する。そのため典型的なコネクタはさほど複雑ではない。ただシンプルでありながら、その全体思想は巧みである。IDMシステムは通信の細かい部分を扱う必要はない。IDMのコアは、通常はそれだけでも非常に複雑な、アイデンティティ管理の汎用ロジックに自由に集中できる。よってコネクタがもたらしてくれる簡素化はなんでも大歓迎である。

一般に、コネクタはソースシステムおよびターゲットシステムの外部インターフェイスにアクセスしている。公開され、文書での裏付けが十分であり、理想を言えばオープン規格にもとづいているようなインターフェイスをコネクタの作成者が選ぶのは当然のことだ。そして実際に多くのシステムがそのようなインターフェイスを有している。ただそのようなインターフェイスの提供を拒む、悪評高きケースもある。しかしたいていは必ず何らかの方法がある。コネクタがアプリケーションデータベースに直接レコードを作成してもよい。データベースルーチンを実行してもよい。アカウント管理用のコマンドラインツール

を実行してもよい。はたまた、ユーザーがテキスト端末を使ってフォームに入力し新アカウントの作成をシミュレーションするといった信じがたいことすら実行してもよい。コネクタが実行すべきことを行う方法は必ずといっていいほど存在する。一部の方法は他の方法よりもすぐれている。

コネクタベースのアーキテクチャはすべての高度なIDMシステムの間でほぼ標準である。しかしコネクタのインターフェイスはかなり異なっているため、異なる IDM システム間でコネクタを相互に交換することはできない。コネクタのインターフェイスはすべて独自のものである。そしてコネクタはしばしば、IDM ソリューションのデプロイによる利益をなんとか疑似的に増やす武器として使用される。一つのケースを除いては。ConnId コネクタ（アイデンティティ管理用コネクタ）フレームワークは、競合する複数の IDM システムによって積極的に使用され開発されている唯一のコネクタインターフェイスである。そしてもちろん、オープンソースのフレームワークである。

コネクタベースの手法がかなり広く普及しているとはいえ、旧式の IDM システムの中にはコネクタを使用していないものもある。コネクタではなくエージェントを使用するIDM製品もある。エージェントはコネクタと同様のジョブを行う。しかしエージェントはIDMシステムのインスタンスの一部ではない。エージェントは接続された各アプリケーションにインストールされ、リモートネットワークプロトコルを使用して IDM システムと通信する。これが大きな負荷となっている。エージェントはいたるところにインストールする必要がある。メンテナンスとバージョンアップも必要である。また微妙な非互換性などもあるかもしれない。各アプリケーション内でサードパーティのコードを実行することも大きなセキュリティ問題になりうる。全体的に見て、エージェントベースシステムを稼働させることは面倒（かつコストもかかり）すぎる。おそらくエージェントの全体構想は、アプリケーションおよびデータベースがネイティブリモートインターフェイスに対応していなかった過去のデジタル時代に生まれたものであろう。そのような状況では、コネクタよりもエージェントの方がふさわしいのは当然だ。ただ、これは幸いにも過去の話である。今日では古いアプリケーションですらリモートインターフェイスを用いてアイデンティティを管理する手段を有している。多くの場合コネクタからのアクセスが容易な web サービスまたは REST サービスである。しかし、アプリケーションの提供するものがたとえコマンドラインのインターフェイスまたは対話型端末セッションだけだとしても、これを十分に扱えるコネクタがある。それゆえ今日では、一般にエージェントベースのシステムは時代遅れと考えられている。

アイデンティティプロビジョニング

おそらく、IDM システムで最も頻繁に使用される機能はプロビジョニングであろう。プロビジョニングの一般的な意味は、アプリケーションやデータベース等といったターゲットシステムのユーザーアカウントのメンテナンスである。これには、アカウントの作成、アカウントの有効期間中に行われるあらゆる変更、有効期間の終了時に行われる恒久的無効化または削除が含まれる。IDMシステムはコネクタを使用してアカウントを操作している。そして実際に、すぐれた IDM システムが管理できるのはアカウントだけではない。わずか数年前までは、グループおよびグループメンバーシップを容易に管理できる機能は実に稀であった。しかし今日では、グループを管理できない IDM システムはほとんど役に立たない。ほぼすべての IDM システムがロールを取り扱う。しかしロールのプロビジョニングと同期まで行えるのはほんのわずかである（例えば新たなロールごとに LDAP グルー

プを自動作成するなど）。すぐれた IDM システムは組織構造の管理、プロビジョニング、同期も行える。ただこうした機能はまだ完全に一般的にはなっていない。

同期およびリコンシリエーション

アイデンティティプロビジョニングはしばしば、IDM システムの最も重要な機能とみなされる。そしてプロビジョニングは間違いなく IDM システムに不可欠な機能である。といってもプロビジョニング以外は一切行わない IDM システムであれば、それはすぐに完全な失敗となろう。新入社員雇用時のアカウント作成、または退職時のアカウント削除だけでは不十分である。極めて短時間で大混乱に陥ってしまうパターンが数々待ち構えているのが現実だ。もしかするとアプリケーションの一つがクラッシュし、データがバックアップから復旧されたかもしれない。すると数時間前に削除されたアカウントが予期せず復活する。そしてそれはそこに留まり、有効であり続け、確認もされず危険である。人事スタッフはみな書類処理に忙しいため、もしかすると管理者が新アシスタント用のアカウントを手動で作成したかもしれない。新アシスタントはとてもかわいらしい目をしていたというのもある。レコードがようやく人事システムへと到着し処理されるときになって、IDM システムは矛盾するアカウントの存在を検出し、処理を止めてしまう。未熟なシステム管理者が新システムの運用ルーチンを試そうとして（百ほどの）アカウントをうっかり削除してしまったかもしれない。このように間違いが起きかねないパターンはあまりにも多い。そして実際のところ、失敗が驚くほど頻繁に発生している。IDM システムが設定を行い、あとはそのことについては忘れるというだけでは不十分である。自立した IDM システムの最も重要な機能の一つは、すべてが順調であり、そして常に正確であることを確実にすることである。アイデンティティ管理とは、アイデンティティを絶えずメンテナンスすることに他ならない。それがなくては IDM システム全体がほとんど役に立たないのだ。

データを正しく保つ方法はデータに変調をきたした時を知ることである。別の言い方をすれば、IDM システムはアプリケーションデータベースのデータに変更があったときを検出しなければならない。IDM システムが変更を検出すれば、その変更に対応するのはそれほど難しくない。つまり変更の検出がカギである。しかしこれにもわずかながら問題がある。変更が発生するごとにアプリケーションが IDM システムに通知するなんてことは期待できないことだ。アプリケーションに手を加えるようなことは避けたい。さもないと IDM デプロイコストが極端に高くなるからだ。アプリケーションはあくまで受動的、そして IDM システムがアクティブであるべきなのだ。そして幸いにもそうする方法はいくつか存在する。

すでに変更を常に把握しているアプリケーションもある。行ごとに最終変更のタイムスタンプを記録しているデータベースもある。レプリケーション目的として最近の変更記録をとっているディレクトリサーバーもある。IDM システムはこのようなメタデータを利用できる。一般に IDM システムは定期的に新規変更のタイムスタンプまたはレプリケーションログをスキャンしている。変更を検出した IDM は変更されたオブジェクトを取得し、そのポリシーに従ってその変更に対応する。メタデータにもとづく変更スキャンは非常に効率的で、数分おきまたは数秒おきに実行することもできる。これによりほぼリアルタイムで変更に対応できる。この方法はさまざまな IDM システムで色々な呼び方がある。「ライブ同期（live synchronization）」、「アクティブ同期（active synchronization）」、または単に「同期（ synchronization）」である。一般的にはこれが好ましい方法である。ただ残念なことに、常に利用できるとは限らない。実際、この機能は極めてまれである。

アプリケーションがほぼリアルタイムで変更を検出できる優れたメタデータを保持していなくても、ほぼすべてのシステムに有効な方法が一つある。その原則は実にシンプルだ。IDM システムがアプリケーション内の全アカウントのリストを取得する。そしてこのリストと、本来あるべきとされるアカウントのリストとを比較する。つまり、ポリシー（*あるべきもの*）と現実（実際に*あるもの*）を比較するということである。不整合があればこれに対応し修正できる。この方法はリコンシリエーション（Reconciliation）と呼ばれる。リコンシリエーションはこのジョブを行うが、それは極めて荒っぽい方法である。全アカウントをリスト化し各アカウントを処理することは簡単明瞭なジョブのように思われるかもしれない。しかしアカウント数が多くポリシーが複雑であれば、このジョブは極端に遅くなりうる。数分のこともあれば、数日かかる可能性もあるのだ。そのため頻繁には実行できない。小規模で単純なシステムでもないかぎり 1 日 1 回はできない。より一般的には（週末に）週 1 回であろう。しかし多くのシステムは月 1 回がせいぜいであり、それより頻繁に行う余裕はない。

他にも方法はある。しかし最もよく使用されているのは同期およびリコンシリエーションである。同期の欠点は、信頼性が完全でないことである。変更ログの期限が切れる、システム時間の同期がとれていないなどの様々な理由により、IDM システムが一部の変更を見過ごしてしまう可能性がある。リコンシリエーションは同期よりも信頼性は高いが、非常に負荷の大きいタスクである。そのためこれら 2 つは合わせて使用されることが多い。同期は常に実行され変更の大部分を取り扱う。リコンシリエーションは週次または月次で実行され、同期にてすり抜けた変更を拾い上げるセーフティネット的な役割を果たす。

すぐれた IDM システムは変更検出に使用した仕組みが何であれ、一貫して変更を取り扱う。こうしたシステムはただひたすら同じポリシーを実行する。しかし（とりわけ旧式の）IDM システムの中には、仕組みごとに別のポリシーをもつものもある。これが IDM システムの設定およびメンテナンスをひどく複雑にしている。

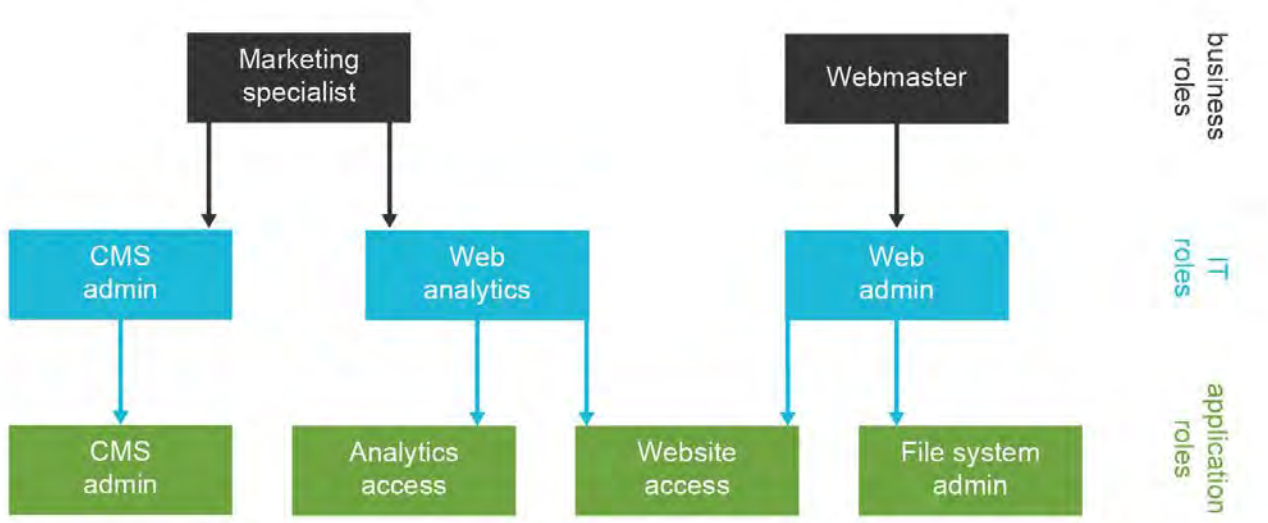
同期およびリコンシリエーションは実に強力な仕組みである。これらを使用すれば不正アカウントをスキャンし自動で無効にすることができる。アプリケーション管理者が実施した変更を検出し、その内容に沿ってユーザープロファイルを更新することもできる。どちらにしても、これは IDM データと現実との整合性を保つための重要な仕組みである。

アイデンティティ管理とロールベースアクセス制御

各ユーザーのアクセス権限をそれぞれ管理することはユーザー数が非常に少ない場合でなければできない。わずか数百人でさえ個別の管理は非常に困難となる。一般にユーザー数が千人を超えるとその負担は耐え難いものとなる。アクセス権限の個別管理は作業量が多いことはもちろん、誤りも生じやすくなる。これはもう何十年も前から認識されてきたことである。その結果、多くのシステムで共通のアクセス権限の組み合わせが生み出され、ロール（役割）が作成された。そして「ロールベースのアクセス制御（Role-Based Access Control：RBAC）」という概念が生まれた。通常、ロールとは職務や責任を表わし、アクセス権限よりも「業務」にはるかに近い。一つのロールで銀行の窓口係、ウェブサイト管理者、またはセールスマネージャーという概念を表すこともある。ユーザーにはアクセス権限ではなくロールが付与される。ロール自体に認可時に使用するアクセス権限が含まれている。ただし、こうした下位レベルのアクセス権限はユーザーからはわからないようになっている。ユーザーは業務上使用しやすいロール名を扱うので十分満足である。

注：用語について。「*RBAC*」という用語は当業界で頻繁に使用されているが、実際の意味は必ずしも明確ではない。おそらく、「*NIST RBAC モデル*」として知られる正式な RBAC 仕様書が存在していることが混乱を招いているのであろう。この具体的な正式モデルを指して RBAC という人もいれば、正式モデルに近いものをいう人もいる。あるいはロールを扱う何らかのものを指す人もいる。我々はかなり広義でこの RBAC という用語を使用する。主要なアイデンティティ管理システム（IDM システム）では、正式な NIST RBAC モデルからヒントを得て実装するのが一般的であるが、必要に応じて当モデルから逸脱することもある。これが我々の RBAC の意味するところである。

ほとんどの RBAC システムはロールの中にさらに別のロールを格納することができ、これによりロール階層が形成される。階層の最上位には「マーケティングスペシャリスト」といったビジネスロールがある。当ロールにはさらに下位ロールが含まれており、たいていは「ウェブサイトアナリティクス」や「CMS 管理者」といったアプリケーションロールである。これらの下位ロールには具体的なアクセス権限が含まれていることもあれば、さらに要素技術に近いロールが含まれていることもある。アクセス権限数およびユーザー数が増えていくような状況では、しばしばロール階層が必須となる。



RBAC の仕組みをもたない IDM システムは真に完全とはいえない。そのため大多数の IDM システムが何らかの方法でロールに対応している。とはいえ、その程度はシステムによって大きく異なる。「RBAC 対応」であることを主張するため必要最小限だけ対応している IDM システムもあれば、非常に優れた、先進的なダイナミック（動的）かつパラメトリックなハイブリッド型 RBAC システムをもつ IDM システムもある。ただ、多くはその 2 パターンの中間あたりに位置する。

ロールベースの仕組みは管理ツールとして実に有効である。実際に、その有効さゆえにしばしば乱用されてしまうこともある。これはとりわけ規模が大きく幾分複雑な環境であるほど真に危険なものとなる。複雑な環境下でのロール設計者は、秩序を保とうとする強い気持ちゆえ、ロールを再利用できる最小単位までいったん細分化してから、アプリケーションロールおよびビジネスロールの形に再度編成する。これが「最小権限の原則」といったセキュリティの模範事例によってさらに展開される。こうした設計者の気持ちはもっとも

であり至極正当なことである。ただし、そのような RBAC システムを維持させるには細心の注意が必要となる。直感的には分かりづらいかも知れないが、ロール数がユーザー数を超えてしまうことは実によくあることなのである。残念ながらこの方法ではユーザー管理の複雑な問題をさらに複雑なロール管理の問題へと発展させてしまう。この現象は「*ロールエクスプロージョン*」として知られている。

ロールエクスプロージョンは本当に危険であり、また決して容易に避けられるものではない。第一世代の IDM のデプロイ時に広く普及した方法は、このエクスプロージョンをただ甘んじて受け入れることであった。中には数十万のロールを自動で作成、（多少なりとも首尾よく）管理できるツールさえ構築するデプロイもあった。しかしこれは持続可能な方法ではない。第二世代の IDM システムでは最初からロールエクスプロージョンの回避をサポートするような機能が備わっている。一般にこのような仕組みは、ロールをさらに動的化するという考えがベースとなっている。ロールはもはや単に特権を束ねたものではない。特権の組み立て時に使用する小さなアルゴリズム論理もいくつか含まれたものとなっている。このアルゴリズムへインプットするのがロール付与時に指定するパラメータである。これによってロールをコピーすることなく、関連する多くの目的に同一ロールを再利用できる。複雑なシステムのモデリングに必要なロール数を大幅に抑え、ロールエクスプロージョンに対する最高の対抗手段となっている。

欠点こそあるものの、ほとんどの実用的な IDM ソリューションには欠かせないのが RBAC システムである。これまでこのシステムを全く別の手法と置き換えようとする試みもいくつかあった。その中にはアクセス管理および関連分野において何らかの成果を挙げているものもある。しかし、アイデンティティ管理においてはそうやすやすと RBAC に取って代わることはできない。ありふれた例の一つが属性ベースアクセス制御（Attribute-Based Access Control：ABAC）である。ABAC はロールを完全にアルゴリズムポリシーへと置き換えることにもとづいた概念である。簡単に言えば、ABAC ポリシーとは、インプットとしてユーザー属性を取得し、それを操作およびコンテキストに関するデータと組み合わせ、アウトプットとして操作の許可／拒否の認可判定を行うという一連のアルゴリズムである。この手法はシンプルで、実行される操作をアクセス管理サーバーが熟知しているようなアクセス管理の環境であれば実に合理的に機能する。しかし IDM 分野ではユーザーの初回ログイン前にアカウントを設定する必要がある。この時点で操作に関するデータは一切ない。さらにコンテキストデータでさえ非常に制限されている。それだけでなく他にも課題があり、ABAC は IDM システムには実にふさわしくない選択肢となってしまっている。よって好むと好まざるとにかかわらず、RBAC は有用な IDM ソリューションをもたらす最も重要な仕組みであり、根付いているのである。

アイデンティティ管理と認可

情報セキュリティにおける認可の基本原則は実に単純明快である。サブジェクト（ユーザー）、オブジェクト（ユーザーがアクセスしようとしている対象）、そして操作を取得することである。このサブジェクト、オブジェクト、操作の 3 つの組み合わせがポリシーで認められているか評価する。認められていなければ拒否する。実にシンプルである。しかしアイデンティティ管理の分野ではまったく違うように考える必要がある。我々は逆にたどる必要がある。IDM システムはユーザーが操作を始める前に当人のアカウントを設定する必要がある。そして実際に操作が始まると、それについては何も把握しない。よって IDM 環

境における認可の概念はとにもかくにも逆になっている。

IDM システムは認可に直接携わることはなく、アプリケーションとデータベース内にアカウントを設定する。しかしユーザーがアプリケーションにログインし操作を実行するとき、IDM システム自体はアクティブではない。ということは、IDM システムは認可に関して何もできないということか？もちろんそんなことはない。たしかに認可の判断を行うわけではない。しかし認可をどう評価するか決定するデータを管理できる。IDM システムはアカウントを適切なグループに置くことができ、それによってある操作は許可され、ある操作は拒否されるようになる。管理するアカウントごとにアクセス制御リスト（access control lists：ACL）を設定できる。直接認可を評価したり実行したりすることはないが、認可の評価に使用するデータを間接的に管理する。これは非常に重要な機能である。

情報セキュリティにおいて認証と認可の２つは非常に重要な概念である。そしてアイデンティティ管理とアクセス管理のどのソリューションにとっても極めて重要である。しかし認証は実にシンプルである。たしかにユーザーは多要素認証の中で利用される数種類のクレデンシャルを使用しているかもしれない。そう聞くと少し恐ろしく思えるかもしれないが、それほど複雑なものではない。認証を管理するポリシー要綱はほんのわずかである。しかもたいていは、認可はまったく一様であり、大半のユーザーが同じ仕組みを利用して認証を行っている。認証の一元化はさほど難しくない（コストはかかるかもしれないが）。よって認証の管理は比較的容易である。

ただ、認可となるとまったく話が違う。それぞれのアプリケーションがもつ認可の仕組みは微妙に違っている。そしてそれらの仕組みを統一するのは容易ではない。大きな障害の一つは、各アプリケーションが異なるオブジェクトを扱っており、それらのオブジェクトはお互いに複雑な関係にあるものや、サブジェクトとも複雑な関係にあるかもしれないことである。操作も単純明快とはまったく言い難い。そしてコンテキストがある。操作ごとに制限があったり、日々の制限、所定の時間またはシステムが所定の状態にあるときのみ実行できる操作などがあるかもしれない。これでは一元化は非常に難しい。また、認可の組み合わせもほぼすべてのユーザーでわずかながら異なっている。つまりばらつきが大きく、管理ポリシーも多いということだ。そのうえ、まったく新しい次元の複雑さをもたらす重要な要素が２つある。パフォーマンスと拡張性である。認可判断は常に評価されている。各要求に対し認可を複数回評価している場合であっても、これは例外ではない。認可処理には速さが必要で、非常に高速でなければならない。ローカルネットワーク内の一往復ですらパフォーマンスキラーになりかねない。パフォーマンスや拡張性の理由から、認可の仕組みは個々のアプリケーションの構造内に強固に統合されていることが多い。たとえば、認可ポリシーを SQL 言語に翻訳し、アプリケーションレベルの SQL クエリにおける付加条項として使用することは一般的に行われている。この技法では、ユーザーのアクセスが認可されていないデータを素早く除外すべくデータベースを有効活用している。非常に効率的な技法であり、おそらく大規模なデータセットを扱う場合の唯一実用的な選択肢であろう。しかしこの手法はアプリケーションデータモデルと密接に結びついており、一般に外部に置くことはほぼ不可能である。

従って近い将来認可を一元化できるという期待は現実的ではないのである。認可ポリシーをアプリケーションに分配する必要がある。ただ、分配された部分的なポリシーを管理することは簡単ではない。組織のセキュリティ上の全体ポリシーに対するアプリケーションポリシーの整合性を誰かが確認しなくてはならない。幸いにも、IDM システムは特に幅広

いシステムにおいてデータの管理と同期を取り扱うようできている。そのため認可ポリシーの管理となれば IDM システムが選ばれるのは当然なのである。

組織構造、ロール、サービス、その他のワイルドライフ

2000年代を振り返ってみると、当時のIDMはユーザーアカウントの管理にほかならなかった。アカウントの作成、無効化、削除で十分であった。しかし今、世界は様変わりしている。アカウントだけの管理では全く不十分なのだ。たしかにアカウント管理の自動化は大きな恩恵をもたらし、複雑なシステムでは少なくとも最小レベルのセキュリティを確保することが必要条件である。しかしそれですら、IDM システムにかかるコストを正当化するにはしばしば不十分である。現に IDM システムは単にアカウント管理だけでなく、実に多くのことを行うことができる。新しい IDM システムはどれもそうである。ユーザーアカウントの管理しかできない旧式の IDM システムが近い将来完全に姿を消すだろうことはほぼ間違いない。

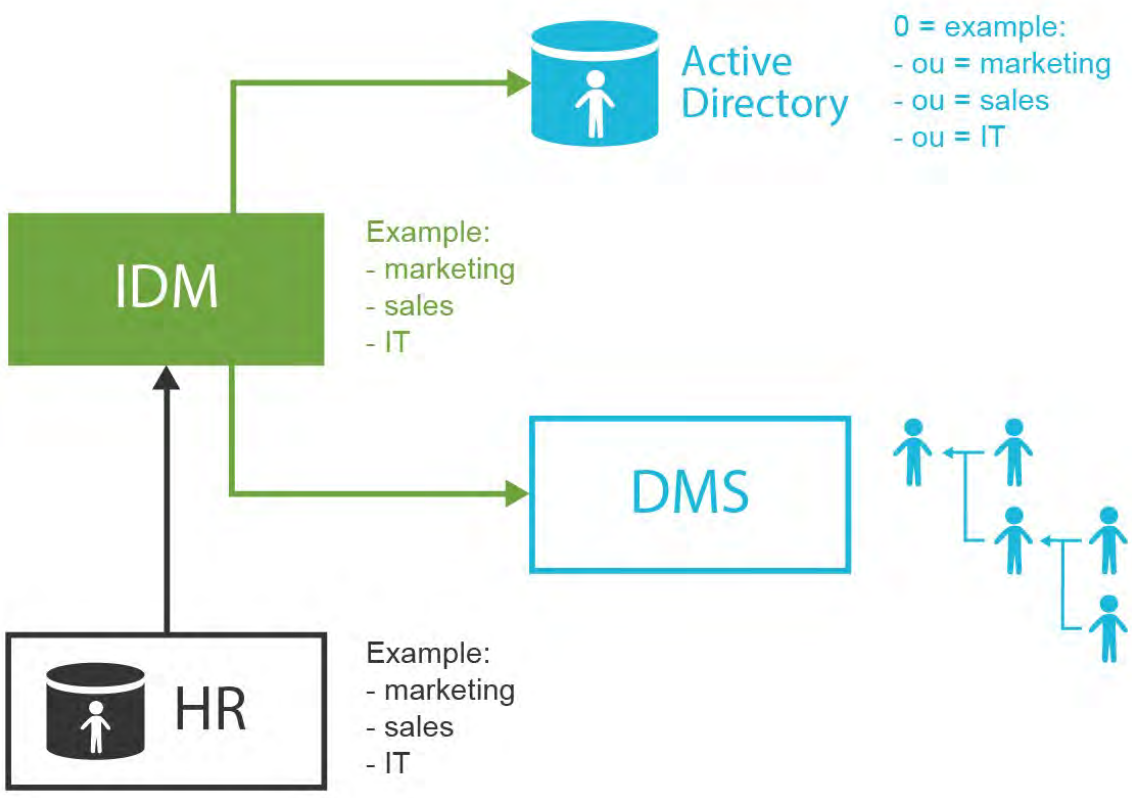
高度な IDM システムが管理できるものは実に多い。

- アカウント。これは言うまでもない。しかし単なるアカウント管理の時代はついに終わった。今や、アカウント属性、グループメンバーシップ、特権、アカウントステータス（有効／無効）、有効日、その他あらゆる詳細を完全に管理できることが標準である。
- グループおよびロール。グループ内のアカウントメンバーシップの管理だけでなく、グループの作成、管理、削除に至るまで、グループのライフサイクル全体を管理することができる。
- 組織構造。権威あるソース（多くは人事システム）から組織構造を取得し、これを必要とするアプリケーションすべてに同期させることができる。または IDM そのものを使用して組織構造をマニュアルで管理することもある。
- サーバー、サービス、「モノ」。まだ IDM の主流ではないが、IDM 原則を使用して従来の IDM の枠をわずかに越えた概念を管理する試験的なソリューションがいくつかある。たとえば新プロジェクトごとに定義済みの仮想マシンを自動でデプロイできる IDM をベースとしたソリューションがある。新 IDM システムは柔軟性があるため、わずかでもアイデンティティの概念に関わるものなら何でも、たとえば仮想マシン、ネットワーク、アプリケーション、構成、デバイスなどほぼすべてを理論上は管理できる。今はまだ極めて独特な機能だが、十中八九、将来はこれに関する話をもっと目にすることになるだろう。

どの機能もすべて興味深いが、特に際立ったものがいくつかある。グループ管理および組織構造はほぼすべての新IDMのデプロイに不可欠な機能である。ほぼフラットでプロジェクトを中心とした組織構造もあれば、12階層の部門や課から成る組織構造もあるだろう。しかしその規模に関係なく、組織構造は管理され、アイデンティティの同期とほぼ同じようにアプリケーション間で同期がとれている必要があるのだ。あなたは組織単位でアクティブディレクトリ内にグループを作成する必要があるかもしれない。そしてそれらを正しく入れ子にしたい。特別チームごとに配布リストを作成したいかもしれない。この作業にかかる経費はできるかぎり抑えたい。そうでないともはや一時的な特別チームではなくなっ

てしまうからだ。プロジェクトに関する情報を問題追跡システムに同期させたいかもしれない。また、新開発プロジェクトごとに別個の wiki スペースおよび新しいソースコードレポジトリを自動作成したいかもしれない。可能性は無限に近い。従来の組織であれ、無駄がなくアジャイルな新しい企業であれ、どちらもその恩恵を受けるだろう。

組織構造管理はグループ管理に密接に関係している。グループはワークグループ、プロジェクト、または組織単位に結びついていることが多い。例えば、IDM システムはプロジェクトごとに複数のグループ（管理グループおよびメンバーグループ）を自動でメンテナンスできる。こうしたグループは認可に使用できる。同様に、アプリケーションレベルでロール、アクセス制御リスト（ACL）、その他認可に使用されることの多いデータ構造を自動でメンテナンスできる。



この機能はほぼすべてのデプロイにメリットを与える一方、ツリーのような機能組織構造にもとづく組織にとっては絶対不可欠である。こうした組織は組織構造から導出された情報にかなり依存している。たとえば、文書作成者の直属の上司は文書管理システムで文書の内容を確認し承認できる。同一部門の従業員だけが文書のドラフト版を閲覧できる。マーケティング課の従業員だけがマーケティングプランを閲覧できる、といったことである。従来より、このデータは非包括的な認可グループおよびリストセットの中にエンコードされている。そしてこのことが、組織再編はひどい悪夢だという事実の一因となっている。ところが IDM システムであればこの状況を大幅に改善できるのだ。IDM はグループを自動作成できる。そうしたグループに確実に正しいユーザーをアサインすることができる。

上司に関する情報を関連するすべてのアプリケーションに同期させることができる、といったことである。そしてすぐれた IDM システムは、ごくわずかの構成オブジェクトを使用してこれらすべてを実行できるのだ。

これではあまりに話がうますぎるようだ。たしかに、これらの機能の対応品質が IDM システムによってかなり違うことは認めるのが公平というものだろう。グループ管理および組織構造管理は最も問題をはらんだ機能のように思われる。実用的で即時利用可能なデプロイを実現できるレベルでこうした概念に対応する IDM システムはほんのわずかに限られている。IDM システムの大半がそれについて何らかの対応をしているものの、実用的なソリューションには相当なカスタマイズが必要とされる。IDM ベンダーが、ほぼすべての IDM のデプロイに必要な機能になぜ目を向けようとしないのかは定かではない。そこで包括的な IDM ソリューションとなると、我々からアドバイスできる重要なことが一つある。それは、IDM 製品を賢く選ぶ、ということだ。

アイデンティティ管理は全員に必要

これは実に大げさに聞こえるかもしれないが、実際のところかなり真理に近い。ある程度複雑なシステムにはアイデンティティ管理が必要である。システムオーナーにはそのような認識がないかも知れないが。ただ、この本を読んでいるあなた方であれば、おそらくその必要性を認識できる数少ない一人になるだろう。その場合、大部分はコストとメリットとの兼ね合いである。アイデンティティ管理には特有の複雑さがあるのだ。非常に小規模なシステムでさえ IDM が必要である。とはいえそのようなシステムでは、コストを正当化するにはメリットが小さすぎるかもしれない。コストとメリットの兼ね合いは中規模システムのほうがずっといい。そして大規模システムであれば IDM は必須である。ここに一つ、幅広く適用できそうな経験則がある。

ユーザー数	
200 人未満	IDM は必要かも知れないが、おそらくコストを正当化できるほどメリットは大きくないだろう。
200 – 2 000	IDM が必要であり、ちょうどコストを正当化できるくらいメリットも十分であろう。ただしコスト効率のよいソリューションを探す必要がある。
2 000 – 20 000	IDM は確実に必要だ。これほどの大人数を手動で管理することなど絶対にできない。正しく実装すれば、コストよりもはるかに大きなメリットが得られるだろう。
20 000 人超	まだ IDM がないとは信じられない。すぐに入手すべきだ。それも今すぐに。私に感謝するのは後でいい。

アイデンディティのガバナンスとコンプライアンス

基本的に、*アイデンディティガバナンス*はアイデンティティ管理をより高いビジネスレベルへ引き上げたものである。アイデンティティ管理では、自動プロビジョニング、同期、ロールの評価、属性の計算など、主にアイデンティティライフサイクルの技術面に重点が置かれている。アイデンディティガバナンスは技術的な詳細から引き出されたもので、ポリシー、ロール、プロセス、データ分析に重点が置かれている。例えばガバナンスシステムは職務分掌ポリシーを取り扱うことができる。アクセスを再認定するプロセスを促進できる。アイデンティティの自動分析および報告、監査、ポリシーデータに重点を置くこと

ができる。ポリシー違反に対処する是正プロセスを促進する。新ポリシーおよび変更後ポリシーの適用を管理し、ポリシーや規定等に対するシステムの遵守状況を評価する。この分野は*ガバナンス・リスク管理・コンプライアンス*（*governance, risk management and compliance*：GRC）と呼ばれることもある。

少なくとも、実用的なデプロイに役立つ何らかのガバナンス機能がほぼすべての IDM システムには必要だろう。そして多くのガバナンス機能が、2〜3 年前の IDM 分野に由来する概念を改良したものである。それゆえ、アイデンティティ管理とアイデンティティガバナンスの境界はかなりあいまいである。あまりにあいまいなため、アイデンティティガバナンスとアイデンティティ管理が一緒に含まれる統合分野に新たな用語も生まれた。その一つが*アイデンティティのガバナンスと管理*（*Identity governance and administration*：IGA）である。この分野（またはサブ分野）はまだかなり日が浅いため、専門用語はもちろん、概念でさえ落ち着くまで少し時間がかかると思われる。このガバナンスは、我々にとってはアイデンティティ管理の進化した延長にすぎない。

しかしガバナンス機能は、基礎となる IDM プラットフォームからは離れた特殊な製品によって実装されているのが一般的のようだ。ほぼすべての商用 IDM およびガバナンスソリューションが（少なくとも）2 つの製品に分れている。この戦略のおかげでベンダーが新たな収入源を得るのは明らかだ。しかし顧客側からすればほとんど意味をなさない。業界は*クローズドループ型の是正*（*Closed-loop remediation*：CLR）という用語まで造った。実際にはこれは、ガバナンスシステムがその基礎となる IDM ソリューションと何とか統合されていることを意味する。時に業界は、妥当なソリューションの自然な部分であるはずのところに、やたらに凝った用語を創作する必要がある。妥当なソリューションとは、統合され正しく調整された一つの製品において IDM 機能およびガバナンス機能の両方を提供するものであることは言うまでもない。

アイデンティティ管理とアクセス管理の完全なソリューション

アイデンティティ管理とアクセス管理の包括的なソリューションは単一コンポーネントでは構築できない。それ一つで必要な機能すべてを提供する製品やソリューションはないからだ。しかも要件は非常に複雑でしばしば相反するものもあるため、それらすべてに対応できる単一製品がでてくるとは到底思えない。

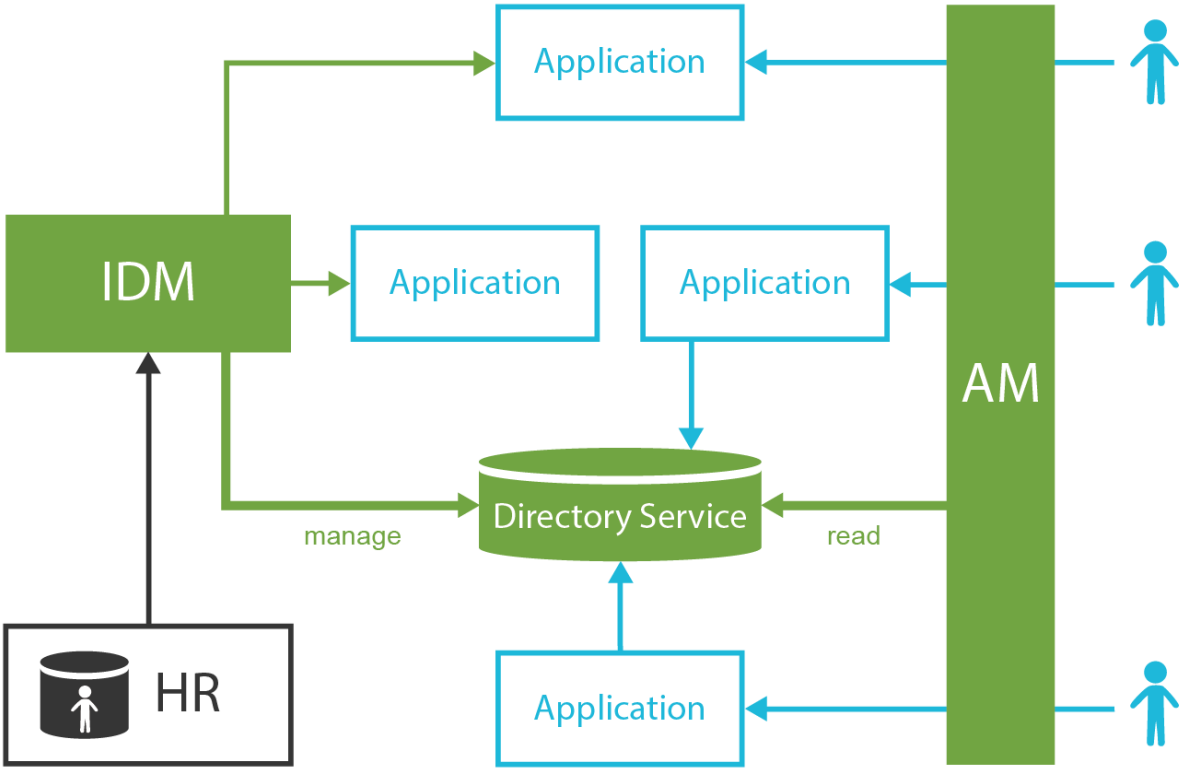
完全なソリューションを構築するには、いくつかのコンポーネントを賢く組み合わせる必要がある。この組み合わせは必ず若干異なってくる。2 つとして同じ IAM ソリューションはないからだ。ただ、実用的な IAM デプロイに必要とされる基本コンポーネントが 3 つある。

- **ディレクトリサービス**または類似のアイデンティティストア。まず一つめのコンポーネントであり、ユーザーアカウント情報を格納するデータベースである。アカウントは他のアプリケーションでも使用できるよう「クリーン」な形式で格納される。たしかにこのデータベースは、このデータベースに接続できるアプリケーションによって広く共有される。一般的には LDAP サーバーの複製トポロジまたはアクティブディレクトリのドメインとして実装される。これは比較的安価でありながら高い可用性という強みがある。とりわけ LDAP サーバートポロジはうんざりするほど拡

張できることが多い。ただし大きな制限事項が一つある。データモデルをシンプルにする必要があるということだ。非常にシンプルに。そしてアイデンディテストアも適切に管理する必要がある。

- **アクセス管理**。ソリューションに必要な２つめの主要コンポーネントであり、認証および（部分的に）認可を取り扱う。アクセス管理は認証の仕組みを統合する。アクセス管理サーバーに認証の仕組みが一つ実装されていれば、統合されたすべてのアプリケーションが簡単にメリットを受けられる。また、シングルサインオン（Single Sign-On：SSO）を提供しアクセスログ等を一元化する。実に役立つコンポーネントである。しかしもちろん制限事項もある。このコンポーネントはアイデンティティデータへのアクセスを必要とする。つまりバックエンドとして信頼性、高い拡張性、確実な整合性を備えたアイデンティティデータベースが必要なのだ。一般にこれはディレクトリサービスによって提供される。ここで明らかな障害となるのがパフォーマンスと可用性である。しかし障害となるのがもう一つある。先の２つよりも少しわかりにくいが、どこから見ても重要なもの、データ品質である。ディレクトリサービスのデータは常に最新状態を保ち、適切に管理されなくてはならない。しかしこれは大局の一部にすぎない。多くのアプリケーションがアイデンティティデータの一部をローカルに保存しており、こうしたデータもディレクトリデータベースと同期させる必要がある。これを十分できるアクセス管理システムはない。そしてアクセス管理がこれを行う意味は全くない。もっとちがう、アーキテクチャ上の責任があるからだ。そのためさらにもう一つ別のコンポーネントが必要となる。
- **アイデンティティ管理**。最後に挙げているが、多くの面で最も重要とされるコンポーネントであり、ソリューション全体の真の頭脳部分である。IDM システムはデータを保守する。システム全体がばらばらになってしまうのを防ぐコンポーネントである。確実にデータを最新の状態の保つとともにポリシーに適合させる。小さくてやっかいなアプリケーションがひたすら作成し続けるすべてのアイデンティティデータを同期させる。グループ、特権、ロール、組織構造など、ディレクトリおよびアクセス管理が適切に機能すべく必要なすべてを保守する。システムの秩序を保守する。さらに生身の人間であるシステム管理者およびセキュリティ担当者は楽しく時を過ごし、息が切れることもなく、ソリューション全体を制御し続けることができる。

これらすべてのコンポーネントがどう連携しあうかを示したのが下図である。



これはまさにコンポジットソリューションである。いくつかコンポーネントがあり、それぞれの機能と特徴は大きく異なっている。しかしこれらが一つのソリューションの中で結ばれたとき、それはそれぞれを単に足し合わせたものよりはるかにすぐれたものとなる。コンポーネントはお互いを支え合っている。3 つのコンポーネントがすべて揃って、このソリューションは完全になる。

ただ完全なソリューションを構築するにはコストも時間もかなりかかるかもしれない。しかしとにかく始めなくてはならない。千里の道も一歩からだ。もし製品一つ分のリソースしかないのであれば、アイデンティティ管理を選ぶべきだ。IDM はその手始めとして適している。アクセス管理ほど高価ではない。しかもかなり早い時期に IAM プログラムにすばらしい価値をもたらしてくれる。特に第二世代の IDM システムは実に投資効果に優れている。オープンソースを選べば初期投資も低く抑えられる。IAM プログラムを始めるなら IDM からスタートするのが最善の選択である。

IAM およびセキュリティ

厳密に言えば、アイデンティティ管理とアクセス管理（Identity and Access Management : IAM）は情報セキュリティ分野の中に完全に収まっているわけではなく、これをはるかに凌駕している。IAM はユーザーに心地よさをもたらし、運用コストを減らし、プロセススピードを上げ、一般に組織効率を高めることができる。これらは情報セキュリティが関与するものではない。しかし IAM が厳密には情報セキュリティの一部ではないといっても、大きく重なる部分はある。IAM は認証、認可、ロギング、ロール管理、情報セキュリティに直接関わるオブジェクトのガバナンスを取り扱う。そのため、IAM と情報セキュリティ

は固く複雑な関係で結ばれているのだ。

すぐれた情報セキュリティには IAM が必須と言っても決して大げさではないだろう。特にアイデンティティ管理（IDM）の部分は絶対に不可欠である。一見しただけでは分かりづらいかも知れないが、これは実にはっきりしている。セキュリティ研究によれば、組織にとって最も深刻な脅威の一つはかなり一貫して内部脅威であるとされている。しかし内部脅威に対し実行可能な技術的対策はあまりない。従業員、請負業者、パートナー、サービスエンジニアといった人員はみな、システムに容易かつ合法的にアクセスできる権利を持っている。彼らは最も強力な暗号と認証さえ正当に通過する。そのキーをもっているからだ。ファイアウォールおよび VPN は彼らを通過させることが目的のため、止めることはない。

ここに脆弱性が存在するのは明らかだ。そしてユーザーが数千人いれば、攻撃者が現れる可能性は確かに存在する。たとえばあるエンジニアが昨日解雇されたとしよう。しかし彼はまだ VPN にアクセスでき、サーバーの管理者権限を有している。彼がもし解雇されたことにまったく納得していないとしたら、おそらくあなたを少し痛い目にあわせたいと強く思うだろう。ひょっとしたら企業レコードの一部を漏らそうと企むかもしれない。つまり動機を持った攻撃者がいることになり、しかも彼はどんな対策からも阻止されることなくアセットに容易にアクセスできるのだ。これがはたしてどんなことになるか、セキュリティ担当者なら総合リスク分析を行わなくても容易に予測できる。

内部脅威に対しては情報セキュリティに明確な答えはない。これは簡単に解決できる問題ではない。明らかに大きなセキュリティトレードオフ（二律背反）があるからだ。ビジネスとしては業務の円滑化と組織の運営継続のため、ユーザーには簡単にアセットにアクセスしてもらいたい。しかしセキュリティ側としては、まさにそうしたユーザーからアセットを守る必要がある。この問題を解決できる特効薬はない。しかしこれを可能にすることがいくつかある。

- **誰が何に対しアクセス権をもっているか記録する。**各ユーザーは、企業を通じて多くのアプリケーションに複数のアカウントを持っている。どのアカウントがどのユーザーのものか把握する。これを手動で行うのは非常に難しい。しかし IDM システムならば、たとえ最低限のシステムであろうと簡単に対応できる。
- **アクセス権を迅速に削除する。**セキュリティインシデントの発生時は秒単位でアクセスを削除する必要がある。従業員が解雇されたら、当人のアカウントは分単位で無効にしなければならない。これをシステム管理者が手動でおこなうこと自体は問題ではない。しかし果たして、セキュリティインシデントが夜遅い時間であっても管理者は対応できるのか？従業員の解雇をシステム管理者の労働時間に合わせられるのか？システム管理者は、ユーザーが残したかもしれないプロセスとバックグラウンドジョブをすべて忘れずに停止できるだろうか？IDM システムであればこれらが簡単にできるのだ。セキュリティスタッフは IDM システムを使用してすべてのアカウントを無効にするだけでよい。そのために必要なのはシングルクリックだけである。
- **ポリシーを適用する。**ユーザーにアサインされた特権について記録しておく。通常

これはユーザーへのロール（およびその他のエンタイトルメント）のアサインを管理することを意味する。機密性の高いロールであれば、必ず先に付与の承認を得てからユーザーにその特権を与えること。ポリシーと現実を比較する。アカウントを作成しエンタイトルメントをアサインするシステム管理者はロボットではない以上、間違いも起こりうる。間違いは必ず見つけ出し修正する。これがベストプラクティスである。ただ、これを手動で行うことはほぼ不可能だ。しかし IDM システムならば、それが平均的なシステムにすぎなくても、何ら問題なくこれに対応できる。

- **不要なロールは削除する。**ロールとエンタイトルメントはいつの間にか蓄積されていく。長期間在籍している従業員は、ほぼすべてのアセットにアクセス権をもっていることが多い。単に在籍中のある時点でそうしたデータが必要だったからだ。その後アセットへのアクセス権は削除されなかった。これが大きなセキュリティリスクとなっている。エンタイトルメントを再確認する紙ベースでのプロセスを構築すればこのリスクは緩和できる。とはいえそのようなプロセスは非常に時間とコストがかかり、エラーも発生しやすい。また定期的に繰り返さなくてはならない。しかし高度な IDM システムであれば、すでにこの再認定プロセスの自動化に対応している。
- **秩序を保つ。**もし最小特権の原則に忠実に従っているのであれば、おそらくより多くのロールがあることにお気づきだろう。ロールは抽象的な概念であり常に進化し続けている。経験豊富なセキュリティの専門家ですら、ロール階層とロール構造の中では簡単に迷ってしまうことがある。普通のエンドユーザーは自分にはどのロールが必要なのかまったくわかっていないことも多い。しかし、すぐれた IDM システムにロールを保持していれば、それらのロールをカテゴリ別で並べ替えることなど大して難しいことではない。これによりロールカタログが作成され、ずっと簡単にロールを理解し使用できるようになる。
- **記録に残す。**特権変更はすべて監査レコードに残す。つまり、新アカウント、アカウントの変更、削除、ユーザー名やアカウント名の変更、ロールアサイン、アサイン解除、承認、ロール定義変更、ポリシー変更などのすべてを記録に残すということである。これを手動で行うのはかなりの重労働だ。そしてエラーを避けることはほぼ不可能だ。しかしマシンならこれを簡単に、しかも確実に実行できる。
- **脆弱性スキャンを行う。**間違いは起こるものだ。システム管理者がアプリケーション問題を診断するとき、ありがちなパスワードを付けてテスト用アカウントを作成することはよくあることだ。こうしたアカウントがかならずしも毎回きれいに削除されているとは限らない。システム管理者は違うユーザーに特権をアサインしてしまうかもしれない。ずっと無効にしておくべきアカウントをヘルプデスクが有効化してしまうかもしれない。こうしたことから本来ないはずのアカウントやアサインしてはならないエンタイトルメントがないか、すべてのアプリケーションにおいて常時スキャンを行わなければならない。これを手動で行うことは相当な作業量だ。全システムを機械的に自動スキャンできない限り、現実的には不可能だ。この作業はリコンシリエーション（Reconciliation）と呼ばれ、しかるべき IDM システムであれば基本機能の一つとなっている。

上述したことはすべて、理論的には手動で行うことができる。しかし現実的には難しい。

実際は IDM システムがもたらす自動化と可視性がなければ、情報セキュリティは大きく損なわれてしまう。IDM システムを伴わない良好な情報セキュリティなどまずありえない。

アイデンティティ管理とアクセス管理のソリューションを構築する

たった一つのアイデンティティ管理とアクセス管理ソリューションがだれにでも適合するということはない。それぞれのデプロイには独自のニーズと特性がある。大銀行でのデプロイなら、おそらくガバナンス、ロール管理、セキュリティに重点が置かれるだろう。小規模な企業ではコスト効率だろう。クラウドプロバイダは拡張性、ユーザーエクスペリエンス、心地よさを重視する。要するに、あるサイズが全部にフィットすることなどないのだ。使用される主要コンポーネントはほとんどどの IAM ソリューションで同じである。ただ製品の選択と構成はかなり違う。ある一つの製品をダウンロードしてインストールすれば、抱えている問題をすべて解決してくれるなどと期待してはならない。そんなことはありえない。ここでカスタマイズがカギとなる。

我々はどんな IAM ソリューションであっても、その中心にあるのはアイデンティティ管理だと思っている。それが我々が midPoint プロジェクトを立ち上げた理由の一つだ。次章以降ではアイデンティティ管理および IDM コンポーネントとしての midPoint の使用を中心に説明する。理論はここまで。ここから先は実践に入る。

第 3 章： midPoint の概要

混乱は自然の法則であり、秩序は人の夢である。

– ヘンリー・アダムス

midPoint はオープンソースのアイデンティティ管理（identity management、IDM）およびアイデンティティガバナンスシステムであり、多くの高度な機能を提供する、豪華で洗練されたシステムである。当システムはオープンソースの IDM 開発を専門とする企業、Evolveum によって管理されている。中核となる開発者は全員 Evolveum の社員である。しかし他にもパートナーや開発者がおり、midPoint 開発に貢献してくれている。

midPoint は第二世代の IDM システムだ。midPoint 開発チームには、第一世代の IDM システムをデプロイした 2000 年代初頭からのベテランが数人いる。当時のそれは必ずしも楽しい経験ではなかった。そこで 2011 年、我々は初期の IDM システムにある間違いを修正すべく midPoint プロジェクトを開始した。他の IDM システムとは違う大きな点のひとつが、midPoint は「実用的であること」という一つの大目標を念頭に設計、実現されたものということである。我々は過去に第一世代の IDM システムを取り扱っており（そして苦戦しており）、あのような経験は 2 度としたくないと思っている。そのため midPoint の根幹には実用性が深く刻まれているのだ。もっと具体的にいうと、実用性とは次のことをいう。

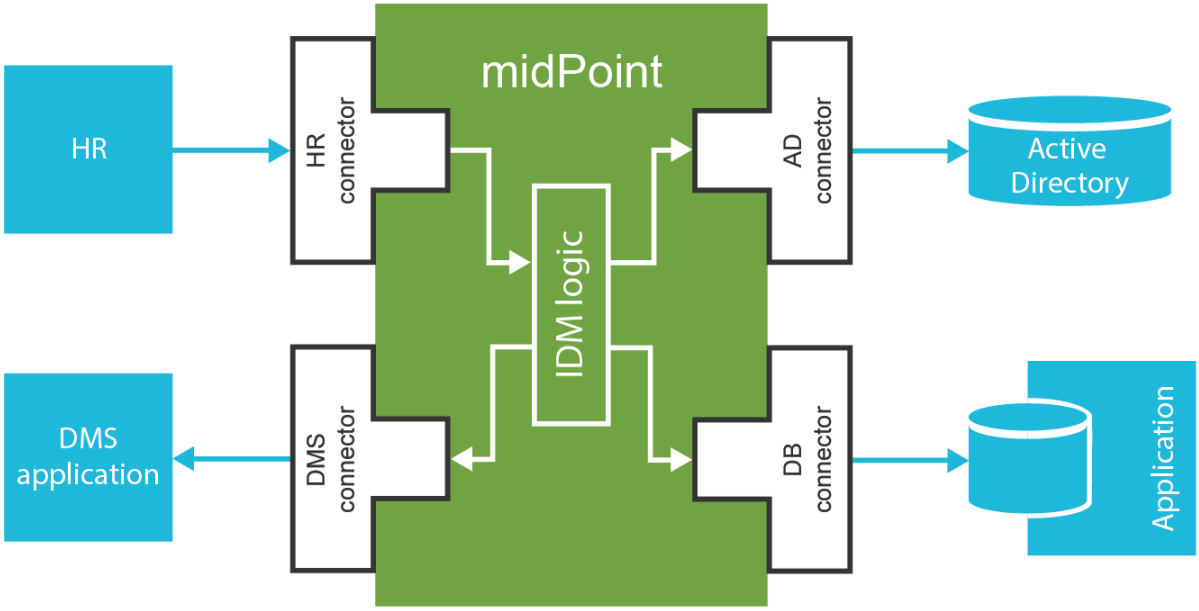
- シンプルであるもの、または頻繁に使用するものは設定が容易であること。パスワード変更の伝達、ユーザーの有効化および無効化、アカウントの同期—これらはできるだけ簡単であるべきだ。スイッチ一つ、あるいはわずかな構成プロパティを設定するだけですむほどシンプルであること。
- より複雑なもの、または使用頻度の少ないものは、若干大変でもよいだろう。たとえば、XML ファイルや JSON ファイルの編集、Groovy または Python スクリプトを数行作成、といったことである。
- 非常に複雑なもの、または非常に独特なものであっても可能でなければならない。ただこれらは簡単ではないだろう。より長大なスクリプトまたは Java クラスの実装が必要かもしれない。ソースコードのフォーキング（分岐）や修正が必要かもしれない。とにかく、ほぼ何でもできなくてはならないのだ。

