

PRACTICAL IDENTITY MANAGEMENT WITH MIDPOINT

BY RADOVAN SEMANČÍK ET AL.

VERSION 2.1
NOVEMBER 2020

Practical Identity Management With MidPoint

Radovan Semančík et al.

Version 2.1, 2020-11-25

Colophon

Practical Identity Management With MidPoint
by Radovan Semančík et al.
Evolveum

Book revision: 2.1
Publication date: 2020-11-25
Corresponding midPoint version: 4.0

© 2015-2020 Radovan Semančík and Evolveum, s.r.o. All rights reserved.

This work is licensed under a
[Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](#).

Major sponsoring for this book was provided by:



Table of Contents

Colophon	1
Introduction	6
1. Understanding Identity and Access Management	10
Directory Services and Other User Databases	10
Directory Servers are Databases	13
Access Management	14
Identity Management	25
Identity Governance and Compliance	44
Complete Identity and Access Management Solution	48
IAM and Security	49
Building Identity and Access Management Solution	51
2. MidPoint Overview	52
How MidPoint Works	52
Case Study	55
Connectors and Resources	56
User and Accounts	59
Initial Import	61
Assignments and Projections	64
Roles	67
There Is Much More	70
What MidPoint Is Not	70
3. Installation and Configuration Principles	72
Requirements	72
Installation	72
MidPoint User Interface	73
User Interface Areas	74
User Interface Concepts	75
Object Details Page	76
MidPoint Configuration Basics	79
Configuration Objects	80
XML, JSON and YAML	81
Maintaining MidPoint Configuration	83
Looking Around MidPoint Installation	84
Logging	84
4. Resources and Mappings	86
Resource Definitions	86
Connectors	88
Bundled and Deployed Connectors	89

Connector Configuration Properties	90
Testing the Resource	92
Resource Schema Basics	93
Hub and Spoke	94
Schema Handling	98
Attribute Handling	99
Mappings	101
Expressions	106
Script Expressions	107
Activation	109
Credentials	110
Complete Provisioning Example	110
Shadows	117
5. Synchronization	120
Synchronization in MidPoint	121
Sources, Targets And Other Creatures	122
Inbound and Outbound Mappings	123
Correlation	126
Synchronization Situations and Reactions	127
Synchronization Tasks	131
Synchronization Example: HR Feed	133
HR Feed Recommendations	141
Synchronization and Provisioning	142
Synchronization Strategies	144
Mapping and Expression Tips and Tricks	145
Expression Functions	147
6. Schema	164
MidPoint Schema	164
Data Unification	165
Basic User Schema	165
Operational, Experimental and Deprecated Items	171
Activation	172
Schema Definition	177
Schema Extensibility	179
PolyString and Protected String	182
Advanced Schema Concepts	186
Type Hierarchy	186
Item Path	190
Conclusion	193
7. Role-Based Access Control	194
Reality, Policy and Assignments	194

Roles	197
Provisioning Roles	198
Roles, Accounts and Attributes.....	201
Role Hierarchy	202
Role Universality	206
Role Hierarchy Structure	207
Assignment Gets Complicated	208
Dynamic Roles	209
Metaroles	212
RBAC, ABAC And The Wildlife	214
8. Object Templates	217
Object Templates	217
Item Definitions In Object Template	219
Automatic Role Assignment in Object Template	223
Autoassignment in Roles	229
Iteration	231
Includes	237
Combining the Ingredients	238
Complete Deployment Example	239
Conclusion	255
9. Organizational Structures	256
Organizational Units	256
Organizational Structure Hierarchy	259
Orgs in the Database	263
Orgs and Roles	264
Managers	266
Relation	270
Multiple Organizational Structures	273
Beyond Users	275
Organizational Structure Synchronization	276
Provisioning Organizational Structure	287
Focus and Projection	294
Conclusion	297
10. Troubleshooting	298
Designed for Visibility	298
Systematic Approach	299
Error Messages and Operation Results	300
Logging	302
Auditing	308
Troubleshooting Clockwork and Projector	310
Troubleshooting Mappings and Expressions	316

Troubleshooting Connectors	319
Troubleshooting Authorizations	322
Reporting a Bug	325
Useful Troubleshooting Tips	328
11. MidPoint Development, Maintenance and Support	330
Professional Development	330
Open Source	330
MidPoint Release Cycle	331
MidPoint Support and Subscriptions	331
MidPoint Community	332
12. Additional Information	334
MidPoint Wiki	334
Samples	334
Book Samples	334
Story Tests	335
MidPoint Mailing List	335
Evolveum Blog	336
To Be Continued	337
Conclusion	341

Introduction

It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to.

— Bilbo Baggins, The Lord of the Rings by J.R.R. Tolkien

Many years ago we started a project. Because we had to. Back then we didn't think too much about business and markets and things like that. We were focused on the technology. Then the project simply went on. It had its ups and downs – but all the time there was pure engineering passion. The effort brought fruits and now there is a product like no other: midPoint.

MidPoint is an identity management and governance platform. We built it from scratch. It is a comprehensive and feature-rich system. MidPoint can handle complete identity lifecycle management and some parts of identity governance and compliance. It can speed up the process that create accounts for new employee, student or customer. MidPoint can automatically disable accounts after the relation to the person has expired. MidPoint manages assignment of roles and privileges to employees, partners, agents, contractors, customers or students. MidPoint keeps an eye that the policies are continually maintained and enforced. It governs the processes of access reviews (attestations). It provides auditing and reporting based on the identity data.

MidPoint is a comprehensive system, and there are not that many products that can do what midPoint does. Yet, midPoint has one critical advantage over the competing products: midPoint is completely open source platform. Open source is the fundamental philosophy of midPoint. We believe that open source is a critical aspect in the development of modern quality software. Open source principle is a guiding principle of midPoint community: partners, contributors supporters and in fact all the engineers that work with midPoint. Open source character means that any engineer can completely understand how midPoint works. It also means that midPoint can be modified as needed, that issues can be fixed quickly and especially to ensure the continuity of midPoint development. After all these years with midPoint, we simply cannot imagine using any identity technology which is not open source.

There are few engineers in our team who have been dealing with identity management deployments since early 2000s. The term "Identity and Access Management" (IAM) was not even invented at that time. We have seen a lot of IAM solutions during our careers. The IDM system was the core of vast majority of these solutions. Whether it is given by our point of view or whether that is the generic rule we do not know for sure. All we know is that midPoint is a really useful tool. When it is used by the right hands, midPoint can do miracles. This is exactly what this book is all about: the right use of midPoint to build a practical Identity Management solutions. This book will tell you how to build and deploy a practical IDM solution. It will also tell you *why* to do it in the first place. The book will explain not just the features and configuration options. It will also describe the motivation and the underlying principles of identity management. Understanding the principles is as at least as important as knowing the mechanics of an IDM product. The book usually describes *how* the things work when they work. It also tries to describe the limitations, drawbacks and pitfalls. The limitations are often much more important than the features, especially when designing a new solution on a green field.

The first chapter is an introduction to the basic concepts of Identity and Access Management (IAM). It is very general and does not deal with midPoint at all. Therefore, if you are familiar with Identity and Access Management feel free to skip the first chapter. However, according to our experience, this chapter has many some things to tell even to an experienced IAM engineers. If you are impatient and want to start directly with midPoint then skip the chapter (you would do that anyway, wouldn't you?). Just please try to find the time to return to the first chapter later. This chapter contains important information to put midPoint in broader context. You will need that information to build a complete IAM solution.

The second chapter describes the midPoint big picture. It shows how midPoint looks like from the outside. It describes how midPoint is usually used and how it behaves. The purpose of this chapter is to familiarize the reader with midPoint workings and basic principles. It describes how midPoint is used.

The third chapter describes the basic concepts of midPoint deployment and configuration. It guides the reader through midPoint installation. It describes how midPoint is customized to suit the needs of a particular deployment. However, midPoint customization is a very complex matter, and this chapter describes just the basic principles. It will take most of the book to fill in the details.

The fourth chapter describes the concepts of resource and mappings. This is the bread-and-butter of an identity management. This chapter will tell you how to create very basic midPoint deployment, how to connect target systems and how to map and transform the data.

The fifth chapter is all about synchronization. Primary purpose of synchronization is to get the data from the source systems such as HR system to midPoint. Yet, midPoint synchronization is much more powerful than that. This chapter also expands the explanation of underlying midPoint principles such as mappings and deltas.

The sixth chapter talks about midPoint schema. MidPoint has a built-in identity data model. Even though this data model is quite rich, it is usually not sufficient to cover all the real-world use cases. Therefore the data model is designed to be extensible. This chapter describes the methods how a new data items can be defined in midPoint schema.

The seventh chapter is all about role-based access control (RBAC). MidPoint role-based model is a very powerful tool to set up complex structures describing job roles, responsibilities, privileges and so on. The role model, and especially the concept of assignment, are generic mechanisms that are used in almost every part of midPoint. Organizational structure management and many identity governance features are built on the foundations described in this chapter.

The eighth chapter is an introduction to object templates. Those templates form a basis of an internal data consistency in midPoint. They can be used to set up simple policies and automation rules. Object templates are a basic workhorse that is used in almost all midPoint deployments.

The ninth chapter describes organizational structures. MidPoint organizational structure mechanisms are generic and very powerful. They can be used to model traditional organizational hierarchies, tree, and even structures that are not exactly trees. The same mechanism can be used to set up projects, teams, workgroups, classes or almost any conceivable grouping concept. This chapter describes how organizational structures are synchronized with the outer world. The concept of *generic synchronization* can be applied to synchronize midPoint objects with almost any external data structure.

The tenth chapter is about troubleshooting. To err is human. Given all the flexibility of midPoint mechanisms configuration mistakes just happen, and it may not be easy to figure out the root cause of problems. Therefore, this chapter provides an overview of midPoint diagnostic facilities and recommendations for their use.

The eleventh chapter provides overview of midPoint development process and overall approach. It is also explained how midPoint development is funded and how midPoint subscriptions work.

The twelfth chapter is a collection of pointers to additional information. This includes a pointer to sample files that accompany this book.

The next chapters are not written yet. The description of policies, entitlements, authorizations, archetypes and all the other advanced topics is missing. This book is not finished yet. Just like midPoint itself this book is written in an incremental and iterative way. Writing a good book is a huge task in itself, and it takes a lot of time. We cannot dedicate that much time to writing the book in one huge chunk. Obviously, a book like this is needed for midPoint community. Therefore we have decided not to wait until the book is complete. We will be continuously publishing those chapters that are reasonably well finished. It is better to have something than to have nothing, isn't it? Please be patient. The whole book will be finished eventually. As always – your support, contributions and sponsoring may considerably speed up things here.

We would like to thank all the midPoint developers, contributors and supporters. There was a lot of people involved in midPoint during all these years. All these people pushed midPoint forward. Most of all, we would like to thank the people that were there when the midPoint project was young and that are still there until this day. We would like to thank Katka Stanovská, Katka Bolement (née Valaliková), Igor Farinič, Ivan Noris, Vilo Repáň, Pavol Mederly and Radovan Semančík. Those were the people that were there when midPoint was young. And they are still the people who are the force that drives midPoint into the future.

Anything that is stated in this book are the opinions of the authors. We have tried really hard to remain objective. However, as hard as we might try, some points of view are difficult to change. We work for Evolveum – a company that is also an independent software vendor. Therefore, our opinions may be slightly biased. We have honestly tried to avoid any biases and follow proper engineering practices. You are the judge and the jury in this matter. You, the reader, will decide whether we have succeeded or not. You have free access to all the necessary information to do that: this book is freely available as is all the midPoint documentation and the source code. We are not hiding anything. Unlike many other vendors we do not want or need to hide any aspect of the software we are producing.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND). This essentially means that you can freely use this book for a personal use. You can retrieve and distribute it at no cost. However, you are not allowed to sell it, modify it or use any parts of this book in commercial projects. There is no direct profit that we make from this book. The primary reason for writing this book is to spread knowledge about midPoint. However, even open source projects such as midPoint need funding. If you use midPoint in a commercial project that is a source of profit we think it is only fair if you share part of that profit with midPoint authors. Therefore, we have chosen the CC BY-NC-ND license for this book. You can use this book freely to learn about midPoint. However, this license does not give you right to take parts of this book and include it in your project documentation. You can point to this book by

URL, but you are not allowed to pass this book to the customer as a part of product documentation in a commercial project. You are not allowed to use this book as material during commercial training. You are not allowed use the book in any way that generates profit. If you need to use this book in such a way, please contact Evolveum, and you can obtain special license to do this. The license fees collected in this way will be used to improve midPoint and especially midPoint documentation. You know as well as we do that this is needed.

Following people have worked on the words and pictures that make up this book:

- Radovan Semančík (author and maintainer)
- Veronika Kolpaščíková (illustrations, corrections)

Yet there is much more people whose work was needed to make this work happen: midPoint developers, contributors, analysts and deployment engineers, specialists and generalists, theoretical scientists and practical engineers, technical staff and business people, people of Evolveum and the people that work for our partners, our families, friends and all the engineers and scientists for generations and generations past. We indeed stand on the shoulders of giants.

Chapter 1. Understanding Identity and Access Management

The beginning of knowledge is the discovery of something we do not understand.

— Frank Herbert

What is identity and access management? Answer to that question is both easy and very complex. The easy part is: Identity and access management (IAM) is a set of information technologies that deal with identities in the cyberspace. The complex part of the answer takes the rest of this book.

This book deals mostly with *Enterprise Identity and Access Management*. That is identity and access management applied to larger organizations such as enterprises, financial institutions, government agencies, universities, health care, etc. The focus is on managing employees, contractors, customers, partners, students and other people that cooperate with the organization. However, many of the mechanisms and principles described in this book can be applied to non-enterprise environments.

The story of identity and access management starts with information security. The security requirements dictate the need for authentication and authorization of the users. Authentication is a mechanism by which the computer checks that the user is really the one he pretends to be. And authorization is a related mechanism by which the computer determines whether to allow or deny specific action to a user. Almost every computer system has some means of authentication and authorization.

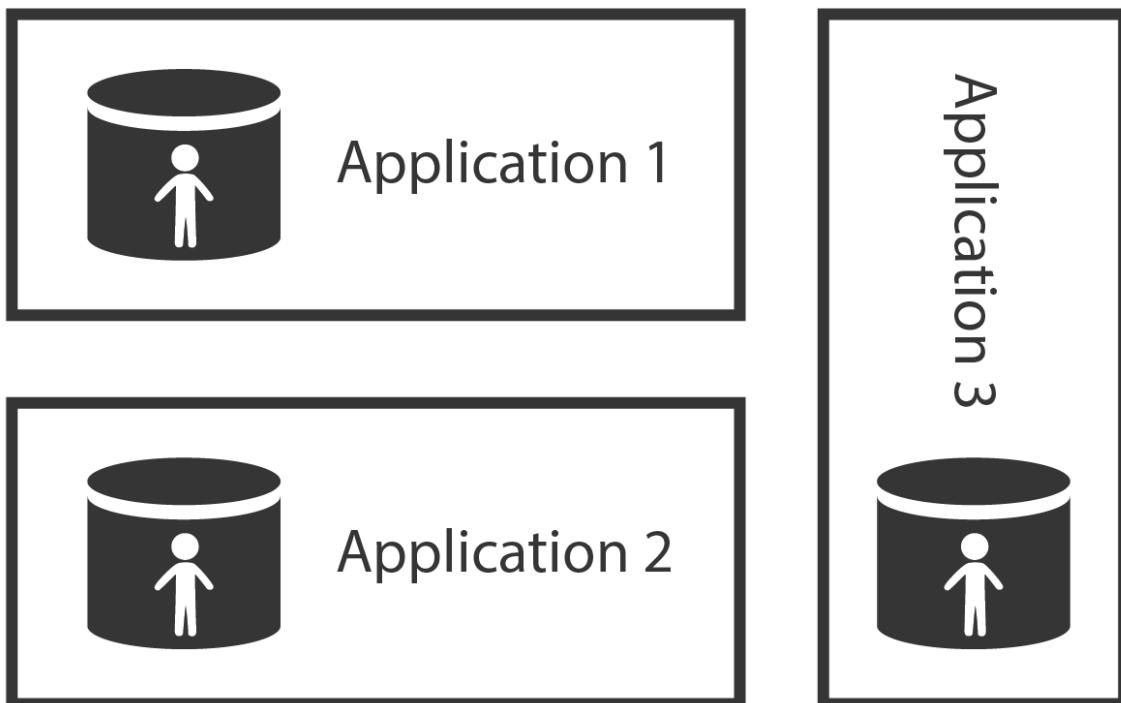
Perhaps the most widespread form of authentication is a password-based "log in" procedure. The user presents an identifier and a password. The computer checks whether the password is valid. For this procedure to work the computer needs an access to the database of all valid users and passwords. Early stand-alone information systems had their own databases that were isolated from the rest of the cyberspace. The data were maintained manually. But the advent of computer networking changed everything. Users were able to access many systems and the systems themselves were connected to each other. Maintaining an isolated user database in each system no longer made much sense. And that's where the real story of digital identity begins.

Directory Services and Other User Databases

The central concept of identity management is a data record that contains information about a person. This concept has many names: user profile, persona, user record, digital identity and many more. The most common name in the context of identity management is *user account*. Accounts usually hold the information that describes the real-world person using a set of attributes such as given name and family name. But probably the most important part is the technical information that relates to operation of an information system for which the account is created. This includes operational parameters such as location of users home directory, wide variety of permission information such as group and role membership, system resource limits and so on. User accounts are represented in a wide variety of forms ranging from relational database records through structured data files to semi-structured text files. But regardless of the specific method used to store

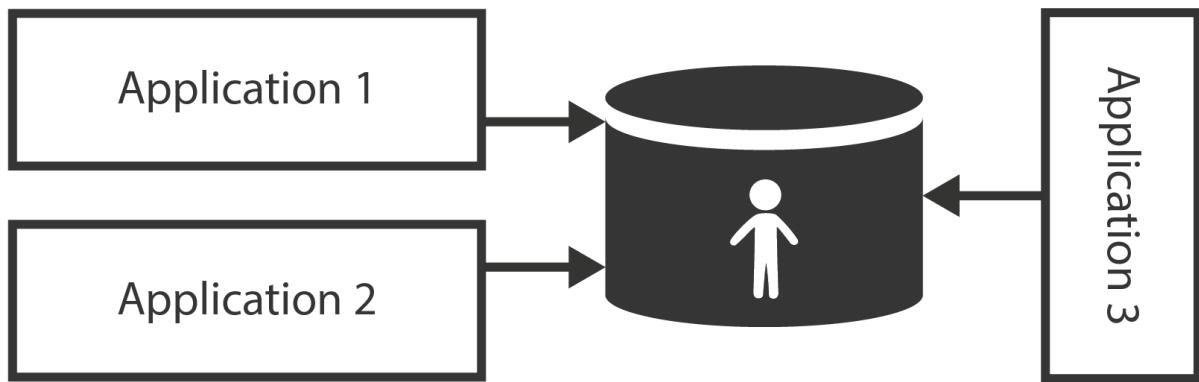
and process the records the *account* is undoubtedly one of the most important concepts of IAM field. And so are the databases where the accounts are stored as accounts, being data records, have to be stored somewhere.

The account databases are as varied as are the account types. Most account databases in the past were implemented as an integral part of the monolithic information system using the same database technology as the system itself used. This is an obvious choice and it remains very popular even today. Therefore many accounts are stored in relational database tables and similar application data stores.



Application data stores are usually tightly bound to the application. Therefore accounts stored in such databases are difficult to share with other applications. However, sharing account data across the organization is more than desirable. It makes very little sense to maintain account data in each database separately – especially if most the accounts are the same in each application. Therefore there is a strong motivation to deploy account databases that can be shared by many applications.

Directory servers are built with the primary purpose to provide shared data storage to applications. While application databases usually use their own proprietary protocol, *directory servers* implement standardized protocols. While databases are built for application-specific data model, directory servers usually extend standardized data model which improves interoperability. While databases may be heavyweight and expensive to scale, directory servers are designed to be lightweight and provide massive scalability. That makes directory servers almost ideal candidates for shared account database.



Shared identity store is making user management easier. An account needs to be created and managed in one place only. Authentication still happens in each application separately. Yet, as the applications use the same credentials from the shared store, the user may use the same password for all the connected applications. This is an improvement over setting the password for each application separately.

Identity management solutions based on shared directory servers are simple and quite cost-efficient. Therefore we have been giving the same advice for many years: if you can connect all your applications to an LDAP server, do not think too much about it and just do it. The problem is that this usually works only for very simple systems.

Lightweight Directory Access Protocol (LDAP)

Lightweight Directory Access Protocol (LDAP) is a standard protocol for accessing directory services. It is an old protocol when judging by Internet age standards. LDAP roots are going as far back as 1980s to a family of telecommunication protocols known as X.500. Even though LDAP may be old it is widely used. It is a very efficient binary protocol that was designed to support massively distributed shared databases. It has small set of well-defined simple operations. The operations and the data meta-model implied by the protocol allow very efficient data replication and horizontal scalability of directory servers. This simplicity contributes to low latencies and high throughput for read operations. The horizontal scalability and relative autonomy of directory server instances is supposed to increase the availability of the directory system. These benefits often come at the expense of slow write operations. As identity data are often read but seldom modified, slower writes are usually a perfectly acceptable trade-off. Therefore, LDAP-based directory servers were and in many places still remain, the most popular databases for identity data.

LDAP is one of the precious few established standards in the IAM field. However, it is far from being perfect. LDAP was created in 1990s, with roots going back to 1980s. There are some problems in original LDAP design, such as grouping mechanisms and some details of search and modify operations. Also, LDAP schema has a distinctive feel of 80s and 90s. LDAP would deserve a major review, to correct the problems and bring the protocol to 21st century. Sadly, there was no major update to LDAP specifications in decades.

Even though LDAP has its problems, it still remains a useful tool. Most LDAP server vendors provide proprietary solutions to LDAP problems. Many organizations store identities in LDAP-enabled data stores. There are many applications that support LDAP, mostly for centralization of

password-based authentication. LDAP still remains a major protocol in Identity and Access Management field. Therefore we will be getting back to the LDAP protocol many times in this book.

Directory Servers are Databases

Directory servers are just databases that store information. Nothing more. The protocols and APIs used to access directory servers are designed as database interfaces. It means that they are good for *storing, searching and retrieving* data. While the user account data often contain entitlement information (permissions, groups, roles, etc.), identity stores are not well suited to *evaluate* them. I.e. directory server can provide information what permissions an account has, but it is not designed to make a *decision* whether to allow or deny a specific operation. And that is not all. Directory servers do not contain data about user *sessions*. It means that directory servers do not know whether user is currently logged in or not. Many directory servers are used for basic authentication and even authorization. Yet, the directory servers were not designed to do this. Directory servers provide only the very basic capabilities. There are plug-ins and extensions that provide partial capabilities to support authentication and authorization. But that does not change the fundamental design principles. Directory servers are databases, not authentication or authorization servers. They should be used as such.

However, many applications use directory servers to centralize password authentication. In fact, this is a good and cost-efficient way to centralize password-based authentication, especially if you are just starting with identity and access management. However, you should be aware this a temporary solution. It has many limitations. The right way to do it is to use *authentication server* instead of directory server. Access Management (AM) technologies can provide that.

Single Directory Server Myth

Shared directory server makes user management easier. However, this is not a complete solution and there are serious limitations to this approach. The heterogeneity of information systems makes it nearly impossible to put all required data into a single directory system.

The obvious problem is the lack of a single, coherent source of information. There are usually several sources of information for a single user. For example a human resources (HR) system is authoritative for the existence of a user in the enterprise. But the HR system is usually not authoritative for assignment of employee identifier such as *username*. There needs to be an algorithm that ensures uniqueness of the username, possibly including uniqueness across all the current and past employees, contractors and partners. Moreover, there may be additional sources of information. For example Management information system may be responsible for determination of user's roles (e.g. in project-oriented organizational structure). Inventory management system may be responsible for assigning telephone number to the user. The groupware system may be an authoritative source of the user's e-mail address and other electronic contact data. There are usually 2 to 20 systems that provide authoritative information for a single user. Therefore, there is no simple way how to feed and maintain the data in the directory system.

And then there are spacial and technological barriers. Many complex applications need local user database. They must store the copies of user records in their own databases to operate efficiently. For example, large billing systems cannot work efficiently with external data (e.g. because of a need to make relational database *join*). Therefore, even if directory server is deployed, these applications

still need to maintain a local copy of identity data. Keeping the copy synchronized with the directory data may seem like a simple task. But it is not. Additionally, there are legacy systems which usually cannot access the external data at all (e.g. they do not support LDAP protocol at all).

Some services need to keep even more state than just a simple database record. For example file servers usually create home directories for users. While the account creation can usually be done in on-demand fashion (e.g. create user directory at first user log-on), the modification and deletion of the account is much more difficult. Directory server will not do that.

Perhaps the most painful problem is the complexity of access control policies. Role names and access control attributes may not have the same meaning in all systems. Different systems usually have different authorization algorithms that are not mutually compatible. While this issue can be solved with per-application access control attributes, the maintenance of these attributes is seldom trivial. If every application has its own set of attributes to control access control policies then the centralized directory provides only a negligible advantage. The attributes may as well reside in the applications themselves. And that's exactly how most deployments end up. Directory servers contain only the groups, groups that usually roughly approximate RBAC roles. Even LDAP standards themselves create a significant obstacle to interoperability in this case. There are at least three or four different and incompatible specifications for group definition in LDAP directories. The standard to manage LDAP groups is not ideal at all. It is especially problematic when managing big groups. Therefore, many directory servers provide their own non-standard improvements, which further complicates interoperability. Yet even these server-specific improvements usually cannot support complex access control policies. Therefore, access control policies and fine-grained authorizations are usually not centralized, they are maintained directly in the application databases.

The *single directory approach* is feasible only in very simple environments or in almost entirely homogeneous environments. In all other cases there is a need to supplement the solution by other identity management technologies.

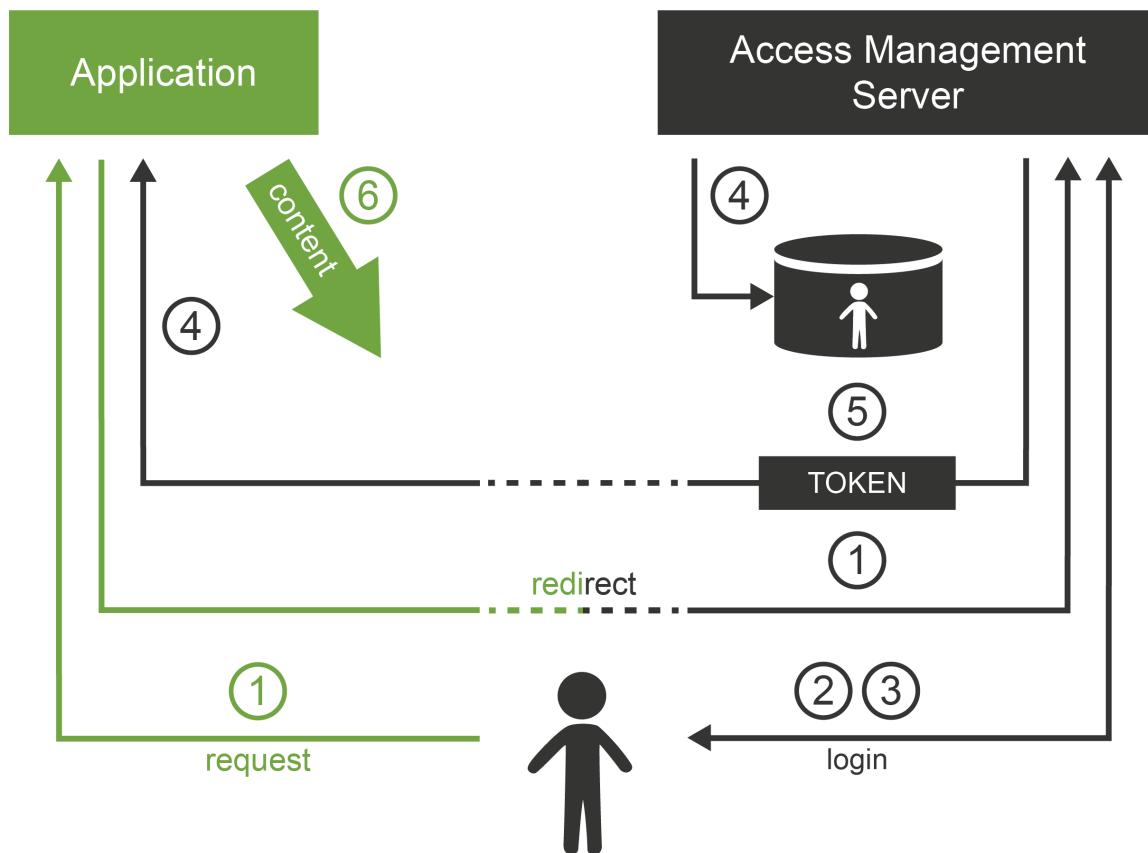
This does not mean that the directory servers or other shared databases are useless. Quite the contrary. They are very useful if they are used correctly. They just cannot be used alone. More components are needed to build a complete solution.

Access Management

While directory systems are not designed to handle complex authentication, *access management* (AM) systems are built to handle just that. Access management systems handle all the flavors of authentication, and even some authorization aspects. The principle of all access management systems is basically the same:

1. Access management system gets between the user and the target application. This can be done by a variety of mechanisms, the most common method is that the applications themselves redirect the user to the AM system if they do not have existing session.
2. Access management system prompts user for the username and password, interacts with authentication token, creates a challenge and prompts for the response or in any other way initiates the authentication procedure.
3. User enters the credentials.

4. Access management system checks the validity of credentials and evaluates access policies.
5. If access is allowed then the AM system redirects user back to the application. The redirection usually contains an access token: a small piece of information that tells the application that the user is authenticated.
6. Application validates the token, creates a local session and allows the access.



After that procedure, the user works with the application normally. Only the first access goes through the AM server. This is important for AM system performance and sizing, and it impacts session management functionality.

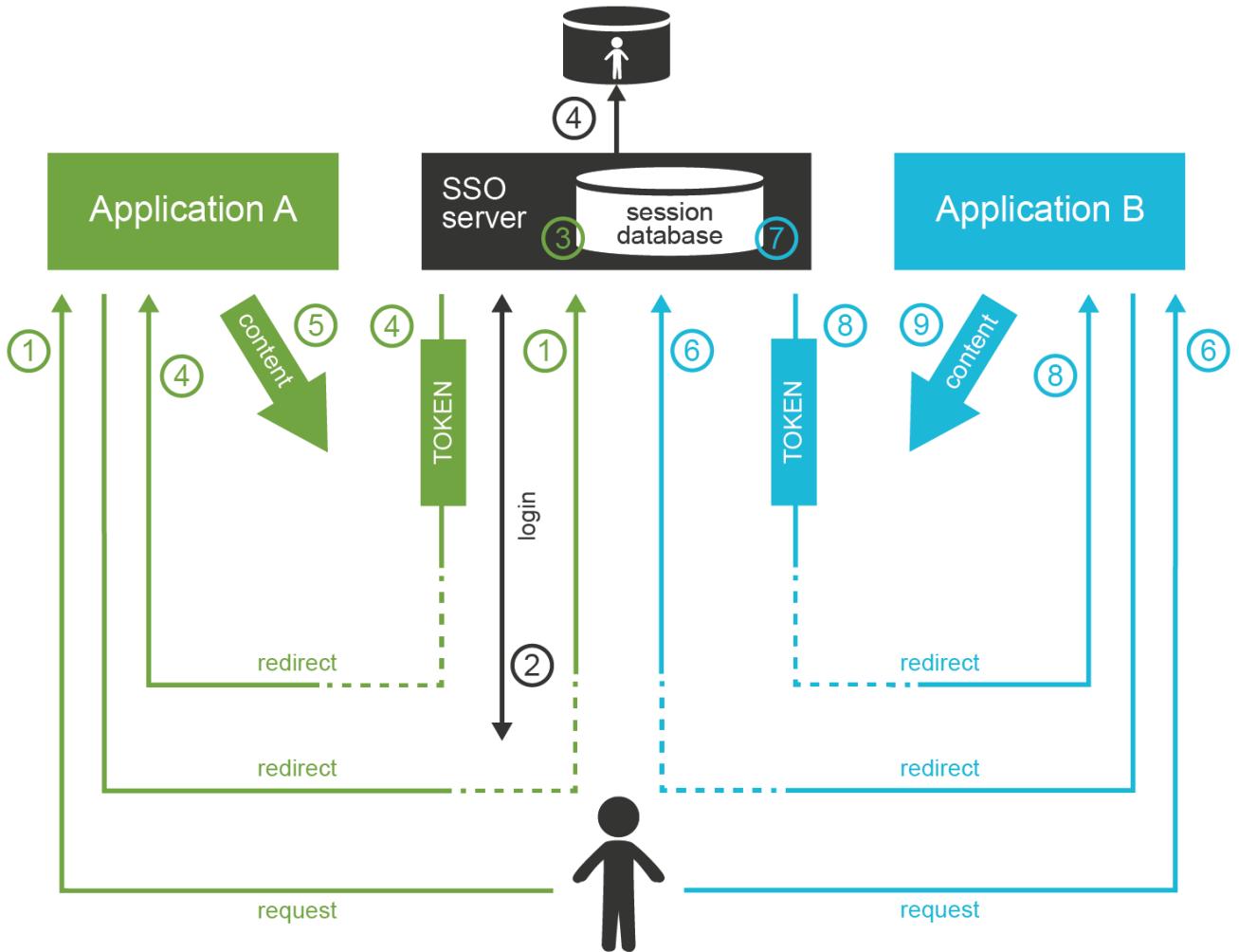
The applications only need to provide the code that integrates with the AM system. Except for that small integration code, applications do not need to provide any authentication code at all. It is the AM system that prompts for the password, not the application. This is a fundamental difference when compared to LDAP-based authentication mechanisms. In the LDAP case, it is the application that prompts for the password. In the AM case, the Access Management server does everything. Many applications do not even care how the user was authenticated. All they need to know is that he *was* authenticated and that the authentication was strong enough. This feature brings a very desirable flexibility to the entire application infrastructure. The authentication mechanism can be changed at any time without disrupting the applications. We live in an era when passwords are migrated to a stronger authentication mechanisms. The flexibility that the AM-based approach brings may play a key part in that migration.

Web Single Sign-On

Single Sign-On (SSO) systems allow user to authenticate once, and access number of different system after that. There are many SSO systems for web applications, however it looks like these systems are all using the same basic principle of operation. This general access management flow is described below:

1. Application A redirects the user to the access management server (SSO server).
2. The access management server authenticates the user.
3. The access management server establishes session (SSO session) with the user browser. This is the crucial part of the SSO mechanism.
4. User is redirected back to the application A. Application A usually establishes a local session with the user.
5. User interacts with application A.
6. When user tries to access application B, the application B redirects user to the access management server.
7. The access management server checks for existence of SSO session. As the user authenticated with the access management server before, there is a valid SSO session.
8. Access management server does not need to authenticate the user again and immediately redirects user back to application B.
9. Application B establishes a local session with the user and proceeds normally.

The user usually does not even realize that he was redirected when accessing application B. There is no interaction between the redirects and the redirects and the processing on the access management server is usually very fast. It looks like the user was logged into the application B all the time.



Authorization in Access Management

The request of a user accessing an application is directly or indirectly passed through the access management server. Therefore, the access management server can analyze the request and evaluate whether the user request is authorized or not. That is a theory. Unfortunately, the situation is much more complicated in practice.

The AM server usually intercepts only the first request to access the application because it would be a performance impact to intercept all the requests. After the first request, the application established a local session and proceeds with the operation without any communication with the AM server. Therefore the AM server can only evaluate authorization during the first request. This means it can only evaluate a very rough-grained authorization decisions. In practice, it usually means that the AM server can make only all-or-nothing authorization decisions: whether a particular user can access all parts of a particular application or that he cannot access the application at all. The AM server usually cannot make any finer-grain decisions just by itself.

Some AM systems provide agents that can be deployed to applications and that enforce a finer-grain authorization decisions. Such agents often rely on HTTP communication, they are making decisions based on the URLs that the user is accessing. This approach might have worked well in the 1990s, but it has only very limited applicability in the age of single-page web applications and

mobile applications. In such cases the authorization is usually applied to *services* rather than *applications*.

However, even applying the authorization to service front-ends does not solve the problem entirely. Sophisticated applications often need to make authorization decisions based on context which is simply not available in the request or user profile at all. E.g. an e-banking application may allow or deny a transaction based on the sum of previous transactions that were made earlier that day. While it may be possible to synchronize all the authorization information into the user profile, it is usually not desirable. It would be a major burden to keep such information updated and consistent, not to mention security concerns. Many authorization schemes rely on a specific business logic, which is very difficult to centralize in an authorization server.

Then there are implementation constraints. In theory, the authorization system should make only allow/deny decisions. However, this is not enough to implement an efficient application. The application cannot afford to list all the objects in the database, pass them to authorization server, and realize that the authorization server denied access to almost all of them. Authorization has to be processes *before* the search operation and additional search filters have to be applied. Which means that authorization mechanisms need to be integrated deep into the application logic. This significantly limits the applicability of centralized authorization mechanisms.

AM systems often come with a promise to unify authorization across all the applications and to centralize management of organization-wide security policies. Unfortunately, such broad promises are seldom fulfilled. The AM system can theoretically evaluate and enforce some authorization statements. This may work well during demonstrations and even in very simple deployments. Yet in complex practical deployments, this capability is extremely limited. The vast majority of the authorization decisions is carried out by each individual application and is completely outside of the reach of an AM system.

SAML and OpenID Connect

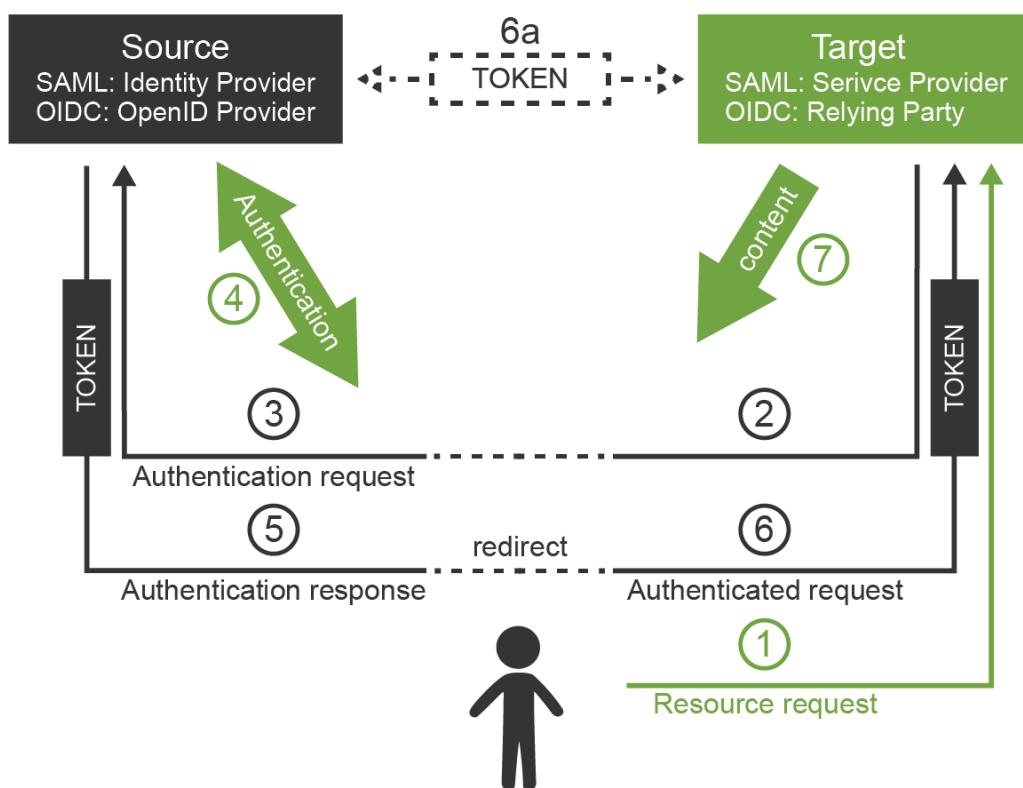
Some access management systems use proprietary protocols to communicate with the applications and agents. This is obviously an interoperability issue – especially when the AM principles are used in the Internet environment. Indeed, it is the Internet that motivated standardization in this field.

The first widespread standardized protocol in this field was Security Assertion Markup Language (SAML). The original intent of SAML was to allow cross-domain sign-on and identity data sharing across organizations on the Internet. SAML is both an access management protocol and a security token format. SAML is quite complex, heavily based on XML standards. Its specifications are long, divided into several profiles, there are many optional elements and features and overall SAML is a set of very rich and flexible mechanisms.

Primary purpose of SAML is transfer of identity information between organizations. There are big SAML-based federations with hundreds of participating organizations. Many e-government solutions are based on SAML, there are big partner networks running on SAML, and overall it looks like SAML is a success. Yet, SAML was a victim of its own flexibility and complexity. The latest fashion trends are not very favorable to SAML. XML and SOAP-based web service mechanisms are getting out of fashion, which impacts popularity of SAML. That has probably motivated the inception of other access management protocols.

The latest fashion favors RESTful services and simpler architectural approaches. All of that probably contributed to the development of OpenID Connect protocol (OIDC). OpenID Connect is based on much simpler mechanisms than SAML, but it is reusing the same basic principles. OpenID connect has a very eventful history. It all started with a bunch of homebrew protocols such as LID or SXIP, that are mostly forgotten today. That was followed by the development of OpenID protocol, which was still very simple. OpenID gained some attention especially with providers of Internet services. Despite its simplicity, OpenID was not very well engineered, and it quickly reached its technological limits. It was obvious that OpenID needs to be significantly improved. At that time, there was almost unrelated protocol called OAuth, which was designed for management of cross-domain authorizations. That protocol was developed into something that was almost, but not quite, entirely unlike the original OAuth protocol. As the result had almost nothing to do with the original OAuth protocol, it is perfectly understandable that it was dubbed OAuth2. In fact, OAuth2 is not really a protocol at all. It is rather a vaguely-defined framework to build other protocols. OAuth2 framework was used to build a cross-domain authentication and user profile protocol. This new protocol is much more similar to SAML than to the original OpenID, therefore it was an obvious choice to call it *OpenID Connect*. Some traditions are just worth maintaining.

Now there are two protocols that are using the same principle and doing almost the same thing. The principle is illustrated in the following diagram.



The interaction goes like this:

1. User is accessing a resource. This can be web page or web application on the target site.
2. Target site does not have a valid session for the user. Therefore it redirects user browser to the

source site. It will add authentication request into that redirect.

3. Browser follows the redirect to the source site. The source site gets the authentication request and parses it.
4. If the user is not already authenticated with the source site then the authentication happens now. The source site prompts for the username, password, certificate, one-time password or whatever credential that is required by the policy. With a bit of luck the authentication succeeds.
5. The source site redirects the browser back to the target site. The source site adds authentication response to the redirect. The most important part of the response is a token. The token directly or indirectly asserts user's identity.
6. The target site will parse the authentication response and process the token. The token may be just a reference to the real token (SAML artifact) or it may be access key to another service that provides the identity (OIDC UserInfo). In that case the target site makes another request (6a). This request is usually a direct one and does not use browser redirects. One way or another, the target site now has claims about user identity.
7. Target site evaluates the identity, processes authorizations and so on. A local session with the user is usually established at this point to skip the authentication redirects on the next request. The target site finally provides the content.

Following table compares the terminology and technologies used in SAML and OIDC worlds.

	SAML World	OpenID Connect World
Source site	Identity Provider (IdP)	Identity Provider (IDP) or OpenID Provider (OP)
Target site	Service Provider (SP)	Relying Party (RP)
Token	SAML Assertion (or artifact)	ID token, access token
Intended for	Web applications, web services (SOAP)	Web applications, mobile applications, REST services
Based on	N/A	OAuth2
Data representation	XML	JSON
Cryptography framework	XMLenc, XMLdsig	JOSE
Token format	SAML	JWT

Careful reader will notice the similarity with the web-based access management mechanisms. That's right. This is the same wheel reinvented over and over again. However, to be completely honest we have limited our description to cover flows for web browser only. Both SAML and OIDC has broader applicability than just web browser flows. And the differences between the protocols are much more obvious in these extended use cases. But the web browser case nicely illustrates the principles and similarities of SAML, OpenID Connect and also the simple web-SSO systems.

Maybe the most important differences between SAML, OIDC and web-SSO systems is the intended use:

- SAML was designed for the web applications and SOAP web services world. It will handle centralized (single-IDP) scenarios very well, but it can also work in decentralized federations. Go for SAML if you are using SOAP and WS-Security or if you plan to build big decentralized federation.
- OpenID Connect was designed mostly for use with social networks and similar Internet services. Its philosophy is still somehow centralized. It will work well if there is one strong identity provider and many relying parties. Technologically it will fit into RESTful world much better than SAML. Current fashion trends are favorable to OIDC.
- Web-SSO systems are designed to be used inside a single organization. This is ideal to implement SSO between several customer-facing applications so the customers will have no idea that they interact with many applications and not just one. The web-SSO systems are not designed to work across organizational boundaries.

Although SAML and OIDC are designed primarily for cross-domain use, it is no big surprise to see them inside a single organization. There is a clear benefit in using an open standardized protocol instead of a proprietary mechanism. However, it has to be expected that the SSO system based on SAML or OIDC will have slightly more complicated setup than a simple Web-SSO system.

Kerberos, Enterprise SSO and Friends

Many of us would like to think that everything is based on web technologies today and that non-web mechanisms are things of the past. Yet, there are still cases that are not web-based and where web-based SSO and AM mechanisms will not work. There is still a lot of legacy applications, especially in the enterprise environment. Applications based on rich clients or even character-based terminal interactions are still not that difficult to find. And then there are network operating systems such as Windows and numerous UNIX variants, there are network access technologies such as VPN or 802.1X and so on. There are still many cases where web-based access management and SSO simply won't work.

These technologies usually pre-date the web. Honestly, the centralized authentication and single sign-on are not entirely new ideas. It is perhaps no big surprise that there are authentication and SSO solutions even for non-web applications.

The classic classroom example of non-web SSO system is Kerberos. The protocol originated at MIT in the 1980s. It is a single sign-on protocol for operating systems and rich clients based on symmetric cryptography. Even though it is a cryptographic protocol it is not too complicated to understand and it definitely withstood the test of time. It has been used to this day, especially for authentication and SSO of network operating systems. It is a part of Windows network domain and it is often the preferred solution for authentication of UNIX servers. The most serious limitation of Kerberos is given by its use of symmetric cryptography. The weakness of symmetric cryptography is key management. Kerberos key management can be quite difficult especially when Kerberos realm gets very big. Key management is also one of the reasons why it is not very realistic to use Kerberos in cross-domain scenarios. However inside a closed organization Kerberos is still a very useful solution.

The major drawback in using Kerberos is that every application and client needs to be "kerberized". In other words everybody that wants to take part in Kerberos authentication needs to have Kerberos support in one's software. There are kerberized versions of many network utilities so this

is usually not a problem for UNIX-based networks. But it is a problem for generic applications. There is some support for Kerberos in common web browsers which is often referred to as "SPNEGO". However this support is usually limited to interoperability with Windows domains. Therefore even though Kerberos is still useful for operating system SSO it is not a generic solution for all applications.

Many network devices use RADIUS protocol for what network engineers call "Authentication, Authorization and Accounting" (AAA). However RADIUS is a back-end protocol. It does not take care of client interactions. The goal of RADIUS is that the network device (e.g. WiFi access point, router or VPN gateway) can validate user credentials that it has received as part of other protocol. The client connecting to VPN or WiFi network does not know anything about RADIUS. Therefore RADIUS is similar to the LDAP protocol and it is not really an access management technology.

Obviously there is no simple and elegant solution that can provide SSO for all enterprise applications. Despite that one technology appeared in the 1990s and early 2000s and promised to deliver universal enterprise SSO solution. It was called "Enterprise Single Sign-On" (ESSO). The ESSO approach was to use agents installed on every client device. The agent detects when login dialog appears on the screen, fills in the username and password and submits the dialog. If the agent is fast enough the user does not even notice the dialog and this creates the impression of Single Sign-On. However, there are obvious drawbacks. The agents need to know all the passwords in a cleartext form. There are ESSO variations with passwords randomly generated or even single-user passwords which partially alleviates this problem. But the drawback is that the ESSO also needs to be integrated with password management of all the applications, which is not entirely easy. However the most serious drawback of ESSO are the agents. These only work on workstations that are strictly controlled by the enterprise. Yet the world is changing, enterprise perimeter has efficiently disappeared, and the enterprise cannot really control all the client devices. Therefore also ESSO is now mostly a thing of the past.

Access Management and the Data

Access Management servers and Identity Providers need to know the data about users to work properly. But it is complicated. The purpose of access management systems is to manage access of users to the applications. Which usually means processing authentication, authorization (partially), auditing the access and so on. For this to work, the AM system needs access to the database where the user data are stored. It needs access to usernames, passwords and other credentials, authorization policies, attributes and so on. The AM systems usually do not store these data themselves. They rely on external databases. In most cases these databases are directory services or noSQL databases. This is an obvious choice: these databases are lightweight, highly available and extremely scalable. The AM system usually need just simple attributes, therefore the limited capabilities of directories and NoSQL databases are not a limiting factor here. Marriage of access management and lightweight database is an obvious and very smart match.

However, there is one critical issue – especially if the AM system is also used as a single sign-on server. The data in the directory service and the data in the applications must be consistent. E.g. it is a huge problem if one user has different usernames in several applications. Which username should he use to log in? Which username should be sent to the applications? There are ways how to handle such situations, but this is usually very cumbersome and expensive. It is much easier to unify the data before the AM system is deployed.

Even though the "M" in AM stands for "management", typical AM system has only a very limited data management capabilities. The AM systems usually assume that the underlying database is already properly managed. E.g. a typical AM system has only a very minimalistic user interface to create, modify and delete user records. Some AM systems may have self-service functionality (such as password reset), but even that functionality is usually very limited. Even though the AM relies on the fact that the data in the AM directory service and the data in applications are consistent there is usually no way how to fully synchronize the data by using the AM system itself. There may be methods for on-demand or opportunistic data updates, e.g. creating user record in the database when the user logs in for the first time. But there are usually no solutions for deleting the records or for updating the records of inactive users.

Therefore the AM systems are usually not deployed alone. The underlying directory service or NoSQL database is almost always a hard requirement for even humblest AM functionality. But for the AM system to really work properly it needs something to manage and synchronize the data. Identity Management (IDM) system is usually used for that purpose. In fact, it is usually strongly recommended deploying directory and IDM system before the AM system. The AM system cannot work without the data. And if the AM works with data that are not maintained properly, it will not take a long time until it fails.

Advantages and Disadvantages of Access Management Systems

Access management systems have significant advantages. Most of the characteristics are given by the AM principle of centralized authentication. As the authentication is carried out by a central access management server, it can be easily controlled and audited. Such centralization can be used to consistently apply authentication policies - and to easily change them when needed. It also allows better utilization of an investment into authentication technologies. E.g. multi-factor or adaptive authentication can be quite expensive if it has to be implemented by every application. But when it is implemented in the AM server, it is re-used by all the applications without additional investment.

However, there are also drawbacks. As the access management is centralized, it is obviously a single point of failure. Nobody is going to log in when the AM server fails. This obviously means major impact on functionality of all applications. Therefore AM servers need to be highly available and scalable. Which is not always an easy task. The AM servers need a very careful sizing as they may easily become a performance bottlenecks. However, perhaps the most severe drawback is the total cost of access management solution. The cost of the AM server itself is usually not a major issue. But the server will not work just by itself. The server needs to be integrated with every application. Even though there are standard protocols, the integration is far from being straightforward. Support for AM standards and protocols in the applications is still not universal. Especially older enterprise applications need to be modified to switch their authentication subsystem to the AM server. This is often so costly that the adoption of AM technologies is often limited just to a handful of enterprise applications. Although recent applications usually have some support for AM protocols, setting it up is still not an easy task. There are subtle incompatibilities and treacherous details, especially if the integration goes beyond mere authentication into authorization and user profile management.

Even though many organizations are planning deployment of an AM system as their first step in the IAM project, this approach seldom succeeds. The project usually plans to integrate 50-80%

applications into the AM solution. But the reality is that only a handful of applications can be easily integrated with the AM system. The rest of the applications is integrated using an IDM system that is hastily added to the project. Therefore it is better to plan ahead: analyze the AM integration effort, prototype the deployment, and make a realistic plan for the AM solution. Make sure the AM can really bring the promised benefits. Starting with IDM and adding AM part later is often much more reasonable strategy.

Homogeneous Access Management Myth

There are at least two popular access management protocols for the web. There are huge identity federations based on SAML. Cloud services and social networks usually use OpenID Connect or its variations. There are variations and related protocols to be used for mobile applications and services. Then there are other SSO protocols, primarily focused on intra-organizational use. There is no single protocol or mechanism that can solve all the problems in the AM world.

Additionaly, the redirection approach of AM systems assumes that the user has something that can display authentication prompts and carry out user interaction. Which is usually a web browser. Therefore, the original variant of access management mechanisms applies mostly to conventional web-based applications. Variations of this approach are also applicable to network services and single-page web applications. However, this approach is usually not directly applicable for applications that use rich clients, operating system authentication and similar "traditional" applications. Browser is not the primary environment that can be used to carry out the authentication in those cases. There are some solutions that usually rely on embedded browser, however that does not change the basic fact that the AM technologies are not entirely suitable for this environment. These applications usually rely on Kerberos as an SSO system or do not integrate with any SSO system at all.

Typical IT environment is composed of a wild mix of technologies and not all of them are entirely web-based. Therefore it is quite unlikely a single AM system can apply to everything that is deployed in your organization. Authentication is very tightly bound to the user interaction, therefore it depends on the method how the user interacts with the application. As the user is using different technologies to interact with the web application, mobile application and operating system then it is obvious that also authentication and SSO methods for these systems will be different.

Therefore it has to be expected that there will be several AM or SSO systems in the organization, each one serving its own technological island. And each island needs to be managed.

Practical Access Management

Unifying access management system, Single Sign-On, cross-domain identity federation, social login, universally-applicable 2-factor authentication – there are the things that people usually want when they think about Identity and Access Management (IAM). These are all perfectly valid requirements. However, everything has its cost. It is notoriously difficult to estimate the cost of access management solutions, because majority of the cost is not in the AM software. Huge part of the total cost is hidden inside existing applications, services and clients. All of this has to be considered when planning an access management project.

Even though the AM is what people usually want, it is usually wise **not** to start with AM as the first

step. AM deployment has many dependencies: unified user database, managed and continually synchronized data, applications that are flexible enough to be integrated and so on. Unless your IT infrastructure is extremely homogeneous and simple, it is very unlikely that these dependencies are already satisfied. Therefore it is almost certain that an AM project attempted at the beginning of the IAM program will not reach its goals. It is much more likely for such AM projects to fail miserably. On the other hand, if the AM project is properly scoped and planned and has realistic goals, there is high chance of success.

Perhaps the best way to evaluate an AM project is to ask several questions:

- Do I really need access management for all applications? Do I need 100% coverage? Can I afford all the costs? Maybe it is enough to integrate just a couple of applications that are source of the worst pain. Do I know which applications are these? Do I know what my users really use during they workday? Do I know what they need?
- What are the real security benefits of AM deployment? Will I be disabling the native authentication to the applications? Even for system administrators? What will I do in case of administration emergencies (e.g. system recovery)? Would system administrators still be able to circumvent the AM system? If yes then what is the real security benefit? If not then what will be the recovery procedure in case the AM system fails?
- Do I really need SSO for older and rarely used applications? What is the real problem here? Is the problem that users are entering the password several times per day? Or is the real problem that they have to enter a different username or password to different applications, and they keep forgetting the credentials? Maybe simple data cleanup and password management will solve the worst problems, and I can save a huge amount of money on AM project?

The access management technologies are the most visible part of the IAM program. But it is also the most expensive part, and the most difficult piece to set up and maintain. Therefore do not underestimate other IAM technologies. Do not try to solve every problem with AM golden hammer. Using the right tool for the job is a good approach in every situation. But in IAM program, it is absolutely critical for success.

Identity Management

Identity management (IDM) is maybe the most overlooked and underestimated technology in the whole identity and access management (IAM) field. Yet IDM is a crucial part of almost every IAM solution. It is IDM that can bring substantial benefits to almost any organization. So, what that mysterious IDM thing really is?

Identity management is exactly what the name says: it is all about managing identities. It is about the processes to create Active Directory accounts and mailboxes for a new employee. IDM sets up accounts for students at the beginning of each school year. IDM makes it possible to immediately disable all access to a suspicious user during a security incident. IDM takes care of adding new privileges and removing old privileges of users during reorganization. IDM makes sure all the accounts are properly disabled when the employee leaves the company. IDM automatically sets up privileges for students and staff appropriate for their classes. IDM records access privileges of temporary workers, partners, support engineers and all the third-party identities that are not maintained in your human resources (HR) system. IDM automates the processes of role request and approval. IDM records every change in user privileges in the audit trail. IDM governs the annual

reviews of roles and access privileges. IDM makes sure the copies of user data that are kept in the applications are synchronized and properly managed. IDM makes sure data are managed according to data protection rules. And IDM does many other things that are absolutely essential for every organization to operate in an efficient and secure manner.

It looks like IDM is the best thing since the sliced bread. So where's the catch? Oh yes, there is a catch. Or it is perhaps better to say that there was a catch. The IDM systems used to be expensive. Very expensive. The IDM systems used to be so expensive, it was very difficult to justify the cost even with such substantial and clear benefits. But that time is over now.

Terminology.

The term *identity management* is often used for the whole identity and access management (IAM) field. This is somehow confusing because technologies such as single sign-on or access management do not really manage the identities. Such technologies manage the access to the applications. Even directory servers do not exactly *manage* the identities. Directory servers store the identities and provide access to them. There is in fact one whole branch of technologies that manage identities. Those systems are responsible for creating identities and maintaining them. Those are sometimes referred to as *identity provisioning*, *identity lifecycle management* or *identity administration systems*. But given the current state of the technology such names are indeed an understatement. Those systems can do much more than just provisioning or management of identity lifecycle. We will refer to these systems simply as *identity management* (IDM) systems. When we refer to the entire field that contains access management, directory services, identity management and governance we will use the term *identity and access management* (IAM).



History of Identity Management

Let's start at the beginning. In the 1990s there was no technology that would be clearly identified as "identity management". Of course, all the problems above had existed almost since the beginning of modern computing. There had always been some solutions for those problems. Historically, most of that solutions were based on paperwork and scripting. That worked quite well - until the big system integration wave spread through the industry in the 1990s and 2000s. As data and processes in individual applications got integrated, the IDM problems became much more pronounced. Manual paper-based processes were just too slow for the age of information superhighways. The scripts were too difficult to maintain in the world where new application is deployed every couple of weeks. The identity integration effort naturally started with the state-of-the-art identity technology of the day: directory services. As we have already shown, the directories were not entirely ideal tools for the job. The directories did not work very well in environment where people thought that LDAP is some kind of dangerous disease, where usernames and identifiers were assigned quite randomly and where every application insisted that the only authoritative data are those stored in its own database.

The integration problems motivated the inception of IDM technologies in early 2000s. Early IDM systems were just data synchronization engines that were somehow hard-coded to operate with users and accounts. Some simple Role-Based Access Control (RBAC) engines and administration interfaces were added a bit later. During mid-2000s there were several more-or-less complete IDM

systems. This was the first generation of the IDM systems. These systems were able to synchronize identity data between applications and provide some basic management capabilities. Even such a simple functionality was a huge success at that time. The IDM systems could synchronize the data without any major modification of the applications, therefore they brought the integration cost to a reasonable level. The problem was that the cost of the IDM systems themselves was quite high. These systems were still somehow crude, therefore the configuration and customization required a very specialized class of engineers. IDM engineers were almost exclusively employed by IDM vendors, big system integrators and expensive consulting companies. This made the deployment of IDM solutions prohibitively expensive for many mid-size and smaller organizations. Even big organizations often deployed IDM solution with quite limited features to make the cost acceptable.

Early IDM systems evolved and improved in time. There were companion products for identity governance and compliance that augmented the functionality. Yet, it is often almost impossible to change the original architecture of a product. Therefore many first-generation IDM products struggle with limitations of the early product design to this day.

All the first-generation IDM systems were commercial closed-source software. Many of these products are still available on the market, and they are even considered to be leaders. However, the closed-source character of the IDM products is itself a huge problem. Every IDM solution has to be more-or-less customized. Which usually means more rather than less. It has to be the IDM system that adapts, and not the applications. Requiring each application to adapt to a standardized IDM interface means a lot of changes in a lot of different places, platforms and languages. The total cost of all necessary modifications adds up to a huge number. Such approach is being tried from time to time, but it almost always fails. It is not a practical approach. While there are many applications in the IT infrastructure, there is just one IDM system. If the IDM system adapts to applications and business processes, the changes are usually smaller, and they are all in one place, implemented in a single platform. The IDM system must be able to adapt. It has to adapt a great deal, and it has to adapt easily and rapidly. Closed-source software is notoriously bad at adapting to requirements that are difficult to predict. Which in practice means that the IDM projects based on first-generation products were hard to use, slow to adapt and expensive. The closed-source software is also prone to vendor lock-in. Once the IDM system is deployed and integrated, it is extremely difficult to replace it with a competing system. The closed-source vendor is the only entity that can modify the system, and the system cannot be efficiently replaced. Which means that the end customer is not in a position to negotiate. Which means high maintenance costs. It naturally follows that the first generation of IDM systems was a huge commercial success. For the vendors, that is.

Then the 2000s were suddenly over, with an economic crash at the end. We can only speculate what were the reasons, but the fact is that around the years 2009-2011 several very interesting new IDM products appeared on the market. One interesting thing is that all of them were more-or-less *open source*. The benefit that open source character brings may be easy to overlook for business-oriented people. However, the benefits of open source in the identity management are almost impossible to overstate. As every single IDM engineer knows, understanding of the IDM product, and the ability to adapt the product, are two critical aspects of any IDM project. Open source is the best way to support both understanding and flexibility. There is also third important advantage: it is almost impossible to create a vendor lock-in situation with an open source product. All the open source products are backed by companies that offer professional support services that are equivalent to the services offered by commercial IDM products. This brings quality assurance for the products and related services. However, the companies does not really "own" the products, there is no way

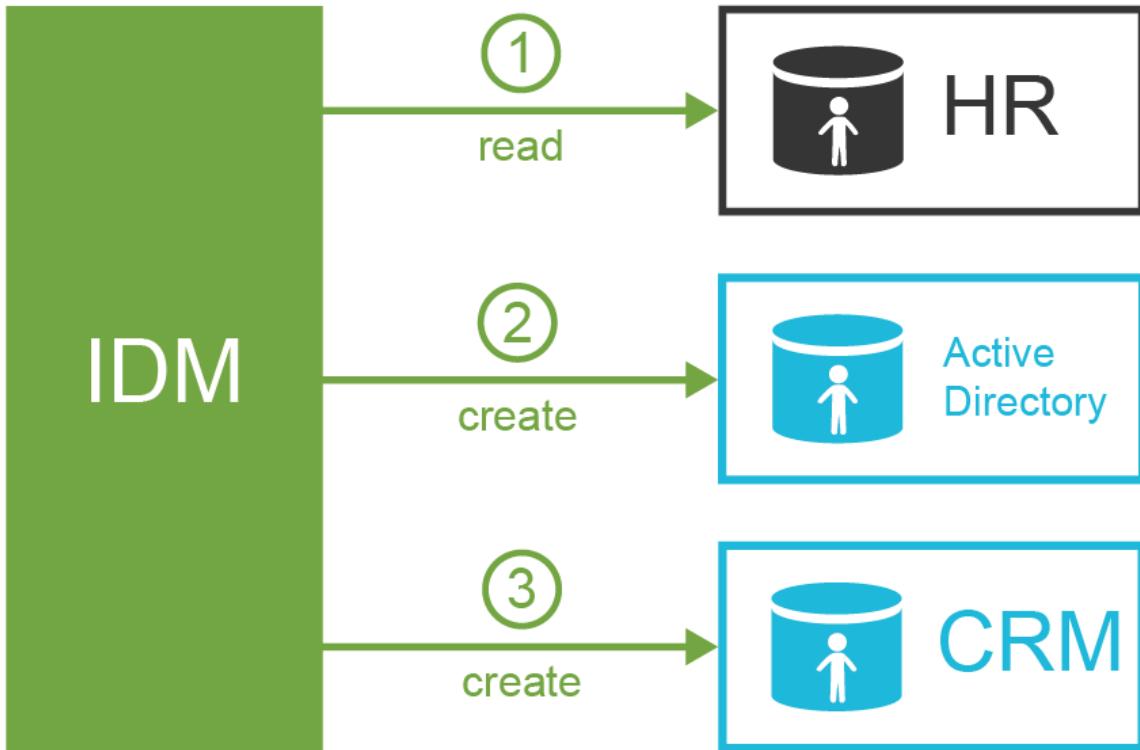
for them to abuse intellectual property rights against the customers. Open source brings new and revolutionary approach, both to technology and business.

What is This Identity Management, Anyway?

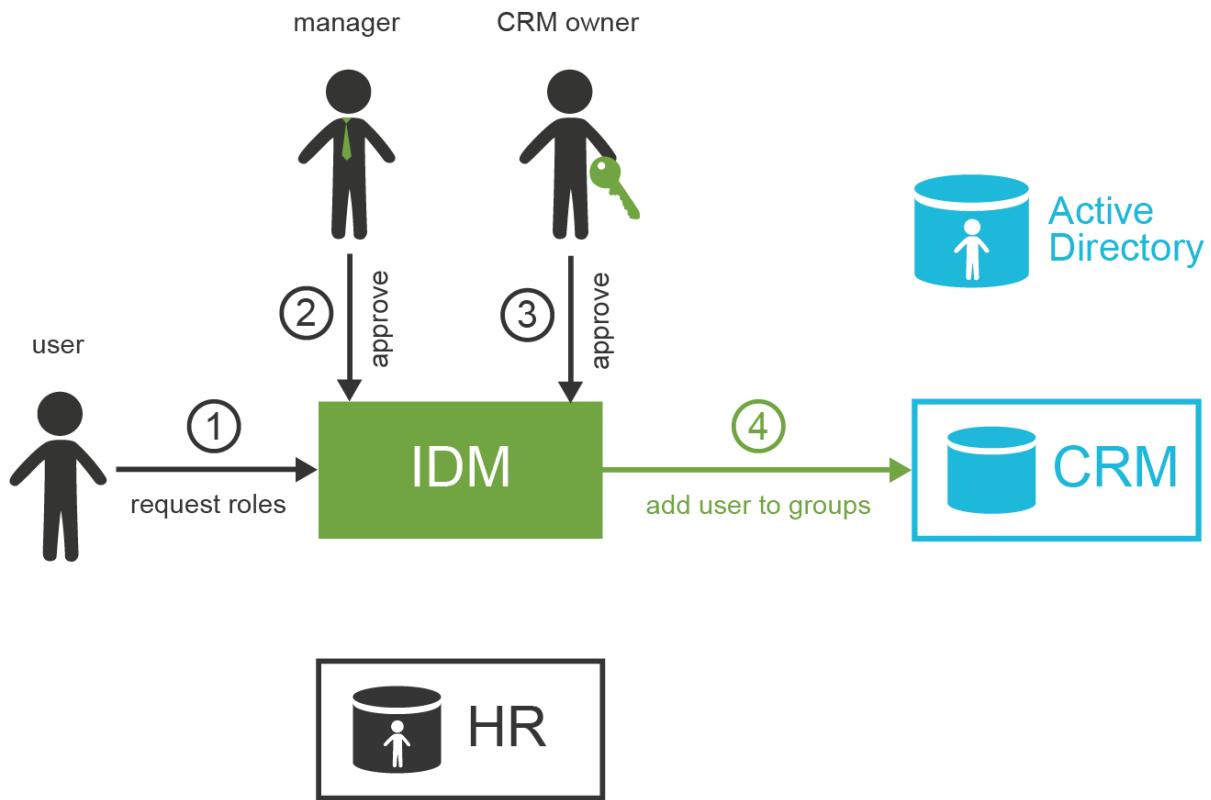
Identity management is a simple term which encompasses a very rich and comprehensive functionality. It contains identity provisioning (and reprovisioning and deprovisioning), synchronization, organizational structure management, role-based access control, data consistency, approval processes, auditing and few dozens of other features. All of that is thoroughly blended and mixed with a pinch of scripting and other seasoning until there is a smooth IDM solution. Therefore, it is quite difficult to tell what identity management is just by using a dictionary-like definition. We would rather describe what identity management is by using a couple of typical usage scenarios.

Let's have a fictional company called ExAmPLE, Inc. This company has few thousand employees, decent partner network, customers and suppliers and all the other things as real-world companies have. And ExAmPLE company has an IDM system running in its IT infrastructure.

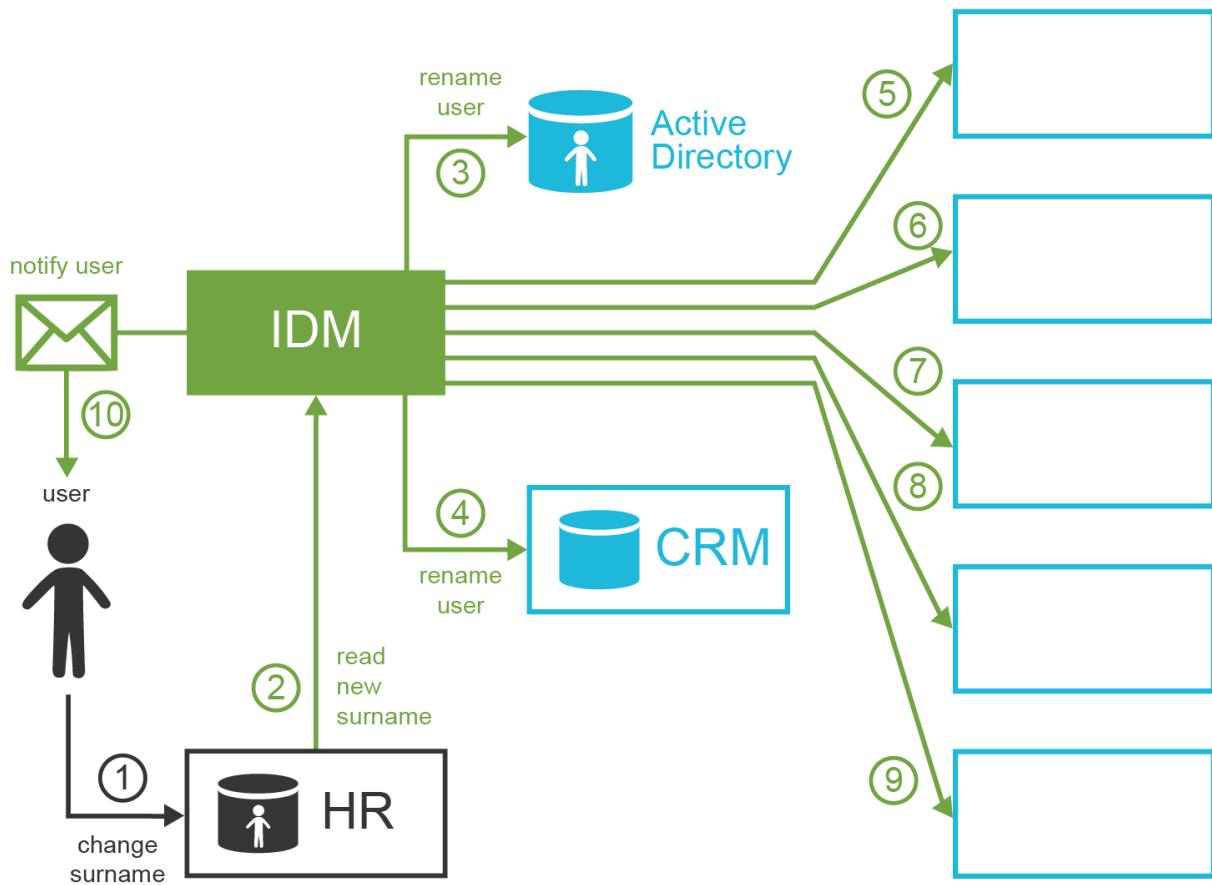
ExAmPLE hires a new employee called Alice. Alice signs an employee contract few days before she starts her employment. The contract is entered into the HR system by the ExAmPLE HR staff. The IDM system periodically scans the HR records, and it discovers the record of a new hire. The IDM systems pulls in the record and analyzes it. The IDM system will take user's name and employee number from the HR record, it will generate a unique username and based on that information it creates a user record in the IDM system. The IDM system also gets the organization code of **11001** from the HR record. The IDM will look inside its organizational tree and discovers that the code **11001** belongs to sales department. Therefore IDM will automatically assign the user to the sales department. The IDM will also process the work position code of **S007** in the HR record. The IDM policies say that the code **S007** means sales agent and that anybody with that code should automatically receive the "Sales Agent" role. Therefore the IDM will assign that role. As this is a core employee, the IDM will automatically create an Active Directory account for the user together with the company mailbox. The account will be placed into the Sales Department organizational unit. The "Sales Agent" role entitles the user to more privileges. Therefore the Active Directory account is automatically assigned to sales groups and distribution lists. The role also gives access to the CRM system, therefore CRM account is also automatically created and assigned to appropriate groups. All of that happens in a couple of seconds after the new HR record is detected. It all happens automatically.



Alice starts her career, and she is a really efficient employee. Therefore she gets more responsibilities. Alice is going to prepare specialized market analyses based on empirical data gathered in the field. ExAmPLE is a really flexible company, always inventing new ways how to make business operations more efficient. Therefore they invented this work position especially to take advantage of Alice's skills. Which means there is no work position code for Alice's new job. But she needs new privileges in the CRM system to do her work efficiently. She needs that right now. Fortunately the ExAmPLE has a flexible IDM system. Alice can log into the IDM system, select the privileges that she needs and request them. The request has to be approved by Alice's manager and by the CRM system owner too. They get the notification about the request, and they can easily approve or reject it in the IDM system. Once the request is approved Alice's CRM account will be automatically assigned to appropriate CRM groups. Alice may start working on her analysis minutes or hours after she has requested the privileges.

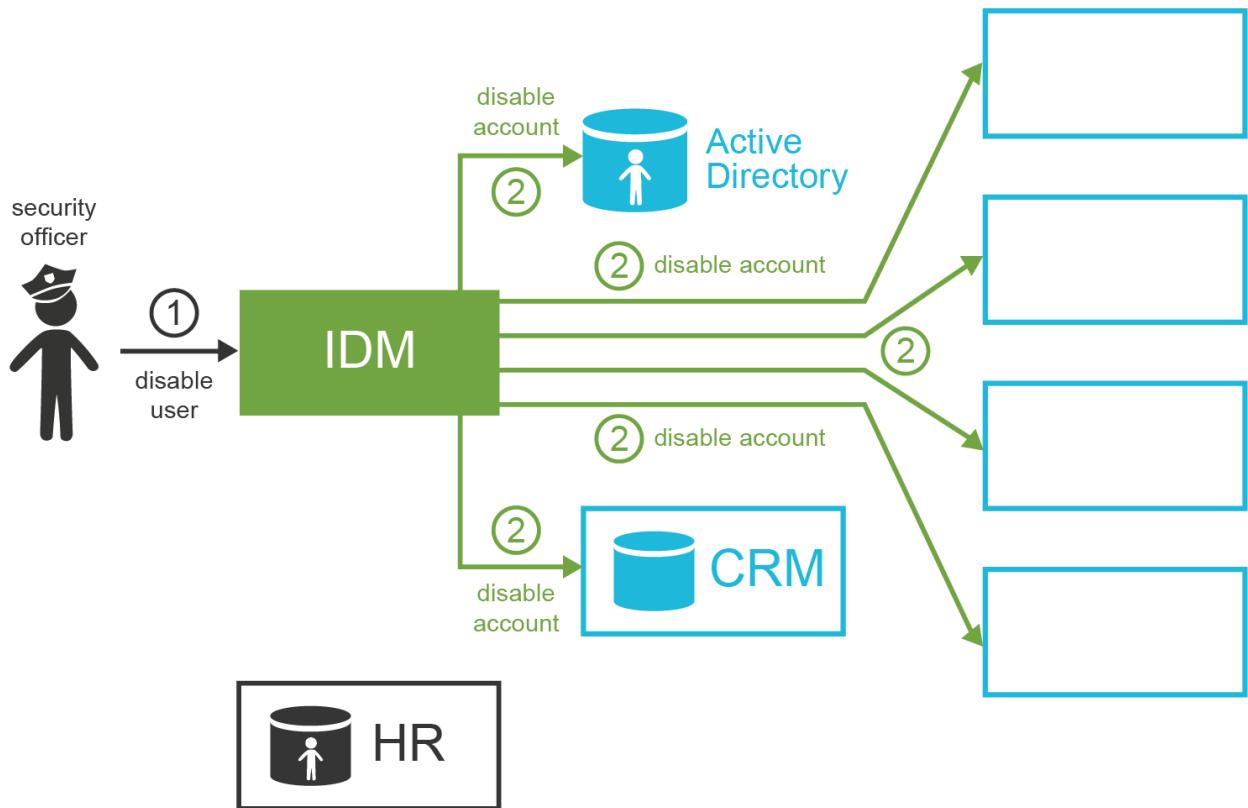


Alice lives happily ever after. One day she decides to get married. Alice, similarly to many other women, has the strange habit of changing her surname after the marriage. Alice has a really responsible work position now, she has accounts in a dozen information systems. This is no easy task to change her name in all of them, is it? In fact, it is very easy because ExAmPLE has its IDM system. Alice goes to the HR department, and the HR staff changes her surname in the HR system. The IDM system will pick up the change and propagate that to all the affected systems. Alice even automatically gets a new e-mail address with her new surname (keeping the old one as an alias). Alice receives a notification that now she can use her new e-mail address. The change is fast, clean and effortless.



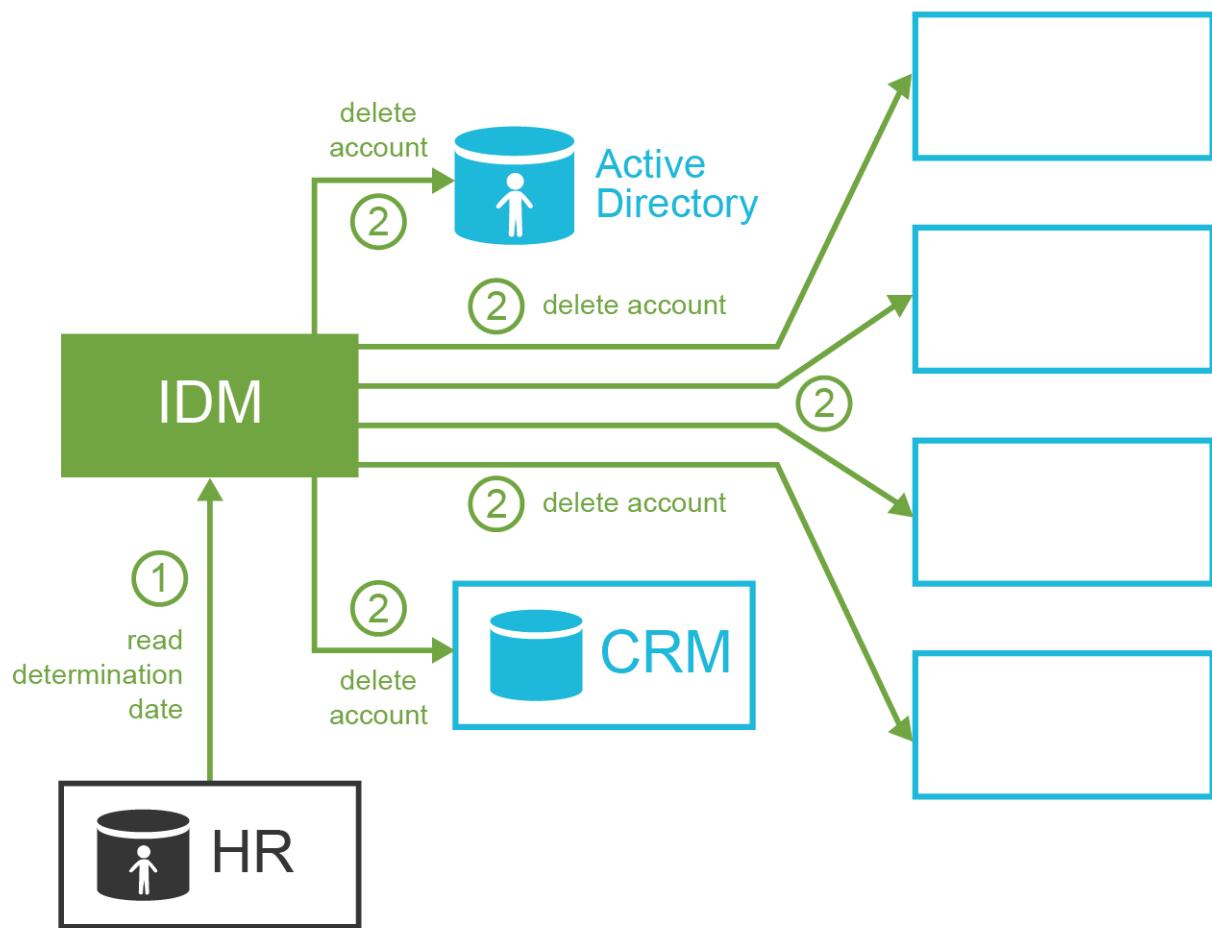
Later that day Alice discovers that her password is about to expire. Changing the password in all the applications would be a huge task. But Alice knows what to do. She logs into the IDM system and changes her password there. The password change is automatically propagated to each affected system according to policy set up by the IT security office.

The following month something unexpected happens. There is a security incident. The security office discovered the incident and now they are investigating it. It looks like it was an insider job. The security officers are using the data from the IDM system to focus their investigation on users that had privileges to access affected information assets. They pinpoint Mallory as a prime suspect. The interesting thing is that Mallory should not have these privileges at all. Luckily the IDM system also keeps an audit trail about every privilege change. Therefore they discover that it was Mallory's colleague Oscar that assigned these privileges to Mallory. Both men are to be interviewed. But as this incident affects sensitive assets there are some preventive measures to be executed before any word about the incident spreads. The security officers use the IDM system to immediately disable all the accounts that Mallory and Oscar have. It takes just a few seconds for IDM to disable these accounts in all the affected applications.

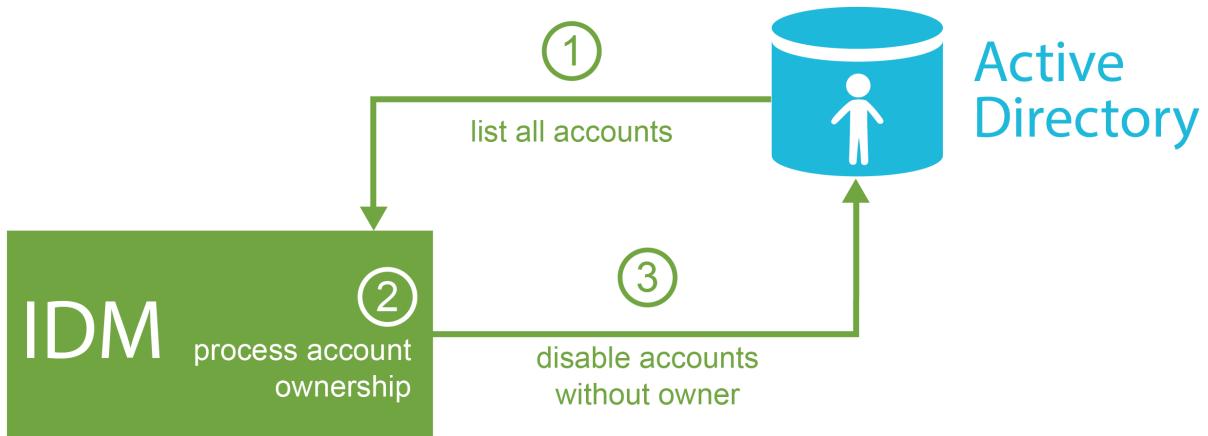


The investigation later reveals that Oscar is mostly innocent. Mallory misused Oscar's trust and tricked him to assign the extra privileges. Mallory abused the privileges to get sensitive data and he tried to sell them. The decision is that Mallory has to immediately leave the company while Oscar may stay. However, as Oscar has shown poor judgment in this case his responsibilities are reduced. The IDM is now used to permanently disable all Mallory's accounts, to re-enable Oscar's accounts and also to revoke sensitive privileges that are considered too risky for Oscar to have.

Few months later Oscar is still ashamed because of his failure. He decides not to prolong his employee contract with ExAmPLE and to leave the company without causing more trouble. Oscar's contract expires at the end of the month. This date is recorded in the HR system and the IDM system takes it from there. Therefore at midnight of the last Oscar's day at work the IDM system automatically deletes all Oscar's accounts. Oscar starts a new career as a barman in New York. He is very successful.



The security office has handled the security incident professionally and the IDM system provided crucial data to make the security response quick and efficient. They receive praise from the board of directors. But the team always tries to improve. They try to learn from the incident and reduce the possibility of such a thing happening again. The team is using data from the IDM system to analyze the privileges assigned to individual users. The usual job of the IDM system is to create and modify accounts in the applications. But the IDM system is using bidirectional communication with the applications. Therefore this analysis is far from being yet another pointless spreadsheet exercise. The analysis is based on real application data processes and unified by the IDM system: what are the real accounts, to which user they belong, what roles they have, which groups they belong and so on. The IDM system can detect accounts that do not have any clear owner. The security team discovers quite a rich collection of testing accounts that were obviously used during the last data center outage half a year ago. The IT operations staff obviously forgot about these accounts after the outage. The security staff disables the accounts using the IDM tools and sets up an automated process to watch out for such accounts in the future.



Based on the IDM data the security officers suspect that there are users that have too many privileges. This is most likely a consequence of the request-and-approval process and these privileges simply accumulated over time. But this is just a suspicion. It is always difficult for a security staff to assess whether particular user should have certain privilege or should not have it. This is especially difficult in flexible organizations such as ExAmPLE, where work responsibilities are often cumulated and organizational structures is somehow fuzzy. Yet there are people that know what each employee should do: the managers. However, there are many managers on many departments and it would be a huge task to talk to each one of them and consult the privileges. The IDM system comes to the rescue once again. The security officers set up automated access recertification campaign. They sort all users to their managers based on the organizational structure which is maintained in the IDM system. Each manager will receive an interactive list of their users and their privileges. The manager must confirm (re-certify) that the user still needs those privileges. This campaign is executed in a very efficient manner as the work is evenly distributed through the organization. Therefore the campaign is completed in a couple of days. At the end the security officers know which privileges are no longer needed and can be removed. This reduces the exposure of the assets which is a very efficient way to reduce residual security risk.

i Experienced identity management professionals certainly realized that this description is slightly idealized. The real world is not a fairy tale and real life with an IDM system is much more complicated than this simple story suggests. Even though the real life is harder than a story in a book, the IDM system remains an indispensable tool for automation and information security management.

How Does The Technology Work?