つまり、通常の要件から外れていないシンプルなソリューションは簡単に実装できる、ということである。IAM プログラムはこのように始まるものが大半だ。この手法ではプロジェクトのかなり早い時点でメリットを得られる。要件がより複雑で独特になるにつれ、労力も大きくなる。それでも初めからすべてを実装するよりははるかに小さい。そしてメリットを正当化できないほどコストが高くなりすぎてしまった場合はその時点でプロジェクトを停止するという選択肢が常にある。midPoint はオープンソースシステムである。ライセンスコストがかからないため、初期コストも減るだろう。midPoint があれば小規模なプロジェクトでさえ実施可能である。

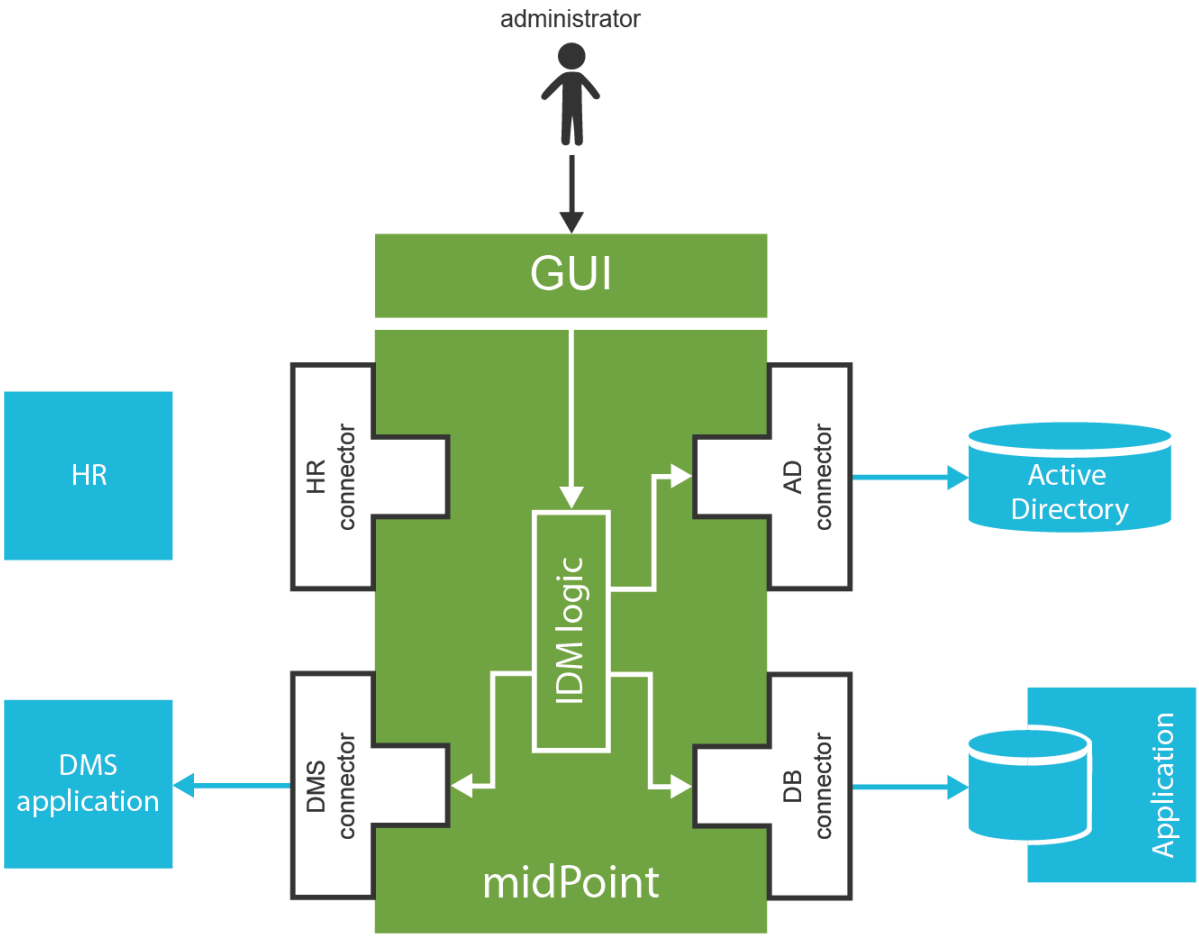
簡単に言うと、midPoint はパレートの法則に従っている。すなわち「2 割の労力で 8 割のメリットを導く」である。この手法をサポートする仕組みは数多くある。midPoint 設計をもとにしたもの、midPoint の開発手法に端を発するもの、さらには Evolveum のビジネスモデルによってサポートされているものもある。詳しくはあとで説明しよう。

midPoint の仕組み

midPointはアイデンティティ管理システムに期待されていること、つまりアイデンティティの管理を行う。そのごく基本的な機能は、様々なアプリケーション、データベース、ディレクトリサーバー、テキストファイル等に保持されているアイデンティティデータを同期させることだ。これらすべてのシステムを、我々は*リソース*と呼んでいる。midPoint はコネクタを使用してこのリソースにつながっており、あるリソースで発生した変更を別のリソースへ伝えることができる。例えば、人事システムに新たな従業員レコードが発生すると、midPoint がこれを検出し、処理を行い、新たなアクティブディレクトリおよび CRM アカウントが作成される。このプロセスを「*同期（synchronization）*」と呼ぶ。



midPointはアイデンティティ変更に使用できるリッチユーザーインタ—フェイスも有している。変更があると、midPoint は関連するすべてのリソースへその伝達も行う。例えば、セキュリティ担当者が midPoint ユーザーインターフェイスにて無効ボタンをクリックしてユーザーを無効にする。すると midPoint はそのユーザーに属するすべてのアカウントをただちに無効にする。



これは midPoint 操作の基本原則である。ただこれは極めて単純に説明したものである。midPoint を偉大にしている重要なことの多くは、対象リソースに変更が適用される前に起こる。これは一見すると簡単そうに見えるが、実はそうではない。処理された変更ごとに、midPoint は次のことを評価する必要がある。

- **ロール**：midPoint はユーザーがアクセス権をもつべきところを割り出す。通常これはユーザーがもつロールによって与えられる。ロール構造は実に豊富である。階層型ロール、パラメータロール、条件付きロールのほか、多くの高度な仕組みがある。
- **組織構造**：一般にユーザーは何らかの組織単位、プロジェクト、チーム、またはグループに属している。その一部はユーザーに追加で特権を付与している場合もある。
- **ステータス**：アカウントは作成、有効化、無効化、削除することができる。処理が必要となる状況はたくさんある。例えば、新入社員の業務開始 1 週間前に無効の状態でアカウントを作成し、勤務初日にそのアカウントを有効にし、勤務最終日に無効にし、離職して 3 カ月後にそれを削除したいこともあるだろう。
- **属性と識別子**： 単純な同期の場合、同期のとれたシステムでは属性と値は同じであると判断される。これは素晴らしい理論であるが、現実世界ではほぼ通用しない。属性名は翻訳が、そして値およびデータタイプは変換が必要である。これはシステ

ムごとに、さらには各システムのインスタンスごとに違う。値を正しく変換するには、スクリプト式での細かいアルゴリズムが必要である。

- **クレデンシャル管理**： 変更したパスワードはリソースへ伝達する必要がある。パスワードはすべてのシステムで同期させたいこともあれば、ごく一部のシステムに限定したいこともある。つまりパスワードポリシーを評価する必要がある。パスワードにはエンコードおよびハッシュが必要なこともあるだろう。
- **整合性**：ターゲットアプリケーションのアカウントは、midPoint による更新後にも変更されている場合がある。現行の変更はもはや適用できないかもしれない、ネイティブ変更と矛盾するかもしれない、変更がすでに一部適用されているかもしれない、アカウントが本来もっていないはずの属性値をもっているかもしれない、またはアカウントがまったく存在していないかもしれない。midPoint はそのような状況を検出しこれに対応しなくてはならない。例えば、削除したアカウントを再作成してから変更を適用するといった対応である。
- **ワークフロー**：midPoint は変更を適用する前に承認が必要か判断する。必要と判断した場合は承認プロセスに沿って要求を出す。
- **通知**：midPoint は新アカウントにアクセスできることをユーザーに通知する。何か異常があれば管理者に通知する。
- **監査**：midPoint は監査証跡にすべての変更を記録する。これはあとでセキュリティ担当者または分析専用エンジンが使用できる。

midPoint が検出する一つ一つの変更に対し、上記の内容がすべて処理、評価、実行される。これらのステップの中には非常に複雑なものもある。たしかに midPoint には複雑なアルゴリズムが数多く実装され、使用できる状態にある。複雑なロール構造、組織構造、時間的制約、パスワードポリシーなどを評価するアルゴリズムがある。唯一必要なことは、これらを正しく構成することである。

しかしmidPointが行うのはそれだけではない。アイデンティティを管理するだけでなく、とにかくアイデンティティに関わるオブジェクトはすべて管理できるのだ。ロール、ロールカタログ、組織構造、グループ、プロジェクト、チーム、サービス、デバイスのほか、ほぼすべてのオブジェクトを管理できる。

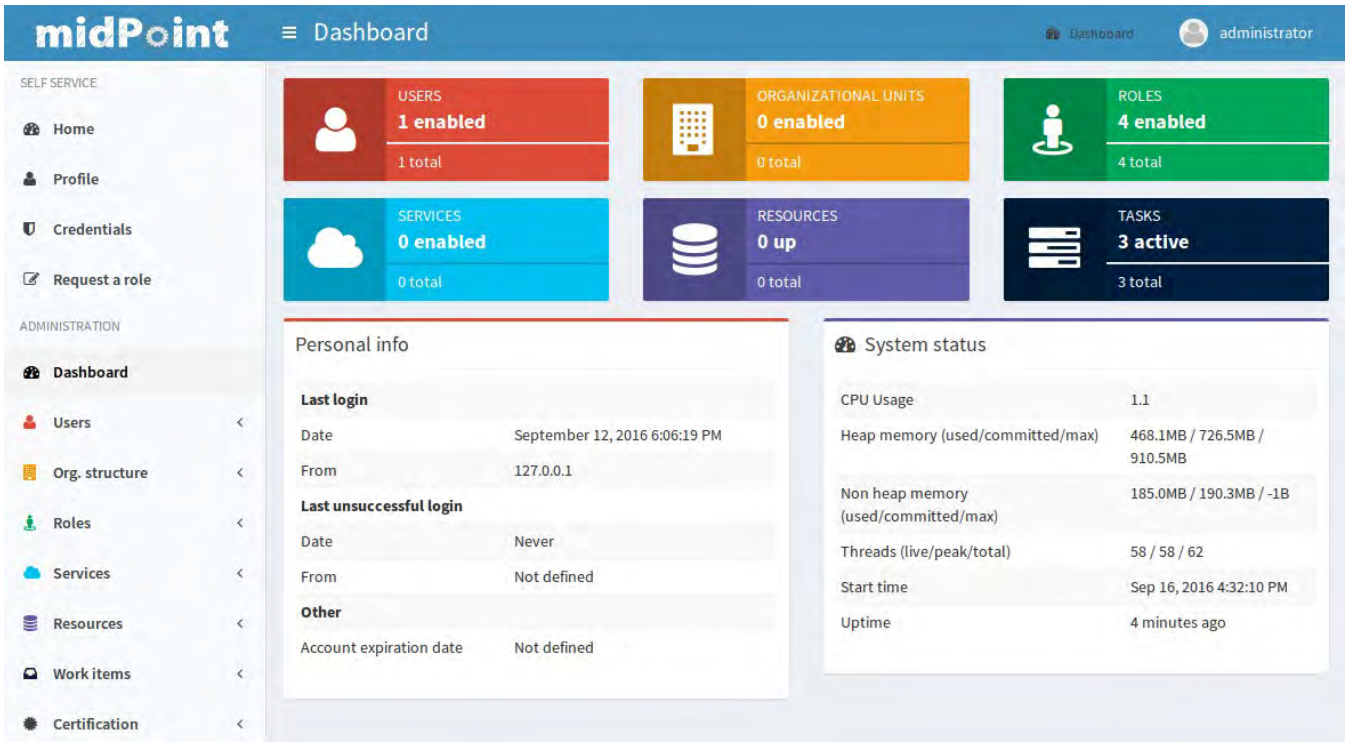
midPointは*アイデンティティガバナンス*システムでもある。アイデンティティ管理機能は、組織全体にポリシーが適用されるよう整合性を確保する。そしてガバナンス機能はポリシーの保持と進化を支える。midPoint ではアクセスの再認定プロセスが行われる。これはユーザーの受け取った特権がまだ必要かを確認するよう管理者に求めるプロセスであり、定期的に発生する。midPoint には、ロールを階層およびカテゴリにどう分類するかの仕組みも含まれている。これはロールエンジニアリング中およびロール定義の保守中の秩序を保つために必要である。また、ロールの選択的実施に関する仕組みも有しており、移行中や新システムと midPoint の接続時には非常に有効である。最近のバージョンには、ポリシー（ロール）のライフサイクルや一般ポリシールール等に対するサポートが導入された。そして今後のバージョンにはその方向でより多くのワークが計画されている。我々は、単に

ポリシーを適用するだけでは不十分なのはよく分かっている。ポリシーは生き物であり、進化する必要があるのだ。

ケーススタディ

本書は実用的なアイデンティティ管理について述べるものである。そこでケーススタディをもとに midPoint の機能を説明し、より実践に近づこうと思う。これからお話するのは架空の企業「ExAmPLE, Inc.」のケーススタディである。ExAmPLE とは「Exemplary Amplified Placeholder Enterprise」のことである。ExAmPLE 社は中規模の金融会社である。当社の業務は情報技術にかなり依存しているため、旧式のアプリケーションからクラウドサービスまで、実にさまざまなアプリケーションと情報システムがある。従業員数は 2〜3 千人で今後の成長も見込まれることから、経営陣は IAM プログラムを開始することに決定した。そのプログラムの第一ステップが、アイデンティティ管理システムとして midPoint をデプロイすることである。

エリックは ExAmPLE 社の IT エンジニアであり、midPoint のインストールおよび構成の責任者である。彼は midPoint 用に新たな Linux 仮想マシンをスピンアップする。midPoint の配布パッケージをダウンロードし、インストール手順に従って操作する。数分後には midPoint のインスタンスが起動し、ユーザーインターフェイスにログインする。



新規でインストールされたばかりの midPoint インスタンスは、わずかに必須オブジェクトが含まれているだけでほぼ空っぽである。しかしエリックは優秀なエンジニアだ。すでに本書を最後まで読み通した彼は何をすべきか正確に把握している。

まず最初は midPoint に従業員データを入力することだ。ExAmPLE 社の従業員データの第一ソースは人事システムである。ただ人事システムは相当大きなソフトウェアであり、これを midPoint に直接接続するのは簡単ではない。さいわい、コンマ区切り（CSV）形式の従業員データをテキストエクスポートファイルで受け取るなら至極簡単である。そこでエリックはこのファイルを使用して従業員データを midPoint に取り込もうと考える。

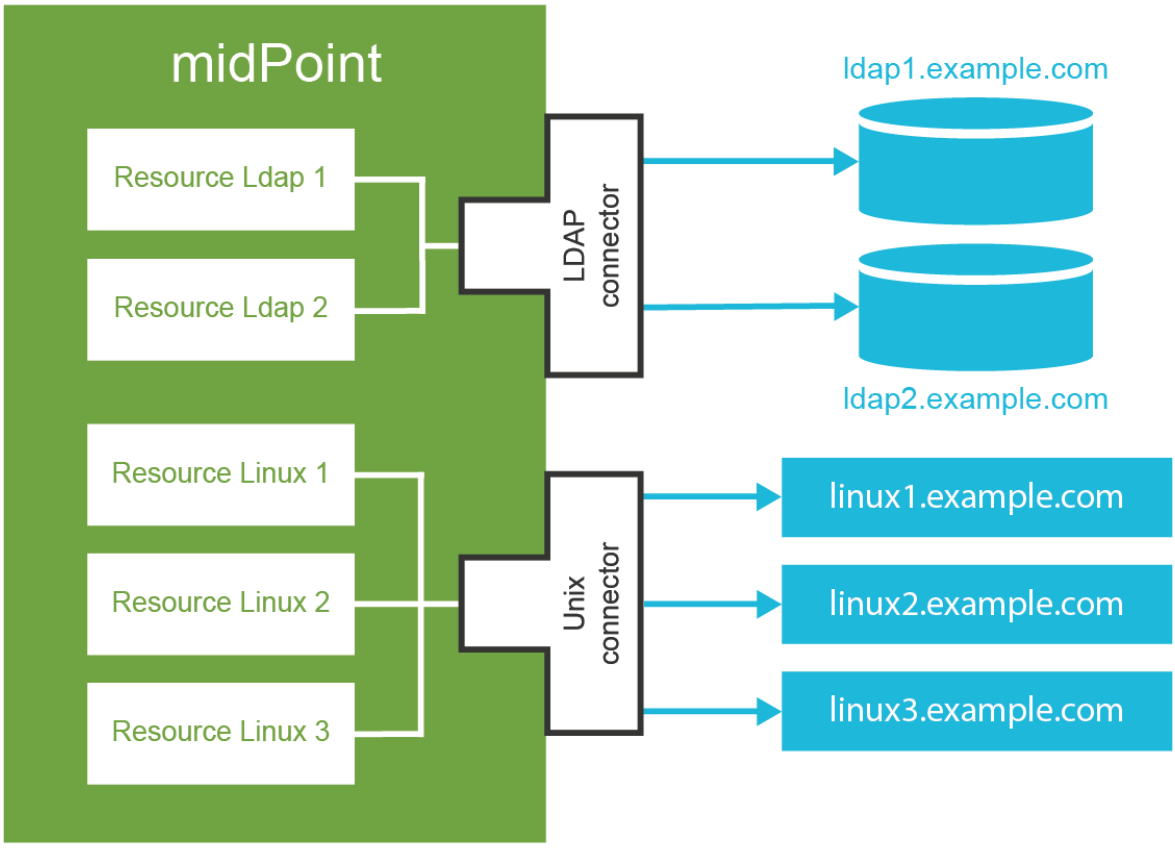
コネクタおよびリソース

midPointは*コネクタ*を使用してすべてのソースシステムおよびターゲットシステムと通信する。コネクタは midPoint に接続されている比較的小さな Java コンポーネントである。通常は接続されたシステムの種類ごとに一つのコネクタがある。そのため、LDAP サーバー、アクティブディレクトリ、データベース、UNIX OS 等に対しそれぞれコネクタがある。コネクタに求められる責務は、プロトコルを翻訳することだ。例えば、LDAP コネクタは midPoint の検索コマンドを LDAP の検索要求に翻訳する。また、UNIX コネクタは SSH セッションを作成し、midPoint の作成コマンドを Linux の useradd バイナリの呼び出しに翻訳する、などである。一方のコネクタは独自の通信プロトコルを使用して話しをする。しかし他方のコネクタは、その情報をすべて midPoint が理解する共通フォーマットへと翻訳する。

コネクタについていえば、ソースシステムとターゲットシステムとでは何の違いもない。ソースシステムにもターゲットシステムにも同じコネクタが使用されている。違いがあるのは midPoint の構成にのみだ。

コネクタは Java バイナリ（JAR ファイル）として配布される。これらを midPoint にデプロイするには、正しいディレクトリに置いて midPoint を再起動するだけでよい。midPoint は起動中に自動でコネクタを検出し検査するだろう。midPoint の配布パッケージには頻繁に使用されるコネクタがわずかながら同梱されている。これらはデプロイする必要はない。自動で利用可能になるからだ。

特殊なタイプのコネクタでは、そのコネクタがサポートするプロトコルで通信するシステムすべてに機能するものがある。たとえば LDAP コネクタは、LDAP に準拠するすべてのサーバーに機能する。コネクタはごく普通のコードにすぎない。あるサーバーへ接続を確立するのに必要なホスト名も、ポートもパスワードも知らない。個々のサーバーについて接続パラメータを規定している構成は、*リソース*と呼ばれる特別な構成オブジェクトに格納されている。midPoint において*リソース*という用語は、midPoint インスタンスに接続されているシステムのことを指す事が多い。

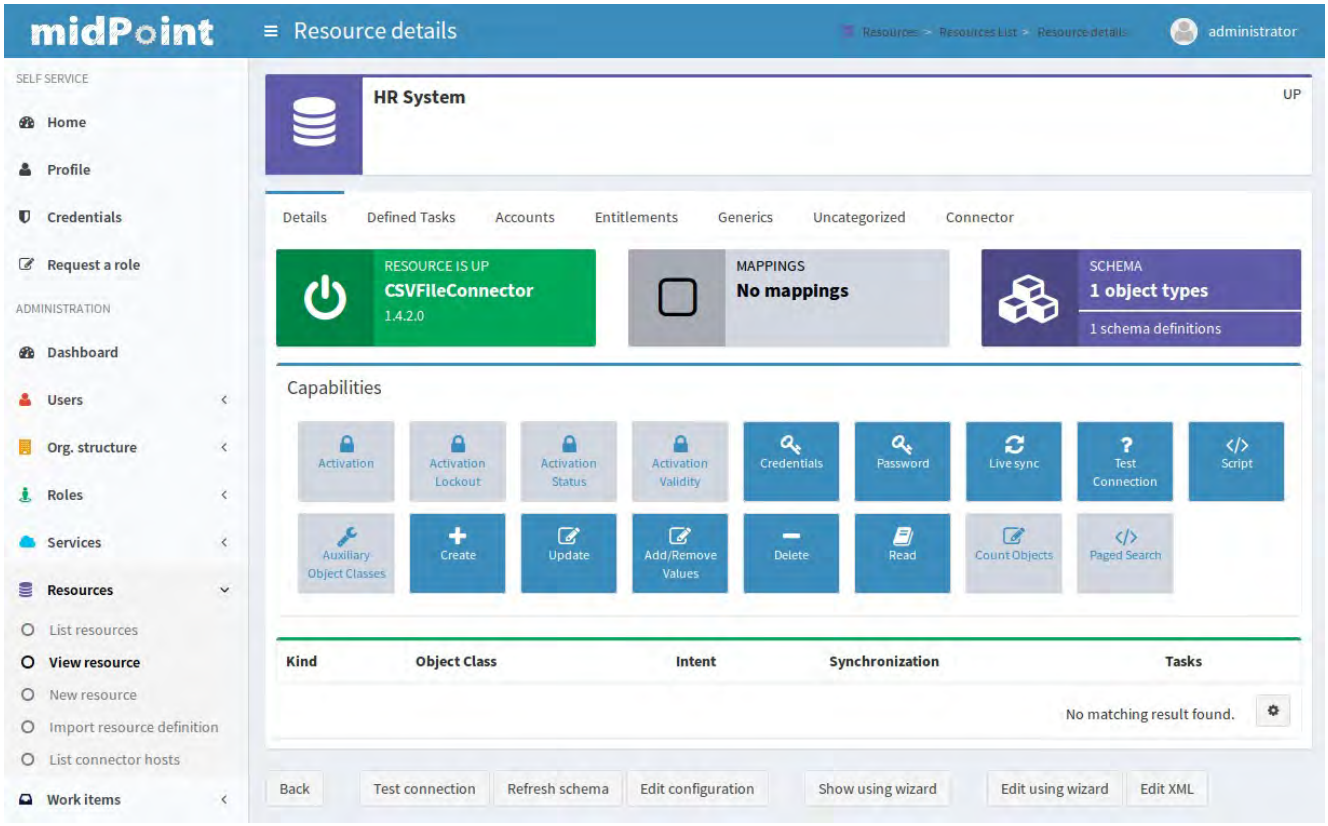


そこでエンジニアであるエリックが、ExAmPLE 社の従業員データを midPoint に取り込むためにしなくてはならないことは、新リソースを定義することである。このリソースとは、人事システムからエクスポートされる CSV ファイルのことである。midPoint の配布パッケージにはすでに CSV ファイルコネクタが含まれているため、わざわざそれをデプロイする必要はない。今彼がすべきは、新リソースの定義を作成することだ。方法は（少なくとも）２つある。その一つが、midPoint のユーザーインターフェイスにある構成ウィザードだ。ウィザードを使用すれば新リソースを一から構成できる。しかし本書の後半を読めばお分かりになるかと思うが、リソースの定義は柔軟性に富んでいて構成オプションも多い。そのため構成ウィザードも非常にリッチであり、新ユーザーであればかなり混乱してしまうかも知れない。そこでエリックにはもう一つの方法のほうがよいだろう。それはサンプルを活用しながらはじめることだ。midPoint の配布パッケージには色々なリソース定義の例がある。オンラインであればもっと多くの例が利用できる。エリックは、CSV リソースの完全な例が含まれている XML ファイルをすぐに見つけ出す。そのファイルを編集して彼の CSV ファイルへのファイルシステムのパスに変更し、自分のファイルのフォーマットに合うように項目名を調整する。最低限のリソース構成は、リソース名、コネクタ、コネクタ構成を指定するだけである。エリックが作成した XML ファイルはほぼこれに近いようだ（分かりやすくするためシンプルにしている）。

```
<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
        <name>HR System</name>
        <connectorRef type="ConnectorType"> ...</connectorRef>
```

```
    <connectorConfiguration>
            <configurationProperties>
                <filePath>/var/opt/midpoint-example/resources/hr.csv</filePath>
                <uniqueAttribute>id</uniqueAttribute>
            </configurationProperties>
      </connectorConfiguration>
</resource>
```

その後彼は midPoint ユーザーインターフェイスの *Configuration（構成）*セクションを開き、XML ファイルを midPoint へとインポートする。インポート操作によって midPoint に新リソース定義が作成される。エリックは今、midPoint ユーザーインターフェイスの *Resource（リソース）*セクションに遷移している。そこには新 CSV リソースがある。リソース名をクリックすると、リソースの詳細画面が表示される。

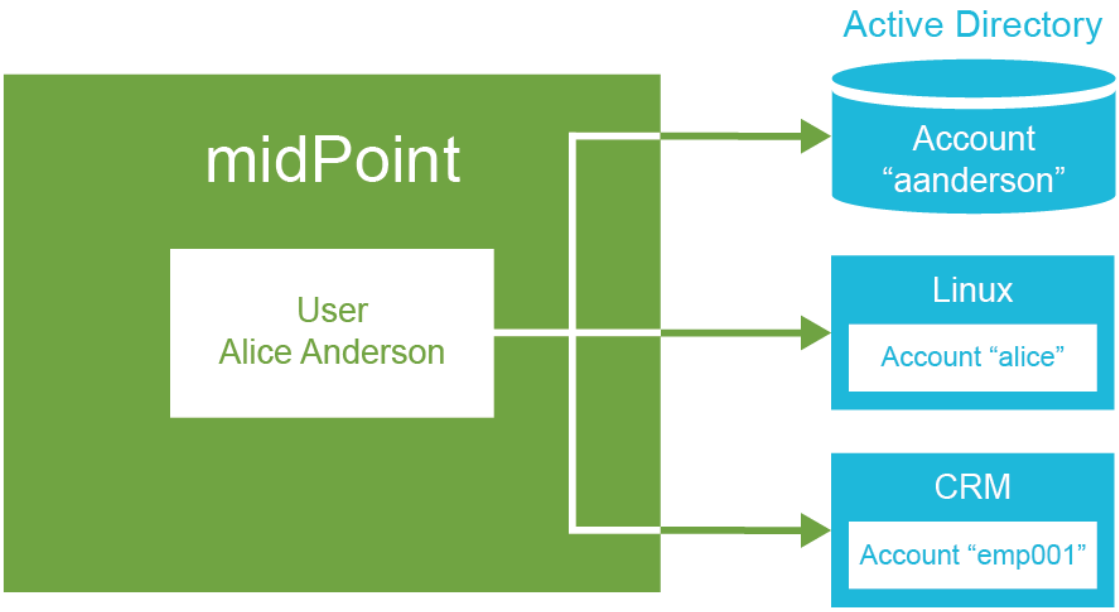


ボタンをクリックしてリソースへの接続をテストすることができる。これはローカルの CSV ファイルのため、このケースでは実際には接続はない。しかしテストによってファイルシステムパスが正しいか、ファイルが存在するか、ファイルを開けるか、チェックする。テストでは *リソーススキーマ*もロードする。midPoint は CSV ファイルヘッダを読み込み、CSV ファイル内のデータ構造が何かを把握する。ここでリソースは使用できる状態となる。

しかしエリックがリソースを使用してできることはまだ多くない。ケーススタディの話を進める前に、ここで midPoint の基本的な概念を少し説明する必要がある。

ユーザーおよびアカウント

*ユーザー*という概念は、IDM 分野全体でおそらく最も重要な概念だろう。*ユーザー*という用語は、従業員、サポートエンジニア、派遣社員、顧客といった物理的な人間を表わしている。一方、*アカウント*とはユーザーがアプリケーションにアクセスできるデータ構造のことである。これは OS、LDAP エントリ、ユーザーの識別子やパスワード等を格納するデータベーステーブルの列にあるアカウントかもしれない。普通は一人のユーザーが多くのアカウントをもっている。そして一般に各リソースに対しアカウントは一つである。



ユーザーを示すデータは midPoint に直接格納される。一方、アカウントを表すデータは「リソース側」に格納される。つまりアカウントは、接続されたアプリケーション、データベース、ディレクトリ、OS に格納されるということだ。midPoint には格納されない。通常の環境では、midPoint はアカウントの識別子とアカウントに関するわずかなメタデータのみを持つ。その他の属性はすべて必要に応じて新たに取得する。アカウントデータの取り込みにはコネクタを使用する。

本書ではユーザーとアカウントの用語を厳密に区別している。また midPoint のユーザーインターフェイスや文書においてもこの２つは厳密に区別されているとお気づきだろう。この専門用語に慣れるととても便利である。

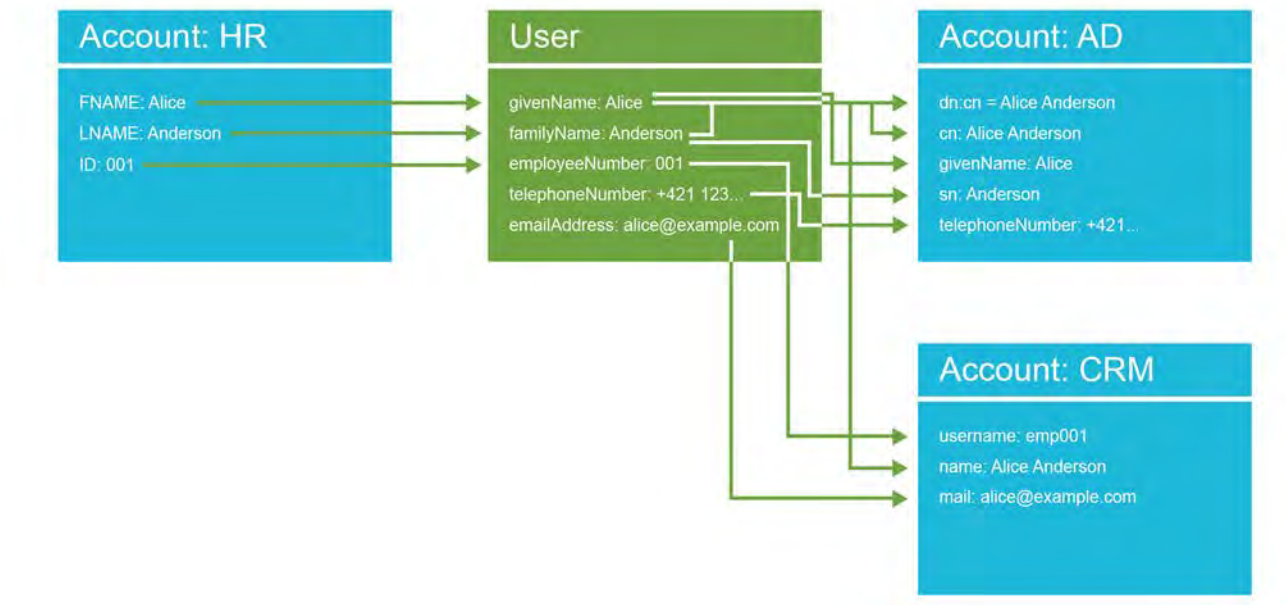
アカウントはアカウントを有するユーザーにリンク付けされている。よって midPoint はどのアカウントが誰に属しているかを知っている。midPoint はあるユーザーのアカウントをすべてリスト化し、データを同期させることができる。また、すべてのアカウントを一度に無効化することなどもできる。このリンクは通常自動で構築され midPoint で保守される。

midPoint にはユーザー用のビルトインデータモデル（スキーマ）が同梱されている。このモデルには、フルネーム、メールアドレス、電話番号などユーザー説明にしばしば使用さ

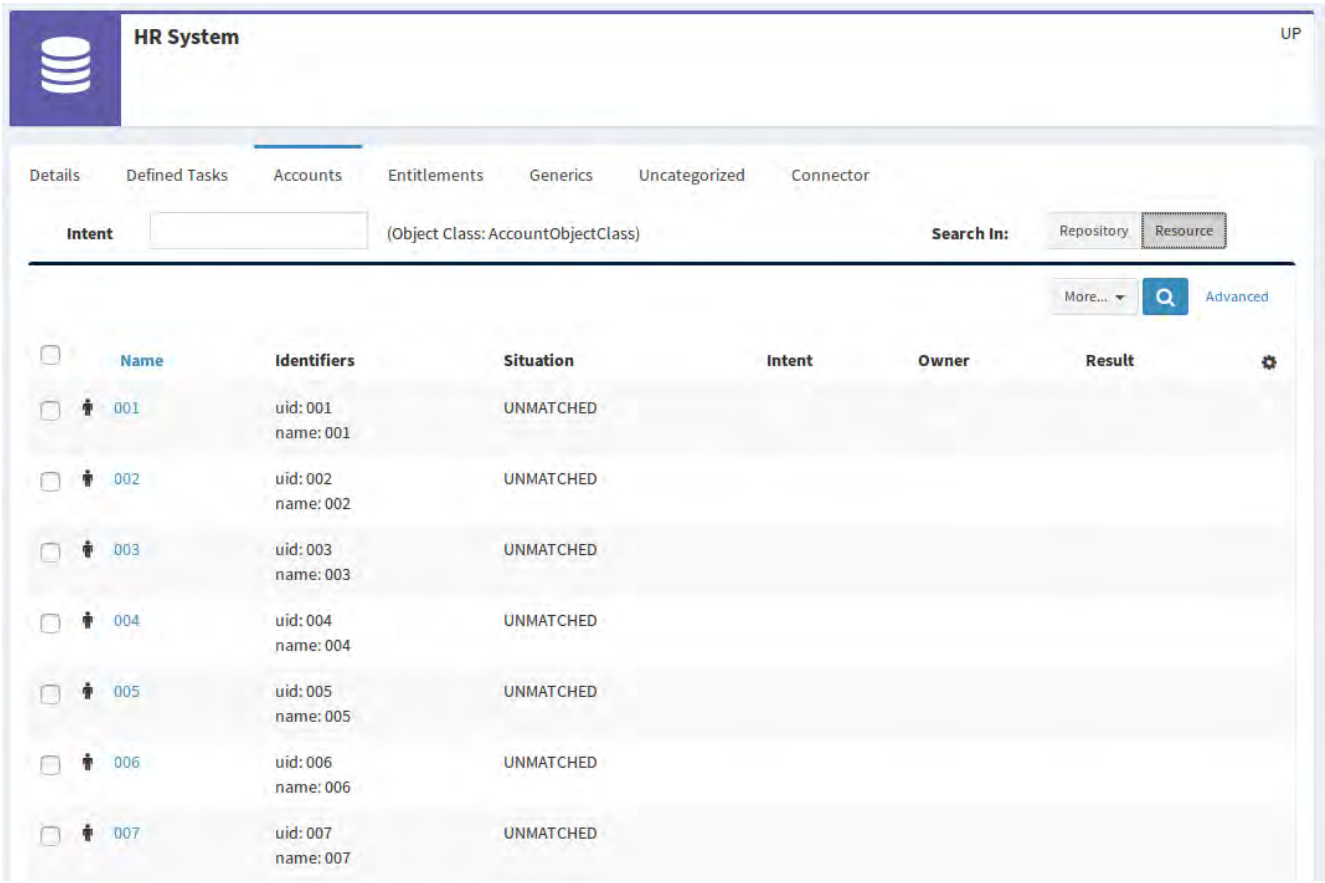
れるプロパティが含まれている。多くのデプロイにとってよい起点となる、妥当なプロパティ構成がある。もちろん、midPoint の大半のオブジェクトと同様、ユーザースキーマは必要に応じてカスタムプロパティで拡張できる。

しかしアカウントには統合されたデータモデルがない。一つにはなりえないのだ。各リソースがそれぞれ違うアカウント属性を持っているかもしれない。名前もタイプも違うかもしれない。値も違う意味を持っているかも知れない。midPoint はこれに対応するよう設計されている。リソースアカウントのスキーマは、midPoint がはじめてそのリソースに接続するときに動的に検出される。すると midPoint は実行時にそのスキーマを解釈し自動でそれに適合する。例えば、midPoint がアカウントに関する情報を表示するとき、ユーザーインターフェイスの項目は検出されたスキーマから動的に生成される。midPoint 自体がこれをすべて行う。追加構成も必要なければ、もちろんコーディングも不要である。

リソース内のアカウントスキーマは大幅に異なるかもしれない。にもかかわらず midPoint は、考えられるあらゆるリソースからのすべてのアカウントを同期させることができる必要がある。この場合、ユーザースキーマは統合データモデルとして機能する。各アカウントのスキーマがユーザースキーマにマップされる。ユーザースキーマは、あなたの組織のユーザーを完全に表す統合データモデルとして設計されている。



さて、再び ExAmPLE 社の話に戻ろう。エリックは人事リソースを構成した。よって彼はユーザーが人事システム内にもっている「アカウント」を見ることができる。エリックは midPoint GUI のリソース詳細ページを開き、[*Accounts（アカウント）*]タブをクリック、さらに[*Resource（リソース）*]ボタンをクリックする（これについてはあとで説明する）。するとアカウントのリストが表示される。



このリストに表示されるのは従業員番号のみである。これは人事システムでは従業員番号が第一識別子として設定されているためである。リンクをクリックすればより詳細が表示される。実際にはこれらは本当のアカウントではない。これらは人事データベースからエクスポートされた CSV ファイル内のラインである。しかし*アイデンティティ*の一部が記述されているため、midPoint はこれらをアカウントと解釈する。midPoint にとって「アカウント」とは、ユーザーを表すリソース側のデータ構造を説明するものとして使用される一般的な用語である。

初回インポート

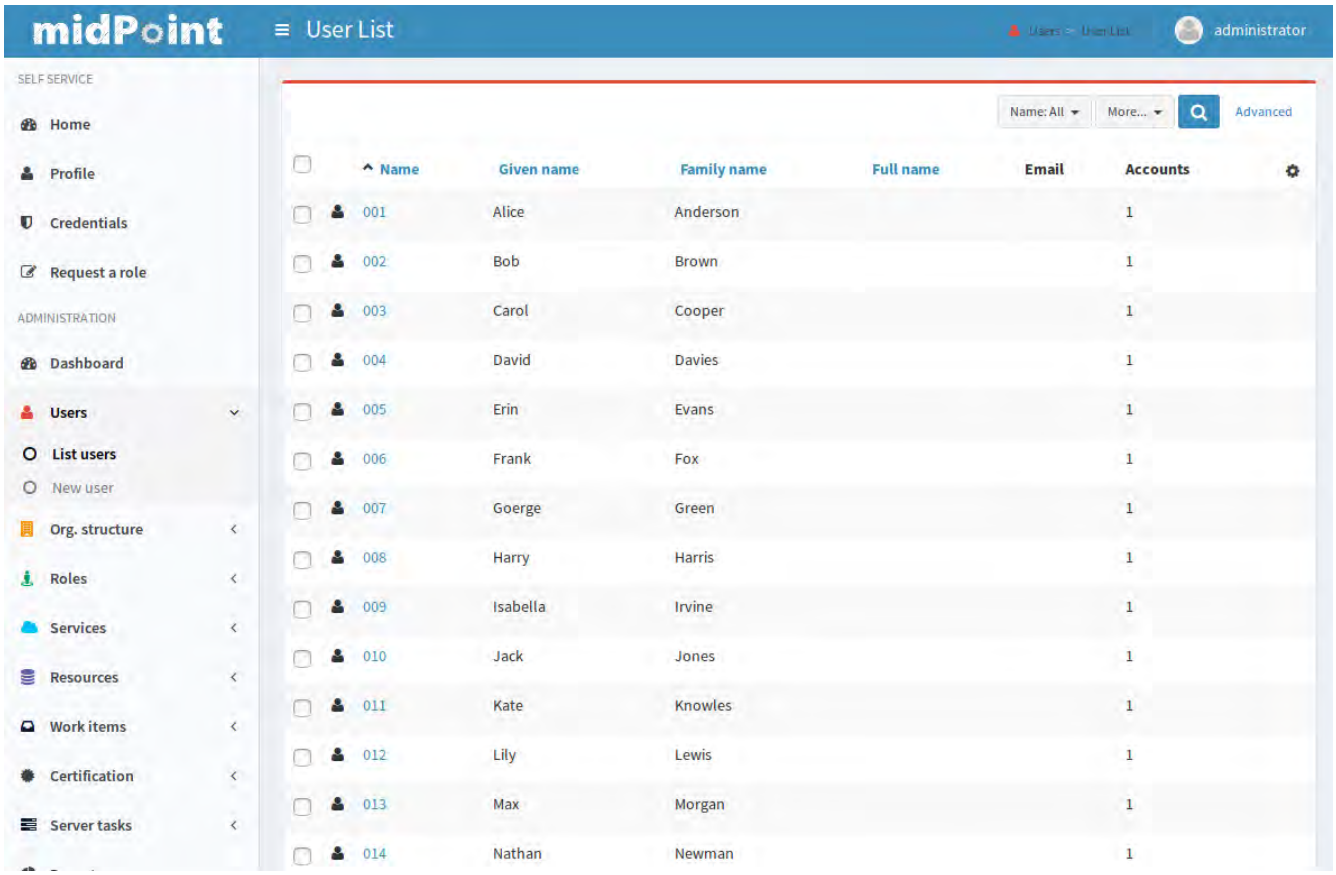
どの IDM システムにとっても*ユーザー*は中心となる概念であり、midPoint も例外ではない。midPoint が正しく機能するには信頼できるユーザー情報が必要である。人事システムはユーザー情報の良好なソースである。エリックは人事システムから midPoint にユーザー情報を取り込む必要がある。すでに人事システムからエクスポートされた CSV ファイルに接続するリソースは設定できている。しかしそのリソースはデフォルトでは何もしないことになっている。そこで CSV ファイルから midPoint にデータを取り込むよう設定しなくてはならない。ここで必要なのが*マッピング*の設定である。マッピングとは、ユーザーとリンク付けされたアカウントとの属性値を同期させるための仕組みである。エリックはデータをインポートするため*インバウンド*マッピングが必要となる。インバウンドマッピングとはリソースから midPoint への方向で値を同期させることである。エリックは GUI の構成ウィザードでリソース定義を開き、そこにマッピングを追加できる。あるいは、簡単に構成サンプルを閲覧して XML 形式でマッピングを追加してもよい。インバウンドマッ

ピングはこのようになる。

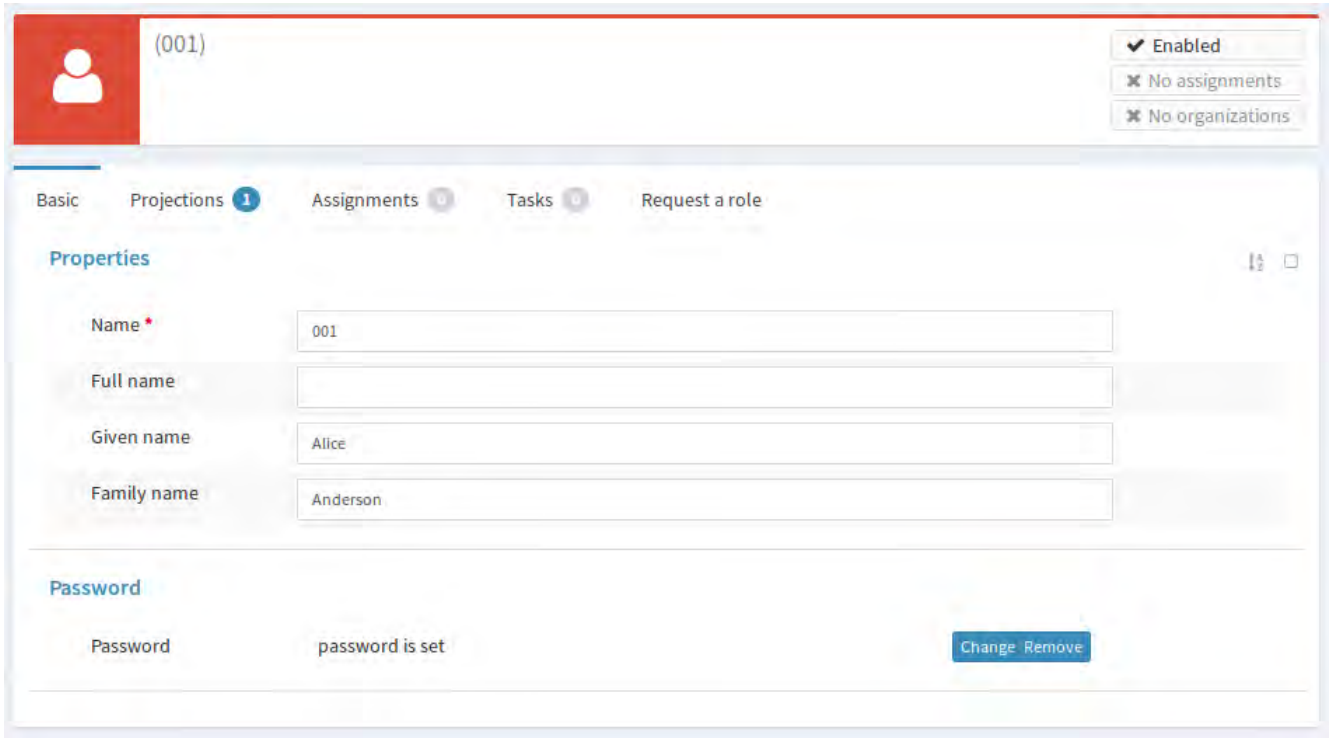
```
<attribute>
        <ref>ri:firstname</ref>
        <inbound>
                <target>
                        <path>$user/givenName</path>
                </target>
        </inbound>
</attribute>
```

これはアカウント（人事）属性の firstname をユーザー（midPoint）プロパティの givenName にマップするマッピングであり、midPoint に対し、マップされた人事属性に変更があるたびにユーザーの family name の値を更新するよう言っている。エリックは人事エクスポートファイルのすべての属性に対し同様のマッピングを追加する。また、リソース定義に*同期*セクションを追加する必要がある。同期セクションは midPoint に新アカウントごとに新ユーザーを作成するよう指示する。この、人事アカウントごとにユーザーを作成することこそ、我々がまさに求めているものである。そしてエリックは変更済みの XML ファイルを midPoint に再インポートする。

これで midPoint は属性を同期させる準備が整う。しかしまだ、人事システムからすべてのデータを引き出すというタスクが必要だ。エリックは人事アカウントのリストが表示されているページへと遷移する。ページの最下部には大きな［*Import（インポート）*］ボタンがあり、これを使ってインポートタスクの管理ができる。エリックはこのボタンをクリックし新インポートタスクを作成する。タスクが開始され数秒間実行される。タスクが終わると、midPoint にユーザーが表示されるようになる。

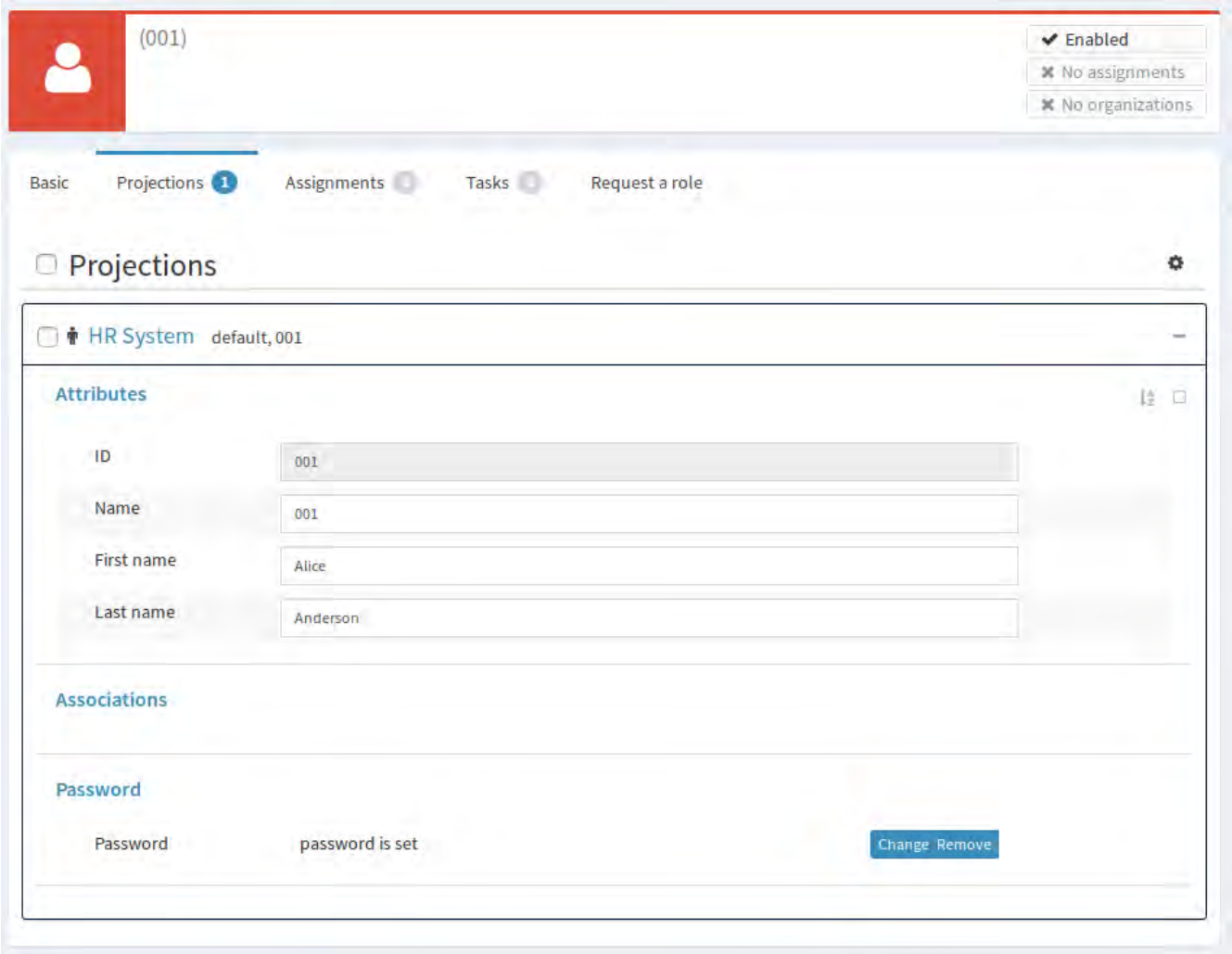


ユーザー名をクリックすれば当該ユーザーの詳細を見ることができる。



このページにはそのユーザーに関して midPoint が把握しているすべての情報が表示される。情報は複数のタブへと分けられている。これらのすべてはのちほど説明しよう。今は

最初の 2 つのタブのみに注目する。［Basic（基本情報）］タブには midPoint が把握しているユーザープロパティが表示される。これらのプロパティは midPoint レポジトリに格納される。midPoint はそのまま使用できるまさにリッチデータモデルを持つが、GUI は実際に使用されるこれらのプロパティを表示するのみである。Name、Given name、Family name は人事リソースからインポートされた情報で、まさにこのページが表示する内容である。では次に 2 つ目のタブをみてみよう。



［Projections（プロジェクション）］タブにはユーザーのアカウントが表示される。現在のところアカウントは 1 つのみで、それはデータのインポートに使用された人事アカウントである。ここに表示されるデータは、アカウントが表示された瞬間にリソースから取得した最新データである。これがユーザーデータとアカウントデータの違いである。ユーザーデータは midPoint レポジトリに保持されるが、一方でアカウントデータは必要に応じて取得される。

ユーザーとアカウントはリンク付けされている。midPoint は当ユーザーがこの特定の HR アカウントから作り出されたことを覚えている。人事アカウントが変更されると、その変更内容がユーザーデータにも同期され適用される。マッピングとはインポートのためだけ

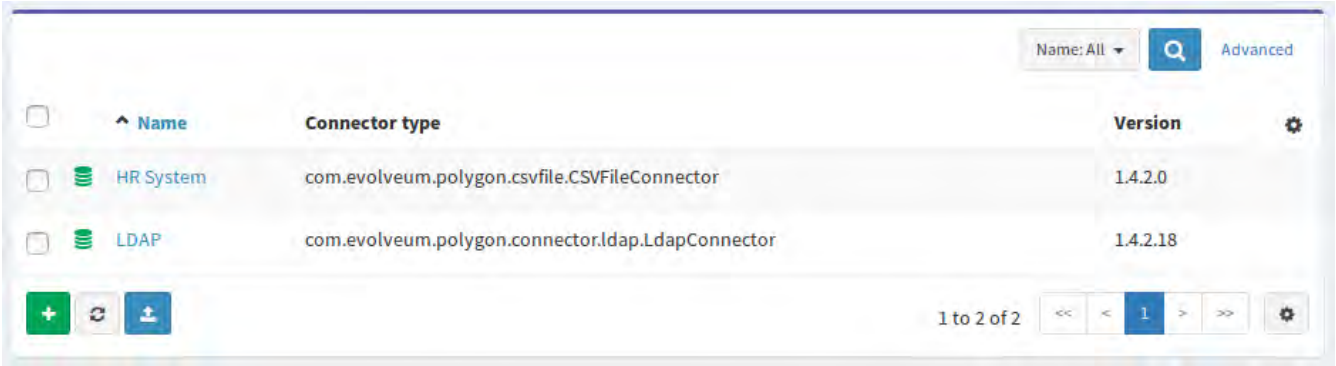
ではないのだ。持続的に機能し、アカウントデータとユーザーデータを常に同期させている。