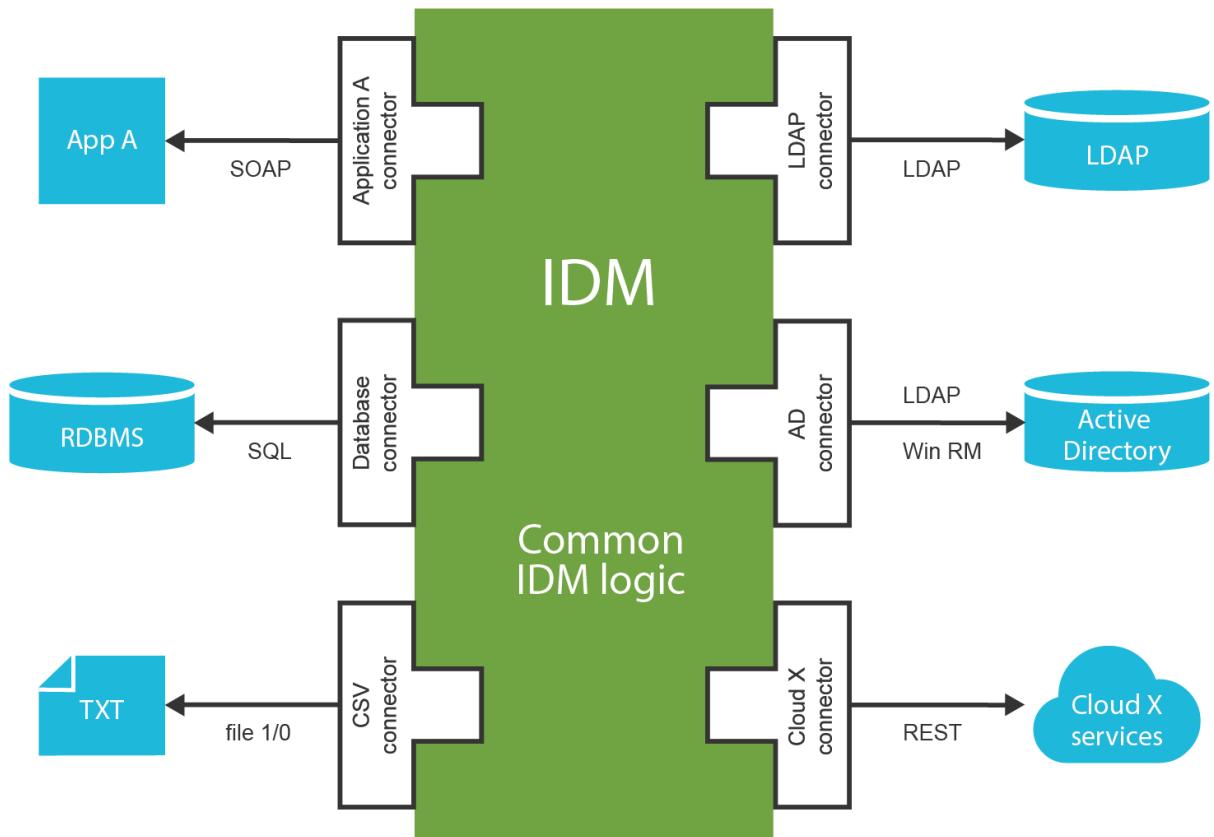
Obviously identity management systems have a lot of advantages for business, processes, efficiency and all that stuff. But how does it really works on a technological level? The basic principle is very simple: identity management system is just a sophisticated data synchronization engine.

Identity management system takes data from the source systems, such as HR databases. It is processing the data, mapping and transforming the values as necessary. It will figure out which records are new. The IDM engine will do some (usually quite complex) processing on the records. That usually includes processing policies such as Role-Based Access Control (RBAC), organizational policies, password policies and so on. The result of this processing is creation or modification of user accounts in other systems such as Active Directory, CRM systems and so on. So basically it is all about getting the data, changing them and moving them around. This does not seem very revolutionary, does it? But it is all about the details. It is the way how the IDM system gathers the data, how it is processing the data and how it is propagating the changes that make all the difference.

Identity Management Connectors

Identity management system must connect to many different applications, databases and information systems. Typical IDM deployment has tens or even hundreds of such connections. Therefore the ease of connecting IDM system with its environment is one of its essential qualities.

Current IDM systems use *connectors* to communicate with all surrounding systems. These connectors are based on similar principles that database drivers. On one end there is unified connector interface that presents that data from all the systems using the same "protocol". On the end of the connector is the native protocol that the application supports. Therefore there are connectors for LDAP and various LDAP variants, SQL protocols and dialects, connectors that are file-based, connectors that invoke web services or REST services and so on. Every slightly advanced IDM system has tens of different connects.



Connector is usually relatively simple piece of code. Primary responsibility of a connector is to adapt communication protocols. Therefore LDAP connector translates the LDAP protocol messages into data represented using a common connector interface. The SQL connector does the same thing with SQL-based protocols. The connector also interprets the operations invoked on the common connector interface by the IDM system. Therefore the LDAP protocol will execute the "create" operation by sending LDAP "add" message to the LDAP server and parsing the reply. Connectors usually implement the basic set of create-read-update-delete (CRUD) operations. Therefore a typical connector is quite a simple piece of code. Despite its simplicity the whole connector idea is a clever one. The IDM system does not need to deal with the communication details. The core of the IDM system can be built to focus on the generic identity management logic which is typically quite complex just by itself. Therefore any simplification that the connectors provide is more than welcome.

Connectors are usually accessing external interfaces of source and target systems. It is natural that the connector authors will choose interfaces that are public, well-documented and based on open standards. Many newer systems have interfaces like that. But there are notorious cases that refuse to provide such an interface. Despite that there is almost always some way to build a connector. The connector may create record directly in the application database. Or it may execute a database routine. Or it may execute a command-line tool for account management. Or it may even do crazy things such as simulation of a user working with text terminal and filling out a form to create new account. There is almost always a way to do what connector needs to do. Just some ways are nicer than others.

The connector-based architecture is pretty much standard among all advanced IDM systems. Yet the connector interfaces significantly vary from one IDM system to another. Therefore the

connectors are not interchangeable between different IDM systems. The connector interfaces are all proprietary. And the connectors are often used as weapons to somehow artificially increase the profit from IDM solution deployment. Except for one case. The ConnId connector framework is the only connector interface that is actively used and developed by several competing IDM systems. It is perhaps no big surprise that ConnId is an open source framework.

Even though connector-based approach is quite widespread, some older IDM systems are not using connectors. Some IDM products use agents instead of connectors. Agent does a similar job than the connector does. However, agent is not part of the IDM system instance. Agents are installed in each connected application and they communicate with the IDM system using a remote network protocol. This is a major burden. The agents need to be installed everywhere. And then they need to be maintained, upgraded, there may be subtle incompatibilities and so on. Also, running a third-party code inside every application can be a major security issue. Overall the agent-based systems are too cumbersome (and too costly) to operate. The whole agent idea perhaps originated somewhere in our digital past when applications and databases haven't supported any native remote interfaces. In such a situation the agents are obviously better than connectors. Fortunately, this is a thing of the past. Today even old applications have some way to manage identities using a remote interface. This is typically some web or REST service that is easy to access from a connector. But even if the application provides only a command-line interface or interactive terminal session there are connectors that can handle that sufficiently well. Therefore today the agent-based systems are generally considered to be obsolete.

Identity Provisioning

Provisioning is perhaps the most frequently used feature in any IDM system. In the generic sense *provisioning* means maintenance of user accounts in applications, databases and other target systems. This includes creation of the account, various modifications during the account lifetime and permanent disable or delete at the end of the lifetime. The IDM system is using *connectors* to manipulate the accounts. And in fact good IDM systems can manage much more than just accounts. Management of groups and group membership was quite a rare feature in early years of IDM technology. Yet today an IDM system that cannot manage groups is almost useless. Almost all IDM systems work with roles. But only few IDM systems can also provision and synchronize the roles (e.g. automatically create LDAP group for each new role). Good IDM system can also manage, provision and synchronize organizational structures. However, this feature is still not entirely common.

Synchronization and Reconciliation

Identity provisioning may be the most important feature of an IDM system. But if an IDM system did just the provisioning and nothing else it would be a quick and utter failure. It is not enough to create an account when a new employee is hired or delete that account when an employee leaves. Reality works in mysterious ways and it can easily make a big mess in a very short time. Maybe there was a crash in one of the applications and the data were restored from a backup. So an account that was deleted few hours ago is unexpectedly resurrected. It stays there, alive, unchecked and dangerous. Maybe an administrator manually created an account for a new assistant because the HR people were all busy to process the papers. And the new assistant had such pretty eyes. When the record finally gets to the HR system and it is processed the IDM system discovers that there is already a conflicting account and it simply stops with an error. Maybe few (hundred)

accounts get accidentally deleted by junior system administrator trying out an innovative system administration routine. There are simply too many ways how things can go wrong. And in reality they do go wrong surprisingly often. It is not enough for an IDM system to just set things up and then forget about it. One of the most important features of any self-respecting IDM system is to make sure that everything is right and also that it stays right all the time. Identity management is all about continuous maintenance of the identities. Without that continuity the whole IDM system is almost useless.

The trick to keep the data in order is to know when they get out of order. In other words, the IDM system must detect when the data in the application databases change. If an IDM system detects that there was a change then it is not that difficult to react to the change and fix it. The secret ingredient is the ability to detect changes. But there's a slight issue with that, isn't it? We cannot expect that the application will send a notification to the IDM system every time a change happens. We do not want to modify the applications, otherwise the IDM deployment will be prohibitively expensive. The application needs to be passive and the IDM system needs to be active. Fortunately, there are several ways how to do that.

Some applications already keep a track of the changes. Some databases record a timestamp of the last change for each row. Some directory servers keep a record of recent changes for the purpose of data replication. Such meta-data can be used by the IDM system. The IDM system may periodically scan the timestamps or replication logs for new changes. When the IDM detects a change it can retrieve the changed objects and react to the change based on its policies. The scanning for changes based on meta-data is usually very efficient therefore it can be executed every couple of minutes. Therefore the reaction to the change can be done almost in the real-time. This method has many names in various IDM systems. It is called "live synchronization", "active synchronization" or simply just "synchronization". Sadly, this method is not always available. In fact this ability is quite rare.

But all is not lost. Even if the application does not maintain good meta-data that allow near-real-time change detection there is still one very simple way that works for almost any system. The IDM system gets the list of all accounts in the application. Then it compares that list with the list of accounts that are *supposed* to be there. Therefore it compares the reality (what *is* there) with the policy (what *should be* there). The IDM system can react to any discrepancies and repair them. This method is called *reconciliation*. It is quite a brutal method, almost barbaric. But it does the job.

Listing all accounts and processing each of them may seem as a straightforward job. But it can be extremely slow if the number of accounts is high and the policies are complex. It can take anything from a few minutes to a few days. Therefore it cannot be executed frequently. Running that once per day is feasible only for small and simple systems. Running it once per week (on weekends) is a more common practice. But many systems cannot afford to run it more frequently than once per month.

There are also other methods. But synchronization and reconciliations are the most frequently used. The drawback of synchronization is that it is not entirely reliable. The IDM system may miss some changes, e.g. due to change log expiration, system times not being synchronized or variety of other reasons. On the other hand, reconciliation is mostly reliable. But it is a very demanding task. Therefore these two methods are often used together. Synchronization runs all the time and handles the vast majority of the changes. Reconciliation runs weekly or monthly and it acts as a safety net to catch the changes that might have escaped during synchronization.

Identity Management and Role-Based Access Control

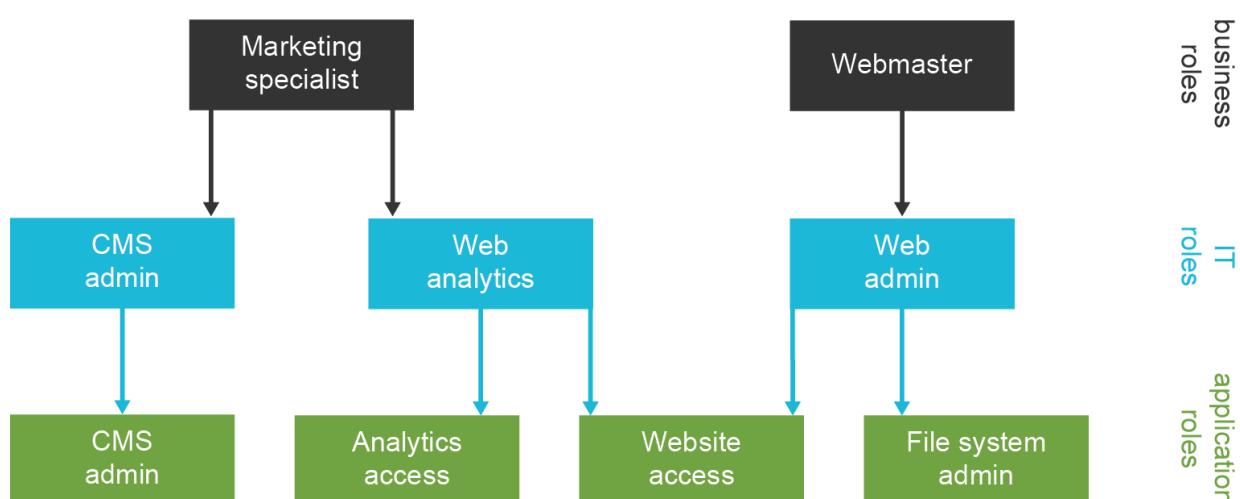
Managing permissions for every user individually is a feasible options only if the number of users is very low. Individual management of permissions becomes very difficult with populations as small as few hundreds of users. When the number of users goes over a thousand such management usually becomes an unbearable burden. The individual management of permissions is not only a huge amount of work, it is also quite an error-prone routine. This has been known for decades. Therefore many systems unified common combinations of permissions into roles and the concept of Role-Based Access Control (RBAC) was born. The roles often represent work positions or responsibilities that are much closer to the “business” than technical permissions. A role may reflect the concepts of bank teller, website administrator or sales manager. User has a role, the role contains permissions, permissions are used for authorization - that is the basic principle of RBAC. The low-level permissions are hidden from the users. Users are quite happy when they deal with the business-friendly role names.

Terminology.

The term *RBAC* is frequently used in the industry, however the actual meaning of RBAC is not always clear. The confusion is perhaps caused by the fact that there is a formal RBAC specification known as *NIST RBAC model*. When people say RBAC some of them mean that specific formal model, others mean anything that is similar to that formal model and yet others mean anything that deals with roles. We use the term RBAC in quite a broad sense. Major identity management systems usually implement a mechanism that is inspired by the formal NIST RBAC model, but the mechanism deviates form the formal model as necessary. That is what we mean when we use the term RBAC.



Most RBAC systems allow for roles to be placed inside other roles thus creating role hierarchy. Top of the hierarchy is usually composed of business roles such as “marketing specialist”. Business roles contain a lower-level roles. These are often application roles such as “website analytics” or “CMS administrator”. These lower-level roles may contain concrete permissions. Or they may contain other roles that are even closer to the underlying technology. And so on, and so on, there are proverbial turtles all the way down. Role hierarchy is often a must when the number of permissions and users gets higher.



No IDM system can be really complete without RBAC mechanism in place. Therefore, the vast majority of IDM systems support roles in one way or another. However, the quality of RBAC support significantly varies. Some IDM systems only support the bare minimum that is required to claim RBAC support. Other systems have excellent and very advanced dynamic and parametric hybrid RBAC systems. Most IDM systems are somewhere in between.

Role-based mechanism is a very useful management tool. In fact the efficiency of role-based mechanism often leads to its overuse. This is a real danger especially in bigger and somehow complex environments. The people that design roles in such environment have a strong motivation to maintain order by dividing the roles to the smallest reusable pieces and then re-combining them in a form of application and business roles. This is further amplified by the security best practices such as the *principle of least privilege*. This is completely understandable and perfectly valid motivation. However, it requires extreme care to keep such RBAC structure maintainable. Even though this may seem counter-intuitive, it is quite common that the number of roles exceeds the number of users in the system. Unfortunately, this approach turns the complex problem of user management to even more complex problem of role management. This phenomenon is known as *role explosion*.

Role explosion is a real danger and it is definitely not something that can be avoided easily. The approach that prevailed in the first-generation IDM deployments was to simply live with the consequences of role explosion. Some IDM deployments even created tools that were able to automatically generate and (more-or-less successfully) manage hundreds of thousands of roles. However, this is not a sustainable approach. The second-generation IDM systems bring features that may help to avoid the role explosion in the first place. Such mechanisms are usually based on the idea to make the roles *dynamic*. The roles are no longer just a static set of privileges. Dynamic roles may contain small pieces of algorithmic logic used to construct the privileges. Input to these algorithms are parameters that are specified when the role is assigned. Therefore the same role can be reused for many related purposes without a need to duplicate the roles. This can significantly limit the number of roles required to model a complex system. This is the best weapon against role explosion that we currently have.

Even though the RBAC system has some drawbacks it is necessary for almost any practical IDM solutions. There were several attempts to replace the RBAC system with a completely different approach. Such attempts have some success in the access management and related field. But those alternatives cannot easily replace RBAC in the identity management. Attribute-Based Access Control (ABAC) is one such popular example. The ABAC idea is based on replacing the roles with pure algorithmic policies. Simply speaking, ABAC policy is a set of algorithms that take user attributes as input. The policy combines that input with the data about operation and context. Output of the policy is a decision whether an operation should be allowed or denied. This approach is simple and it may work reasonably well in the access management world where the AM server knows a lot of details about the operation that just takes place. But in the IDM field we need to set up the account before the user logs in for the first time. There are no data about the operation yet. And even contextual data are very limited. That, together with other issues, makes ABAC a very poor choice for an IDM system. Therefore whether you like it or not, RBAC is the primary mechanism of any practical IDM solution. And it is here to stay.

Identity Management and Authorizations

The basic principle of authorization in the information security is quite straightforward: take the *subject* (user), *object* (the things that user is trying to access) and the *operation*. Evaluate whether the policy allows that subject-object-operation triple. If policy does not allow it then deny the operation. This is quite simple. But in the identity management field we need to think quite differently. We need to work backwards. The IDM system needs to set up an account for a user before the user initiates any operation. And when user really starts an operation then the IDM system will not know anything about it. Therefore the concept of authorization in the IDM world is somehow turned completely upside down.

The IDM system does not take direct part in authorization. IDM system sets up accounts in applications and databases. But the IDM system itself is not active when user logs into an application and executes the operations. Does that mean IDM system cannot do anything about authorizations? Definitely not. The IDM system does not enforce authorization decisions. But the IDM can manage the data that determine how the authorization is evaluated. IDM system can place the account to the correct groups, which will cause certain operations to be allowed and other operations denied. IDM system can set up an access control lists (ACLs) for each account that it manages. IDM system is not evaluating or enforcing the authorizations directly. But it indirectly manages the data that are used to evaluate authorizations. And this is an extremely important feature.

Authentication and authorizations are two very prominent concepts of information security. And they are vitally important for any identity and access management solution. However, authentication is quite simple in principle. Yes, the user may have several credential types used in adaptive multi-factor authentication. But while that description sounds a bit scary it is still not that complex. There are just a couple of policy statements that govern authentication. Also, authentication is typically quite uniform: most users are authenticating using the same mechanism. Authentication is not that difficult to centralize (although it may be expensive). Authentication is therefore relatively easy to manage.

But it is quite a different story for authorization. Every application has slightly different authorization mechanism. And these mechanisms are not easy to unify. One of the major obstacles is that every application works with different objects, the objects may have complex relations with other objects and all of them may also have complex relations with the subjects. The operations are also far from being straightforward as they may be parametrized. And then there is context. There may be per-operation limits, daily limits, operations allowed only during certain times or when system is in certain state. And so on. This is very difficult to centralize. Also, almost every user has slightly different combination of authorizations. Which means that there is a great variability and a lot of policies to manage. And then there are two crucial aspects that add whole new dimension of complexity: performance and scalability. Authorization decisions are evaluated all the time. It is not rare to see an authorization evaluated several times for each request. Authorization processing needs to be fast. Really fast. Even a round-trip across a local network may be a performance killer. Due to complexity and performance reasons the authorization mechanisms are often tightly integrated into the fabric of each individual application. E.g. it is a common practice that authorization policies are translated to SQL and they are used as an additional clauses in application-level SQL queries. This technique is taking advantage of the database engine to quickly filter out the data that the user is not authorized to access. This method is very efficient and it is

perhaps the only practical option when dealing with large-scale data sets. However this approach is tightly bound to the application data model and it is usually almost impossible to externalize.

Therefore it is not realistic to expect that the authorization could be centralized anytime soon. The authorization policies need to be distributed into the applications. But managing partial and distributed policies is not an easy task. Someone has to make sure that the application policies are consistent with the overall security policy of the organization. Fortunately, the IDM systems are designed especially to handle management and synchronization of data in broad range of systems. Therefore the IDM system is the obvious choice when it comes to management of authorization policies.

Organizational Structure, Roles, Services and Other Wildlife

Back in the 2000s the IDM was all about managing user accounts. It was enough to create, disable and delete an account to have a successful IDM deployment. But the world is a different place now. Managing the accounts is simply not enough anymore. Yes, automated account management brings significant benefits and it is a necessary condition to get at least a minimal level of security in complex systems. But account management is often not enough to justify the cost of an IDM system. Therefore current IDM systems can do much more than just a simple account management.

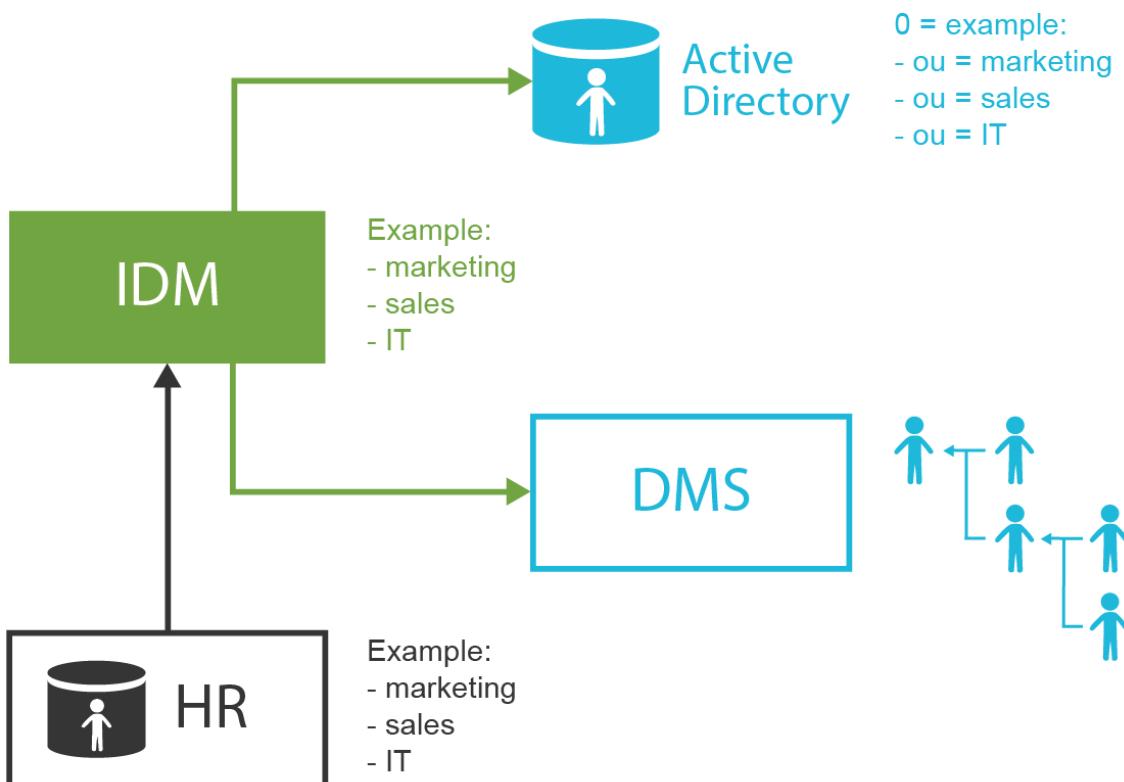
There are many things that an advanced IDM system can manage:

- Accounts. Obviously. Many IDM systems can fully manage account attributes, groups membership, privileges, account status (enabled/disabled), validity dates and all the other details.
- Groups and roles. Apart from managing the membership of accounts in groups the IDM system can take care of the whole group life-cycle: create a group, manage it and delete it.
- Organizational structure. The IDM system can take organizational structure from its authoritative source (usually HR) and synchronize it to all the applications that need it. Or the IDM itself may be used to manually maintain an organizational structure.
- Servers, services, devices and "things". While this is not yet IDM mainstream, there are some experimental solutions that use IDM principles to manage concepts that are slightly outside the traditional IDM scope. E.g. there is an IDM-based solution that can automatically deploy predefined set of virtual machines for each new project. The new IDM systems are so flexible that they can theoretically manage everything that is at least marginally related to the concept of identity: virtual machines, networks, applications, configurations, devices ... almost anything. This is still quite a unique functionality. But it is very likely that we will see more stories about this in the future.

While all these features are interesting, some of them clearly stand out. The management of groups and organizational structure are those that are absolutely critical for almost any new IDM deployment. Your organizational structure may be almost flat and project-oriented or you may have twelve levels of divisions and sections. But regardless of the size and shape of your organizational structure it needs to be managed and synchronized across applications in pretty much the same way as identities are synchronized. You may need to create groups in Active Directory for each of your organizational unit. You want them to be correctly nested. You may want to create distribution list for each of your ad-hoc team. And you want this operation to have as little overhead as possible otherwise the teams cannot really be managed in ad-hoc fashion. You may

want to synchronize the information about projects into your issue tracking system. You may also want to automatically create a separate wiki space and a new source code repository for each new development project. The possibilities are almost endless. Both the traditional organizations and the new lean and agile companies will benefit from that.

Organizational structure management is closely related to group management. The groups are often bound to workgroups, projects or organizational units. E.g. an IDM system can automatically maintain several groups for each project (admin and member groups). Those groups can be used for authorization. Similarly, an IDM system can automatically maintain application-level roles, access control lists (ACLs) and other data structures that are usually used for authorization.



While this functionality provides benefits in almost any deployment, organizational structure management is absolutely crucial for organizations that are based on tree-like functional organizational structures. These organizations heavily rely on the information derived from organizational structure. E.g. direct manager of the document author can review and approve the document in the document management system. Only the employees in the same division can see the document draft. Only the employees of a marketing section can see marketing plans. And so on. Traditionally, such data are encoded into an incomprehensible set of authorization groups and lists. And that contributes to the fact that reorganizations are a total nightmare for IT administrators. However, an IDM system can significantly improve the situation. IDM can create the groups automatically. It can make sure that the right users are assigned into these groups. It can synchronize information about the managers into all affected applications. And so on. And a good IDM system can do all of that using just a handful of configuration objects.

This seems to be almost too good to be true. And it is fair to admit that the quality of organizational management features significantly varies among IDM systems. Group management and organizational structure management seem to be a very problematic feature. Only few IDM systems support these concepts at the level that allows practical out-of-box deployment. Most IDM systems have some support for that, but any practical solution requires heavy customization. It is not clear why IDM vendors do not pay attention to features that are required for almost any IDM deployment. Therefore when it comes to a comprehensive IDM solution there is one crucial advice that we could give: choose the IDM product wisely.

Everybody Needs Identity Management

Such a title may look like a huge exaggeration. But in fact it is very close to the truth. Every non-trivial system has a need for identity management, even though the system owners may not realize that. As you are reading this book, chances are that you are one of the few people that can see the need. In that case it is all mostly about costs/benefits calculation. Identity management has some inherent complexity. While even very small systems need IDM, the benefits are likely to be too small to justify the costs. The cost/benefit ratio is much better for mid-size. And IDM is an absolute necessity for large-scale systems. There seems be a rule of thumb that has quite broad applicability:

Number of users	Recommendation
Less than 200	You may need IDM, but the benefits are probably too small to justify the costs.
200 – 2 000	You need IDM and the benefits may be just enough to justify the costs. But you still need to look for a very cost-efficient solution.
2 000 – 20 000	You really need IDM. You simply cannot manage that crowd manually. If you implement IDM properly the benefits will be much higher than the costs.
More than 20 000	I can't believe that you do not have any IDM yet. Go and get one. Right now. You can thank me later.

Identity Governance and Compliance

Identity governance is basically an identity management taken to a higher business level. The identity management proper is focused mainly on technical aspects of identity life-cycle such as automatic provisioning, synchronization, evaluation of the roles and computing attributes. On the other hand, identity governance abstracts from the technical details and it is focused on policies, roles, business rules, processes and data analysis. E.g. a governance system may deal with segregation of duties policy. It may drive the process of access re-certification. It may focus on automatic analysis and reporting of the identity, auditing and policy data. It will drive remediation processes to address policy violations. It will manage application of new and changed policies, evaluate how is your system compliant with policies and regulations and so on. This field is sometimes referred to as *governance, risk management and compliance* (GRC).

Almost all IDM systems will need at least some governance features to be of any use in practical deployments. And many governance features are just refinement of concepts that originated in the IDM field many years ago. Therefore the boundary between identity management and identity governance is quite fuzzy. The boundary is so fuzzy that new terms were invented for the unified field that includes the identity management proper together with identity governance. *Identity governance and administration* (IGA) is one of these terms. This field (or sub-field) is still quite young, therefore it is expected that the terminology and even the concepts need some time to settle down. For us the governance is just a natural continuation of identity management evolution.

However, it seems to be a common practice that identity governance features are implemented by specialized products that are separated from their underlying IDM platforms. Almost all commercial IDM and governance solutions are divided into (at least) two products. This strategy obviously brings new revenue streams for the vendors. But it makes almost no sense at all from customer point of view. The industry has even coined a term *closed-loop remediation* (CLR) which in fact means that the governance system is somehow integrated with the underlying IDM solution. Industry sometimes has a need for inventing fancy marketing terms for something that should be natural part of any reasonable solution. It perhaps comes without saying that reasonable IDM solutions should offer both the IDM and governance features in one unified and well aligned product.

Below is a list of features that belong to the governance/compliance category. As the boundary of governance is so fuzzy, there are also features that are just related to governance.

- **Delegated administration.** Basic IDM deployments are usually based on the idea of an omnipotent system administrator that can do almost anything. And then there are end users that can do almost nothing. While this concept may work in small and simple deployments, it is not sufficient for larger systems. Large organizations usually need to delegate some administration privileges to other users. There may be HR personnel, people that are responsible for management of their organizational units, administrator responsible for a particular group of systems and so on.
- **Deputies.** Delegated administration is very useful, but it is quite static. It is given by policies that are not entirely easy to change. But there is often a need for ad-hoc delegation, such as a temporary delegation of privileges during manager's vacation. Such a manager could nominate a deputy that should receive parts of manager's privileges. This is all done on an ad-hoc basis, initiated by an explicit action of the manager.
- **RBAC-related policies**, such as Segregation of Duties (SoD) policy. Simply speaking SoD policy ensures that conflicting duties cannot be accumulated with a single person. This is usually implemented by using a role exclusion mechanisms. However, it may go deeper. E.g. it may be required that each request is approved by at least two people.
- **Policies related to organizational structure.** Organizational structure may look like a simple harmless tree, but in reality it is far from being simple or harmless. In theory the organizational structure should be managed by business or operations departments such as HR. But the reality is often quite different. Business departments lack the tools and processes to efficiently manage organizational structure. Therefore it is often an IDM system that assumes the responsibility for organizational structure management. In such cases there is a need to police the organizational structure. For example there may be policies that mandate a single manager for each department. In that case the IDM system may need to handle situations that there is no manager

or too many managers.

- **Dynamic approval schemes.** Approval processes are usually considered to be part of basic identity management functionality. Those are usually implemented by some kind of general-purpose workflow engine. However, this is often a source of maintenance problems, especially in deployments that are focused on identity governance functionality. In such cases the approval processes are no longer simple quasi-linear workflows. Approval processes tend to be very dynamic and their nature is determined by the policies. Workflow engines have a very hard time coping with such a dynamic situation. IDM system that implement special-purpose policy-based approval engines provide much better solutions.
- **Entitlement management** is mostly an identity management thing. It deals with entitlements of user's accounts in target systems such as role or group membership. However, this process can go both ways. Governance systems may provide a "entitlement discovery" features that take entitlements as inputs. This can be used evaluate compliance and policy violations, but it may also be a valuable input for role engineering.
- **Role mining.** IDM systems are seldom deployed on a green field. In the common case there are existing systems in place, there are application roles, entitlements and privileges. It is not an easy job to create IDM roles that map to this environment. This is usually a slow and tedious process. However, IDM system can retrieve all the existing information and use it to propose role structure. This is not a fully deterministic process, it requires a lot of user interaction, tuning and it is often based on a machine learning capabilities. It is not a replacement for role engineering expertise. However, machine-assisted role mining can significantly speed up the process.
- **Re-certification campaigns.** Assignment of roles is often an easy task. Request a role, role goes through an approval process and the role is assigned. And then everybody forgets about it. There is a significant incentive to request assignment of a new role. But there is almost no incentive to request unassignment of a role that is no longer needed. This leads to an accumulation of privileges over time. And such accumulation may reach dangerous levels for employees with long and rich job transfer history. Therefore there are re-certification campaigns that are also known as "certification", "access certification" or "attestation" mechanisms. The goal of those campaign is to confirm ("certify" or "testify") that the user still needs the privileges that were assigned previously. Re-certification campaigns are designed to be conducted on a large number of users in a very efficient manner. Therefore there are special processes and a very specific user interface is provided to conduct such campaigns.
- **Role governance** is usually quite a complex matter. Typical IDM deployments will have a large number of roles. It is quite hard to define those roles in the first place. But then it is even harder to maintain them. Environment is changing all the time, therefore the roles have to change as well. It is usually beyond the powers of a single administrator to do so. Therefore many role owners are usually nominated to take care of role maintenance. Roles are often grouped into applications, categories, catalogs or functional areas. The IDM system must make sure that the owners have the right privileges to do their job. The IDM system should also take care that each role has at least one owner, that role definitions are periodically reviewed and so on.
- **Role lifecycle management** is a dynamic part of role governance. Role changes are likely to have a serious impact on overall security of the system. Therefore it may not be desirable to simply delegate role management duties. It may be much more sensible to require that role changes has to be approved before being applied. New roles are also created all the time and

old roles are decommissioned. The IDM system may need to make sure that a decommissioned role is not assigned to any new user. But such role may still be needed in the system during a phase-out period.

- **Role modeling.** A change of a single role often does not make much sense just by itself. The roles are usually designed in such a way that a set of roles works together and forms a role model. Therefore approval of each individual role change may be too annoying and even harmful. E.g. there may be an inconsistent situation in case that one change is approved and another is rejected. Therefore roles and policies are grouped into models, the models are reviewed, versioned and applied in their entirety.
- **Simulation.** IDM deployments tend to be complex. There are many relations, interactions and policies. It is no easy task to predict the effects of a change in a role, policy or organizational structure. Therefore some IDM systems provide a simulation features that provide predictions and impact analyses of planned changes.
- **Compliance policies,** reporting and management. Policies in the identity management world are usually designed to be strictly enforced. This works fine for fundamental policies that are part of simple IDM deployments. However, the big problem is how to apply new policies - especially policies that are mandated by regulations, recommendations and best practices. In such cases it is almost certain that significant part of your organization will not be compliant with such new policy. Applying the policy and immediately enforcing it is likely to cause a major business disruption. However, it is almost impossible to prepare for new policies and to mitigate their impact without knowing which users and roles are affected. Therefore the policies are usually applied, but they are not enforced yet. The policies are used to evaluate the compliance impact. Compliance reports can be used to find the users that are affected by the policy in order to remedy the situation. Compliance reports may also be used to track the extent and progress of compliance.
- **Remediation.** Good IDM deployments strive for automation. All the processes and actions that can be automated are automated. E.g. if a role is unassigned and user does no longer needs an account, such account is automatically deleted or disabled. However, there are actions that cannot be automated because they require decision of a living and thinking human being. Approvals are one example of such processes. However, there are more situations like that. And many of those require more initiative than a simple yes/no decision. One such example is organizational structure management. There is usually a rule that each department must have a manager. But what should IDM system do in case that a department manager is fired? IDM system cannot stop that operation, as there are certainly good reasons to revoke all privileges of that manager. Therefore the result is that there is now a department without a manager. And the IDM system itself cannot do anything about that. That is where remediation comes to the rescue. Remediation process is started after the operation. The remediation process will ask a responsible person to nominate a new manager for the department. There may be a broad variety of remediation processes. Simple process will ask for yes/no decisions or may ask to nominate users. But there are often generic processes that apply to completely unexpected situations.
- **Risk management automation.** Information security is not a project, it is a process. It starts with risk analysis, planning, actions and then it goes back to analysis and planning and actions and so on and so on for ever and ever. Risk analysis is the part that takes a huge amount of time and effort - especially when it comes to analysis of insider threat as there is usually a lot of insiders to analyze. However, an IDM system can help to reduce the risk analysis effort. Each

role assigned to a user is a risk. If roles are marked with relative risk levels, IDM system can compute the accumulation of risk for each user. As each role gives access to a particular set of assets, the IDM system may provide data to evaluate asset exposure to users.

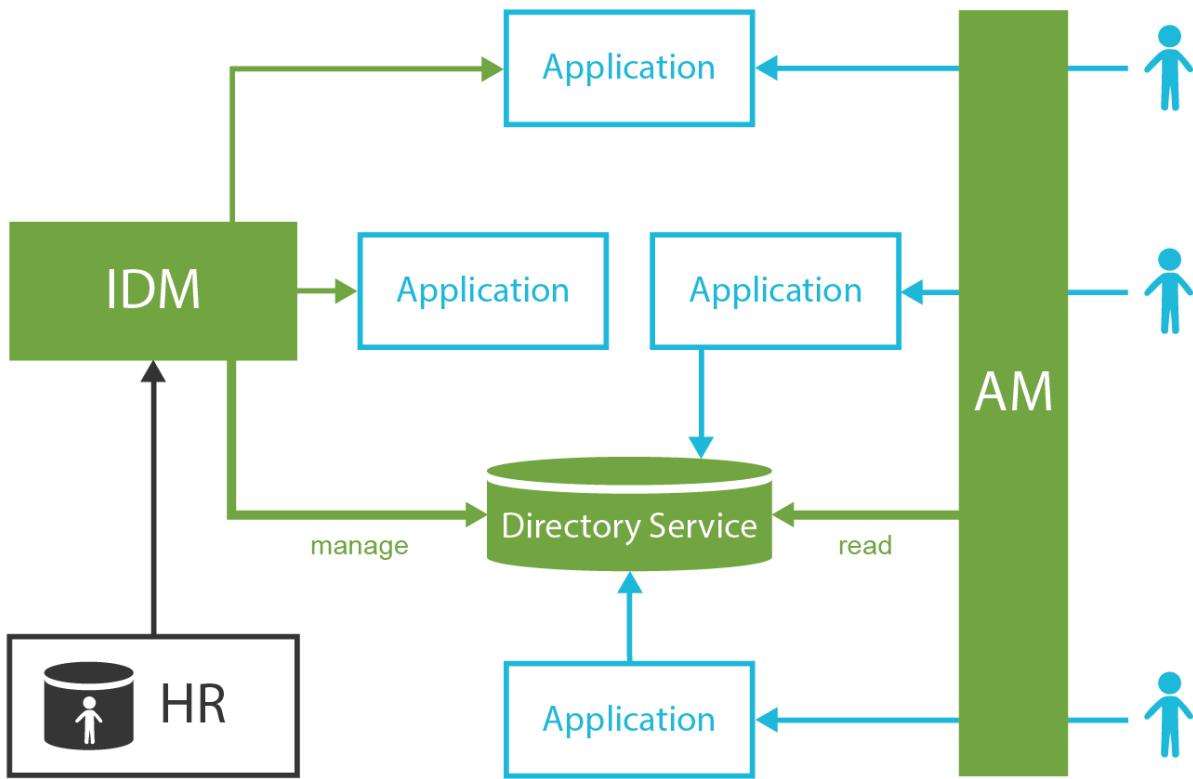
Complete Identity and Access Management Solution

A comprehensive Identity and Access Management solution cannot be built by using just a single component. There is no single product or solution that will provide all the necessary features. And as the requirements are so complex and often even contradictory it is very unlikely that there ever will be any single product that can do it all.

A clever combination of several components is needed to build complete solution. The right mix of ingredients for this IAM soup will always be slightly different as no two IAM solutions are the same. But there are three basic components that are required for any practical IAM deployment:

- **Directory service** or a similar identity store is the first component. This is the database that stores user account information. The accounts are stored there in a “clean” form that can be used by other applications. This database is then widely shared by applications that are capable to connect to it. This part of the solution is usually implemented as a replicated LDAP server topology or Active Directory domain. This has an advantage of relatively low cost and high availability. But there is one major limitation: the data model needs to be simple. Very simple. And the identity store needs to be properly managed.
- **Access Management** is a second major component of the solution. It takes care of authentication and (partially) authorization. Access management unifies authentication mechanisms. If an authentication mechanism is implemented in the access management server then all integrated applications can easily benefit. It also provides Single Sign-On (SSO), centralizes access logs and so on. It is a very useful component. But of course, there are limitations. AM system needs access to identity data. Therefore it needs reliable, very scalable and absolutely consistent identity database as a back-end. This is usually provided by the directory service. Performance and availability are the obvious obstacles here. But there is one more obstacle which is less obvious but every bit as important: data quality. The data in the directory service must be up to date and properly managed. But that is only part of the picture. As most applications store some pieces of identity data locally, these data also need to be synchronized with the directory database. No access management system can do this well enough. And there is no point for AM to do it at all. The AM system has a very different architectural responsibilities. Therefore yet another component is needed.
- **Identity Management** is the last but in many ways the most important component. This is the real brain of the whole solution. The IDM system maintains the data. It is the component that keeps the entire system from falling apart. It makes sure the data are up to date and compliant with the policies. It synchronizes all the pieces of identity data that those pesky little applications always keep creating. It maintains groups, privileges, roles, organizational structures and all the other things necessary for the directory and the access management to work properly. It maintains order in the system. And it allows living and breathing system administrators and security officers to live happily, to breath easily and to keep control over the whole solution.

The following diagram shows how all these components fit together.



This is truly a composite solution. There are several components that have vastly different features and characteristics. But when bound together into one solution, the result is something that is much more than just a sum of its part. The components support each other. The solution cannot be complete unless all three components are in place.

However, building a complete solution may be quite expensive and it may take a long time. You have to start somewhere. But if you have resources for just one product then choose identity management. IDM is a good start. It is not that expensive as access management. And IDM brings good value even quite early in the IAM program. Especially the second generation IDM systems are very good at repaying the investment. Going for open source product will also keep the initial investment down. Starting with IDM is usually the best choice to start the IAM program.

IAM and Security

Strictly speaking, Identity and Access Management (IAM) does not entirely fit into the information security field. The IAM goes far beyond information security. IAM can bring user comfort, reduce operation costs, speed up processes and generally improve the efficiency of the organization. This is not what information security is concerned with. But even though IAM is not strictly part of information security there is still a huge overlap. IAM deals with authentication, authorization, auditing, role management and governance of objects that are directly related to the information security. Therefore IAM and information security have an intimate and very complicated relationship.

It is perhaps not too bold to say that the IAM is a pre-requisite to good information security. Especially the identity management (IDM) part is absolutely critical - even though this may not be that obvious at the first sight. But the evidence speaks clearly. Security studies quite consistently

rate the insider threat as one of the most severe threats for an organization. However, there is not much that the technical countermeasures can do about the insider threat. The employee, contractor, partner, serviceman - they all are getting the access rights to your systems easily and legally. They will legally pass through even the strongest encryption and authentication because they have got the keys. Firewalls and VPNs will not stop them because those people are meant to pass through them to do their jobs.

Vulnerabilities are there, obviously. And with the population of thousands of users there is a good chance that there is also an attacker. Maybe one particular engineer was fired yesterday. But he still has VPN access and administration rights to the servers. And as he might not be entirely happy about the way how he has been treated the chances are he might be quite inclined to make your life a bit harder. Maybe leaking some company records would do the trick. Now we have a motivated attacker who will not be stopped by any countermeasures and who can easily access the assets. Any security officer can predict the result without a need for a comprehensive risk analysis.

Information security has no clear answers to the insider threat. And this is no easy issue to solve as there is obviously a major security trade-off. The business wants users to access the assets easily to do their jobs. To keep the wheels of an organization turning. But security needs to protect the assets from the very same users. And there is no silver bullet to solve this issue. However there is a couple of things that can be done to improve the situation:

- **Record who has access to what.** Each user has accounts in many applications through the enterprise. Keep track which account belongs to which user. It is very difficult to do that manually. But even the worst IDM system can do that.
- **Remove access quickly.** If there is a security incident then the access rights need to be removed in order of seconds. If an employee is fired then the accounts have to be disabled in order of minutes. It is not a problem for a system administrator to do that manually. But will the administrator be available during a security incident late in the night? Would you synchronize layoffs with the work time of system administrators? Wouldn't system administrators forget to stop all the processes and background jobs that the user might have left behind? IDM system can do that easily. Security staff can simply disable all the accounts by using IDM system. Single click is all that is needed.
- **Enforce policies.** Keep track about the privileges that were assigned to users. This usually means managing assignment of roles (and other entitlements) to users. Make sure that the assignment of sensitive roles is approved before user gets the privileges. Compare the policies and the reality. System administrators that create accounts and assign entitlements are not robots. Mistakes can happen. Make sure the mistakes are discovered and remediated. This is the natural best practice. But it is almost impossible to do manually. Yet even an average IDM system can do that without any problems.
- **Remove unnecessary roles.** Role assignments and entitlements tend to accumulate over time. Long-time employees often have access to almost any asset simply because they needed the data at some point in their career. And the access to the asset was never removed since. This is a huge security risk. It can be mitigated by inventing a paper-based process to review the entitlements. But that process is very slow, costly, error-prone and it has to be repeated in regular intervals. But advanced IDM systems already support automation of this re-certification process.
- **Maintain order.** If you closely follow the principle of least privilege then you have probably

realized that you have more roles than you have users. Roles are abstract concepts and they are constantly evolving. Even experienced security professionals can easily get lost in the role hierarchies and structures. The ordinary end users often have absolutely no idea what roles they need. Yet, it is not that hard to sort the roles to categories if you maintain them in a good IDM system. This creates a role catalog that is much easier to understand, use and maintain.

- **Keep track.** Keep an audit record about any privilege change. This means keeping track of all new accounts, account modifications, deletions, user and account renames, role assignments and unassignments, approvals, role definition changes, policy changes and so on. This is a huge task to do manually. And it is almost impossible to avoid mistakes. But a machine can do that easily and reliably.
- **Scan for vulnerabilities.** Mistakes happen. System administrators often create testing accounts for troubleshooting purposes. And there is an old tradition to set trivial passwords to such accounts. These accounts are not always cleaned up after the troubleshooting is done. And there may be worse mistakes. System administrators may assign privileges to a wrong user. Help desk may enable account that should be permanently disabled. Therefore, all the applications have to be permanently scanned for accounts that should not be there and for entitlements that should not be assigned. This is simply too much work to be done manually. It is not really feasible unless a machine can scan all the system automatically. This is called reconciliation, and it is one of the basic functionalities of any decent IDM system.

Theoretically all of these things can be done manually. But it is not feasible in practice. The reality is that information security seriously suffers - unless there is an IDM system that brings automation and visibility. Good information security without an IDM system is hardly possible.

Building Identity and Access Management Solution

There is no single identity and access management solution that would suit everybody. Every deployment has specific needs and characteristics. Deployment in a big bank will probably focus on governance, role management and security. Deployment in small enterprise will focus on cost efficiency. Cloud provider will focus on scalability, user experience and comfort. Simply speaking one size does not fit all. Almost all IAM solutions use the same principal components. But product choice and configuration will significantly vary. Do not expect that you download a product, install it and that it will solve all your problems. It won't. Customization is the key.

We consider identity management to be heart and brain of any IAM solution. This is one of the reasons why we have started midPoint project. The rest of this book will focus almost exclusively on identity management and the use of midPoint as the IDM component. This is the place where theory ends and practice begins.

Chapter 2. MidPoint Overview

Chaos was the law of nature; Order was the dream of man.

— Henry Adams

MidPoint is an open source identity management (IDM) and identity governance system. It is a very rich and sophisticated system that provides many advanced features. MidPoint is maintained by Evolveum – a company dedicated to open source development. All midPoint core developers work for Evolveum. However, there are also partners and other engineers that are contributing to midPoint development.

MidPoint is a second-generation IDM system. There are few veterans in midPoint development team that deployed first-generation IDM systems since early 2000s. That was not always a pleasant experience. Therefore in 2011 we started midPoint project to correct the mistakes of early IDM systems. One of the main differences between midPoint and other IDM systems is that midPoint is designed and implemented with one primary goal in mind: to be practical. We had been dealing (and struggling) with first-generation IDM systems in the past and we do not want to live through that experience again. Therefore practicality goes very deep into the very foundations of midPoint. To be more concrete, practicality means:

- Things that are simple or used often should be easy to configure. Propagation of changed password, user enable/disable, account synchronization – these should be as easy as possible. As simple as flicking a switch or setting few configuration properties.
- Things that are more complex or used less frequently may be a bit harder. Such as editing XML or JSON file or writing few lines of Groovy or Python script.
- Things that are very complex or very unusual should be still possible. However these might not be easy. It may require longer scripts or implementing some Java classes. It may require forking and modifying the source code. But it must be possible to do almost anything.

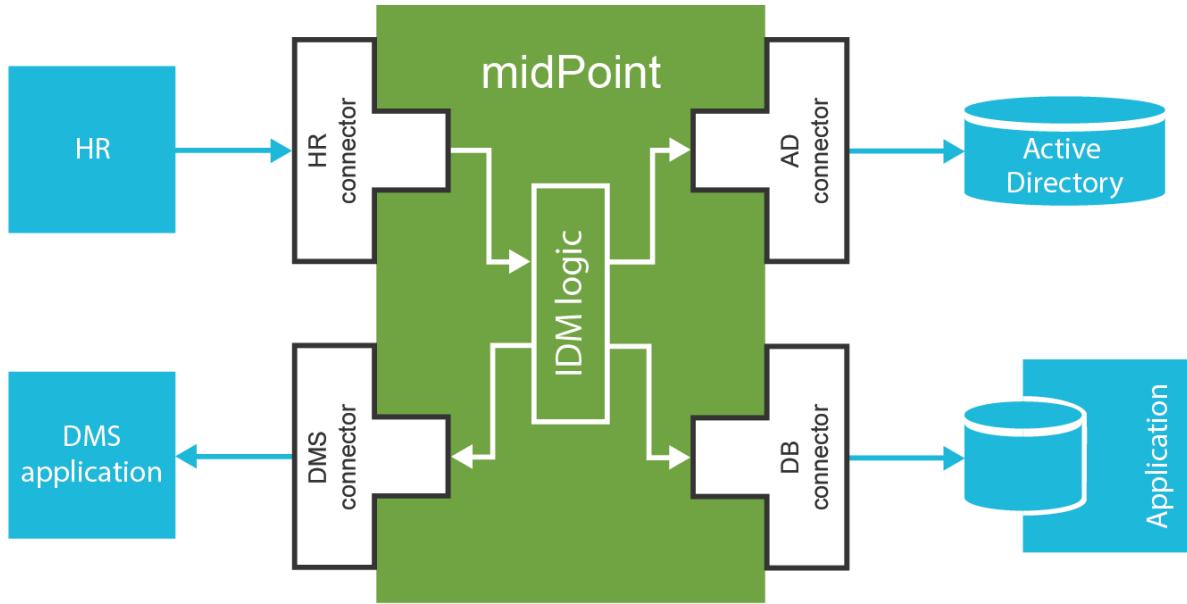
This means that simple solutions which do not deviate from the usual requirements will be easy to implement. Most IAM programs start like this. This approach allows to gain the benefits very early in the project. The effort grows as the requirements are getting more complex and more unusual. But the effort is still much lower than implementing everything from scratch. And there is always an option to stop the project at any point where the costs are getting too high to justify the benefits. MidPoint is an open source system. Therefore there is no license cost that would offset the initial costs. Even small projects are feasible with midPoint.

Simply speaking, midPoint is following the Pareto principle: 20% effort brings 80% benefits. There are many mechanisms that support this approach. Some are based in midPoint design, some originate from midPoint development practices and some are even supported by the Evolveum business model. But more about that later.

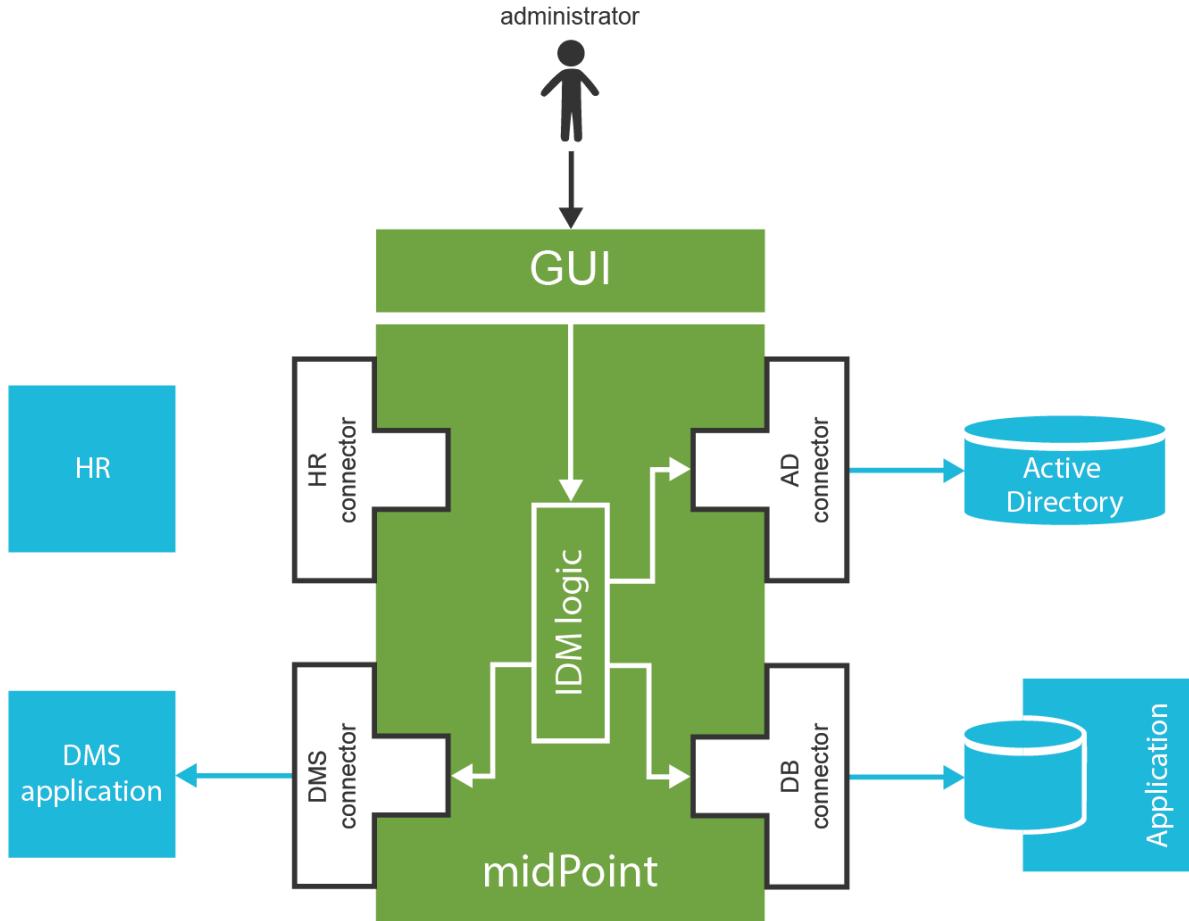
How MidPoint Works

MidPoint does what any identity management system is supposed to do: it manages identities. The very basic functionality of midPoint is the synchronization of identity data that are stored in

various applications, databases, directory servers, text files and so on. We call all these systems *resources*. MidPoint is using *connectors* to reach the resources. MidPoint can propagate change that happened in one resource to other resources. E.g. an employee record appears in the HR system, it is picked up by midPoint, processed, transformed and then new Active Directory and CRM accounts are created. This is the process that we call *synchronization*.



MidPoint has a rich graphical user interface (GUI) that can be used to manage the identities. Changes made by system administrators are automatically propagated to all affected resources. E.g. security officer disables a user by clicking on disable button in midPoint user interface. Then MidPoint makes sure that all accounts that belong to the user are immediately disabled.



This is the essence of midPoint operation. This may sound simple, but this description is extremely simplified. The reality is much more complicated. Most of the important things happen inside midPoint before the changes are applied to target resources. This may look simple at the first sight, but it is not. For each change that midPoint detects it needs to evaluate:

- **Roles:** MidPoint computes where the user should have access. This is usually given by the roles that the user has. The role structure is often quite rich. There may be hierarchical roles, parametric roles, conditional roles and a lot of other advanced mechanisms.
- **Organizational structure:** Users usually belong to some organizational units, projects, teams or groups. Some of them may give additional privileges to the user.
- **Status and life cycle:** Accounts can be created, enabled, disabled, archived or deleted. There are many situations that need to be processed. E.g. we may want to create a disabled account one week before a new employee starts his work, enable the account on his first day, disable the account on his last day and delete it three months after he leaves.
- **Attributes and identifiers:** Simple synchronization scenarios assume that attributes and values will be the same in all the synchronized systems. That is a nice theory, but it almost never works like that in real world. Attribute names need to be translated, values need to be transformed, data types need to be converted. This is different for each system and even for each instance of each system. Small algorithms in form of scripting expressions are usually needed to properly transform the values.
- **Credential management:** Password changes need to be propagated to the resources.

Sometimes we want to synchronize password with all systems, sometimes we want just a subset of systems. Password policies need to be evaluated, password history needs to be checked, password may need to be encoded and hashed before storage.

- **Consistency:** The account in the target application might have changed since midPoint has updated it. The current change may no longer be applicable to the current state of the account. The change that midPoint wants to make may conflict with the native change, the change may be partially applied already, the account may have attribute values that it should not have or the account may not exist at all. MidPoint has to detect such situations and react accordingly, e.g. by re-creating a deleted account before applying the changes.
- **Approvals:** MidPoint determines if any of the changes need to be approved before they are applied. If that is the case then midPoint drives the request through an approval process.
- **Notifications:** MidPoint notifies the user that he can access a new account. It notifies the administrator if something goes wrong.
- **Audit:** MidPoint records all the changes into an audit trail. This can later be used by security officers or specialized analytic engines.

This is a lot of things to process, evaluate and execute. Some of these steps are quite complex. And indeed there are many complex algorithms implemented in midPoint. There are algorithms that evaluate complex role structures, organizational structures, temporal constraints, password policies and so on. The only thing that is needed is to configure them properly.

However, midPoint does even more than that. MidPoint does not only manage identities, it can also manage any object that is anyhow related to identity management. MidPoint can manage roles, role catalogs, organizational structures, groups, projects, teams, services, devices and almost any other object.

MidPoint is also an *identity governance system*. The job of identity management features is to make sure that the policies are consistently applied through the organization. The governance features assist with the maintenance and evolution of those policies. MidPoint implements access recertification process. This is a recurring process that asks managers to confirm that the users still need the privileges that they have previously received. MidPoint contains mechanism to sort roles into hierarchies and categories. That is necessary to maintain order during role engineering and maintenance of role definitions. MidPoint has mechanisms for selective enforcement of role which comes very useful during migrations and when new system is connected to midPoint. MidPoint has support for policy lifecycle, general policy rules and so on. And more work in that direction is planned in future midPoint versions. We fully understand that it is not enough to simply apply the policies. Policies are living things and they need to evolve.

Case Study

This book is about practical identity management. Therefore we will get very close to a practice by demonstrating midPoint features using a case study. This is a case study of a fictional company ExAmPLE, Inc. The name stands for "Exemplary Amplified Placeholder Enterprise". ExAmPLE is a mid-sized financial company. Its operation heavily relies on information technologies, therefore there is a diverse set of applications and information systems ranging from legacy applications to cloud services. As ExAmPLE has few thousand employees and there is a good potential for growth the management has decided to start an IAM program. The first step of the program is deployment

of midPoint as the identity management system.

Eric is an IT engineer at ExAmPLE. He has taken the responsibility to install and configure midPoint. Eric spins up a new Linux virtual machine for midPoint. He downloads midPoint distribution package and follows the installation instructions. Couple of minutes later midPoint instance starts up. Eric logs in to the midPoint user interface.

The screenshot shows the midPoint Info dashboard. On the left, there's a sidebar with 'SELF SERVICE' and 'ADMINISTRATION' sections. Under 'SELF SERVICE', there are links for Home, Profile, Credentials, Request a role, and a collapsed section for Archetypes. Under 'ADMINISTRATION', there are links for Dashboards (Info dashboard is selected), Admin dashboard, Users, Org. structure, Roles, Services, and Archetypes. The main content area has a header 'Info dashboard'. It features six colored boxes: red for USERS (1 enabled, 1 total), orange for ORGANIZATIONAL UNITS (0 enabled, 0 total), green for ROLES (5 enabled, 5 total), blue for SERVICES (0 enabled, 0 total), purple for RESOURCES (0 up, 0 total), and dark blue for TASKS (3 active, 3 total). Below these are two sections: 'Personal info' (Last login: Date Never, From Not defined; Last unsuccessful login: Date Never, From Not defined; Other: Account expiration date Not defined) and 'System status' (CPU Usage: 0.3, Heap memory: 377.8MB / 3.3GB / 3.6GB, Non heap memory: 244.8MB / 254.9MB / -1B, Threads (live/peak/total): 59 / 62 / 72, Start time: October 25, 2019 5:09:43 PM, Uptime: 4 minutes).

MidPoint instance is almost empty after fresh installation. It contains only a couple of essential objects. But Eric is a smart engineer. He has already read through this book and he knows exactly what he needs to do.

First thing to do is to populate midPoint with employee data. The primary source of ExAmPLE employee data is an HR system. The HR system is quite big piece of software and it is not easy to connect to that system directly. Fortunately, it is quite easy to get a text export of the employee data in comma-separated (CSV) format. Eric plans to use this file to get employee data to midPoint.

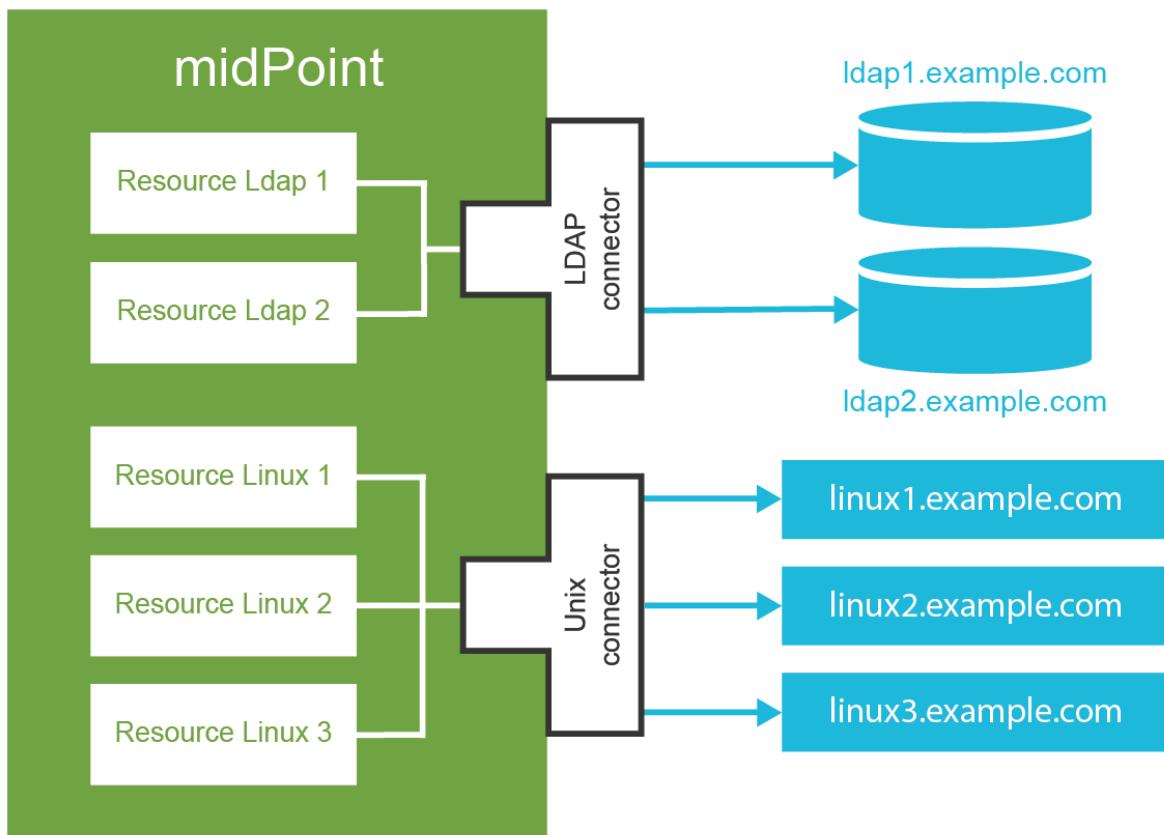
Connectors and Resources

MidPoint communicates with all the source and target systems by the means of connectors. Connectors are relatively small Java components that are plugged into midPoint. There is usually one connector for each type of the connected system. Therefore there are connectors for LDAP servers, Active Directory, databases, UNIX operating systems and so on. The responsibility of a connector is to translate protocols. E.g. LDAP connector will translate midPoint search commands to LDAP search requests. The UNIX connector will create an SSH session and translate midPoint create command to the invocation of Linux useradd binary. And so on. Each connector talks using its own communication protocol on one side. But on the other side the connectors translate the information to a common format that is understood by midPoint.

There is no distinction between source and target system when it comes to the connector. The same connectors are used for source and target systems. The difference is only in midPoint configuration.

The connectors are distributed as Java binaries (JAR files). To deploy them to midPoint you just

need to place them in the correct directory and restart midPoint. MidPoint will automatically discover and examine the connectors during start-up. A handful of frequently used connectors is bundled into midPoint distribution. These connectors do not need to be deployed. They are automatically available.



Connector of a specific type works for all the systems that communicate by the protocol supported by connector. E.g. LDAP connector works for all the LDAP-compliant servers. Connector is just a very generic piece of code. It does not know the hostname, port or passwords that are needed to establish a connection to a particular server. The configuration that specify connection parameters for individual server is stored in special configuration object called *resource*. The term *resource* in midPoint terminology generally means any system which is connected to a midPoint instance.

Therefore what Eric the Engineer needs to do to get ExAmPLE employees into midPoint is to define a new resource. This resource will represent the CSV file exported from the HR system. MidPoint distribution contains CSV file connector already, therefore there is no need to deploy it explicitly. All that Eric has to do is to create a new resource definition. There are (at least) two ways how to do it. Firstly, there is a configuration wizard in midPoint user interface. Eric can use the wizard to configure a new resource from scratch. But as you will see later in this book, the resource definition is quite complex and it has many configuration options. This makes the configuration wizard very rich and it may be quite confusing for new users. Therefore it is better for Eric to use the other approach: start from an example. There are examples of various resource definitions in the midPoint distribution package and even more examples are available on-line. Therefore Eric quickly locates a XML file that contains a complete example of a CSV resource. He edits the file to change the filesystem path to his CSV file and adjusts the names of the columns to match the format

of his file. The very minimal resource configuration specifies just the resource name, connector and connector configuration. The XML file that Eric creates looks approximately like this (simplified for clarity):

```
<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
    <name>HR System</name>
    <connectorRef type="ConnectorType"> ...</connectorRef>
    <connectorConfiguration>
        <configurationProperties>
            <filePath>/opt/midpoint/var/resources/hr.csv</filePath>
            <uniqueAttribute>empno</uniqueAttribute>
        </configurationProperties>
    </connectorConfiguration>
</resource>
```



If you are a hands-on type of an engineer you probably want to follow what Eric is doing in your own midPoint instance. All the files that Eric is using are provided in a form of ready-to-use samples. Please see [Additional Information](#) chapter at the end of this book for the details.

Then Eric goes to *Configuration* section of midPoint user interface and imports the XML file into midPoint. Import operation creates new resource definition in midPoint. Eric now navigates to *Resources* section of the midPoint user interface. The new CSV resource is there. When Eric clicks on the resource name a resource details screen appears.

The screenshot shows the midPoint user interface with the following details:

- Left Sidebar:** Shows navigation categories: SELF SERVICE (Home, Profile), ADMINISTRATION (Request a role, Dashboards, Users, Org. structure, Roles, Services, Archetypes), and Resources (All resources, View resource, New resource, Import resource definition, All connector hosts).
- Top Bar:** Shows the title "midPoint", the current location "Resources > Resources List > Resource details", and the user "administrator".
- Resource Details Page:**
 - Resource Name:** HR System
 - Status:** RESOURCE IS UP, CsvConnector 2.3
 - Mappings:** No mappings
 - Schema:** 1 object types, 1 schema definitions
 - Capabilities:** A grid of 16 icons representing various resource operations like Activation, Create, Update, Delete, Read, etc.
 - Table:** A table with columns Kind, Object Class, Intent, Synchronization, and Tasks. It displays the message "No matching result found."
 - Bottom Buttons:** Back, Test connection, Refresh schema, Edit configuration, Show using wizard, Edit using wizard, and Edit XML.

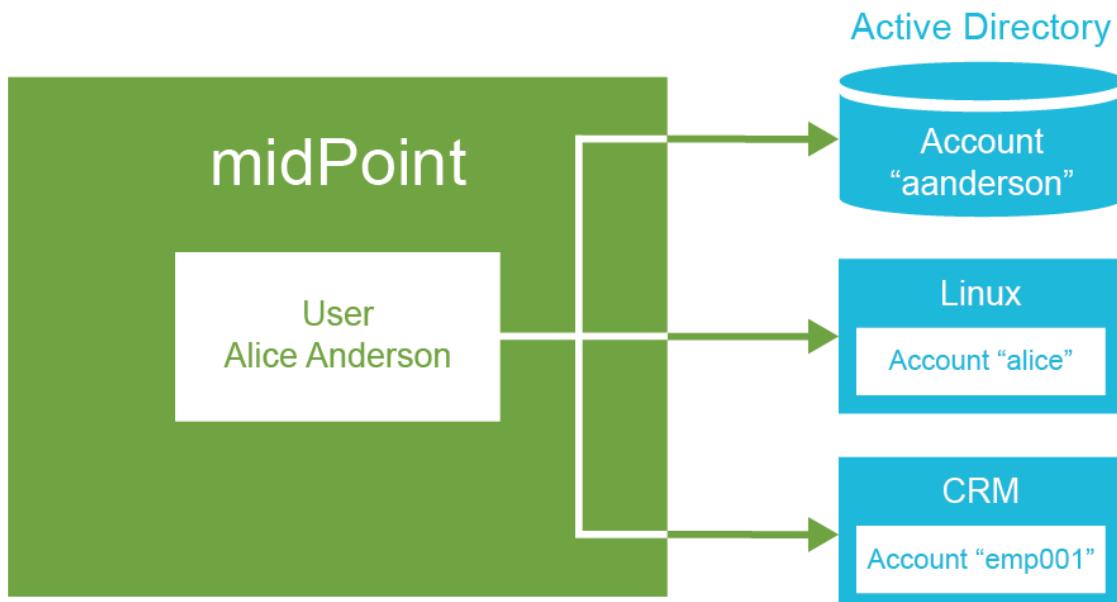
Eric can click on the button at the bottom of the screen to test connection to the resource. As this is a local CSV file there is no real connection. But the test checks that the filesystem path is correct,

that the file exists and that it can be opened. The test also loads *resource schema*. MidPoint reads the CSV file header to discover the structure of the data in the CSV file. The resource is now prepared for use.

There is not much that Eric can do with the resource yet. We need to explain a couple of essential midPoint concepts before moving forward with our case study.

User and Accounts

The concept of user is perhaps the most important concept in the entire IDM field. The term *user* represents physical person: an employee, support engineer, temporary worker, student, customer, etc. On the other hand the term *account* refers to the data structure that allows the user to access applications. This may be an account in the operating system, LDAP entry, row in the database table and so on. Typically, one user has many accounts – usually one account for each resource.



The data that represent users are stored directly in midPoint. While the data that represent accounts are stored "on the resource side". Which means accounts are stored in the connected applications, databases, directories and operating systems. Accounts are not stored in midPoint. Under normal circumstances MidPoint keeps just account identifiers and some meta-data about the accounts. All other attributes are retrieved when needed. MidPoint is using the connectors to fetch account data.

We will strictly distinguish the terms *user* and *account* in this book. Such a strong distinction is also made in the midPoint user interface and documentation. It is very helpful to get used to this terminology.

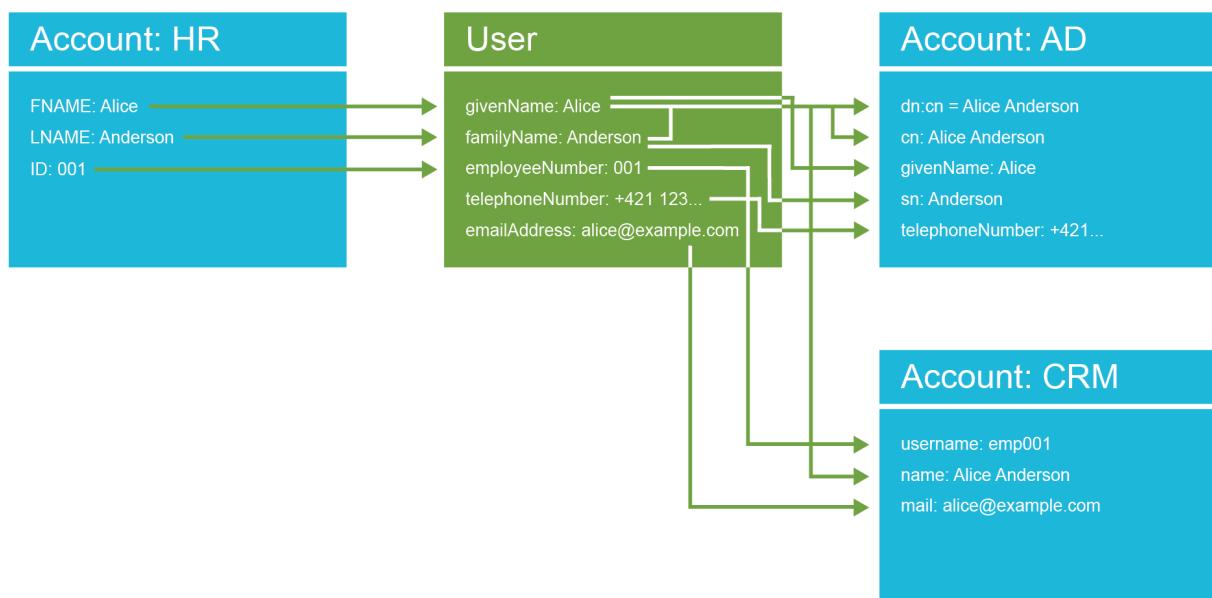
Accounts are linked to users that own the accounts. Therefore midPoint knows which account belongs to which user. MidPoint can list all the accounts for any user, it can synchronize the data, it can disable all the accounts of a particular user and so on. This user-account *link* is almost always

automatically established and maintained by midPoint.

MidPoint comes with a built-in data model (schema) for users. It contains properties that are very often used to describe users such as full name, e-mail address and telephone number. There is a reasonable set of properties that should be a good starting point for most deployments. Of course, as most midPoint objects, the user schema can be extended with custom properties if needed.

However, there is no built-in data model for accounts. Such data model would not be possible. Every resource may have different account attributes. There may be different names, different types and the values may have different meaning. MidPoint is designed to handle those differences. Schema for resource accounts is dynamically discovered when midPoint connects to the resource for the first time. MidPoint interprets the schema and automatically adapts to it. E.g. when midPoint displays information about an account, the user interface fields are dynamically generated from the discovered schema. MidPoint does that all by itself. No extra configuration and no coding is necessary.

Account schema may significantly differ from resource to resource. Yet midPoint must be able to synchronize all the accounts from any kind of resource imaginable. In this case the *user schema* works as a unified data model. The schema of each account is mapped to the user schema.



Getting back to our ExAmPLE story, Eric has a HR resource configured. Therefore he can see the "accounts" that the users have in the HR system. Eric opens the resource detail page in the midPoint GUI, clicks on *Accounts* tab and then on the *Resource* button (we'll explain that later). The list of accounts appears:

	Name	Identifiers	Situation	Intent	Owner	Pending operations
<input type="checkbox"/>	001	empno:001		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	002	empno:002		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	003	empno:003		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	004	empno:004		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	005	empno:005		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	006	empno:006		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	007	empno:007		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>
<input type="checkbox"/>	008	empno:008		unknown		<input type="button" value="More..."/> <input type="button" value="Advanced"/>

All that can be seen in this list are just employee numbers, because employee number is set as the primary identifier for the HR system. Clicking on the link will display more details. In fact these are not real accounts. These are lines in the CSV file exported from the HR database. But they describe some aspects of *identity* and therefore midPoint interprets them as accounts. For midPoint "account" is a generic term used to describe any resource-side data structure that represents the user.

Initial Import

The *user* is a central concept for any IDM system and midPoint is no exception. MidPoint needs reliable information about users to work correctly. The HR system is usually a good source of user information. Eric needs to get that information from the HR system into midPoint. He has already set up a resource that connects to the CSV file exported from the HR system. But the resource does not do anything by default. It has to be configured to pull the information from the file into midPoint. What Eric needs is a set of *mappings*. Mapping is a mechanism for synchronization of attribute values between user and linked accounts. In this case Eric needs *inbound* mappings to import the data. Inbound mappings synchronize the value in the direction from the resource into midPoint. Eric can open the resource definition in the configuration wizard in GUI and he can add the mappings there. Or he can simply look at the configuration samples again and add the mappings in the XML form. Inbound mapping looks like this:

```

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

```

This is a mapping that maps the account (HR) attribute `firstname` to user (midPoint) property `givenName`. This tells midPoint to always update a value of user's given name when the mapped HR attribute changes. Eric adds similar mappings for all the attributes in the HR export file. Eric also needs to add *synchronization* section to the resource definition. The synchronization section instructs midPoint to create a new user for each new account. This is exactly what Eric wants: create a user for each HR account. Eric then re-imports the modified XML file into midPoint.

MidPoint is now ready to synchronize the attributes. But we still need a *task* to pull all the data from the HR system. Eric navigates to the page that shows the list of HR accounts. At the bottom of that page there is a big *Import* button that can be used to manage the import tasks. Eric clicks on that button and creates a new import task. The task is started and it runs for a couple of seconds. After the task is done Eric can look at users in midPoint:

	Name	Given name	Family name	Full name	Email	Accounts	
<input type="checkbox"/>	001	Alice	Anderson			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	002	Bob	Brown			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	003	Carol	Cooper			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	004	David	Davies			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	005	Erin	Evans			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	006	Frank	Fox			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	007	Goerge	Green			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	008	Harry	Harris			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	009	Isabella	Irvine			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	010	Jack	Jones			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	011	Kate	Knowles			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	012	Lily	Lewis			1	<input type="button" value="Edit"/>
<input type="checkbox"/>	013	Max	Morgan			1	<input type="button" value="Edit"/>

Eric can see details about the user by clicking on the username:

The screenshot shows the 'Edit User' page for 'User '001''. The left sidebar has sections for SELF SERVICE (Home, Profile, Credentials, Request a role) and ADMINISTRATION (Dashboards, Users, All users, Edit user, New user, Org. structure, Roles, Services, Archetypes). The main area has tabs: Basic, Projections (1), Assignments (0), History (0), Cases (0), Personas (0), Delegations (0), and Delegated to me (0). The 'Basic' tab is selected. It displays user properties: Name (001), Full name (empty), Given name (Alice), Family name (Anderson). Below these are sections for Activation (password is set) and Password (Value: password is set). Top right buttons: Enabled (checked), No assignments (unchecked), No organizations (unchecked).

This page shows all the details about the user that midPoint knows about. The details are sorted to several tabs and we are going to explain all of that later in this book. For now we only care about first two tabs. The *Basic* tab shows user properties as midPoint knows them. These properties are stored in midPoint repository. MidPoint has quite a rich data model that can be used out-of-the-box, but the GUI only shows those properties that are actually used. The "name", given name and family name were imported from the HR resource and that's what the page shows.

Let's have a look at the second tab now:

The screenshot shows the 'Edit User' page for 'User '001''. The left sidebar has sections for SELF SERVICE (Home, Profile, Credentials, Request a role) and ADMINISTRATION (Dashboards, Users, All users, Edit user, New user, Org. structure, Roles, Services, Archetypes). The main area has tabs: Basic, Projections (1), Assignments (0), History (0), Cases (0), Personas (0), Delegations (0), and Delegated to me (0). The 'Projections' tab is selected. It displays a table of user accounts:

	Name	Resource	Object Class	Kind	Intent	Pending operation
<input type="checkbox"/>	001	HR System	AccountObjectClass	ACCOUNT	default	

Bottom right of the table: 1 to 1 of 1, navigation buttons (<<, <, >, >>), and a settings gear icon.

The *Projections* tab shows user's accounts. Currently there is only one account and it is the HR account that was used to import the data. Account details are displayed by clicking on account identifier:

The data that are displayed here are really fresh. Account details were retrieved from the resource at the very moment that the account was displayed. This is the difference between user data and account data: user data are kept in midPoint repository, while account data are retrieved from the resource as needed.

The user and the account are linked. MidPoint remembers that this user originated from this specific HR account. If the HR account is modified then the change is synchronized and applied to the user data. The mappings are not just for the import. They can work continually and keep the account and user data synchronized all the time.

Assignments and Projections

The concepts of an account is all about the reality: it shows the data that are there at this very moment. It shows what *is* there. But identity management is all about policies. Policies, by definition, specify what *should be* there. Policies specify what is right. But as every citizen knows all too well, the things that *are* and the things that *should be* do not always match perfectly. We are no idealists. Therefore we have designed midPoint from the day one to acknowledge that there may be a difference between reality and policy. Primary role of midPoint is to manage that difference. And completely align policy and reality in the long run.

This kind of thinking is easy to see in midPoint user interface. There is *Projections* tab in the user details page. It shows the accounts that the user has right now. It shows the real state in which the accounts are. It shows the reality. And then there is *Assignments* tab. This tab shows the policy. This tab shows what accounts, roles, organizations, or services are assigned to the user. This tab shows what user should have.

The screenshot shows the midPoint interface for editing a user named '001'. The left sidebar has sections for SELF SERVICE (Home, Profile, Credentials, Request a role) and ADMINISTRATION (Dashboards, Users, All users, Edit user, New user). The main area shows a user card for '001' with a red profile picture. The 'Assignments' tab is selected, showing 0 assignments. There are tabs for Basic, Projections (1), Assignments (0), History, Cases (0), Personas (0), Delegations (0), and Delegated to me (0). Below these are filters for All, Role, Organization, Service, and Resource. A search bar with 'Advanced' and a 'More...' button is at the top right. The results table has columns for Name, Activation, and More data. It shows two entries: 'HR System' and 'LDAP', both with activation status 'Not activated'. A message at the bottom says 'No matching result found.'

To demonstrate how the assignments work we need a new resource. Therefore let Eric connect a new resource to midPoint. This time it will be new, clean and empty LDAP server. So Eric once again locates the proper example, modifies the configuration and imports it to midPoint. In a while there is a new LDAP resource. Eric wants to synchronize all the users to the LDAP server. Therefore Eric has to define mappings once again. But this time these will be *outbound* mappings as Eric wants to propagate data out of midPoint into the (LDAP) resource. We will cover the details of mapping configuration later, so now let's just see the results. We have two resources now:

The screenshot shows the midPoint 'Resources List' page. The left sidebar includes SELF SERVICE and ADMINISTRATION sections. The main area displays a table of resources with columns for Name, Connector type, and Version. Two entries are listed: 'HR System' (com.evolveum.polygon.connector.csv.CsvConnector, Version 2.3) and 'LDAP' (com.evolveum.polygon.connector.ldap.LdapConnector, Version 2.3). A toolbar below the table includes buttons for creating (+), deleting (-), and managing resources. Navigation buttons at the bottom indicate '1 to 2 of 2'.

But how do we create an account on that LDAP resource? The right way to do this is to let midPoint know that a user *should have* an account on that resource. In midPoint terminology we say, that we are *assigning* the resource to the user. All that Eric needs to do is to navigate to user details page, click on the *Assignments* tab, use *New assignment* button to add an assignment for the LDAP resource and click *Save*:

The screenshot shows the midPoint interface with a sidebar on the left containing 'SELF SERVICE' and 'ADMINISTRATION' sections. The main area is titled 'Select object(s)' and shows a search bar with tabs for 'Role', 'Org', 'Service', and 'Resource'. The 'Resource' tab is selected, showing a table with columns 'Name' and 'Description'. Two resources are listed: 'HR System' and 'LDAP'. The 'LDAP' resource has a checked checkbox and a detailed description: 'LDAP resource using a ConnId LDAP connector. It contains configuration for use with OpenLDAP servers.' Below the table are 'Parameters' dropdowns for 'Kind' (set to 'Undefined') and 'Intent' (set to 'Undefined'). A navigation bar at the bottom shows '1 to 2 of 2' with buttons for navigating between pages.

After the click on *Save* button a lot of complex things happen. But simply speaking midPoint recomputes what the user should have and what the user has. MidPoint detects that the user should have an LDAP account now (because there is a new assignment for it). But no such account exists. Therefore midPoint creates the account.

When Eric opens the user details again and navigates to the *Projections* tab he can see that there are two accounts now:

The screenshot shows the user details page for a user named '(001)'. The top navigation bar includes tabs for 'Basic', 'Projections' (selected), 'Assignments', 'History', 'Cases', 'Personas', 'Delegations', and 'Delegated to me'. The 'Projections' tab shows a table with columns: 'Name', 'Resource', 'Object Class', 'Kind', 'Intent', and 'Pending operation'. There are two entries: one for '001' (Resource: HR System, Object Class: AccountObjectClass, Kind: ACCOUNT, Intent: default) and another for 'uid=001,ou=people,dc=example,dc=com' (Resource: LDAP, Object Class: inetOrgPerson, Kind: ACCOUNT, Intent: default). A green '+' button is at the bottom left of the table.

There is an HR account that was used to create the user in the first place. And there is also LDAP account that was created as a reaction to a new assignment.

The screenshot shows a user profile page for '001'. At the top, there's a red header bar with a person icon, the identifier '(001)', and two checkboxes: 'Enabled' (checked) and 'No organizations'. Below the header, a navigation bar includes tabs for 'Basic', 'Projections (2)', 'Assignments (1)', 'History', 'Cases (0)', 'Personas (0)', 'Delegations (0)', and 'Delegated to me (0)'. The main content area displays the user's distinguished name: 'uid=001,ou=people,dc=example,dc=com' with a copy icon. Below it, the resource is identified as 'LDAP' and the object class as 'inetOrgPerson'. The 'Intent' is set to 'default'. A section titled 'Attributes' lists various user properties:

Entry UUID	697038da-8dc9-1039-80f1-6dc8fba249e2	<input type="button" value=""/>
Distinguished Name	uid=001,ou=people,dc=example,dc=com	<input type="button" value=""/>
Login Name	001	<input type="button" value=""/> <input type="button" value=""/>
description	Created by midPoint	<input type="button" value=""/> <input type="button" value=""/>
Surname	Anderson	<input type="button" value=""/> <input type="button" value=""/>
Given Name	Alice	<input type="button" value=""/> <input type="button" value=""/>
createTimestamp	10/28/2019	<input type="button" value=""/> : <input type="button" value=""/> 22 <input type="button" value=""/> PM <input type="button" value=""/>
Common Name	001	<input type="button" value=""/> <input type="button" value=""/>

[Show empty fields](#)



Careful reader may have noticed that the two accounts have vastly different attributes. That's right. Every account has a different *schema*. MidPoint automatically discovers the schema. Then midPoint dynamically interprets the schema to display the attributes in GUI, to validate the inputs, to check for errors in mappings and so on. MidPoint does everything by itself without any need to write a single line of code. MidPoint is completely based on the concept of schema and it takes full advantage of it.

There is reality and there is policy. There are accounts and there are assignments. Ideally these two things should match perfectly. And midPoint will try really hard to make them match. But there may be exceptions. Careful reader surely noticed that there is HR account but there is no assignment for that account. And yet midPoint has not deleted the HR account. That is because the HR system is what we call a "pure source" system. MidPoint does not write to the HR, it only reads from it. Writes to the CSV export file would be overwritten by the next export anyway, so there is no point in writing there. Therefore the HR resource has an exception specified in its configuration: it allows the HR account to exist even if there is no assignment for it. We can keep the HR account linked to the user by using this method. We can see the data that were used to create the user. This improves overall visibility and it greatly helps with diagnostics of configuration issues.

Roles

It would be a daunting task if Eric had to assign every individual account for every individual resource to every user. Typical IDM deployment has thousands of users and dozens of resources. Such deployment would be very difficult to manage using only direct resource assignments.

But there is a better way, of course. We can define some *roles*. The concept of *role-based access*

control (RBAC) is a well-established practice and the roles are really the bread-and-butter of identity management. The basic idea of RBAC is to group privileges into roles. Then the roles are assigned to the users instead of privileges. E.g. let's create a **Webmaster** role. Then put all the privileges that webmaster should have into that role. And let's assign the role to every user that works as a webmaster. This simplifies the privilege management. If there are two webmasters there is no need to think about the individual privileges that a webmaster should have. Just assign the role and the role has everything that is needed. It is also easy to change webmasters: unassign role from one user, assign to another user. It is also easy if you add a new web server. Just add the privilege for accessing new server into the **Webmaster** role. And all webmasters will have it.