アサインおよびプロジェクション

アカウントの概念は現実性にほかならない。そこに、ここに、そして今あるデータを示すものだ。そこに何が*ある*かを示している。しかしアイデンティティ管理の主要部分を占めるものはポリシーである。定義によれば、ポリシーとはそこに何が*あるべきか*を規定したものである。ポリシーは正しいものを規定する。しかし誰もがよく知ってのとおり、「*現実にある*こと」と「そうで*あるべき*こと」は必ずしも完全に一致するとは限らない。我々は理想主義者ではない。はじめから、現実とポリシーとでは違いがあるかもしれないことを認識しながら midPoint を設計してきた。そしてその差異を管理し、長期的にはこれを完全に解消することが midPoint の重要な役割である。

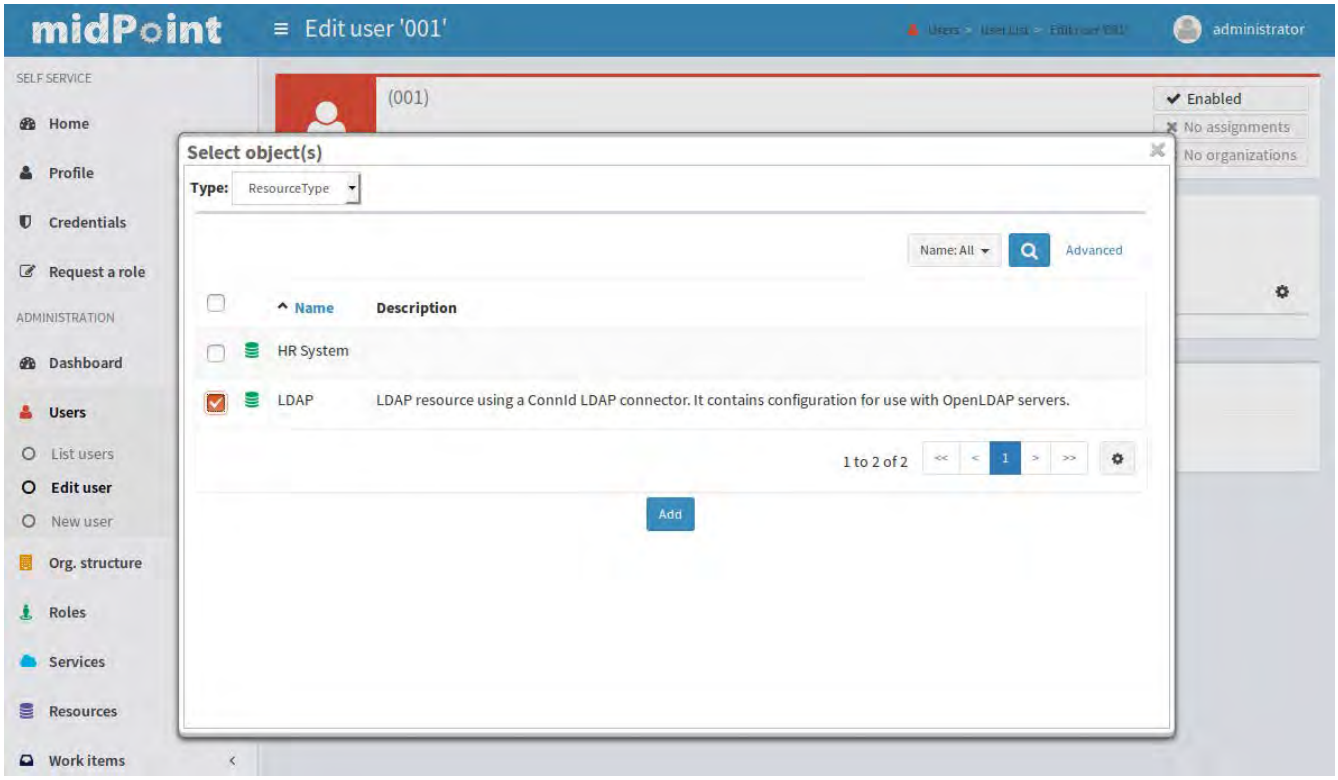
このような考え方は midPoint のユーザーインターフェイスによく反映されている。ユーザーの詳細情報ページには［*Projections（プロジェクション）*］タブがある。このタブは現実を表示する。ユーザーがまさに今持っているアカウントを表示する。アカウントの本当の状態を表示する。このタブは現実を表わしている。そして次に[*Assignments（アサイン）*]タブがある。こちらのタブはポリシーを表示する。そのユーザーにどのアカウント、ロール、組織、またはサービスがアサインされているかを表示する。このタブはユーザーが*持つべき*ものを表わしている。

アサインがどのように機能するかを例示するには、新たなリソースが必要である。そこでエリックには新リソースを midPoint に接続してもらおう。ここではその新リソースとは、新しくクリーンで空っぽの LDAP サーバーである。そこでエリックは再び適切な例を検索し、構成を変更してから midPoint にそれをインポートする。すぐに新たな LDAP リソースが現れる。エリックはすべてのユーザーを LDAP サーバーに同期させたいと思っている。そこで再びマッピングを定義しなければならない。しかし今回は midPoint からのデータを（LDAP）リソースへ伝達したいため、アウトバウンドマッピングとなる。マッピング構成についてはあとで詳しく説明する。ここではとにかく結果だけお見せしよう。今やリソースは 2 つになった。



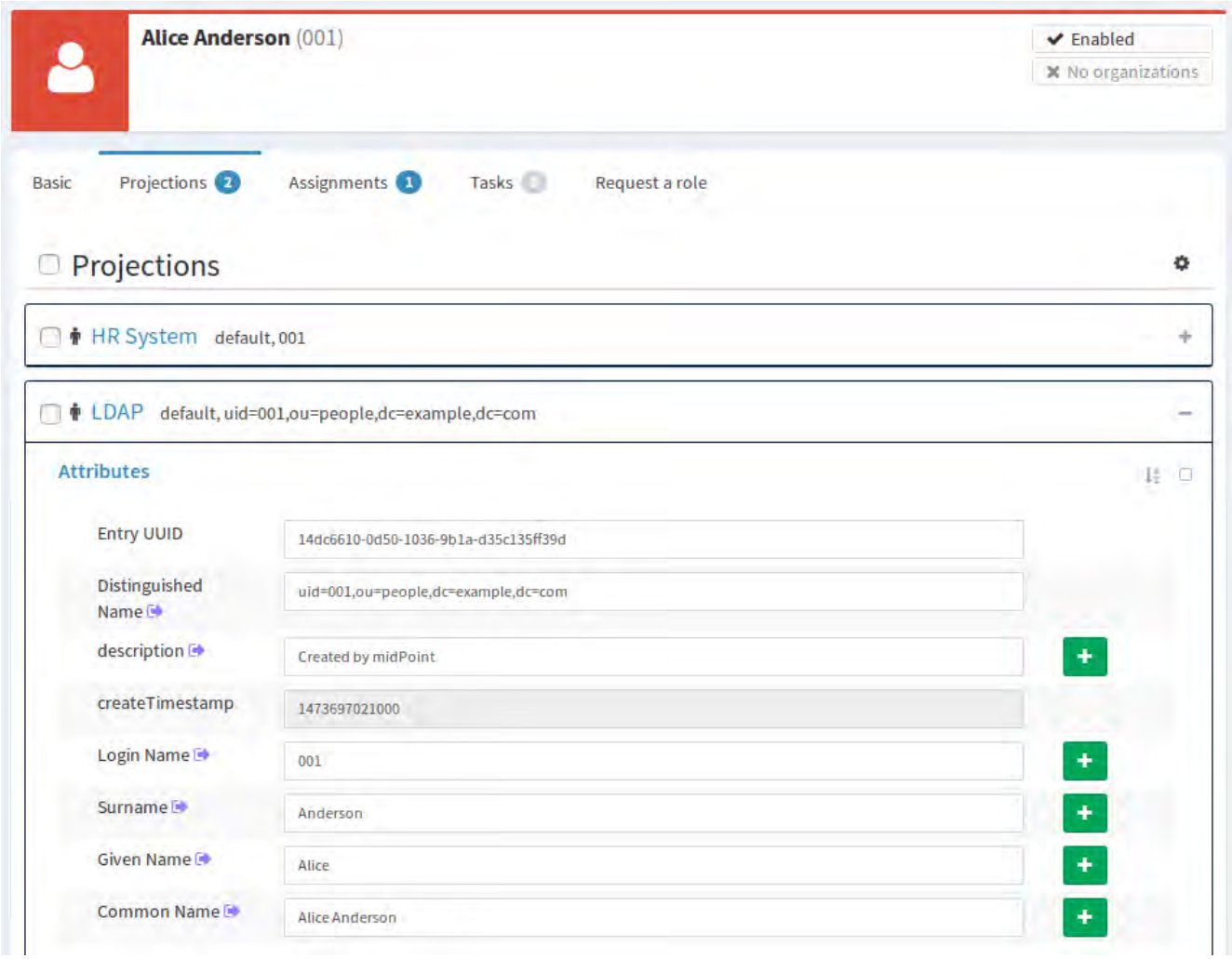
しかし LDAP リソースにはどうやってアカウントを作成するか？正しい方法としては、ユーザーがそのリソースにアカウントを*持つべき*であることを midPoint に知らせることであ

る。midPoint 用語で言うなら、ユーザーにリソースを*アサインする*ということだ。ここでエリックに必要な作業は、ユーザー詳細ページに遷移し、［*Assingments（アサイン）*］タブをクリック、歯車ボタンで LDAP リソースのアサインを追加したら［*Save（保存）*］をクリックするだけである。



［Save］ボタンをクリックすると色々と複雑なことが起こる。簡単に言ってしまえば、ユーザーが持つべきものと持っているものを midPoint が再計算する。midPoint は、ユーザーは今 LDAP をもっているはず（なぜなら LDAP の新たなアサインがあるから）ということを検出する。しかしそのようなアカウントは存在しない。そこで midPoint がそのアカウントを作成する。

エリックが再びユーザー詳細ページを開き［Projections（プロジェクション）］タブへ遷移すると、アカウントが 2 つ表示されているのが分かる。



一つは最初にユーザーを作成時に使用された人事アカウント、そしてもう一つが新規のアサインに対応して作成された LDAP アカウントである。

情報： 注意深い読者なら、これら 2 つのアカウントが持つ属性は大幅に違うことにお気づきだろう。その通りである。アカウントごとに*スキーマ*は違う。midPoint はリソースへの初回接続時にそのスキーマを自動で見つける。するとそのスキーマを動的に解釈して、GUI に属性を表示、入力内容を検証、マッピングにエラーがないかチェック、等を行う。コードを書く必要は一切ない。midPoint は自身ですべてを行うのだ。midPoint は完全にスキーマの概念にもとづいており、それを最大限活用している。

現実があり、ポリシーがある。アカウントがあり、アサインがある。理想を言えばこれら 2 つは一致しているべきだろう。そして midPoint は懸命にそれらを一致させようとするだろう。しかし例外もあるかもしれない。注意深い読者はここで再びお気づきだろうが、人事アカウントはあっても、そのアカウントについてのアサインはない。そして midPoint はまだそのアカウントを削除していない。これは人事システムがいわゆる「ピュアソース」システムだからである。midPoint は人事に対し書き込むことはない。そこから読み込むだけである。CSV エクスポートファイルへ書き込みを行ったとしても、結局は次回エクスポート時に上書きされてしまうため書き込んでも意味がない。そのため人事リソースはそ

の構成に例外を定めてもらっており、そのおかげで人事アカウントはアサインがなくとも存在できるのである。この方法を利用して、我々はユーザーにリンク付けされた人事アカウントを保持することができる。我々はユーザー作成時に使用されたデータを確認できる。これにより全体の可視性が向上し、構成問題の診断に役立つ。

ロール

もしエリックが各リソースのアカウントを個別に各ユーザーに付与しなければならないとしたら、それはあまりに気の遠くなる作業だろう。一般に IDM のデプロイでは、ユーザーは数千人、リソースは数十になることが多い。そのようなデプロイでは、前述のようにただ直接アサインを行いながら管理していくことは非常に難しい。

そこで当然ながらもっとよい方法がある。ロールである。ロールベースのアクセス制御（*role-based access control :* RBAC）という概念は十分に確立した慣行であり、ロールはアイデンティティ管理における重要な要素である。RBAC では特権をロールにグループ化することが基本的な考え方である。ユーザーには特権ではなくロールが付与される。例として、「ウェブマスター」ロールを作成してみよう。ロールを作成したら、そこにウェブマスターが持つべき特権をすべてまとめる。そしてウェブマスターであるユーザー全員にこのロールを付与しよう。これで特権管理が簡単になる。ウェブマスターが 2 人いても、ウェブマスターが持つべき特権を個々に考えなくてよいのだ。ロールを付与するだけで、そのロールには必要なものがすべて含まれている。またウェブマスターの変更も簡単だ。ユーザーからロールのアサインを解除し、また別のユーザーに付与すればよい。ウェブサーバーを新規追加する場合も簡単である。その新しいサーバーへのアクセス権をウェブマスターロールに追加するだけである。そうすればウェブマスター全員がそのアクセス権を有することとなる。

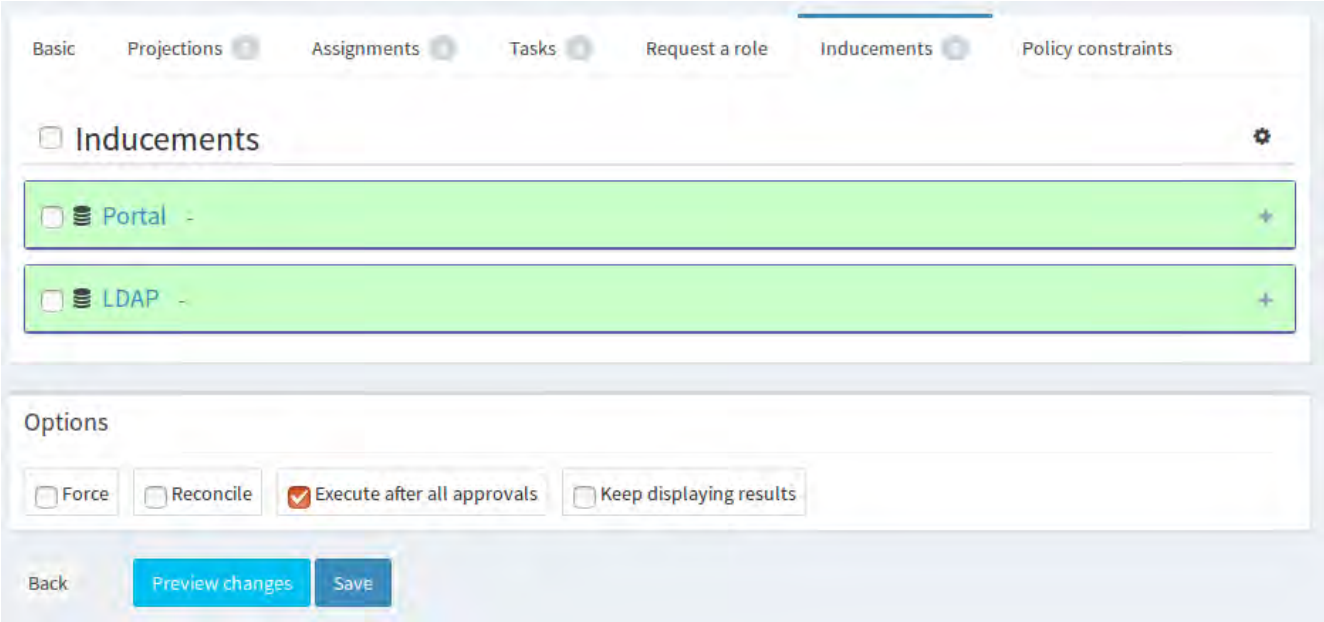
以上がロールの理論である。しかしエリックにはどう役立つだろうか？まず、ロールの材料を得るために新しいリソースをいくつか追加しよう。今リソースは人事、LDAP、CRM、ポータルの 4 つになった。手始めとしてはこれでよいだろう。では今からロールエンジニアリングを始めよう。

普通はどの組織にもほぼ全員がもつロールが一つあり、多くの場合それは「従業員」または「インターン」である。このロールは、従業員がアクセスできなくてはならないすべてのシステム（すなわち Windows ドメインへのログイン、e メール、従業員ポータルといったシステム）へのアクセス権を与える。ExAmPLE 社も例外ではない。この場合、基本ロールは 2 つのシステムにアカウントを作成すべきである。

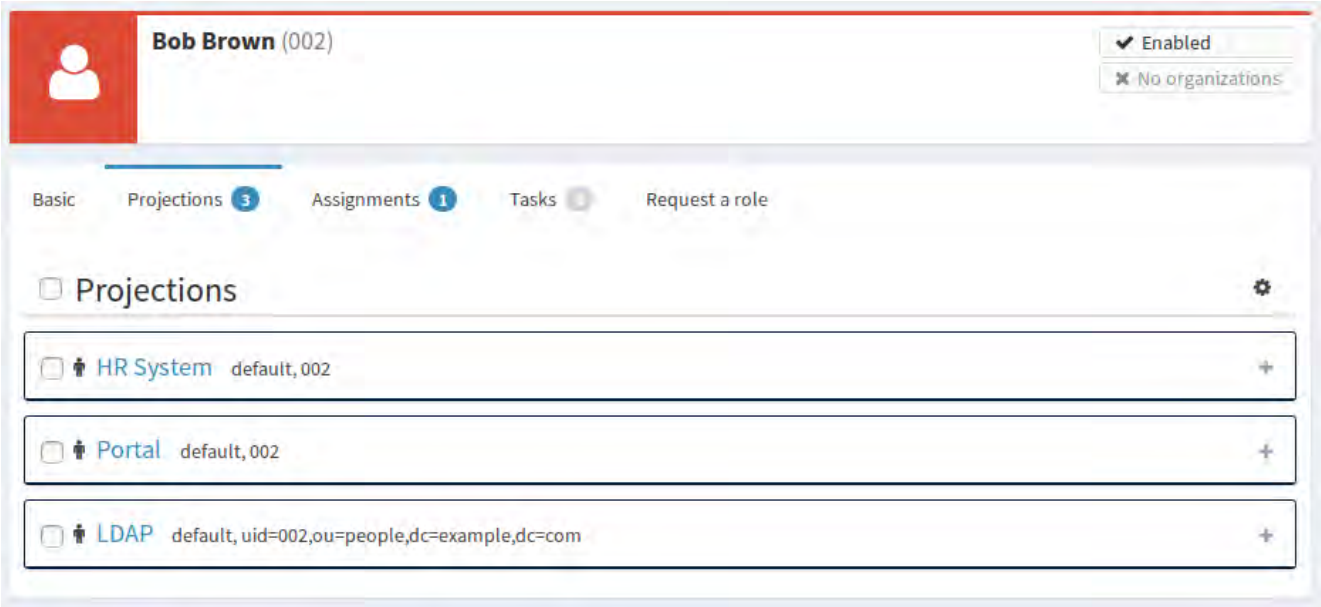
- LDAP サーバー : 多くのアプリケーションが LDAP に接続されており、LDAP を認証に使用している。ExAmPLE 社の従業員全員にそのアカウントをもたせたい。
- ポータル: 従業員全員に必要な、あらゆる細かいサービスが揃った企業イントラネットポータルである。

midPoint のユーザーインターフェイスなら簡単にできる。エリックは Roles（ロール） > New role（新ロール） に遷移する。新ロール名（「Employee（従業員）」）とその内容を入力する。次に「*Inducements（誘導）*」タブに移動する。ここではロール定義を行う。

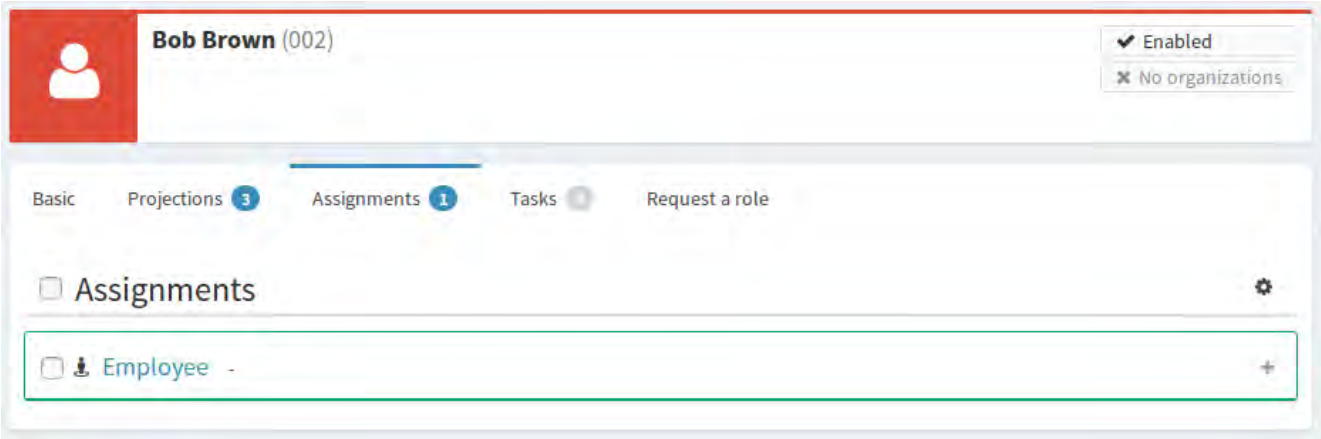
誘導はアサインとほぼ同じである。ただしロール自体にアクセス権を与えることはない。このロールを持つユーザーにアクセス権を与える。つまり一種の間接的なアサインである。エリックは歯車ボタンをクリックして 2 つのリソースにいての誘導をロールに追加する。



「Save（保存）」ボタンをクリックすると新ロールが作成される。これでロールをユーザーに付与できる状態となる。エリックは作業を進め、「*従業員*」ロールをユーザーの一人 Bob に付与する。するとこのロールが与えるアカウントがすべて midPoint で自動作成される。



まずボブのユーザーレコードの初回作成時に使用された人事アカウントがある。そしてボブが*従業員*ロールを持っていることで作成された 2 つのアカウントがある。



この操作はどちらの方向でも実行される。つまりエリックが*従業員*ロールのアサインを解除すれば、このロールが与えていたアカウントも削除される。エリックはこのようなロールをいくつでも作成できる。例えば CRM へのアクセス権を持つセールスエージェント用のロール、CRM のより高度な特権を持つ営業マネージャー用のロールなどである。midPoint は大量のロールを扱えるよう設計されている。各ロールはリソースを独自に組み合わせることができる。一人のユーザーが複数のロールを持ち、そのすべてのロールがユーザーに与えている特権を、midPoint はシームレスに結合する。例えば、2 つのロールが CRM アクセス権をそのユーザーに与えても、CRM アカウントは 1 つだけ作成される。そしてこの 2 つのロールのどちらかが解除されても CRM アカウントは残る。削除されないのは、もう一つのロールがまだそのアカウントを必要としているからだ。しかしその CRM ロールも削除されると、いよいよ CRM アカウントも削除される。midPoint はこのロジックすべてに対応する。

もちろん、ロールができることはもっとたくさんある。

- ロールはグループにアカウントをアサイン、特権を付与し、一般にはアカウントのエンタイトルメントを管理できる。
- ロールは所定のアカウント属性値（クリアランスレベル、コンパートメント等）を必須化できる。
- ロールにはカスタムロジック（スクリプト）を含めてもよい。
- ロールの中にロールがあるという階層構造でもよい。
- ロールを所定時間のみアサインすることもできる。
- ロールは条件付きロールおよびパラメータロールでもよい。
- その他にもたくさんある。

ロールはまさにアイデンティティ管理の真髄である。本書ではほぼすべての部分でロールが出てくるだろう。

その他多数の機能

エンジニアであるエリックは自社のアイデンティティ管理システムとなる midPoint を構成すべく、いくつか基本手順を行った。しかしこれはまだ本当に基本的な構成である。注意深い読者なら、まだやらなくてはいけないことがたくさんあることはとっくにお気づきだろう。例えば、従業員の氏名は自動生成されない。従業員の番号は識別子として使用されており、ここではもっと分かりやすいものが欲しい。*従業員*ロールを手動ではなく自動でアサインしたい、といったことである。改善すべきことはまだたくさんある。幸い、どこを見ればよいかさえ分かれば、こうしたことはすべて midPoint を使用して簡単にできる。また、これらはすべて本書にて後述する。新機能は各章で midPoint の原則について適切な説明を交えつつ、少しずつ ExAmPLE 社のソリューションに盛り込まれていく。midPoint は柔軟性の高い包括的なシステムであり、学ぶべきことはまだたくさんある。本章では midPoint 機能のごくわずかな部分を取り上げたにすぎない。

midPoint の本質ではないもの

今、あなたはおそらく midPoint の本質について何らかの考えをお持ちだろう。しかし、midPoint の本質ではないことを理解することも非常に重要である。アイデンティティ管理とアクセス管理（IAM）の分野は、時として非常に紛らわしいのかも知れない。おそらく midPoint チームが midPoint 機能についてあまり筋の通らない質問を受けることがあるのは、そのような理由からだろう。

第一に、midPoint は認証サーバーではない。midPoint はユーザー名とパスワードを検証するためのものではない。そう、ユーザーに関するデータ（パスワードを含む）を保守するものである。しかし midPoint が保守するデータモデルは非常に複雑である。直接アプリケーションにさらされるというわけではない。それでは効率がよくないだろう。

もし midPoint にユーザーを管理させたいだけでなく、アプリケーションに統合認証サービスをもたせたいなら、一つソリューションがある。それはデータを LDAP サーバーに公開することだ。midPoint は LDAP サーバーに簡単にデータを追加することができる。midPoint は LDAP サーバーで全ユーザーのアカウントを保守する。midPoint はパスワードを管理する。しかしアプリケーションが midPoint に直接話しかけることはない。LDAP サーバーに話しかけるのだ。これは誰にとってもありがたい。LDAP は多くのアプリケーションが対応している標準プロトコルである。また LDAP サーバーはきわめて高速であり、そしてこれでもかというほどの拡張性がある。よってここでは midPoint と任意の LDAP サーバーを組み合わせて使うとよいだろう。これが一般に行われるものであり、完璧に動作する。

midPoint が認証サーバーではないということは、当然ながらシングルサインオン（SSO）サーバーでもない。しかしもし SSO が必要ならば、midPoint の管理能力が必要となろう。また、拡張性のあるディレクトリシステム（LDAP）も必要となる。ただ実際の SSO を得るにはもう一つコンポーネントが必要だ。SSO サーバーである。SSO サーバーはクローズドソースとオープンソースの両方についてたくさんの選択肢がある。LDAP と組み合わせるなら、midPoint が管理するソリューションが一番うまくいくだろう。

多くの混乱に見舞われそうなことの一つが認可である。最初に誤解のないよう事実をはっきりさせておこう。midPoint は認可サーバーではない。ポリシー定義点（Policy Decision

Point：PDP）でもなければ、もちろんポリシー施行点（Policy Enforcemet Point：PEP）でもない。アプリケーションから認可を抜き取り、ただ「そのために midPoint を使用」することはできない。それでは機能しない。

midPoint をポリシー管理点（policy management point：PMP）として考えることはできる。midPoint のコア内には、実に洗練された認可関連ロジックがある。しかしそのロジックは「サブジェクト S はオブジェクト X で操作 O を実行する権限があるか？」といった質問に答えるためのものではない。midPoint のロジックは独特である。midPoint は認可判断には関係していない。認可ポリシーの管理を取り扱う。midPointは対象アプリケーションに認可ポリシーを設定する。するとアプリケーションがこれらのポリシーを評価する。これははるかに効率的でより信頼できる方法である。認証とはちがい、認可の判断は常時行われている。要求があるごとに少なくとも 1 回は行われるが、通常は要求につき複数回であることが多い。アプリケーションが内部でこうした判断を行えば、認可サーバーを往復する必要はない。パフォーマンスは大幅に向上する。しかも単一障害点はない。アプリケーションはその内部にすべてのデータをもっているため、midPoint にエラーがあっても認可フローが妨げられることはない。エラーになるコンポーネントが一つ減れば、システムの信頼性は高まる。しかも認可ポリシーは midPoint が一元管理する。ポリシー変更があれば、midPoint がすべての関連アプリケーションを更新する。お決まりのデメリットを抱えることなく、すべてのメリットを得られるのだ。

midPoint はすべきことを行う。すなわちアイデンティティ、エンタイトルメント、組織構造、ポリシーを管理することである。しかし必要でないことは行わない。また、すでに他の技術で十分まかなえることも行わない。midPoint は車輪の再発明のようにわざわざ同じものを一から作りなおすことはしない。その必要がないからだ。midPoint は車輪などではない。midPoint はすべての車輪の上に成り立っている。midPoint はその運転手である。

第 4 章： インストールの原則および構成原則

「ガイド」にまちがいはない。現実のほうがしょっちゅうまちがっているのだ。

– ダグラス・アダムス「銀河ヒッチハイク・ガイド『宇宙の果てのレストラン』」

本章では midPoint システムのインストールおよび初期構成の手順について説明する。手順内容は Linux システムでインストールする場合とする。それが midPoint にとって群を抜いて最も一般的なオペレーティング環境だからである。ただし、midPoint はプラットフォームに依存しておらず、Java が実行される環境であれば稼働できる。経験豊かなエンジニアなら、本書の手順を難なく他のオペレーティング環境に適合させて実行できるだろう。

本章に記載の midPoint インストールは非常に基本的なものである。midPoint をはじめて学ぶ場合や、midPoint 構成の開発、デモ、その他同様の目的であれば、これが最適である。実に便利なインストールであり、これを使用して日々の開発作業を行っている。ただし本番環境へのデプロイで使用するなら、このインストールを少々調整する必要がある。調整については本章で説明するが、本番対応のインストールに関する詳細な説明は後半の章で行う。本章では midPoint の入門編として最適なインストールを説明する。

要件

midPoint は、ほぼどのマシンでも稼働する。必要となるのは約 4GB の RAM だけだ。おそらく本当の制約要素となるのはこれだけだろう。もっと正式なシステム要件の定義を確認したければ、midPoint wiki をご覧いただきたい（「追加情報」の章を参照）。

ソフトウェア側で必要なものは次のとおり。

- **Java 8** 実行環境（JRE）または 開発環境（JDK）。JRE または JDK が動作すること。OS 配布のパッケージを使用してもよい。または Java をダウンロードしてスタンドアロンパッケージとしてインストールしてもよい。どちらも動作するはずである。ただ、JAVA のインストールを指定するため PATH の設定および環境変数 JAVA_HOME の設定を忘れずに行うこと。また無制限強度の暗号化のため、JAVA 暗号化拡張機能（JCE）をダウンロードし、これをインストールしてもよい。ただしこれがなくても、midPoint は機能できる。
- **midPoint 配布パッケージ。** Evolveum ウェブサイトより midPoint の最新版をダウンロードすること。あなたは midpoint- 3.7.1-dist.zip.のようなアーカイブを探している。このアーカイブには、midPoint の実行に必要なものがすべて含まれている。都合のよい一時作業ディレクトリにアーカイブを抽出すること。

midPoint を扱うときは、どのソフトウェアパッケージも入手可能な最新バージョンを使用するようお勧めする。我々は midPoint を常に最新の状態に保ち、他の技術に遅れないようにするために最大限努力している。

midPoint のインストール

midPoint は Java web アプリケーションである。しかしバージョン 3.7 以降はスタンドアロン型パッケージでの配布となっている。配布パッケージには midPoint の実行に必要なものがすべて（Java プラットフォームそのものは除く）揃っている。よって Java プラットフォームがインストール済みであれば、midPoint を実行するには以下を始めるだけでよい。

1. midPoint 配布パッケージから start.sh（Unix）または start.bat（Windows）スクリプトを見つける。これらは bin ディレクトリに格納されている。
2. スクリプトを実行する。

これでほぼ終了だ。midPoint が起動し、埋め込み web コンテナやデータベースのほか、すべての midPoint コンポーネントが初期化される。これには 1〜2 分かかる場合もある。アプリケーションが初期化されると、midPoint の HTTP ポート（デフォルトのポート番号は 8080）に接続してアクセスできる。これで midPoint を使用できるようになる。

midPoint のユーザーインターフェイス

以下の URL より midPoint のユーザーインターフェイスにアクセスする。

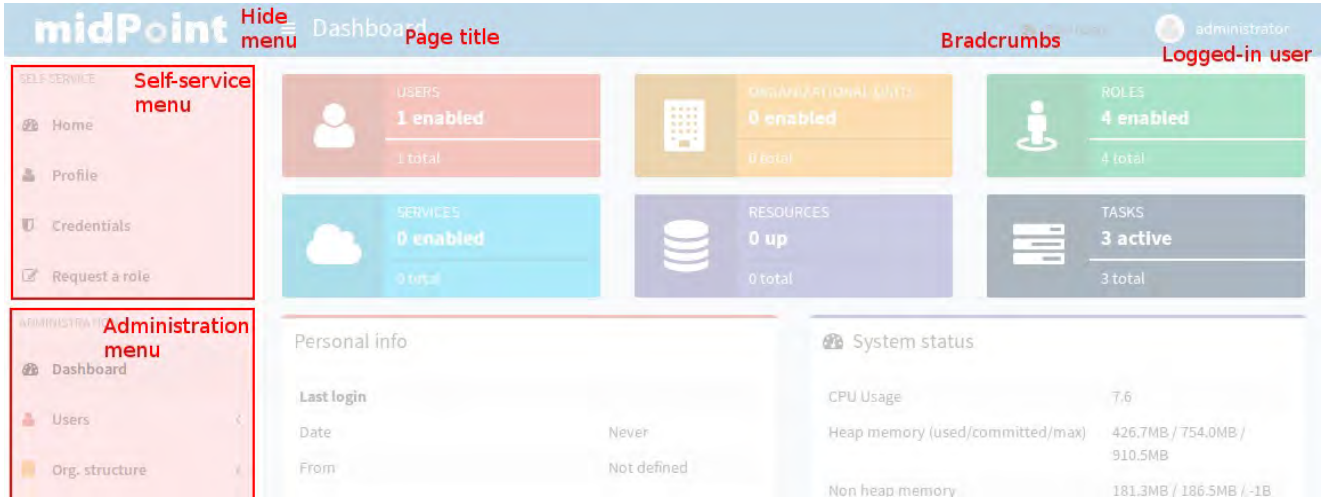
http://*hostname*:8080/midpoint

以下のクレデンシャルを入力してログインする。

ユーザー名：administrator
パスワード： 5ecr3t

これで管理者（administrator）としてログインしていることになる。管理者であるユーザーはスーパーユーザー権限を持っているため、midPoint の GUI 上でなんでも閲覧、実行できる。

midPoint GUI の外観は構造化されており、どのユーザーインターフェイスエリアであってもレイアウトと制御は同じである。



ユーザーインターフェイスの操作でメインとなるツールはメニューである。ユーザーインターフェイスは機能上は 3 つの部分に分かれており、そのためメニューも 3 部ある。

- **Self-service（セルフサービス）**ユーザーインターフェイスでは、ユーザーが自分で行えること（アカウントリストの表示、パスワード変更、ロール要求等）を取り扱う。ここは比較的シンプルなユーザーインターフェイス部である。しばしばすべてのユーザーが利用できる。
- **Administration（管理）**ユーザーインターフェイスでは、他のユーザー、ロール、組織構造のほか、同様の midPoint オブジェクトの管理を取り扱う。このユーザーインターフェイスは非常に包括的でかなり複雑である。通常は特権ユーザーのみがアクセスできる部分である。委任管理およびロール管理の対応に使用されることが多いため、セキュリティ担当者、リソースオーナー、ロールエンジニアのほか、同様の専門的なユーザー向けの部分である。
- **Configuration（構成）**ユーザーインターフェイスでは、midPoint 自体の構成を取り扱う。midPoint の挙動のカスタマイズ、midPoint をデプロイする際の基礎となる基本ポリシーやルールの設定に使用される。通常はアイデンティティ管理エンジニアのみが使用する部分である。

ユーザーインターフェイスのエリア

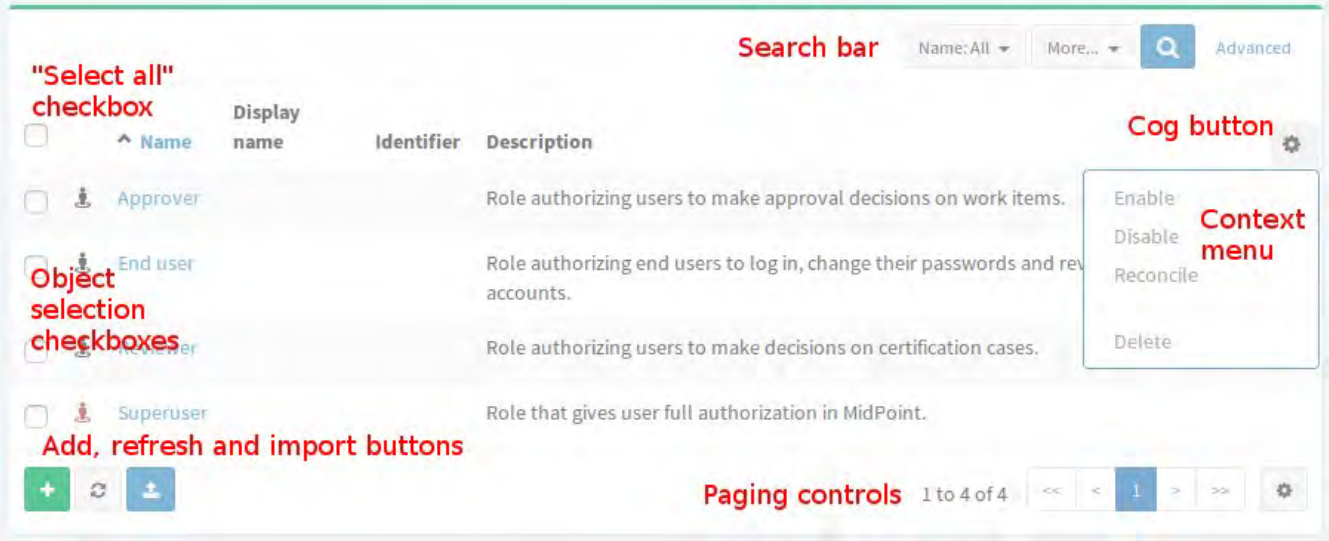
midPoint のユーザーインターフェイスは実に豊富である。そのユーザーインターフェイスの最も重要な部分について簡単にまとめたものが以下のリストである。

- **Home（ホーム）**ページには、ユーザー自身のアカウント、要求、作業項目等に関する簡単なステータスが表示される。midPoint にログインしたエンドユーザーが最初に目にするページである。
- **Profile （プロファイル）**ページでは、ユーザーが自身のプロファイルを閲覧、編集できる。
- **Credentials（クレデンシャル）**ページでは、ユーザーが自身の認証情報（パスワード等）を変更できる。

- **Role request （ロール要求）**ページでは、ユーザーが必要なロールを選択しそのアサインを要求できる。
- **Dashboard （ダッシュボード）**ページには、midPoint のインストールに関する統計情報が表示される。これはシステム管理者が一目でシステムの状態を確認できるシステムダッシュボードである。
- **User （ユーザー）**ページには midPoint ユーザーが一覧表示され、ユーザー編集、ユーザーの新規作成ができる。
- **Organizational structure （組織構造）**ページには組織図が表示される。ここではツリーのような機能別組織構造やフラットなプロジェクト型組織、ロールカタログ等のあらゆる並列した組織構造を管理できる。
- **Role（ロール）**ページではロールを一覧表示し管理できる。ユーザーインターフェイスのこの部分で、ロールの作成および定義を行う。
- **Service （サービス）**ページでは、デバイス、サーバー、アプリケーションといったサービスを一覧表示し定義できる。
- **Resource （リソース）**ページではリソースを一覧表示し管理する。ここではコネクタに関連して新リソースを定義、テスト等ができる。
- **Work item（作業項目）**ページには、ユーザーがやるべきことが一覧表示される。何かを承認しなければならないときや、ワークフロー内で手動での作業が発生すると作業項目が作成される。
- **Certification （検証）**ページではアクセスの検証（再認定、認証）を取り扱う。ここでは検証キャンペーンを作成、管理できる。
- **Server task（サーバータスク）**ページには midPoint サーバーで稼働中のタスクが表示される。そのタスクとは、定期的に行われる同期、インポート、ユーザー要求の実行などである。これらはサーバー上で実行されるが、すぐに同期させて行うことはできないものである。
- **Report （レポート）**ページではレポートを定義、実行できる。これらのページでは一般に印刷可能な定期レポートを取り扱う。
- **Configuration（構成）**エリアは、システムのデフォルト構成、レポジトリオブジェクト、ロギング、一括処理等、midPoint の構成を管理する多くのページで構成されている。

ユーザーインターフェイスの概念

midPoint のユーザーインターフェイスでは、できるかぎりすべての部分に同じ概念が使用されている。例えば、すべてのオブジェクト（ユーザー、ロール等）のリストはすべて下図のような外観である。



表の各行は一つのオブジェクト（ユーザー、ロール、サービス、タスク等）を示している。オブジェクト名をクリックすると、通常はそのオブジェクトの詳細ページが開く。また、色分けされたオブジェクトアイコンもある。上部の検索バーを使用すれば、特定のオブジェクトを検索したり、オブジェクトをフィルタ表示させたりできる。「歯車ボタン」は midPoint で何かを実行するための万能ボタンである。通常は歯車ボタンをクリックするとアクションのコンテキストメニューが開く。表ヘッダー部の歯車ボタンには、選択されたオブジェクトすべてに適用されるアクションが含まれている。各行の歯車ボタンには、個々のオブジェクトのみに適用されるアクションが含まれている。左下隅のボタンは、新オブジェクトの作成やインポート、画面の更新といったグローバル処理を実行する。「Import（インポート）」ボタンは特に有用である。このボタンを使えば、新オブジェクトを XML/JSON/YAML 形式でインポートできる。ページングの制御は右下隅で行う。

midPoint ではナビゲーション操作がより簡単になるよう、カラーコードを統一している。ユーザーやロールのほか、オブジェクトタイプにはそれぞれ所定のカラーとアイコンがある。これはオブジェクトのタイプを示しており、メニュー、情報ボックス、リスト、ボックスタイトルのアクセント等にできるかぎり使用されている。主なカラーおよびアイコンはダッシュボードに表示されている：

ユーザー関連の制御はすべて赤色、組織構造を扱う制御はすべて黄色、そしてロールは緑色、といった具合である。このカラーコードは midPoint のユーザーインターフェイスのほぼ全般に一貫して適用されている。

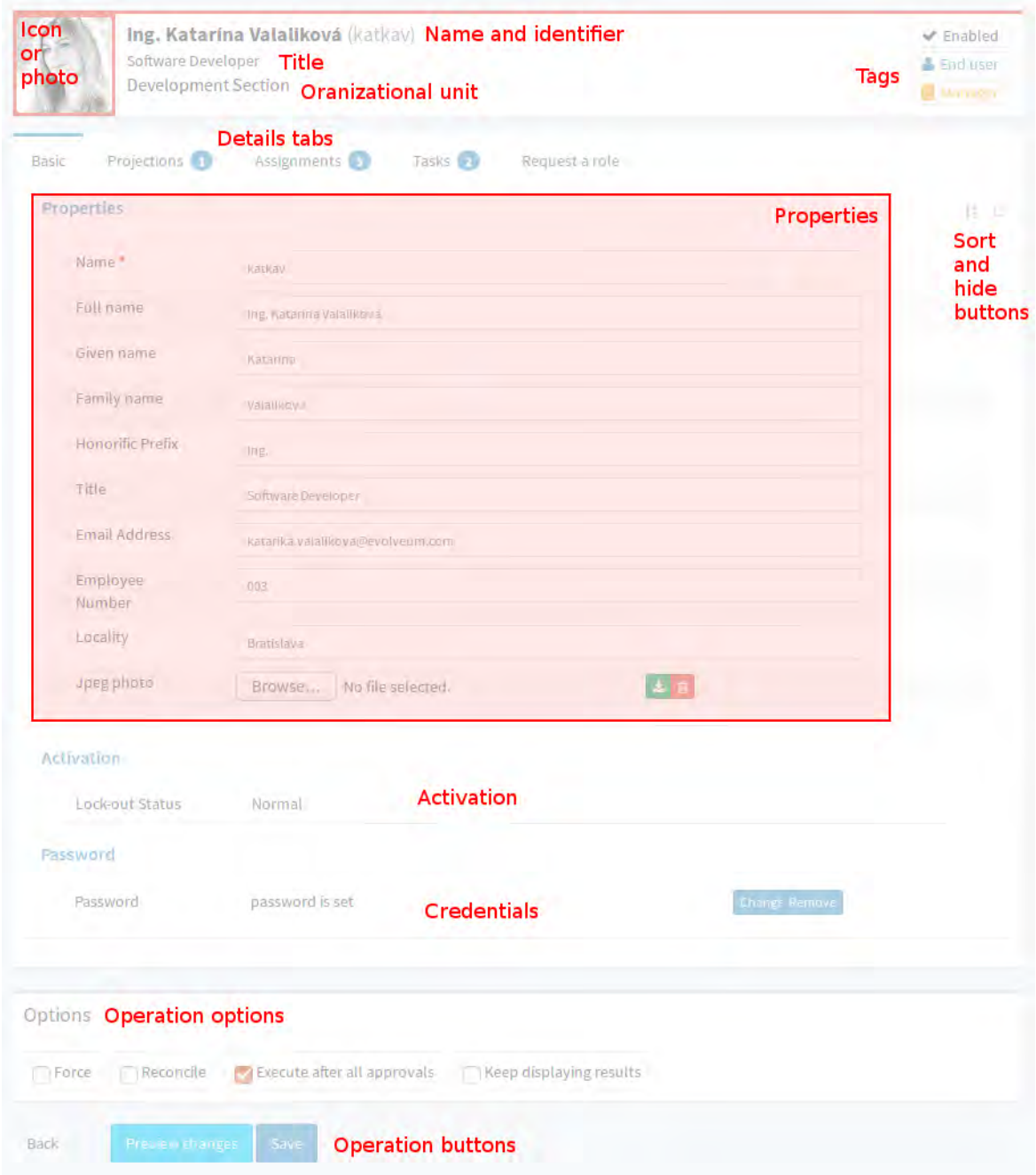
ユーザーリスト上に表示されるオブジェクトアイコンにも同様のカラーコードが適用されている。ただし、そこで使用されているカラーは（当ページのアイコンを見れば明らかなように）オブジェクトのタイプを示すものではない。アイコンのカラーはオブジェクトの状態を示している。

- 黒色のアイコンは通常の状態を示し、特に留意すべきものはないことを示唆している。
- 灰色のアイコンは非アクティブな状態を示す。オブジェクトが無効かアーカイブされている、あるいはアクティブでない理由が他にあることを示唆している。
- 赤色のアイコンは特権のある状態であることを示す。オブジェクトがより高度な特権をもっているか、セキュリティの観点から注意が必要かもしれないことを示唆している。例：管理者ユーザー、スーパーユーザーのロール等。
- 青色のアイコンは典型的なエンドユーザーによるアクセスを示す。オブジェクトはアクセスされているが、安全かつ非特権的な操作に限定されていることを示唆している。例：エンドユーザーロールをもつユーザー。
- 黄色のアイコンは管理能力を示す。例：組織単位のマネージャーであるユーザー。

情報：midPoint ではどのオブジェクトもほぼ等しく、ユーザー、ロール、組織単位、サービスはみなほとんど同様に扱われる。これらのオブジェクトを表示するリストも同じであり、オブジェクトの詳細を表示する各ページも同じである。すべてのオブジェクトにプロパティがあり、同じように有効化／無効化でき、ほぼ同様に認可を受けなければならない等である。強力な原則をいくつか設計して、それを繰り返し適用するのが midPoint の基本的な姿勢である。

オブジェクトの詳細ページ

オブジェクト名をクリックすると、通常はそのオブジェクトの詳細ページが表示される。ユーザーやロール等、midPoint の一般的なオブジェクトの詳細ページは互いによく似ている。レイアウトと制御は同じである。たとえば、ユーザーの詳細ページは次のように表示される。



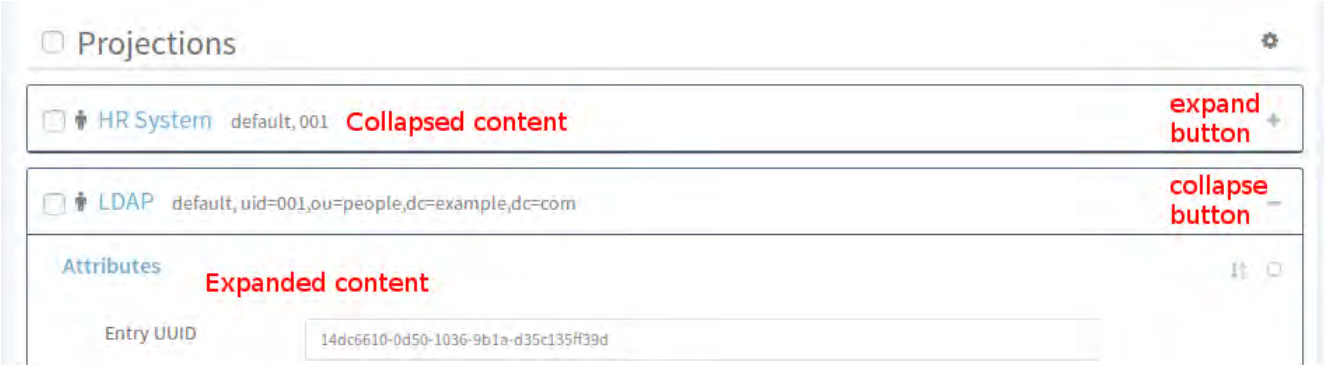
ページの上部は情報エリアである。このエリアにはユーザーの写真（またはアイコン）のほか、ユーザー名や ID などの基本情報が表示される。また、組織構造内におけるそのオブジェクトの所属先も表示する。さらにいくつかの「タグ」もあり、オブジェクトに関する興味深い情報（オブジェクトは有効かどうか、特権をもっているか等）を表示する。

情報エリアの下の画面はいくつかのタブに分かれており、それぞれのタブでオブジェクトの各側面が表示される。通常、一つ目のタブにはオブジェクトのプロパティが表示される。それ以外にも「projections（プロジェクション）」、「assignments（アサイン）」「inducements（誘導）」を表示するタブがある。これらは本書の後半で説明する。いずれにせよ、どのタブも現在表示されているオブジェクトに関する情報を表示するものである。おそらく今のところは一つ目のタブが一番興味深いものだろう。このタブにはオブジェクトのプロパティ、すなわち属性を表示するダイアログが含まれている。属性が表示され、現在ログイン中のユーザーの認可によっては編集も可能である。基本的なプロパティだけでなく、他にもセクションがある。例えば Activation（アクティベーション）セクションでは、オブジェクトの有効／無効、アクティベーションの日付、その他アクティベーションに関する情報が表示される。Credentials（クレデンシャル）セクションでは、パスワード等の認証情報を変更できる。Properties（プロパティ）セクションの上部には小さなボタンが 2 つある。一つはプロパティの表示順を変更できるボタンである。そしてもう一つはプロパティの表示方法を全表示または一部表示を切り替えるものである。

操作のオプションおよびボタンは詳細ページの下部に配置されている。これらのボタンによって操作が開始される。最も一般的な操作は変更を保存することであり、それが [Save（保存）] ボタンの目的である。変更を保存することは、ユーザープロパティの変更、ロールのアサイン、パスワードの変更、ユーザーの無効化など、ほぼすべての操作を開始する万国共通の方法である。詳細ページのどこかのタブで編集を行っても実際にはまだ何も起こらない。midPoint は編集していることを覚えているだけである。実際にものごとが起こるのは保存ボタンをクリックするときのみである。これが、ひとつの操作で複数のことを実行するという仕組みである。これに慣れるには少し時間がかかるかもしれない。とにかく保存ボタンのクリックを忘れてはならない。

ボタンの上にある操作オプションは、操作の挙動を変更するのに使用する。これらのオプションは midPoint の内部チェックをパスしない操作であっても強制的に実行させることができる。たとえ midPoint がデータ調整は不要と考えていても、リコンシリエーションを行うオプションがある、といったことである。これらのオプションにチェックマークを付けるか否かによって、midPoint がどう操作を実行するかが変わってくる。

midPoint のユーザーインターフェイスでは、内部的にとても複雑なオブジェクトを表示しなければならないことが多い。しかも非常にコンパクトに。ユーザーのアカウント、アサイン、ロール誘導などのリストがこれにあてはまる。この場合、midPoint は展開可能なビューを使用する。オブジェクトは最初折りたたまれた状態で、基本データのみが表示されている。こうしたオブジェクトは展開してより詳細情報を表示させることができる。



midPoint ユーザーインターフェイスの展開／折り畳みボタンを見てみよう。これらのボタンをクリックすれば、求めている追加情報を得ることができるだろう。また、通常はオブジェクトのラベルをクリックしてもビューが展開される。

midPoint 構成の基本事項

midPoint 構成の原則は、一般的なシステム管理者が期待するものとはまったく違う。midPoint には構成ファイルがほとんどない。midPoint はその構成の大部分を自身の構成データベース内に格納している。これにはいくつか理由がある。

- midPoint の構成は**複雑**である。これは一般的なシステム管理者が想定しているような「構成」ではない。midPoint には多数のリソース定義が含まれており、そのリソース定義にはマッピングが含まれている。マッピングにはさらにスクリプトが含まれている場合もある。ロール、ポリシー、テンプレートなどがあり、これらのオブジェクトはあまりにも複雑なためシンプルな構成ファイルでは表現できない。
- midPoint の構成は**拡張性**がある。例外なく、一つの midPoint デプロイで数千のロール定義または組織単位、数十のリソース定義、そして数多くのテンプレートおよびポリシーがある。これらすべてを効率的に格納する必要があり、よって midPoint は数百万人のユーザーを伴うデプロイを取り扱うことができる。構成は検索可能であることも必要である。テキストファイルで数千のロールを管理してもまったくうまくいかないだろう。
- midPoint の構成には**可用性**が必要である。midPoint にはクラスターとして機能するノードが複数あるデプロイもある。片方のノードで行われた構成変更はもう一方のノードで見えなくてはならない。この場合シンプルな構成ファイルではうまくいかないだろう。

midPoint が構成についてまったく違う手法をとっているのはこのような理由からである。構成は定義済みの構造化オブジェクトの形式で midPoint のデータベースに格納される。我々はこのデータベースを *midPoint レポジトリ*と呼んでいる。

構成オブジェクト

midPoint ではすべてがオブジェクトである。構成の一つ一つの部位が構造化オブジェクトとして表され midPoint レポジトリに格納される。それは下記のような形になる。

```
<role oid="8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc">
        <name>Basic User</name>
        <requestable>true</requestable>
```

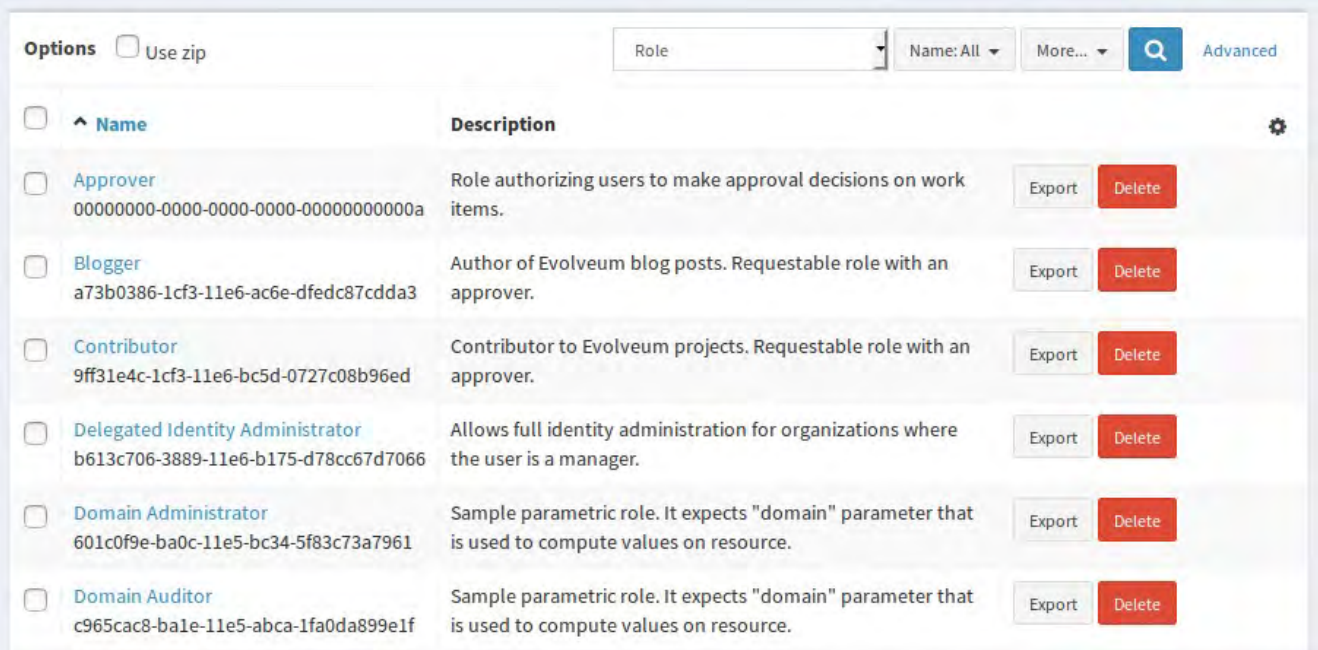
```
        <roleType>business</roleType>
        <inducement>
                <targetRef oid="f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61" type="RoleType"/>
        </inducement>
</role>
```

オブジェクトはそれぞれ識別子をもっている。我々はこの識別子を **OID** と呼んでいる。これは単に「オブジェクト識別子（Object Idenfitifer）」を短縮したものである（LDAP やASN.1OID とは無関係）。この識別子はたいていはランダムに生成されたUUID である。そしてシステム全体で一意の値でなくてはならない。一度オブジェクトに付与された ID は決して変わることはない。これがいわゆる*永続 ID である*。内部的に使用するものであり、midPoint ユーザーが目にすることはまずない。

すべてのオブジェクトには**名前**もある。名前は人間が判読できいつでも変更できる。ユーザーに表示される値である。

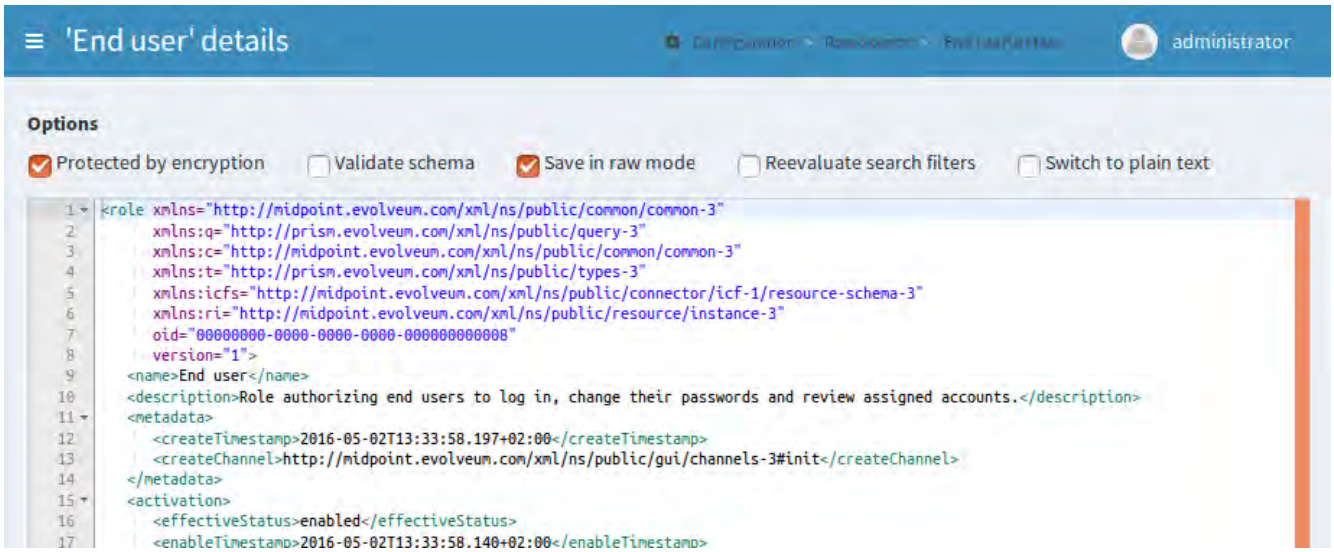
そしてオブジェクトにはプロパティがある。いや、むしろ「項目」だ。midPoint の各オブジェクトタイプがもつ項目の組み合わせは少しずつ違っている。これがいわゆる「*スキーマ*」である。項目は文字列、整数、Boolean 値といったシンプルなプロパティのこともあれば、複雑な構造であったり、オブジェクト間の参照であることもある。データモデルは非常にリッチである。実際にその説明に本書の大半を占めているほどリッチである。というのもデータモデルの説明は midPoint 機能の説明でもあるからだ。

midPoint の構成オブジェクトは、midPoint ユーザーインターフェイスにて *Configuration（構成）* > *Repository Objects（レポジトリオブジェクト）*へ遷移し、オブジェクトタイプを選択すれば確認できる。下図は「ロール」タイプのオブジェクトを表示したものである。



XML、JSON、YAML

オブジェクトは midPoint ユーザーからは見えないネイティブ形式で midPoint レポジトリに格納される。ただし人間が判読できる表記もあり、XML 形式、および（midPoint3.5 以降は）JSON 形式、YAML 形式でも表すことができる。オブジェクトはすべて XML 形式で midPoint にインポートでき、同じく XML 形式で midPoint からエクスポートすることもできる。さらに埋め込みエディタを使用すれば midPoint 上で直接編集もできる。［*Repository objects（レポジトリオブジェクト）*］ページ上で任意のオブジェクトをクリックするだけである。



オブジェクトを XML/JSON/YAML 形式でエクスポート、インポート、編集できる機能は不可欠であるが、それは以下の理由によるものである。

- **人間が判読できる。**（管理者が判読できるといったほうが適切であろう）。自分の好きなエディタで構成を作成、編集、保持でき、midPoint にインポートできる。内容の確認ができる。コピー＆ペーストもできる。これは特にそうだろう。コピー＆ペースト機能がなければ、システム管理者は効率的に作業できないからだ。
- **転送できる。**あるシステム（開発環境等）からエクスポートでき、別のシステム（テスト環境等）に容易に転送できる。バックアップや復元も簡単である。データ共有（構成サンプルの形などで）も簡単である。
- **バージョン管理ができる。**エクスポートされた構成を一般のバージョン管理システムに容易に置くことができる。これはデプロイプロジェクトや構成管理において不可欠である。

よって midPoint 構成は両方の環境にとって最善のものをもっている。構成はデータベースに格納されるため、効率的に処理でき、使用可能になるといったことがある。しかしテキスト形式も有しているため、管理も容易である。

XML 形式、JSON 形式、YAML 形式は同等と考えられる。midPoint 3.5 以降ではこれらのどの形式でもオブジェクトを書き込むことができる。

XML 形式：

```
<role oid="8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc">
        <name>Basic User</name>
        <requestable>true</requestable>
        <roleType>business</roleType>
        <inducement>
                <targetRef oid="f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61" type="RoleType"/>
        </inducement>
</role>
```

JSON 形式：

```
{
        "role" : {
                "oid" : "8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc",
                "name" : "Basic User",
                "requestable" : true,
                "roleType" : "business",
                "inducement" : {
                        "targetRef" : {
                                "oid" : "f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61",
                                                "type" : "RoleType"
                        }
                }
        }
}
```

YAML 形式：

```
role:
        oid: "8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc"
        name: "Basic User"
        requestable: true
        roleType: "business"
        inducement:
                - targetRef:
                        oid: "f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61"
                        type: "RoleType"
```

本書に出てくる例はほとんどが XML 表記を使用している。XML 形式は、わかりやすくするためほとんどが簡素化されている。名前空間の定義もなければ、名前空間の接頭辞もない、といった具合である。すべての詳細を含めた完全なファイルは midPoint の配布パッケージ、midPoint ソースコード、またはそれ以外の場所にある。詳細は追加情報の章を参照されたい。

midPoint 構成の保守

midPoint 構成を保守するには、実用的な方法が 2 つある。

一つ目は midPoint にて構成を保守する方法である。midPoint のウィザードとユーザーインターフェイスツールを使用してオブジェクトの新規作成や変更を行う。バックアップをとり、バージョン管理下に置くため等の目的から、オブジェクトは定期的にエクスポートする。まずはじめるならこの方法が簡単である。ただ遅かれ早かれ、構成をコピー＆ペーストする機能が必要だときっと気付くだろう。構成を他のチームメンバーと共有する必要も。そしてユーザーインターフェイスの効率性はすぐれたテキストエディターを持つ経験

豊かなエンジニアほどではないことにも気付くだろう。

そこで 2 つ目の方法がある。それは midPoint の外部にテキスト形式で構成ファイルを保守し、必要に応じて midPoint にインポートする、というものである。midPoint ユーザーインターフェイスの *Configuration（構成）> Import object（オブジェクトをインポート）* ページに遷移してオブジェクトをインポートできる。またほぼすべてのオブジェクトリストテーブルには青色のインポートアイコンがあり、このページへと導いてくれる。オブジェクトを再インポートする際は上書きオプションのチェックを忘れないこと。

このインポート方法を使用すれば、正しいバージョン管理と良好なチームワークをはるかに簡単に保つことができる。midPointに慣れてしまえばもっと効率はよくなると思われる。構成の一部をサンプル、ドキュメント、または他のプロジェクトからコピーできるからだ。これで効率的に作業ができる。midPoint での作業は「まさに」構成であり、通常はプログラミングはほぼないのだが、この作業方法はソフトウェア開発者が使用する方法にきわめて近い。そして我々はこうした方法が非常に効率的であることを知っている。

midPoint の外部で構成ファイルを保守すれば、midPoint ユーザーインターフェイスを使用してファイルを個々にインポートできる。これにはかなり違和感を感じるかもしれない。しかし中規模のプロジェクトでさえ、この方法は驚くほどうまくいくのだ。しかし midPoint 3.5 以降であれば、もっとよい方法がある。Eclipse IDE 環境へのプラグインがあるため、Eclipse プロジェクトの形式で構成ファイルを保守できるのだ。このプラグインによって、変更した構成ファイルも簡単にダウンロードおよびアップロードできる。Eclipse はバージョン管理システムおよびその他の開発ツールともよく統合するため、大規模かつ複雑なプロジェクトには理想的な手法のように思われる。

インストールについて

我々は今 midPoint インストールを実行中であり、あなたはその構成方法をある程度理解しているはずである。しかし構成についてさらに詳しい内容に入る前に、midPoint のインストールを見てみよう。先へ進む前に理解しておくべきことがいくつかある。今理解しておけば、あとで大幅な時間の節約になるだろう。

midPoint は動作するためデータベースを持つ必要がある。このデータベースは構成、ユーザー、リソース定義、アカウントリンク、監査証跡のほかあらゆるものを格納するのに使用される。本番へのデプロイには適切な関係データベース（PostgreSQL、MySQL/MariaDB、Microsoft SQL、Oracle のいずれか）を強くお勧めする。しかし開発およびデモ目的であれば、midPoint に含まれている組み込みデータベースエンジン（H2 データベース）がある。デフォルトでは midPoint のインストール時にこの組み込みデータベースは初期化される。midPoint の新規デプロイにおいて、すぐに構成オブジェクトの格納に使用されるのがこの組み込みデータベースである。このデータベースには特別な構成は不要で、データベースのスキーマが自動で適用され、midPoint と共に起動、停止される。それゆえ、開発用途に最適である。

midPoint 構成の大部分はこのデータベース（midPoint レポジトリ）に格納される。しかしここには格納できないものもいくつかある。たとえばデータベースそのものへの接続パラメータだ。この目的のために midPoint には config.xml.という小さな構成ファイルがあ

る。midPoint にはデータベースに格納できないデータ（暗号化キー、コネクタバイナリ等）を格納する場所が必要である。

そのため midPoint にはファイルシステム上に特別なディレクトリが必要である。我々はこれを *midPoint ホームディレクトリ*と呼んでいる。場所が明示的に指定されていない場合は、midPoint の配布パッケージがインストールされたディレクトリ、すなわち midPoint の起動スクリプトが置かれている bin ディレクトリと同じレベルに、名前 var 付きのディレクトリが新規作成される。midPoint が/opt/midpoint ディレクトリにインストールされたと仮定しよう。すると midPoint ホームは/opt/midpoint/var ディレクトリに配置される。

この midPoint ホームディレクトリの場所は midpoint.home Java システムプロパティを使用して変更できる。これは JVM コマンドラインで-Dmidpoint.home を指定して行う。または、midPointのデフォルト起動スクリプトを使用している場合は、MIDPOINT_HOME 環境変数を使用して midPoint のホームディレクトリの場所を設定できる。

midPoint は初回起動時にデータベースが空っぽであることを検出する。すると最小限の構成オブジェクトをデータベースに格納する。これには Superuser ロールやユーザー administrator 等のオブジェクトが含まれている。これらのオブジェクトがないと midPoint の新インスタンスにログインできないため、自動でインポートされるようになっている。これらはデータベースにまだないときのみインポートされる。あとでこれらを変更しても、midPoint は上書きは行わない。

ロギング

midPoint に何か問題があるとき、これを診断する最も重要な仕組みはおそらくロギングであろう。midPoint のユーザーインターフェイスが何かおかしいと思うとき、あなたの頭の中に思い浮かぶのはロギングのはずだ。我々は midPoint のユーザーインターフェイスを使いやすくしようと努力しており、高品質なエラーレポートに大いに注意を払っている。とはいえ限界はある。ユーザーインターフェイスが表示するエラーは長々と続いた原因と影響の結果であるため、主要原因を直接指し示すものではないかもしれない。または、実はまったくエラーはなく、midPoint が本来すべきことをしていないだけかもしれない。ここでロギングが役に立つ。

midPoint ではメッセージのログをとるのに Java ロギング機能を使用している。midPoint のログファイル名は midpoint.log であり、midPoint ホームディレクトリの log サブディレクトリ（/opt/midpoint/var/log/midpoint.log）に格納される。ロギングレベルは、デフォルトでは通常の midPoint 操作に合うように設定されている。これは INFO 以上のレベルのメッセージはログがとられ、それよりも詳細なレベルのメッセージはログがとられていない、ということになる。もし midPoint の問題を診断したければ、ロギングレベルを DEBUG か、極端な場合はTRACEにまで切り替える必要があるだろう。ロギングレベルはmidPointユーザーインターフェイスにて調整できる。調整するには *Configuration（構成）> Logging（ロギング）*に遷移する。

第 5 章： リソースおよびマッピング

悲観主義者は風に恨みを言い、楽観主義者は風が
変わるのを期待し、現実主義者は帆を合わせる。

– ウィリアム・アーサー・ウォード

リソースオブジェクトの読み書き、属性の同期、属性値のマッピング、スクリプトを使用した変換—これらはみな midPoint の基本機能である。自尊心ある IDM デプロイにはこれらの機能が絶対に不可欠であり、IDM エンジニアであればこれらの機能に精通しているべきである。そしてこれがまさに本章の目的であり、プロビジョニングエンジンとして midPoint を使用するのに必要な構成を説明する。

すべてのシステムが同じインターフェイス、通信プロトコル、スキーマで一致すると期待するのは現実的でない。これまでにも IAM のランドスケープを統一しようとする試みはいくつかあった。しかしそれらは必ずしも成功しなかった。LDAP プロトコルは 1990 年代からあるが、その実装の相互運用はいまだに 100%ではない。アイデンティティプロビジョニングにおいてはさらに悪い。プロビジョニングの標準プロトコルを規定しようとする試みもいくつかあったが、そのどれも完全な相互運用性を実現するには至らなかった。アイデンティティ統合における最も深刻な問題点は、明らかにスキーマである。どのアプリケーションも、アカウント、グループ、特権など、アイデンティティ関連のオブジェクトを表現する独自のデータモデルをもっている。アプリケーションが何らかの標準スキーマを使用してそのデータモデルを公開しようとしても、必ず小さな（しかし重要な）細かい点や違いがある。midPoint はこの問題に効果的なソリューションを提供する。アプリケーションのインターフェイスおよびスキーマは、デプロイのために使用しようと選んだ一般的なアイデンティティスキーマに合わせるか、*マップされる*必要がある。インターフェイスおよびスキーマのマッピングに数分もかからないような優れたアプリケーションもあれば、数日または数週間というハードな作業を要するような手に負えないアプリケーションもある。しかし何らかの統合および構成作業は必ず必要だ。そして本章ではその方法を説明する。

リソースの定義

*リソース*は midPoint の最も重要な概念の一つである。リソースは midPoint に接続されるシステムである。典型的なリソースといえば、midPoint がアカウントを管理するターゲットシステムである。しかし人事データベースのようなソースシステムもリソースとして定義される。midPoint においては、ソースリソースとターゲットリソースとでは厳密な区別はない。ソースリソースとターゲットリソースはどちらも同様に定義されている。リソースは一度にソースとターゲットの両方として機能することさえできる。さらに同一の属性に対しソースおよびターゲットのどちらにもなりうる。これは midPoint 独自の相対変化モデルによるものである（以下参照）。

midPoint に接続されたシステムはみなリソースである。midPoint はリソースとの通信方

法を必要とする。midPoint は通信プロトコル、ホスト名、パスワード等を把握しなければならないため、*リソース定義オブジェクト*をもっている。これは midPoint レポジトリに格納される通常の midPoint 構成オブジェクトである。リソース定義にはだいたい以下のものが含まれる。

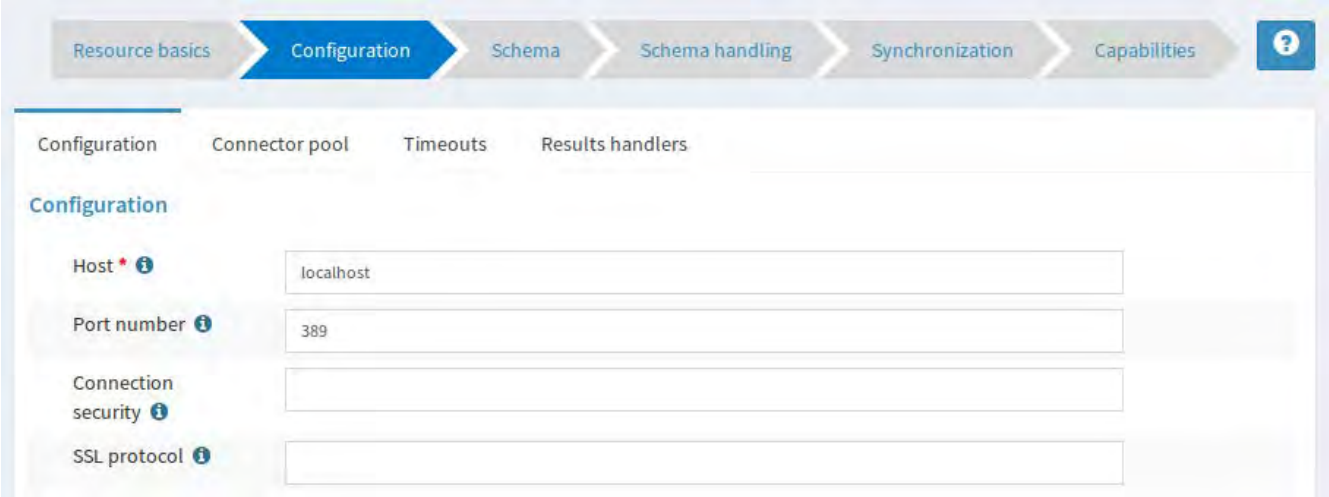
- リソースの**名前**および概要
- リソースとの通信に使用する**コネクタへの参照**
- ホスト名、ポート、通信設定等を定義する**コネクタ構成**プロパティ。コネクタの初期化に使用する。
- midPoint にとって重要な**オブジェクトタイプ** の定義。通常は典型的なアカウントがどう見えるかを説明した定義である。ただしグループ、ロール、組織単位など、もっと色々ある場合もある。
- オブジェクトタイプの定義には決まって**マッピング**が含まれている。マッピングでは、midPoint からリソース（ターゲットリソース）へ、またはリソースから midPoint（ソースリソース）へ属性を同期させる方法を定義している。
- 不明アカウントの検出時やリソースからのアカウント削除時等に、midPoint は何をすべきかを定義した**同期**設定。

リソース定義を XML 形式で表示（簡素化したもの）するとこのようになる。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
	<name>LDAP</name>
	<connectorRef oid="028159cc-f976-457f-be70-9e9fa079bcf7"/>
	<connectorConfiguration>
		<configurationProperties>
			<port>389</port>
			<host>localhost</host>
			<baseContext>dc=example,dc=com</baseContext>
			...
		</configurationProperties>
	</connectorConfiguration>
	<schemaHandling>
		<objectType>
			<kind>account</kind>
			<default>true</default>
			<objectClass>ri:inetOrgPerson</objectClass>
			...
		</objectType>
	</schemaHandling>
</resource>
```

リソース定義は非常にリッチ（かつ強力）な構成オブジェクトである。おそらく midPoint のシステム全体で最もリッチな構成オブジェクトであろう。そのため一からリソース定義を作成することは、一般には簡単なタスクではない。コネクタ構成、識別子の規則、必須属性、属性値フォーマット等、考慮すべきこともたくさんある。そこで一般によく行われることは、同じコネクタを使用しているリソース定義サンプルを見つけて、自分たちのニーズに合うよう変更を加えることである。ただ、これを実行するにはリソース定義がどう機能するかを理解する必要がある。次セクション以降でそのことについて説明しよう。

XML/JSON/YAML が苦手な人や本当に一からリソースを作成したい人向けには、midPoint ユーザーインターフェイスのリソースウィザードがある。このウィザードを使用すれば、グラフィカルユーザーインターフェイスを使用してリソースの作成および編集ができる。



ただしリソースウィザードを使うことを好む人の場合でも、やはり既存のサンプルから始めたほうが簡単であろう。自分の状況に一番合ったサンプルを見つけ、midPoint にインポートしてからウィザードを使用して内容を変更するとよい。

土台となるリソースサンプルはたくさんあり、そのほとんどは midPoint 配布パッケージに入っている。しかしそれ以外にもサンプルを見つけられる場所がある。追加情報の章を参照されたい。

コネクタ

どのリソースも動作するにはコネクタが必要である。コネクタは小さな Java コードであり、リソースとの通信に使用される。起動時に、midPoint は利用できるコネクタを探す。そして起動中に検出した各コネクタに対し構成オブジェクトを自動で新規作成する。検出したコネクタのリストは、midPoint ユーザーインターフェイスの Configuration（構成）> Repository objects（レポジトリオブジェクト）> Connector（コネクタ）にて確認できる。midPoint が検出するコネクタには、それぞれ 1 つずつオブジェクトがある。コネクタオブジェクトはこのようになっている（簡単に表示）。

```
<connector oid="028159cc-f976-457f-be70-9e9fa079bcf7">
        <name>ICF com.evolveum.polygon.connector.ldap.LdapConnector v1.4.2.18</name>
        <framework>http://midpoint.evolveum.com/xml/ns/public/connector/icf- 1</framework>

<connectorType>com.evolveum.polygon.connector.ldap.LdapConnector</connectorType>
        <connectorVersion>1.4.2.18</connectorVersion>
        <connectorBundle>com.evolveum.polygon.connector-ldap</connectorBundle>
        <namespace>http://midpoint.evolveum.com/xml/ns/…</namespace>
        <schema>
                ...
        </schema>
<connector>
```

リソース定義は適切なコネクタオブジェクトを指し示す必要がある。従ってコネクタリストから正しいコネクタを選択し、その OID を覚えておく。リソース構成ではコネクタ OID を以下のように使用する。

```
<resource>
        <name>LDAP</name>
        <connectorRef  oid="028159cc-f976-457f-be70-9e9fa079bcf7"/>
        ...
</resource>
```

これはコネクタおよびリソースをリンク付けする素直な方法である。ただし一番便利な方法というわけではない。midPoint はコネクタオブジェクトを自動で作成する。そのためコネクタオブジェクトの OID は固定ではない。検出されたコネクタに対し、midPoint のインスタンスはそれぞれ違う OID をもっている。よって、すべての midPoint インスタンスで必ず LDAP コネクタを使用するリソースが必要であれば、前述の OID を使用するだけでは無理である。しかし別の方法がある。固定 OID ではなく、検索フィルタを使用すればよい。

```
<resource>
        <name>LDAP</name>
        <connectorRef type="ConnectorType">
                <filter>
                        <q:equal>
                                <q:path>connectorType</q:path>

<q:value>com.evolveum.polygon.connector.ldap.LdapConnector</q:value>
                        </q:equal>
                        </filter>
        </connectorRef>
        ...
</resource>
```

検索フィルタの詳しい内容については後ほど説明しよう。今はとにかく、いくつかの基本原則を知ることが大切である。このリソース定義がインポートされると、midPoint は connectorRef 参照に OID がないことに気付く。そして検索フィルタがあることに気付く。そこで midPoint はその検索フィルタを実行する。この場合、値が com.evolveum.polygon.connector.ldap.LdapConnector であるプロパティ ConnectorType をもつ ConnectorType タイプのオブジェクトが検索される。そこで midPoint は LDAP コネクタの検出時に生成された OID に関係なく、LDAP コネクタを見つけ出す。そして見つけ出したオブジェクトの OID を取得する。この OID は connectorRef 参照に置かれるため、midPoint は直接このコネクタを見つけるようになり、リソース使用時に毎回検索を実行する必要がなくなる。

リソース定義を所定のコネクタタイプに結びつけるにはほぼ必ずこの方法が使用される。この方法なら、どの midPoint デプロイでもうまくいくというメリットがあるため、すべてのサンプルにおいても使用されている。

同梱コネクタおよびデプロイしたコネクタ

リソースはクラスごとに独自のコネクタが必要である。すべての一般的な LDAP サーバー

に対応するLDAPコネクタがある。汎用データベーステーブルと連携するコネクタがある。これらのコネクタはまったく一般的なものである。しかしコネクタの多くはSAP R/3、LifeRayポータル、Drupal など、特定のアプリケーションもしくはソフトウェアシステムを対象に作られたものである。

わずかながら、ほぼすべての midPoint デプロイで使用される非常に一般的なコネクタがある。こうしたコネクタは midPoint にバンドル（同梱）されている。つまり、これらのコネクタは midPoint アプリケーションの一部であり、常に利用可能ということである。その midPoint の一部となるコネクタバンドルが次の 3 つである。

- LDAP コネクタバンドル。以下のものが含まれる。
  - **LDAP コネクタ**。LDAPv3 に適合するサーバーと連携する。
  - **アクティブディレクトリ**コネクタ。LDAP プロトコル上で Microsoft の Active Directory と連携できる。
  - **e ディレクトリ** コネクタ。e ディレクトリ LDAP 実装の特殊性に対応するため変更が加えられた LDAP コネクタである。
- 一般的な関係データベーステーブルに接続できるコネクタ付きの**データベーステーブル** コネクタバンドル
- カンマ区切り（CSV）テキストファイルと連携するコネクタ付きの **CSV** コネクタバンドル

これらのコネクタは midPoint で常に利用できる。それ以外のコネクタは midPoint にデプロイしなければならない。デプロイプロセスは実に簡単である。

1. コネクタバイナリ（JAR ファイル）を探す。
2. そのバイナリを、midPoint ホームディレクトリに置かれている icf-connectors ディレクトリにコピーする。
3. midPoint を再起動する。

起動後、midPoint は icf-connectors ディレクトリをスキャンする。新しいコネクタを検出し、それらのコネクタ構成オブジェクトを作成する。

コネクタ構成プロパティ

コネクタがリソースと連携できるようになるには構成が必要である。この構成はだいたいはホスト名、ポート、管理者ユーザー名、パスワード、接続セキュリティ設定といった接続パラメータで構成される。接続構成プロパティはリソース定義オブジェクトにて指定される。簡易表記ではこのようになる。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
    <name>LDAP</name>
    <connectorRef oid="028159cc-f976-457f-be70-9e9fa079bcf7"/>
    <connectorConfiguration>
        <configurationProperties>
            <port>389</port>
            <host>localhost</host>
```

```
            <baseContext>dc=example,dc=com</baseContext>
            ...
        </configurationProperties>
    </connectorConfiguration>
    ...
</resource>
```

非常に幅広い構成プロパティがあり、コネクタにはそれぞれ固有のセットがある。リソース定義のテキスト表現を処理しつつ、同時にサンプル、コネクタ文書、あるいはコネクタのソースコードさえ調べて構成プロパティの名前を見つける必要があるだろう。これは難しそうに見えるかもしれないが、見事に実行可能な手法である。しかし他にも方法がある。まず第一に、リソースウィザードである。このウィザードはコネクタ構成プロパティをすべて知っており、構成形式でそのプロパティを表示する。そして*コネクタスキーマ*から構成プロパティの定義を取得する。コネクタスキーマはそのコネクタが対応するプロパティ、すなわち名前、タイプ、多重度などを定義したものである。コネクタスキーマは schema タグの下のコネクタ構成オブジェクトに格納される。そのためたとえ XML/JSON/YAML ファイルのみで作業しているとしても、そのスキーマを参照してどのコネクタ構成プロパティが対応しているかを把握できる。

コネクタスキーマはコネクタの名前空間も定義する。一般に midPoint の名前空間は、競合するスキーマ拡張を分離するために使用される。また、データモデルのバージョン管理にも使用される。名前空間の使用は midPoint のほぼすべての部分で省略できるが、まだすべての部分とまでは至っていない。これをまだ必要とする部分はわずかながらあり、その一つがコネクタ構成である。ただこれは有意義なことでもある。というのも、ここでは名前空間が安全性を高める対策としても使用されているからだ。端的に言えば、構成プロパティは適切に名前空間で修飾されているべきである。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
    <name>LDAP</name>
    <connectorRef oid="028159cc-f976-457f-be70-9e9fa079bcf7"/>
    <connectorConfiguration
        xmlns:icfc="http://midpoint.evolveum.com/xml/ns/public/connector/icf-
1/connector-schema-3"
        xmlns:icfcldap="http://midpoint.evolveum.com/xml/ns/public/connector/icf-
1/bundle/com.evolveum.polygon.connector-
ldap/com.evolveum.polygon.connector.ldap.LdapConnector">
<icfc:configurationProperties>
<icfcldap:port>389</icfcldap:port>
<icfcldap:host>localhost</icfcldap:host>
<icfcldap:baseContext>dc=example,dc=com</icfcldap:baseContext>
...
</icfc:configurationProperties>
</connectorConfiguration>
...
</resource>
```

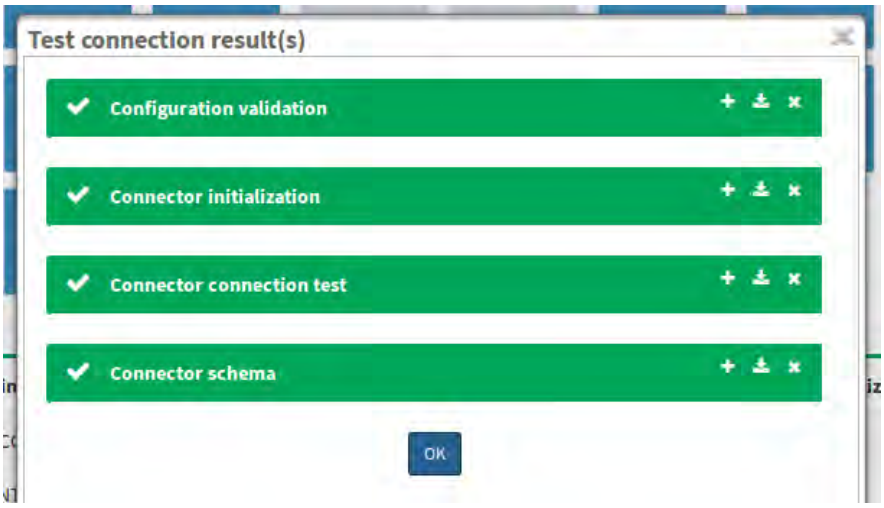
midPoint ののちのバージョンでは名前空間の使用を完全に省略できるようになるだろう。今はただ、サンプルから名前空間の URI をコピーすればよい。何が起こっているかを完全に理解する必要はない。ただ一つだけ、構成プロパティの名前空間はコネクタオブジェクトに定義された名前空間と同じでなくてはならない。これはコネクタのバンドル名とコネクタ名から成る長い URI である。例：

http://midpoint.evolveum.com/xml/ns/public/connector/icf-1/bundle/com.evolveum.polygon.connector-ldap/com.evolveum.polygon.connector.ldap.LdapConnector

この名前空間が一致していないと、コネクタは動作を拒否する。これは同じ名前だがまったく意味が違う構成プロパティの別のコネクタにおいて、あるコネクタの構成を偶発的に使用してしまうのを防ぐ、安全対策である。

リソースのテスト

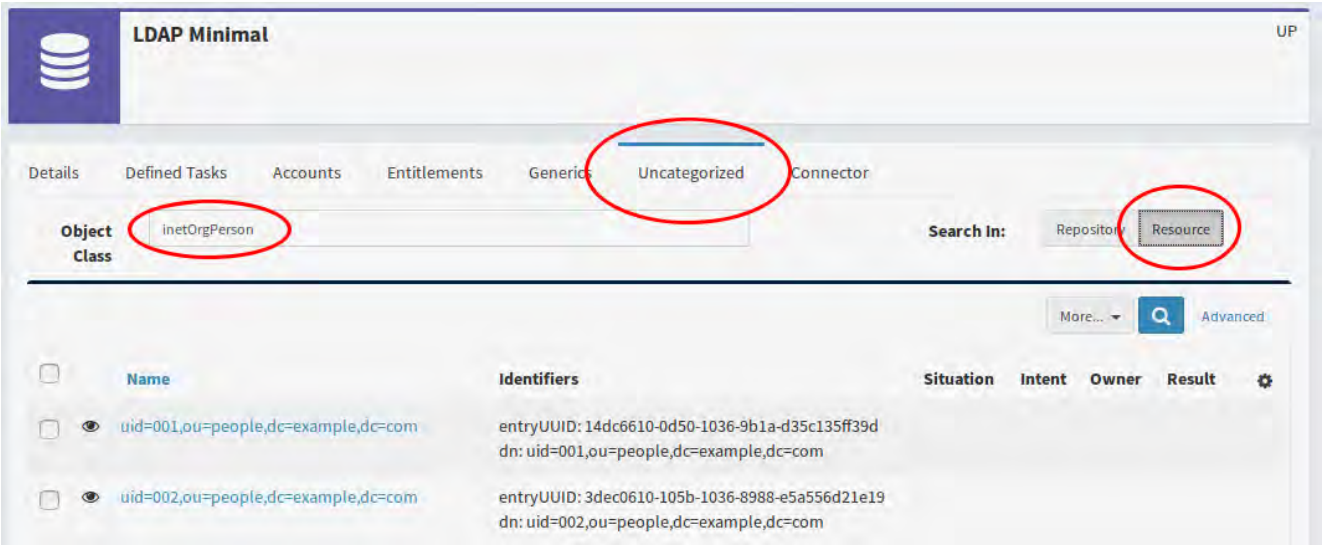
最小限のリソース定義には、名前、コネクタ参照、コネクタ構成プロパティがあるだけである。このテストを行えば、リソースは生きている兆候を示すはずである。そこで今から適切なサンプルファイルを選ぶ。そこから最低限のものだけを残し、パラメータを変更したうえでリソースを midPoint にインポートする。これで *Resources（リソース）* > *List resources（リストリソース）*に遷移すると、リスト上にそのリソースが表示されているはずだ。そのリソースの隣のアイコンは十中八九、緑でもなければ赤でもなく、黒である。緑色のアイコンはリソースが稼働していることを示す。赤色はエラーがあることを、黒色は「まだ分かっていない」ことを示す。ここでリソースラベルをクリックし、リソースの詳細ページを開く。ページの下部に［*Test Connection （テスト接続）*］ボタンがあるので、これをクリックする。しばらく時間がかかるかもしれない。midPoint は指定されたパラメータでコネクタを初期化中である。このコネクタを使用してリソースへの接続がチェックされる。パラメータが正しければ midPoint はリソースにアクセスでき、緑色が表示される。



コネクタの初期化中、パラメータ、またはネットワーク接続にエラーがあった場合は、ここでエラーが表示される。この場合は構成を修正し、再度実行する。すべてがうまくいくとリソースアイコンが緑色に変わる。これで最低限稼働するリソースを得たことになる。

このような最低限のリソースでもできることはもっとある。例えばリソースのコンテンツを確認できる。リソースの詳細ページへ遷移したら［*Uncategorized（未分類）*］タブに切り替えて、リソースが対応しているオブジェクトクラスの一つを選ぶ。［*Object class（オブジェクトクラス）*］の入力ボックス内をクリックするだけで、候補が表示される。右側の［*Resource（リソース）*］ボタンをクリックする。midPoint はリソースへ移動し、指定されたオブジェクトクラスのすべてのオブジェクトを一覧にし、そのリストを表示す

る。これでオブジェクトをクリックすればその詳細を閲覧できる。



これは非常に役立つ機能であるが、その理由は次のとおりだ。まず 1 つ目は、リソース接続が動作しているかだけでなく、実際にコネクタからオブジェクトが見えるかということも確認できることだ。2 つ目は、リソースが対応しているオブジェクトクラスについて何らかの考えを得られることである。そして 3 つ目は、いくつかのオブジェクトを見ることで、データがどんな構造になっているか、使用されている属性は何か、典型的な値は何かといった基本的な概略が得られることである。あとでマッピングを設定する段階になったときに、こうした情報をありがたいと思うだろう。

リソーススキーマの基本

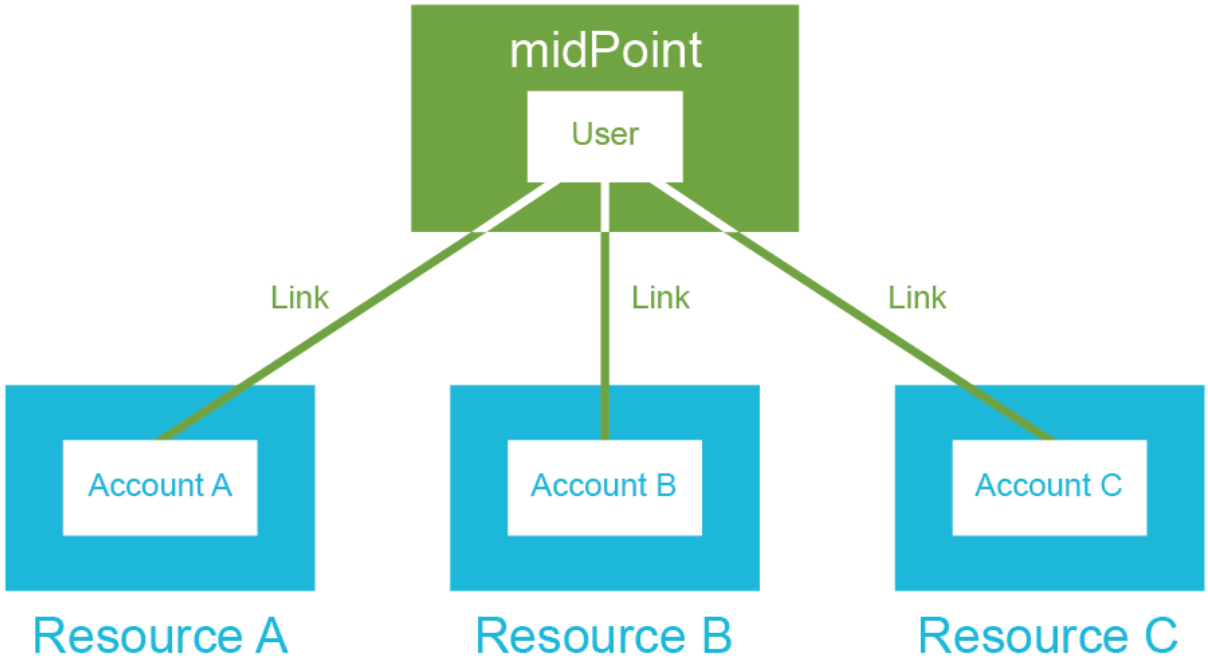
従来のアイデンティティ管理システムが扱っていた唯一のリソースオブジェクトは*アカウント*であった。しかしもはやそれでは十分ではない。良好なアイデンティティ管理システムならば、アカウント、グループ、組織単位、特権、ロール、アクセス制御リストなど、あらゆるタイプのリソースオブジェクトを管理する必要がある。midPoint には*オブジェクトクラス*がある。コネクタによって midPoint にアクセスできるようになるリソースオブジェクトのタイプのことである。典型的なリソースは少なくとも*アカウント*のオブジェクトクラスに対応しており、さらに多くのオブジェクトクラスにも対応するかもしれない。各オブジェクトクラスは全く異なる属性をもつ。名前もタイプも異なり、必須のものもあれば任意のものもあるだろう。

オブジェクトクラスおよびそれらの属性をまとめた定義が、いわゆる*リソーススキーマ*である。当然ながらリソーススキーマはリソースごとに異なっている。同じコネクタを使用しているリソースどうしでさえ、リソーススキーマは違う場合がある（例えばカスタムスキーマ拡張が異なる 2 つの LDAP サーバーなど）。midPoint は賢いシステムであり、リソーススキーマを自動検出できる。リソースの初回使用時には、midPoint がリソースにアクセスしてスキーマを取得する。取得したリソーススキーマはリソース定義オブジェクト内の schema タグの下に格納される。ここでスキーマを確認、調べることができる。ただし気を付けなければならないことは、スキーマはかなりリッチで大きいかもしれないことだ。

リソーススキーマは極めて重要である。midPoint はアカウントやグループなどリソースオブジェクトの操作が必要となるたびにリソーススキーマを使用するだろう。マッピングの検証にもリソーススキーマを使用するだろう。スキーマはタイプの自動変換に使用されるだろう。そしてすべてにおいて最も重要なのは、リソーススキーマを使用してユーザーインターフェイスにリソースオブジェクトが表示されるだろうことである。midPoint は自動でリソーススキーマに適応する。そのためのカスタムコードは 1 行たりとも必要ない。

ハブアンドスポーク方式

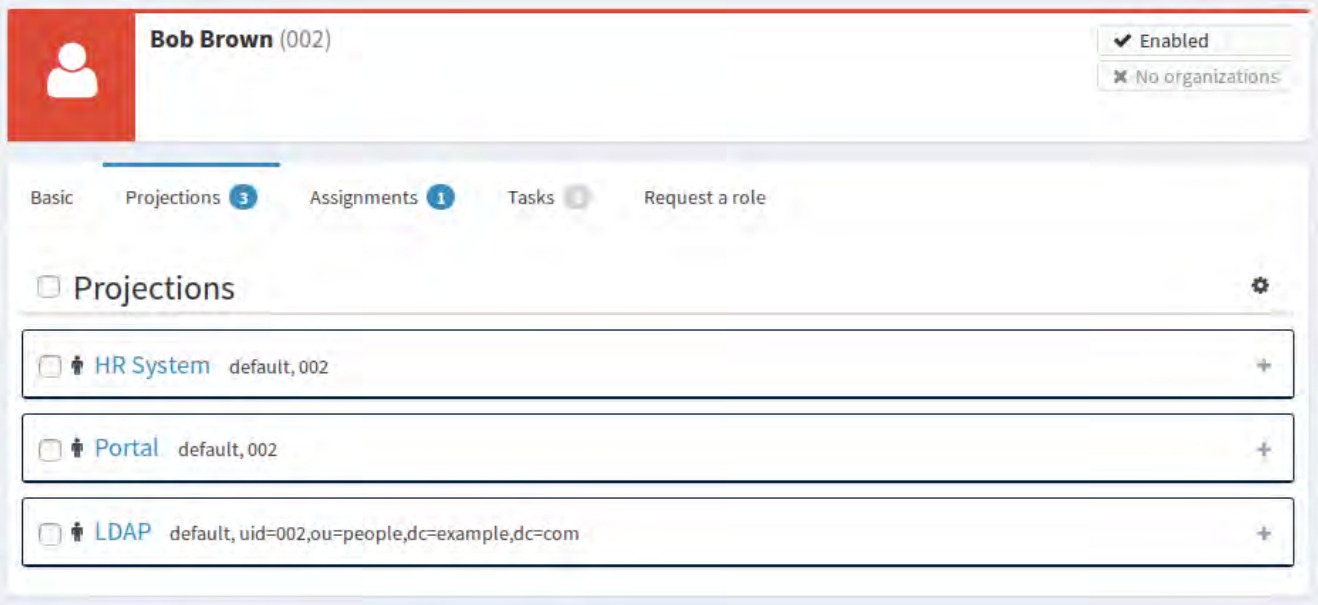
midPoint のトポロジーは midPoint を中心に置いたスター型（別名「ハブアンドスポーク方式」）である。これは物理的にも論理的にも当てはまるトポロジーである。



つまり、アカウント A は midPoint *ユーザー*と同期させることができ、さらに midPoint *ユーザー*は *アカウント B* と同期させることができる。しかし *アカウント A をアカウント B* に直接同期させることはできない。こうしようと決めたのは、midPoint 設計のかなり早い段階で十分熟慮した上でのことである。そしてそう決めたのは実にもっともな理由がある。ただ、心配しなくてよい。各リソースで些細な操作を行うごとに一つ一つ同期プロセスを設定する必要はない。midPoint はそれを対処できるくらい十分賢いのだ。あなたは同期ポリシーを設定するだけでよい。midPoint がその実行に対応する。リソースは midPoint に同期させることができる、ただし 2 つのリソースどうしを直接同期させることはできないことだけは覚えておいてほしい。

同一人物を表わす *アカウント*と *ユーザー*は相互にリンクされている。この *リンク*は midPoint が作成、保守する関係である。よって midPoint はある特定のアカウントのオーナーが誰であるかを知っている。また、ユーザーがどのアカウントを持っているかも把握している。だから midPoint は、どのアカウントをどのユーザーに同期させる必要があるかを知っているのである。リンクは必ず正しくなければならない。さもなくば midPoint はデータを

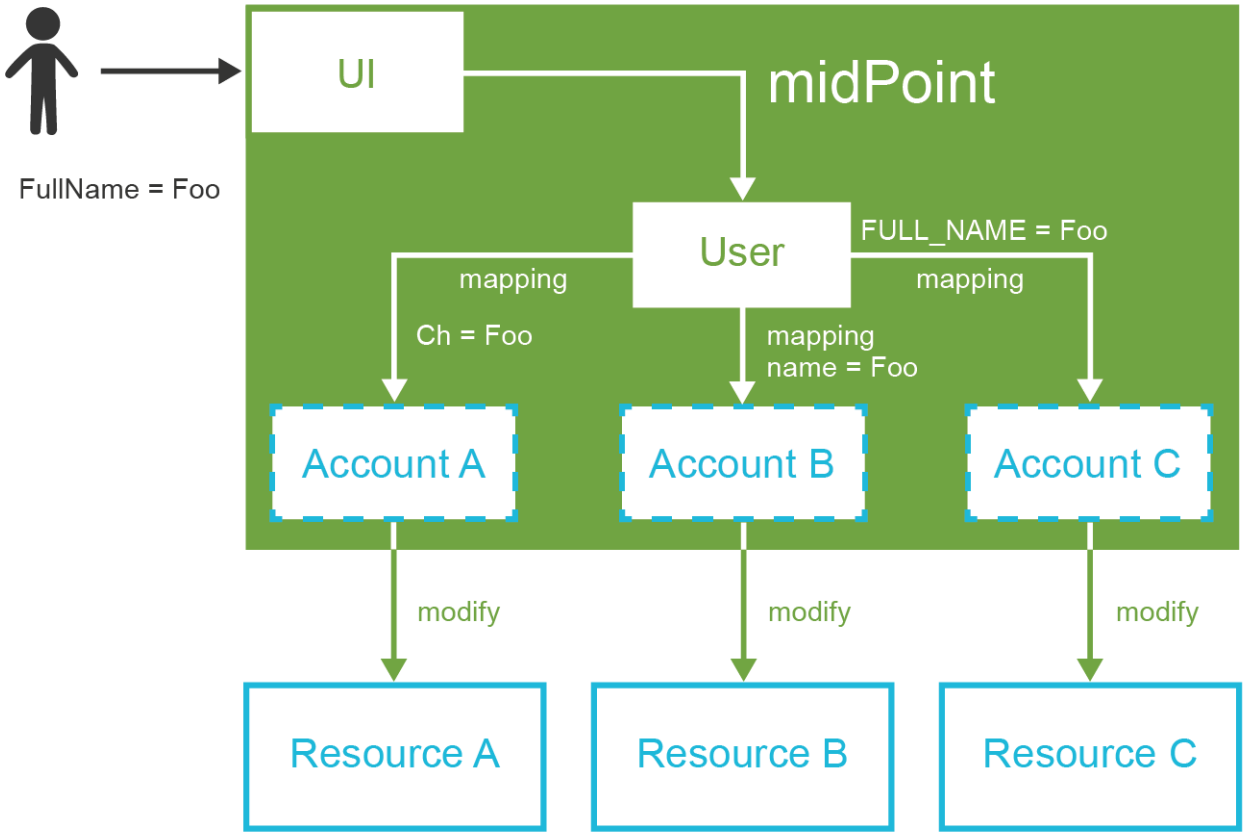
確実に同期させることができないからだ。そのため midPoint はリンクの保守に十分注意を払っているが、これは必ずしも簡単ではない。というのも、名前が変更されたアカウント、誤って削除され再度作成されたアカウントなど、稀にしか発生しないが厄介なケース（コーナーケース）があるからだ。しかし midPoint はこのようなケースにも対応できるようになっており、リンクは保守される。そしてリンクにより、midPoint はユーザーインターフェイスにユーザーのすべてのアカウントを一覧表示できる。



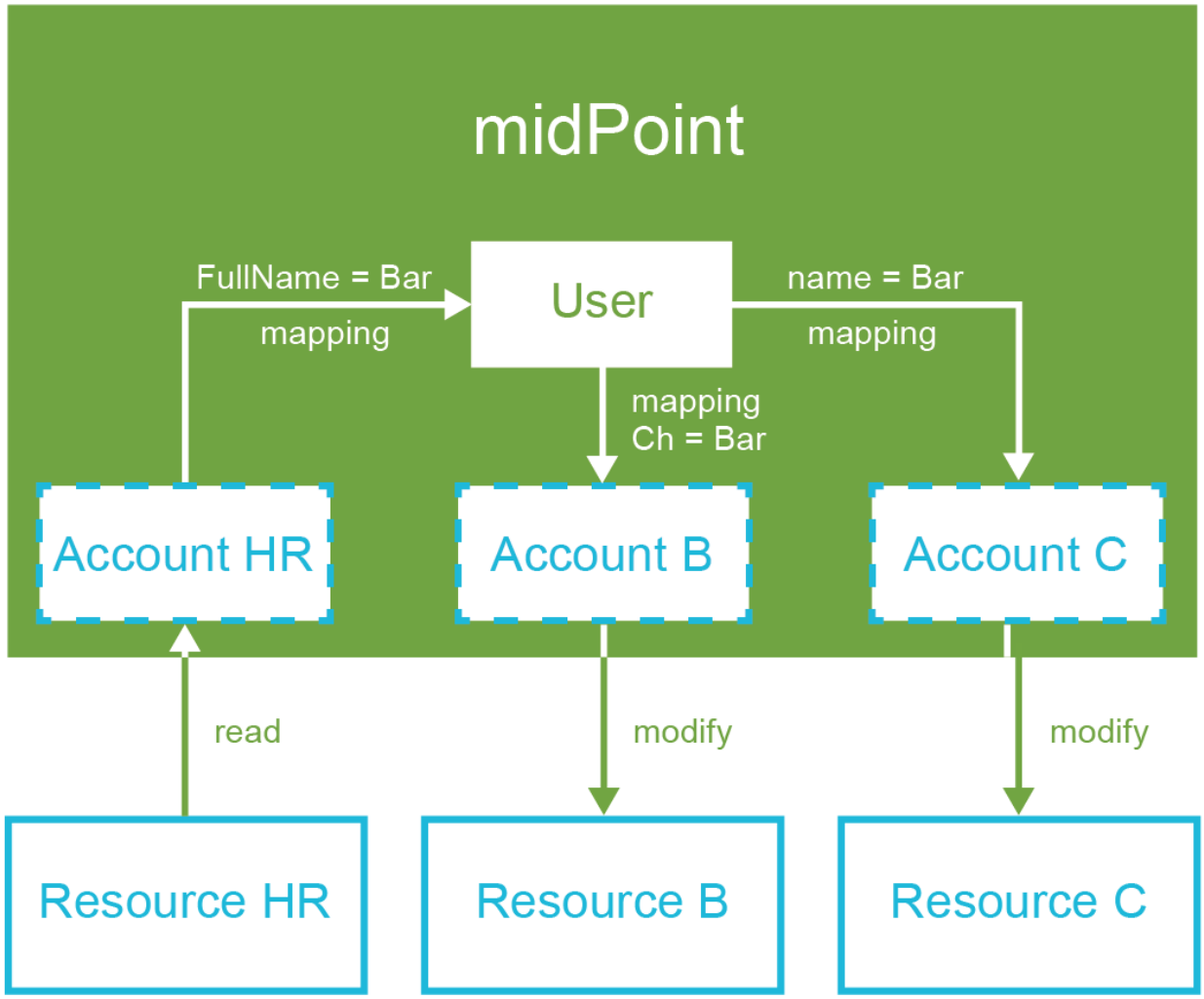
midPoint のユーザーは、midPoint 用語で *フォーカス（焦点）*と呼ばれる。そしてアカウントは *プロジェクション（投影）*と呼ばれる。スクリーン上に投影を作成するため焦点から多量の光線を送る投光装置を想像できるだろうか。これは midPoint 構築時に我々が選んだ比喩である。そしてより適切な表現がなかったために、この用語が今もそのまま残って使用されている。さて、本書で頻繁に出てくるフォーカスとプロジェクションの概念に話を戻そう。とりあえず今は、*プロジェクション*とは*アカウント*であることを覚えておく必要がある。

midPointは自身が保守しているリンクをたどることにより、どのアカウントがどのユーザーに属しているかを知る。しかし同期させる属性がどれかはどうやって分かるのだろうか？値はどうやって変換する？どちら側が信頼できるのか？これを扱っているのがマッピングである。マッピングは柔軟性の高いデータレプリケーションレシピのようなものである。midPoint により、任意の方向で属性ごとにマッピングを定義できる。マッピングは非常に繊細な粒度レベルで同期を制御するのに使用される。

同期の原則を手短に説明するなら、サンプルを使用するのがおそらく最良の方法であろう。1 つ目の例は、midPoint ユーザーインターフェイスのユーザープロパティの変更である。［Save（保存）］ボタンを押すと midPoint ユーザーインターフェイスは midPoint コアに変更内容を送信する。midPoint コアの同期コードは、この特定のユーザーに属しているすべてのアカウントを見つけるため、リンクをたどる。そしてマッピングが適用され、変更されたユーザープロパティをそれらのアカウントに同期させる。アカウントの変更はリソースに伝達され、ユーザー変更は midPoint レポジトリに格納される。



2つ目の例は若干違う。このケースではアカウントデータの変更がトリガーになっている。これは人事システムの従業員レコードの変更かもしれない。midPoint はその変更を検出し新アカウントを読み込む。そしてリンクを使用してそのアカウントが属するユーザーを見つけ出す。さらに別のリンクも使用して関連するであろう他のアカウントもすべて見つけ出す。1つ目のケースと同様にマッピングが適用される。まず人事アカウントからユーザーへマッピングが適用される。その結果ユーザープロパティが変更される。その後、1つ目のケースと同様のプロセスが行われる。ユーザー変更は関連するすべてのアカウントに自動で適用される。



これら２つのケースは midPoint にとってはまったく異なるユーザーケースである。1 つ目のケースはシステム管理者による手動でのデータ変更である。２つ目のケースは人事システムからの自動データフィードである。しかしご覧の通り、その実行に使用された原則はほとんど同じである。これは midPoint の基本的な姿勢、つまり機能再利用の徹底、包括的に原則を適用した結果である。あなたはただ、どうしたいか（ポリシー）を定義すればよいだけだ。midPoint は必要に応じてそれが実行されるよう対処する。

情報：なぜスター型トポロジーなのか？スター型または「ハブアンドスポーク方式」はシステム統合を示す実に流行りの専門用語であった（そして今もそうである）。その基本的な考えは非常に理にかなっている。一つ一つのノードを他のすべてのノードと同期させようとすれば、必要な接続数は一気に増えてしまう。実際に接続数はノード数の二乗に比例する。数学者によれば、それは計算量 $O(n^2)$だという。しかしこの接続がすべて中心部の「ハブ」に向かうように並べなおせば、接続数は大幅に減る。接続数はノード数に比例、すなわち計算量は O(n)となる。特にリソースが多いデプロイとなれば、これは大きく違ってくる。ただしこの手法は、スター型トポロジーが物理的かつ論理的であるときのみうまくいく。すなわち、中心部である「ハブ」がデータ同期において内部的になおも $O(n^2)$ポリシーを必要としている場合は、すべてのリソースをこのハブに接続したとしてもほとん

ど意味がないのだ。そうなるとその複雑さはブラックボックスに隠すしかなかったのだが、これでは真に解決したことにはならなかった。しかし midPoint は違う。midPoint は真の「ハブ」である。だからこそ、midPoint はアカウントどうしが直接同期しあうことに対応していないのだ。我々は外部的にも内部的にもシンプルかつクリーンで保守できるシステムを追求している。

スキーマ操作

リソーススキーマは非常に重要な概念であり、リソースが対応するオブジェクトクラスは何か、そしてそれらはどう表示されるかを定義するものである。これについてはすでに説明した。しかし重要なのはオブジェクトがどう表示されるかを知ることだけではない。オブジェクトで何をするかを知ることも重要である。そして*スキーマ操作*とはこのことにほかならない。

*スキーマ操作*とはリソース定義の一部であり、midPoint が実際に使用するのはリソーススキーマのどのオブジェクトクラスであるかを指定する。そして中でも一番重要なことは、オブジェクトクラスをどう使用するかを指定する点である。これはマッピングを格納する場所である。アカウントとグループの関連付けを定義する場所である。スキーマを拡張、微調整できる場所である。簡単に言えば、リソースに関する構成の大半が行われる場所ということである。

スキーマ操作セクションにはいくつかの*オブジェクトタイプ*の定義が含まれている。各*オブジェクトタイプ*とは midPoint が扱う一つの「もの」、たとえばデフォルトアカウント、テストアカウント、グループ、組織単位等である。まず簡単なものから始めてみよう。ここではオブジェクトタイプを一つだけ、デフォルトアカウントを定義してみよう。以下のようになる。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                </objectType>
        </schemaHandling>
</resource>
```