That's the theory. But how does it work for Eric? First of all let's add a handful of new resources – to get some material for the roles. So now we have four resources: HR, LDAP, CRM and Portal. That's a good start. Let's do some role engineering now.

Many organizations have one role that almost every user has. It is often **Employee** or **Staff** role. This role gives access to all the systems that an employee should have access to: Windows domain login, e-mail, employee portal – things like that. The ExAmPLE company is no exception. In this case the basic role should create accounts in two systems:

- **LDAP server:** many applications are connected to LDAP and use it for authentication. We want every ExAmPLE employee to have account there.
 - **Portal:** this is enterprise intranet portal with lots of small services essential for every employee.

It is simple to create such role in midPoint user interface. Eric navigates to *Roles > New role*. Fills in the name of the new role (**Employee**). Then he goes to the *Inducements* tab. This is where the role definition takes place. Inducements are almost the same as assignments. However, inducements do not give access to the role itself. Inducements give access to the users that have this role. So they are kind of indirect assignments. Eric clicks on *New inducement* button and adds inducements for the two resources into the role:

Basic	Projections (0)	Assignments (0)	Cases (0)	Applicable Policies	Inducements (0)		
All	Role	Organization	Service	Resource	Policy rule	Induced entitlements	Focus mappings
<input type="checkbox"/>	Name	Activation 	More data	 			
<input type="checkbox"/>  LDAP	enabled			  			
<input type="checkbox"/>  Portal	enabled			  			
				1 to 2 of 2	     		

Eric clicks on *Save* button and the new role is created. Now it is ready to be assigned to the users. Eric goes on and assigns **Employee** role to user Bob:

The screenshot shows the MidPoint interface for managing roles. At the top, there's a red header bar with a user icon and the identifier '(002)'. To the right are three checkboxes: 'Enabled' (checked), 'No assignments' (unchecked), and 'No organizations' (unchecked). Below the header are several tabs: Basic, Projections (1), Assignments (1), History, Cases (0), Personas (0), Delegations (0), and Delegated to me (0). Under the 'Assignments' tab, there are four buttons: All (selected), Role, Organization, Service, and Resource. A search bar with a magnifying glass icon and an 'Advanced' button are also present. The main content area displays a table with one row. The first column has a checkbox and a person icon. The second column contains the name 'Employee'. The third column shows 'Activation' status as 'enabled'. The fourth column is labeled 'More data' and contains a minus sign and a checkmark icon. At the bottom of the table are edit and delete icons. Below the table, a pagination bar shows '1 to 1 of 1' and navigation arrows.

MidPoint automatically creates all the accounts given by the role:

This screenshot shows the MidPoint interface after assigning the 'Employee' role to user '(002)'. The top bar now shows 'Projections (3)' instead of 'Assignments (1)'. The table below lists three accounts created under this role. The columns are: Name (with a person icon), Resource (with a person icon), Object Class (AccountObjectClass), Kind (ACCOUNT), Intent (default), and Pending operation (checkboxes). The accounts listed are: '002' (Resource: HR System, Object Class: AccountObjectClass, Kind: ACCOUNT, Intent: default), 'uid=002,ou=people,dc=example,dc=com' (Resource: LDAP, Object Class: inetOrgPerson, Kind: ACCOUNT, Intent: default), and another '002' (Resource: Portal, Object Class: AccountObjectClass, Kind: ACCOUNT, Intent: default). The bottom of the table has a green plus icon and a pagination bar showing '1 to 3 of 3'.

There is the HR account that was used to create the Bob user record in the first place. And then there are the two accounts that were created because Bob has the **Employee** role.

This operation works in both directions: if Eric unassigns the **Employee** role, the accounts given by the role will be deleted. Eric can create any number of roles like this: roles for Sales agents with CRM access, roles for Sales managers with higher CRM privileges and so on. MidPoint is designed to handle large number of roles. Each role can have its own combination of resources. MidPoint seamlessly merges the privileges given by all the roles a user has. E.g. if two roles give CRM access to the user, only one CRM account will be created. If one of these roles is unassigned then CRM account remains there. The account is not deleted yet because it is given by the other role. Only when the last CRM role is removed that's the point where the account gets deleted. MidPoint takes care of all that logic.

Of course, there is much more that the roles can do:

- Roles can assign accounts to groups, give the privileges and manage account entitlements.
- Roles can mandate specific account attribute values, e.g. clearance levels, compartments, etc.

- Roles may contain custom logic (scripts).
- Roles may be hierarchical: there may be roles within roles.
- Roles may be assigned for a specified time.
- Roles may be conditional and parametric.
- ... and much much more.

Roles are really the essence of identity management. We will be dealing with roles in almost all the parts of this book.

There Is Much More

Eric the Engineer has done a few basic steps to configure midPoint as an identity management system for his company. But this is still a very basic configuration. Careful readers have already noticed a lot of things that need to be done. E.g. employee full name is not automatically generated. Employee numbers are used as identifiers and we would like to have something that is more user-friendly. We would like to automatically assign the [Employee](#) role instead of doing that manually. And so on. There are still a lot of things to improve. Fortunately, all of that is very easy to do with midPoint once you know where to look. And we will be dealing with all these things in the rest of this book. New functionality will be administered to the ExAmPLE solution in small doses in each chapter – together with a proper explanation of midPoint principles. MidPoint is a very flexible and comprehensive system and there are still a lot of things to learn. This chapter covered only a minuscule part of midPoint functionality.

What MidPoint Is Not

Now you probably have some idea what midPoint is. However, it is also very important to understand what midPoint is not. Identity and Access Management (IAM) field is a combination of many technologies and it may sometimes be quite confusing. That is perhaps the reason why the midPoint team occasionally gets questions about midPoint functionality that simply do not make much sense.

First of all, midPoint is not an authentication server. MidPoint is not designed to validate your username and password. Yes, midPoint maintains data about users (including passwords). But the data model that midPoint maintains is quite complex. It is not meant to be exposed to applications directly. That would not be efficient.

If you want midPoint to manage users but you also want your applications to have a centralized authentication services there is a solution: publish the data to the LDAP server. Connect LDAP server to midPoint as a resource and let midPoint populate and maintain the LDAP sever data. The application will not talk to midPoint directly. They will talk to the LDAP server. This is better for everybody: LDAP is a standard protocol well supported in many applications. LDAP servers are also extremely fast and scalable ad nauseam. Therefore use the combination of midPoint and an LDAP server of your choice. That's what people usually do and it works perfectly.

As midPoint is not an authentication server it obviously is not a Single Sign-On (SSO) server either. If you want SSO you will need a dedicated SSO server. There are plenty of SSO servers to choose

from in both the closed-source and open-source worlds. You will also need a scalable directory system (LDAP) to store the data for the SSO server. And you will probably still need midPoint to manage the data.

One of the things that seems to be shrouded in a lot of confusion is authorization. To get the record straight from the beginning: midPoint is not an authorization server. It is not a policy decision point (PDP) and it definitely is not a policy enforcement point (PEP). You cannot rip authorization out of your application and just “use midPoint for that”. That does not work.

You can think of midPoint as a policy management point (PMP). MidPoint has a lot of sophisticated authorization-related logic inside its core. But that logic is not designed to answer questions such as “Is subject S authorized to execute operation O on object X?”. MidPoint logic is different. MidPoint is not concerned with making authorization decisions. It is concerned about managing the authorization policies. MidPoint sets up the authorization policies in target applications. And the applications evaluate these policies themselves. This is a much more efficient and more reliable method. Unlike authentication, the authorizations decisions are done all the time. Authorization is evaluated at least once per every request. If the application makes these decision internally then there is no need to a round-trip to the authorization server. Performance is significantly increased. And there is no single point of failure. MidPoint failure will not interrupt authorization flow because the application has all the data inside. One less component to cause a failure. And still, the policies are centrally managed by midPoint. When a policy changes midPoint updates all the affected applications. You get all the benefits without the usual drawbacks.

MidPoint does what it is supposed to do: it *manages* identities, entitlements, organizational structures and policies. But midPoint does not do things that are not necessary. It does not do the things that other technologies already do well. MidPoint does not reinvent the wheel. There is no need for this. MidPoint is not the wheel. MidPoint sits above all the wheels. MidPoint is the chauffeur.

Chapter 3. Installation and Configuration Principles

The *Guide* is definitive. Reality is frequently inaccurate.

— The Hitchhiker's Guide to the Galaxy, The Restaurant at the End of the Universe by Douglas Adams

This chapter provides instructions for installation and initial configuration of a midPoint system. The instructions describe installation on Linux system because that is by far the most common operating environment for midPoint deployments. However, midPoint is platform-independent and it can run on any environment where Java is running. Any experienced engineer will have no trouble adapting these instructions to fit a different operating system.

MidPoint installation described in this chapter is a very basic one. It is ideal for initial exploration of midPoint, development of midPoint configurations, demonstrations and similar purposes. It is a very convenient installation and we use it every day for development work. However, to use midPoint in a production deployment the installation need to be slightly adjusted. The adjustments are mentioned in this chapter, but the full description of the production-ready installation is provided in later chapters. This chapter gives you midPoint installation that is ideal to get you started.

Requirements

MidPoint will run on almost any machine. All you need is approximately 4GB RAM. That's perhaps the only real limiting factor. If you look for more formal system requirements definition then you will find that in midPoint wiki (see [Additional Information](#) chapter at the end of the book).

From the software side you will need:

- **Java 11** runtime environment (JRE) or development environment (JDK). Any JRE or JDK should work. You can use the packages from your operating system distribution. Or you can download Java and install it as a standalone package. Both should work well. Just do not forget to set your `PATH` and `JAVA_HOME` environment variables to point to the Java installation.
- **MidPoint distribution package.** Download the latest version of midPoint from the Evolveum website. You are looking for an archive that looks like `midpoint-4.0-dist.zip`. This archive contains everything that you will need to run midPoint.

We recommend to use the latest available versions of all software packages when dealing with midPoint. We are trying really hard to always keep midPoint up-to-date with the rest of the technologies.

Installation

MidPoint is Java web application distributed in a stand-alone package. The distribution package contains everything that midPoint needs to run – except for Java platform itself. Therefore as long as the Java platform is installed all that is needed to run midPoint is to start it:

1. Extract the files from the distribution package to a location where you want to install midPoint.
2. Locate `start.sh` (Unix) or `start.bat` (Windows) script in midPoint distribution package. It should be located in bin directory.
3. Execute the start script

And that is pretty much it. MidPoint will start. It will initialize the embedded web container, database and all the other midPoint components. That can take a minute or two. After the application is initialized you can access it by connecting to midPoint HTTP port, which defaults to 8080. You can start working with midPoint now.

MidPoint User Interface

Use the following URL to access midPoint user interface:

<http://hostname:8080/midpoint>

Log in with the following credentials:

Username: `administrator`

Password: `5ecr3t`

Now you are logged-in as the `administrator`. This user has superuser privileges therefore you can see everything and you can do anything in the midPoint user interface.

MidPoint user interface is structured. It has the same layout and controls for all the user interface areas:

Primary tool for user interface interaction is the menu. MidPoint user interface is functionally divided into three parts, therefore there are also three parts of the menu:

- **Self-service** user interface deals with the things that the user can do for oneself: displaying list of account, changing password, requesting a role and so on. This is relatively simple part of the user interface. It is often accessible to all the users.
- **Administration** user interface deals with management of other users, roles, organizational structures and similar midPoint objects. This is a very comprehensive and considerably complex part of the user interface. Usually only privileged users have access to this part of the

user interface. This part of user interface is often used to support delegated administration and role management therefore it is also meant for security officers, resource owners, role engineers and similar expert users.

- **Configuration** user interface deals with configuration of midPoint system itself. It is used to customize midPoint behavior, set fundamental policies and rules that form the foundation of midPoint deployments. This part of user interface is usually used only by identity management engineers.

The menu can be hidden by clicking on the button at the top of the screen. The top bar also contains the title of the current user interface page and breadcrumbs. Breadcrumbs show where you currently are in the user interface and how you got there. The breadcrumbs can be used to “find your way home” and back to the previous page. The use of browser *Back* button is not recommended. Please use the *Back* buttons that are present in midPoint user interface or use breadcrumbs.

User Interface Areas

MidPoint user interface is quite rich. Following list provides short description of the most important parts of the user interface.

- **Home** page gives a brief status about users own accounts, requests, work items and so on. This is a page designed to be the first page that will be displayed to the end user after log in to midPoint.
- **Profile** page allows users to see or edit their own profile.
- **Credentials** page allows users to change their own credentials, such as password.
- **Role request** page allows users to select the roles that they need and then request assignment of the roles.
- **Dashboard** pages shows a couple of dashboards designed to provide a lot of useful information at the first sight. The built-in system dashboard shows statistics about midPoint installation.
- **User** pages list users in midPoint, allows to create and edit users.
- **Organizational structure** pages show the organizational structure trees. Many parallel organizational structures can be managed here, such as tree-like functional organizational structure, flat project-oriented structure, role catalogs and so on.
- **Role** pages allow to list and manage roles. Roles can be created and defined in this part of the user interface.
- **Service** pages allow to list and define services, such as devices, servers, applications and so on.
- **Archetype** pages define specific object types that can be used to customize behavior of midPoint objects.
- **Resource** pages list and manage resources. New resource can be defined here, associated with the connector, tested, etc.
- **Cases** pages list the things that the users have to do. Work items are created if user has to approve something or if there is some manual step in the process.
- **Certification** pages deal with access certification (re-certification, attestation). Certification

campaigns can be created and managed here.

- **Server task** pages show the tasks that are running on midPoint servers. These may be scheduled synchronization tasks, import tasks, running user requests – everything that runs on the servers and cannot be executed immediately in a synchronous way.
- **Report** pages allow defining and running reports. These pages typically deal with scheduled printable reports.
- **Configuration** area contains many pages that manage midPoint configuration: system default configuration, repository objects, logging, bulk actions and so on.

User Interface Concepts

MidPoint user interface is using the same concepts and controls in all its parts. For example all the lists of all the objects (users, roles, ...) look like this:

The screenshot shows a table view of roles in MidPoint. The columns are: "Select all" checkbox, Name (with a dropdown arrow), Display Name, Description, Identifier, Projections, and Action buttons. The rows represent different role types: Approver, Delegator, Employee, End user, Reviewer, and Superuser. Each row has a detailed description in the Description column. The Action buttons include a dropdown menu with options: Enable, Disable, Reconcile, and Delete. The bottom left corner features buttons for Add, Refresh, Import, and Export. The bottom right corner shows Paging controls with a page number 1.

"Select all" checkbox						Search bar	Name: All	More...	Advanced
	Name	Display Name	Description	Identifier	Projections				
<input type="checkbox"/>	Approver		Role authorizing users to make approval decisions on work items.		0				
<input type="checkbox"/>	Delegator		Role authorizing users to delegate their own privileges to any other user.		0				
<input type="checkbox"/>	Employee				0				
<input type="checkbox"/>	End user		Role authorizing end users to log in, change their passwords and review assigned accounts. Note: This role definition is just an example. It should be tailored for each specific deployment.		0				
<input type="checkbox"/>	Reviewer		Role authorizing users to make decisions on certification cases.		0				
<input type="checkbox"/>	Superuser		Role that gives user full authorization in MidPoint.		0				

Add, refresh, import and export buttons

Paging controls
1 to 6 of 6 << < 1 > >> ⚙

Each row represents one object: user, roles, service, task, etc. There is also a color-coded object icon. The search bar at the top can be used to look for a specific object or to filter the object view. Right side is reserved for action buttons. Buttons in the table header trigger actions that apply to all selected objects. Buttons in each table row trigger actions that apply only to that individual object. The buttons in the bottom-left corner execute global actions, such as creating or importing new object, exporting objects and refreshing the view. The *Import* button is especially useful. It allows importing new object in XML/JSON/YAML form. Paging controls are in the bottom-right corner.

MidPoint has a unified color-code that makes the navigation easier. Users, roles and other object types have their specific color and icon. This indicates the object type and it is used whenever possible: menu, information boxes, lists, box title accents and so on. The primary colors and icons are shown in the dashboard:



All user-related controls are red, all controls that deal with organizational structure are yellow. Roles are green. And so on. This color code is applied consistently through the midPoint user interface.

Similar color code applies to object icons when displayed in user lists. However, the color that is used there does not indicate object type but rather an *archetype*. Archetypes are sub-types that are often used to distinguish similar objects. For example archetypes can be used to sort users to employees, contractors, customers and so on. Look and behavior of "archetyped" objects is configurable. Default midPoint configuration contains just a couple of archetypes. Those archetypes apply red color to system users and roles. Objects that do have any archetype behave in a different way. Their color indicate status of the object:

- Black icons indicate normal state. It suggests that there is nothing special to see here.
- Gray icons indicate non-active state. It suggests that the object is disabled, archived or there is another reason why the object is not active.
- Blue icons indicate typical end-user access. It suggests that the object has an access, but the access is limited only to safe, non-privileges operations. E.g. users with the end-user role.
- Yellow icons indicate management capabilities. E.g. users that are managers of organizational units.



All objects are equal in midPoint. MidPoint will handle users, roles, organizational units and services in the same way. The lists used to display these objects are the same, the pages that display object details are the same. All the objects have properties, they can be enabled/disabled in the same way, they are subject to authorizations in the same way and so on. It is a midPoint philosophy to design several powerful principles and then apply them over and over again.

Object Details Page

When a user clicks on a name of any object in the object list then object details page appears. The detail pages for common midPoint objects such as user or role are very similar to each other. They have the same layout and controls. E.g. user details page looks like this:



Ing. Katarína Valaliková (katkav) Name and identifier
 Software Developer **Title**
 Development Section **Organizational unit**

Enabled
 Manager

Tags

Details tabs

- Basic
- Projections 0
- Assignments 4
- History
- Cases 0
- Triggers 0
- Personas 0
- Delegations 0
- Delegated to me 0

Properties * ⓘ View control buttons (sort, show/hide)

Name ⓘ	katkav	⋮
Full name ⓘ	Ing. Katarína Valaliková	⋮
Given name ⓘ	Katka	⋮
Family name ⓘ	Valaliková	⋮
Honorific Prefix ⓘ	Ing.	⋮
Title ⓘ	Software Developer	⋮
Email ⓘ	katarika.valalikova@evolveum.com	⋮
Employee Number ⓘ	003	⋮
Locality ⓘ	Bratislava	⋮
Jpeg photo ⓘ	Browse... No file selected.	⋮

Show empty fields

Activation ⓘ ⌄ ⓘ Show empty fields

Password * ⓘ ⌄ ⓘ Show empty fields

Options Operation options

Force Reconcile Execute after all approvals Keep displaying results

Back Preview changes Save **Operation buttons** Edit raw

There is an information area at the top of the page. That area shows user photo (or icon) and provides some basic information such as user name and identifier. It also shows where the object belongs in the organizational structure. There is also a couple of "tags" that show interesting details about the object: whether the object is enabled, whether it has special privileges and so on.

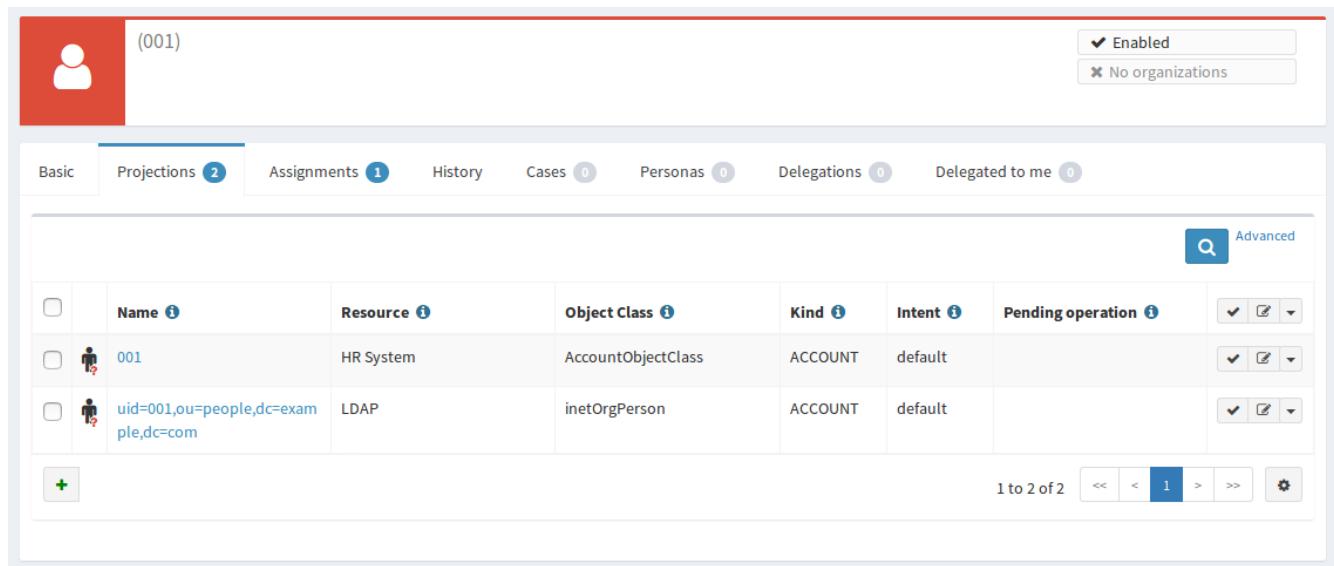
The screen below the information area is divided into several tabs. Each tab shows one aspect of the object. There are tabs that show projections, assignments, inducements – we will come to that later in this book. The first tab is perhaps the most interesting right now. It contains a dialog that shows basic object properties: the attributes of the object. Properties are displayed and they can be edited – depending on the authorizations of currently logged-in user. In addition to the basic properties there are also other sections. E.g. activation section shows whether the object is enabled or disabled, it shows the activation dates and other activation details. The credentials section allows changing password and other credentials. There are several little buttons at the top of each section. One of them can be used to change the ordering in which the properties are shown. The other button toggles the display of metadata. There may be additional buttons depending on the situation.

Operation options and buttons are located at the bottom of the details page. The buttons initiate the operations. The most common operation is just to save the changes and that's what the **Save** button is for. Saving the changes is a universal way how to start almost any operation: change of user

properties, assignment of roles, change of password, user disable, etc. When you make edits in any of the tabs on the details page then nothing really happens yet. MidPoint just remembers what you are editing. The operation is executed only when you click the Save button. This is our method how to execute several changes in one operation. It may require some time to get used to it. Just do not forget to click the save Button.

Operation options are used to modify the behavior of the operation. These options may force to execute operations that fail to pass midPoint internal checks. There is an option to reconcile the data even if midPoint thinks that reconciliation is not needed. And so on. Checking or unchecking these options influences the way how midPoint executes the operation.

MidPoint user interface often needs to present objects that are internally quite complex. It does not make sense to present all the details at once. These objects need to be presented in quite a compact form that can be expanded to show the details. This applies to list of user's accounts, assignments, role inducements, etc. The objects are initially displayed as in a form of a simple list, displaying only the basic data:



The screenshot shows a user profile page for a user named '(001)'. At the top right, there are two checkboxes: 'Enabled' (checked) and 'No organizations' (unchecked). Below the header, there is a navigation bar with tabs: Basic (selected), Projections (2), Assignments (1), History, Cases (0), Personas (0), Delegations (0), and Delegated to me (0). An 'Advanced' search bar is located above the main table. The main content is a table listing user projections. The columns are: checkbox, Name (with a person icon), Resource, Object Class, Kind, Intent, Pending operation, and a set of checkboxes. Two rows are visible:

	Name	Resource	Object Class	Kind	Intent	Pending operation	Actions
<input type="checkbox"/>	001	HR System	AccountObjectClass	ACCOUNT	default		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	uid=001,ou=people,dc=example,dc=com	LDAP	inetOrgPerson	ACCOUNT	default		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>

At the bottom left is a green '+' button, and at the bottom right are navigation buttons for pages 1 to 2 of 2.

The list above shows user's *projections*. Those are usually accounts that are linked to user object. Click on account name shows account details:

Account details display is shown in place of user details. This may be slightly confusing. But account details can be often complex, therefore all the available screen space is needed to display them. The *Cancel* and *Done* buttons at the bottom can be used to return back to *Projections* tab. Click on *Done* button will not start the operation yet, it only changes the view. Therefore do not confuse those buttons with *Back* and *Save* buttons on the very bottom on the page. The operation starts when *Save* button is clicked.

MidPoint Configuration Basics

The principle of midPoint configuration is quite different from what would a typical system administrator expect. There are almost no configuration files in midPoint. MidPoint is storing vast majority of its configuration in its configuration database. There are several reasons for this:

- MidPoint configuration is **complex**. MidPoint configuration is not what a typical system administrator would think of like a "configuration". It contains numerous resource definitions that in turn contains mappings that in turn may contain scripts. There are roles, policies, templates, ... and these objects are too complex to be expressed in simple configuration files.
- MidPoint configuration is **scalable**. It is no exception that a midPoint deployment has thousands of role definitions or organizational units, tens of resource definitions and a significant number of templates and policies. All of that needs to be stored efficiently, so midPoint can handle deployments that manage millions of identities. The configuration also needs to be searchable. Managing thousands of roles in plain text files simply won't work.
- MidPoint configuration needs to be **available**. There are midPoint deployments with several nodes working together in a cluster. Configuration change done on one node has to be seen by other nodes. Simple configuration files would not work here.

Therefore midPoint has a completely different approach to configuration. The configuration is stored in a form of well-defined structured objects in the midPoint database. We call that database *midPoint repository*.

Configuration Objects

Everything is an object in midPoint. Every piece of configuration is represented as a structured object and stored in midPoint repository. Object may look like this:

```
<role oid="8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc">
    <name>Basic User</name>
    <description>Basic user role. Almost all users have it.</description>
    <requestable>true</requestable>
    <inducement>
        <targetRef oid="f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61" type="RoleType"/>
    </inducement>
</role>
```

Every object has its identifier. We call that identifier *OID* which stands for "object identifier" (it has nothing to do with LDAP or ASN.1 OIDs). OID is usually randomly-generated universally unique identifier (UUID). OID value has to be unique in a whole system. This identifier is *persistent* – it is assigned to the object and it should never change. OID is used for internal purposes and it is almost never displayed to midPoint user.

Every object has a *name*. Name is human-readable and it can change any time. The value of name is usually displayed to users. This is the values that ordinary users understand as an identifier.

And then there are other object properties. Or rather *items*. Each type of midPoint object has a slightly different set of these items. That's what we call *schema*. The items may be simple properties such as string, integer or boolean values. But there also may be complex structures and references between objects. MidPoint data model is quite rich. It is in fact so rich that its description will take better part of this book, because description of the data model is also description of midPoint features.

You can see midPoint configuration objects in midPoint user interface by navigating to *Configuration > Repository Objects* and selecting object type. The following picture shows objects of type "Role":

Options	<input type="checkbox"/> Use zip <input checked="" type="checkbox"/> Show all items	Oid filter	<input type="button" value="Search"/>	Role	<input type="button" value="Name: All"/>	<input type="button" value="More..."/>	<input type="button" value="Advanced"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/> Name						
<input type="checkbox"/>	Approver 00000000-0000-0000-0000-00000000000a	Role authorizing users to make approval decisions on work items.	<input type="button" value="Export"/>	<input type="button" value="Delete"/>			
<input type="checkbox"/>	Delegator 00000000-0000-0000-0000-00000000000c	Role authorizing users to delegate their own privileges to any other user.	<input type="button" value="Export"/>	<input type="button" value="Delete"/>			
<input type="checkbox"/>	Employee db646296-3fc7-4061-8222-8a3fa6ae0786		<input type="button" value="Export"/>	<input type="button" value="Delete"/>			
<input type="checkbox"/>	End user 00000000-0000-0000-0000-000000000008	Role authorizing end users to log in, change their passwords and review assigned accounts. Note: This role definition is just an example. It should be tailored for each specific deployment.	<input type="button" value="Export"/>	<input type="button" value="Delete"/>			
<input type="checkbox"/>	Reviewer 00000000-0000-0000-0000-00000000000b	Role authorizing users to make decisions on certification cases.	<input type="button" value="Export"/>	<input type="button" value="Delete"/>			
<input type="checkbox"/>	Superuser 00000000-0000-0000-0000-000000000004	Role that gives user full authorization in MidPoint.	<input type="button" value="Export"/>	<input type="button" value="Delete"/>			

XML, JSON and YAML

The objects are stored in the midPoint repository in a native form which is hidden from midPoint users. However, the objects also have a human-readable representation. They can be represented in XML, JSON and YAML forms. All the objects can be imported into midPoint in any of those forms. They can be exported from midPoint in any of those forms. They can be even edited directly in midPoint using embedded editor. Just click on any object in the *Repository objects* page:

The ability to export, import and edit objects in XML/JSON/YAML form is absolutely essential,

because:

- It is **human-readable** (or rather administrator-readable). The configuration can be created, edited and maintained in your favorite editor and then imported into midPoint. It can be reviewed. It can be copied and pasted. Especially that. No system administrator can live efficiently without an ability to copy and paste.
- It is **transferable**. It can be exported from one system (e.g. development environment) and easily transferred to another system (e.g. testing environment). It can be easily backed-up and restored. It can be easily shared, e.g. in a form of configuration samples.
- It is **versionable**. The exported configuration can be easily put under any ordinary version control system. This is essential for deployment projects and configuration management.

Therefore the midPoint configuration has the best of both worlds. It is stored in database, so it can be processed efficiently, it can be made available and so on. But it also has a text form, so it can be easily managed.

The XML, JSON and YAML forms are considered to be equivalent. Objects can be written in any of these forms.

XML form of role object

```
<role oid="8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc">
    <name>Basic User</name>
    <description>Basic user role. Almost all users have it.</description>
    <requestable>true</requestable>
    <inducement>
        <targetRef oid="f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61" type="RoleType"/>
    </inducement>
</role>
```

JSON form of role object

```
{
    "role" : {
        "oid" : "8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc",
        "name" : "Basic User",
        "description" : "Basic user role. Almost all users have it.",
        "requestable" : true,
        "inducement" : {
            "targetRef" : {
                "oid" : "f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61",
                "type" : "RoleType"
            }
        }
    }
}
```

```
role:  
  oid: "8ebab0bc-7e7e-11e6-a7bc-57de1cd45ecc"  
  name: "Basic User"  
  description: "Basic user role. Almost all users have it."  
  requestable: true  
  inducement:  
    - targetRef:  
        oid: "f92e67c2-7e7e-11e6-a306-7bf6aa2e8c61"  
        type: "RoleType"
```

Most of the examples in this book are in XML notation. The XML form is almost always simplified for clarity: there are no namespace definitions, no namespace prefixes and so on. The complete files with all the details can be found in midPoint distribution package, midPoint source code or in other places. See [Additional Information](#) chapter for more details.

Maintaining MidPoint Configuration

When it comes to maintenance of midPoint configuration there are two practical methods how to do that.

First method is to maintain the configuration in midPoint: use midPoint wizards and user interface tools to create new objects and modify them. Export the objects in regular intervals so they are backed up, placed under version control and so on. This is an easy method to start with. But sooner or later you will probably figure out that you need the ability to copy and paste parts of the configuration. That you need to share the configuration with other team members. And that no user interface is ever as efficient as an experienced engineer with a good text editor.

Then there is a second method: maintain the configuration files in text form outside of midPoint. Import them to midPoint as needed. The objects can be imported in midPoint user interface by going to *Configuration > Import object* page. There are also import buttons in almost all the object list tables that also lead to that page.

It is much easier to maintain a proper version control and a good teamwork using the import method. It also seems to be more efficient once you get used to midPoint: pieces of configuration can be copied from samples, documentation or from other projects. This makes the work efficient. Although work with midPoint is "just" configuration and there is usually almost no programming, this method of work is quite close to the methods that software developers use. And we know that these methods work quite efficiently.

If you maintain the configuration files out of midPoint you can import them individually using midPoint user interface. This may seem like quite an uncomfortable way. But it works surprisingly well even for a mid-size projects. However, there is also a much better way. MidPoint has a plug-in into Eclipse IDE environment that allows you to maintain the configuration files in form of Eclipse projects. The plug-in allows easy download and upload of changed configuration files. As Eclipse also has good integration for version control systems and other development tools this seems like an ideal approach for large and complex projects.

Looking Around MidPoint Installation

Now we have a running midPoint installation and you should have some understanding of how to configure it. But before we plunge into the details about configuration let's have a look at midPoint installation. There are few things that need to be understood before going on. It will save a lot of time later on.

MidPoint needs its own database to work. We call that database *midPoint repository*. The database is used to store the configuration, users, resource definitions, account links, audit trails and a lot of other things. Proper relational database (PostgreSQL, MySQL/MariaDB, Microsoft SQL or Oracle) is strongly recommended for production deployment. But for development and demonstration purposes midPoint contains an embedded database engine (H2 database). This embedded database is initialized by default when midPoint is installed. And it is that embedded database that is used to store the configuration objects right now in your fresh midPoint deployment. This database does not need any special configuration, the database schema is applied automatically and it is started and stopped together with midPoint. Therefore it is ideal for demonstration and development purposes.

Vast majority of midPoint configuration is stored in the database. But there are few things that cannot be stored there. Such as connection parameters to the database itself. For that purpose midPoint has a small configuration file called `config.xml`. MidPoint also needs a place where to store other data that cannot be in the database, such as cryptographic keys, connector binaries and so on.

MidPoint needs a special directory on a filesystem for that purpose. We call it midPoint home directory. New directory with the name `var` is created in directory where midPoint distribution package was installed, i.e. at the same level as the `bin` directory where midPoint start scripts are located. Assuming that midPoint was installed in `/opt/midpoint` directory then midPoint home will be located in `/opt/midpoint/var` directory.

The location of midPoint home directory can be changed by using the `midpoint.home` Java system property. This is done by specifying `-Dmidpoint.home` in the JVM command-line. Or in case that the default midPoint start scripts are used `MIDPOINT_HOME` environment variable can be used to set the location of midPoint home directory.

When midPoint starts for the first time it starts with an empty database. MidPoint populates the database with a minimal set of configuration objects. This set contains objects such as the `Superuser` role and user `administrator`. These objects get imported automatically because if they are not there you will not be able to log into the new midPoint instance. These objects are imported only if they are not already present in the database. If you modify them later then midPoint will not overwrite them.

Logging

Logging is perhaps the most important mechanism to diagnose any issues with midPoint. Logging should be *the* thing that comes to your mind anytime you cast a puzzled look at midPoint user interface. We try to make midPoint user interface convenient to use and we pay a lot of attention to good error reporting. But there are always some limits. The error that the user interface displays

may be just a result of a long chain of causes and effects. Error messages in the user interface may not directly point to the primary cause. Or maybe there is no error at all, just midPoint does not do what it is supposed to do. That is the point where logging comes to the rescue.

MidPoint is using Java logging facilities to log its messages. MidPoint log file name is `midpoint.log` and it is stored in the `log` subdirectory in midPoint home directory (`/opt/midpoint/var/log/midpoint.log`). The default logging level is set up more-or-less to suit normal midPoint operation. This means that the messages on level `INFO` and above are logged while the finer levels are not logged. If you want to diagnose midPoint issues you will need to switch the logging levels to `DEBUG`, or in extreme cases even to `TRACE`. The logging levels can be adjusted in midPoint user interface. Navigate to *Configuration > System > Logging*.

MidPoint is not a simple system. There are complex interactions, there is usually a lot of custom configuration, customizations, expressions and so on. Diagnostics of midPoint issues is in itself no easy task. Therefore there is a dedicated [Troubleshooting](#) chapter in this book.

Chapter 4. Resources and Mappings

The pessimist complains about the wind; the optimist expects it to change; the realist adjusts the sails.

— William Arthur Ward

Reading and writing resource objects, attribute synchronization, mapping of attribute values, their transformation using scripts – these are the basic midPoint features. These features are absolutely essential for any self-respecting IDM deployment and all IDM engineers should be more than familiar with them. And this is exactly the purpose of this chapter: describe necessary configuration to use midPoint as a provisioning engine.

It is not very realistic to expect that all the systems will agree on the same interface, communication protocol and schema for identity management. There were several attempts to unify the IAM landscape, but none of them was entirely successful. The LDAP protocol was created in the 1990s. But even for such a mature protocol the LDAP implementations are still not 100% interoperable. The situation is even worse for identity provisioning protocols. There were several attempts to specify a standard provisioning protocol, but all of them failed to deliver complete interoperability. The worst pain point of identity integration is undoubtedly the schema. Every application has its own data model for representation of accounts, groups, privileges and other identity-related objects. Even if the application tries to expose that data model using some kind of standard schema there will always be small (but important) differences. MidPoint provides a practical solution to this problem. Application interfaces and their schemas need to be aligned or *mapped* to a common identity schema that you choose to use for your deployment. This chapter will tell you how to do it.

Resource Definitions

Resource is one of the most important concepts in midPoint. Any system connected to midPoint is a *resource*. Resources are typically target systems where midPoint manages accounts. But source systems such as HR databases are also considered to be resources. There is no strict distinction between source and target resource in midPoint. Both source and target resources are defined in exactly the same way. Resource can even act as both source and target at the same time.

MidPoint needs a way how to communicate with the resource. MidPoint has to know communication protocol, hostname, passwords, etc. For that purpose midPoint has *resource definition objects*. These are ordinary midPoint configuration objects stored in midPoint repository. Resource definition usually contains:

- **Name** of the resource and its description
- **Reference to the connector** which is used to communicate with the resource
- **Connector configuration properties** that define resource host name, port, communication settings and so on. Those properties are used to initialize the connector.
- Definition of **object types** that are interesting for midPoint. This is typically a definition that describes how a typical account looks like. But there may be much more: groups, roles,

organizational units, ...

- Object type definitions typically contain **mappings**. Mappings define how attributes are synchronized from midPoint to resource or from resource to midPoint.
- **Synchronization** settings that define what midPoint should do if it discovers unknown account, if the account is deleted on the resource and so on.

Resource definition looks like this in its XML form (simplified):

```
<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>
```

Resource definition is a very rich (and powerful) configuration object. It is maybe the richest configuration object in the entire midPoint system. Creating resource definition from scratch is usually no easy task. There is a lot of things to consider: connector configuration, identifier conventions, mandatory attributes, attribute value formats and so on. What we usually do is to locate a resource definition sample for a similar resource. Then we modify the sample to suit our needs. However, you need to understand how the resource definitions work to do this efficiently. Next few sections will explain the structure and function of resource definitions.

Believe it or not, there are people that do not like XML/JSON/YAML. There are also people that really want to start creating the resource from scratch. For all those people there is a resource wizard in the midPoint user interface. The wizard can be used to create and edit resource using a graphical user interface.

Configuration Connector pool Timeouts Results handlers

Configuration

Host * <small>i</small>	localhost
Port number <small>i</small>	389
Connection security <small>i</small>	
SSL protocol <small>i</small>	

However, even if resource wizard is your preferred way, it may be still easier to start with an existing sample. Find the sample that is the best match for your situation, import it in midPoint and then use the wizard to modify it.

There are many resource samples to start from. Most of them are located in midPoint distribution package. But there are other places to look for samples. Please see [Additional Information](#) chapter for suggestions.

Connectors

Every resource needs a connector to work. Connectors are small pieces of Java code that are used to communicate with the resource. MidPoint looks for available connectors when it starts up. MidPoint will automatically create new configuration object for each connector that it discovers during startup. The list of discovered connectors can be seen in midPoint user interface in *Configuration > Repository objects > Connector*. The connector objects look like this:

```
<connector oid="028159cc-f976-457f-be70-9e9fa079bcf7">
    <name>ConnId com.evolveum.polygon.connector.ldap.LdapConnector v2.0</name>
    <framework>http://midpoint.evolveum.com/xml/ns/public/connector/icf-1</framework>
    <connectorType>com.evolveum.polygon.connector.ldap.LdapConnector</connectorType>
    <connectorVersion>2.0</connectorVersion>
    <connectorBundle>com.evolveum.polygon.connector-ldap</connectorBundle>
    <namespace>http://midpoint.evolveum.com/xml/ns/…</namespace>
    <schema>
        ...
    </schema>
</connector>
```

The resource definition needs to point to the appropriate connector object. Therefore select the right connector from the connector list and remember its OID. Then use the connector OID in the resource configuration like this:

```

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

```

This is a straightforward way how to link connector and resource. However, it is not the most convenient one. MidPoint creates connector objects automatically. Therefore the OIDs of the connector objects are not fixed. Every midPoint instance will have different OID for the discovered connectors. Therefore if we want a resource that is always using the LDAP connector in all the midPoint instances we cannot do that by just using OIDs. But there is another way. You can use search filter instead of fixed OID:

```

<resource>
    <name>My LDAP Server</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>

```

The detailed explanation of the search filters will come later. For now it is important to know just few basic principles. When this resource definition is imported, midPoint notices that there is no OID in the `connectorRef` reference. It also notices that there is a search filter. Therefore midPoint executes that search filter. In this case it looks for object of `ConnectorType` type that has property `connectorType` with value `com.evolveum.polygon.connector.ldap.LdapConnector`. Therefore midPoint finds LDAP connector regardless of the OID that was generated when midPoint discovered that connector. Then midPoint takes the OID of the object that it has found. The OID is placed to the `connectorRef` reference, so midPoint can find the connector directly and it does not need to execute the search every time the resource is used.

This is the method that is frequently used to bind resource definition to a specific connector type. It has the advantage that it works in all midPoint deployments. Therefore it is also used in configuration the samples.

Bundled and Deployed Connectors

Each class of resources needs its own connector. There is an LDAP connector that supports all the common LDAP servers. There are connectors that work with generic database tables. These connectors are quite generic. But most connectors are built for a specific application or software system: Linux servers, SAP R/3, Siebel, etc.

There is a handful of connectors that are so generic that they are used in almost all midPoint deployments. These connectors are bundled with midPoint. That means that they are part of the midPoint application package and they are always available. These three connector bundles are part of midPoint:

- LDAP Connector bundle, which contains:
 - LDAP connector that works with most LDAPv3-compliant servers.
 - Active Directory connector that can work with Microsoft Active Directory over LDAP protocol.
- DatabaseTable connector bundle with a connector that can connect to a generic relational database table.
- CSV connector bundle with a connector that works with comma-separated (CSV) text files.

These connectors are always available in midPoint. Other connectors must be deployed into midPoint. Connector deployment is a very straightforward process:

1. Locate the connector binary (JAR file).
2. Copy the binary into the **icf-connectors** directory which is located in midPoint home directory.
3. Restart midPoint

MidPoint will scan the **icf-connectors** directory when it starts up. It will discover any new connectors and create a connector configuration objects for them.