これは些細なものに見えるかもしれないが、このような最小限の定義さえ midPoint には重要である。この定義は midPoint に、このリソースのデフォルトアカウントは inetOrgPerson オブジェクトクラスを持っていることを教えてくれている。LDAP サーバーのようなリソースは多数のオブジェクトクラスをもっている。それらの大半は一切使用されない。しばしば、アカウント作成に使用できる代わりのオブジェクトクラスが複数ある。よってどのオブジェクトが正しいものかを midPoint に伝えることが重要である。そしてそれを行うのがこの定義である。この定義が定められると、リソース詳細ページの［*Accounts（アカウント）*］タブにアカウントが表示されようになる（それ以前は［*Generics（ジェネリクス）*］タブにのみ表示されていた）。これはその定義が正しく動作していることを表している。

聡明な読者なら上記の例の中で *kind（カインド）*の定義に気づくだろう。アカウントに*カインド*を設定することは、このオブジェクトタイプ定義が（驚くことに）アカウントを表していることを示している。midPoint はあらゆるタイプのオブジェクトに対応している。しかし特別な役割をもつタイプが 2 つある。一つはユーザーを表わす*アカウント*であり、もう一つはアカウントに特権を付与する*エンタイトルメント*である。タイプがアカウントかエンタイトルメントかのどちらかであると分かっていれば、midPoint はそのオブジェクトを賢く取り扱うことができる。そしてこれを教えてくれるのがまさに*カインド*定義である。また、サブタイプの定義に使用できる *intent（インテント）*という任意設定もある。詳しくは後述する。

スキーマ操作セクションを使用してリソーススキーマの一部を拡張（または上書きまで）することもできる。例えば以下の例では、このオブジェクトタイプの表示名を設定している。表示名はアカウントを表示する際にユーザーインターフェイスで使用される。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <b>&lt;displayName&gt;Default account&lt;/displayName&gt;</b>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                </objectType>
        </schemaHandling>
</resource>
```

しかし、スキーマ操作で使用される最も強力な機能は、属性に対処する能力である。次の数セクションでは、この能力について説明する。

属性の操作

アカウントやグループ等のリソースオブジェクトはほとんどがまさに属性の束である。IDM マジックはほぼすべて、正しい属性に正しい値を設定することである。リソース定義のスキーマ操作セクションは、基本属性の挙動を定義する場所である。

オブジェクトタイプ定義には、我々が気にかけている各属性の挙動を定義するセクションが含まれている。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                        <b>&lt;attribute&gt;</b>
                                <b>&lt;ref&gt;ri:dn&lt;/ref&gt;</b>
                                <b>&lt;!-- behavior of "dn" attribute defined here --&gt;</b>
                        <b>&lt;/attribute&gt;</b>
                        <b>&lt;attribute&gt;</b>
                                <b>&lt;ref&gt;ri:cn&lt;/ref&gt;</b>
                                <b>&lt;!-- behavior of "cn" attribute defined here --&gt;</b>
```

```
                <b></attribute></b>
                ...
            </objectType>
        </schemaHandling>
</resource>
```

我々が必要とする属性の一つ一つに attribute セクションがある。ここではあらゆる詳細を定義できる。たとえば属性の表示名（ユーザーインターフェイスが使用する名前）、制約事項、スキーマ拡張設定および上書き設定などだ。しかしここで一番重要となるのがマッピングである。一番シンプルな形のマッピングはこのようになる。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                        ...
                        <attribute>
                                <ref>ri:cn</ref>
                                <b><outbound>
                                        <source>
                                                <path>$focus/fullName</path>
                                        </source>
                                <outbound></b>
                        </attribute>
                        ...
                </objectType>
        </schemaHandling>
</resource>
```

これは、cn 属性の値がフォーカスオブジェクト（ユーザーであることが多い）の fullName プロパティから取得されることを意味している。これは非常にシンプルなマッピングであり、値変換もなければ条件もない。つまり複雑なものは一切ない。これは多くのマッピングがどのようになっているか、ということである。しかし、マッピングは非常に強力で複雑になりうるものである。これについては次セクションで説明しよう。

属性セクションは、リソース上の一般的なユーザーアカウントが持つ属性の設定に使用されることが多い。つまり識別子をアサインし、フルネームを設定し、説明属性および電話番号属性といったものを設定する。これを直接リソース定義内に持つことは非常に便利な方法である。もしそうすれば、細かいことは一切指定せずとも簡単にユーザーにアカウントを付与できる。midPoint は当セクションでマッピングを使用してアカウント属性に正しい値を取り込む。現実世界のリソース定義がどのようになるか感触をつかむため、今ここで、リソース定義のサンプルをいくつか見てみるとよいだろう。サンプルは midPoint の配布パッケージに含まれているが、オンラインでも見つけることができる。詳細は追加情報の章を参照されたい。

情報：「ri」名前空間について。オブジェクトクラスや属性を参照すると、常に「ri」という名前空間接頭辞が使用されていることにお気づきだろう。厳密にはこれは正しい表記

である。オブジェクトクラスおよび属性はリソーススキーマ内に定義されており、「ri」とはこのスキーマの名前空間である。名前空間の使用は midPoint のほぼすべての部分で省略できることになっているが、サンプルでは今も「ri」名前空間が使用されている。これは懐古的な理由によるところが大きい。ちなみに「ri」とは「リソースインスタンス（resource instance）」の短縮形である。

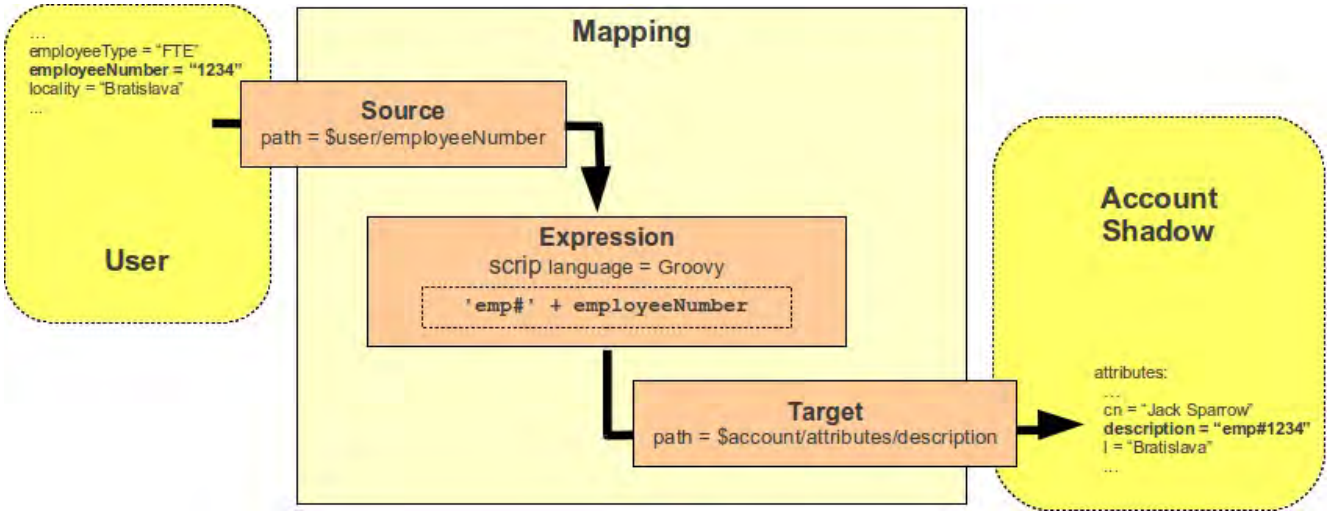
マッピング

マッピングとは一つ以上の入力プロパティを取得し、それらを変換してその結果を別のプロパティに置く、柔軟性の高い仕組みである。マッピングは midPoint のいたるところで使用される。とはいえその使用が最も重要な部分は、おそらく、アカウント属性値が設定される、リソース定義のスキーマ操作部であろう。ある場所から別の場所へ値をコピーするだけの実にシンプルなマッピングであればすでに見ているため、今こそまさにマッピングの全貌を見てみよう。

マッピングは次の 3 つの基本部分で構成される。

- **Source（ソース）**部は、マッピングのデータソースを定義する。通常これらはマッピング入力変数として理解されている。マッピングのデータ取得先を定義するのがソースである。
- **Expression（式）**部は、データの変換方法、生成方法、または「他方」への受け渡し方法を定義する。ここにはロジックが含まれているため、マッピングの中で最も柔軟な部分である。様々な可能性がある。
- **Target（ターゲット）**部は、マッピングの結果をどうすべきか、計算値はどこへいくべきかを定義する。

マッピングの基本原則と上記 3 つの部分をまとめたものが下図である。



この図は、ユーザープロパティの employeeNumber を取得し、それを Groovy スクリプト式を使用して description アカウント属性に変換するというシンプルなマッピングを示したものである。

マッピングのソース部は、ユーザープロパティの employeeNumber にもとづく単一ソー

スがあることを定義する。マッピングが相対変化（デルタ）やマッピング依存等を正しく処理するには、ソース定義が重要である。ソース定義はマッピングに対し、employeeNumberユーザープロパティの値を式に受け渡すべきであることを教える。

式部にはシンプルなGroovyスクリプトが含まれており、このスクリプトはソース定義が指定した従業員番号値に接頭辞emp#を追加する。式部は非常に柔軟性があり、値の変換、値の新規生成、固定値の使用、変更なしでの値の受け渡し等、使用できる方法が数多くある。

ターゲット部は式の結果をどう使用すべきかを定義する。このケースでは、結果はdescriptionアカウント属性として使用されることになっている。マッピングがターゲットプロパティの適切な定義を見つけ出すにはターゲット定義が必要である。そのためには、確実に式が正しいデータタイプを生成し、確実に他のスキーマ制約事項が守られていなければならない（例えば、単数値対複数値）。

このマッピング例は同じ構造を使用してXMLで記述できる。

```
<mapping>
	<source>
		<path>$focus/employeeNumber</path>
	</source>
	<expression>
		<script>
			<code>'emp#' + employeeNumber</code>
		</script>
	</expression>
	<target>
		<path>$projection/attributes/description</path>
	</target>
</mapping>
```

マッピングのすべての部分が必須というわけではない。式が存在していなければ、一切変更せずにただソースをターゲットへコピーする「*as-is（そのまま）*」式が想定される。マッピングの中には周囲のコンテキストによって暗黙のうちに定義される部分もあるかもしれない。例えば、マッピングを使用してスキーマ操作セクションに属性挙動を定義する場合は、ターゲットまたはソースが暗に示される。よってこのようなマッピングではソースまたはターゲットのどちらかを定義すれば十分であることが多い。

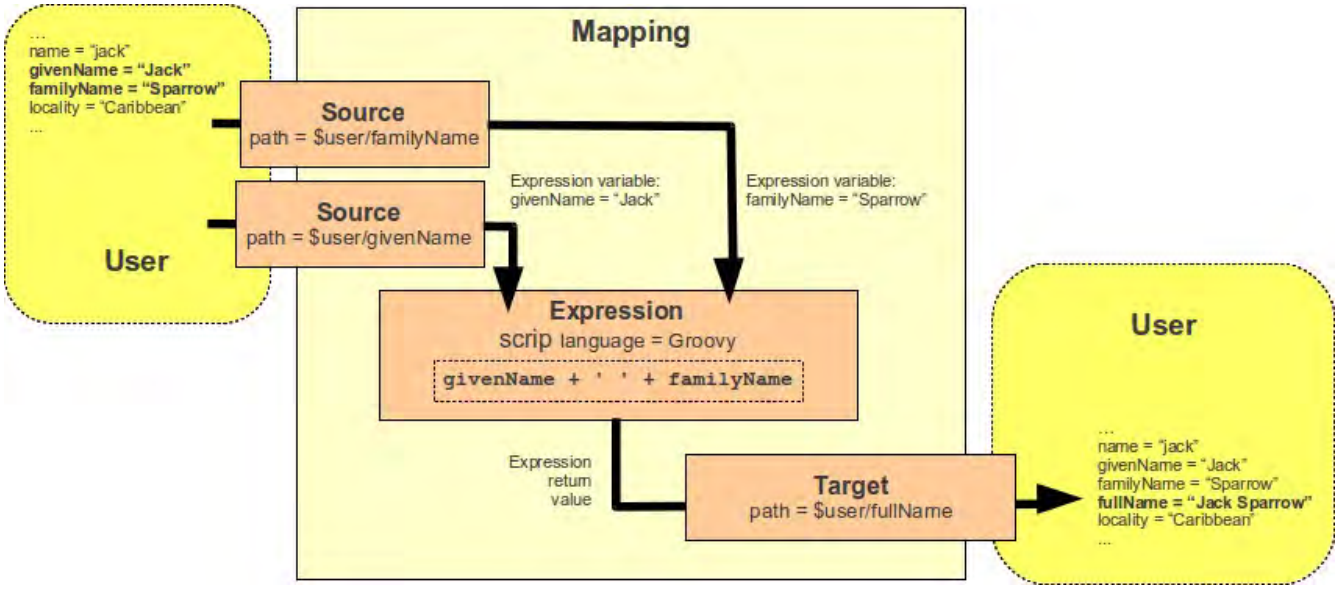
```
<attribute>
	<ref>ri:sn</ref>
	<outbound>
		<source>
			<path>$focus/familyName</path>
		</source>
	</outbound>
</attribute>
```

これは前のセクションで見たものをほぼ正確に表記したものである。マッピングソースはユーザーのfamilyNameプロパティとして明示的に指定されている。マッピングターゲットは、マッピング定義の対象となる属性であることが暗黙的に設定されている。そして明示的に定義されている式はないため、変換なしのシンプルな値のコピーであることがデフォルト設定される。

このケースでは、マッピング表記をさらにもう少し短くできる。マッピングソースがフォーカスオブジェクト（ユーザー）のプロパティの一つであることは明らかなため、接頭辞の $focus は省略してもよい。

```
<attribute>
        <ref>ri:sn</ref>
        <outbound>
                <source>
                        <path>familyName</path>
                </source>
        </outbound>
</attribute>
```

これらは非常にシンプルな例である。のちほど説明するが、マッピングはもっと色々なことができる。しかし今ここで説明しなければならないことがあと一つある。マッピングが扱えるソースはたった一つだけではない。下図は 2 つの引数として名前と名字を取得するマッピングを示したものである。このマッピングでは、2 つの値を間にスペースを一つ入れてつなげることでフルネームを生成している。これは、コンポーネントからユーザーのフルネームを作成すべく頻繁に使用されるマッピングの種類である。このマッピングはシンプルに見えるかもしれないが、実は内側には洗練された仕組みがいくつか隠されている。



このマッピングを XML で表記すると次のようになる。

```
<mapping>
        <source>
                <path>givenName</path>
        </source>
        <source>
                <path>familyName</path>
        </source>
        <expression>
                <script>
                        <code>givenName + ' ' + familyName</code>
                </script>
        </expression>
        <target>
                <path>fullName</path>
```

```
        </target>
</mapping>
```

ソース定義で指定されたソースは 2 つある。ユーザープロパティの givenName および familyName である。このマッピングでは*スクリプト式*を使用してこれらの値を合体させ一つの値にしている。この合体した値はユーザーの fullName プロパティへの入力として使用される。

この例は、マッピングが洗練されていることも示している。ソースの一つが変更されるとき、または全体の再計算が要求されるときのみ、マッピングが評価される。givenName と familyName のどちらも変更がなければ、その式を再評価する必要はない。マッピングにて明示的なソース定義が必要なのは、これが一番の理由である。もしそのような定義がなければ、式をいつどうやって再評価すべきか確実に判断することは（現実的には）難しい。

情報：$user および$account の変数について。変数である$focus および$projection が midPoint 3.0 に導入されたのは*一般的な同期*の機能に対応した結果である。式の対象となるオブジェクトはユーザーまたはアカウントだけではないだろう。そこではかなり広範囲のオブジェクトが使用されるかもしれない。そこで *フォーカス*および *プロジェクション*という一般的な概念が導入され、これを反映すべく変数名が変更された。かつての変数である$user および $account は今でも使用できるが、その使用は非推奨となっている。今でも古いサンプルの一部では使用されている。

マッピングはあらゆる状況で使用されている。そして場合によっては、本当に権威的である必要がある。マッピングはターゲットに値を強要しなければならない。しかし我々はデフォルト値を提供したいだけのときもあり、そのように設定したら、マッピングはターゲット値を決して変えるべきではない。そのような理由から、マッピングはさまざまな*強度*に設定できるようになっている。各強度についての説明は下表のとおり。

強度	**説明**
weak（弱）	ターゲットに値がないときのみ適用される。強度が「弱」のマッピングは*デフォルト値*の設定に使用されることが多い。
normal （普通）	ソースプロパティに変更があるときのみマッピングを適用する。強度が「普通」のマッピングは「*最後の変更を優先する*」戦略の実現に使用されることが多い。midPoint にて値が変更されると、マッピングが適用されターゲットが変更される。ターゲットが直接変更された場合は、midPoint にて次の変更が発生するまでマッピングによる上書きはない。デフォルトで設定されているのはこの強度である。強度の指定がない場合、「*普通*」が前提となる。
strong（強）	必ず適用される。強度が「強」のマッピングは特定の値を*強制*する。

強度は strength タグを使用してどのマッピングでも指定できる。

```
<attribute>
        <ref>ri:sn</ref>
        <outbound>
                <strength>strong</strength>
```

```
            <source>
                <path>$focus/familyName</path>
            </source>
        </outbound>
</attribute>
```

マッピングの強度については次のような経験則が役立つかもしれない。それは、ポリシーを強制したければ「*強*」のマッピングを使用する。デフォルト値を設定したいだけであれば「*弱*」のマッピングを使用する。どうしたいか分からなければ「*普通*」のマッピングでおそらく問題ないだろう、というものである。

式

式はマッピングの中で最も柔軟な部分である。式には、最もシンプルな *as is（そのまま）*、*scripting expressions（スクリプト式）*から、midPoint レポジトリ中を検索する特殊用途の式まで、約数十もの実に様々な種類がある。使用する式の種類はマッピングの expression セクション内で使用されるタグによって決定する。我々はこれらを「*式評価者（expression evaluator）*」と呼んでいる。expression evaluator については midPoint Wiki にて詳しく説明している。ここでは最も頻繁に使用される数種類のみを見てみよう。

expression evaluator	タグ	説明
As is （アズイズ、そのまま）	asIs	一切変換せずに値をコピーする。
Literal（リテラル）	value	ターゲット内にリテラル値（定数値）を格納する。
Generate （ジェネレート）	generate	乱数値を生成する。
Script（スクリプト）	script	スクリプトを実行する、ターゲット内にスクリプト出力を格納する。

一番シンプルな expression evaluator は asIs である。取得したソースをターゲットへコピーするだけである。ソースが一つだけであれば明らかに機能する。デフォルトで設定されているのもこの expression evaluator である。マッピングにて式の指定がない場合はこの asIs が前提となる。

asIs はこのように使用される。

```
<attribute>
        <ref>ri:sn</ref>
        <outbound>
                <source>
                        <path>familyName</path>
                </source>
                <expression>
                        <asIs/>
                </expression>
        </outbound>
</attribute>
```

literal 式はターゲットに定数値をおくために使用される。常に同じ値を生成するため、ソースは一切不要である。以下のようになる。

```
<attribute>
```

```
        <ref>ri:o</ref>
        <outbound>
                <expression>
                        <value>ExAmPLE, Inc.</value>
                </expression>
        </outbound>
</attribute>
```

generate 式は乱数値を生成するために使用されることから、もっぱらパスワード生成に使用される。この式については、あとでクレデンシャルについて説明するときに取り上げる。

スクリプト式

最も興味深く、かつ最も柔軟性のある expression evaluator といえば間違いなく script expression evaluator であろう。これにより、任意のスクリプトコードを実行して値を変換することができる。基本原則はシンプルである。ソースプロパティからの値をスクリプト変数に格納する。スクリプトが実行され、出力を生成する。出力はターゲット内に格納される。

スクリプト式をもつマッピングはすでに見てきた。

```
<mapping>
        <source>
                <path>givenName</path>
        </source>
        <source>
                <path>familyName</path>
        </source>
        <expression>
                <script>
                        <code>givenName + ' ' + familyName</code>
                </script>
        </expression>
        <target>
                <path>fullName</path>
        </target>
</mapping>
```

ソースが 2 つある。givenName と familyName である。これらのユーザープロパティの値は、同じ名前、すなわち givenName と familyName をもつスクリプト変数に置かれる。するとスクリプトはその変数で実行することを何でも行うだろう。最終的にはスクリプトは値を返さなければならない。上記のスクリプトは Groovy 言語で記述されているため、返り値は最後に評価された式の値である。このケースでは、それがスクリプトにおいて 2 つの変数の間にスペースを一つ入れそれらを連結する唯一の式である。スクリプトの返り値は出力に置かれる。このケースでは fullName ユーザープロパティである。

値を変換してから属性に格納するためにスクリプトが使用されることが多い。非常にありふれたケースの一つが LDAP 識別名（distinguished name：DN）である。DN は uid=foobar,ou=people,dc=example,dc=com という形式の複合値である。しかしシンプルなスクリプトを使用して簡単にこのような値を生成できる。

```
<attribute>
```

```
        <ref>ri:dn</ref>
        <outbound>
                <source>
                        <path>name</path>
                </source>
                <expression>
                        <script>
                                <code>
                                        'uid=' + name + ',ou=people,dc=example,dc=com'
                                </code>
                        </script>
                <expression>
        </outbound>
</attribute>
```

（賢い読者は今ごろきっと疑わしい表情を浮かべていることだろう。たしかに、これはLDAP DN の生成方法として必ずしも正しいわけではない。後ほど訂正するので、我慢してお付き合いいただきたい）。

midPoint は次の 3 つのスクリプト言語に対応している。

- **Groovy**:これがデフォルトのスクリプト言語である。
- **JavaScript (ECMAscript)**
- **Python**

これら3つの言語は単一のmidPointデプロイであっても任意に混在させることができる。もちろんそのような状況は推奨されないが。言語 URI を使用して式ごとに言語を選ぶことができる。

```
                <expression>
                        <script>
                                <language>http://midpoint.evolveum.com/xml/ns/public/expression/
language#python</language>
                                <code>
                                        "Python is %s, name is %s" % ("king", name)
                                </code>
                        </script>
                <expression>
```

スクリプト式を書く時は、使用するテキスト形式（XML、JSON、または YAML）に正しくエスケープしなければならない文字があることに留意する。例えば、論理演算で頻繁に使用されるアンパサンド文字（&）は、XML では&のようにエスケープが必要となる。

スクリプト式はほぼ何でも行うことができる。そしてさらに多くのものがある。しかし本セクションでは最初の手がかりとしてごく基本的なことのみ説明した。スクリプト式については、本書で何度もでてくるだろう。

アクティベーション

midPoint では、*アクティベーション*という用語はユーザーがアクティブかどうかを示す一連のプロパティを指すものとして使用されている。これには、ユーザーが有効か、無効か、

アーカイブされているか、いつ有効にすべきか、いつまでアクティブでいるべきか、等を示すプロパティが含まれる。1990 年代には、シンプルな有効／無効フラグだけで十分だったかもしれない。しかし今はとてもそれだけでは足りない。ゆえに midPoint の*アクティベーション*は実にリッチなデータ構造になっている。詳しいことは後で説明するとして、今は基本的な考えのみ説明する。

アクティベーションで最も重要な概念は*管理ステータス*である。管理ステータスはオブジェクト（ユーザー）の「管理状態」を定義する。すなわち、当該ユーザーが有効か無効か、またはアーカイブされているかを管理者が明示的に決定するものである。管理ステータス以外には、有効期間、ロックアウトステータス、様々なタイムスタンプ、メタデータがある。しかしこれらは後ほど説明しよう。

認識すべき重要なことは、ユーザーとアカウントの両方がアクティベーションプロパティをもっており、それらはほぼ同じということである。ユーザーおよびアカウントのアクティベーションは同じプロパティ名、意味、データ形式を使用しているが、これは重要なことである。なぜなら、おそらくあなたはアカウントアクティベーションにユーザーアクティベーションをまねてほしいだろうからだ。例えばユーザーが無効になったら当該ユーザーのアカウントもすべて無効になるといった具合である。midPoint ではユーザーとアカウントのアクティベーションが一致しているため、これは非常に簡単である。つまり必要なのはシンプルなマッピングだけである。そのためにリソーススキーマ操作セクションには特別な場所が設けられている。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                        <!-- attribute handling comes here --->
                        <activation>
                                <administrativeStatus>
                                        <outbound/>
                                </administrativeStatus>
                        </activation>
                </objectType>
        </schemaHandling>
</resource>
```

こんなにもシンプルである。ほかに必要なものはない。ユーザーには administrativeStatus プロパティがあり、アカウントにも administrativeStatus プロパティがある。よって midPoint はマッピングのソースとターゲットが何か分かる。また、administrativeStatus プロパティの値も両方とも同じである。よってデフォルト設定である asIs のマッピングで問題ない。midPoint が知るべきことは、マッピングがともかく存在しているということ、そして我々は値をマップしたいということだけである。midPoint はすべての詳細情報を入力する。

このマッピングが存在している状態でユーザーが無効になると、アカウントも無効になる。ユーザーが有効になると、アカウントもこれに追随する。

クレデンシャル

クレデンシャル管理はアイデンティティ管理における重要な部分である。midPoint はクレデンシャルを多くのアカウントに簡単に同期できるようになっている。アクティベーションと同様、ユーザーとアカウントのクレデンシャルデータ構造も一致している。そのため、パスワードの同期に必要なのはシンプルなマッピングのみである。

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
        <name>LDAP</name>
        ...
        <schemaHandling>
                <objectType>
                        <kind>account</kind>
                        <default>true</default>
                        <objectClass>ri:inetOrgPerson</objectClass>
                        <!-- attribute handling comes here --->
                        <credentials>
                                <password>
                                        <outbound/>
                                </password>
                        </credentials>
                </objectType>
        </schemaHandling>
</resource>
```

midPoint にてユーザーパスワードに変更があると、このようなマッピングをもつリソースすべてに変更後のパスワードが伝達される。

プロビジョニングの完全例

本セクションでは、LDAP ディレクトリに接続する完全な実例を説明する。下記の構成を使用して、OpenLDAP サーバーに自動でアカウントを作成する。全体構成は単一のリソース定義ファイルに格納されている。以下の段落にてファイルの各部分について説明する。なお、分かりやすいように XML の簡易表記を使用している。midPoint で直接利用できる形式での完全なファイルは、本書に掲載の他のすべてのサンプルと同じ場所に格納されている（詳細は追加情報の章を参照されたい）。

リソース定義はオブジェクトタイプではじまり、OID、名前、説明となる。これらについては一目瞭然である。

```
<resource oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c">

        <name>OpenLDAP</name>

        <description>
                LDAP resource using a ConnId LDAP connector. It contains configuration for use with OpenLDAP
                servers.
                This is a sample used in the "Practical Identity Management with midPoint" book,
chapter 4.
        </description>
        …
```

次にくるのがコネクタ参照だ。LDAP コネクタを指し示すため、ここでは検索フィルタを使用している動的参照を利用し、そのコネクタを見つけ出す。

```
        …
```

```
<connectorRef type="ConnectorType">
        <filter>
                <q:equal>
                        <q:path>c:connectorType</q:path>
                        <q:value>com.evolveum.polygon.connector.ldap.LdapConnector</q:value>
                </q:equal>
        </filter>
</connectorRef>
…
```

このオブジェクトがインポートされると、参照は解決される。この解決プロセスでは検索フィルタが使用され、そのフィルタで指定した connectorType に応じてコネクタオブジェクトが検索される。

次にくるのがコネクタ構成である。このブロックではホスト名、ポート、パスワード等、コネクタ構成プロパティを指定する。

```
<connectorConfiguration>
        <icfc:configurationProperties>
                <icfcldap:port>389</icfcldap:port>
                <icfcldap:host>localhost</icfcldap:host>
                <icfcldap:baseContext>dc=example,dc=com</icfcldap:baseContext>
                <icfcldap:bindDn>cn=idm,ou=Administrators,dc=example,dc=com</icfcldap:bindDn>
                <icfcldap:bindPassword><t:clearValue>secret</t:clearValue></icfcldap:bindPassword>
                <icfcldap:passwordHashAlgorithm>SSHA</icfcldap:passwordHashAlgorithm>
                <icfcldap:vlvSortAttribute>uid,cn,ou,dc</icfcldap:vlvSortAttribute>
                <icfcldap:vlvSortOrderingRule>2.5.13.3</icfcldap:vlvSortOrderingRule>
                <icfcldap:operationalAttributes>memberOf</icfcldap:operationalAttributes>
                <icfcldap:operationalAttributes>createTimestamp</icfcldap:operationalAttributes>
                </icfc:configurationProperties>
                <icfc:resultsHandlerConfiguration>
                <icfc:enableNormalizingResultsHandler>false</icfc:enableNormalizingR esultsHandler>
                <icfc:enableFilteredResultsHandler>false</icfc:enableFilteredResultsHandler>
                <icfc:enableAttributesToGetSearchResultsHandler>false</icfc:enableAttributesToGetSearc
hResultsHandler>
        </icfc:resultsHandlerConfiguration>
</connectorConfiguration>
```

本ブロックの最後の部分では、ConnId フレームワークの結果ハンドラを無効にすべきと定義する。ConnId 結果フィルタリングは旧式の仕組みであり、ほとんどのコネクタがもはや必要としていない。多くの場合むしろ有害ですらある。あいにくデフォルトではこの仕組みがオンになっている。そのためすべてのハンドラを明示的にオフにできるよう、リソース構成の大半にこの部分が含まれている。

これらの部分だけでも最小限のリソースはすでに定義されているはずである。名前、コネクタ参照、コネクタ構成だけでも定義すれば、リソースを midPoint にインポートできるはずである。接続テストもパスするはずであり、リソースの内容を閲覧できるようになるだろう。しかしまだ IDM ロジックや自動化は一切ない。そこで次にこれらを追加する。

通常、コネクタ構成に続く次の要素はスキーマである。しかし、リソース定義を含むどのファイルを見てみても、ほとんどの場合、そのような要素は見つからないだろう。schema 要素は midPoint によるリソースへの初回接続時に midPoint にて自動生成される。よって定義にスキーマ要素を含める必要はないのである。

定義に含めなければならないのは midPoint によるスキーマの取り扱い方法である。これは schemaHandling セクションに定義される。schemaHandling セクションには一つだけ objectType 定義が含まれる。OpenLDAP サーバー上で通常のユーザーアカウントを取り扱う方法を定義してみよう。

```
…
<schemaHandling>
        <objectType>
                <kind>account</kind>
                <displayName>Normal Account</displayName>
                <default>true</default>
                <objectClass>ri:inetOrgPerson</objectClass>
                …
```

これは我々が取り扱おうとしているオブジェクトの*カインド*を定義する場合である。この場合それはアカウントである。これは*デフォルトの*アカウントである。つまり、アカウントタイプが明示的に指定されていないと、この定義が使用されるということである。また、表示名の指定もある。これは自動化ロジックで使用されることはなく、この定義を参照するときにユーザーインターフェイスによって使用される。そして最後が*オブジェクトクラス*の指定である。オブジェクトクラスの inetOrgPerson はアカウントの新規作成で使用される。オブジェクトクラスの指定ではアカウントが持てる属性を定める。

objectType（オブジェクトタイプ）定義には属性操作の指定も含まれる。自動または特別な方法で扱いたい属性ごとにセクションが 1 つある。定義は最も重要な属性、すなわち LDAP の識別名（distinguished name：DN）からはじまる。

```
                …
                <attribute>
                        <ref>ri:dn</ref>
                        <displayName>Distinguished Name</displayName>
                        <limitations>
                                <minOccurs>0</minOccurs>
                        </limitations>
                        <outbound>
                                <source>
                                        <path>$focus/name</path>
                                </source>
                                <expression>
                                        <script>
                                                <code>
                                                        basic.composeDnWithSuffix('uid', name,
'ou=people,dc=example,dc=com')
                                                </code>
                                        </script>
                                </expression>
                        </outbound>
                </attribute>
                …
```

ref 要素は取り扱おうとしている属性の名前を指定する。実際にこれはリソース定義の自動生成された schema 部への参照である。そのあとに表示名の定義が続く。表示名はこの属性と連携する項目のラベルとしてユーザーインターフェイスによって使用される。表示名をこのように定義すると、デフォルトで使用される分かりにくい「dn」ではなく、正確

な「Distinguished Name（識別名）」ラベルが使用されるようになる。

制限定義については、今は省略しよう。後ほど説明する。

次にアウトバウンドマッピング定義がくる。ここで自動化ロジックを指定する。ここで DN 値が計算される。ユーザーオブジェクトの name プロパティがこのマッピングのソースである。name プロパティにはたいていユーザー名（ログイン名）が含まれている。このマッピングではその値が式に使用される。式では以下のような形で DN が作成されるはずである。

uid=*username*,ou=people,dc=example,dc=com

この式は優れている。式自体で機能し、DN を作成するためライブラリ機能を呼び出す。DN 作成ならシンプルに文字列連結を使用すればよいと思われるかもしれない。しかし最終的な DN でエスケープが必要な特定の文字が DN コンポーネントに含まれていると、この方法では失敗してしまう。composeDnWithSuffix ライブラリ機能はこれに対応しており、適切な DN を作成する。

DN 作成が必要となるたびにアウトバウンドマッピングは評価される。オブジェクトの新規作成時には当然ながらこのマッピングが発生する。しかしユーザーの名前変更（ユーザー名変更）の際も同じマッピングが使用される。マッピングがソースの指定を必要とする理由がここにある。名前の変更はかなり扱いが難しく、複雑な操作であることが多い。安くはないだろうし、アプリケーションによっては一切元に戻せない可能性もある。本当に必要でないかぎり、絶対に DN 変更は誘発したくない。マッピングソースの指定によって我々は DN 変更が*いつ*必要か分かる。このケースでは、ユーザーオブジェクト変更の name プロパティで DN を変更するよう教えてくれている。

さて、ここで limitations セクションに戻るのがよいだろう。dn 属性はスキーマによって必須属性と定義されている。これはまったく正しい。DN がなければ LDAP オブジェクトを作成できないからだ。midPoint はすべてにおいてスキーマを使用している。それゆえ、この LDAP アカウントを編集するためフォームを表示するとき、midPoint は DN に値があるか確認する。DN は必須属性である。しかしユーザーにフォームへの DN 入力はさせたくないのが普通である。自動でこれを計算したい。これがまさにアウトバウンドマッピングの重要な点である。とはいえ、式がいつ値を計算し、いつ値を計算しないのかが midPoint にはわからない。式は一般的な Groovy コードである。midPoint が分かるかぎりでは、その式は何でもできる。それゆえ式があるとしても、midPoint はスキーマに固執し、なおもユーザーによる DN の入力を必要とするだろう。しかし我々は式を記述しており、その式が（妥当な）入力に対し値を生成することを知っている。したがって我々は DN がもはや必須ではないことを、そしてユーザーが GUI フォームに DN を入力しなくても問題ないことを midPoint に教えたい。そしてそれがまさに limitations セクションが行うことである。このセクションは自動生成されたスキーマをオーバーライド（上書き）し、dn 属性を必須から任意へと変える。

そしてこれで以上である。今我々は dn 属性の挙動を定義した。他の属性の挙動を定義する場合も同様の手法を使用できる。例えば、cn 属性の操作は同様の定義をもつ。

```
…
<attribute>
        <ref>ri:cn</ref>
        <displayName>Common Name</displayName>
        <limitations>
                <minOccurs>0</minOccurs>
        </limitations>
        <outbound>
                <source>
                        <path>$focus/fullName</path>
                </source>
        </outbound>
</attribute>
…
```

このケースではアウトバウンドマッピングはあるが明示的な式はない。式がない場合、ソースから値が取得され特に変更は加えられない（「as is（そのまま）」）。したがって属性 cn はユーザーのプロパティ fullName と同じ値を持つことになる。

また、マッピングがなくても属性を定義することができる。

```
…
<attribute>
        <ref>ri:entryUUID</ref>
        <displayName>Entry UUID</displayName>
</attribute>
…
```

これは、midPoint は entryUUID 属性に対して一切自動処理を提供しないことを意味する。この定義は属性に分かりやすい表示名を設定するためだけに使用する。

マッピングおよび式の柔軟性はほぼ無制限である。例えば、以下の定義では description 属性に対し静的な値を設定する。

```
…
<attribute>
        <ref>ri:description</ref>
        <outbound>
                <strength>weak</strength>
                <expression>
                        <value>Created by midPoint</value>
                </expression>
        </outbound>
</attribute>
…
```

静的な値に対しては、ソースはまったく意味をなさないため、マッピングにはソースがない。値は常に同じである。また、このマッピングの強度が *weak（弱）* であることもわかるだろう。このマッピングは、description 属性に値がないときのみ、その属性の設定に使用される。現行値を上書きすることはない。

inetOrgPerson オブジェクトクラスは、スキーマハンドリングセクションに定義されたものよりもずっと多くの属性をもっている。これらの属性はユーザーインターフェイスに自動で表示される。midPoint は生成されたリソーススキーマを使用してそれらの名前とタイプを判断する。midPoint がこれらの属性を表示し、ユーザーはこれを変更でき、midPoint がその変更を実行する。しかしそれ以外には、midPoint はこれらの属性について特別な処理を行うことはない。schemaHandling セクションにすべての属性を列挙しなくても問題ない。必要なのは、特別な方法で処理したい属性のみを定義することだけである。

例が完成する前に説明しなければならない定義があと 2 つある。一つは*アクティベーション*定義である。これは非常にシンプルだ。

```
…
<activation>
        <administrativeStatus>
                <outbound/>
        </administrativeStatus>
</activation>
…
```

この定義は、アクティベーションの管理ステータスの処理を指定するものである。このステータスプロパティは、アカウントが有効か無効かを指定する。midPoint のアクティベーションプロパティはやや特殊である。midPoint はアクティベーションプロパティの意味と値を理解する。また、ユーザーアクティベーションとアカウントアクティベーションが通常は一緒にマップされると予期している。したがって、midPoint にはそのようなマッピングが必要であることを話すだけでよい。midPoint はすでにソース（ユーザーアクティベーション）とターゲット（アカウントアクティベーション）を知っている。ユーザーが無効になればアカウントも無効になる。ユーザーが有効になればアカウントも有効になる。

注意：賢い読者は今頃きっと頭をかきむしっていることだろう。アカウントを有効もしくは無効にする方法を定めた LDAP 標準はない。それに加え、OpenLDAP には無効アカウントの概念すらない。となると、midPoint はどうやって LDAP アカウントを無効にする方法を知るのか？実をいうと、midPoint はその方法を知らない。シンプルなアクティベーションマッピングを例示したかったため、ここでは若干詩的許容を使ったのだ。ただこの場合、それ単体では機能しない。OpenLDAP リソースにはこの*機能*がない。しかし方法はある。この機能はシミュレーションできるのだ。これについては後ほど話そう。とにかく今は、この全く何もしないアクティベーションマッピングの素晴らしさに驚嘆しようではないか。

リソースがうまく機能するには、*クレデンシャル*のマッピングも定義する必要がある。このケースでは、それはパスワードマッピングである。

```
…
<credentials>
        <password>
                <outbound/>
        </password>
</credentials>
…
```

アクティベーションと同様、クレデンシャルもまた midPoint では特殊である。midPoint はクレデンシャルがどう機能するか、それらの値が何かなどを理解する。midPoint はパスワード等のユーザークレデンシャルが通常はアカウントクレデンシャルにマップされることを予期している。それゆえ midPoint が知るべきことは、そのマッピングが必要ということだけである。midPointは自動でソースとターゲットを判断する。アカウントはユーザーと同じパスワードを持つだろう。アカウントの新規作成時にユーザーのパスワードが使用される。そしてユーザーがパスワードを変更すると、その変更がアカウントにも伝わる。

そしてこれで以上である。これではじめて（ほぼ）稼働するリソースをもつことになる。midPointに定義をインポートしテストできる。ユーザーにリソースをアサインするだけだ。OpenLDAP アカウントが作成され、DN および不可欠な属性がすべて自動で計算される。midPoint はアカウントを作成するとそのことを覚えている。よって簡単にそのアカウントを削除できる。リソースのアサインを解除するとアカウントが削除される。これが自動プロビジョニングとデプロビジョニングの仕組みである。真のプログラミングは一切不要である。必要なのは宣言型の指定と 1〜2 行の非常にシンプルなスクリプトのみだ。この構成はすべて 2〜3 分でできる。そしてこれは基本的にすべてのアプリケーションに共通のプロセスである。唯一違うのはコネクタである。コネクタは構成プロパティが違い、属性名も異なる。しかしその原則とツールは同じである。この方法で多くの異種アプリケーションを簡単に接続できる。コネクタおよびマッピングはその違いを隠している。結局、midPointにはすべての「リソース」が同じに見える。これらの管理には同じ原則が使用される。それゆえ、大規模であっても管理を効率よく行うことができる。

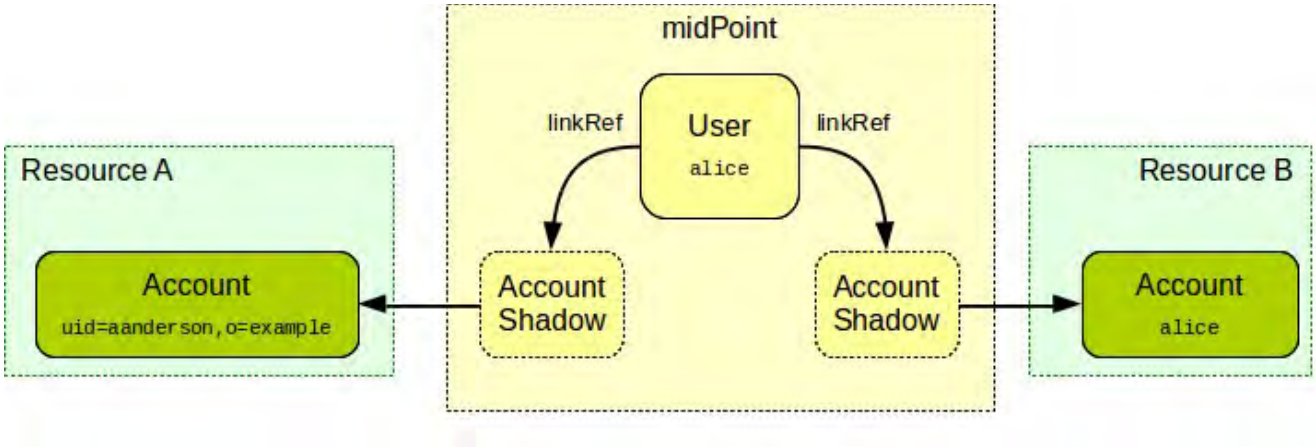
しかもこの構成はまだ極めて基本的なものである。我々はまだ midPoint でできることを表面的に論じているにすぎない。次章以降にも見ていかねばならないものはたくさんある。

シャドウ

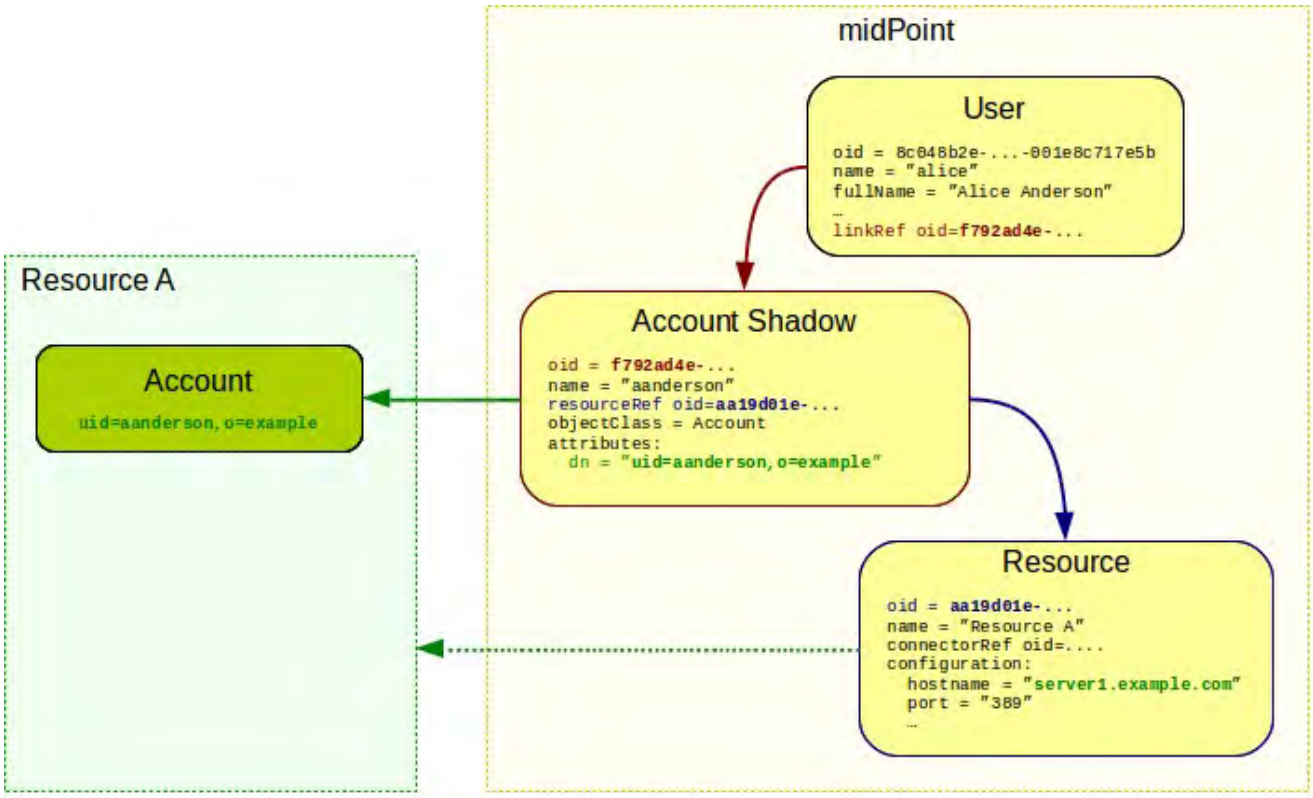
ユーザーとアカウントをリンク付けすることは、しかるべきアイデンティティ管理システムの基本原則の一つである。しかし、このリンクの実現は驚くほど難しい。確実にアカウントを特定する方法は数多くあり、システムごとにかなり違う。ユーザー名だけでアカウントを特定できるシステムもあるが、それだと名前変更を確実に検出することがかなり難しい。この点を改善すべく別の固定識別子を導入しているシステムもある。その識別子はリソースによって付与され、決して変わることはない。しかしユーザー名はなおも二次識別子として使用されており、しかも一意でなければならない。ほかには、2 つ以上の値からなる複合識別子をもつシステムがある。また、グローバルに一意の識別子をもつものもあれば、システム内のオブジェクトクラスの中でのみ一意の識別子をもつシステムもある。階層構造を持つ識別子をもつシステムもあれば、フラットな非構造化識別子をもつものもある。大文字小文字を区別した文字列の識別子もあれば、区別しないものもあり、バイナリの識別子さえある。ほかにも様々だ。早い話、信頼性の高い識別は本当に複雑である。

ユーザーオブジェクトをアカウント識別のあらゆる繊細な詳細情報で汚すようなことはしたくない。そのため、リソース関連の詳細と識別の複雑さをすべて隠す midPoint オブジェクトを別に作成した。我々はこれを単純に*シャドウ（shadow）*と読んでいる。真のアカウントの影として動作するからである。midPoint が新アカウントを検出すると、そのアカウント用に*シャドウ*が新規作成される。アカウントが変更されたこと（名前変更等）を検

出すると、midPointはそのシャドウを自動で更新する。アカウントが削除されるとmidPointはそのシャドウも削除する。ただ厳密には、シャドウは midPoint の通常のオブジェクトである。よってシャドウはオブジェクト識別子（OID）をもっている。他のオブジェクトは通常のオブジェクト参照を利用してシャドウを指し示すことができる。そしてこれがまさにユーザーとアカウントのリンクが構築される仕組みである。



シャドウオブジェクトには、アカウントまたは他のリソースオブジェクト（グループあるいは組織単位等）を確実に特定するために必要なすべての情報が含まれている。またシャドウオブジェクトは、識別を完了させるため該当するリソース定義を指し示す。シャドウは多目的なオブジェクトである。midPoint で他の目的でも使用される。リソースオブジェクトに関するメタデータを記録する。キャッシュされた属性値の保持にも使用される（この機能は midPoint 3.5 ではまだ実験段階）。さらに近い将来には、midPoint がオンライン通信チャネル（コネクタ）をもたないリソースオブジェクトの状態を保持する目的でも使用する予定である。それゆえシャドウは非常に複雑なオブジェクトである。シャドウについてより詳しく例を示したのが下図である。



この図が少し恐ろしく見えたとしても、心配はいらない。シャドウは複雑かもしれないが、midPoint ユーザーからはほとんど見えない。midPoint が自動で保守する。通常の環境ではシャドウの保守に必要なものはすべて midPoint が取得するため、これに関して特別に構成する必要はない。ここでシャドウオブジェクトの仕組みについても説明しているのは、完全を期すためという理由がほとんどである。それにこの知識が役立つ状況があるかもしれない。そのような状況は、多くは midPoint の構成にエラーがありシャドウが誤って作成された場合である。そうなった場合はすべてのシャドウを完全消去（パージ）して最初からやり直す必要があるだろう。しかし注意してほしい。シャドウはユーザーとアカウントのリンク付けに使用するため、シャドウを完全消去してしまうとリンクも失われてしまう。ただ、たいていは、それさえもさほど重要ではない。次章で説明する同期方法を使用すれば、簡単にリンクを再作成できるからだ。

第 6 章：　同期

データを得る前に理論を立てるのは重大な過ちだ。

– シャーロック・ホームズ
（アーサー・コナン・ドイル著
「シャーロック・ホームズの冒険」）

どのソフトウェアシステムにとっても、データは生命線である。ソフトウェアアーキテクトにとってデータ管理はもっとも重要な責務の一つである。しかしデータ管理は非常に厄介になりうるものであり、経験豊富なソフトウェアアーキテクトであればこのことは百も承知である。ソフトウェアアーキテクチャにおける重要な原則の一つが、「繰り返しを避ける（Do not repeat yourself：DRY 原則）」としてしばしば明確に述べられている。データを繰り返してはならない。データのコピーがあってはならない。コピーがなければ普遍的な真実のソースは一つということになる。コピーがなければデータは常に一貫している。コピーがないということは矛盾がないということである。真実は一つだけ、正確で極めて明白である。データは一か所で保持し、一か所のみにする。こういう理論である。

しかしいざ実践となると話が違ってくる。実際には相容れない技術が数多くある。関係データベースに構築されたアプリケーションは、ディレクトリサービスのデータを直接使用できない。関係データベースどうしでさえ組み合わせるのは簡単ではない。各アプリケーションは違うデータモデルを念頭に置いて設計されている。データ変換やブリッジングという技術があり、これらは互換性問題を解決するアダプターとして機能する。しかしコストがかかる。データブリッジでは待ち時間が追加されるため、ほぼ例外なくパフォーマンスに相当な悪影響を与えてしまう。トランザクション処理と整合性が実に問題となる。アダプターはクリティカルパス上に追加されるコンポーネントであり、もし故障すれば大きな痛手となる。結果として生じるシステムは運用上不安定であることが多い。小さなコンポーネントでさえ故障すればシステム全体の故障となってしまう。その解決に途方もない複雑さとコストが伴うことは言うまでもない。

一方、マイアプリケーションに必要なデータをすべてコピーするならば実に手ごろである。アプリケーションはたった一つの等質な仕組みを使用して簡単にデータにアクセスできる。他のコンポーネントに故障があってもクリティカルパスに影響はない。そしてパフォーマンス的にもかなり優れている。データコピーによって厄介な問題はほぼすべて解決する。ただ一つ小さなことを除いては。それはデータを常に最新の状態に保つということである。そこで出番となるのが同期の仕組みである。

実際にはデータのコピーを複数もつことは避けられない。アイデンティティデータもその例外ではない。実際、アイデンティティデータはしばしばコピーによって最も影響を受ける。そしてこれは大いに納得がいく。ほぼすべてのアプリケーションがユーザーに関する何らかのデータを保持している。そして通常そのようなデータは、セキュリティおよび個人情報の観点から非常に慎重な扱いを要するものである。

そのデータのコピーは避けられない。そこで我々ができる最善の策は、そのコピーを必ず

同期させるということだ。アプリケーションの中にはLDAPまたはディレクトリ同期のサポートが組み込まれているものもある。しかしこうした仕組みは品質的にかなり劣っていることが多い。例えば、多くのアプリケーションがログイン時におけるディレクトリサービスとのオンデマンド同期機能を提供している。一般的にその動作は以下の通りだ。

1. ユーザーがアプリケーションのログインダイアログにユーザー名およびパスワードを入力する。
2. アプリケーションがディレクトリサービスに接続し入力されたパスワードを検証する。
3. パスワードが正しければ、アプリケーションはディレクトリからユーザーデータを取り出す。
4. アプリケーションはユーザーデータのコピーをローカルに格納する。
5. 通常通り、その後はローカルコピーのデータが使用される。

この手法は最初のうちは非常にうまくいく。ただしばらくするとデータがおかしくなり始める。ユーザーの名前変更があっても、ローカルコピーは更新されない。ユーザーが削除されても、ローカルコピーはいつまでも残っている。ディレクトリサービスに反映されないローカルアカウントや特権があるため、何年も検出されないままになってしまうのだ。

これよりはましであろう高度な同期プロセスをもつアプリケーションもある。しかしこれをうまく行うアプリケーションはほとんどお目にかかれない。そこにはもっともな理由がある。同期は見かけよりもはるかに難しいのだ。両サイドにデータの矛盾があるかもしれない。ネットワークエラーがあるかもしれない。構成エラーがあるかもしれない。データモデルは、時間がたてば進化していく。ポリシーは変わりつつある。データを確実に同期させることは簡単ではない。したがって、アイデンティティデータの同期に特化した特別なシステムがある。アイデンティティ管理システムである。

midPointにおける同期

同期そのものはmidPointの基本原則の一つである。同期の仕組みは当初よりmidPoint設計に盛り込まれてきた切り離せない部分である。実際、midPointが通常行うことの多くは同期である。それがはっきりと分かるのは、例えばリコンシリエーションプロセスでリソース上のアカウント属性とmidPointのユーザープロパティを同期させるといったケースである。一方、あまり目立たないケースもある。たとえばmidPointがユーザー用のアカウントを新規作成する必要がある際の、ごく普通のプロビジョニングケースである。しかしこのケースも実際には同期なのだ。midPointユーザープロパティが、リソース上に作成されたまだからっぽの新しいアカウントと同期されるからである。midPoint操作の大半で、直接的または間接的に同期の仕組みが使用されているのである。

情報：この仕組みを再利用することはmidPointの基本原則の一つである。midPointの設計をしたとき、我々はその機能ごとに別々の原則を創り出すようなことはしなかった。むしろmidPointのあらゆるところで再利用するため、非常に汎用的な原則をわずかに設計しただけである。同期はそうした原則の一つである。同期ロジックのコア部分を実装する

コードが一つある。このコードは相互に関連するオブジェクトを「整合」させる必要があるたびに使用される。*ユーザーとアカウント*のリコンシリエーション、通常のプロビジョニング、ロールベースのプロビジョニング、ライブ同期、整合性など、ほとんどすべてにおいてこの同じコードが使用される。

midPointの同期は特定のニーズに合わせて微調整し最適化できるほぼ連続した機能スペクトルである。それでも、大まかで重複するいくつかのカテゴリに分類することができる。

- **ライブ同期（Live synchronization）**はほぼリアルタイムで同期する仕組みである。midPoint は継続的にリソースをスキャンし変更がないか確認する。変更を検出すると、midPoint はただちにこれを処理する。通常待ち時間はリソースの能力に依存するものの、だいたいは数秒から数分の間である。変更のみがライブ同期によって処理される。そのため大規模なデプロイであってもレスポンスが早く、非常に効率のよい仕組みである。
- **リコンシリエーション（Reconciliation）**はデータを比較しそれらを補正するプロセスである。アカウントが調整されると、midPoint はそのアカウントが持つべき属性値を計算する。この計算値をアカウントが持つ本当の値と比較する。差異があれば修正される。リコンシリエーションは全てのアカウントをひとつずつ比較する、非常に重厚な仕組みである。しかし非常に信頼性の高い仕組みでもある。ライブ同期で捉えきれなかったミスを修正でき、また、重大な故障や破損などが発生した際もデータを修正できる。一般にリコンシリエーションは定期的に実行される。しかしその性質上、たいていはオフピーク時（夜間や週末）に実行される。
- **インポート（Import）**はリソースから midPoint へデータを取り込む一回限りのプロセスである。インポートはmidPointに初回データを取り込むために使用される。あるいは新リソースを midPoint に接続するために使用される場合もある。小さな違いはわずかにあるものの、インポートはリコンシリエーションとほぼ同じである。ただ目的が違うため、たいていはインポートポリシー（マッピング）の構成も若干違う。インポートはスケジュールされておらず、必要に応じて手動でトリガーされることが一般的だ。
- **便宜的同期（Opportunistic synchronization）**は midPoint に極めて特有の、非常に特殊なものである。midPoint が何らかの不備を検出すると、便宜的同期が自動でトリガーされる。例えば、midPoint がアカウントを変更しようとしても、該当するアカウントがないことを発見する。するとその単一アカウントのためだけに同期の仕組みがトリガーされる。通常これはアカウントの再作成を意味する。便宜的同期では、midPoint がアカウントを新規作成しようとするがすでにそのアカウントが存在しているときにもトリガーされる。この手法によって midPoint は自己修復システムとなる。何らかの問題に遭遇しても、midPoint はしばしばおのずとそれを修正できる。

データ矛盾の発見の仕方は各仕組みによって異なる。ライブ同期は積極的に新たな変更を探し、リコンシリエーションはデータを一つ一つ比較する。そして便宜的同期はまさに偶然で矛盾を発見する。しかしデータ矛盾に対してはどの仕組みも同じように対応する。システムの修正方法を定めているポリシーは一つだけだ。もちろん、挙動には若干の差もあるだろう。例えば、*インポート*の挙動は*リコンシリエーション*とは若干違うようにしたい

ものである。そして midPoint ではそれができる。しかし全体としては同期方法は大きな一つのスキームがあるだけだ。これにはもっともな理由がある。どのようにして問題が発見されたかはあまり大したことではない。本当に大切なのはそれが修正されることである。そしてそのために 4 つも違う構成を維持していたいとは思わない。一つポリシーがあれば十分である。midPoint は各特定の状況に対し、構成のどの部分を適用すればよいか分かっている。そしてそれをまさに自動で行う。この統合手法によって、midPoint の同期の仕組みの構成は大幅に簡素化されている。またこれは、各同期の仕組みの境界線が実にあいまいな所以でもある。実際のところ、これはまさに複数のファセットをもつ大きな単一の仕組みである。

ソース、ターゲット、その他生成物

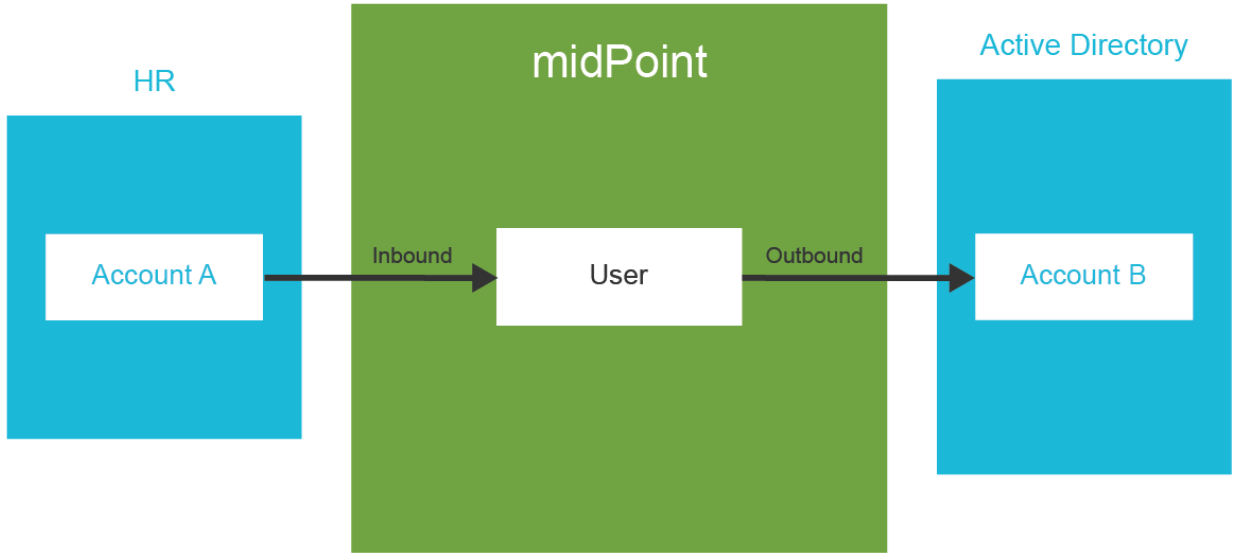
通常、理想的なアイデンティティ管理のデプロイは人事（HR）システムから開始される。人事システムはアイデンティティのすべてに関するレコードを持っており、*権威あるソース*のリソースである。アイデンティティ管理システムは人事システムから全てのデータを取り込み、再計算してから*ターゲットリソース*上にアカウントを作成する。めでたし、めでたしである。

さて、現実にもどろう。人事リソースは多くの場合権威あるソースであることに間違いはない。しかしそのソースには制限がある。従業員のデータしかないのだ。しかも部分的な情報だけである。例えばユーザー名がなく、ロジックで生成しなければならない。初期パスワードがない。組織構造のアサインはしばしば不完全か欠如しているか、あるいは信頼性に欠けている。こうしたことから、人事システムは*部分的に権威あるソース*にすぎないのだ。我々のシステムにアクセスしなくてはならない請負業者、パートナー、供給業者、サポートエンジニアといったアイデンティティには、ほかにも権威あるソースがあるだろう。これらは*付加的なソースシステム*である。そして、しばしばアクティブディレクトリとなるディレクトリシステムがある。これは、理論上は*ターゲットリソース*のはずである。しかしたいていはここにも信頼できる情報がある。例えば、ユーザー名を生成するアルゴリズムは、アクティブディレクトリにすでに取り込まれているユーザー名をもとにするかもしれない。アクティブディレクトリは E メールアドレスの作成にも必要だろう。ディレクトリシステムは電話番号、オフィス番号といった情報についてやや信頼できるソースとしても使用される。よってこのようなリソースは*ターゲット*リソースもあれば*ソース*リソースもある。そして最後がターゲットリソースである。これらはいずれにしても権威あるリソースではない。アイデンティティ管理システムはターゲットリソースには書き込みを行うだけである。さもなければどうなるか？ターゲットリソースに矛盾するアカウントがすでに存在しているため新入社員のアカウントを新規作成できないと、いったいどうなるか？そこにあるはずのないアカウントが本当にないか、どうやって確認したらよいか？結局はターゲットシステムにも価値のある情報が含まれていることが分かる。

現実はソース、ターゲット、セミソース、ターゲット／ソース、準ターゲットというリソースがやみくもに組み合わさっており、これらを定義済みのボックスに入れることはほぼ不可能である。そのため midPoint ではソースリソースまたはターゲットリソースの概念をわざわざ定義していない。すべてのリソースがソースにもターゲットにもなりうる。そして各属性の権威性は非常に細かいレベルで制御できる。現実世界のほぼすべての状況を、このモデルに簡単に適合させることができる。

インバウンドマッピングおよびアウトバウンドマッピング

midPoint は断固として再利用の原則にもとづいている。プロビジョニング中の属性の挙動はマッピングで制御していることは前章で説明した。よって、同期中の属性の挙動もマッピングで制御していることはさほど驚くことでもないだろう。実際、プロビジョニングとはまさに同期の特殊なケースである。組み合わさった仕組みを表したのが下図である。



マッピングは次のように 2 種類ある。

- **インバウンドマッピング**はデータを midPoint へマップすることである。このマッピングでは、ソースリソースからデータを取得し、それを変換したのち、その結果をユーザーオブジェクトに適用する。
- **アウトバウンドマッピング**は midPoint からのデータをマップする。このマッピングでは、ユーザープロパティを取得し、それらを変換したのち、その結果をターゲットシステム内の属性に適用する。

マッピング自体はインバウンドかアウトバンドかに関わらずほぼ同じである。どちらもソース、ターゲット、式、条件等をもっており、ソースとターゲットが逆になるだけである。

	インバウンドマッピング	アウトバウンドマッピング
方向	リソース → midPoint	midPoint → リソース
マッピングソース	リソースオブジェクト（例：アカウント）	フォーカスオブジェクト（例：ユーザー）
マッピングターゲット	フォーカスオブジェクト（例：ユーザー）	リソースオブジェクト（例：アカウント）

これで以上である。前章で使用したものと同じマッピングを考えてみよう。ただ方向を逆にするだけである。今、マッピングはアカウントからデータを取得しその結果がユーザーオブジェクトに適用される。このとおりである。

```
<attribute>
	<ref>ri:lastname</ref>
	<inbound>
		<target>
			<path>$focus/familyName</path>
		</target>
	</inbound>
</attribute>
```

このマッピングはリソースから lastname 属性の値を取得し、その値を midPoint ユーザーの familyName プロパティに格納する。

残りはアウトバウンドマッピングと同じである。インバウンドマッピングにおける式および評価者はすべて、アウトバウンドマッピングの場合と同じように使用することができる。例えば Groovy の式を使用して、midPoint に格納する前に値をサニタイズできる。

```
<attribute>
	<ref>ri:lastname</ref>
	<inbound>
		<expression>
			<script>
				<code>lastname?.trim()</code>
			</script>
		</expression>
		<target>
			<path>$focus/familyName</path>
		</target>
	</inbound>
</attribute>
```

また、アクティベーションやパスワードマッピングにさえ同じ手法をとることができる。ただしパスワードマッピングについては一つ違いがある。パスワードは普通、書き込み専用の値である。パスワードが書き込まれると、通常はハッシュ化され、元の値はもはや取得できなくなる。そこで人事システムのように従業員パスワードを一切保存しないリソースがある。それは我々の読み込んでいる本当のアカウントではなく、コネクタがアカウントとして提供している通常のデータベースエントリにすぎないからである。パスワードのインバウンド同期は一筋縄ではいかず、しばしばあらゆる計画や工夫が必要となる。しかし、実に頻繁に用いられている方法が一つある。ユーザーの初期パスワードはたいていランダムに生成される。これは非常によくあるケースのため、midPoint はこれを簡単に行うことができる。

```
<credentials>
	<password>
		<inbound>
			<strength>weak</strength>
			<expression>
				<generate/>
			</expression>
		</inbound>
	</password>
</credentials>
```

このマッピングはユーザーに対しランダムなパスワードを生成する。このマッピングもgenerate expression evaluator もどちらも非常にスマートである。マッピングは、明示的な指定を必要とせずとも、ターゲットがユーザーパスワードであることを知っている。これに加え、generate expression evaluator はユーザーパスワードポリシーを考慮する。ただどんなランダムパスワードでも生成すればよいということではない。パスワードポリシーを考慮しなければ、我々はパスワードを生成できる。しかしそのパスワードは短すぎるか、長すぎるか、弱すぎてポリシーを満たさないだろう。または強すぎていずれにせよ役に立たないだろう。そのため generate 式はパスワードポリシーを検索して、特定のパスワードポリシーにちょうど適合するランダムパスワードを生成するのである。

ここで、もっと重要な知っておくべきことがある。インバウンドパスワードのマッピングは*弱*になっている。これにはもっともな理由がある。我々は、midPoint のパスワードをランダム生成されたパスワードへと置き換えられたくはない。ランダムパスワードを設定したいのはそれが初期パスワードのときだけである。ここで弱のマッピングがまさに役に立つ。ターゲットに既存の値が一切ないときのみ新たな値が設定されるからだ。よってこのマッピングによって、すでに設定されているパスワードが上書きされることはない。

注：midPoint にはアカウントどうしの直接の同期はない。すでに説明したように、midPoint はスター型トポロジ（別名「ハブアンドスポーク方式」）に従っている。よって同期が行われるのはアカウントからユーザー（インバウンド）またはユーザーからアカウント（アウトバウンド）のいずれかとなる。アカウントどうしの同期の効果は、インバウンドとアウトバウンドの同期の仕組みを組み合わせることで得られる。

相関関係

からっぽの midPoint にすべての人事レコードをインポートすることは極めて簡単である。インバウンドマッピングを設定し、インポートタスクを開始すればすべて行われる。しかし実際の状況はもっとずっと複雑である。からっぽの midPoint で同期のアルゴリズムが機能することはまずない。ライブ同期およびリコンシリエーションは midPoint の現行ユーザーがいてこそうまく機能することになっている。インポートでさえ、例えば付加的データソースからのデータを実行中の midPoint デプロイにインポートしようとするような場合は非常に複雑になりうる。インポートセットには新規ユーザーもいるだろうが、現行ユーザーのアカウントもあるかもしれない。その違いを説明し、違う方法で状況に対応する必要がある。もちろん midPoint にはこれに対応する簡単なソリューションがある。相関関係の仕組みである。

相関関係式は新アカウントと現行ユーザーを結ぶ方法である。その働きは次のとおりである。新アカウントを検出した midPoint は、そのたびに検出したアカウントを現行ユーザーにリンク付けしようとする。ここで相関関係式が使用される。相関関係式は、実際はパラメータ検索クエリである。この検索クエリは新アカウントごとに作成され、そのアカウントが属するユーザーを検索するのに使用される。相関関係式で最も簡単な形式は、識別子を利用して調べることである。

```
<correlation>
    <q:equal>
        <q:path>employeeNumber</q:path>
        <expression>
```

```
                <path>$shadow/attributes/empno</path>
            </expression>
        </q:equal>
</correlation>
```

この相関関係クエリによって、アカウントのempno属性値が取得される。この値は、midPointがメモリ上で計算する検索クエリに置かれる。empno 属性が 007 に設定されたアカウントの場合、 得られる検索フィルタは次のようになる。

```
        <q:equal>
            <q:path>employeeNumber</q:path>
            <q:value>007</q:value>
        </q:equal>
```

midPoint はこの検索フィルタを使用してユーザーを検索する。employeeNumber プロパティが 007 に設定されたユーザーがいれば、そのユーザーが当アカウントのオーナーとみなされる。

midPointはデータ表示について独自の仕組みをもち、オブジェクト構造ももっている。よってオブジェクト構造とうまく連携するよう設計された独自のクエリ言語ももっている。このクエリ言語は他の大半のクエリ言語の構造を模範としているため、習得は難しくない。言語自体は本書の後半および midPoint の文書にて説明する。ただ、このことについてはあまり心配しなくてもよい。相関関係式の大部分は非常に簡単である。実際のところ、多くは上記の例で使用されたような単一の equal 句にすぎない。

これは少しややこしく思われるかもしれないが、必要なことである。相関関係式は検索フィルタでなければならない。なぜなら大量のユーザーからたった一人のユーザーを見つける唯一の効率的な方法だからである。アカウントを一つ一つスキャンしていくことはできない。このためにデータベースの検索能力を活用する必要がある。

同期の状態と対応

相関関係式を使用して新アカウントのオーナーを見つけることができる。これはソリューションの一部ではあるがソリューション全体ではない。オーナーが見つかればアクションは実に明確である。そのアカウントをユーザーにリンク付けし、あとは通常どおりに進む。しかしオーナーが見つからなかったらどうするか？このリソースは権威あるリソースであろうから、そのアカウントをもとにユーザーを新規作成したい。あるいはこれがターゲットリソースとのリコンシリエーションであるとしたら、その場合は不正アカウントを発見したことになる。そのようなアカウントは無効にしたいだろう。また、オーナーが複数見つかった場合はどうするか？これはいよいよややこしいことになる。そこでこれを理解し管理できるよう、midPoint には*同期の状態（synchronization situations）*という概念がある。

midPoint がアカウント変更を扱うときは必ず、そのアカウントの*状態*が判断されるのである。その状態は、アカウントがすでにユーザーにリンク付けされているのか、我々はそのユーザーを知っているのか、もしリンク付けがまだであれば我々はユーザーを判断できないのか、といったことを示している。それぞれの状態について説明したのが下表である。

状態	説明
linked （リンク済み）	アカウントはオーナーに適切にリンク付けされている。これは通常の状態である。
unlinked（未リンク）	アカウントはオーナーにリンク付けされていないが、誰がオーナーかは分かる。相関関係式によってオーナーが誰か分かっているからだ。この場合、リンクは存在すべきと midPoint は考えているが、リンク付けはまだされていない。
unmatched（不一致）	アカウントはリンク付けされておらず、誰がオーナーかもわからない。相関関係式は候補者を一切返してこない。
disputed（無効）	アカウントはリンク付けされておらず､オーナー候補が複数人存在する。相関関係式は複数人の候補者を返してきている。
collision（衝突）	アカウントが複数のオーナーにリンク付けされている。通常の状況ではこれはありえないことである。間違ったカスタマイズやソフトウェアのバグが原因であることが多い。
deleted（削除済み）	アカウントが存在していたがリソース上からは削除されている。

同期の*状態*が判断されれば、midPoint は適切な*対応*が何か分かる。unlinked のような状態に対しては、とるべき対応も実に明確である。しかし他の状態、たとえば unmatched のような状態に対しては対応の仕方も非常にばらつきがある。midPoint が状態ごとに個別に対応を設定できるのはこのばらつきがあるがゆえである。以下のように定義済みの対応がいくつかある。

アクション	説明
Add focus （フォーカスの追加）	midPoint の新ユーザーが作成されアカウントにリンク付けされる。この対応はアカウント作成時に権威あるリソースを対象に構成されたものである事が多い。
Delete focus （フォーカスの削除）	アカウントのオーナーである midPoint ユーザーが削除される。この対応はアカウント削除時に権威あるリソースを対象に構成されたものである事が多い。
Inactivate focus （フォーカスの無効化）	アカウントのオーナーである midPoint ユーザーが無効化される。これも権威あるリソースを対象に使用される。しかし他よりも控え目な対応である。
Link（リンク付け）	ユーザーとアカウント間でリンクが作成される。
Unlink（リンクの解除）	ユーザーとアカウント間のリンクが削除される。アカウントはもはやユーザーとリンク付けされない。
Delete shadow （シャドウの削除）	アカウントを削除する。権威のないリソース上に不正アカウントが検出された際の一般的な対応である。
Inactivate shadow （シャドウの無効化）	アカウントを無効化する。こちらも不正アカウントが検出された際に使用される。

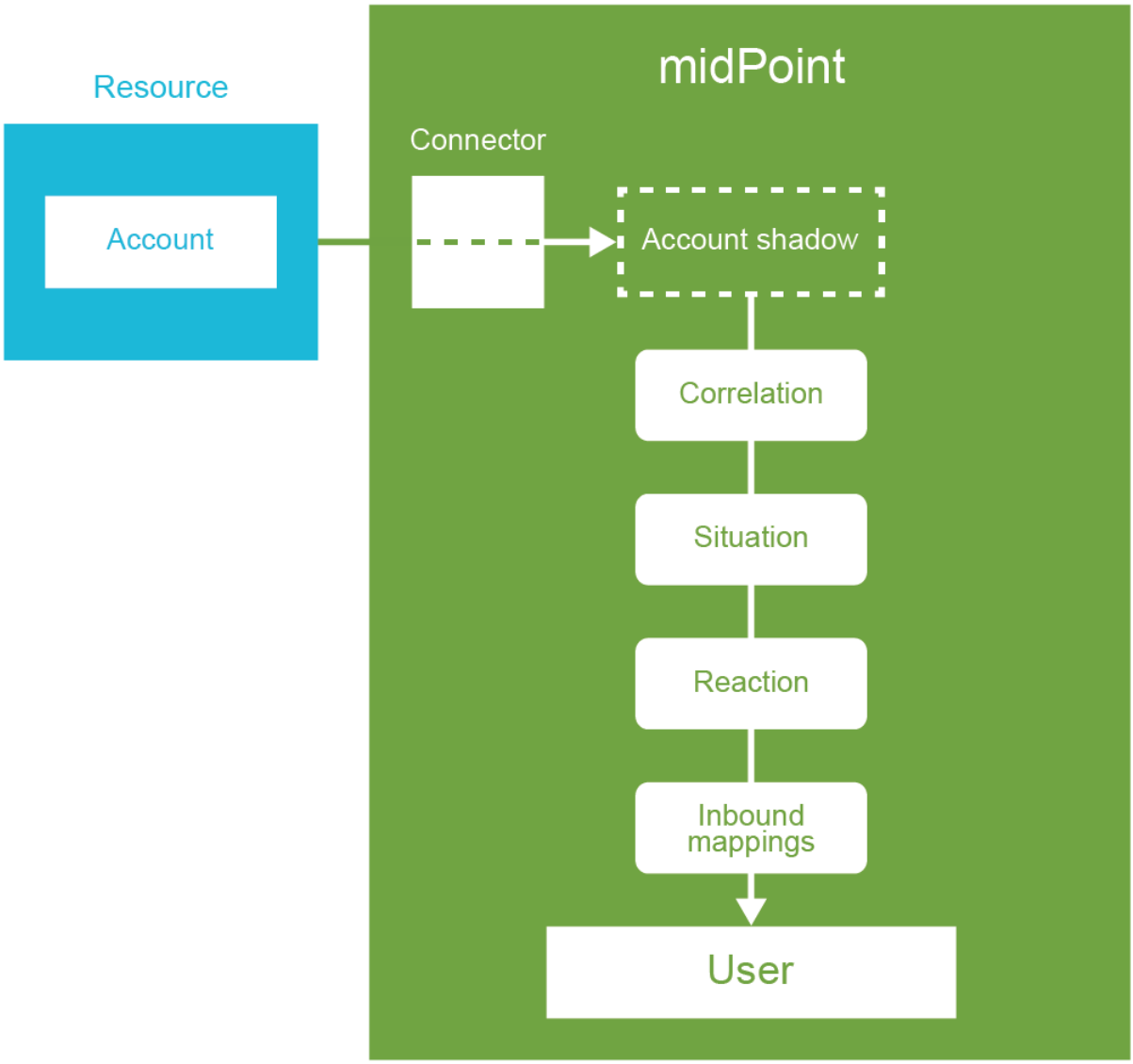
対応が明示的に設定されていない状態に対しては midPoint は何も行わない。midPoint レポジトリにその状態が記録されるだけである。これは、アクションが明示的に設定されていないかぎりデータは変更しないという midPoint の基本的な姿勢の一部である。

対応はリソース構成の同期セクションで定義できる。

```
<synchronization>
        <objectSynchronization>
                <correlation>...</correlation>
                <reaction>
                        <situation>linked</situation>
                        <synchronize>true</synchronize>
                </reaction>
                <reaction>
                        <situation>deleted</situation>
                        <action>
                                <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/act
ion-3#deleteFocus</handlerUri>
                        </action>
                </reaction>
                <reaction>
                        <situation>unlinked</situation>
                        <action>
                                <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#link</handler
Uri>
                        </action>
                </reaction>
                <reaction>
                        <situation>unmatched</situation>
                        <action>
                                <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/act
ion-3#addFocus</handlerUri>
                        </action>
                </reaction>
        </objectSynchronization>
</synchronization>
```

この構成の大半はおそらく説明不要だろう。これは典型的な権威あるリソースである。そのリソース上に新アカウントがあり、オーナーがいなければ（状態：unmatched）、ユーザーが新規作成される（アクション：addFocus）。アカウントがリソースから削除されれば（状態：deleted）、ユーザーも削除される（アクション：deleteFocus）。リンクされているべきだが実際はされていないアカウントが見つかった場合は（状態：unlinked）、そのリンク付けが行われる（アクション：link）。唯一説明しておくに値することはリンク付けされた状態に対する対応である。このケースではすべきことはそれほどない。すべてはきちんと整っているように見える。しかしまだ正しく設定されていない属性があるかもしれない。インバウンドマッピングのことは覚えているだろうか？このセクションではインバウンドマッピングについて触れてさえもいなかった。そしてそれにはもっともな理由がある。インバウンドマッピングはこの段階では評価されない。その評価は状態および対応が評価されたあとにのみ発生する。アカウントがすべて正しくリンク付けされ（またはリンク解除され）、インバウンドマッピングが有効なソースおよびターゲットをもてるようにするために、評価は必要である。しかしインバウンドマッピングもアウトバウンド

マッピングも評価はそれ自体では発生しない。midPoint は明示的に設定されていないかぎりデータを変更しない。unmatched、deleted、unlinked の状態に対しては対応がある。この場合は、midPoint がすべてを完全に同期させるものと判断し、よってすべてのマッピングとポリシーが自動で評価される。しかし linked の状態に対しては対応がない。こちらの場合は何も明示的に設定されていないため、midPoint は何もすべきでないと判断する。そこで synchronize プロパティである。たとえ明示的なアクションが設定されていなくとも、このプロパティを使用すれば midPoint に強制的に完全同期を実行させることができる。反対に明示的なアクションが設定されていても、これを使用して完全同期を避けることもできる。



上図はインバウンド同期中の通常のイベントの流れを示したものである。

1. リソースデータベースにアカウントが格納される。
2. 適切なアイデンティティコネクタを使用してアカウントを読み込む。
3. midPoint にアカウントのシャドウが作成される。
4. アカウントのオーナーシップを特定するため相関関係式が評価される（アカウントがすでにユーザーにリンク付けされているかどうか）。
5. アカウントのオーナーシップと状態をもとに同期の状態が判断される。
6. リソース構成をもとに、その状況に対する適切な対応が決まる。
7. アカウント値をユーザーにマップするためインバウンドマッピングが評価される。

このプロセスの内容は分かりやすいよう少し簡略にしていることにご留意いただきたい。またこのプロセスから明らかに外れることもある。例えばユーザーを削除するような場合はインバウンドマッピングが省略される。また、対応に「同期」が含まれていない場合などでもマッピングは省略される。しかし一般的にはインバウンド同期中に発生するのはこのプロセスである。

注 : midPoint は拡張性のあるシステムである。上記の同期対応をいくつかあらかじめ組み立てたものがある。これらの対応は、同期中に発生するほとんどの状態に対処できる。ただ、完全にカスタマイズされた対応ができるようシステムを拡張することもありうる。midPointはそのために設計されたものだからだ。それが理論である。ただ現在のところは、midPoint のこの部分の一部を拡張できるのにすぎない。完全な拡張機能は計画されたものの実装されなかった。よって、同期対応の拡張は可能だが、実際これを実現しようとすると非常に大変であり相当な開発工数が必要かもしれない。しかし別の方法がある。midPoint開発チームは、当初計画したとおりこの拡張機能を是が非でも完成させたいと思っている。ただ、これまで midPoint の顧客は他の機能を優先させてきた。midPoint のサブスクリプション契約者およびスポンサーらは開発への資金援助をしてくれているため、midPointの開発では彼らが優先するものに従わなければならない。そこでもし完全同期対応の拡張性（またはその他の機能）に興味があるようなら、ぜひ midPoint サブスクリプションへの契約あるいは当機能へのスポンサーになることをご検討いただきたい。

同期のタスク

さて、インバウンドの同期の仕組みについては分かった。midPoint がアカウントを読み込むと相関関係が適用され、状態が判断され対応が実行される。しかし本当に最初の段階についてはまだ詳細を説明していない。つまり、実際には midPoint はどうやってアカウントを読み込むのか？という点である。理由がなければ何も起こらない。よって midPointには実際に新アカウント、変更アカウント、削除アカウントを検索するアクティブなコンポーネントがあるはずである。そのコンポーネントとは*同期タスク*である。

midPoint の*タスク*は midPoint サーバー内で稼働するアクティブプロセスである。*タスク*という概念について話すのはこれが初めてであるが、これで最後では全くない。midPointでは様々な目的でタスクが使用される。長時間実行される操作、ワークフロー、多数のオブジェクトを扱うアクション（一括アクション）の追跡に使用される。そしてクリーンアップジョブ、報告、その他さまざまな機能を実行するタスクがある。タスクの概念は非常に強力で柔軟性がある。タスクを使用して短時間の一度限りの操作の実行を追跡できる。また、定期的に予定されているアクションを実行することができる。あるいは、長時間実行されるプロセスの追跡にも使用できる。タスクは本書のほぼすべての章で使用されるだろう。

タスクはほぼすべての同期の仕組みを稼働させるアクティブコンポーネントとして使用される。

- **リコンシリエーション（Reconcilliation）タスク**は特定のリソースからのすべてのアカウントを順にリストアップする。検出されたアカウントごとにリコンシリエーションを実行する。つまり、midPoint は特定のアカウントがどう見えるべきかを計算し、その計算値を実際のアカウント属性と比較する。通常このタスクは定期的

に実行されるようスケジュールされている。ただし実行する間隔は非常に長い（数日間から数週間）。

- **ライブ同期（Live synchronization）タスク** は特定のリソース内の変更を検索（ポーリング）する。このタスクでは、作成されたアカウント、変更されたアカウント、削除されたアカウントを検索する。変更内容を取得し、それを midPoint の同期の仕組みへと受け渡す。このタスクは定期的に実行されるよう必ずスケジュールされており、実行の間隔も非常に短い（数分あるいは数秒の場合もある）。
- **リソースからのインポート（Import from resource）タスク**は特定のリソースからのすべてのアカウントをリストアップする。このタスクはアカウントがまさに作成されたばかりのようにふるまう。通常このタスクは、これらのアカウントをもとにユーザーを作成するよう、あるいはこれらのアカウントを現行ユーザーにリンク付けするよう midPoint に働きかけるものである。このタスクはスケジュールされず、ほぼ必ず手動で実行される。

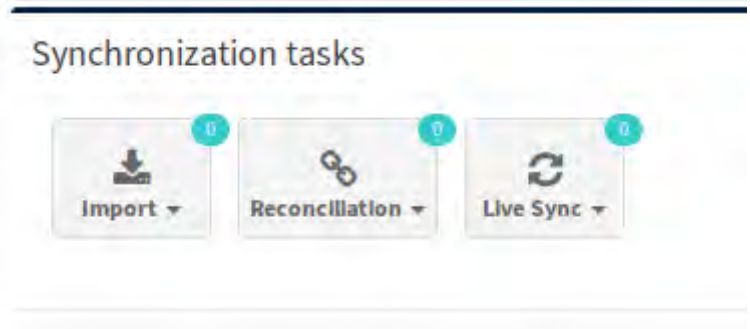
これらの同期タスクは、それぞれ違う仕組みをもとに変更を検出している。しかし、タスクがアカウント（または*デルタ*。後述）を取得した後の処理はどのタスクであってもすべて同じである。すべてのタスクが、同じ構成とポリシーをもとに同じアルゴリズムへとつながる。よって、すべてのはじまりがリコンシリエーションタスクであったのか、またはライブ同期タスクであったのかは問題ではない。結局はすべて同じ、相関関係—状況—対応—マッピングというプロセスになる。

しかし同期を開始するにはタスクが必要である。それはアクティブな部分であり、同期プロセスを開始する引き金である。タスクがなければ同期はうまく機能しない。タスクがなくても同期が「発生」しうる方法はある。例えばユーザーインターフェイス操作への対応として、あるいは、無関係の操作中に新アカウントが検出された場合などである。しかし実際のデプロイでは適切に機能する同期タスクが一つ以上は必要である。このタスクは同期ケースの大半を処理する。

厳密に言えば、タスクはまったく奇妙なものである。タスクは midPoint の他の大半のオブジェクトと同様、自身のデータおよび構成をもっている。しかしタスクはアクティブである。したがってタスクの稼働中はタスクに関連付けられた CPU スレッドがある。タスクの進捗を監視する仕組みがある。midPointのノードのひとつに障害が発生しても別のノードにフェイルオーバーできるよう、クラスター対応である必要がある。タスクは実にリッチであり、取扱うには少しやっかいである。しかし midPoint はタスクの操作を実にシンプルなものにしている。タスクは midPoint の通常のオブジェクトとして表示される。よって他のオブジェクトと同様、midPoint に XML/JSON/YAML 形式でインポートできる。スケジュール変更やパラメータの変更など、XML/JSON/YAML 形式で簡単に編集できる。もちろん、XML/JSON/YAML 形式を使用しても制御できない、タスクだけがもつ特別な機能（中断、再開等）もある。ただタスク管理の大半は、midPoint の他のオブジェクトの場合とまったく同じ方法で行うことができる。

よって XML を取得しそれを midPoint にインポートするだけでタスクは作成できる。そしてこれが往々にして同期タスクを管理する方法である。XML 形式のリソース定義が作成されると、しばしば関連する同期タスクがある。リソースおよび必要とされるすべての同

期タスクはどちらもまとめてインポートできる。ただし同期タスクは midPoint ユーザーインターフェイスからも作成できる。たいてい、同期タスクはリソースの詳細ページにある特殊用途のボタンを使用して作成される。



同期タスクが作成されると、それらはmidPointユーザーインターフェイスの「Server tasks（サーバータスク）」にて他のタスクの場合と同じように管理できる。

同期例：人事データのフィード

本セクションでは人事データをmidPointにフィードする完全な実例を紹介する。ExAmPLE社の人事システムは旧式で複雑である。よって、統合方法としては構造化テキストのエクスポートを使用することが一番簡単である。人事システムは従業員データを毎晩 CSV テキストファイルにエクスポートするよう設定されている。midPointがこのエクスポートファイルを取得しユーザーに関するデータを更新する。

この構成は 3 つの段階で使用される。まず 1 つが、データを midPoint にインポートするためのシンプルな設定である。これは一度だけ実行される操作であり、システム管理者が手動で開始する。するとこの構成は定期的にリコンシリエーションを行うため更新される。リコンシリエーションでは毎回すべてのデータレコードを比較し、必要があれば更新を行う。このケースでは、これは何ら問題はない。さらに最後にこの構成をライブ同期で使用する方法をお見せしよう。

構成のコア部分は単一のリソース定義ファイルに格納されている。以下の段落にてファイルの各部分について説明する。リコンシリエーションタスクおよびライブ同期タスクには構成ファイルがいくつか追加されている。なお、分かりやすいように XML の簡易表記を使用している。midPoint で直接利用できる形式での完全なファイルは、本書に掲載の他のすべてのサンプルと同じ場所に格納されている（詳細は追加情報の章を参照されたい）。

この人事リソースはソースリソースであり、ここからmidPoint内へとデータを「プル（引っ張り出す）」するために使用する。しかし前述したとおり、midPoint ではソースリソースとターゲットリソースに基本的な違いはない。よってこの人事リソースはまったく通常の方法で始まる。CSV コネクタおよびコネクタ構成への参照がある。

```
<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
        <name>HR System</name>
        <connectorRef>...</connectorRef>
        <connectorConfiguration>
                <configurationProperties>
                        <filePath>/var/opt/midpoint/resources/hr.csv</filePath>
                        <encoding>utf-8</encoding>
                        <fieldDelimiter>,</fieldDelimiter>
```

```
            <multivalueDelimiter>;</multivalueDelimiter>
            <uniqueAttribute>empno</uniqueAttribute>
            <passwordAttribute>password</passwordAttribute>
        </configurationProperties>
    </connectorConfiguration>
    ...
```

次のセクションはスキーマ操作構成である。ここはさらに少し興味深い。スキーマ操作セクションには人事アカウント属性を対象とするインバウンドマッピングが含まれる。

```
    <schemaHandling>
        <objectType>
            <objectClass>ri:AccountObjectClass</objectClass>
            <attribute>
                <ref>ri:empno</ref>
                <inbound>
                    <target>
                        <path>$focus/name</path>
                    </target>
                </inbound>
                <inbound>
                    <target>
                        <path>$focus/employeeNumber</path>
                    </target>
                </inbound>
            </attribute>
            <attribute>
                <ref>ri:firstname</ref>
                <inbound>
                    <target>
                        <path>$focus/givenName</path>
                    </target>
                </inbound>
            </attribute>
            <attribute>
                <ref>ri:lastname</ref>
                <inbound>
                    <target>
                        <path>$focus/familyName</path>
                    </target>
                </inbound>
            </attribute>
            ...
```

アカウント属性 empno が midPoint ユーザープロパティ name と employeeNumber にマップされる。アカウント属性 firstname と lastname がそれぞれ givenName と familyName のプロパティにマップされる。これについてはおそらく説明不要であろう。

構成の次の部分では、アクティベーションおよびクレデンシャルに関するマッピングを指定する。

```
            <activation>
                <administrativeStatus>
                    <inbound/>
                </administrativeStatus>
            </activation>

            <credentials>
                <password>
```

```
                    <inbound>
                        <strength>weak</strength>
                        <expression>
                            <generate/>
                        </expression>
                    </inbound>
                </password>
            </credentials>
            ...
```

アクティベーションマッピングは実にシンプルである。midPoint のアクティベーションは非常に特殊な概念である。midPoint はアクティベーション属性およびそれらの意味を知っている。そのため多くの詳細を指定する必要はない。アクティベーションマッピングは、管理ステータスのマップ方向がインバウンドであるべきことを指定するだけである。そしてこれで以上である。

しかしクレデンシャルのマッピングとなると少し説明が必要である。midPoint が人事アカウントとみなしているものは、正確にはアカウントではない。たいていは人事データベース内のレコードにすぎない。誰もこの人事レコードを使用して人事システムにログインすることはない。それゆえこれらに関連するパスワードはないのだ。しかし midPoint のユーザーにはパスワードが必要である。そこでこれからそのパスワードを生成する。このパスワード生成では、前述した generate 式で「弱」のマッピングを使用する。

マッピングが重要なことは疑う余地もない。midPointユーザーにどのようにアカウントデータを反映させるかを指定するものだからだ。ただしアカウントを作成すべきか削除すべきかはマッピングでは指定しない。マッピングはデータの制御はしても、*ライフサイクル*の制御はしない。このリソースを本当に権威あるものにするのが次の構成セクションである。

```
    <synchronization>
        <objectSynchronization>
            <enabled>true</enabled>
            <correlation>
                <q:equal>
                    <q:path>employeeNumber</q:path>
                    <expression>
                        <path>$shadow/attributes/empno</path>
                    </expression>
                </q:equal>
            </correlation>
            <reaction>
                <situation>linked</situation>
                <synchronize>true</synchronize>
            </reaction>
            <reaction>
                <situation>deleted</situation>
                <synchronize>true</synchronize>
                <action>
                    <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#deleteF
ocus</handlerUri>
                </action>
            </reaction>
            <reaction>
                <situation>unlinked</situation>
                <synchronize>true</synchronize>
                <action>
```

```
                        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#link</h
andlerUri>
                    </action>
                </reaction>
                <reaction>
                    <situation>unmatched</situation>
                    <synchronize>true</synchronize>
                    <action>
                        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/mode
l/action-3#addFocus</handlerUri>
                    </action>
                </reaction>
            </objectSynchronization>
        </synchronization>
```

本章の情報があれば、この構成はかなり読みやすいはずである。これは典型的な権威あるリソースが、通常どのように機能するかというものである。そのリソース上に新アカウントがあり、オーナーがいなければ（状態：unmatched）、ユーザーを新規作成する（アクション：addFocus）。新アカウントがあり、そのアカウントの現行ユーザーが見つかれば（状態：unlinked）、それらをリンク付けするだけである（対応：link）。アカウントがすでにリンク付けされているなら（状態： linked）、データを同期させるだけである。実際に他のすべての状態においても同様にデータを同期させる。ただし最後の一つを除いては。アカウントが人事システムから削除されている場合（状態：deleted）、midPointユーザーも削除したい（対応：deleteFocus）。ユーザーが削除されるためデータを同期させても意味はない。midPointはこれを知っているため、マッピングの適用を省略する。

まだリンク付けされていないアカウントのオーナーシップは相関関係式によって特定される。この場合、その式がアカウント属性のempnoとユーザープロパティのemployeeNumberとを比較する。値が合致すればそのユーザーがアカウントのオーナーとみなされる。

このリソースにはもう一つ省略していたものがある。

```
    <projection>
        <assignmentPolicyEnforcement>none</assignmentPolicyEnforcement>
    </projection>
```

これはmidPointの*アサイン*の挙動を調整する設定である。すでに説明したようにmidPointのすべてのリソースは等しく作成される。ソースリソースはターゲットリソースと同じルールに従わなければならない。midPointの基本ルールの一つは、アカウントは特定の存在理由がなければ存在すべきではない、というものである。midPointの用語では、どのアカウントもその存在を正当化する*アサイン*があるからこそ存在している。この手法はまさに（正しく機能する）リソースの大部分に求められるものだが、ソースリソースに必ずしも理想的なものではない。ソースリソースはその逆に機能する。実際に人事アカウントはmidPointユーザーが存在する原因であって結果ではない。よって、アサインの挙動を制御する、assignmentPolicyEnforcementという実に便利な設定がある。この設定は様々なシナリオで使用されるが、多くはデータ移行のためであり、洗練された挙動をまったくしないリソースを制御するために使用される。しかしこのケースでは、当リソースに対するアサインの実行を完全にオフするために使用される。このリソースは権威あるソースのため、アサイ

ンを実行してもあまり意味はない。このリソースの挙動はリソース構成の synchronization セクションにて定義される。

さて、リソース構成を完成させよう。この構成ではコネクタ、マッピング、同期ポリシーを設定する。この構成は全種類の同期、すなわちインポート、リコンシリエーション、ライブ同期に共通であり、これらはすべて同じ設定を使用する。構成ということになると、これらの同期方法で唯一違う点は同期タスクの設定方法である。インポートタスクが設定されると、リソースアカウントのインポートが実行される。リコンシリエーションタスクが設定されると、リコンシリエーションが実行される。それはあくまでタスク内である。そして実際、これらのタスクはユーザーインターフェイスの便利なボタンを使用すれば簡単に設定できる。しかし我々はこの分野で、あえて少し厳しい道をいこうと思う。よって思い切ってタスクを XML 形式でインポートすることにする。

最初のタスクはインポートタスクである。このタスクでは、人事 CSV ファイル内のすべてのアカウントを列挙する。そして各アカウントが今作成されたばかりというふりをする。このタスクが初めて実行されるのであれば、その結果、状態は unmatched または unlinked となる。よって midPoint ユーザーを新規作成するか、アカウントを現行ユーザーにリンク付けする。

```
<task oid=”7c57adc2-a857-11e7-83ac-0f212d965f5b”>
        <name>HR Import</name>
        <taskIdentifier>7c57adc2-a857-11e7-83ac-0f212d965f5b</taskIdentifier>
        <ownerRef oid="00000000-0000-0000-0000-000000000002"/>
        <executionStatus>runnable</executionStatus>
        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/synchronization/task/import/handler-3
</handlerUri>
        <objectRef        oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf"
type="c:ResourceType"/>
        <recurrence>single</recurrence>
</task>
```

これはまさにインポートタスクの基本構造である。midPoint のすべてのオブジェクトと同様、タスクには名前がある。それからタスク管理の内部的な目的で使用されるタスク識別子がある。通常これはタスクオブジェクトの OID と同じである。タスクにはオーナーの定義が必要である。オーナーはタスクを実行しているユーザーである。このユーザーが持つ認可によって実行してもよいタスクが決まるため、これは重要である。また、監査ログに記録されるアイデンティティでもある。このケースでは administrator が当タスクのオーナーである。タスクの実行ステータスは、タスクが稼働中か、中断したのか、または完了しているかを教えてくれる。それからハンドラ URI がある。ハンドラ URI は、タスクが実際に何をやるかを指定する。それは（間接的に）サーバーが実行するコードを参照する。このケースでは、これがリソースからアカウントをインポートする同期タスクであることを、タスク URI が指定する。また、アカウントのインポート元であるリソースは objectRef 参照が指定する。これが我々の人事リソースを指し示す。そして最後が recurrence（繰り返し）である。繰り返しはタスクの実行が single（一度のみ）か、recurring（繰り返し）なのかを指定する。

このタスクの XML 定義が midPoint にインポートされると、サーバーはただちにタスクを実行しようとする。つまり、人事リソースからアカウントのインポートがただちに始まる

ということである。タスクの進捗は midPoint ユーザーインターフェイスのサーバータスクセクションで監視できる。インポートタスクは繰り返し実行されるタスクではない。実行されるのは一回のみだ。タスクを再実行する必要があれば、midPoint ユーザーインターフェイスより行うことができる。しかし midPoint に明示的に知らせない限り、そのタスクが実行されることはない。これが典型的なインポートタスクの仕組みである。通常インポートタスクは新リソースがシステムに接続されると実行される。そしてすべてが設定され、相関され、リンク付けされると、インポートタスクはもはや不要となる。

賢い読者は、インポートタスクがもう一度実行されるとどうなるか？とたずねるだろう。その答えはシンプルである。大した事はない。タスクが、アカウントが作成されたばかりのふりをしたとしても、midPoint は簡単にはだまされない。実際のところ、midPoint がすでにそのアカウントのシャドウをもっていて、それがユーザーにリンク付けされていれば、アカウントが作成されたばかりというのはさすがに信じがたいことだろう？よって midPoint は落ち着きを保ったまま続行するだろう。もしアカウント属性に変更があれば、その変更はユーザーに反映される。しかしそれだけである。特に大きな事は起こらない。

インポートタスクはリソースからのデータを midPoint に取り込む。しかしインポートは繰り返しタスクではないため、データの同期は維持しないし、そうするように設計されてもいない。ただ、他のタスクでこの目的で設計されたものがある。リコンシリエーションタスクはその一つだ。リコンシリエーションタスクはリソース上のすべてのアカウントを列挙し、midPoint のデータと比較する。