Connector Configuration Properties

Connector need a configuration to be able to work with resource. This configuration usually consists of connection parameters such as hostname, port, administrative username, password, connection security settings and so on. The connector configuration properties are specified in resource definition object. In a simplified form it looks like this:

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
    <name>My LDAP Server</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>
```

There may be a very broad range of configuration properties - and every connector has its own set. While working just with the text representation of the resource definition you will need to find out

the names of the configuration properties by looking at the samples, connector documentation or maybe even connector source code. It may look difficult but this is perfectly viable approach. However, there are other ways. Firstly there is the resource wizard. The wizard knows all the connector configuration properties and it will present the properties in a configuration form. The wizard takes the definition of the configuration properties from *connector schema*. Connector schema is a definition of the properties that the connector supports: their names, types, multiplicity and so on. The connector schema is stored in the connector configuration object under the schema tag. Therefore even if you are working only with the XML/JSON/YAML files you can have a look at that schema to figure out what connector configuration properties are supported.

The connector schema also defines connector namespace. Generally speaking namespaces in midPoint are used to isolate schema extensions that might conflict and they are also used for data model versioning. The use of namespaces is optional in almost all parts of midPoint - but not yet in all the parts. Connector configuration is one of the few parts where namespaces should still be used. And it also makes some sense, as namespaces are used here as an additional safety mechanism. To keep a long story short, the configuration properties should be properly namespace-qualified:

```
<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>
```

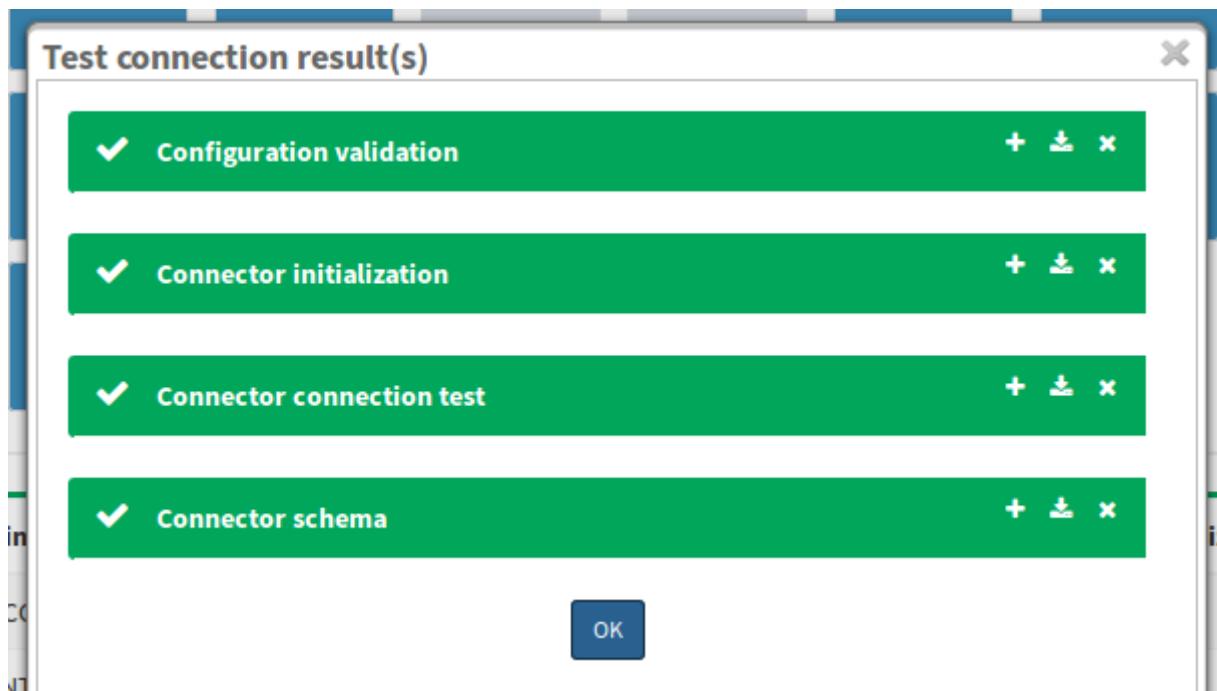
The use of namespaces will be completely optional in later midPoint versions. For now just copy the namespace URIs from the samples. You do not have to completely understand what is going on. Just one thing: the namespace of the configuration properties should be the same as the namespace defined in the connector object. This is a long URI that is composed of connector bundle name and connector name.

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

If the namespace does not match then the connector will refuse to work. This is a safety mechanism that prohibits accidental use of configuration from one connector in another connector where the configuration properties may have the same name but a completely different meaning.

Testing the Resource

Minimal resource definition has just the name, connector reference and connector configuration properties. After that the resource should show the first signs of life. Therefore select a suitable sample file now. Strip it down to the minimum, modify connector configuration properties and import the resource into midPoint. You should be able to see your resource in the list in *Resources > List resources*. The icon next to your resource is most likely black - not green and not red. Green icon means that the resource is working, red icon means that there is an error, black means "I do not know yet". Click on the resource label. Resource details page should appear. There is a *Test Connection* button at the bottom of the page. Click on that button. It may take a while now. MidPoint is initializing the connector with the configuration properties that you have specified. Then the connector will be used to check connection to the resource. If the parameters were correct and midPoint can reach the resource you will see green lights:



If there are any errors during connector initialization, configuration or network connection you will see the errors here. In that case correct the configuration properties and try again. If everything works well then the resource icon turns green. Now we have a minimal working resource.

There are few more things that you can do with such a minimal resource. For example you can look at the resource content. Navigate to the resource details page and switch to *Uncategorized* tab. Select one of the object classes that the resource supports. Just click inside the *Object class* input box and the suggestions will appear. Now click on the *Resource* button on the right side. MidPoint connects to the resource, lists all the objects of the given object class and displays the list. Now you can click on any object to see the details.

	Name	Identifiers	Situation	Intent	Owner	Result	More...	Advanced
<input type="checkbox"/>	uid=001,ou=people,dc=example,dc=com	entryUUID: 14dc6610-0d50-1036-9b1a-d35c135ff39d dn: uid=001,ou=people,dc=example,dc=com						
<input type="checkbox"/>	uid=002,ou=people,dc=example,dc=com	entryUUID: 3dec0610-105b-1036-8988-e5a556d21e19 dn: uid=002,ou=people,dc=example,dc=com						

That is a very useful feature for several reasons. Firstly, you can check that not just the resource connection works, but that the connector can actually retrieve the objects. Secondly, you will get some idea about the object classes that the resource supports. And thirdly, by looking at several objects you can get basic overview of how the data are structured: what attributes are used and what are the typical values. You will appreciate that information later on when we will be setting up mappings.

Resource Schema Basics

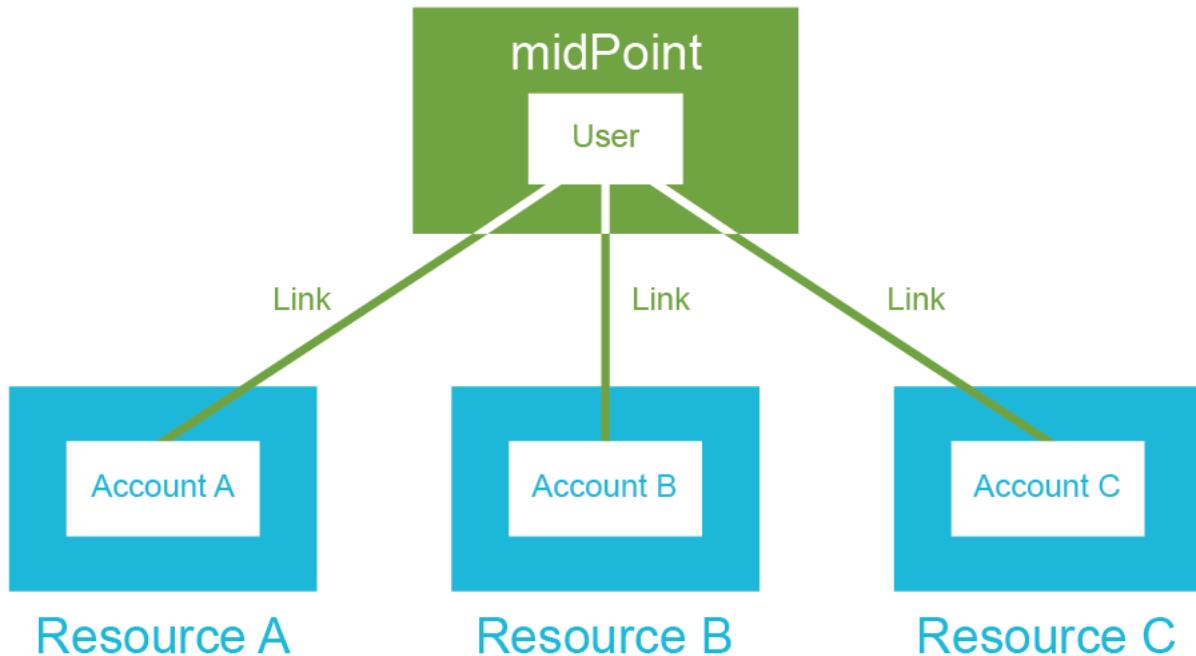
The only resource object that early identity management systems dealt with was an *account*. That is not sufficient any more. Good identity management system needs to manage many different types of resource objects: accounts, groups, organizational units, privileges, roles, access control lists and so on. In midPoint these are the *object classes*: types of resource objects that are made accessible to midPoint by the connector. A minimal resource supports at least the *account* object class, but a typical resource supports more object classes. Each object class may have a completely different set of attributes: different names, different types, some may be mandatory, some optional.

The collective definition of the object classes and their attributes is what we call *resource schema*. Obviously, resource schema is different for every resource. Even resources that are using the same connector may have different resource schema (e.g. two LDAP servers with different custom schema extensions). MidPoint is a smart system and it is capable of automatic resource schema discovery. MidPoint will reach out to the resource and retrieve the schema when the resource is used for the first time. Retrieved resource schema is stored under the schema tag in resource definition object. You can have a look and examine the schema there. But beware, the schema may be quite rich and big.

Resource schema is absolutely crucial concept. MidPoint takes advantage of resource schema whenever it needs to work with resource objects such as accounts or groups. MidPoint uses resource schema to validate mappings. The schema is used for automatic type conversions. And most importantly of all: resource schema is used to display resource objects in user interface. MidPoint adapts to resource schema automatically. Not a single line of custom code is needed to do that.

Hub and Spoke

MidPoint topology is a star (a.k.a. "hub and spoke") with midPoint at the center. This is both physical and logical topology of midPoint deployments.



This means that the *account A* can be synchronized with midPoint user and then midPoint user can be synchronized with *account B*. But *account A* cannot be synchronized directly to *account B*. This is deliberate decision that was made very early in midPoint design and we have very good reasons for it.

Accounts and user that represent the same person are *linked* together. This *link* is a relation that midPoint creates and maintains. Therefore midPoint knows who is the owner of a particular account. MidPoint also knows which accounts the user has. That is how midPoint knows which account needs to be synchronized with which user. It is critical for the links to be correct otherwise midPoint cannot reliably synchronize the data. Therefore midPoint takes great care to maintain the links. And that is not always an easy task. There are strange corner cases such as renamed accounts or accounts that were deleted by mistake and re-created. But midPoint is built to handle such cases. The links are always maintained. And it is the link that allows midPoint to list all user's accounts in the user interface.

The screenshot shows the midPoint user interface for managing user accounts. At the top, there's a red header bar with a user icon and the identifier '(002)'. To the right of the header are two checkboxes: one checked for 'Enabled' and one unchecked for 'No organizations'. Below the header, a navigation bar includes tabs for 'Basic', 'Projections' (selected, with a count of 3), 'Assignments' (1), 'History', 'Cases' (0), 'Personas' (0), 'Delegations' (0), and 'Delegated to me' (0). A search bar with a magnifying glass icon and an 'Advanced' button are also present.

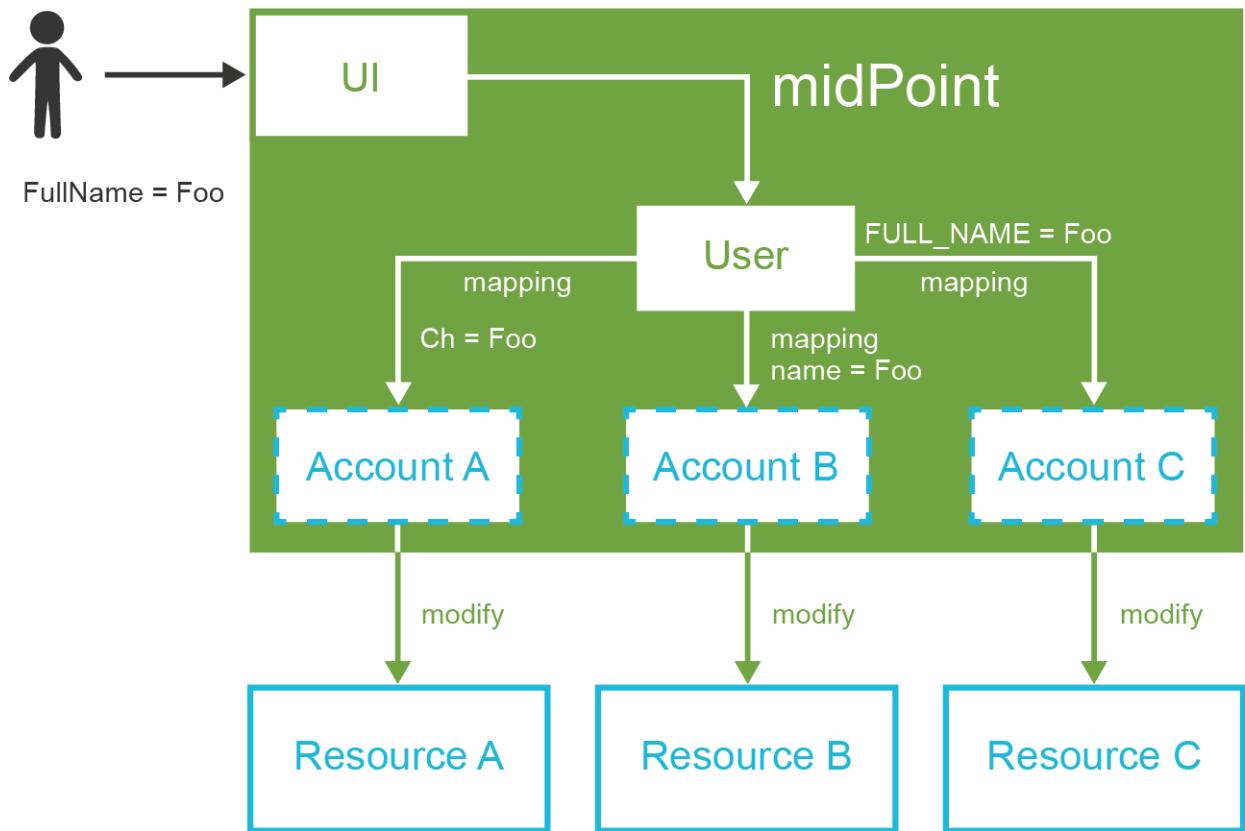
	Name <i>i</i>	Resource <i>i</i>	Object Class <i>i</i>	Kind <i>i</i>	Intent <i>i</i>	Pending operation <i>i</i>	Actions
<input type="checkbox"/>	002	HR System	AccountObjectClass	ACCOUNT	default		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	uid=002,ou=people,dc=example,dc=com	LDAP	inetOrgPerson	ACCOUNT	default		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/>	002	Portal	AccountObjectClass	ACCOUNT	default		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>

At the bottom of the table area, there's a green '+' button for adding new entries. Below the table, a pagination control shows '1 to 3 of 3' and navigation icons (<<, <, >, >>). On the far right, there's a gear icon for settings.

The user in midPoint is known as *focus* in midPoint terminology. The accounts are known as *projections*. You can imagine a light projector that sends many light beams from its focal point to create a projection on the screen. This is the metaphor that we have chosen when building midPoint. And for the lack of better words this terminology remains in use even now. We will get back to the concept of focus and projections many times in this book. For now you just need to remember that *projection* means an *account*.

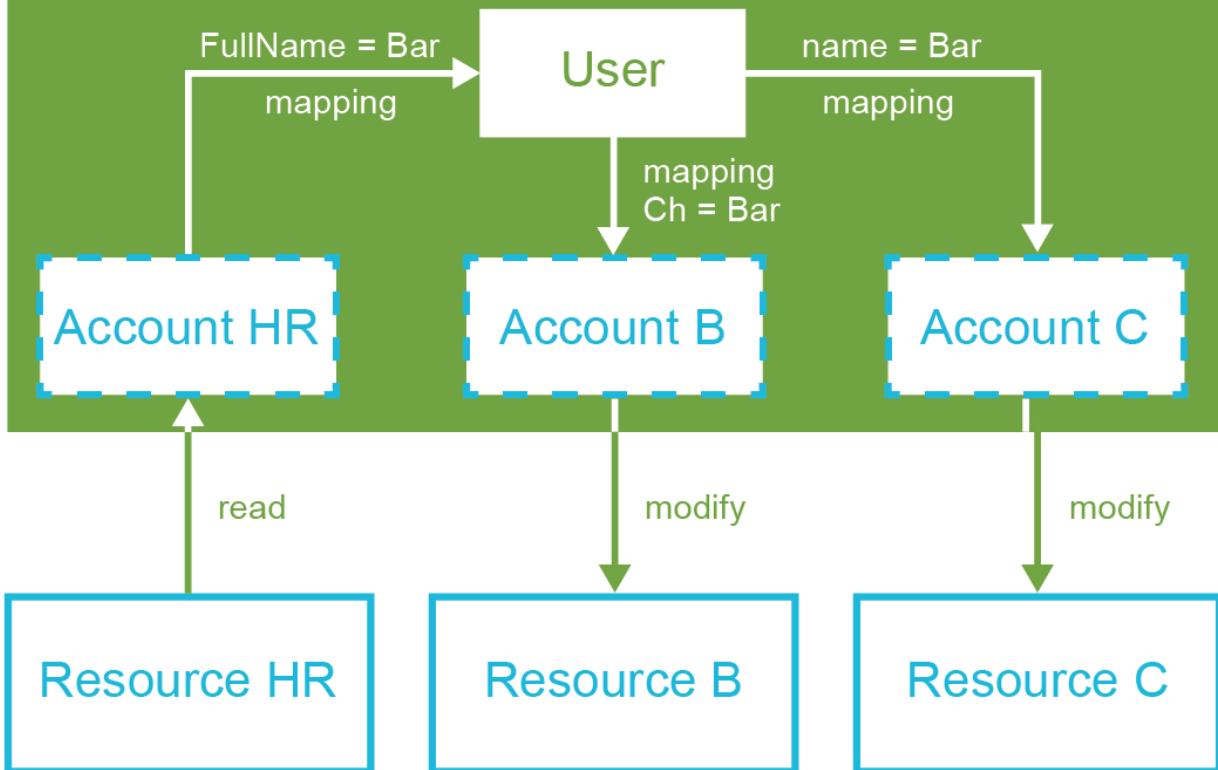
MidPoint knows which account belongs to which user by following links that it maintains. But how does midPoint know which attributes to synchronize? How to transform the values? And which side is the authoritative one? Mappings take care of that. Mapping is like a flexible data replication recipe. MidPoint allows to define mappings for each attribute in any direction. The mappings are used to control the synchronization on a very fine granularity.

Perhaps the best way to summarize synchronization principles is to illustrate them using a couple of examples. The first example is a modification of user properties in midPoint user interface. When the Save button is pressed then midPoint user interface sends the modification to midPoint core engine. The synchronization code in midPoint core follows the links to find all the accounts that belong to this specific user. Then the mappings are applied to synchronize the changed user properties to the accounts. Account changes are propagated to the resources and user changes are stored in midPoint repository.



The second example is slightly different. This case starts with a change of account data. This may be a change of an employee record in HR system. MidPoint detects that change and reads the changed account. MidPoint follows the link to find the user to which the account belongs. Then it follows other links from the user to find all the other accounts that may be affected. Similarly to the previous case the mappings are applied. The mappings from the HR account to the user are applied first. The result is a modification of user properties. Then a process identical to the previous case takes place. User modifications are automatically applied to all affected accounts.

midPoint



Those two cases look to be quite different. First case is a manual change of data by system administrator. Second case is an automatic data feed from the HR system. But as you can see the principles that are used to implement those two cases are almost exactly the same. This is the consequence of midPoint philosophy: radical reuse of functionality and generic application of principles. You just need to define what you want to do (the policy). MidPoint takes care that it is done when it needs to be done.



Why the star topology?

The star or "hub and spoke" were (and still are) the big buzzwords of system integration. And the basic idea makes a lot of sense. If every node needs to be synchronized with every other node then the number of required connections grows quite steeply. It is in fact proportional to the square of the number of nodes. Mathematicians say that is has $O(n^2)$ complexity. However, if you rearrange the connections so that they all point to the central "hub" then the number of connections is significantly reduced. It is proportional to the number of nodes: $O(n)$ complexity. This is a huge difference, especially in deployments with many resources. However, this approach works well only if the star topology is both physical and logical. I.e. it makes very little sense to connect all resources to a central "hub" if that hub still internally needs $O(n^2)$ policies to synchronize the data. That would only hide the complexity in a black box, but the complexity is still there. However, midPoint is different. MidPoint is a real "hub". This is the reason why midPoint does not support synchronization of accounts directly with each other. We want to have simple, clean and maintainable system, both externally and internally.

Schema Handling

Resource schema is a very important concept. It defines what object classes are supported by the resource and how they look like. But it is not important to know only how the objects look like. It is also important to know what to do with them. And that is what the *schema handling* is all about.

Schema handling is a part of resource definition object. It specifies which object classes from the resource schema are actually used by midPoint. And most importantly of all it specifies how they are used. This is the place where mappings are stored. This is the place where account-group associations are defined. This is the place where schema can be augmented and tweaked. Simply speaking, this is the place where most of the resource-related configuration takes place.

Schema handling section contains definition of several *object types*. Each *object type* refers to one "thing" that midPoint works with: default account, testing account, group, organizational unit and so on. Let's start with something simple and let's define just one object type now: default account. It looks like this:

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

This may seem trivial, but even such a minimal definition is important for midPoint. This definition tells midPoint that default account on this resource has `inetOrgPerson` object class. Resources such as LDAP servers may have dozens of object classes. Most of them are not used at all. There are often several alternative object classes that can be used to create accounts. It is important to tell midPoint which object class is the right one. And that's what this definition does. Once this definition is in place, the accounts appear on the *Accounts* tab of the resource details page (they were visible only on the *Generics* tab before). This is a sign that the definition works correctly.

A clever reader surely noticed definition of `kind` in the above example. Setting `kind` to `account` indicates that this object type definition represents (surprisingly) an account. MidPoint supports many types of objects. But two types have a special place: `accounts` that represents the users and `entitlements` that give privileges to accounts. MidPoint can handle the objects in a smart way if it knows that it is either account or entitlement. And the `kind` definition tells just that. There is also optional `intent` setting that can be used to define subtypes. But more on that later.

The schema handling section can also be used to augment (or even override) some parts of the resource schema. E.g. following example sets a display name for this object type. The display name will be used by the user interface when it displays the account.

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

However, the most powerful feature that is used in the schema handling is the ability to deal with attributes. Following sections are all about that.

Attribute Handling

Resource objects such as accounts or groups are mostly just a bunch of attributes. Almost all of the IDM magic is about setting the correct attribute to the correct value. The schema handling section of the resource definition is the place where the basic attribute behavior is defined.

The object type definition contains sections that define behavior of each attribute that we care about:

```

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

```

There is an **attribute** element for every attribute that we need. Lot of details can be defined here: display name of the attribute that will be used by the user interface, limitations and schema augmentation, override settings and so on. But the most important things that go there are the mappings. In the simplest form a mapping looks like this:

```

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

```

This means that the value of the `cn` attribute will be taken from the `fullName` property of the focal object (which is typically a user). This is a very simple mapping, there is no value transformation, no condition – nothing complicated at all. This is how a lot of mappings look like. But mappings can also be very powerful and complex. That will be described in next section.

The `attribute` sections are used to set up the attributes that a typical user account on the resource has. Those will assign identifiers, set up full name, set description and telephone number attributes and things like that. It is a very convenient approach to have this directly in the resource definition. We can simply assign the accounts to the user without specifying any details. MidPoint evaluates the mappings in `attribute` sections to populate account attributes with the correct values. Now it is perhaps a good time to have a look at some sample resource definitions to get a feel how a real-world resource definition looks like. The samples are located in the midPoint distribution package or you can find them on-line. See [Additional Information](#) chapter for more details.



The "ri" namespace.

You may have noticed that "ri" namespace prefix is used whenever we refer to the object classes or attributes. In a strict sense this is the correct notation. Object classes and attributes are defined in resource schema and the "ri" is the namespace of that schema. While the use of namespaces should be optional in almost all parts of midPoint, we are still using the "ri" namespace in samples. Mostly due to the nostalgic reasons. By the way, "ri" stands for "resource instance".

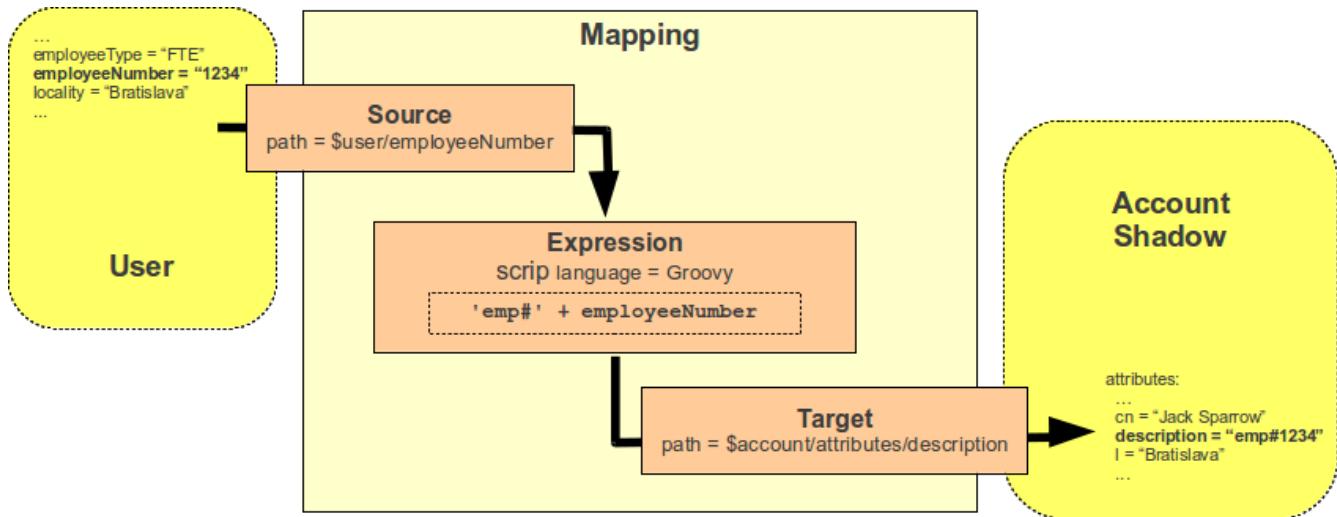
Mappings

Mapping is a very flexible mechanism that takes one or more input properties, transforms them and puts the result in another property. Mappings are used all over midPoint. But perhaps the most important use of mappings is in the schema handling part of the resource definition where they set up account attribute values. We have already seen a very simple mapping that simply copies the values from one place to another. Now it is the time to look at mapping in its entirety.

Mapping consists of the three basic parts:

- **Source** part defines the data sources of the mapping. These are usually understood as mapping input variables. Source defines where mapping gets its data from.
- **Expression** part defines how the data are transformed, generated or passed on to the "other side". This is the most flexible part of the mapping as it contains the logic. There is a broad variety of possibilities, including support for scripting expressions.
- **Target** part defines what to do with the results of the mapping, where the computed values should go.

The three parts of the mapping as well as the basic principle is illustrated in the following diagram:



The diagram shows a mapping that takes `employeeNumber` user property and transforms it to `description` account attribute by using a simple Groovy script expression.

The `source` part of the mapping defines that there is a single source which is based on `employeeNumber` user property. Source definitions are important for the mapping to correctly process relative changes (deltas), mapping dependencies, etc. The source definition tells mapping that the value of `employeeNumber` user property should be passed to an expression.

The `expression` part contains a simple Groovy script that prepends the prefix `emp#` to the employee number value specified by the source definition. The `expression` part of the mapping is very flexible and there is a lot of ways that can be used to transform a value, generate new value, use a fixed value, pass a value without any change and so on.

The `target` part defines how the result of the expression should be used. In this case the result is to be used as a `description` account attribute. The `target` definition is necessary so the mapping can locate appropriate definition of the target property and therefore make sure that the expression produces a correct data type and that other schema constraints are maintained (e.g. single vs multiple values).

This mapping can be expressed in XML:

```

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

```

Not all parts of the mapping are mandatory. If the expression is not present then "as is" expression

is assumed. Such expression simply copies the source to target without any transformation. Some parts of the mapping may be implicitly defined by the surrounding context. E.g. target or source is implicit if the mapping is used to define attribute behavior in the schema handling section. Therefore it is usually sufficient to define either source or target for mappings in **schemaHandling**:

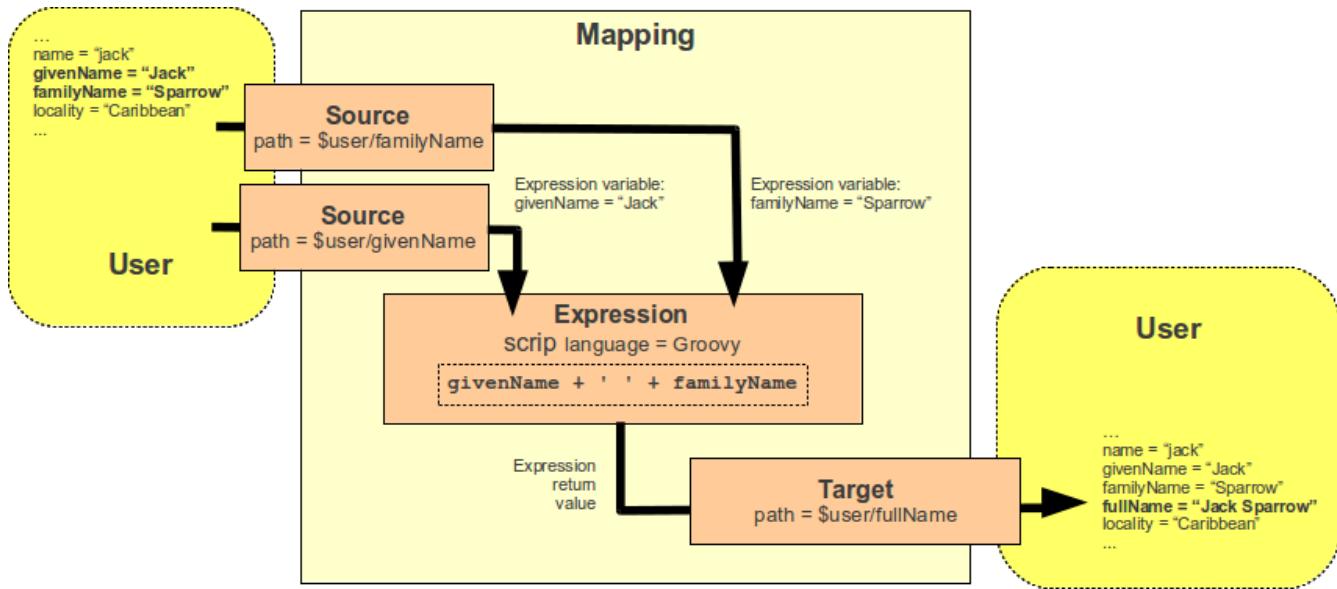
```
<schemaHandling>
...
<attribute>
    <ref>ri:sn</ref>
    <outbound>
        <source>
            <path>$focus/familyName</path>
        </source>
    </outbound>
</attribute>
...
</schemaHandling>
```

This is the notation that you have seen in the previous section. Mapping source is explicitly specified as the **familyName** property of the user. Mapping target is implicitly set to be the attribute for which the mapping is defined. As no expression is explicitly defined it defaults to a simple copy of the value without any transformation (**asIs**).

Mapping notation can even be shortened a bit more in this case. It is quite clear that the mapping source will be one of the properties of the focal object (user). Therefore the **\$focus** prefix can be omitted:

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

Those examples are still very simple. Mappings can do much more – as you will learn later on. But there is one more thing that we need to explain here. Mappings are designed to work with more than just a single source. Following diagram illustrates a mapping that takes two arguments: given name and family name. The mapping produces full name by concatenating these values with a space in between. This is the kind of mapping that is frequently used to construct user's full name from its components. While the mapping may seem simple there are some sophisticated mechanisms hidden inside.



The mapping is represented in the XML form as follows:

```

<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>

```

There are two sources specified by the source definitions: user property `givenName` and another user property `familyName`. The mapping is using *script expression* to combine the values into a single value which is used to populate user's `fullName` property.

This example also illustrates that the mappings are smart. The mapping may be evaluated only if one of the sources changes or if a full recompute is requested. In case that neither `givenName` nor `familyName` changes there is no need to re-evaluate that expression. This is one of the reasons for requiring explicit source definition in the mappings. Without such definitions it is not (realistically) possible to reliably determine when and how the expression should be re-evaluated.



`$user` and `$account` variables.

Variables `$focus` and `$projection` were introduced in midPoint 3.0 as a consequence of the generic synchronization feature. The objects that the expression works with might not be just user or account. A much broader range of objects may be used. Therefore generic concepts of focus and projections were introduced and the variable names were changed to reflect that. The old variables `$user` and `$account` can still be used, but their use is deprecated. Despite that they are still used in some older examples. It is never easy to completely eliminate the burden of history, is it?

Mappings are used all over midPoint, in many places and situations. Sometimes a mapping needs to be really authoritative. It has to enforce the value to the target. But sometimes we want to provide a default value and the mapping should never change the target value once it is set. Therefore mapping can be set to various levels of *strength*: from weak to strong. Following table describes how that works:

Strength	Description
<code>weak</code>	Mapping is applied only if the target has no value. Weak mappings are usually used to set <i>default values</i> .
<code>normal</code>	Mapping is applied only if there is a change in source properties. Normal-strength mappings are used to implement the <i>last change wins</i> strategy. If the value was modified in midPoint then the mapping is applied and target is modified. If the target is modified directly then the mapping does not overwrite the target value – until the next change in midPoint. This is the default behavior of mappings. If no strength is specified then <code>normal</code> is assumed.
<code>strong</code>	Mapping is always applied. Strong mappings <i>enforce</i> particular values.

The strength can be specified in any mapping by using the `strength` tag:

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

When it comes to mapping strength then the following rule of the thumb may be useful: If you want

to enforce policy use *strong* mappings. If you just want to set a default value use *weak* mapping. If you are not sure what you are doing then *normal* mappings will probably work just fine.

Expressions

Expression is the most flexible part of the mapping. There are approximately dozen different type of expressions ranging from the simplest *as is* expression through the *scripting expressions* all the way to a special purpose expressions that search through midPoint repository. Expression type is determined by the element that is used inside the **expression** section of the mapping. We refer to those elements as *expression evaluators*. You can find detailed description of expression evaluators in midPoint Wiki. We are going to deal only with few types that are most frequently used now:

Expression Evaluator	Element	Description
As is	asIs	Copies the value without any transformation.
Literal	value	Stores literal (constant) value in the target.
Generate	generate	Generates a random value.
Script	script	Executes a script, stores script output in the target.

The simplest expression evaluator is **asIs**. It simply takes the source and copies that to the target. It obviously works only if there is just one source. It is also the default expression evaluator. If no expression is specified in the mapping then **asIs** is assumed. It is used like this:

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

Literal expression evaluator is used to place a constant value in the target. This expression does not need any source at all. It always produces the same value. It looks like this:

```

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

```

The **generate** expression evaluator is used to generate a random value. As such it is used almost exclusively to generate passwords. We will deal with that expression later when we will be dealing with credentials.

Script Expressions

The most interesting expression evaluator is undoubtedly the **script** expression evaluator. It allows to execute arbitrary scripting code to transform the value. Basic principle is simple: values from source properties are stored in the script variables. Script is executed and it produces an output. The output is stored in the target.

We have already seen a mapping that has a scripting expression:

```

<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>

```

There are two sources: **givenName** and **familyName**. The values of these user properties are placed in variables used by the script. The variables that have the same names: **givenName** and **familyName**. Then the script may do whatever it does with the variables. At the end the script has to return a value. The script above is written in Groovy, therefore the return value is the value of the last evaluated expression. In this case it is the only expression in the script which concatenates the two variables with a space in between. Script return value is placed in the output, which in this case is **fullName** user property.

Scripts are often used to transform the values before they are stored in account attributes. One very common case is construction of LDAP distinguished name (DN). The DN is a complex value in the form of `uid=foobar,ou=people,dc=example,dc=com`. However it is easy to construct such value using a simple script:

```
<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>
```

(A clever reader surely has a suspicious look on his face now. Of course, this is not entirely correct way how to compose LDAP DN. But please bear with us. We will correct that later.)

Midpoint supports three scripting languages:

- **Groovy**: This is the default scripting language.
- **JavaScript (ECMAScript)**
- **Python**

All three languages can be arbitrarily mixed even in a single midPoint deployment. Although quite understandably such a practice is not recommended. The language can be selected for each individual expression by using language URI:

```
<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>
```

When writing scripting expression, please keep in mind that some characters must be properly escaped in the text format that you are using (XML, JSON or YAML). E.g. the ampersand character (

8) so frequently used for logical operations needs to be escaped as & in XML.

Scripting expressions can do almost anything. And there is still more to them that meets the eye. This section provides only the very basic description to get you started. Will get back to the scripting expressions many times in this book.

Activation

The term *activation* is used in midPoint to denote a set of properties that describe whether an object is active. This includes properties that describe whether the user is enabled, disabled, archived, since when he should be enabled, to what date he should be active and so on. The simple enabled/disabled flag might have been sufficient in the 1990s. But that was a long time ago. We need much more than that. Therefore the *activation* is quite a rich data structure in midPoint. We are going to describe just the basic idea now, the details will follow later.

The most important activation concept is *administrative status*. Administrative status defines "administrative state" of the object (user). I.e. the explicit decision of an administrator whether the user is enabled or disabled. Except for administrative status there are also validity times, lockout status, various timestamps and metadata. But we will get to that later.

The important thing to realize is that both user and the accounts have activation properties - and they are almost the same. The user and account activation are using the same property names, meaning and data formats. This is important, because you would probably want account activation to follow user activation. E.g. if user is disabled then also all his accounts should be disabled. This is very easy to do in midPoint because the user and account activation are compatible. Therefore all it takes is a very simple mapping. There is a special place in the resource schema handling section for that:

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
    <name>My LDAP Server</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>
```

It is as simple as that. Just an empty mapping represented by empty `outbound` element. User has `administrativeStatus` property, account has `administrativeStatus` property, therefore midPoint

knows what is the source and target of the mapping. The values of the `administrativeStatus` property has the same type and meaning on both sides. Therefore the default `asIs` mapping is just fine. All that midPoint needs to know is that the mapping exists at all. That we want to map the value. That is a reason for having empty `outbound` element there. MidPoint will fill in all the details.

When this mapping is in place and the user gets disabled, the account will be disabled as well. When the user gets enabled, the account will follow suit.

Credentials

Credential management is important part of identity management. MidPoint is built to easily synchronize credentials to many accounts. Similarly to activation, credential data structures of user and accounts are aligned. Therefore all that is needed to synchronize password to an account is a simple empty mapping:

```
<resource oid="b4101662-7902-11e6-9f14-53e18426fe81">
    <name>My LDAP Server</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>
```

When the user password in midPoint is changed the changed password will be propagated to all the resources that have a mapping like this.

Complete Provisioning Example

This section describes a complete working example of connection to the LDAP directory. The configuration below is used to automatically create accounts in OpenLDAP server. Entire configuration is contained in a single resource definition file. Following paragraphs explain individual parts of the file. Simplified XML notation is used for clarity. Complete file in a form directly usable in midPoint can be found at the same place as all the other samples in this book (see [Additional Information](#) chapter for details).

Resource definition begins with object type, OID, name and description. These are self-explanatory:

```

<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>
    ...

```

Connector reference comes next. We want to point to the LDAP connector. Here we use dynamic reference that is using search filter to locate the connector:

```

    ...
    <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>
    ...

```

The reference is resolved when this object is imported to midPoint. The resolution process takes the search filter and it looks for connector object with the **ConnectorType** specified in the filter.

Connector configuration goes next. This block specifies connector configuration properties such as hostname, port, passwords and so on.

```

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

```

The last part of this block defines that the ConnId framework result handlers should be disabled. ConnId result filtering is a legacy mechanism and most connectors do not need that any more. It may even be harmful in many cases. Unfortunately this mechanism is turned on by default. Therefore most resource configurations contain this part to explicitly turn all the handlers off.

These parts alone should already define a minimal resource. If you define just the name, connector reference and connector configuration you should be able to import the resource to midPoint. The connection test should pass and you should be able to browse resource content. However there is absolutely no IDM logic or automation yet. That is what we are going to add next.

Element that usually follows connector configuration is schema. However if you look at almost any file that contains resource definition you will find no such element. The schema element is automatically generated by midPoint when midPoint connects to the resource for the first time. Therefore there is no need to include it in the definition.

What we have to include in the definition is the way how midPoint handles the schema. This is defined in `schemaHandling` section. Our `schemaHandling` section contains just one `objectType` definition. We are going to define how to handle ordinary user accounts on our OpenLDAP server.

```

...
<schemaHandling>
  <objectType>
    <kind>account</kind>
    <display Name>Normal Account</display Name>
    <default>true</default>
    <objectClass>ri:inetOrgPerson</objectClass>
  ...

```

This is the place where we define the *kind* of objects that we are going to handle. In this case it is **account**. This object is *default* account. Which means that it will be used in case that the account type is not explicitly specified. There is also specification of a display name. Display name is not used in automation logic. It is used by the user interface when referring to this definition. And finally, there is specification of the *object class*. The **inetOrgPerson** object class will be used to create new accounts. The object class specification determines what attributes the account can have.

The **objectType** definition also includes a specification of attribute handling. There is one section for each attribute that we want to handle in automated or special way. It starts with the most important attribute: LDAP distinguished name (DN):

```

...
<attribute>
  <ref>ri:dn</ref>
  <display Name>Distinguished Name</display Name>
  <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>
...

```

The **ref** element specifies name of the attribute that we are going to work with. In fact this is a reference to automatically-generated schema part of resource definition. Definition of display name follows. Display name is used by the user interface as a label for the user interface elements (fields) that work with this attribute. This definition sets a nice "Distinguished Name" label instead of cryptic "dn" which would be used by default.

Let's skip the limitation definition now. We will come back to that later.

The outbound mapping definition follows. This is where the automation logic is specified. This is the place where the DN value is computed. The name property of the user object is the source for this mapping. The name property usually contains username (login name). This value is used by scripting expression in this mapping. The expression is supposed to create a DN in the form:

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

The expression here is a clever one. It does not do the work all by itself. It invokes a library function to compose the DN. It may look like a good idea to use simple string concatenation to construct a DN. However, that will fail in case that the DN components contain certain characters that need to be escaped in the final DN. The `composeDnWithSuffix` library function takes care of that and creates a proper DN.

The outbound mapping will be evaluated whenever we need to construct a DN. This obviously happens when a new object is created. But the same mapping is used when a user is renamed (i.e. his username changes). This is the reason that the mapping needs specification of source. Rename is often quite tricky and complicated operation. It may not be cheap and in some applications it may not be entirely reversible. We definitely do not want to trigger DN changes unless they are really needed. The specification of the mapping source tells us *when* the DN change is needed. In this case it tells us to change the DN in the name property of the user object changes.

Now it is the right time to go back to the [limitations](#) section. The `dn` attribute is defined as mandatory attribute by the schema. And that is perfectly correct: LDAP object cannot be created without a DN. MidPoint is using schema for everything. Therefore when midPoint displays a form to edit this LDAP account it will check that DN has a value because it is a mandatory attribute. However, normally we do not want users to enter the DN in the user interface forms. We want to compute DN automatically. And that is exactly the point of the outbound mapping. Yet midPoint does not know when the expression computes a value and when it does not. The expression is a generic piece of Groovy code. As far as midPoint can see the expression can produce any value, including empty one. Therefore even if there is an expression, midPoint sticks to the schema and it still requires that DN value is entered by the user. However we have written the expression and we know that it will produce a value for any (reasonable) input. Therefore we want to tell midPoint that the DN is no longer mandatory – that it is OK for user to enter no DN value in user interface forms. And that is exactly what the [limitations](#) section does. This section overrides the automatically generated schema and it turns the `dn` attribute from mandatory to optional.

And that is all. Now we have defined the behavior of the `dn` attribute. We can use similar approach to define the behavior of other attributes as well. E.g. the handling of the `cn` attribute has similar definition:

```

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

```

In this case there is outbound mapping, but it has no explicit expression. Which means that the value is taken from the source without any change ("as is"). Therefore the attribute cn will have the same value as user property **fullName**.

It is also possible to define an attribute without any mapping:

```

...
<attribute>
    <ref>ri:entryUUID</ref>
    <displayName>Entry UUID</displayName>
</attribute>
...

```

This means that midPoint will not provide any automatic handling for the **entryUUID** attribute. This definition is used just to set a user-friendly display name for the attribute.

Mappings and expressions have almost unlimited flexibility. E.g. the following definition sets a static value for the **description** attribute:

```

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

```

This mapping has no source because the source does not make any sense for static value. Static

values are always the same. You can also notice that this mapping is *weak*. It will be used to set the **description** attribute only if that attribute does not have any value already. It will not overwrite existing values.