```
<task oid="bbe4ceac-a85c-11e7-a49f-0f5777d22906">
   <name>HR Reconciliation</name>
   <taskIdentifier>bbe4ceac-a85c-11e7-a49f-0f5777d22906</taskIdentifier>
   <ownerRef oid="00000000-0000-0000-0000-000000000002"/>
   <executionStatus>runnable</executionStatus>
   <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/synchronization/task/reconciliation/handler-3</handlerUri>
   <objectRef oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf" type="c:ResourceType"/>
   <recurrence>recurring</recurrence>
   <schedule>
      <cronLikePattern>0 0 1 ? * SAT</cronLikePattern>
      <misfireAction>executeImmediately</misfireAction>
   </schedule>
</task>
```

リコンシリエーションタスクの定義はインポートタスクの定義とほぼ同じである。しかし極めて重要な違いがある。まず第一にハンドラ URI が違う。このタスクをリコンシリエーションタスクにしているのがこれである。そしてタスクは繰り返しである。つまり midPoint はこのタスクの実行を繰り返すということである。また実行スケジュールもあるため、サーバーはこのタスクをいつ実行すべきか分かる。リコンシリエーションタスクはたいていリソースを大変消費することから、ごく特定のオフピーク時に実行したいものである。そのため cron ライクなパターンを使用して実行スケジュールを定義する。UNIX に慣れている読者なら、このことはきっとよくご存じだろう。形式は「秒 分 時 日 月 曜日 年」である。よってこのタスクは毎週土曜日の 01:00:00am に実行される。また、ミスファイヤ（失敗）時のアクションについての定義もある。ミスファイヤとは、タスクを実行することになっ

ている時にサーバーがダウンしている状況のことである。よって、土曜日の早い時間帯にサーバーがダウンしていた場合は、当サーバーが起動するとすぐにタスクが実行される。

リコンシリエーションタスクはまさにアイデンティティ管理における主力である。ほぼすべてのリソースで使用でき、かつ信頼性も高い。あらゆる問題の修正、新ポリシーの適用、紛失したアカウントや不正アカウントの検索などに使用されることも多い。実に役立つツールである。ただデメリットもある。リコンシリエーションはすべてのアカウントを反復処理し、アカウントに適用できるすべてのポリシーをアカウントごとに一つずつ再計算する。リソースをかなり大量消費しかねないのだ。ポリシーが複雑でユーザーが多く、リソースが低速であれば、それはもう相当な負担になる。極端な場合は、数時間から数日さえかかることもありうる。しかも小規模なデプロイであってもリコンシリエーションは少しも簡単ではない。問題は midPoint にあるのではない。midPoint は拡張して負荷に対応できる。しかしすべてのアカウントを列挙することは、たいていはリソースに受け入れがたいほどの負荷を与えてしまいかねない。したがってリコンシリエーションは頻繁に実行されるものではない。日次、週次、あるいは月次でのリコンシリエーションが一般的のようである。リコンシリエーションは信頼できるが完全にリアルタイムと呼ばれるものではない。しかしもちろん、midPoint にはより高速の代替手段がある。

ライブ同期はリアルタイム同期のオプションである。もっと正確に言えば、リアルタイム*に近い*同期である。ライブ同期の実際の待ち時間は、数秒または数分の範囲内であり、ほとんどの実際例にとって十分な速さである。また、ライブ同期はリソース効率も非常によい。全体的にみて、リコンシリエーションよりもライブ同期の方が高速かつ軽い。ただ、ライブ同期はすべてのリソースで利用できるわけではない。リソースから最近の変更を取得できるかどうかに依存する。よってそうした変更を記録するリソースにのみ、ライブ同期を利用できる。変更を記録する特定の仕組みはリソース毎に違う。シンプルな変更タイムスタンプ程度の基本的なものもあれば、複雑な変更ログであるものもある。しかし、その仕組みはコネクタが最近の変更を検出できる程度には優れていなければならず、その検出を数秒おきにできる程度には効率的でなければならない。そのような仕組みが利用でき、かつコネクタがその仕組みの使用方法を知っているのであれば、ライブ同期の設定は簡単である。必要なのは同期タスクのみである。

```
<task oid="7c57adc2-a857-11e7-83ac-0f212d965f5b">
   <name>HR Live Synchronization</name>
   <extension>
      <mext:kind>account</mext:kind>
   </extension>
   <taskIdentifier>7c57adc2-a857-11e7-83ac-0f212d965f5b</taskIdentifier>
   <ownerRef oid="00000000-0000-0000-0000-000000000002"/>
   <executionStatus>runnable</executionStatus>
   <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/synchronization/task/live-sync/handler-3</handlerUri>
   <objectRef oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf" type="c:ResourceType"/>
   <recurrence>recurring</recurrence>
   <schedule>
      <interval>10</interval>
   </schedule>
</task>
```

さて、タスク定義は分かりやすくあるべきである。これをライブ同期タスクにする、別のハンドラ URI がある。また、別の種類のスケジューリングもある。我々はこのタスクを特定の時刻に実行するのではなく、定期的な間隔で継続的に実行させたい。このケースではその間隔が 10 秒に設定されている。ライブ同期を実行させ続けるのに必要なのはそれだけである。

midPoint でリコンシリエーションまたはライブ同期などの構成方法を設定することは、まさにタスクを設定することである。構成の残りの部分はどの方法でも同じである。よって同じリソースにライブ同期およびリコンシリエーションの両方を稼働させることは非常に簡単である。2 つのタスクを作成するだけだ。そして実際、これは実に一般的な構成である。変更内容をすばやく効率的に取得するためにライブ同期を使用する。すべての変更が確実に処理され、ポリシーが一貫して適用されていることを確保するためにリコンシリエーションを使用する。

そしてこれで以上である。さて人事データをフィードし稼働させよう。しかしまだいくつか問題がある。賢い読者はきっとお気づきだろう、この人事リソースはあまり良くないと。この人事フィードから作成された midPoint ユーザーは名前と名字を持っている。しかしそのフルネーム項目は空っぽである。ただ心配はいらない。これについては後半の章で*ユーザーテンプレート*を活用して整理する。また、ユーザーは従業員番号をユーザー名としてもっている。実際これはアカウントの名前を変更する必要がなくなるため一部のデプロイには非常によい手法である。ただあまり使い勝手のよい手法ではない。よってほとんどのデプロイで、より便利なユーザー名を生成する必要がある。midPoint でそれを行うのは簡単であり、これについても後ほど説明する。同期を完全に設定するにはまだ学ぶべきことがたくさん残っている。

人事データフィードの推奨

すべてのリソースは midPoint で等しく作成される。ただしほぼ例外なく、ソースリソースは若干特別な立場にある。midPoint の仕組みはどのリソースであっても同じであるが、ソースからのデータはソリューション全体に大きな影響を与えることが多い。コンピュータエンジニアリングの世界には、「*無意味なデータを入力すると、無意味な結果が返ってくる（garbage in, garbage out）*」という昔からの格言がある。データフィードに一つエラーがあるだけで、いたるところにあらゆる問題を引き起こしかねない。よってデータソースをきちんとすることが重要である。そしてたいていはこれが IDM プロジェクトにおける第一段階の一つである。

通常、データソースの設定は反復プロセスである。特にソースデータの品質が不明だと、反復処理が多い場合がある。たいていのプロセスはこのようになる。

1. 持っている情報をもとに最初のースリソースの定義を設定する。コネクタを設定し接続をテストする。アカウントを閲覧できるか確認する。マッピングおよび同期ポリシーを設定する。
2. いくつかの個人アカウントでインポートプロセスをテストする。リソース詳細ページに遷移し、［Accounts（アカウント）］タブをクリックしてアカウントを一覧表示する。アカウントを選択してその隣にある［Context（コンテキスト）］メニュー

から［*Import（インポート）*］（小さな歯車ボタン）を選択する。個人アカウントのインポートが開始される。アカウントは一つだけのため、エラーの確認はより簡単である（ステップ 6 を参照）。

3. インポートタスクを作成しすべてのアカウントのインポートを実行する。
4. タスクエラーがないか確認する。タスク詳細ページでサマリを取得できる。
5. エラーがなければ、次はユーザーを確認する。すべて OK であればおめでとう。インポートはうまくいったわけだ。しかし最初の数回のうちは、なかなかこうはいかないだろう。
6. おそらくシステムログを調べて個々のインポートエラーの詳細を確認する必要があるだろう。midPoint では、エラーの詳細な分析はログに大きく依存している。ログレベルの調整方法およびログメッセージの内容については､本書のトラブルシューティングセクションを参照されたい。
7. エラーの中には､マッピングやポリシーのエラーによって引き起こされたものもあるかもしれない。一般にこうしたエラーは修正しやすい。誤った入力データまたは想定外の入力データが原因でエラーが発生することはほぼ常にあることだ。正しい方法はそのデータを修正することである。しかしそれが必ずしも可能とはかぎらない（実際、ほとんどの場合不可能である）。しかし入力データのエラーの大半は、midPoint にて若干工夫するだけで修正できる。まさにマッピングの力を使うのだ。
8. 同じことを繰り返す。エラーが深刻でなければ、ただインポートタスクを再実行すればよい。これでたいていはうまくいく。ただマッピングにエラーがあると、すべてのデータが完全にくずれてしまう。この場合はおそらく白紙の状態からはじめるのがベストであろう。我々はみな人間であり、特に最初のうちはこのような状況は頻繁に起こるものである。そこであなたを助けてくれる特別な機能がある。Configuration（構成） > Repository Objects（レポジトリオブジェクト）に遷移しよう。コンテキストメニューを開く小さな歯車ボタンがある。［Delete all identities（すべてのアイデンティティを削除）］を選択する。我々は親しみを込めてこのボタンを「ラクサティブボタン」と呼んでいる。短いダイアログがポップアップ表示され、正確にどのアイデンティティ（ユーザー、シャドウ等）を削除するか指定するよう求められる。これは、白紙状態に戻すがすべての構成（リソース、テンプレート、タスク）は保持する非常に便利な方法である。
9. ステップ 2 に進む。終了するまで繰り返す。

IDM デプロイの初期ステップには人事フィードが含まれており､我々はこの人事フィードから開始することを強く推奨する。適度に信頼性のある人事データから midPoint ユーザーを作成し、これに他のリソースを関連付けるほうが通常は簡単である。また、人事システムからのインポートデータを正しく取得するにはたいてい少し手直しがかかってしまう。そのため白紙になるよう midPoint を簡単にクリーンアップできることは、きわめて便利である。ただしこれができるのは、人事フィードが midPoint に接続された最初のリソースである場合のみである。

賢い読者は、ソースフィードが CSV ファイルから取得されると我々が想定していたこと

に気づくであろう。たしかにそのケースが大半である。新入社員や新規請負業者が会社に加わる場合、たいていは急ぐ必要はない。その情報は少なくとも数日前には人事システムに入力されるため、日次での CSV エクスポートでまったく問題ないのだ。しかしもっと迅速なレスポンスが必要なケースもあるだろう。あるいは CSV エクスポートを扱うような、さらなる負荷はかけたくないかもしれない。もちろんソリューションはある。理論上はソースリソースにはどのコネクタでも使用できる。そして実際には人事システムから直接データを取得するのに特化した、人事システム専用のコネクタがある。たとえば、Oracle HCM システム用のコネクタだ。あいにく SAP 人事システムからデータを取得できるコネクタはまだない。

同期およびプロビジョニング

同期とプロビジョニングは密接に結びついている。前章で説明したプロビジョニングに関することは同期にも当てはまる。実際のところ、プロビジョニングおよび同期はまさに基本的な仕組みが同じアプリケーションである。プロビジョニングはユーザーの変更から開始される。同期はもう少し早く始まる。インバウンドマッピングを使用して、ソースシステムからユーザーへと値がマップされる。そしてインバウンドマッピング評価を行った結果として、ユーザーオブジェクトが変更される。midPoint の原則によれば、ユーザーの変更方法は問題ではない。対応は同じである。つまり、アカウントは必要に応じてプロビジョンされるか、変更されるか、削除される。

通常、同期（*インバウンド*処理）およびプロビジョニング（*アウトバウンド*処理）は同一のシームレスな操作の中で行われる。たとえば、人事コネクタが従業員の名字に変更があったことを検出する。この変更が midPoint ユーザーに適用されると、midPoint ユーザーの名字が更新される。この操作はすべてのテンプレート、ロール、アウトバウンドマッピングを評価することで継続する。通常、アウトバウンドマッピングでは名字の変更をリソース属性にマップする。よってユーザーにリンク付けされたリソースアカウントがただちに更新される。これらすべてが単一の操作の中で行われる。それが midPoint のしくみである。midPointは人間ではない。やるべきことを先延ばしにするようなことは絶対にしない。ただちに実行できる操作であれば、それを決して後回しにはしないのだ。常に1回目でデータを整えようとする。よって、もっと旧式の IDM システムにあるような特殊な伝達タスクやプロビジョニングタスクはない。midPoint にはそれらは必要ないからだ。

一つの操作ですべてを行うことで他にもメリットがある。一つの操作であれば、midPoint は詳細をすべて、つまり原因は何だったのか、どんな結果か、正確には何が変更されたのかということを把握する。これはトラブルシューティングの際に重要である。原因と結果を切り離す IDM システムもある。そうやって分割する手法にも何らかのメリットはあるかもしれない。しかし、ある特定の結果がなぜ生じたかエンジニアが知る必要がある場合、これはひどい悪夢となる。一方 midPoint では、原因も結果も単一操作の中でお互いに関係をもたせている。おかげで何が起こっているかを把握するのははるかに簡単である。監査証跡に整然と記録することもできる。そしてもう一つ大きなメリットがある。midPoint はどんな変化があったかを正確に知っているということだ。つまり、midPoint が知っているのはプロパティの新しい値だけではない。古い値、および追加ないしは削除された値も

知っているということだ。これはその変化を完全に説明するものであり、我々はこれをデルタと呼ぶ。これは操作の初めに記録され、その操作が終了するまでずっと伝達される。ゆえにマッピングはスマートであろう。この手法によってあらゆる興味深い挙動パターンが可能になる。たとえば、midPoint による「最後の変更を優先する」ポリシーの実行が非常に簡単になる。この場合、midPoint は操作中に本当に変更される属性のみを上書きするだけである。そして他の値については何もせずにいられる。実際、これが midPoint のデフォルトの挙動である。新しい IDM システムのデプロイ時には、これは非常に有用な挙動である。

操作を慎重に処理することで、旧式の IDM システムでは実現できない構成も可能になる。たとえばソースでありターゲットでもあるリソース、といった具合である。実際に多くの IDM システムがソースでもありターゲットでもあるリソースを持つことができる。ただし、そのリソースがある一つの属性にはソースであり、別の属性にはターゲットである場合に限る。しかし midPoint であれば、ソースとターゲットとで属性が同じリソースとうまくやっていくことができる。実際に同じプロパティに対し一度に多くのソースとターゲットが存在する場合がある。これは確かに非常に有用な構成である。電話番号プロパティについて考えてみてほしい。通常はユーザーが自分で設定するものだろう。これを何らかの特殊なセルフサービスで設定するかもしれない、コールセンターのコールによって更新されるかもしれない、ユーザーが自身のアクティブディレクトリプロファイルで更新するかもしれないなど、情報の変更方法はたくさんある。しかし我々はこのプロパティの整合性を確保したい。電話番号はどこでも同じであってほしい。それがどこで変更されたかは構わない。ただ、どこかで最後に行われた変更をすべての他のシステムに伝達させたい。midPoint ならこれが簡単にできる。同じ属性についてインバウンドマッピングとアウトバウンドマッピングの両方を指定するだけである。

```
<attribute>
        <ref>ri:mobile</ref>
        <outbound>
                <source>
                        <path>$focus/telephoneNumber</path>
                </source>
        </outbound>
        <inbound>
                <target>
                        <path>$focus/telephoneNumber</path>
                </target>
        </inbound>
</attribute>
```

このケースでは、ユーザープロパティの telephoneNumber の変更がアカウント属性 mobile に伝達される（アウトバウンド変更）。しかし、アカウント属性 mobile の変更もユーザープロパティの telephoneNumber へと伝達される（インバウンド変更）。最後の変更が優先される。賢い読者はきっと今ごろ、それでは無限ループになるとぼやいているだろう。ただ心配はいらない。操作はインバウンド部とアウトバウンド部の両方におよぶため、midPoint はいつ停止すべきかを知っている。コネクタ自身が引き起こした変更をコネクタが検出して、コネクタがループを引き起こさないようにする仕組みさえある。midPoint はこのループを自動で停止させる。

情報：同期とプロビジョニングは midPoint にて密接に結びついている。実際にこれら 2 つは、別の方向に適用されたほぼ同じ仕組みである。ではなぜリソース構成には 2 つのセクションがあるのか？なぜ schemaHandling と synchronization があるのか？なぜ 1 つだけではないのか？その答えはシンプルである。歴史だ。1 日で完璧なソフトウェアは構築されない。すべての実用的なシステムと同じように、midPoint も改善を続けながら進化してきた。midPoint は当初非常に優れた設計であった。当時の設計を振り返ってみると、midPoint開発のほぼすべてが正しく予想されその設計の中で説明がつくことは実に明らかである。しかし、たまには過ちがある。midPoint の初期のデータモデル設計では、同期とプロビジョニングの仕組みには大きな違いがあると考えられていた。そしてそのために 2 つのセクションがあった。しかし midPoint の進化版が初期設計よりも向上し、我々は同期とプロビジョニングの仕組みを統合する方法を見出した。ただ互換性を確保したかったため初期のデータモデルはそのまま変更していない。セクションが 1 つではなく 2 つあるのは、単に表面上不完全なだけであって、何ら大きなトラブルにはならない。しかし互換性のない変更は間違いなく midPoint デプロイの整合性に影響するだろう。そして我々はmidPointの整合性とアップグレード性を高く評価している。よって今日まで2つのセクションが残されたままとなっている。ただ今後もずっと残すわけではない。過去の過ちをいつまでも考えるようなことはしない。互換性のない変更を行うのに適切な時がきたら、これら 2 つのセクションを再統合するつもりだ。おそらく、midPoint4.0 をリリースするタイミングでそうなるだろう。

マッピングおよび式のヒントとコツ

マッピングおよび式は共に非常に強力な仕組みを実現する。実際、midPoint 構成の大半は正しいマッピングの設定に関することである。しかし大きな力には大きな責任も伴うため、マッピングは一見すると少しおそろしく思われるかもしれない。ただマッピングと式が少し簡単になるようなヒントやコツがいくつかある。

マッピングの多くはどのコンテキストで使用されるかを知っている。よってソースとターゲットをマッピングするパスを短縮、あるいは完全に省略することもできる。これは人事フィードの例で使用されたアクティベーションおよびクレデンシャルマッピングにおいて明らかである。しかし通常のマッピングパスでさえ短縮できる。たとえば、アウトバウンドマッピングソースを例に考えてみよう。

```
<outbound>
        <source>
                <path>$focus/telephoneNumber</path>
        </source>
</outbound>
```

マッピングは、そのソースがフォーカス（ユーザー）であることを知っているため、定義を短縮してもよい。

```
<outbound>
        <source>
                <path>telephoneNumber</path>
        </source>
</outbound>
```

midPoint の典型的なデプロイでは、数十あるいは数百のマッピングがある。数千のマッピングを伴うデプロイであっても確実に実行できる。そのためそれらのマッピングを保守することは必ずしも簡単ではないだろう。これを簡単にできるものが 2 つある。オプションとしてマッピング名を指定できる能力がある。マッピング名はログファイルおよび一部のエラーメッセージに表示される。これにより、問題を引き起こしているマッピングの特定が簡単になるかもしれない。あるいはログファイルからマッピング実行の痕跡を見つけるのに役立つだろう。マッピングは記述を持つこともできる。この記述は汎用コメントまたはマッピングの文書として使用できる。マッピングが何をするかを説明することができる。

```
<attribute>
    <ref>ri:mobile</ref>
    <outbound>
        <name>ldap-mobile</name>
        <description>
            Mapping that sets value for LDAP mobile attribute based on user's telephone number.
        </description>
        <source>
            <path>$focus/telephoneNumber</path>
        </source>
    </outbound>
</attribute>
```

マッピングはきわめて複雑になりかねない。マッピングには複数行のスクリプト式があるかもしれない。そして何がインプットでアウトプットなのか、必ずしも明らかではないかもしれない。そうしたことから、それぞれのマッピングおよび式はトレース（追跡）を有効にする能力を持っている。

```
<attribute>
    <ref>ri:mobile</ref>
    <outbound>
        <trace>true</trace>
        <source>
            <path>$focus/telephoneNumber</path>
        </source>
        <expression>
            <trace>true</trace>
            <script>
                <code>...</code>
            </script>
        </expression>
    </outbound>
</attribute>
```

トレースが有効である場合は、マッピングまたは式の実行がログファイルに記録される。トレースはマッピングレベルと式レベルの両方で有効にできる。マッピングトレースのほうが短い。こちらはマッピングのインプットおよびアウトプットの概要を提供する。式レベルのトレースはそれよりはるかに具体的である。

ただこのレベルのトレースでさえ、式コードのデバッグには不十分なこともある。そこでロギングのための特別な式関数がある。任意メッセージはスクリプト式コードでログ記録してもよい。

```
<expression>
    <script>
        <code>
            …
            log.info(“Value of foo is {}”, foo)
            …
        </code>
    </script>
</expression>
```

式関数

midPoint のカスタマイズの大半は、一般には式、特にスクリプト式で行われる。スクリプト式は汎用プログラミング言語のどのコードでも実行できる。よってスクリプトはいかようにもデータを変更できるし、あるいはどんな関数でも実行できる。しかしスクリプトにて使用頻度の高いものがある。そこで midPoint では、スクリプトが使用できる役立つ方法が揃った便利なスクリプトライブラリを提供している。

頻繁に使用されるスクリプトライブラリが 2 つある。

- **Basic script library（基本スクリプトライブラリ）**は文字列操作、オブジェクトプロパティの取得等のための基本的な関数を提供している。シンプルかつ効率的な独立型関数であり、どの式でも使用できる。
- **MidPoint script library（midPoint スクリプトライブラリ）**は、IDM 専用ロジックおよび midPoint 専用ロジックを含んだ、よりハイレベルな midPoint 関数へのアクセスを提供する。このライブラリを使用すれば、ほぼすべての midPoint 機能にアクセスできる。ただし、このライブラリの動作が確実でない可能性のある場所がいくつかある（例：相関関係式）。

ライブラリはスクリプトコードから実に使い勝手のよいつくりになっている。ライブラリの呼び出し方法の具体的な詳細はスクリプト言語に依存する一方で、通常、ライブラリは basic シンボルおよび midPoint シンボルを使用してアクセスできる。Groovy スクリプトで基本ライブラリからの関数 norm()は以下のように呼び出せる。

```
<expression>
    <script>
        <code>
            …
            basic.norm('Guľôčka v jamôčke!')
            …
        </code>
    </script>
</expression>
```

JavaScript および Python からのライブラリの呼び出しもほぼ同じであり、賢い読者はこれを難なく理解できると我々は確信している。それよりも難しいのは、ライブラリ内にある関数である。この目的のため、midPoint wiki では 1 ページを使ってすべてのライブラリを列挙しており、また、ライブラリ関数の文書へのリンクも掲載している。

今のところ当セクションで説明したのは 2 つのライブラリだけである。実は話しはこれですべてではない。賢い読者は、前セクションで説明したロギング関数もスクリプトライブ

ラリであることがきっとお分かりだろう。そして将来的にはもっとライブラリが増えるだろう。

リソース機能

midPointに接続するソースシステムおよびターゲットシステムはどれも等しく構築されているわけではない。実際、これらのシステムの機能はかなり異なっている。大半のシステムはアカウントを作成できる。しかしアカウントの削除はこれらすべてができるとは限らない。アカウントを無効にできるだけで、永遠に保持しつづけるシステムもある。アカウントの有効化または無効化すらできないシステムもある。パスワード認証に対応するシステムが大半であるが、そうでないシステムもある、といった具合である。コネクタによっても制限がもたらされるかもしれない。ターゲットシステムがある機能に対応しているとしても、コネクタにその機能を使用するための適切なコードがないかもしれない。midPointは、同期操作およびプロビジョニング操作の実行時にこうしたすべての違いを考慮する必要がある。

midPoint ではシステムおよびコネクタのこうした機能を*リソース機能*と呼んでいる。実際には機能は非常に複雑かもしれないが、基本的にはコネクタおよびリソースが実行できるものごとのリストにすぎない。midPoint はリソース機能を分っている。よってデータを正しく提示できる。例えば、読み取り専用のリソース上ではアカウントの変更は行わない、といったことである。

一般に機能は midPoint が自動で検出し、それぞれはただ動作するだけである。機能維持のための追加処理は普通はない。しかしときには機能を微調整せねばならないことがある。コネクタがリソース機能を十分に検出できないかもしれない。読み取り専用のリソースがあるがコネクタがそれを知る方法がないかもしれない。よって midPoint にて書き込み機能を手動で無効にしなければならない。こうした理由から、次の 2 セットの機能がある。

- **ネイティブ機能（Native capabilities）**はコネクタが検出する機能である。必ず midPoint で自動生成されるものであり、管理者はこれらの機能を変更してはならない。
- **構成済み機能（Configured capabilities）**は管理者が修正した機能である。構成済み機能は、ネイティブ機能を上書きするために使用される。構成済み機能は通常は空っぽである。つまりネイティブ機能のみが使用されているということである。

管理者が機能を微調整できる方法は数多くある。しかし同期およびプロビジョニング上、特に興味深いものは一つのみである。それは疑似アクティベーションである。

midPoint のコネクタは、企業のカスタムアプリケーション用のコネクタなど、特定のシステム向けに専用に適応させることができる。スペクトラムの反対側は、幅広いシステムやアプリケーションに合わせられる汎用コネクタである。LDAP、CSV、データベーステーブルはそうした汎用コネクタの例である。これらは大変有用でほぼすべての midPoint デプロイで使用されている。しかしデータベーステーブルまたは CSV ファイルのアカウントを無効にする方法は標準化されていない。アカウントのアクティベーションステータスを表す様々なカラムや値が使用されている。さらに LDAP ディレクトリ内のアカウントを無効する方法も標準化されていない。しかし midPoint はアカウントが無効か有効かを知ること

でかなりのメリットを得ている。よって、アカウントのアクティベーションステータスを示すのにどの属性とどんな値が使用されているかを midPoint に教える方法がある。構成済み機能を使用するのはそのためである。

```
<capabilities>
        <configured>
                <cap:activation>
                        <cap:status>
                                <cap:attribute>ri:active</cap:attribute>
                                <cap:enableValue>true</cap:enableValue>
                                <cap:disableValue>false</cap:disableValue>
                        </cap:status>
                </cap:activation>
        </configured>
</capabilities>
```

上記の構成済み機能では、アカウントアクティベーションステータスを制御する属性としてリソース属性の active を指定している。この属性が true 値に設定されていれば、アカウントは有効である。この属性が false 値に設定されていれば、アカウントは無効である。これで以上である。この構成済み機能がリソース定義の一部になると、midPoint はリソースがアカウントを有効・無効化できるかのようにふるまう。アカウントを無効にする試みは、ユーザーに意識させることなく属性 active の変更へと変換される。しかしそれは逆方向にも機能する。アカウントの属性 active が false 値に設定されていると、midPoint はそのアカウントを無効として表示する。これを実現するためにロジックやマッピングを追加する必要はない。この機能がそれをすべて行うからだ。

同期例：LDAP アカウントの相関関係

前述の例では人事フィードに対する同期の使用について説明した。そこでは同期の仕組みを使用していることが明確であった。しかし midPoint の同期は、データを midPoint にただフィードするだけよりもはるかに柔軟性がある。同期は典型的なターゲットリソースに対してさえも使用できる。その場合、同期の目的は複数あることが多い。

- **初回移行（Initial migration）**：新リソースを midPoint に接続するプロセスである。通常、リソースを midPoint に接続した時点ですでにリソースにはアカウントが存在している。少なくともそうしたアカウントの一部は midPoint に存在しているユーザーに該当する（例：人事フィードから作成されたユーザーなど）。よってリソースからのアカウントは midPoint にすでに存在するユーザーと相関関係をもたせる必要がある。同期はまさにこれにうってつけの仕組みである。
- **不正アカウントの検出（Detection of illegal accounts）**:一般に、セキュリティポリシーは特定のリソースにアカウントを必要とする人々だけが、そのリソースにアカウントを持つように設定される。これは*最小特権の原則*として知られている。しかし典型的な IDM デプロイでは、システム管理者が自由にアカウントを作成するのを禁止するものは何もない。これがしばしば望まれるのは、緊急時にはシステムを完全に制御することが重要なためである。しかしたとえ緊急時であっても、その緊急事態が終わった際には状態がポリシーに沿っていることを確かめたい。midPoint では定期的にターゲットシステムをスキャンすることでこれを簡単に行うことができる。同期の仕組みを使用すれば、正当性のないアカウントを検出しこれを削除ま

たは無効化できる。ここでも同期の仕組みによってこれを簡単に行うことができる。

- **属性値の同期（Attribute value synchronization）**：通常、ターゲットリソース内のアカウントは midPoint がプロビジョニングアクションを行ったことで作成される。しかしアカウントの属性値は、実際は midPoint のデータのコピーである。属性値はシステム管理者が簡単に変更できたり、データ復旧手続き中に古い値に設定されるなど、あらゆる方法で同期から外れてしまいかねない。midPoint はこうした属性が同期され、長期間同期がとれた状態を確保できる。この目的に、同期の仕組みはうってつけである。

旧式の IDM システムで同期が行われた目的は、ソースリソースから IDM システムにデータを取得することがほとんどであった。しかし midPoint における同期はそれよりもはるかに強力である。同期はソースシステムおよびターゲットシステムに適用でき、データをプル（引っ張り出す）し、プッシュ（突っ込む）し、矛盾を検出しこれを修正できる。同期は多目的な仕組みである。ここでふたたび再利用の原則である。同期の仕組みはあらゆる目的のために再利用できるのだ。

この例では、同期を使用して既存の LDAP サーバーを midPoint に接続しようとしている。midPoint はすでに人事システムに接続しているものとする。人事データはすでにインポート済みである。今やすべての従業員が midPoint ユーザーである。そしてこの LDAP サーバーがある。これは実に重要な LDAP サーバーである。当サーバーは会社のイントラネットポータルだけでなく、より小さなあらゆる Web アプリケーションでも使用されている。これらのアプリケーションは、ユーザー認証およびアクセス認可にこの LDAP サーバーを使用している。LDAP サーバーは数年前にデプロイされた。最初に人事データが取り込まれた。しかしそれからずっとこの LDAP サーバーはシステム管理者が手動で管理していた。そのため、元従業員に属するアカウントも一部あると思われる。また、アカウントが行方不明になることもあったかもしれない。そして誤ったデータをもつアカウントがある可能性は非常に高い。

最初のタスクはこのリソース用にコネクタを設定することである。LDAP サーバーは常にアイデンティティ管理目的で使用されるため、midPoint には非常に優れた LDAP コネクタが付属している。必要なのは、そのコネクタを使用できるようリソースを設定するだけである。

```
<resource oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c">
    <name>OpenLDAP</name>

    <connectorRef type="ConnectorType">
        <filter>
            <q:equal>
                <q:path>connectorType</q:path>
                <q:value>com.evolveum.polygon.connector.ldap.LdapConnector</q:value>
            </q:equal>
        </filter>
    </connectorRef>
```

ここで我々が見ているのは、コネクタを参照するやや洗練された方法である。これまで見てきたのは OID による直接参照のみであった。これは midPoint のほぼすべての参照にうまく機能する。OID が決して変わらないからである。しかしコネクタが若干扱いにくい。コネクタが検出されると、コネクタを表示するオブジェクトが midPoint によって動的に

作成される。よって OID はその時に無作為に生成される。システム管理者がその OID を予想できるような実用的な方法はない。それでも我々は、リソース定義のインポート時にその定義が特定のコネクタを参照するようにしたい。そこでオブジェクト参照を指定する別の方法がある。この方法では OID の直接参照ではなく検索フィルタが使用されている。このリソース定義が midPoint にインポートされると、midPoint は検索フィルタを使用して LDAP コネクタを探す。コネクタが見つかるとそのコネクタの OID が参照（connectorRef）に置かれる。次回は midPoint がこのリソースを使用するため、直接 OID をなぞることができる。これは非常に便利な方法である。しかし幾分制限もある。まず、フィルタはインポート中にしか解釈されない。つまり一度しか解釈されないということである。インポート時にコネクタがなければ、参照を手動で修正する必要がある。2 つ目にこの手法がうまくいくのは、midPoint にデプロイされた LDAP コネクタが一つのみの場合である。大体はそうである。しかしコネクタフレームワークには同じ種類で違うバージョンの複数のコネクタを含めることができる。これはコネクタの段階的なアップグレードやコネクタの新バージョンのテストなどに非常に役立つ機能である。しかし、検索フィルタが複数のオブジェクトと一致した場合、インポートは失敗してしまう。この場合、コネクタ参照は手動で設定しなければならない。

いったん LDAP コネクタへの正しい参照をもってしまえば、あとはコネクタが LDAP サーバーにアクセスできるようリソースを構成するだけでよい。

```
        <connectorConfiguration>
                <icfc:configurationProperties>
                        <cc:port>389</cc:port>
                        <cc:host>localhost</cc:host>
                        <cc:baseContext>dc=example,dc=com</cc:baseContext>
                        <cc:bindDn>cn=idm,ou=Administrators,dc=example,dc=com</cc:bindDn>


<cc:bindPassword><t:clearValue>secret</t:clearValue></cc:bindPassword>
                        ...
                </icfc:configurationProperties>
                <icfc:resultsHandlerConfiguration>
                        <icfc:enableNormalizingResultsHandler>false</icfc:enableNormalizingResultsHandler>
                        <icfc:enableFilteredResultsHandler>false</icfc:enableFilteredResultsHandler>
                        <icfc:enableAttributesToGetSearchResultsHandler>false</icfc:enableAttributesToGetSearc
hResultsHandler>
                </icfc:resultsHandlerConfiguration>
        </connectorConfiguration>
        ...
```

これはすべて、本書にて既出の他のリソースの構成と非常に似ている。おそらく結果ハンドラの構成を除けば、その構成はまったく説明不要であるはずだ。*結果ハンドラ*は ConnId コネクタフレームワークに付随する小さなヘルパーである。当ハンドラの目的は、より簡易なコネクタが検索結果を絞り込み、事後処理を行うのをサポートすることである。しかし LDAP コネクタは普通の簡易コネクタではない。結果ハンドラのような厄介なものからのサポートがなくても何でもできる、成熟した多機能コネクタである。ここでは ConnId 結果ハンドラは一切値を追加しない。これらは実際には有害ですらありえる。LDAP プロトコルは、一部の注意すべき例外を除いて、LDAP データのほぼすべての側面に適用され

る大文字小文字の区別といった特徴をもっている。コネクタはこうした特徴を知っているが、ハンドラは知らない。それゆえ、ハンドラがオン（デフォルト設定）になっていると、邪魔をしてデータを台無しにしかねない。そのため多機能コネクタを使用するときは、ハンドラを明示的にオフにすることを常に強く推奨する。

注: 本書に掲載の他のすべての例と同様、上記の XML 例は 分かりやすいよう簡略化した省略形になっている。この形式の例を midPoint にインポートすることはできない。インポート可能な完全な例は本書に付属することになっているファイルにある。追加情報の章を参照されたい。

上記の基本的なリソース構成はリソースへ接続するのに十分である。そのため、リソース詳細ページのテスト接続ボタンはうまくいくはずである。この構成はアカウントを列挙するのにも使用できる。しかし、LDAP サーバーは多くのオブジェクトクラスに対応しており、midPointはどのオブジェクトクラスがアカウントを表すのかをまだ知らない。そこで、我々のリソースにはスキーマ操作セクションが必要になる。

```
<schemaHandling>
    <objectType>
        <kind>account</kind>
        <displayName>Normal Account</displayName>
        <default>true</default>
        <objectClass>ri:inetOrgPerson</objectClass>
        <attribute>
            <ref>ri:dn</ref>
            <displayName>Distinguished Name</displayName>
            <limitations>
                <minOccurs>0</minOccurs>
            </limitations>
            <outbound>
                <source>
                    <path>$focus/name</path>
                </source>
                <expression>
                    <script>
                        <code>
                            basic.composeDnWithSuffix('uid',                    name,
ou=people,dc=example,dc=com')
                        </code>
                    </script>
                </expression>
            </outbound>
        </attribute>
        ...
```

inetOrgPerson オブジェクトクラスには、LDAP の必須属性ごとにアウトバウンドマッピングがあるはずである。これらのマッピングはターゲットリソース定義の典型である。いったんこれを行ってしまえば、midPoint に都合よく LDAP を列挙できるようになるはずだ。ただし［Repository（レポジトリ）］ビューではなく［Resource（リソース）］ビューに切り替える必要がある。アカウントは LDAP サーバーに格納され、midPoint はそれらにアクセスできる。よってアカウントは［Resource（リソース）］ビューに一覧表示される。しかし midPoint はまだこれらのアカウントを処理していない。つまり midPoint レポジトリにはアカウントシャドウがないことになる。［Repository（レポジトリ）］ビューに何も表示されないのはそのためである。しかし今からそれについてあることを行う。

それは、リソースアカウントをインポート（またはリコンシリエーション）することである。しかし今これを行おうとしても、実際は何も起こらないだろう。アカウントがユーザーにリンク付けされていないため、midPoint は属性を同期しない。そして midPoint はアカウントに何かをするよう指示されてはいない。したがって midPoint は何もしない。それが midPoint 原則の一つである。midPoint は明確に指示がないかぎりアカウント変更は行わない。何もしないのが midPoint のデフォルト設定である。データを台無しにしてしまうよりは何もしないほうがいい。

アカウントがインポートできるようになる前に、まずこのリソースに対する同期構成を設定する必要がある。LDAP サーバーには、midPoint にすでに存在するユーザーに属すべきアカウントがある。それらをリンク付けさせたいが、手動では行いたくない。これを自動で行う相関関係式を設定するほうがよい。

```
<synchronization>
    <objectSynchronization>
        <objectClass>ri:inetOrgPerson</objectClass>
        <kind>account</kind>
        <intent>default</intent>
        <focusType>UserType</focusType>
        <enabled>true</enabled>
        <correlation>
            <q:equal>
                <q:path>employeeNumber</q:path>
                <expression>
                    <path>$account/attributes/employeeNumber</path>
                </expression>
            </q:equal>
        </correlation>
```

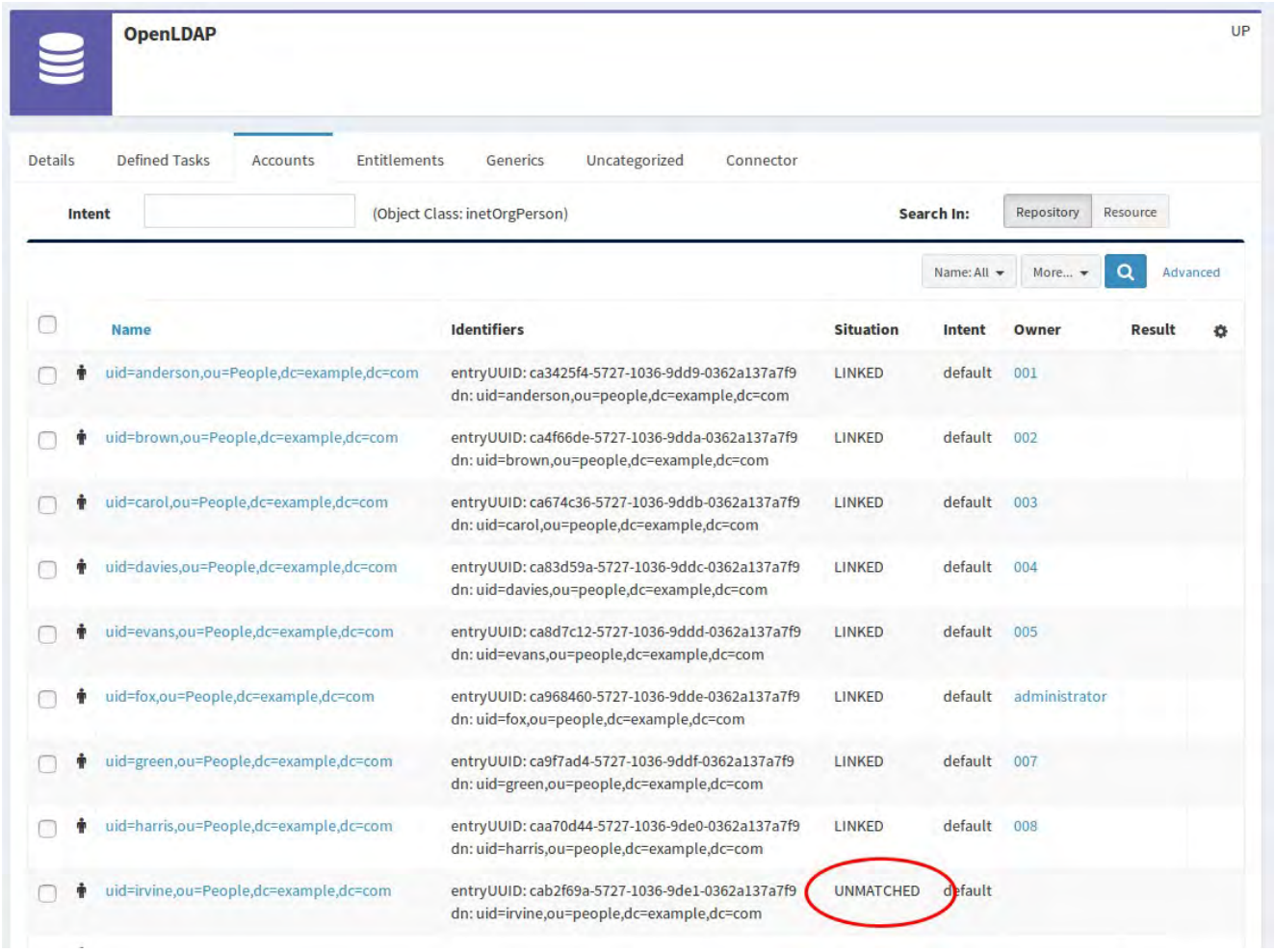
この相関関係式はアカウント属性の employeeNumber と、同じく employeeNumber という名前のユーザープロパティを照合する。簡単に言えば、アカウントとユーザーの従業員番号が一致すれば、それらはリンク付けされるべきと判断する。このケースでは、同期の状態がunlinked（リンク付けされるべきであるがまだされていない状態）であるとmidPointは判断する。すると我々は midPoint にアカウントのリンク付けをさせたいため、適切な対応を定義する。

```
        <reaction>
            <situation>unlinked</situation>
            <synchronize>true</synchronize>
            <reaction>
                <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#link</handlerUri>
            </action>
        </action>
```

上記はオーナーが見つかるアカウントを対象にしたものである。ではその他のアカウントはどうするのか？今は何もしない。よって他の対応は一切定義する必要はない。これはいくぶん意外なことかもしれない。不正アカウントはもちたくない、だからきっとunmatched（不一致）の状態のアカウントは削除する対応を定義したくなるだろう。これはよい手法だが、ここではまだ早すぎる。unmatched だからといってそれらのアカウントをまだ削除したくない。employeeNumber 属性の紛失または入力ミスであるだけで、完全に正当な

アカウントもあるかもしれないからだ。よってとりあえず今はこの対応一つだけにしておく。

さて、今こそインポートタスクまたはリコンシリエーションタスクを開始する適切なタイミングである。タスクが終了すると状況はこのようになる。



LDAPサーバーにはかなりよいデータがあったようだ。アカウントのほとんどが、そのオーナーと適切に相関がとられリンク付けされた。しかし相関がとられなかったアカウントもわずかながらある。これらのアカウントは結局 unmatched の状態となってしまった。これを解決するには、これらのアカウントを該当するユーザーに手動でリンク付けすればよい。unmatched となったエントリの隣にある歯車ボタンをクリックしてコンテキストメニューから［Change owner（オーナー変更）］を選択する。ダイアログが表示されるので適切なユーザー（Isabella Irvine）を選択する。するとアカウントがそのユーザーにリンク付けされる。このプロセスを繰り返しながら unmatched の状態にあるアカウントをすべてリンク付けする。

上のスクリーンショットには一つ興味深いことがある。uid=carol と指定されている LDAP アカウントをみてほしい。他のアカウントのほとんどが、uid 値がユーザーの名字から取られているのに対し、このアカウントは例外である。この uid が明らかに間違っているとしても midPoint はそのアカウントを正しいユーザー（Carol Cooper）にリンク付けして

いる。これは、相関関係には employeeNumber を使用するよう midPoint を設定しておいたためである。その結果、ユーザー名が決まりごとに違反しているアカウントでさえ、そのオーナーに自動でリンク付けできるのだ。ただし、相関関係に使用可能な信頼できる情報がある場合に限るが。

すべてのアカウントが各オーナーにすべてリンクされたときが、同期ポリシーを完了させる適切なタイミングである。ここで、まだ unmatched のアカウントがあれば midPoint にそれを無効にするよう指示することができる。LDAP サーバーに不正アカウントが作成されている場合もそうである。また、LDAP サーバーから削除されたアカウントのリンクを解除するよう指示することもできる。

```
<reaction>
    <situation>unmatched</situation>
    <action>
        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#deleteShadow</handlerUri>
    </action>
</reaction>
<reaction>
    <situation>deleted</situation>
    <action>
        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#unlink</handlerUri>
    </action>
</reaction>
```

LDAP サーバーには誤った属性値をもつアカウントがあるかもしれない。ここでいう「誤った」とは、アウトバウンドマッピングが計算する値と違うという意味である。しかし midPoint はまだその値を修正しない。明示的に指示しないかぎり、アカウントを変更しないという midPoint の原則は覚えているだろうか？これらのアカウントは linked（リンク済み）の状態である。そしてこの状態に対しては何の対応も設定していない。よってここで midPoint に値の同期をとるよう指示する必要がある。

```
<reaction>
    <situation>linked</situation>
    <synchronize>true</synchronize>
</reaction>
```

ここで賢い読者はきっと、我々は何か忘れているのでは？と思うことだろう。たしかにそうだ。属性値はリコンシリエーションプロセスを実行して同期がとられる。しかしアウトバウンドマッピングはリコンシリエーションでは機能しない。強度について明確に定義されていないため、midPoint は *normal（普通）*の強度だと判断する。これらのマッピングは「最後の変更を優先する」戦略を実行することになっている。よって、最後に変更されたのがアカウント属性なのかユーザープロパティなのかを、midPoint が知らないため、リコンシリエーションはアカウントデータを上書きできない。midPoint は知らない場合は何も行わないのだ。我々はデータを台無しにしたくない。そのため今必要なことは、我々が本当にしたいこと、つまり midPoint にマッピングの強度が本当は *strong（強）*であると知ってもらうことである。

```
<attribute>
    <ref>ri:cn</ref>
    <displayName>Common Name</displayName>
```