The **inetOrgPerson** object class has much more attributes than those defined in the **schemaHandling** section. Those attributes will be automatically displayed in the user interface. MidPoint will use the generated resource schema to determine their names and types. MidPoint will display these attributes, the user can change them and midPoint will execute those changes. But apart from that midPoint will not do any special handling on those attributes. It is all right not to enumerate all the attributes in **schemaHandling** section. You only need to define those attributes which you want to handle in a special way.

There are two more definitions to describe before our example is complete. First definition is the **activation** definition. It is very simple:

```
...
<activation>
    <administrativeStatus>
        <outbound/>
    </administrativeStatus>
</activation>
...
```

This is a definition that specifies handling of the activation administrative status. This status property specifies whether account is enabled or disabled. Activation properties are somehow special in midPoint. MidPoint understands the meaning and the values of activation properties. MidPoint also expects that user activation and account activation will be usually mapped together. Therefore it is enough to tell midPoint that you want such mapping. MidPoint already knows the source (user activation) and the target (account activation). If the user is disabled then the account will get disabled. If the user is enabled then the account will get enabled.



Clever reader is surely scratching his head now. There is no LDAP standard that specifies how to enable or disable accounts. In addition to this, OpenLDAP does not even have a concept of disabled account at all! Therefore how does midPoint know how to disable an LDAP account? To tell the truth, midPoint does not know it. We have taken a bit of a poetic license here as we wanted to demonstrate a simple activation mapping. But in this case it will not work just by itself. OpenLDAP resource simply does not have this *capability*. However there is a way. Activation capability can be simulated. We will deal with that later. For now let's just marvel in the beauty of this very elegant activation mapping that does absolutely nothing.

The last thing that we need for the resource to work well is to define a mapping for *credentials*. In this case it is a password mapping:

```
...
<credentials>
    <password>
        <outbound/>
    </password>
</credentials>
...
```

Similarly to activation, the credentials are handled in a special way in midPoint. MidPoint understands how credentials work, what are their values and how they are used. MidPoint also expects that user credentials such as passwords will be usually mapped to the account credentials. Therefore all that midPoint needs to know is that you want to do that mapping. It can automatically determine the source and target. The account will have the same password as the user. User's password is used when a new account is created. And when user changes his password the change is also propagated to the account.

That is it. Now you have your first (almost) working resource. You can import the definition to midPoint and test it. Simply assign the resource to a user. The OpenLDAP account will be created - the DN and all the essential attributes will be automatically computed. When midPoint creates an account for a user remembers who is the owner of that account. Therefore it can be easily delete the account when needed. Unassign the resource and the account will get deleted. This is how automated provisioning and deprovisioning works. No real programming is needed. Just a declarative specification and a line or two of very simple scripting. Such configuration can be done in a couple of minutes. And it is essentially the same process for all the applications. Just the connector is different. The connector has different configuration properties, the attribute names are different - but the principles and the tools are the same. It is easy to connect many heterogeneous applications in this way. The connectors and the mappings are hiding the differences. In the end all the "resources" look the same to midPoint. The same principles are used to manage them. Therefore the management can be done efficiently even at a large scale.

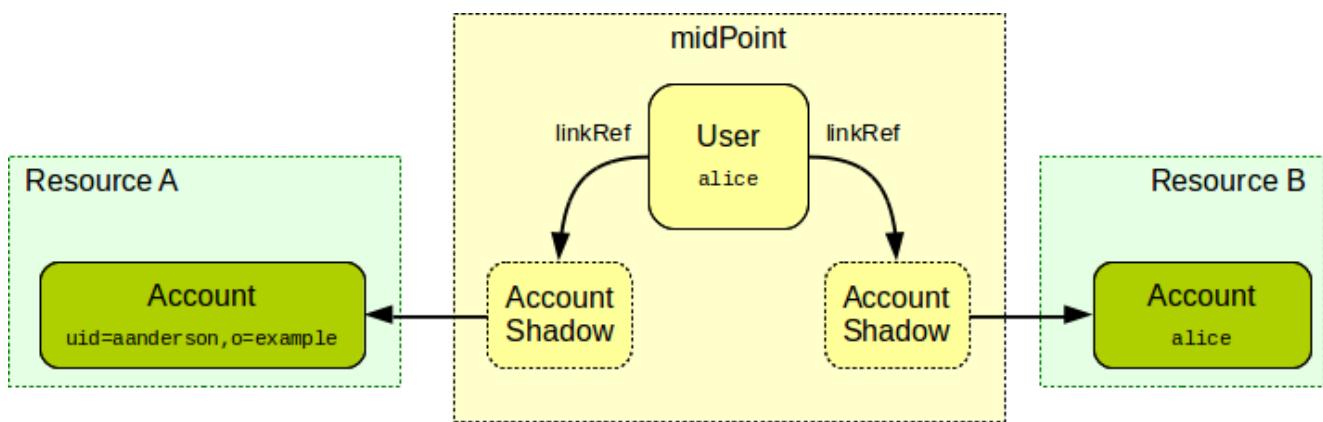
Yet this configuration is still extremely simple. We are just scratching the surface of what midPoint can do. There is much more to see in next chapters. But we need to explain more of the fundamental midPoint concepts before getting there.

Shadows

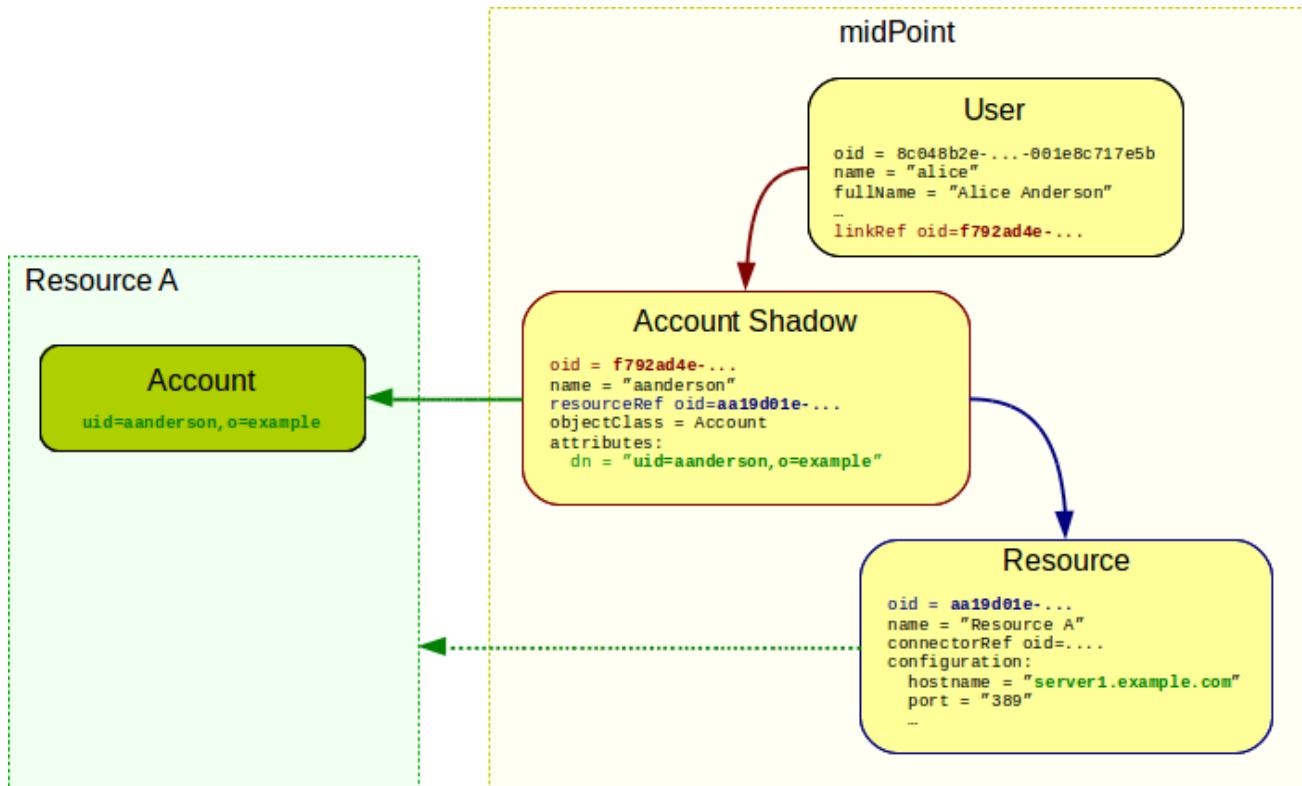
Linking users and accounts is one of the basic principle of any decent identity management system. However it is surprisingly difficult to implement such link. There are numerous methods how to reliably identify accounts and they vary from system to system. Some systems identify accounts only by username - which makes reliable detection of renames quite difficult. Other systems improve on that by introducing another identifier, an identifier that is persistent. Identifier value is assigned by the resource and it never changes. However, username is still used as secondary identifier and it has to be unique. Yet another system has compound identifiers that consist of two or more values. Some system have globally-unique identifiers while other systems have identifiers that are only unique in their own object class. Some systems have hierarchical structured identifiers, others have flat unstructured identifiers. Some identifiers are case-sensitive strings, others are case-insensitive, some identifiers follow complex normalization rules and yet another

identifiers are binary and completely opaque. To make long story short: reliable identification is really complicated.

We do not want to pollute user object with all the delicate details of account identification. Therefore we have created a separate midPoint object that hides all the resource-related details and identification complexities. We call it simply *shadow* because it behaves as a shadow of the real account. When midPoint detects new account a *shadow* is automatically created to represent the account in midPoint repository. When midPoint detects that account has changed (e.g. it was renamed) then midPoint automatically updates the shadow. When the account is deleted midPoint also deletes the shadow. Technically, shadow is still an ordinary midPoint object. Therefore shadow has object identifier (OID). Other objects can simply point to the shadow using ordinary object reference. And that is exactly how user-account links are implemented:



The *shadow* objects contain all the data that are needed to reliably identify an account - or any other resource object such as group or organizational unit. Shadow also points to the corresponding resource definition to make the identification complete. Shadows are multi-purpose objects and they have many uses in midPoint. Shadows record meta-data about resource objects. They are used to hold cached values of the attributes (this functionality is still experimental in midPoint 4.0). Shadows can be used to hold the state of the resource objects for which midPoint does not have on-line communication channel and the operations are executed manually (a.k.a. "manual resources"). Therefore shadows are quite complex objects. Following picture provides more substantial example of a shadow.



Do not worry if that picture looks a bit scary. Shadows may be complex, but they are almost always invisible to midPoint users. Shadows are automatically and transparently maintained by midPoint core. Under normal circumstances, MidPoint takes all that is needed to maintain the shadow and no special configuration is needed for that. We describe the mechanics of the shadow objects here mostly for the sake of completeness. But there may be situations when this knowledge may be useful. These are usually situations when midPoint was mis-configured and the shadows were created incorrectly. In that case you may need to purge all shadows and start over. But beware: shadows are used to link users and accounts therefore if you purge the shadows you will lose the links. Yet, even that is usually not that painful. The synchronization methods described in the next chapter may be used to easily re-create the links.

Chapter 5. Synchronization

It is a capital mistake to theorize before one has data.

— Sherlock Holmes, The Adventures of Sherlock Holmes by Arthur Conan Doyle

Data are the lifeblood of any software system. Ensuring proper management of the data is one the primary responsibilities of a software architect. But data management can be very tricky – as any experienced software architect knows only too well. One of the important principle of software architecture is often formulated as "do not repeat yourself". This applies to code as it applies to data: though shall not repeat the data. There is one original, authoritative value. And there should not be any copies of that value. Ever. There is just one universal source of truth. If there are no copies then the data are always consistent. No copies means no contradictions. Just one truth, precise and crystal-clear. Keep data in one place and one place only.

That is the theory.

However, practice has a different opinion to offer. In practice there are many incompatible technologies. Applications built on relational databases cannot directly use data from directory services. Even relational databases do not fit together easily. Each application is designed with a different data model in mind. There are data translation and bridging technologies that work as adapters to resolve compatibility issues. Those are elegant solutions. But there is a cost to pay. The adapters add latencies and they almost always have major negative impact on performance. Even worse, transaction handling and data consistency is very problematic. Such adapters are additional components on a critical path and they failures are very painful. The resulting system is often operationally fragile: failure of even a minor component means a failure of the entire system. Not to speak about the enormous complexity and cost of the solution.

On the other hand, copying all the data into my application database is so convenient. The application can access the data easily, using just one homogeneous mechanism. Failure of other components are not affecting the critical path. And it is all so much better for performance. Copying the data solves almost all of the troublesome issues. Except for one small detail: the problem of keeping the data up to date. And that is where the synchronization mechanisms come in.

However hard you may try, it is almost impossible to avoid maintaining copies of the data. Identity data are no exception. In fact identity data are often the most heavily affected. And it makes a lot of sense. Applications are built for users to use them. Therefore almost every application keeps some kind of data about users. In addition to that, such data are usually very sensitive from security and privacy point of view.

We cannot avoid copying the data. The best thing that we can do is to keep the copies managed and synchronized. Some applications have built-in support for LDAP or directory synchronization. But those mechanisms are usually quite weak and fragile. For example many applications provide capability for on-demand synchronization with directory service on login time. It usually works like this:

1. User enters username and password to application login dialog.
2. The application connects to the directory service to validate the password.

3. If the password is correct then the application retrieves user data from the directory.
4. The application stores copy of user data locally.
5. Business as usual. Local copy of the data is used ever since.

This approach works quite well at the beginning. But after a while the data begin to stink. Users are renamed, but the local copies are not updated. Users are deleted, but the local copies stay around forever. There are local accounts and privileges that are not reflected back to the directory service and therefore remain undetected for years. Which means that we have a serious security and data protection problem here. And we do not even know that the problem is there.

Some applications have more advanced synchronization processes that can do better than this. However, an application that does synchronization well is still an extremely rare sight. And there is a good reason for this. Synchronization is much harder than it seems. There may be data inconsistencies on both sides. There may be networking errors. Configuration errors. Data models are evolving in time. Policies are changing. It is no easy task to reliably synchronize the data in such environment. Therefore there is a special breed of systems that specialize in synchronization of identity data: identity management systems.

Synchronization in MidPoint

Synchronization is one of the basic mechanisms of midPoint. Synchronization mechanisms are integral part of midPoint design from its very beginning. Many of the things that midPoint normally does are in fact just different flavors of synchronization. There are obvious cases such as reconciliation process synchronizing account attributes with midPoint user properties. But there are also less obvious cases, such as a completely ordinary provisioning case when midPoint needs to create a new account for a user. Even that case is in fact a synchronization: midPoint user properties are synchronized with a new empty account on the resource. Majority of midPoint operations are directly or indirectly using the synchronization principles.



Reuse of the mechanisms is one of fundamental principles of midPoint design. When we have designed midPoint we have not invented a separate mechanism for every midPoint feature. We have rather designed few very generic principles that are re-used at many places in midPoint. Synchronization is one of these principles. There is one code that implements the core of the synchronization logic. And that code is used whenever we need to "align" objects that relate to each other. The same code is used for user-account reconciliation, ordinary provisioning, role-based provisioning, live synchronization, data consistency ... almost everywhere.

MidPoint synchronization is almost a continuous functionality spectrum that can be tweaked and tuned to match specific needs. Yet, the synchronization mechanisms can be divided to several broad and slightly overlapping categories:

- **Live synchronization** is almost real-time synchronization mechanism. MidPoint continually scans the resource for changes. If changes are detected then those changes are immediately processed by midPoint. The actual latencies depend on the capabilities of the resource but usual numbers range from few seconds to few minutes. Only recent changes are processed by live synchronization. Therefore it is a very efficient mechanism which usually has fast responses

even in large-scale deployments.

- **Reconciliation** is a process that compares the data and corrects them. When an account is reconciled midPoint computes the attribute values that the account should have. The computed values are compared to the real values that the account has. Any differences are corrected. Reconciliation is quite heavy-weight mechanism, comparing all the accounts one-by-one. But it is also a very reliable mechanism. It can correct mistakes that were missed by live synchronization, it can correct data after major failures and corruptions and so on. Reconciliation is usually executed in regular intervals. However due to its nature it is usually executed during off-peak times (nights and weekends).
- **Import** is usually a one-time process to get data from the resource to midPoint. Import is used to populate midPoint with initial data or it may be used to connect a new resource to midPoint. Import is almost the same as reconciliation with only a few minor differences. However its purpose is different and therefore there is usually also a slightly different configuration of import policies (mappings). Import is usually not scheduled, it is manually triggered when needed.
- **Opportunistic synchronization** is a very special kind of animal which is quite unique to midPoint. Opportunistic synchronization is triggered automatically when midPoint discovers that something is not in order. For example if midPoint tries to modify an account, but it discovers that the account is not there. Then a synchronization mechanism is triggered just for that single account. Which usually means that the account is re-created. The opportunistic synchronization is also triggered when midPoint tries to create a new account, but the account is already there. This approach makes midPoint a self-healing system. If midPoint runs into a problem it can often correct the problem by itself.

Individual mechanisms differ in a way data inconsistency is discovered: livesync will actively look for new changes, reconciliation will compare the data one-by-one and opportunistic synchronization will discover inconsistency by pure chance. But all the mechanism react to inconsistency in the same way. There is only one policy that specifies how to fix the system. Of course, there may be slight deviations in the behavior. For example we usually want *import* to behave in slightly different way than *reconciliation*. And midPoint allows that. But overall, there is just one big synchronization mechanism. And this has a good reason. It does not really matter how the problem was discovered. What really matters is that the problem gets fixed. We do not want to maintain four separate configurations for that. Having one policy is enough. MidPoint knows which part of the configuration need to be applied in each specific situation. And it does it automatically. This unifying approach significantly simplifies the configuration of midPoint synchronization mechanisms. And that is also a reason why the boundaries of individual synchronization mechanisms are quite fuzzy. In fact this is just a single big mechanism with several facets.

Sources, Targets And Other Creatures

The tale of idealistic identity management deployment starts with a human resources (HR) system. The HR system is supposed to have records for all the identities therefore it is *authoritative source* resource. Identity management system pulls in all the data from HR, recomputes them and creates accounts on *target resources*. And they lived happily ever after.

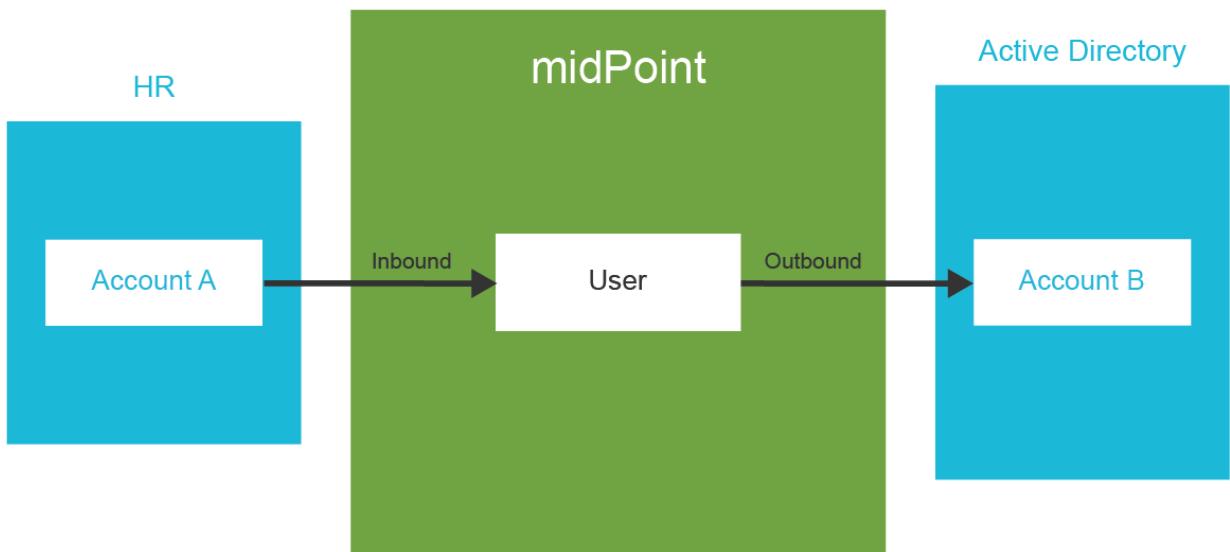
Now, let's get back to reality. The HR resource is indeed an authoritative source of data in many real-world cases. But it is a limited source. It contains only data about employees. And it has only a

partial information. For example there is no username. Username has to be generated by IDM logic. There is no initial password. Organizational structure assignment is often incomplete, missing or unreliable. Therefore it is only a *partially-authoritative source*. There may be additional authoritative sources for contractors, partners, suppliers, support engineers and other identities that need to access our systems. These are *additional source systems*. Then there is a directory system which is often Active Directory. This should be a *target resource* in theory. But there usually are pieces of authoritative information in here. For example an algorithm to generate a username may be based on the usernames that are already taken in the Active Directory. The active directory may also be needed to create an e-mail address. Directory systems are also used as semi-authoritative sources for telephone numbers, office numbers and so on. Therefore such resources are both *target* and *source* resources. And then there are finally target resources. These are not authoritative in any way. Identity management system will only write to these. Or ... will it? What happens when a conflicting account already exists on such resource and therefore we cannot create a new account for a new employee. And how do we check if there are no accounts that are not supposed to be there? It turns out that even the target systems contain valuable information after all.

The reality brings a wild mix of source, target, semi-source, target/source and quasi-target resources that are almost impossible to put into a pre-defined boxes. Therefore midPoint does not bother to define a concept of "source" or "target" resource. All resources can be both sources and targets and the authoritativeness of each attribute can be controlled on a very fine level. Almost every real-world situation can easily fit into this model.

Inbound and Outbound Mappings

MidPoint is firmly based on the principle of reuse. Previous chapter explained that behavior of attributes during provisioning is controlled by *mappings*. Therefore it is perhaps no big surprise that the behavior of attributes during synchronization is also controlled by mappings. In fact, provisioning is just a special case of synchronization. Following picture explains the combined mechanism.



There are two types of mappings:

- **Inbound mappings** map data *into* midPoint. These mappings take the data from the source resources, transform them and apply the result to the user object.
- **Outbound mappings** map data *out of* midPoint. These mappings take user properties, transform them and apply the result to account attributes in target systems.

The mappings themselves are almost the same regardless whether they are inbound or outbound. They have sources, targets, expressions, conditions, etc. Just the sources and targets are reversed:

	Inbound mapping	Outbound mapping
Direction	resource → midPoint	midPoint → resource
Mapping source	resource object (e.g. account)	focal object (e.g. user)
Mapping target	focal object (e.g. user)	resource object (e.g. account)

That is it. Just think about the same mappings that were used in previous chapter, just flip the direction. Now the mapping will take data from the account and the results will be applied to user object. Like this:

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

This mapping will take the value of `lastname` attribute from the resource and store the value in `familyName` property of midPoint user.

The rest is the same as outbound mappings. All the expressions and evaluators can be used for inbound mappings in the same way as for outbound mappings. For example a Groovy expression can be used to sanitize the value before it is stored in midPoint:

```

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

```

The same approach can also be taken for activation and even for password mappings. However, there is one difference for password mappings. Password are usually write-only value. When the password is written it is usually hashed and the original value cannot be retrieved any longer. Then there are resource such as HR systems that do not store employee passwords at all because those are not really accounts that we are reading. Those are just regular database entries that the connector presents as accounts. Inbound password synchronization is almost never easy and it often requires a lot of planning and ingenuity. However, there is one method that is used quite often. The initial user passwords are usually randomly generated. As this is a very common case midPoint can do this easily:

```

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

```

This mapping generates random password for a user. Both the mapping and the **generate** expression evaluators are quite smart. The mapping knows that the target is user password without any need to explicitly specify that. In addition to that the generate expression evaluator will take user password policy into consideration. It does not make sense to generate any random password. If we do not consider password policy then we can generate password that is too short, too long, too weak to pass the policy or to strong to be useful in any way. Therefore the **generate** expression will look for password policy and generate a random password that just matches the specified password policy.

There are more important details to see here. The inbound password mapping is *weak*. And there is good reason for this. We do not want midPoint password to be replaced by randomly generated password. We only want to set a random password in case that it is an initial password, the first

and only password. And that is exactly what a weak mapping does: it sets new value only if the target does not have any existing value. Therefore this mapping will not overwrite passwords that are already set.



There is no direct account-account synchronization in midPoint. As explained before, midPoint follows start topology (a.k.a. "hub and spoke"). Therefore the synchronization is either from account to user (inbound) or from user to account (outbound). The effect of account-account synchronization is achieved by combining inbound and outbound synchronization mechanisms.

Correlation

It is all quite easy to import all HR records into an empty midPoint. Set up inbound mappings, start import task, wait a bit and all is done. But practical situations are much more complex. Synchronization algorithm usually do not run on a green field. Live synchronization and reconciliation are supposed to work with pre-existing midPoint users. And import is usually not trivial either, for example in cases when we try to import data from an additional data source into a running midPoint deployment. Some users in the import set are new, but there may be accounts for existing users. We need to tell the difference between brand new account and an account that belongs to an existing user. We need to handle those situations in a different way. Of course, midPoint has an easy solution for this: correlation mechanism.

Correlation expression is a method how to connect newly-discovered accounts and existing users. It works like this: whenever midPoint discovers new account it will try to link that account to an existing user. Correlation expression is used to do this. Correlation expression is in fact a parametric search query. Such search query is constructed for every new account and it is used to look for users that the account belongs to. The easiest form of the correlation expression is to look by using an identifier:

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

This correlation query takes the value of `empno` attribute of the account. This value is placed into the search query that midPoint computes in memory. Given an account with `empno` attribute set to `007`, the resulting search filter looks like this:

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

MidPoint looks for users that match this search filter. If there is an user with `employeeNumber` property set to `007` then such user is considered to be an owner of the account.

MidPoint has its own data representation mechanism and object structure. Therefore midPoint also has its own query language that is designed to work with the object structure. The query language is not difficult to learn as it follows the structure of other common query languages. The language itself is described later in the book and in midPoint documentation. But do not worry about this too much now. Vast majority of correlation expressions is very simple. In fact it is usually just a single `equal` clause just like that one used in the example above.

Using search queries for correlation may seem a little bit too complex. But it is necessary. The correlation expression must be a search filter because that is the only efficient way how to find single user in a large set of other users. We cannot scan the accounts one-by-one. We need to utilize the search power of the database for this.

Synchronization Situations and Reactions

Correlation expression can be used to find an owner for a new account. That is a part of the solution but not entire solution. If the owner is found then the action is quite obvious: link the account to the user and proceed as usual. But what to do if the owner is not found? This resource may be an authoritative resource and therefore we want to create a new user based on the account. Or this may be a reconciliation with a target resource and in that case this means that we have found an illegal account. We probably want to disable such account. And what to do if more than one owner is found? This can all become quite complicated. Therefore midPoint as a concept of *synchronization situations* to make it understandable and manageable.

Whenever midPoint deals with a change on an account the *situation* of that account is determined. The situation reflects whether this account is already linked to the user, whether we know the candidate owner, but it is not linked yet, whether we cannot determine the owner and so on. Individual situations are explained in the following table.

Situation	Description
<code>linked</code>	The account is properly linked to the owner. This is the normal situation.
<code>unlinked</code>	The account is not linked to the owner, but we know who the owner is. Correlation expression told us who the owner is. In this case midPoint thinks that the link should exist, but it is not linked yet.
<code>unmatched</code>	The account is not linked and we not even know who the owner is. The correlation expression haven't returned any candidates.
<code>disputed</code>	The account is not linked, but there are more potential owners. The correlation expression returned more than one candidate.

Situation	Description
collision	The account is linked to more than one owner. This should not happen under normal circumstances. This is usually caused by faulty customizations or software bugs.
deleted	There was an existing account but it was deleted on the resource.

After synchronization *situation* is determined, midPoint continues by figuring out what a proper *reaction* is. The reaction is quite clear for some situations (e.g. **unlinked**), but there is a lot of variability for other situations (e.g. **unmatched**). This variability is a reason that midPoint allows to set a reaction for each situation individually. There are several pre-defined reactions:

Action	Description
Add focus	New midPoint user will be created and it will be linked to the account. This is usually a reaction configured for authoritative resources, used in situation when a new account is discovered.
Delete focus	MidPoint user that owns the account will be deleted. This is usually a reaction configured for authoritative resources, used in situations when midPoint detects that an account was deleted.
Inactivate focus	MidPoint user that owns the account will be disabled. This is also used for authoritative resources. But this is a milder reaction.
Link	The user-account link will be created.
Unlink	The user-account link will be removed. The account will no longer be linked to the user.
Delete shadow	The account will be deleted. This is the usual reaction when illegal account is detected on non-authoritative resource.
Inactivate shadow	The account will be disabled. Usually a milder reaction to an illegal account.

If no reaction is explicitly configured for a situation then midPoint does nothing. Just the situation is recorded in midPoint repository. This is part of midPoint philosophy not to change the data unless an action was explicitly configured.

The reactions can be defined in the synchronization section of resource configuration:

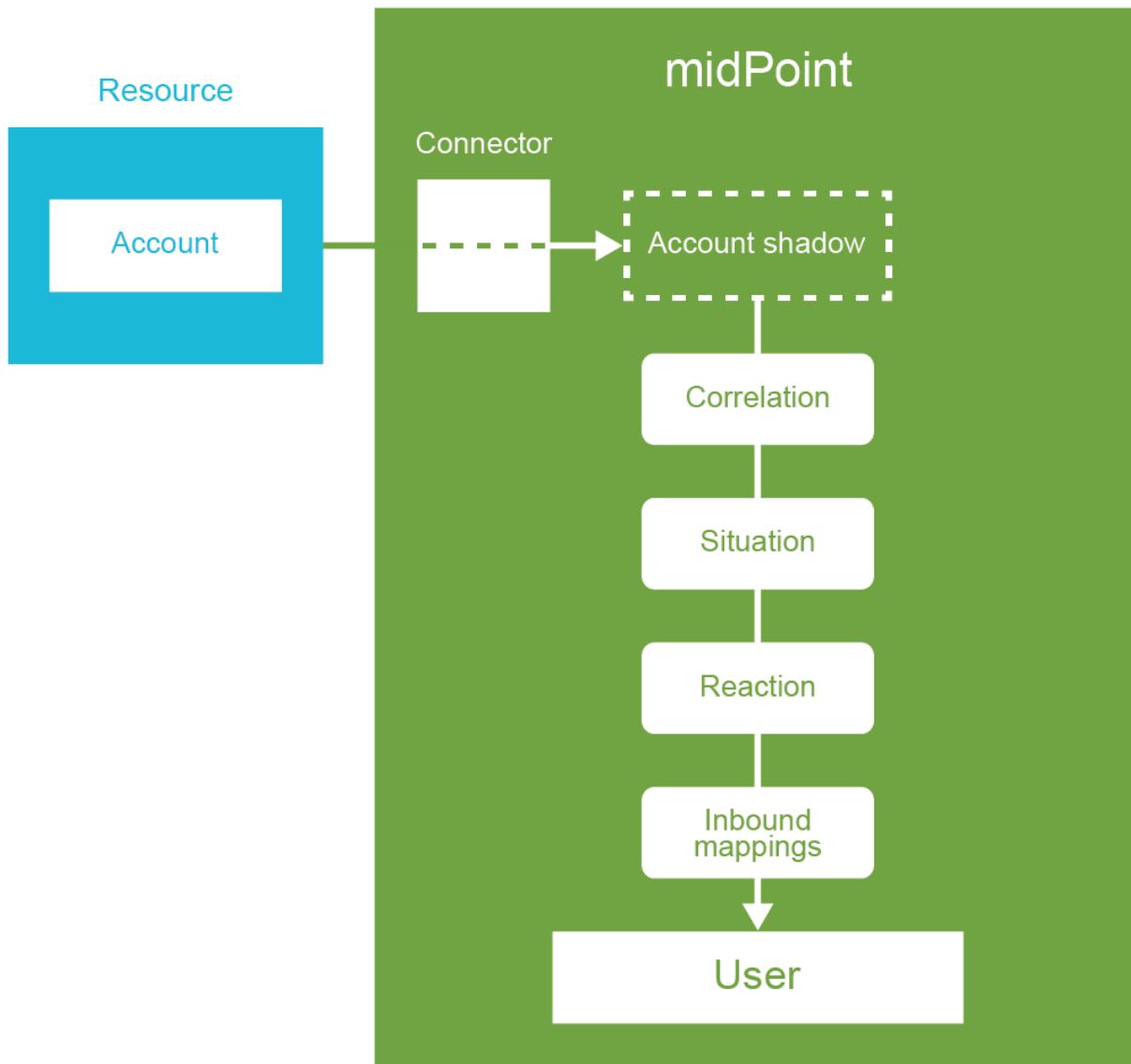
```

<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/action-3#deleteFocus</handlerUri>
      </action>
    </reaction>
    <reaction>
      <situation>unlinked</situation>
      <action>
        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#link</handlerUri>
      </action>
    </reaction>
    <reaction>
      <situation>unmatched</situation>
      <action>
        <handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-3#addFocus</handlerUri>
      </action>
    </reaction>
  </objectSynchronization>
</synchronization>

```

Most of the configuration is perhaps self-explanatory. This is a typical authoritative resource. If there is a new account on the resource and we do not have an owner (situation: **unmatched**) then create a new user (action: **addFocus**). If the account is deleted from the resource (situation: **deleted**) then delete the user as well (action: **deleteFocus**). If we happen to find an account which should be linked but it is not (situation: **unlinked**) then link it (action: **link**). The only think that deserves an explanation is the reaction to linked situation. In this case there is not much to do. Everything seems to be in order. However there may still be attributes that are not set correctly. Remember the inbound mappings? The inbound mappings were not even mentioned in this section yet. And for a good reason. The inbound mappings are not evaluated at this stage. Evaluation of inbound mappings happen only after the situations and reactions are evaluated. We need this so all the accounts are properly linked (or unlinked) and the inbound mappings have valid sources and targets. But the evaluation of inbound and outbound mappings do not happen by themselves. MidPoint does not change the data unless it is explicitly configured to. There are reactions for **unmatched**, **deleted** and **unlinked** situations. Therefore in those cases midPoint assumes that it is expected to fully synchronize everything and therefore all the mappings and policies are evaluated automatically. But there is no reaction for **linked** situation. In that case midPoint assumes that it should do nothing as nothing is explicitly configured. Hence the **synchronize** property. This property

can be used to force midPoint to do full synchronization even if there is no explicit action configured. And it can also be used to avoid full synchronization even if explicit action is configured.



The figure above illustrates the usual sequence of events during inbound synchronization:

1. Account is stored in the resource database.
2. Appropriate identity connector is used to read the account.
3. Account shadow is created in midPoint.
4. Correlation expression is evaluated to determine account ownership (if the account is not already linked to a user).
5. Synchronization situation is determined based on account ownership and state of the account.
6. Appropriate reaction to the situation is determined based on resource configuration.
7. Inbound mappings are evaluated to map account values to the user.

Please note that the description of this process is slightly simplified for clarity. There are also

obvious deviations from this process. E.g. inbound mappings are skipped in case that the user is about to be deleted, the mappings are also skipped if the reaction does not include “synchronization” and so on. But generally this is what usually happens during inbound synchronization.

MidPoint is an extensible system. There are several prefabricated synchronization reactions described above. Those reactions can handle vast majority of situations that happen during synchronization. However, there is a possibility to extend the system with completely custom reactions. MidPoint was designed for this. This is the theory. However, currently this part of midPoint is only partially extensible. Full extensibility feature was planned, but it was never implemented. Therefore extensibility of synchronization reaction is possible, but it might be quite hard to achieve this in practice and it may require significant development effort. But there is another way. MidPoint development team would absolutely love to finish this extensibility feature as it was originally planned. However, existing midPoint customers had so far prioritized other features. MidPoint subscribers and sponsors are funding the development therefore midPoint development must follow their priorities. Therefore if you are interested in full synchronization reaction extensibility (or any other feature) please consider purchasing midPoint subscription or sponsoring the feature.



Synchronization Tasks

Now we know how the inbound synchronization works: midPoint reads the account, then correlation is applied, situation determined and reaction executed. However, we have not yet discussed the details of the very first step: how does midPoint actually read the account? Nothing happens without a reason, therefore there must be some active component in midPoint that actually looks for the new, changed and deleted accounts. And that component is a *synchronization task*.

MidPoint *task* is an active process that is running inside midPoint server. This is the first time that we encountered the concept of a *task*, but it is definitely not the last one. Tasks are used for many purposes in midPoint. They are used to track long-running operations, approvals and actions that work on large sets of objects (bulk actions). There are tasks that execute cleanup jobs, compile reports and provide variety of other functions. The concept of tasks is a very powerful and flexible one. Tasks can be used to track execution of a short one-off operations. Tasks can be used to execute scheduled actions in regular intervals. Or tasks can be used to track long-running processes. We will be using tasks in almost every chapter of this book.

Tasks are used as an active component to "run" almost all synchronization mechanisms:

- **Reconciliation task** is listing all the accounts from a specific resource. The task executes reconciliation process for every account that is found. This means that midPoint computes how that particular account should look like and then the computed values are compared with real account attributes. This task is usually scheduled for regular execution using quite a long execution interval (days or weeks).
- **Live synchronization task** is looking (polling) for changes in a specific resource. The task will

look for created, modified and deleted accounts. The task will get a description of the change and pass that to midPoint synchronization mechanisms. This task is almost always scheduled for regular execution and the execution interval is very short (minutes or seconds).

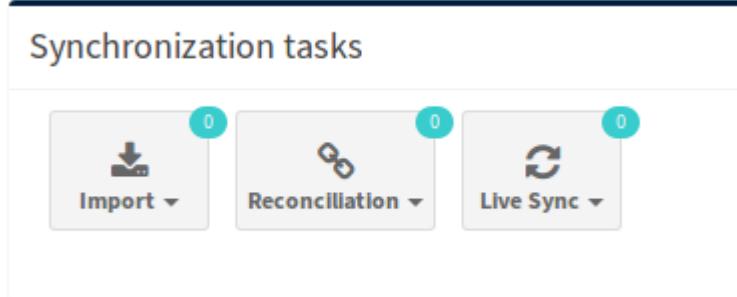
- **Import from resource task** is listing all the accounts from a specific resource. The task will pretend that the accounts were just created. This usually motivates midPoint to create users based on those accounts or link these accounts to existing users. This task is usually not scheduled and it is almost always executed manually.

Each type of synchronization task is detecting changes using a different mechanism. However, once the task detects the change or reads the account then the processing is the same for all tasks. All the tasks lead to the same algorithms based on the same configuration and policies. Therefore it does not matter whether it has all started in reconciliation or live synchronization task. It will all end up in the same correlation-situation-reaction-mapping process.

However, the tasks are necessary to initiate the synchronization. They are the active part, the spark that starts the synchronization process. Without the tasks the synchronization does not really work. There are ways how the synchronization can "happen" even without a task, e.g. as a reaction to user interface operation or if a new account is discovered during an unrelated operation. But practical deployments need at least one synchronization task to work properly. This task takes care of vast majority of synchronization cases.

Strictly speaking tasks are quite a strange kind of animal. Tasks have their data and configuration as most other midPoint objects. But tasks are active. Therefore there are CPU threads associated with the tasks when the tasks are running. There are mechanisms how to monitor task progress. The tasks need to be cluster-aware so they can fail over to a different midPoint node if one node fails. The tasks are quite rich and a bit tricky to handle. But midPoint is making task handling reasonably simple. Tasks are represented as ordinary midPoint objects. Therefore they can be imported to midPoint in XML/JSON/YAML form as any other object. Tasks can be easily edited in their XML/JSON/YAML form to change the scheduling, modify the parameters and so on. Of course, there are some special functions that only the tasks have (such as suspend and resume) that cannot be controlled using the XML/JSON/YAML format. But vast majority of task management can be done using the very same methods that are used to control other midPoint objects.

Tasks can be created by taking the XML and importing that to midPoint. And that's the way how synchronization tasks are often managed. When an XML-formatted resource definition is created then there is often an associated synchronization task. Which means that both resource and all the necessary synchronization tasks can be imported together. Synchronization tasks can also be created from midPoint user user interface. They are usually created by using special-purpose buttons in resource detail pages.



Once the synchronization tasks are created they can be managed in the same way as other tasks are managed: in the *Server tasks* part of the midPoint user interface.

Synchronization Example: HR Feed

This section describes complete working example that feeds HR data into midPoint. The ExAmPLE company HR system is an old and complex system. Therefore the easiest integration method is to use structured text exports. The HR system is set up to export the employee data to a comma-separated text file (CSV) every night. MidPoint takes this export file and updates the data about users.

This configuration is done in three steps. First we will use a simple setup to import the data into midPoint. This is an operation that is executed only once. Then the configuration will be updated to run scheduled reconciliation task. Reconciliation compares all the data records every time and it makes any necessary updates. Even though this method would be perfectly acceptable for the company of this size, we still set up a live synchronization task.

The core of the configuration is contained in a single resource definition file. Following paragraphs explain individual parts of the file. There are few additional configuration files for reconciliation and live synchronization tasks. Simplified XML notation is used for clarity. The complete file in a form that is directly usable in midPoint can be found at the same place as all the other samples in this book (see [Additional Information](#) chapter for details).

This HR resource is a data source. It will be used to "pull" the data inside midPoint. However, as we have described previously, there is no fundamental difference between source and target resources in midPoint. Therefore this HR resource starts in entirely ordinary way. There is a reference to the CSV connector and the connector configuration:

```
<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>
    ...

```

The next section is schema handling configuration. That is where it becomes slightly more interesting. The schema handling section contains inbound mappings for HR account attributes:

```

...
<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>
  ...

```

The account attribute `empno` is mapped to midPoint user properties `name` and `employeeNumber`. Account attributes `firstname` and `lastname` are mapped to `givenName` and `familyName` properties respectively. This is perhaps self-explanatory.

The next part of the configuration specifies mappings for activation and credentials:

```

...
<activation>
    <administrativeStatus>
        <inbound/>
    </administrativeStatus>
</activation>

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

```

The activation mapping is very simple. Activation is a very specific concept in midPoint. MidPoint knows activation attributes and their meaning. Therefore there is no need to specify a lot of details. The activation mapping simply specifies that the administrative status should be mapped in the inbound direction. And that is it.

However the mapping for credentials needs a bit of explanation. What midPoint sees as HR accounts are not exactly accounts. They are usually just records in the HR database. Nobody is using these HR records to log into the HR systems. Therefore there is no password associated with them. But we need a password for the users in midPoint. Therefore we are going to generate them. And for that we are going to use the weak mapping with generate expression that was explained above.

The mappings are undoubtedly important. The mappings specify how are the account data reflected to midPoint user. But the mappings do not specify whether the accounts should be created or deleted. Mappings control the data, but they do not control the *lifecycle*. It is the next configuration section that makes this resource really authoritative:

```

...
<synchronization>
  <objectSynchronization>
    <enabled>true</enabled>
    <correlation>
      <q:equal>
        <q:path>employeeNumber</q:path>
        <expression>
          <path>$projection/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#deleteFocus</handlerUri>
      <action>
    </reaction>
    <reaction>
      <situation>unlinked</situation>
      <synchronize>true</synchronize>
      <action>

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

<handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-
3#addFocus</handlerUri>
      <action>
    </reaction>
    <objectSynchronization>
  </synchronization>
...

```

Given the information in this chapter this configuration should be quite easy to read. This is how a typical authoritative resource works. If there is a new account on the resource and we do not have

an owner (situation: `unmatched`) then we create a new user (action: `addFocus`). If there is a new account for which we can find existing owner (situation: `unlinked`) then simply link it (reaction: `link`). If the account is linked already (situation: `linked`) then we just synchronize the data. In fact, we will synchronize data for all the other situations as well. Except the last one. If the account is deleted in the HR system (situation: `deleted`) then we want to delete midPoint user as well (reaction: `deleteFocus`). As the user gets deleted there is no point in synchronizing the data. MidPoint knows that and skips application of mappings.

The ownership of the accounts that are not already linked is determined by the correlation expression. In this case the expression will be comparing account attribute `empno` with user property `employeeNumber`. If the values match then the user is considered to be an owner of the account.