```
            <limitations>
                <minOccurs>0</minOccurs>
                <maxOccurs>1</maxOccurs>
            </limitations>
            <outbound>
                <strength>strong</strength>
                <source>
                    <path>$focus/fullName</path>
                </source>
            </outbound>
        </attribute>
        ...
```

賢い読者は再び気になるだろう。この limitations とは何なのか？と。簡単に言えば、この limitations（制限）では属性がオプション（minOccurs=0）であり、単数値（maxOccurs=1）であることが指定されている。しかし、midPoint はスキーマを完全に認識せず、一切自身では解決しないことになっているのか？そのとおりである。この limitations 要素を使用してスキーマからの情報を上書きする必要があるのはこのためである。属性 cn は LDAP スキーマにて必須と指定されている。しかし我々は今この属性についてアウトバウンドマッピングを指定したばかりである。それゆえ、midPoint ユーザーが属性 cn の値を提供しない場合であっても、この式を使用してその値を判断できる。よって LDAP スキーマが属性 cn を必須と指定しているとしても、midPoint ではこの属性をオプションとして提示したい。そこでこの minOccurs 制限である。LDAP の特徴に精通している人であればおそらく maxOccurs 制限はすぐにおわかりだろう。LDAP の世界では、デフォルト設定はすべて複数値になっている。そのため、アカウントの識別子と名前に一般的に使用される属性さえも複数値である。ただ、それらを本当に複数値属性として使用する人はいない。アプリケーションが cn 属性で 2 つの値に遭遇していたら、アプリケーションの大半はおそらく破裂してしまうだろうからだ。しかし LDAP スキーマでは、これらの属性は正式に複数値と定義されており、midPoint はそれを LDAP コネクタから取得する。maxOccurs 制限はそのスキーマを上書きし、midPoint にこの属性をあたかも単数値であるかのように扱うよう強制する。

そしてこれで以上である。これで、LDAP サーバーを監視し、規定からはずれた属性値を修正し、不正アカウントは削除するようリコンシリエーションタスクをスケジュールに組み込むことができる。このようにして、純粋なターゲットリソースの場合であっても同期タスクが役立つのである。

しかし最後に一言注意を述べておく。これらのアカウントは同期がとられ既存の midPoint ユーザーにリンク付けされた。アカウントは midPoint によって作成されたものではない。よって midPoint にはこれらのアカウントが存在すべきだと言うものがなにもない。midPoint の用語でいうと、これらのアカウントに対する *アサイン* が一切ないということである。midPoint はポリシーと現実とを明確に区別する。よって midPoint はこれらのアカウントが存在していることは分かっているものの、その存在を正当化するポリシーステートメントはない。デフォルト設定では、midPoint は何もせずアカウントの存在をそのままにしておく。同じく、アサインにて明示的な変更がある場合のみアカウントは作成または削除される。しかしこれは繊細な状況である。midPoint の管理者が注意していないと、リンク付けはされているがアサインされていないアカウントは簡単に削除されてしまいかねない。

もちろんそのような状況に対応する方法もある。その一つはリンクと共にアサインを作成することである。この方法に興味があれば、midPoint wiki にて「legalize（正当化）」でキーワード検索してほしい。ただ、もっとよい対応方法は他にもいくつかある。おそらく最適な方法がロールを利用することだろう（ロールベースアクセス制御：RBAC）。これは次章で説明する。その前にまだ同期について知っておくべきことがいくつかある。

リコンシリエーション

リコンシリエーション（Reconciliation）は、アカウントの実際の状態（現実）とあるべき状態（ポリシー）とを比較するプロセスである。アカウントの比較だけでなく、不整合があればその調整も行う。よってリコンシリエーションはリソース上の誤ったデータを修正することができる。しかしそれは逆方向にも機能する。midPoint 内のデータも修正できるのだ。つまりリコンシリエーションはアイデンティティ管理ツールボックスの中で最も役立つツールの一つなのである。

リコンシリエーションは midPoint にて様々な方法で使用できる。一人の特定のユーザーについて、midPoint のユーザーインターフェイスを使用することでリコンシリエーションを開始できる。そうすると midPoint は、全ユーザーのアカウントの値を、マッピングを使用して計算された値と比較する。差異があればアカウント値を修正する。これは単一ユーザーのみのリコンシリエーション設定を試すのであれば完璧な手法である。また、一人の特定ユーザーの値を修正するのにも役立つ機能である。

特定のユーザーを対象としたリコンシリエーションであれば役立つだろう。しかしこれはその場しのぎの手法である。しかしアイデンティティ管理においては、通常はシステミックな手法のほうが好まれる。よってリコンシリエーションは調整タスクの形で使用できる。リコンシリエーションタスクはリソース上のすべてのアカウントを列挙し、各アカウントに対し一つずつ調整を行う。このようにしてすべてのリソースアカウントの同期が確保される。

リコンシリエーションについては、少し意外と思われかねないことがいくつかある。まず 1 つが、アカウントのリコンシリエーションによってユーザー変更が発生する可能性があるということだ。これは当アカウントにインバウンドマッピングがある場合に発生するが、これはおそらく極めて予想されることである。しかしもしユーザーが変更されると、その変更はたいていはアウトバウンドマッピングを経由して他のアカウントに伝わるだろう。midPoint は引き延ばしにすることを嫌うため、そうした変更はただちに実行しようとするだろう。よって一つのアカウントが調整されることで他のアカウントにも変更が発生する可能性がある。2 つ目に、リコンシリエーションは強度が normal（普通）のマッピングをスキップするということである。その理由はすでに説明したが、これは経験豊富な midPoint エンジニアでさえ時に驚かされることである。マップされた値をずっとアカウント内に確実に存在させたい場合には、マッピングは強であることが正しい選択である。

すでに midPoint のユーザーインターフェイスを探索した好奇心の強い読者なら、きっと「再計算（recompute）」機能にお気づきだろう。再計算が行うことはリコンシリエーションとほぼ同じのように見える。しかし微妙な違いがある。再計算はアカウントデータの取り込みを強制しない。このケースでは、midPoint が計算のためどうしてもアカウント属性を必要とする場合のみ、リソースから取り込まれる。通常これは、*弱*または*強*のマッピングが使用される場合のみ発生する。しかしもし*普通*のマッピングしかない場合は、再計算

ではアカウントデータは読み込まれない。midPoint は、このプロセス中にリソースから取り込まれるアカウントについてのみ、アカウント属性値を比較し修正する。これが再計算の仕組みである。再計算の目的は midPoint *ユーザー*のデータを修正することである。アカウントデータの修正は多少なりともまさに再計算の副作用である。一方、リコンシリエーションは計算に必要か否かに関係なく、すべてのアカウントを読み込もうとする。そのためすべてのアカウントのすべての属性が修正される。これがリコンシリエーションの目的、つまりアカウントデータの修正である。

再計算*タスク*とリコンシリエーション*タスク*にはほかにも違いがある。再計算タスクの目的はユーザーデータの修正である。よって再計算タスクは midPoint ユーザーに対して処理を繰り返す。そのためリソース上の新アカウントは検出せず、アカウントが削除されても見過ごしてしまうかもしれない。しかしリコンシリエーションタスクは違う。実際にリコンシリエーションタスクにはいくつか段階がある。第一段階ではすべてのリソースアカウントを列挙する。各アカウントのオーナーを判断し、属性を比較し修正する。しかしこのプロセスではリソース上の実際のアカウントに対し処理を繰り返すため、新アカウントも検出できる。第一段階が終わると、第二段階が始まる。第二段階は midPoint 内に保存されているアカウントシャドウを確認する。このタスクでは、第一段階で処理されなかったシャドウを検索する。これらは少し前にはリソース上に存在していたが、その後消えてしまったアカウントである。そのようにしてリコンシリエーションは削除されたアカウントを検出する。

デルタ

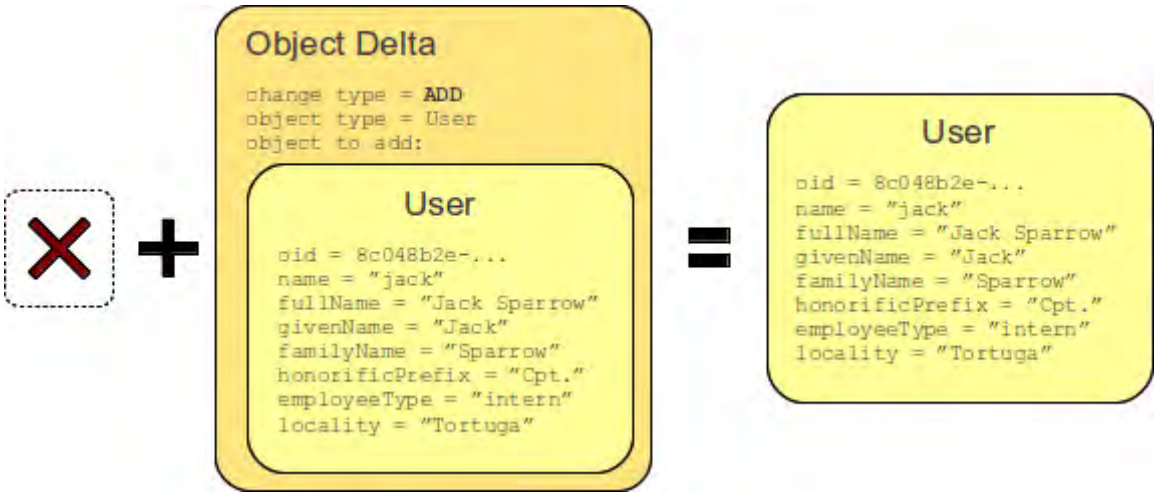
リコンシリエーションは実に有効な仕組みである。信頼性が高く綿密であるが、極めて低速でもあり、計算リソースおよびネットワークリソースを非常に消費する。リコンシリエーションがなぜこのように重いかには理由がある。リコンシリエーションは*完全（absolute）な状態*のアカウントを処理する。つまり、すべての属性のすべての値をもつアカウントをすべて読み込むということである。その後すべてを再計算する。変更のなかった属性および値さえもである。これは従来からある信頼性の高い計算方法であり、旧式のアイデンティティ管理システムの大半でみられる仕組みでもある。

しかしより良い方法もある。変更された属性が一つだけと知っていれば、その属性だけを再計算できる。それ以外の属性は気にする必要はない。そして属性 foo がそのように変更され、新値 bar があることを知っていればさらによい。必要なのはその値 bar を再計算するだけで他は一切気にしなくてよいからだ。これが我々の言うところの*相対的変更（relative change）*である。何が変更されたかに関心があるだけで、変更のない属性および値についてはあまり気にしない。このようなことが midPoint の内部で行われている。midPoint は*相対主義*である。

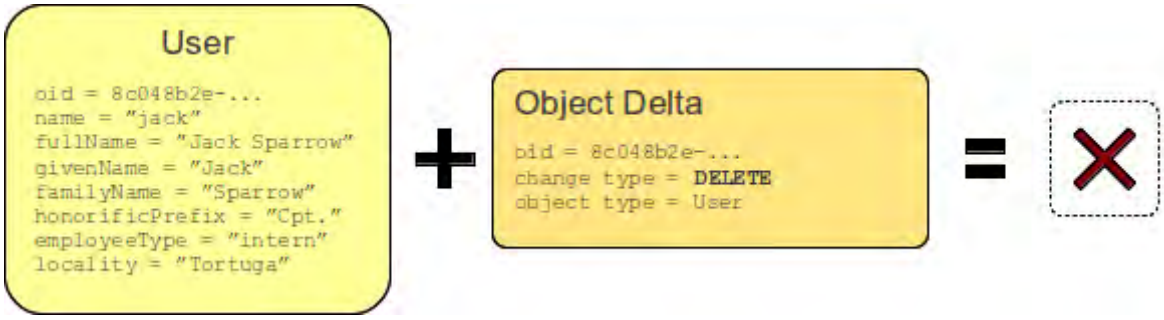
ここで*デルタ*が登場する。デルタとは midPoint の単一オブジェクトの変化量を表すデータ構造である。*追加デルタ（Add delta）*は、midPoint の新オブジェクトがまさに作成されようとしていることを表す。*変更デルタ（Modify delta）*は midPoint の既存オブジェクトでプロパティの一部が変更になったことを表す。*削除デルタ（Delete delta）*はオブジェクトが削除されようとしていることを表す。これは実に強力な仕組みである。思い出して

みてほしい、midPoint 内のものはすべて（ユーザー、アカウント、リソース、ロール、セキュリティポリシーなど）オブジェクトとして表すことができることを。つまりデルタはどんな変化でも表すことができる。その変化とは、ユーザーパスワードの変更、アカウントの削除、コネクタ構成の変更、あるいは新パスワードポリシーの導入かもしれない。これらすべての変化を統一した方法で表すことができるなら、それらを統一した方法で扱うこともできるということだ。midPoint にとって構成変更も含めすべての変更を監査証跡に記録することは簡単なことだ。どの変更も承認プロセスを簡単に経由させることができる。こういった具合である。midPoint は変更を取扱うための比較的シンプルな仕組みを作成できる。そして作成された仕組みは（ほぼ）すべてのオブジェクトの変更に適用することができる。

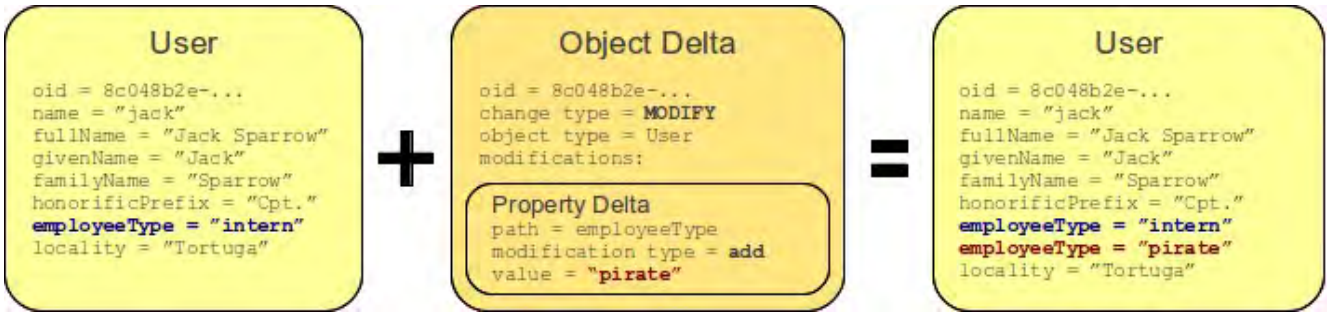
デルタの構造をもっと詳しくみてみよう。デルタには*追加*、*変更*、*削除*の 3 つの種類がある。*追加デルタ*は実にシンプルである。これには新規作成されるオブジェクトが含まれる。



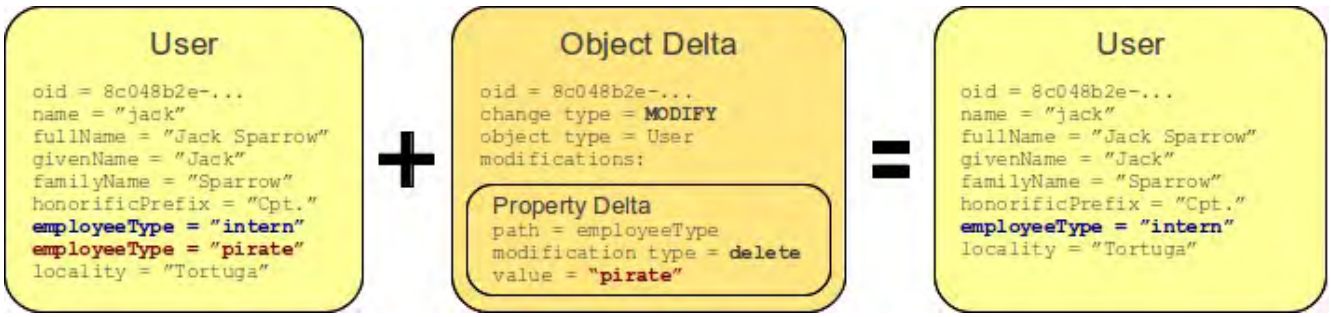
*削除デルタ*は さらにシンプルである。削除するオブジェクトのオブジェクト識別子（OID）だけが含まれる。



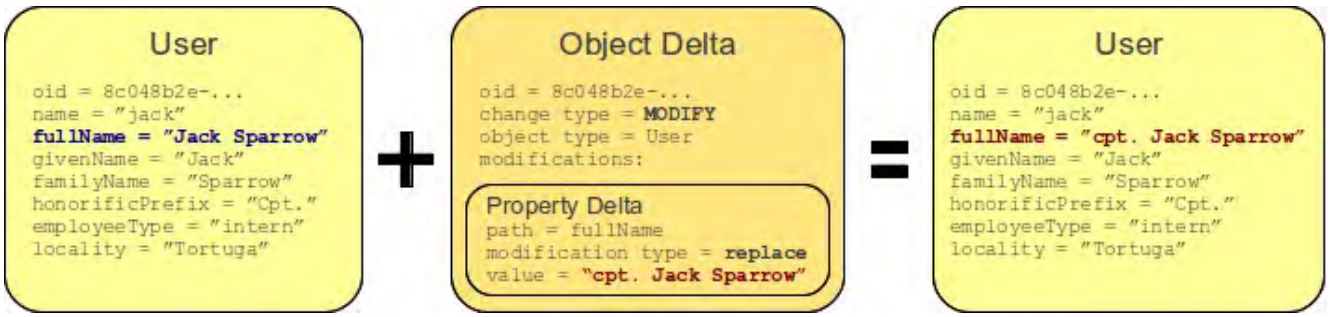
そして最後が*変更デルタ*である。このデルタには変更された既存オブジェクトのプロパティについての概要が含まれている。しかしオブジェクトは様々な方法で変更できるため、*修正デルタ*はこのツリーの中で最も複雑である。修正デルタには*項目デルタ*のリストが含まれている。各*項目デルタ*はオブジェクトの特定の部分がどう変更されるかを表している。たとえば、以下のデルタではユーザープロパティの employeeType に新しい値 pirate が追加されることを示している。



おそらく項目デルタには*追加*、*削除*、*置換*という３つの修正タイプがあるだろう。*追加修正*は 項目に新しい値が１つまたは複数追加されることである。*削除修正*は 項目から値が１つまたは複数削除されることである。



*追加*と*削除*のどちらの場合もデルタにて言及されていない値は影響を受けない。ただし*置換修正*は違う。この修正では、その項目のすべての既存値が除かれ、デルタからの値に置き換えられることになる。



デルタは単数値および複数値のいずれの項目にも対応できるようになっている。実際、*追加修正*デルタおよび*削除修正*デルタは複数値の項目を考慮して特に設計されている。これらのデルタは大量の値をもつ複数値属性があってさえも効率的に動作できる。そこにはもっともな理由がある。アイデンティティ管理の分野では、複数値のプロパティは極めて一般的なのだ。ロール、グループ、特権、アクセス制御リストの通常の実装方法について考えてみてほしい。LDAP サーバーで大規模なグループを管理したことがある人なら、そのときの経験が実に鮮やかに蘇るだろう。しかし midPoint はそのような状況を取り扱うよう設計されている。

ユーザーインターフェイス、マッピング、認可、監査からデータを保存するコンポーネントに至るまで midPoint 内のすべてはデルタを扱うよう設計されている。マッピングは相対論的な方法で設計されている。マッピングのソースを明示的に指定する必要があるのはこのためである。マッピングの実行を制御するため、マッピングソースの定義はデルタの

項目と適合する。デルタは midPoint の演算全体に浸透している。デルタはマッピングにインプットされるが、マッピングはアウトプットとしてデルタを生成する。よって 1 本の完全なチェーンをもつことができる。インバウンドマッピングの結果であるデルタがユーザーオブジェクトに適用されるが、これらのデルタはアウトバウンドマッピングへのインプットでもある。midPoint 内ではすべてが*相対主義*である。

最初のうちは複雑に見えるかもしれないが、慣れるだろうから心配はいらない。それにこの手法には明らかに大きなメリットがある。ただ、賢い読者にとっては特に驚くことでもないようだ。この相対論的な手法で計算リソースの大部分をどうやって節約できるのか？ふつう我々はリソースからとにかくアカウント全体を取り込む。よってすべての属性の再計算には害はない。計算そのものは高速であり、低速なのは取り込み操作である。そうだろう？賢い読者は確かに正しい。少なくとも部分的には。実際に大半のリソースが、効率的な単一操作にてすべてのアカウント属性を取り込む。そしてこの場合、相対論的方法でいくならば効率は大きく向上しない。しかし例外がある。例えば、大きな属性の値をすべて（大規模なグループのメンバー全員などを）返すことはしないリソースもあるだろう。すべての値を取り込むには要求を追加する必要がある。そしてかなり大規模なグループの場合、たしかに要求は多いだろう。相対論的手法が大きなメリットがあるのはこのようなケースである。次のセクションでライブ同期を見れば、そのメリットはさらに明確になるだろう。ただ、相対論的手法である最も重要な理由はパフォーマンスではない。相対論的手法を選ぶ極めて強い理由が一つある。それは整合性である。整合性は分散システムを設計しようとする多くのエンジニアをひどく苦しめるものである。実際、アイデンティティ管理ソリューションは分散システムである。ただ、まさに非常に疎結合型の分散システムである。コネクタのロッキングまたはトランザクションのサポートはない。サポートがあったとしても、リソースの大部分はアイデンティティ管理 API にこうした整合性のしくみを提供できない。つまり midPoint は従来の整合性の仕組みに依存できないということである。そしてここに相対論的手法が非常に役立つのである。相対論的計算はロッキングやトランザクションがなくとも正しい結果が得られる可能性が非常に高い。これは典型的なアイデンティティ管理のデプロイにとって実に素晴らしいことである。そして、相対論的な計算が揺らぎかねない稀なケースには常に最終手段としてリコンシリエーションが控えている。midPoint の相対主義的な性質のおかげで、リコンシリエーションの必要性が大幅に減っている。

長々と説明したが、おそらくこれで賢い読者も満足してくれたものと思われる。少なくともしばらくは。しかしかなりシンプルにまとめるとこうである。midPoint の相対論的手法は奇跡を起こしうる。たとえば、midPoint リソースはソースおよびターゲットの両方になりうる。そして単一属性でさえ情報のソースおよびターゲットの両方になりうるのだが、このような機能を可能にするのが相対論的手法である。この原則、つまり相対性は比較的シンプルである。しかし midPoint におけるその効果は、まさに革新的であることにほかならない。

ライブ同期

midPoint はさまざまな仕組みの同期を備えている。一方に低速で厳しいが信頼性のあるリ

コンシリエーションがあり、他方にはライブ同期がある。ライブ同期はほぼリアルタイムの同期機能を提供できる軽量な仕組みである。ライブ同期は特にリソース上に起こった最近の変更を探す。そしてそのような変更を検出すると、それらをただちに処理するよう設計されている。ライブ同期が適切に使用されれば、一般的には同期遅延は数秒または数分の単位である。

リコンシリエーションと違い、ライブ同期は手動ではトリガーできない。手動でトリガーしてもほとんど意味がないからだ。ライブ同期は長時間実行されるタスクの中で、新たな変更がないか短い間隔で繰り返し検索する。リソースが同期をとるよう構成されているのなら、ライブ同期の実行に必要なのはライブ同期タスクを設定することだけである。midPoint ユーザーインターフェイスから簡単にこれを行うことができる。また、ライブ同期タスクの例は前述の人事フィードセクションにてすでに紹介済みである。

ライブ同期タスクは一定の間隔で起動する。そして起動するごとにコネクタを呼び出す。ライブ同期ができるコネクタは、特別な操作を使用してリソースから新たな変更を取得する。理論上は、コネクタはどんな合理的な仕組みにも対応できる。しかしリソースが使用するのは、たいてい次の 2 つの仕組みである。

- **タイムスタンプベースの同期（Timestamp-based synchronization）：** リソースはアカウントごとに最後に行われた変更のタイムスタンプを記録する。コネクタは前回のスキャン後に変更されたアカウントをすべて検索する。これは非常にシンプルかつ比較的効率的な方法である。ただ一つ大きな制限がある。それは削除済みのアカウントは検出できないということである。削除済みのアカウントにはタイムスタンプがないため、コネクタは同期スキャンにてそれを見つけることはない。
- **変更ログベースの同期（Changelog-based synchronization）：** リソースは最近の変更の「ログ」を保持している。コネクタはログを確認し、前回のスキャン後にログに追加された変更をすべて処理する。これは非常に効率的かつ柔軟性のある方法である。しかしこの方法はシンプルではなく、対応しているリソースもほんのわずかである。

どのライブ同期方法であっても「最近」発生した変更は何かを記録する必要がある。すなわち、midPoint がすでに処理した変更はどれか、まだなのはどれかということである。たいていは midPoint が覚えておくべき値がある。前回スキャンのタイムスタンプ、変更ログの最後のシーケンス番号、前回処理した変更の連番などである。コネクタはそれぞれ違う値をもっており、それらの値はコネクタ独自の意味を持っている。midPoint はこうした値を「トークン」として参照する。直近のトークンがライブ同期タスクに格納される。このようにして midPoint は処理した変更を記録する。（稀であることを祈るが、）リソースおよび midPoint のトークンがずれてしまうことがある。これはネットワーク時間が同期から外れるなどしてバックアップからリソースデータベースが復旧したこと等により発生する場合がある。もし発生した場合は、ライブ同期タスクからそのトークンを削除するだけで、たいていは再び同期が実行されるようになる。

ライブ同期は高速かつ効率性が高い。しかし信頼性は完全ではない。midPoint は変更の一部を見過ごしてしまう可能性がある。非常に稀とはいえ、起こりうることではある。この

ような場合に状況を確実に修正してくれるのがリコンシリエーションである。思い出してほしい、同期の仕組みはすべて同じ構成を共有していることを。そして同一リソース上で同時にライブ同期とリコンシリエーションをまったく問題なく実行できることを。ただもちろん、リコンシリエーションの実行頻度はライブ同期より少ないほうがよいだろう。

結論

アイデンティティ管理分野において同期は最も重要な仕組みの一つである。同期の主要目的はデータを midPoint に取り組むことである。アイデンティティ管理デプロイを始めるにあたって、まずはデータを midPoint に取り込む、これが実によい手法である。人事システムからデータを取り込むのだ。そのデータとアクティブディレクトリとを相互に関係させる。主なリソースをすべて midPoint に接続し、データを互いに関連付ける。この段階で midPoint は何も変更を行う必要はない。実際のところ完璧な手法としては、この段階ではすべてのリソースを読み取り専用にしておくことである。あくまでも midPoint からデータが見えることがポイントである。しかしなぜこれが必要なのか？

- そのデータが本当はどんな品質なのかを見極める。ほとんどのシステムオーナーは少なくともそこにどんなデータセットがあるかは把握している。しかし、データが処理され検証されるまでは、そのデータの品質を判断することはほぼ不可能だ。それがまさにこの段階でmidPointができることである。これはデータのクリーンアッププランや浄化プランを立てるのに欠かせない情報である。
- そこにあるアカウント数、アカウントのタイプ数を把握する。おそらく従業員アカウントがあるのは至極明らかである。しかし請負業者、サプライヤ、サポートエンジニアのアカウントはあるのか？これらのアカウントはアクティブなのか？命名規則は？システム管理者は従業員アカウントを使用して管理を行っているのか？または専用の高特権のアカウントを使用しているのか？こうした情報はプロビジョニングポリシーの設定時に欠かせないものである。
- アカウントのアサインおよびエンタイトルメントを把握する。すべての従業員がアクティブディレクトリにアカウントを持っているか？頻繁に使用されるグループはあるか？組織構造がどのようにアカウントに影響を与えるか？こうした情報はロールベースのアクセス制御構造やその他のポリシーを設計する際に非常に役に立つ。
- セキュリティの脆弱性を確実に把握する。かなり前に削除されているはずの孤立したアカウントがあるか？先日の夜間緊急時の後に放置されたままのテストアカウントはあるか？いかにも、アイデンティティ管理のないセキュリティなどないのである。

これは好調なスタートである。これがデプロイの第一ステップで行うすべてだとしても、やはり大きなメリットはある。より優れた可視性を得ることとなり、そのような可視性にはより優れたセキュリティが伴う。また、環境を分析し、アイデンティティ管理のデプロイに向けた次ステップを計画するためのデータを得る。もはや知らないという状態ではなくなるのだ。これは実に重要である。データを得ずして理論を立てるのは実に大きな過ちである。

第 7 章： midPoint の開発、保守、サポート

影からしか観察しない者は、太陽の明るさに
不平は言えない。

– フランク・ハーバート

midPoint は*プロフェッショナルオープンソース*プロジェクトである。*つまり、midPoint は*プロフェッショナルな方法を使用して開発されるものの、製品は*オープンソース*ライセンスのもと入手できるということである。

プロフェッショナル開発

midPoint は熟練した開発者らが開発している。その開発は、数十年ものソフトウェアエンジニアリング経験をもつ midPoint 中核チームのシニア開発者らが率いている。そしてまだキャリアの浅い、若い開発者たちもわずかながらいる。第一に midPoint 開発チームは協働して次世代ソフトウェアを開発することを楽しむ開発者らによるコミュニティである。midPoint の開発においてはすべてプロフェッショナルであることが厳しく要求されるが、プロジェクトを本当に前に進めているものは大半がエンジニアリングへの情熱である。

中核となる開発者は全員 Evolveum の社員である。Evolveum は midPoint を創った会社である。これを保守しているのも Evolveum である。midPoint の作業の大部分は midPoint 中核チームが行っている。中心的開発者らは全員、midPoint の仕事に対して支払いを受けている。彼らは midPoint で得られる収入によって自分たちの生計をたてることができる。midPoint から得られる Evolveum の収入があればこそ、開発者らのすべての時間を midPoint の開発にあててもらうよう確保できる。つまり midPoint が適切に保守されるということだ。

プロフェッショナルな開発とは、midPoint の開発と保守にプロフェッショナルなソフトウェアエンジニアリング方法が使用されるということである。midPoint 開発は継続的インテグレーションの原則に断固としてもとづいている。文字通り、数千のインテグレーション自動テストが midPoint のビルドプロセスに直接統合されている。ほかにも数千の自動テストが日々実行されている。現実世界の構成を入念に反映したテストがある。実際のリソースを使用したテストがある。どれも midPoint の開発に欠かせない部分である。midPoint は包括的で柔軟性の高いシステムである。midPoint が信頼性をもって動作するにはプロ品質の確保が欠かせない。

オープンソース

midPoint はオープンソースプロジェクトである。つまり、オープンソースライセンスのもと midPoint のソースコードがすべて利用できるということである。我々はもっともリベラルなライセンスの一つであることから、アパッチライセンス 2.0 を選んだ。しかしオープンソースにはライセンスだけではないもっと多くのことがある。Evolveum はオープンソースの手法に全力で取り組んでいる。midPoint は完全に公に開発されている。これまでの midPoint ソースコードはすべて公開されている。それぞれの開発者の一つ一つのコミッ

トをただちに誰もが利用できる。midPoint の完全なソースコードが利用できる。意図的に隠しているようなプライベートな部分は一切ない。追加機能を持つプライベートブランチは一切ない。サポートブランチでさえすべて完全に公開されている。ソースコードについていえば、midPoint はこれ以上ないくらいオープンソース方法に忠実である。

midPoint の大部分の開発を Evolveum が担っているとしても、 オープンソースであることは midPoint の成功に絶対に不可欠なことである。midPoint ユーザーが midPoint を完全に理解できる唯一の方法がオープンソースだからだ。重要なソフトウェアはすべて何らかのカスタマイズが必要であり、オープンソースはカスタマイズという究極の力をもたらしてくれる。オープンソースのおかげで参加できる。オープンソースはベンダーロックインを防ぐ素晴らしい手法であり、プロジェクトの寿命を延ばしてくれる。オープンソースには実に多くのメリットがある。Evolveum はオープンソースの手法に全力で取り組んでいる。

midPoint はオープンソースプロジェクトとして開始された。midPoint ソースコードは初日から利用可能であった。我々がそれについて言いたいことがある限り、midPoint はずっとオープンソースのままであろう。

midPoint のリリースサイクル

midPoint の開発サイクルは安定している。毎年 2 回、*機能リリース*がある。その名のとおりこのリリースは新機能と大きな向上を盛り込んだものとなっている。これに加え、*メンテナンスリリース*も何回かある。これにはバグフィックスや小さな改善が盛り込まれている。メンテナンスリリースは必要に応じて発行され、厳密なスケジュールはない。メンテナンスリリースのタイミングはmidPointサブスクリプション契約者からの影響を受ける。

midPoint サブスクリプション方式

midPoint のサブスクリプション方式は Evolveum が提供しているサービスである。サブスクリプションの種類としてはスコープおよびサービスレベルの違うものをいくつか取り揃えている。ただ一般的に言えば、サブスクリプションサービスの大半はサードラインサポートである。つまり基本的には、我々が midPoint のバグを修正するということである。バグなのか構成上の問題なのかはっきりしない、際どい問題を診断するサポートも間違いなく含まれている。簡単に言ってしまえば、midPoint のデプロイを問題なく確実に実行させるための方法がこの midPoint サブスクリプション方式いうことだ。一から midPoint のデプロイをサポートするサブスクリプションサービスもある。また、一部には機能開発や機能向上を含むサブスクリプションサービス（いわゆるフォースラインサポート）もある。プロジェクトに必要なことをすべて midPoint で確実に実現できるようにするには、こうしたサブスクリプション方式が理想的な方法である。

midPoint の開発と保守において重要な資金調達はサブスクリプションを通じて行われる。そのため、サブスクリプション契約者からの問題解決や機能の要求をより優先するのはごく当たり前のことである。これにより、midPoint の中核チームが他のタスクに従事できる時間は制限される。そこでいくつかの規則がある。

- midPoint の新機能はすべて*出資される*ものとする。つまり、有効な midPoint サブ

スクリプションを有する顧客がその機能を推薦し支持しているということである。もちろん、機能開発を含むハイレベルなサブスクリプションでなければならない。あるいは、誰かがその機能にかかる開発コストを負担するという方法もある。ただ、機能への直接的な資金援助は非常に限られている。midPoint 開発能力の大部分はサブスクリプション契約者のために確保されているからだ。

- midPoint のアーキテクチャおよび品質は Evolveum チームに一義的な責任がある。Evolveum の収入の一部は midPoint の保守、つまりアーキテクチャを最新の状態に保ち、システミックな品質向上を行い、長期的に midPoint のメンテナンスを行う等のために確保される。稀なケースではあるが、Evolveum が機能のスポンサーになることもある。たいていは midPoint 開発を正しい方向に向ける戦略的な機能である。または特に興味深い機能を求めての、実験的な機能であるかもしれない。ただ Evolveum が「顧客」の機能に対しスポンサーになることは一切ない。そうした機能はサブスクリプションを通じて賄われる必要がある。
- Evolveum は、いずれは midPoint 内のバグをすべて修正する。こうしたバグ修正は midPoint のプライマリ開発ブランチ（*マスターブランチ*）に委ねられる。うまくいった修正は次回の機能リリースに盛り込まれる。しかし midPoint のリリースサイクルは固定のため、各リリースにてすべてのバグが修正されるわけではない。サブスクリプション契約者に影響のあるバグの修正がまず優先される。なおも時間があれば、（契約者でない人々から報告された）他のバグも修正される。ただこれについての保証はない。リリース前に時間切れになってしまった場合は、非契約者から報告された機能は修正されない。実際にそのような修正は、数回のリリースを待ってようやく修正される場合がある。
- 各機能リリースには*サポートブランチ*がある。ここからメンテナンスリリースが発生する。しかし、バグ修正または改善はそれぞれマスターブランチにてまず開発される。そしてサポートブランチに移植しなければならない。それには時間がかかる。そのため非常に厳密な規則を設けている。バグ修正、改善、その他のアップデートは次のような場合にのみサポートブランチに移植される。
  - サポートブランチへの移植がmidPointのサブスクリプション契約者により明示的に要求された場合。
  - セキュリティ問題の場合。セキュリティ問題は絶対的に優先される。レポート元（サブスクリプション契約者か否か）に関係なく直ちに修正される。また、セキュリティ修正はアクティブなサポートブランチに自動で移植される。
  - 多くのユーザーに影響を与える重大な問題を修正した場合。

簡単に言えば、自分のために midPoint を確実に機能させたければサブスクリプションに契約するとよい。サブスクリプション方式はあなたをサポートする。サブスクリプションとはそういうものだ。しかし、サブスクリプションを通じて得られた資金により、長期的な midPoint の保守および新機能開発が可能となる。正しい行動は、midPoint のサブスクリプションに契約することである。

midPoint コミュニティ

midPoint は素晴らしいオープンソースプロジェクトである。そして優れたオープンソース

プロジェクトはみなそうであるように、midPoint もまた活気に満ちたコミュニティを有している。それはエンジニアリングコミュニティとビジネスコミュニティの両方である。エンジニアリングコミュニティの主なコミュニケーションチャネルは midPoint のメーリングリストである。

メーリングリストは、midPoint の機能に関する話し合い、新リリースのアナウンス、構成問題の話し合い、開発チームへフィードバックの提供などに利用される。midPoint コミュニティは誰にでも開かれている。

ビジネスコミュニティはほぼ Evolveum のパートナーで構成されている。Evolveum パートナーは、midPoint ソリューションの提供、ファーストラインおよびセカンドラインのサポートサービスの提供、プロフェッショナルサービスや midPoint にもとづくカスタマイズされたソリューション等の提供を行う。可能性は無限だ。ビジネスコミュニティでさえも開かれている。エントリレベルのパートナーシップは誰にでも開かれている。ただし、パートナーシップのレベルはいくつかあり、パートナーによってはレベルアップに多少の努力が必要となる。midPoint パートナーのネットワークは豊か（かつ拡大中）である。パートナーは世界のほぼどこにでも midPoint にもとづいたソリューションを提供できる。

第 8 章： つづく

Hanc marginis exiguitas non caperet.
（この余白はそれを書くには狭すぎる）
– ピエール・ド・フェルマー

当時はこれで以上であった。その時は実際に本書の最終章であったのだ。ただ midPoint の機能をすべて紹介しきれたわけではない。まだほど遠い状態であると言わざるを得ない。midPoint が提供できるもののほんの表面に触れたにすぎないのだ。他の章はまだ執筆していないが、以下の内容について書かねばならないことはたくさんある。

- **ロールベースのアクセス制御（Role-based access control : RBAC）:** アサイン、ロール、条件付きロール、パラメータロール、ロールベースのプロビジョニング、メタロール。ロールに関心があるなら、これはぜひ楽しみにしていただきたい。我々はロールの中にロールを置こうとしており、ロールのためのロールを作成して、文字通り N 番目にロールベースの概念を取り入れる。
- **スキーマ:** midPoint はデフォルトでかなりリッチなアイデンティティデータモデルをもっている。midPoint ユーザーは選択肢としてあらゆる有用なプロパティをもっている。しかしデフォルトのデータモデルで十分足りることはなく、拡張する必要がある。よって本章ではそれについてだけ触れる予定である。つまりユーザープロパティのほか、ロールおよび基本的には他のすべての midPoint オブジェクトの拡張方法である。
- **オブジェクトテンプレート:** midPoint は、たとえば名前と名字から氏名を計算するなど、オブジェクトを自動で再計算できる。しかしこのシンプルな概念はあらゆる目的に使用できる。たとえばロールの自動アサイン、値のルックアップ、複数のソースからの値を統合、外部サービスからの値の取得などである。
- **組織構造:** 組織、部門、部署はもちろんのこと、プロジェクト、チーム、臨時のワークグループ、分野、テナントのほか、実際に思いつけるものはなんでも該当する。これは人々をグループとして組織化し、ポリシーを適用することにほかならない。しかし midPoint の組織構造はそれよりもさらに柔軟である。ロールや midPoint オブジェクトさえまとめることができるのだ。あらゆる形式の組織構造はビルの基礎ブロックであり、これを使用して非常に興味深い構成を創りだすことができる。
- **認可:** midPoint は機密データを取り扱うため、そうしたデータへのアクセスを厳しく管理する必要がある。そのため midPoint には、midPoint 自体へのアクセスを管理するきめ細やかな認可システムがある。認可の仕組みは非常に強力であり、委任管理から部分的なマルチテナントまで、多くのシナリオが可能である。
- **その他のアイデンティティ管理:** ほかにもまだ言及していない興味深い機能がたくさんある。たとえば反復処理、パスワードポリシー、通知、補助オブジェクトクラス、プロビジョニング依存関係、代理、定数、機能ライブラリ、プロビジョニング

スクリプトなどである。小さな機能かもしれないが、パズルを組み立てるにはどれも欠かせないピースである。これらの機能を使わずして完全なアイデンティティ管理ソリューションはほぼありえない。

- **ワークフローおよび承認:** 利用可能なロールの閲覧、ロールの選択、要求、承認、アサイン、およびアカウントのアサイン。多くのアイデンティティ管理デプロイにおいて、これは基本的事項である。もちろん midPoint では簡単に実行できる。しかしまだあるのだ。複数レベルの承認、オプションの承認手続き、承認者の動的な選択、エスカレーションなど。midPoint はこうした機能の大部分をすでに内蔵しているため、ただ構成するだけでよい。
- **エンタイトルメント:** 問題ないかアカウントを管理する。しかしそれがすべてではない。通常のアカウントと管理者アカウントには大きな違いがある。さいわい、midPoint はグループのメンバーシップ、ロール、特権のアサイン、その他のエンタイトルメントを簡単に管理できる。ここでは、アクティブディレクトリグループ、配布リスト、Unix グループなどの、リソース上のエンタイトルメントのことをいう。midPoint はこれを極めて簡単に管理できるようになっている。
- **一般的な同期:** 真のアイデンティティ管理システムならばアカウントとユーザーを同期させることができる。しかしロール、組織ユニット、グループといったものはどうなのか？ご想像どおり、midPoint なら可能である。実際のところ、ユーザーとアカウントの同期に使用される原則と同じものが直接再利用できる。あなたは自分の部署をアクティディレクトリにOUオブジェクトとして自動で表示することをご希望か？さあ、どうぞ。そこにユーザーを置くことをご希望か？簡単である。さらに部署に入れ子になった LDAP グループを作成することをご希望か？問題ない。midPoint であればすべてできる。
- **手動リソース:** 言うまでもなく、あなたはリソースの大部分をコネクタで midPoint に接続し、自動で管理できるようにしたいだろう。しかしたいていは、一部適合しない厄介なリソースがあるものだ。とはいえコネクタの構築コストを正当化できるほど大きいものではない。おそらくそのコネクタがリソースを管理できるよい方法はない。そこでこれを救うべく再び midPoint の出番である。midPoint は、コネクタではなくシステム管理者が作業を行うという*手動リソース*の概念をもっている。データを読み込み、プロビジョニングは手動で行うという準手動リソースの作成方法もある。さらにこれを ITSM システムに統合する方法もある。
- **監査および履歴:** 監査機能がなかったら、アイデンティティ管理システムは完全とはいえない。midPoint はユーザー、アカウント、ロールの変更といった各操作や、さらには内部の構成変更でさえ監査証跡に格納できる。これは midPoint とデータウェアハウスまたは SIEM システムとの統合に使用できる形式で格納される。また midPoint ユーザーインターフェイスには監査証跡を表示する機能がある。さらに過去の内容も閲覧できる。特定の時点におけるオブジェクトの状態を再構築できるのだ。

- **ポリシールール**:midPointは単なるアイデンティティ管理システムではない。midPointのアイデンティティガバナンス機能は、*ポリシールール*の強力かつ共通の概念にもとづいている。このルールを使用してロールの除外を表現でき、職務分掌（segregation of duties：SoD）ポリシーを定義できる。ポリシーベースの承認の定義にも使用できる。ロールのライフサイクルを制御できる。コンプライアンスポリシーを定義できる。当ルールはこれらすべてが可能である。アイデンティティを*管理すべく*、当ルールが存在する。
- **アクセス検証:** これには検証（certification）、再認定（re-certification）、認証（attestation）など色々な呼称があるが、どんな呼称であれ、どれも同じプロセスである。簡単に言うと、ユーザーがもっているロールがまだ必要であることを確認するため、ユーザーに付与されたロールを確認する方法である。これは、必然的にアドホック操作に傾いてしまうような環境であっても*最小特権の原則*を確実にする方法である。しかしほぼすべての環境で非常に有効な仕組みである。midPoint が提供する検証の仕組みは、ロールアサインに特化した、定期的に実施される一括再認定から、新組織への移動後に使用される単独の臨時再認定に至るまで、じつにさまざまである。
- **データ保護:** アイデンティティ管理はもはや、誰もが何でもできるような未開拓の領域（wild west）ではない。今や厳格なデータ保護ルール、規制、法律がある。midPoint はすぐれたアイデンティティガバナンスシステムとして、データ保護とプライバシーポリシーの管理をサポートできる。midPoint は、EU 一般データ保護規則（GDPR）といったデータ保護規制のコンプライアンス管理に大変役に立つ。
- **ユーザーインターフェイスのカスタマイズ:** midPointはユーザーのセルフサービス、アイデンティティ管理、システム構成に使用できる汎用ユーザーインターフェイスをもっている。このユーザーインターフェイスは動的であるよう設計されている。リソーススキーマ、midPopint スキーマ拡張、認可等に自動で適合する。それゆえ、通常であればユーザーインターフェイスのカスタマイズは一切不要である。しかしデプロイによってはデフォルトの挙動から離れる必要がある場合がある。そして midPoint にはそれを行う用意がある。ユーザーインターフェイスのカスタマイズには、カラー、スタイルシート、ローカライズ、カスタムフォーム、タブ、全体を一新するカスタムページなど様々な方法がある。極端なケースでは、見た感じでは midPoint とわからないほど異なるものに、カスタマイズすることもできる。
- **midPoint サービスとの統合:** midPoint はすばらしいシステムである。しかし、どんなにすばらしいソフトウェアも孤立していては役に立たない。常にシステムどうしは統合する必要がある。統合は midPoint のコネクタを通じて確立される。それがコネクタの真の役目だからだ。しかし、コネクタの能力を超えて midPoint と他のシステムを統合させる必要がしばしば発生する。おそらく midPoint とのやりとりが必要なパスワードリセットアプリケーションがあるだろう。おそらく midPoint のデータを取得する必要がある分析ソフトウェアがあるだろう。midPoint は最初からサービスベースのアプリケーションであるよう設計されたものである。それゆえ、機能を備えた REST サービスおよび SOAP サービスがある。実際、midPoint が行うほぼすべてがこれらのサービスを利用して制御できる。

- **トラブルシューティング:** midPoint は健全な設計のもとに構築されており、明確な原則をもち、その仕組みは一般的かつ再利用可能で柔軟性がある。そして midPoint の設計には秩序と意味がある。これらすべてが揃っていても、midPoint はなおも大きく複雑なシステムである。だれにも過ちはあるものだ。midPoint の仕組みの柔軟性をすべて考慮すれば、構成の間違いは起きても不思議ではない。間違いによる影響は必ずしも明確でないこともあり、その根本原因を突き止めるのは難しいかもしれない。さいわい、midPoint は強力な自己診断の仕組みを備えている。midPoint はデータのロギング方法および診断データの提示方法についてはほぼ融通がきかない。コネクタからユーザーインターフェイスにいたるまで、midPoint のすべてのコンポーネントおよびすべてのレベルには診断機能がある。しかし診断機能がどこから始まり、どう効率的に使用されるかは必ずしも明確ではない。これをすべて説明することが、本章の目指すところである。
- **高度な概念:** これまでの章で説明していない機能がまだいくつかある。整合性の仕組み、ペルソナ、マルチコネクタリソースなどである。中にはめったに使用されない機能もあるが、もしかしたらプロジェクトの救い主になってくれるかもしれない。また常時使用される機能もあるが、それらは midPoint の自然な一部であり、ほぼ見えないものである。しかしこれら機能はすべて説明するに値するものである。また、midPoint 自体がどのようにして開発されるのかを説明する必要がある。多くの実験的な機能および不完全な機能があるからだ。しかしこれは midPoint であるため、こうした機能でさえ非常に有効かもしれない。本章は midPoint を独特な方法で拡張したい人や midPoint 開発に寄与したい人にとっても興味深いであろう。
- **midPoint のデプロイ:** midPoint パッケージのダウンロードからシステム稼働まではさまざまな道のりがある。他より簡単な道のりもある。midPoint は、アイデンティティ管理実務における長年の経験をもとに設計されたものである。よって midPoint は、アイデンティティ管理における悪評高い問題の一部を効率的に克服できる仕組みをもっている。ただし midPoint が適切に使用されることが前提だ。本章は実際のプロジェクトで midPoint をどう使用すべきか助言することが目的である。たとえばプロジェクトの計画方法、集めるべき情報、デプロイの設計方法、環境の準備方法、移行計画、プロジェクトの拡張、変更などである。
- **IAM プログラムの管理:** アイデンティティ管理は情報セキュリティに非常によく似ている。アイデンティティ管理には終わりがない。アイデンティティ管理はプロジェクトではない。*プログラム*である。データの収集、計画、実行という終わりのないサイクルである。アイデンティティ管理を取り巻く環境は常に変化しているため、アイデンティティ管理もまた変化しなくてはならない。しかし midPoint はこうした長続きを実現できるよう設計されたものである。本章はこの終わりのないサイクルをどう扱うか説明する。拡張のため midPoint の構成をオープンにする方法。データ収集方法。新機能要求の取り扱い方法。アップグレードの実施方法。アイデンティティ管理ソリューションを持続可能にしておく方法、である。

- **デプロイの例:** 本書はすべての章で例を多用している。しかしこれらは midPoint 機能の特定の一面を表わすためのものにすぎない。本章は違う。midPoint の実際のソリューションのさまざまな例を完全に網羅する。結局のところ、そのやり方をコピー＆ペーストすることが最適な学習法の一つである。
- **用語解説:** アイデンティティ管理およびガバナンスで使用されている専門用語は、初心者にはまったくなじみのない言葉に聞こえるかもしれない。よっておそらく IDMimsh と英語の辞書を作成したほうがよいだろう。

これらの章はまだ存在していない。まだ執筆されていないからだ。当然のことながら、これらの章を執筆するのは Evolveum チームのメンバーが最適だ。midPoint を設計、実装したメンバー、midPoint のデプロイをサポートするメンバー、日々midPoint で作業しているメンバー、とにかく midPoint にどっぷり漬かっているメンバーである。しかし彼らはただのエンジニアだ。生計を立てなくてはならない。日々の責任を脇へ追いやってまで本書に取り掛かることはできない。本書を完成させるにはどうしても資金調達が必要だ。本書は無料で利用できるため、次章以降の執筆を賄えるような直接的な収入は、本書からは発生しない。方法は一つ、スポンサーだけである。

もし本書を気に入ってくれたなら、どうか次章以降のスポンサーになることを検討していただきたい。残念ながら、市場経済はまったく容赦ない。よってスポンサーがいなければ新しい章はほぼありえないだろうことは実に単純明快なことである。できれば本書のスポンサーになっていただきたい。もしスポンサーになるだけの余裕がないなら、せめて midPoint に関するメッセージと、本書に関するメッセージを広めるサポートをしていただきたい。そうしたサポートはどんな形であっても大歓迎である。

第 9 章： 追加情報

必要なデータが揃ってなければロジックなんて役に立たない。

– レト二世
フランク・ハーバート『デューン／砂丘の子供たち』

我々はできるかぎり本書ですべてを網羅しようと努力している。しかし、謙虚な IDM エンジニアが求める情報をすべて載せられる本はおそらくないだろう。そこで本章では midPoint に関する追加情報を提供する情報源について紹介する。

midPoint Wiki

midPoint wiki は midPoint に関する情報を最も網羅しており、midPoint のすべての文書が格納されている。しかしそれだけではない。コネクタの文書、midPoint アーキテクチャ、開発者文書、midPoint の本質および、midPoint のリリースプランやロードマップを含むあらゆる種類の情報がある。すべてが公開されている。

ただ、wiki はあまりに情報が広範囲にわたるため、見たいページをなかなか見つけにくいことも多い。こうした情報を整理するためにできうることはすべてやってきた。ページは階層状に構成されている。多くのページには追加情報を指し示す「関連項目（See Also）」セクションがある。しかしやってみればお分かりのとおり、wiki で何か見たいものがあるなら、少なくともそれを検索するためのヒントのようなものが必要である。本書がそのヒントを提供してくれるはずだ。探しているものが分かれば、wiki の検索バーが助けてくれる。正しい検索用語を入力すれば、高い確率ですぐに正しいページを見つけられるだろう。

URL: http://wiki.evolveum.com/

サンプル

midPoint プロジェクトでは実に豊富なサンプルコレクションを用意している。その種類はサンプルリソース定義、ロール例、組織構造例などさまざまだ。多くは XML 形式で用意されている。サンプルは midPoint ソースコードとともに GitHub に保管されている。また midPoint 配布パッケージにも含まれている。

URL（最新バージョン）: https://github.com/Evolveum/midpoint/tree/master/samples

書物サンプル

本書にはさまざまなところから例や構成を抜粋して掲載している。もっと小さな抜粋は、midPoint wiki やサンプルファイル（上記）からもってきたものもある。

midPoint デプロイのほぼ完全な構成を載せている章もある。これらの構成は midPoint サンプル内に別フォルダをもっている。midPoint サンプルから「book（書物）」を探してみてほしい。本書で使用された重要ファイルはすべてそこにあり、章番号で分類されている。

URL: https://github.com/Evolveum/midpoint/tree/master/samples/book

ストーリーテスト

midPoint の開発者は完全なエンドツーエンドで自動化されたテストを作成、保守したいと考えている。通常こうしたテストは現実世界での midPoint デプロイがきっかけとなる。我々はこれを*ストーリーテスト*と呼んでいる。midPoint の品質および継続性の維持に重要なものであるが、すばらしいヒントを得ることもでき、ストーリーテストはしばしば有用な midPoint 構成例を提供してきた。

Wiki での説明： https://wiki.evolveum.com/display/midPoint/Story+Tests

コードおよび構成： https://github.com/Evolveum/midpoint/tree/master/testing/story

midPoint のメーリングリスト

midPoint プロジェクトは何年も活発なコミュニティを魅了してきた。そのコミュニティの主なコミュニケーションチャネルは midPoint のメーリングリストである。メーリングリストを使用して、アナウンス、ユーザー提案、そしていわゆる Eolveum メンバーが言うところの*コミュニティサポート*が行われている。midPoint に関する質問にも使用されている。たいていはベテランのコミュニティメンバーらが回答し、追加情報のヒントも提供してくれる。midPoint の開発チーム全体もメーリングリストに掲載されており、必要に応じて回答している。しかしこれはあくまでもベストエフォート（保証はしないがベストは尽くす）型のサービスである。どうか当コミュニケーションチャネルを乱用したりせず、以下のコミュニティガイドラインを遵守していただきたい。

1. **礼儀正しくあれ。**メーリングリストはベストエフォート型のサービスである。メーリングリストの質問に答えても誰も（直接）報酬を得られるわけではない。質問に答えているエンジニアらは、コミュニティをサポートしたいという気持ちから日々の業務に加えそれを行っている。だからサポートを求めるときは、失礼のないようにする。一方、質問に答えるときは、どうか他のメンバーに敬意を払っていただきたい。誰しもスタート地点はあり、当然ながら初心者ユーザーはすべてを知っているわけではない。スキルに差があってもそこは我慢していただきたい。
2. **聞くまえにまず調べてみる。**Google 検索、midPoint wiki やメーリングリストアーカイブで検索すればすぐにわかるような取るに足らない質問はしないこと。エラーが発生したら、そのエラーメッセージをじっくり読み、ありそうな原因をじっくり考えてみる。その構成で少し実験をしてみる。midPoint wiki のトラブルシューティングセクションをみてみる。質問をするまえに少なくとも数分間は自分で調べてみる。それでも答えが見つからなかったら、そのときはメーリングリストで質問するのがよいだろう。
3. **詳細を説明する。***「midPoint がこわれたので助けてください」*といった内容を投稿してもおそらく何の回答も得られないだろう。問題の内容をもっと詳しく説明する。必ず関係する構成部分も記載する。必ずエラーメッセージも含める。必要に応じてログファイルも参照する。そして何より重要なのは、何を達成しようとしているのかを説明することである。おそらく問題の根本は、完全に誤った手法を使用していることにあると思われる。コミュニティはあなたに正しい方向を指し示してくれるだろう、ただそれは彼らがあなたの目的を把握できたときのみである。

4. **返報する。**メーリングリストは、ユーザーが質問し開発者が回答するという一方通行のコミュニケーションチャネルではない。midPoint 開発者以外のコミュニティメンバーに配布される重要な知識体系がすでに整備されている。本ガイドラインを遵守して質問をすれば、たいていは回答が得られるだろう。しかしそのためには回答してくれる誰かが必要である。つまり、ただ質問をするだけではいけない。誰かの質問に対する答えをもし知っていたら、どうかすすんで答えてあげてほしい。回答が完璧でなくても気にする必要はない。たとえ部分的な回答であっても初心者ユーザーには大歓迎だろう。簡単に言ってしまえば、コミュニティからテイクする（受け取っている）だけではだめだということだ。コミュニティからもらったものに対してあなたも報いる努力をしてほしい。

また、Evolveum や midPoint 開発者に直接質問を送りたくなるかもしれない。しかし開発者らには大勢の midPoint ユーザー、パートナー、顧客、貢献者がおり、日々の業務を行っている。midPoint の中心的開発者にとって一番の責務は、midPoint の開発を確実に継続させることである。当然ではあるが、開発者らは midPoint プロジェクトに資金援助をもたらしてくれるタスクにより時間を割きたいと考えている。よってコミュニケーションにあたっては厳格に優先度を設定している。midPoint のサブスクリプション契約者らへの回答は最も優先度が高く、その次がメーリングリストである。一方、コミュニティからの個人的なメッセージは断然低い。我々は midPoint に関する知識を効率的に広めたい。その点でメーリングリストはよいが、個人的なコミュニケーションはそうではない。それが、このような優先度になっている主な理由である。それに、もし一人の開発者に直接連絡して質問しても、それに回答できるのはその開発者だけである。しかしメーリングリストに質問を送信すれば、それに回答できる人はもっと多い。よって有効なサブスクリプション契約をもっていないかぎり、最適な手段はメーリングリストということになる。

メーリングリストの URL: http://lists.evolveum.com/mailman/listinfo/midpoint

第 10 章： 結言

本書はまだ完成していない。それどころかまだ始まったばかりだ。金銭的に許されるかぎり、できるだけ早く本書の執筆は少しずつ反復しながら加筆されている。

あなた方からの貢献、寄付があってこそ、本書の完成は近くなるだろう。また、midPoint のサブスクリプション契約も検討していただきたい。サブスクリプションは midPoint 開発の資金源である。Evolveum は他分野で多額の利益をあげてそこからオープンソース開発の資金を賄えるような大企業ではない。他の分野なんてない。Evolveum はオープンソースのみの企業である。我々が従事しているのはオープンソースだけである。また、Evolveum はベンチャー企業でもない。ベンチャーキャピタルによる資金援助は受けておらず、数百万ドルも費やせるような資金はない。自己資金による企業である。費やせるのはサブスクリプション、スポンサーのついた機能やサービスから得た収入だけである。ほかに収入はない。金銭的に許せば、midPoint の開発は早くすすめられる。midPoint の文書作成および本書にも同じことが言える。Evolveum の収入に比例して進んでいく。だからもし本書を気に入ってくれたなら、ぜひ我々のサポートを検討していただきたい。お金も時間もどちらも大歓迎である。

少なくとも我々が本書の執筆を楽しんだのと同じくらい、そしてそもそも midPoint 構築を楽しんだのと同じくらい、あなた方も本書を楽しんでくれたことを願っている。midPoint は極めて独特なソフトウェアプロジェクトであり、あなた方なしでは不可能であろう。Evolveum チーム全員が、この素晴らしいプロジェクトを現実にしてくれた、かつての、現在の、そして今後の midPoint 支援者全員に感謝している。共にこの興味深く有用なソフトウェア製品を創り上げてきた。共に midPoint をより良いものにできると信じている。

皆様に感謝申し上げる。