There is one more detail in this resource that we have skipped:

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

This is a setting that adjusts the behavior of midPoint *assignments*. As was already mentioned all resources in midPoint are created equal. The source resources must follow the same rules as target resources. And one of the fundamental rules of midPoint is that there should not be any account without a specific reason to exist. In midPoint terminology every account exists because there is an *assignment* that justifies its existence. While this approach is exactly what we want for vast majority of (well behaving) resources, it is not exactly ideal for source resources. Those resources work the other way around. The HR account is in fact a cause for midPoint user existence, not its effect. Therefore there is really useful `assignmentPolicyEnforcement` setting that controls the behavior of assignments. This setting is used in a variety of scenarios, mostly for data migration and to tame resources that just won't behave in a civilized manner. But in this case the setting is used to turn off the assignment enforcement for this resource entirely. As this resource is an authoritative source the assignment enforcement does not make much sense. Behavior of this resource is defined by the `synchronization` section of resource configuration.

Resource configuration is complete now. This configuration sets up the connector, mappings and synchronization policies. This configuration is the same for all the synchronization flavors: import, reconciliation and live sync - they will all use the same settings. When it comes to configuration, the only difference between those synchronization flavors is the way how the synchronization tasks are set up. If an import task is set up then import of resource accounts will be executed. If reconciliation task is set up the reconciliation will be executed. It is all in the tasks. And synchronization tasks can be easily set up using those convenient buttons in the user interface. But we like to make our lives a bit painful in our part of the world. Therefore we are going to go hardcore and we import the tasks in the XML form.

First task is an import task. This task lists all the accounts in the HR CSV file. The task pretends that each of the accounts was just created. If the task is executed for the first time then resulting situation of the accounts is going to be either `unmatched` or `unlinked`. Therefore the task creates new

midPoint users or links the accounts to existing users.

```
<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>
```

This is a very basic structure of the task. Similarly to all midPoint objects a task has a name. Then there is a task identifier which is used for internal task management purposes. This is usually the same as task object OID. Task needs definition of an owner. The owner is a user that is executing the task. This is important, because authorizations of task owner determine what the task is allowed to do. This is also the identity that will be recorded in the audit log. In this case **administrator** is owner of this task. Task execution status tells whether the task is running, it is suspended or finished. Then there is a handler URI. The handler URI specifies what the task really does. It (indirectly) refers to the code that the server executes. In this case the task URI specifies that this is a synchronization task that imports accounts from the resource. And the resource is specified by the objectRef reference. This points to our HR resource. The last item is recurrence. Recurrence specifies whether the task runs only once (single) or whether the execution should be repeated (recurring).

When this XML definition is imported to midPoint, the server tries to execute the task. That means that import of accounts from the HR resource starts immediately. Progress of the task can be monitored in the Server tasks section of midPoint user interface. The import task is not a recurring task. Therefore it will run only once. If you need to re-run the task you can do that from midPoint user interface. But the task will not be executed unless you explicitly tell midPoint to do so. This is how typical import tasks work. They are usually executed when a new resource is connected to the system. And once everything is set up, correlated and linked then the import task is not needed any more.

A clever reader may ask what happens when the import task is executed more than once. The answer is simple: not much. Even if the task pretends that the accounts were just created, midPoint is not fooled easily. In fact it is hard to believe that the account was just created if midPoint already has shadow for that account and it is linked to a user, isn't it? Therefore midPoint is going to stay calm and carry on. If there is any change in the account attribute than the change will be reflected to the user. But that is it. No big drama here.

Import task will get the data from the resource into midPoint. But as import is not a recurring task it will not keep the data synchronized. Import tasks are not designed to do so. But there are other tasks that are designed for continuous synchronization. Reconciliation task is one of these. Reconciliation task lists all the accounts on a resource and compares that with data in midPoint.

```

<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>

```

Definition of a reconciliation task is almost the same as the definition of import task. But there are crucial differences. First of all there is different handler URI. This is what makes this task a reconciliation task. Then the task is recurring. This means that midPoint will repeat execution of the task. Therefore there is also execution schedule so the server knows when to execute the task. Reconciliation tasks are usually resource-intensive therefore we usually want to execute them at a very specific off-peak times. For that reason the execution schedule is defined using a cron-like pattern. UNIX-friendly readers will be surely familiar with this. The format is:

seconds minutes hours day-of-month month day-of-week year

Therefore this task will be executed every Saturday at 01:00:00am. There is also definition of misfire action. Misfire is a situation when the server is down at the time when the task is supposed to run. Therefore if the server is down in the early hours of Saturday this task will be executed as soon as the server starts up.

Reconciliation task is a real workhorse of identity management. It can be used in almost any resource. It is very reliable. It is often used to fix many problems, apply new policies, look for missing accounts, illegal accounts and so on. It is indeed a really useful tool. But it has its downside. Reconciliation iterates through all the accounts, it recomputes all the applicable policies for every account, one-by-one. Therefore it may be quite resource-intensive. It may be even quite brutal if the policies are complex, user population is high and the resources are slow. This can take hours or even days in extreme cases. But even for smaller deployments reconciliation is not entirely easy. The problem is not in midPoint. MidPoint can be usually scaled up to handle the load. But listing all the accounts often may put unacceptable load on the resources. Therefore reconciliation is not executed often. Daily, weekly or even monthly reconciliation seems to be a common approach. Reconciliation is reliable, but it is not entirely what we would call "real-time". But of course, midPoint has a faster alternative.

Live synchronization is the way to go for real-time synchronization. Or rather almost real-time synchronization. Practical latencies for live synchronization are in the range of seconds or minutes, which is fast enough for most practical cases. Live synchronization is also quite resource-efficient. Overall it is much faster and much lighter than reconciliation. But live synchronization is not available for all resources. Live synchronization depends on the ability to get recent changes from

the resource in a very efficient way. Therefore it is only available for resources that record the changes. The specific mechanism to record the changes may vary from resource to resource. It may be as basic as a simple modification timestamp or it may be a complex change log. But it has to be good enough for the connector to discover recent changes and it must be efficient enough for the connector to do that every couple of seconds. If such mechanism is available and the connector knows how to use it then setting up live synchronization is easy. All that is needed is synchronization task.

```
<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>
```

This task definition should be easy to understand now. There is a different handler URI that makes this a live synchronization task. There is also a different type of scheduling. We do not want to execute this task at a specific time. We rather want to execute it all the time at regular intervals. In this case the interval is set to 10 seconds. And that is all that is needed to have live synchronization running. If the HR CSV file is changed now, the changes will get automatically processed by midPoint.

Setting up configuration flavors is just a matter of setting up the tasks. The rest of the configuration is the same for all flavors. Therefore it is very easy to run both live synchronization and reconciliation for the same resource. Just create two tasks. And it fact this is quite a common setup. Live synchronization is used to get the changes quickly and efficiently. And reconciliation is used to make sure all the changes were processed and that the policies are applied consistently.

That is all. Now we have the HR feed up and running. However, there are still few issues. A clever reader would surely notice that this is not a very good HR resource. MidPoint users created from this HR feed have given name and family name, but the full name field is empty. But do not worry. We will sort that out in later chapters with the help of *object template*. Also the users have employee number as their username. This may be in fact a very good approach for some deployments as it avoids the need to rename accounts. However, it is not a very user-friendly approach. Therefore most deployments need to generate more convenient usernames. This is easy to do with midPoint and we will also address that later. There is still a lot of things to learn before we get to a complete synchronization set up.

HR Feed Recommendations

All resources are created equal in midPoint. However, source resources almost always have slightly special standing. Even though midPoint mechanisms are the same for all resources, the data coming from the sources often have significant impact on the entire solution. There is this traditional computer engineering wisdom: *garbage in, garbage out*. An error in data feed may cause a lot of problems everywhere. Therefore it is important to get the data sources right. And this is usually one of the first steps in an IDM project.

Unfortunately, source data feed is usually quite difficult to set up correctly. And it is almost impossible to get it right at the first try. Therefore setup of a data source is usually an iterative process. And there may be many iterations - especially if the quality of the source data is unknown. The process usually goes like this:

1. Set up initial source resource definition based on the information you have. Set up connector and test connection. Check that you can see the accounts. Set up mappings and synchronization policy.
2. Test the import process on a couple of individual accounts. Navigate to the resource details pages, click on Accounts tab to list accounts, choose an account and click on the small *Import* button at the left-hand side of the table row. Import of that individual account will start. Just that one account. It is easier to see the errors (see step 6) by using this method.
3. Fix any errors that you see and repeat step 2.
4. Create an import task and run import of all accounts.
5. Examine task errors. You can use task details page to get the summary.
6. If there are no errors the examine the users. If everything is OK then congratulations. You have a good import. However, this is unlikely to happen of first few couple of attempts.
7. You will probably need to have a look into system logs to learn the details of individual import failures. MidPoint heavily relies on logs for detailed error analysis. See the [Troubleshooting](#) chapter of this book to learn how to adjust log levels and how to understand the log messages.
8. Some errors are likely to be caused by the errors in your mappings and policies. Those are usually easy to fix. But there are usually worse errors as well – errors caused by wrong or unexpected input data. The right way would be to fix the data. But that is not always possible (in fact it is almost never a feasible option). Fortunately, most of the input data errors can be fixed (read: "worked around") in midPoint with a bit of ingenuity. Just use the power of the mappings.
9. Rinse and repeat. If the errors you get are not severe then you may simply re-run the import task. This often works just fine. But if the problem was in a mapping that completely ruined all the data then it is perhaps best to start with a blank slate. We are all just human and this situation happens quite often, especially in the beginning while you are still learning. Therefore there is a special feature to help you out. Navigate to *Configuration > Repository Objects*. There is a small unassuming expand button in the top-right part of the screen. That button opens a context menu. Select “Delete all identities” item. That is what we lovingly call “laxative button”. A brief dialog will pop up asking you to specify which identities exactly are to be deleted (users, shadows, ...). This is a very convenient way how to get back to a black slate, but keep all the configuration (resources, templates, tasks).

10. Goto step 2. Repeat until done.

If the initial IDM deployment step includes an HR feed we strongly recommend to start with that HR feed. It is significant benefit to have authoritative HR data in the midPoint to start with. It is usually easier to correlate other resources to midPoint users later on, if the users were created from a reasonably-reliable HR data. Also, it will usually take some tweaking to get the HR import right. The possibility to easily clean up midPoint and get to a clean slate is extremely useful. But that is possible only if the HR feed is the first resource that is connected to midPoint.

A clever reader would notice, that we assumed that the source feed will be taken from a CSV file. And this is indeed the case in majority of cases. If a new employee or contractor is about to join the company there is usually no hurry. This information is entered into the HR system at least few days in advance therefore daily CVS export is perfectly acceptable. However, there may be cases when we want a faster response. Or maybe we do not want additional burden of dealing with CSV exports. Of course, there is solution. In theory any connector can be used for source resource. And in fact there are specialized connectors that are taking data directly from the HR system. For example there is a connector for Oracle HCM system. Unfortunately, there is no connector that can take data from SAP HR system yet.

Synchronization and Provisioning

Synchronization and provisioning are intimately connected. Everything that we have explained about provisioning in the previous chapter also applies to synchronization. In fact provisioning and synchronization are just applications of the same basic mechanisms. Provisioning starts with modification of a user. Synchronization starts a bit earlier: inbound mappings are used to map values from source system to the user. The result of inbound mapping evaluation is that the user object is modified. According to midPoint principles it does not matter how the user was modified. The reaction is the same: accounts are provisioned, modified or deleted as needed.

The synchronization (*inbound* processing) and provisioning (*outbound* processing) usually happen in the same seamless operation. For example the HR connector detects update in the last name of an employee. That modification is applied to midPoint user, therefore the family name of midPoint user is updated. The operation continues by evaluating all templates, roles and outbound mappings. The outbound mappings usually map the family name change to the resource attributes. Therefore the resource accounts liked to the user are immediately updated. All of that happens in a single operation. That is how midPoint works. MidPoint is not a human. It will never procrastinate (unless explicitly instructed do to so). MidPoint will not postpone the operation for later if the operation can be executed immediately. MidPoint tries to get the data right on a first try. Therefore there no specialized propagation or provisioning tasks that you might know from older IDM systems. MidPoint does not need them.

There other advantages in doing everything in one operation. It is all one operation therefore midPoint knows all the details: what was the cause, what is the effect, what exactly has been changed. This is important for troubleshooting. Some IDM systems decouple the cause and the effect. Such a divided approach may have its advantages, but it is an absolute nightmare when an engineer needs to figure out why a certain effect happened. But midPoint has both the cause and the effects correlated together in a single operation. Therefore it is much easier to figure out what is going on. And it can also be neatly recorded in the audit trail. And there is another huge advantage:

midPoint knows exactly what has been changed. This means that midPoint does not only know the new value of a property. MidPoint knows also the old value and values that were added or removed. This is a complete description of the change that we call *delta*. This is recorded at the beginning of the operation and propagated all the way until the operation is done. Therefore the mappings may be smart. This approach enables a lot of interesting behavioral patterns. For example it is quite easy for midPoint to implement the "last change wins" policy. In this case midPoint will simply overwrite only those attributes that are really changed in operation. MidPoint can leave other values untouched. In fact, this is the default behavior of midPoint. And it is a very useful behavior during deployment of a new IDM system.

Careful processing of the operations allows configurations that are not feasible with older IDM systems, e.g. a resource that is both a source and a target. In fact a lot of IDM systems can have resource that is both a source and a target - as long as it is a source for one attribute and a target for another attribute. But midPoint can live with a resource where the same attribute is both a source and a target. And in fact there may be many sources and many targets for the same property at the same time. And this is indeed very useful configuration. Just think about telephone number property. This is usually something that the user sets up himself. This may be set up by some kind of specialized self-service, it may be updated by a call center call, the user may update that in his Active Directory profile ... there are many ways how this information is changed. But we want this property to be consistent. We want telephone number to be the same everywhere. And we do not care where it was changed. We just want to propagate the last change from anywhere to all the other systems. MidPoint can easily do this. Just specify both inbound and outbound mappings for the same attribute:

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

In this case the change in user property `telephoneNumber` will be propagated to the account attribute `mobile` (outbound change). But also a change in the account attribute `mobile` will be propagated back to user property `telephoneNumber` (inbound change). Last change wins. A clever reader certainly grumbles something about infinite loops now. But do not worry. MidPoint can see complete operation context, both inbound and outbound sides. Therefore midPoint knows when to stop processing the operation. There are even mechanisms how to avoid loops caused by connectors detecting changes caused by the connector itself. MidPoint will break those loops automatically.

Synchronization and provisioning are in fact almost the same mechanism applied in a different direction. Then why there two sections in the resource configuration? Why there is `schemaHandling` and `synchronization`? Why not just one? The answer is simple: history. No software is created perfect on day one. Similarly to other software systems, midPoint went through an evolutionary process of continuous improvement. MidPoint had a very good design at the beginning. Looking back at the initial design, now it is quite clear that almost all of midPoint developments were correctly foreseen and accounted for in the design. However, there are occasional mistakes. The initial midPoint data model design expected that there will be major differences between synchronization and provisioning mechanisms. Therefore there were two sections, one for each mechanism. But midPoint evolution improved the initial design and we have found a way how to unify synchronization and provisioning mechanisms into one. However, we have not modified the initial data model because we wanted to keep compatibility. Having two sections instead of one is only a cosmetic imperfection. It does not cause any major trouble. But incompatible change would certainly affect continuity of midPoint deployments. And we highly value midPoint continuity and upgradeability. Therefore the two sections remained to this day. However, they will not remain there forever. We are not going to dwell on old mistakes for too long. These two section will be reunited once there is a proper time to make incompatible changes. Which will probably happen in the future when the time comes to release midPoint 5.0.



Synchronization Strategies

Synchronization is simple in theory. However, as usual, the devil is in the details. Similarly to provisioning, there is no one single "synchronization protocol" that would work for all the source systems. Every system type has its own way to synchronize data. Some systems (such as LDAP) even have several mechanisms. And then there are source systems that have no practical way how to implement synchronization. We would refer to such methods as *synchronization strategies*.

Specific details of each synchronization strategy is an internal matter of connector implementation. The strategy is configured on connector level and the details are, theoretically, hidden inside the connector. MidPoint does not know and does not need to know what synchronization strategy is used. That might work in an ideal world. But we live in practical world and there are many details that leak through the *synchronization* abstraction.

Let us take LDAP as an example. LDAP is, theoretically, a standard. However, the standard does not specify any synchronization mechanism. There is experimental RFC4533, however that is not widely adopted. However, synchronization capabilities are needed and therefore every major LDAP server provides some mechanisms. Some mechanisms are quite good, some are not. There is ancient "Retro change log", going back to Netscape/iPlanet LDAP servers, but still used today. Active Directory has its own "DirSync" synchronization mechanism. OpenLDAP has yet another mechanism based on access log. There is RFC4533, which is used so rarely that there was no request to implement it in midPoint LDAP connector. And then there is a "catch all" synchronization that looks for recent changes based on `modifyTimestamp`.

In theory, all the synchronization strategies above should be equivalent. But they are not. For example, some variants of "Retro change log" synchronization cannot reliably detect rename operations. There may be problem with delete operations as well, especially if coupled with rename operations. Almost every mechanism has its quirks. And then there is the `modifyTimestamp`, which is the most problematic of all.

Unfortunately, it is quite common to use a synchronization strategy based on last modification timestamp. Not just for LDAP, but also for database tables and other types of source systems. This is perhaps understandable, as this is a very simple mechanism. However, it has a lot of problems. The obvious problems can be caused by de-synchronized time on network, although in the age of Network Time Protocol (NTP) this should not be a problem. The other problem is a timestamp granularity. If the timestamp is granular to one second, that can be a big problem. One second is a very long time for a computer. A lot can happen in one second. Therefore the connector has to include the "boundary" second both consecutive synchronization runs, which means that the records may be processed twice. Going for millisecond granularity makes the problem less severe, but the problem is still there.

However, the worst problem is that this strategy cannot detect deleted objects. Deleted objects are not there any more, they do not have last modification timestamp, therefore they will not be included the search. Which means that there must be a reconciliation process running together with synchronization. But wait a minute, it is usually reconciliation to run reconciliation anyway, as a form of "safety net", isn't it? It is, but the difference is timing. It is one thing to run reconciliation once a week to make sure that no records were missed. And it is a completely different thing to run it every hour to make sure deleted objects are properly handled. This makes a huge difference, especially for deployments with millions of entries. Strategies based on last modification timestamp may look like a good idea at the beginning. But they usually turn to a major liability in the long run. Avoid them if you can.

The bottom line is, that synchronization strategies are not created equal. In fact, the individual strategies tend to have vastly different characteristics. Our advice is to learn how each synchronization strategy works, what are the limitations and when it fails. Also, avoid the use of strategies based on last modification timestamp if there is any other viable alternative.

Mapping and Expression Tips and Tricks

Mappings and expressions form a very powerful mechanism. In fact most of midPoint configuration is about setting up correct mappings. But with great power comes great responsibility and mappings may look a bit intimidating at a first sight. Fortunately, there are few tip and tricks that may life with mappings and expressions a bit easier.

Most mappings are aware of the context in which they are used. Therefore paths of mapping sources and targets can be shortened - or even left out entirely. Activation and credential mappings used in the HR feed example are the obvious cases. But even paths in ordinary mapping may be shortened. For example take the outbound mapping source:

```

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

```

As the mapping knows that its source is a focus (user) the definition may be shortened:

```

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

```

Typical midPoint deployment has tens or hundreds of mappings. Deployments with thousands of mappings are definitely feasible. Therefore it may not be entirely easy to maintain the mappings. Therefore there are two things that can make this easier. There is an ability to optionally specify mapping name. Mapping name will appear in the log files and some error messages. Therefore it may be easier to identify which mapping is causing problems or it may help locate the trace of mapping execution in the log file. Mapping can also have a description. The description can be used as a general-purpose comment or a documentation of the mapping. The description can be used to explain what the mapping does.

```

<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>

```

Mappings can become quite complex. There may be multi-line scripting expression in the mapping and it may not entirely obvious what is the input and output. Therefore each mapping and each expression have an ability to enable tracing:

```

<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>

```

If tracing is enabled then the mapping or expression execution will be recorded in the log files. Tracing can be enabled at both mapping level and expression level. Mapping tracing is shorter. It provides overview of the mapping inputs and outputs. Expression-level tracing is much more detailed.

However, even this level of tracing may not be enough to debug expression code. Therefore there is a special expression function for logging. Arbitrary messages may be logged by script expression code:

```

<expression>
  <script>
    <code>
      ...
      log.info("Value of foo is {}", foo)
      ...
    </code>
  </script>
</expression>

```

Generally speaking, troubleshooting of mappings may be quite difficult as it is often intertwined with midPoint internal algorithms. But there are ways how to do it. The [Troubleshooting](#) chapter provides much more details on this.

Expression Functions

Expressions in general and scripting expressions in particular are the place where most midPoint customization takes place. Scripting expressions are able to execute any code in a general-purpose programming language. Therefore the script can transform the data in any way or it can execute any function. Quite naturally there are functions that are frequently used in the scripts. Therefore midPoint provides convenient scripting libraries full of useful methods ready to be used in scripting expressions.

There are two scripting libraries that are used very often:

- **Basic script library** provides very basic functions for string operations, object property retrieval, etc. These are simple, efficient stand-alone functions. These functions can be used in every expression.
- **MidPoint script library** provides access to higher-level midPoint functions contain IDM-specific and midPoint-specific logic. This library can be used to access almost all midPoint functionality. But there are few places where this library may not work reliably (e.g. correlation expression).

The libraries are designed to be very easy to use from the scripting code. While the specific details how to invoke the library depend on the scripting language, the libraries are usually accessible by the use of `basic` and `midpoint` symbols. Function `norm()` from the basic library can be invoked in a Groovy script like this:

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

Invocation of the libraries from JavaScript and Python is almost the same and we are sure that a clever reader will have no trouble figuring that out. What is more difficult to figure out is which functions the libraries provide. For that purpose there is a page in midPoint wiki that lists all the libraries and this page also has a link to library function documentation. Look for [Script Expression Functions](#) page in midPoint wiki.

Only two libraries were mentioned in this section so far. However, this is not a whole story. A clever reader has certainly figured out that the logging function described in previous section is also a scripting library. And there may be more libraries in the future.

Resource Capabilities

The systems that midPoint connects to are not created equal. In fact, those system significantly differ in their capabilities. Most systems can create accounts. But not all of them can delete accounts. There are systems that keep the accounts forever, the accounts can be just permanently disabled. And yet another systems cannot enable or disable accounts. While most systems support password authentication, other system do not. There is a lot of natural diversity in the provisioning wilderness. And even the connector may introduce limitations. Even if target system supports a particular feature, connector may not have appropriate code to use it. MidPoint needs to take all these differences into consideration when executing synchronization and provisioning operations.

MidPoint refers to these features of the systems and connectors as *resource capabilities*. Although capabilities may in fact be quite complex, they are essentially just a list of things that a connector

and resource can do. MidPoint is aware of the resource capabilities. Therefore midPoint can work with resource data correctly. E.g. midPoint will not try to modify account on a read-only resource.

Capabilities are usually automatically discovered by midPoint and everything just works out of the box. There is usually no extra work to maintain the capabilities. But sometimes there is a need to tweak the capabilities a bit. Maybe the connector cannot detect resource capabilities well enough. Maybe there is a read-only resource, but the connector has no way of knowing this. Therefore the write capabilities have to be manually disabled in midPoint. For that reason there are two sets of capabilities:

- **Native capabilities** are capabilities detected by the connector. Those are always automatically generated by midPoint. Those capabilities should not be modified by administrator.
- **Configured capabilities** are the capabilities modified by the administrator. Configured capabilities are used to override native capabilities. Configured capabilities are usually empty, which means that only native capabilities are used.

There are many ways how the capabilities can be tweaked by the administrator. But there is one case that is particularly interesting for synchronization and provisioning: simulated activation capability.

MidPoint connectors can be tailored specifically for a particular system. E.g. there are often connectors that are developed specifically for one custom enterprise application. At the other side of the spectrum are generic connectors that can fit a wide variety of systems and applications. LDAP, CSV and database table connectors are examples of such generic connectors. Such connectors are very useful and they are used in almost every midPoint deployment. However, there is no standardized way how to disable an account in database table or a CSV file. Various columns and various values are used to represent account activation status. Quite surprisingly, there is no standardized way how to disable an account in LDAP directory either. But that is a bad news for midPoint. MidPoint takes a significant advantage from knowing whether account is disabled or enabled. We had to do something about it. And we did. There is way how to tell midPoint which attribute and what values are used to represent account activation status. Configured activation capability is used for that purpose:

```
<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>
```

Configured capability above specifies resource attribute `active` as the attribute that controls account activation status. If this attribute is set to value `true` then the account is enabled. If the attribute is set to value `false` then the account is disabled. That is it. Once this configured capability

is part of resource definition then midPoint will pretend that the resource can enable and disable accounts. Attempt to disable account will be transparently translated to modification of attribute active. But it also works the other way around. If an account has attribute `active` set to `false` value midPoint will display that account as disabled. No extra logic or mapping is needed to achieve that. The capability does it all.

Synchronization Example: LDAP Account Correlation

Previous example demonstrated the use of synchronization for HR feed. That is the most obvious use of synchronization mechanisms. However, midPoint synchronization is much more flexible than just feeding data to midPoint. Synchronization can be used even for target resources. In that case the synchronization is usually used for several purposes:

- **Initial migration:** This is a process of connecting new resource to midPoint. There are usually accounts that already exist in the resource at the time when a resource is connected to midPoint. It is likely that at least some of the accounts correspond to the users that are present in midPoint (e.g. users created from the HR feed). Therefore the accounts from the resource need to be correlated to the users that already exist in midPoint. Synchronization is the right mechanism for this.
- **Detection of illegal accounts:** Security policies are usually set up in such a way that only those people that need an account on a particular resource should have that account. This is known as the *principle of least privilege*. However, in typical IDM deployment there is nothing that would prohibit system administrator to create any accounts at will. And this is often desirable because there are emergency situations where full control over the system is crucial. But even for emergency cases, we want to make sure that the situation is aligned with policies when the emergency is over. MidPoint can easily do that by scanning the target systems in regular intervals. Synchronization mechanisms can be used to detect accounts that do not have any legal basis and delete or disable such accounts. Again, synchronization mechanism can do that easily.
- **Attribute value synchronization:** Accounts in target resources are usually created as a result of midPoint provisioning action. However, account attribute values are in fact copies of the data in midPoint. Attribute values can easily be changed by system administrator, may be set to old values during data recovery procedure or they can get out of sync by a variety of other means. MidPoint can make sure that the attributes are synchronized and that they stay synchronized for a long time. Synchronization mechanisms are ideal for this purpose.

Older IDM systems used synchronization mostly to get data from the source resources to IDM system. But synchronization in midPoint is much more powerful. It can be applied to source systems and target systems, it can pull data, push data, detect inconsistencies and fix them. Synchronization is a general purpose mechanism. This is the principle of reuse again. Synchronization mechanism can be reused for variety of purposes.

In this example we will be using synchronization to connect existing LDAP server to midPoint. We assume that our midPoint is already connected to the HR system. We have imported the HR data. Now we have midPoint users created for all our employees. And then there is this LDAP server. It is really important LDAP server. This server is used by company intranet portal and also by a variety of smaller web applications. Those applications are using the LDAP server for user authentication and access authorization. The LDAP server was deployed few years ago. Initially it was populated

by the HR data. But the LDAP server was managed manually by a system administrator during all these years. Therefore it is expected that there will be some accounts that belong to former employees. Also, it might have happened that some accounts are missing. And it is quite likely that a lot of the accounts have wrong data.

First task is to set up the connector for this resource. As LDAP servers are used for identity management purpose all the time, MidPoint comes with a really good LDAP connector. All we need is to set up the resource to use that connector:

```
<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>
    ...

```

What we can see here is a slightly more sophisticated method to reference the connector. So far we have seen only a direct reference by OID. This works well for almost all the references in midPoint because OID never changes. But connectors are a bit tricky. Objects that represent connectors are dynamically created by midPoint when a connector is discovered. Therefore the OID is generated at random when midPoint starts. There is no practical way how a system administrator can predict that OID. But we still want our resource definitions to refer to a particular connector when we import the definition. Therefore there is an alternative way how to specify object references. This method is using a search filter instead of direct OID reference. When this resource definition is imported to midPoint then midPoint will use that filter and look for LDAP connector. If that connector is found then the OID of that connector is placed in the reference (`connectorRef`). Therefore the next time midPoint will be using this resource it can follow the OID directly. This is a very convenient method. But there are few limitations. Firstly, the filter is resolved only during import. That means it is resolved only once. If the connector is not present at import time then the reference needs to be corrected manually. Secondly, this approach works if there is only one LDAP connector deployed to midPoint. This is usually the case. But the connector framework can contain several connectors of the same type in different versions. This is a very useful feature for gradual connector upgrades, testing of new connector versions and so on. But in case that the filter matches more than one object the import will fail. In that case the connector reference has to be set up manually.

Once we have proper reference to LDAP connector we need to configure the connection:

```

...
<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:enableAttributesToGetSearchResultsHandler>
    </icfc:resultsHandlerConfiguration>
</connectorConfiguration>
...

```

This is all very similar to the configuration of the other resource that were already presented in this book. It should be quite self-explanatory – except perhaps for the configuration of result handlers. *Result handlers* are little helpers that come with the ConnId connector framework. The purpose of the result handlers is to assist simpler connectors in filtering and post-processing search results. But LDAP connector is no ordinary simple connector. LDAP connector is mature and full-featured connector that can do everything without any help from such annoying little creatures as those result handlers. ConnId result handlers do not add any value here. In fact they may even be harmful. LDAP protocol has a lot of peculiarities such as case-insensitivity that applies to almost all the aspects of LDAP data – except for some notable exceptions. The connector is aware of those peculiarities but the handlers are not. Therefore if the handlers are turned on (which is the default) they may get in the way and ruin the data. Therefore it is always strongly recommended to explicitly turn off the handler when a full-featured connector is used.

 The XML example above, as all other examples in this book, is simplified and shortened for clarity. You will not be able to import the example in this form into midPoint. For a full importable examples see the files that are supposed to accompany this book. Please see [Additional Information](#) chapter.

The basic resource configuration above is sufficient to connect to the resource. Therefore the test connection operation on resource details page should be successful. This configuration may also be used to list the accounts. However, LDAP servers support many object classes and midPoint does not yet know which object class represents account. Therefore we need to add schema handling section to our resource:

```

...
<schemaHandling>
  <objectType>
    <kind>account</kind>
    <display Name>Normal Account</display Name>
    <default>true</default>
    <objectClass>ri:inetOrgPerson</objectClass>
    <attribute>
      <ref>ri:dn</ref>
      <display Name>Distinguished Name</display Name>
      <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>
  ...

```

There should be outbound mapping for each mandatory LDAP attribute for the `inetOrgPerson` object class. Those mapping are very typical for a target resource definition.

Once we set up the schema handling we should be able to conveniently list LDAP accounts in midPoint. However, we need to switch to the *Resource* view instead of *Repository* view. The accounts are stored in the LDAP server and midPoint can access them. Therefore the accounts are listed in the *Resource* view. But midPoint have not processed the accounts yet. Therefore there are no account shadows in midPoint repository. And that is the reason that the *Repository* view is empty. But now we are going to do something about it.

We are going to import (or reconcile) the resource accounts. But if we try to do this now nothing would really happen. The accounts are not linked to users therefore midPoint will not synchronize the attributes. And midPoint was not told to do anything with the accounts. Therefore midPoint will do nothing. That is one of midPoint principles: midPoint will not change the accounts in any way unless it is explicitly told to do so. Default midPoint configuration is to do nothing. We would rather do nothing than to destroy the data.

Before we can import the accounts we need to set up the synchronization configuration for this resource. There are accounts in the LDAP server that should belong to users that already exist in midPoint. We want to link them. But we do not want to do the linking manually. We would rather

set up a correlation expression that does this automatically:

```
...
<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>$projection/attributes/employeeNumber</path>
        </expression>
      </q:equal>
    </correlation>
  ...

```

This correlation expression is going to match account attribute `employeeNumber` and user property that is also named `employeeNumber`. Simply speaking: if account and user employee numbers match then we assume that they should be linked. In that case midPoint decides that synchronization situation is unlinked (they should be linked, but they are not yet linked). We want midPoint to link the account in this case, therefore we define appropriate reaction:

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

```

This will take care of accounts for whose we can find an owner. But what to do with other accounts? We will do nothing about that yet. Therefore we do not need to define any other reactions. This may be somehow surprising. We do not want illegal accounts, do we? Then perhaps we would like to see a reaction to delete unmatched accounts, right? That would be a good approach, but it is just too early for this. We do not want to delete unmatched account just now. There may be accounts that are perfectly legal, just the `employeeNumber` attribute is missing or mistyped. Data errors like those happen all the time, especially when the data were managed manually. We do not want to over-react and start deleting accounts too early. Therefore we will go just with this one synchronization reaction for now.

Now it is the right time to start import or reconciliation task. After the task is finished the situation may look like this:

The screenshot shows the OpenLDAP interface with the title "OpenLDAP". The top navigation bar includes tabs for "Details", "Defined Tasks", "Accounts" (which is selected), "Entitlements", "Generics", "Uncategorized", and "Connector". Below the tabs, there is a search bar with the placeholder "(Object Class: inetOrgPerson)" and buttons for "Search In:" (Repository, Resource), "Up", and "Advanced".

Summary

State	Count	State	Count	State	Count	State	Count	State	Count	State	Count	Total
Deleted	0	Linked	24	Unmatched	1	Disputed	0	Unlinked	1	Nothing	0	26

Accounts

	Name	Identifiers	Situation	Intent	Owner	Pending operations
<input type="checkbox"/>	uid=anderson,ou=people,dc=example,dc=com	entryUUID: 1fb48d1c-b5d4-1039-9f29-45b0e1394789 dn: uid=anderson,ou=people,dc=example,dc=com	LINKED	default	001	<input type="button" value="More..."/>
<input type="checkbox"/>	uid=brown,ou=people,dc=example,dc=com	entryUUID: 1fb5cb82-b5d4-1039-9f2a-45b0e1394789 dn: uid=brown,ou=people,dc=example,dc=com	LINKED	default	002	<input type="button" value="More..."/>
<input type="checkbox"/>	uid=carol,ou=people,dc=example,dc=com	entryUUID: 1fb68086-b5d4-1039-9f2b-45b0e1394789 dn: uid=carol,ou=people,dc=example,dc=com	LINKED	default	003	<input type="button" value="More..."/>
<input type="checkbox"/>	uid=irvine,ou=people,dc=example,dc=com	entryUUID: 76cd1c4e-b5d5-1039-9f4b-45b0e1394789 dn: uid=irvine,ou=people,dc=example,dc=com	UNMATCHED	default		<input type="button" value="More..."/>

It looks like we had quite a good data in the LDAP server. Most of the accounts were successfully correlated and linked to their owners. But there are few accounts that were not correlated. Those accounts ended up in unmatched situation. You can resolve this situation by manually linking the unmatched accounts to their users. Simply click on the small triangle button next to the unmatched entry and select *Change owner* from the context menu. Then select the right user (Isabella Irvine) in the dialog that appears. After that the account is linked to the user. Repeat this process to link all unmatched accounts.

There is one interesting thing in the screenshot above. Have a look at the LDAP account identified by **uid=carol**. While most other accounts have their uid value taken from the surname of the user, this account is an exception. Even though the uid is obviously wrong, midPoint have linked the account correctly to the user (Carol Cooper). The reason is that we have set up midPoint to use employeeNumber for correlation. The result is that even accounts whose usernames violate the convention can be automatically linked to their owners - as long as there is any reliable piece of information that can be used for correlation.

When all the accounts are all linked to their owners it is the right time to complete the synchronization policy. Now we can tell midPoint to delete any unmatched account. That is the case

when an illegal account is created in LDAP server. We can also tell midPoint to unlink any account that was deleted in LDAP server:

```
...
<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>
...

```

There may be some accounts in the LDAP server that have wrong attribute values. By "wrong" we mean that the attributes have different values than the values that are computed by the outbound mappings. But midPoint will not correct those values just yet. Remember the midPoint principle that it will not change the account unless we have explicitly told to do so? Those accounts are in the **linked** situation. And we have not configured any reaction for this situation. Therefore now we need to tell midPoint to synchronize the values:

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

A clever reader is now surely wondering whether we have forgotten something. And indeed we have. Attribute values are synchronized by running reconciliation process. But our outbound mappings will not work in reconciliation. They do not have any explicit definition of strength, therefore midPoint assumes **normal** strength. Those mappings are supposed to implement the *last change wins* strategy. Therefore reconciliation cannot overwrite the account data as midPoint does not know whether it was account attribute or user property that was the last to change. If midPoint is not sure about something then it will do nothing. We do not want to destroy the data. Therefore what we need to do now is to let midPoint know that we really mean it, that the mappings are really **strong**:

```

...
<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>
...

```

Clever reader is uneasy once again. What is this `limitations` thing here? Simply speaking, the limitations specify that the attribute is optional (`minOccurs=0`) and that it is single-valued (`maxOccurs=1`). But, isn't midPoint supposed to be completely schema-aware and figure that all by itself? Yes, it is. And in fact, that is the reason why we need to override the information from the schema using this `limitations` element here. The `cn` attribute is specified in LDAP schema as a mandatory attribute. However, we have just specified outbound mapping for that attribute. Therefore even if midPoint user does not provide any value for attribute `cn`, we can still determine that value by using the expression. Therefore even though LDAP schema specifies attribute `cn` as mandatory, we want to present that attribute as optional in midPoint. Hence the `minOccurs` limitation. And the `maxOccurs` limitation is immediately obvious to anyone who is intimately familiar with LDAP peculiarities. In the LDAP world, almost everything is multi-valued by default. Therefore even commonly used attributes for account identifiers and names are multi-valued. Nobody is really using them as multi-valued attributes because vast majority of applications will probably explode if they ever encounter two values in the `cn` attribute. But those attributes are formally defined as multi-valued in LDAP schema and that is what midPoint gets from LDAP connector. The `maxOccurs` limitation is overriding the schema and forcing midPoint to handle this attribute as if it was single-value attribute.

That is all. Now you can schedule reconciliation tasks to keep an eye on the LDAP server. The task will correct any attribute values that step out of line and delete any illegal accounts. This is how synchronization tasks can be useful even in case of pure target resources.

However, there is one last word of warning. Those accounts were synchronized and linked to existing midPoint users. The accounts were not created by midPoint. Therefore there is nothing in midPoint that would say that those accounts should exits. In midPoint parlance there are no `assignments` for those accounts. MidPoint makes clear distinction between policy and reality. Therefore midPoint is aware that those accounts exist, but there is no policy statement that would justify their existence. By default midPoint does nothing and it will let the accounts live. The accounts will be created or deleted only if there is an explicit change in the assignments. There is no such change now, therefore the accounts are not deleted. But this is a fragile situation. Accounts that are linked but not assigned can easily get deleted if midPoint administrator is not careful. Of

course, there are methods to handle such situations. One way would be to create the assignments together with the links. Those that are interested in this method should look up keyword "legalize" in midPoint wiki. But there are much better methods how to handle this. Perhaps the best approach would be to utilize the roles (RBAC). Which is the topic of the [Role-Based Access Control](#) chapter below. But there are still more things to learn about synchronization until we get there.

Reconciliation

Reconciliation is a process of comparing current state of an account (reality) to a desired state of the account (policy). Reconciliation does not only compare the accounts, it is fixing the inconsistencies. Therefore reconciliation can correct wrong data on resources. But it also works the other way. It can correct the data in midPoint. Therefore reconciliation is one the most useful tools in the identity management toolbox.

Reconciliation can be used in a variety of ways. Reconciliation can be initiated for one specific user by using midPoint user interface. In that case midPoint compares the values of all user's accounts to the values that were computed using the mappings. If there is difference midPoint corrects account values. This approach is perfect for testing reconciliation setting on just a single user. This feature is also useful for fixing values of one specific user.

Reconciliation of a specific user may be useful, but it is an ad-hoc approach. We usually favor systemic approaches in identity management. Therefore reconciliation can be used in a form of a reconciliation task. Reconciliation task lists all the accounts on the resource and then it reconciles each account, one by one. This is a way how to keep all resource accounts continuously synchronized.

There is a couple of things about reconciliation that can be somehow surprising. Firstly, reconciliation of an account may cause modification of a user. This happens if there are inbound mappings for that account. This is perhaps quite expected. But if user is changed then such change may propagate to other accounts on other resources, usually by the means of outbound mapping. MidPoint does not like procrastination and therefore it will try to execute those changes immediately. But it means that reconciliation of one account may cause changes to other accounts. Which makes a lot of sense, yet it may be quite surprising. Secondly, reconciliation will skip any normal-strength mappings. We have already explained the reasons for that, but this is something that can surprise even an experienced midPoint engineer from time to time. If we are sure that we want the mapped value to be present in the account all the time then strong mappings are the way to go.

A curious reader that has already explored midPoint user interface has surely noticed *recompute* function. What recompute does looks almost exactly the same as reconciliation. But there are subtle differences. Recompute will not force the fetch of account data. In this case the account attributes will be fetched from the resource only if midPoint inevitably needs them for the computation. This usually happens if **weak** or **strong** mappings are used. But if there are **normal** mappings only then recompute may not read account data. MidPoint will compare and correct account attribute values only for those accounts that are fetched from resource during this process. That is how recompute works. The purpose of a recompute is to correct data of midPoint users, which means evaluation of object templates and other policies. Correcting account data is more or less just a side effect of a recompute. On the other hand, reconciliation always tries to read all the accounts regardless whether they are needed for computation or not. Therefore all the attributes on all the accounts are

fixed. That is the purpose of reconciliation: correct the account data.

There is yet another difference between recompute and reconcile *tasks*. The purpose of a recompute task is to correct user data. Therefore recompute task will iterate over midPoint users. Recompute task will not detect new accounts on the resource and it may even overlook if an account is deleted. But reconciliation task is different. In fact reconciliation task has several stages. Main reconciliation stage lists all resource accounts. It determines owner of each account, compare the attributes and correct them. But as this process iterates over real accounts on a resource, it can also detect new accounts. When the main stage is completed then the next phase is looks at account shadows stored in midPoint. The task looks for shadows that have not been processed in the main phase. Those are accounts that used to be on the resource some time ago but that have disappeared. That is how reconciliation detects deleted accounts.

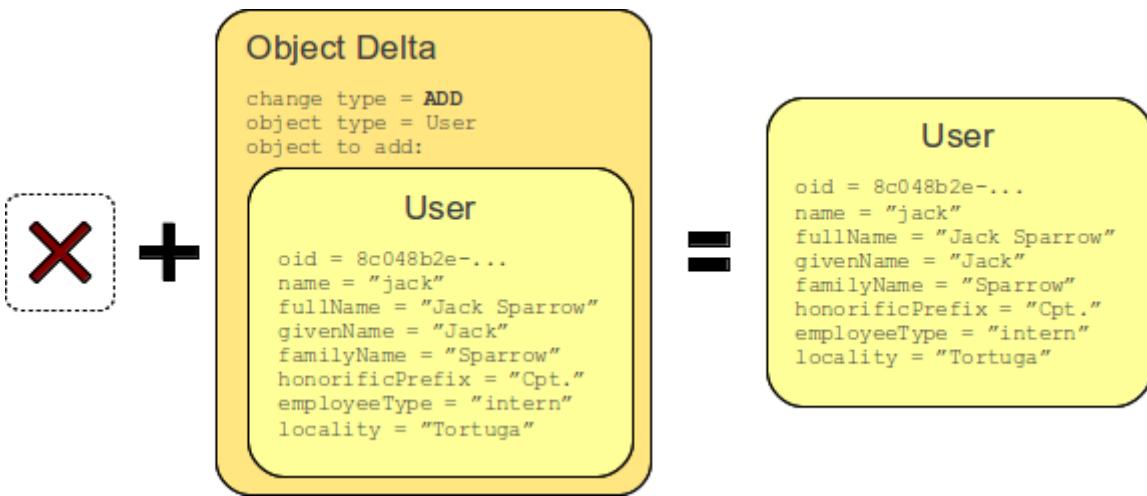
Deltas

Reconciliation is really useful mechanism. It is reliable and thorough, but it is also quite slow and it consumes a lot of computational and network resources. And there are reasons why reconciliation is such a heavyweight beast. Reconciliation works with *absolute* state of accounts. It means that reconciliation is reading all the accounts with all the values of all the attributes. Then it recomputes everything. Even those attributes and values that were not changed are recomputed. This is a very traditional and reliable way of computation and that is also the way how most older identity management systems work.

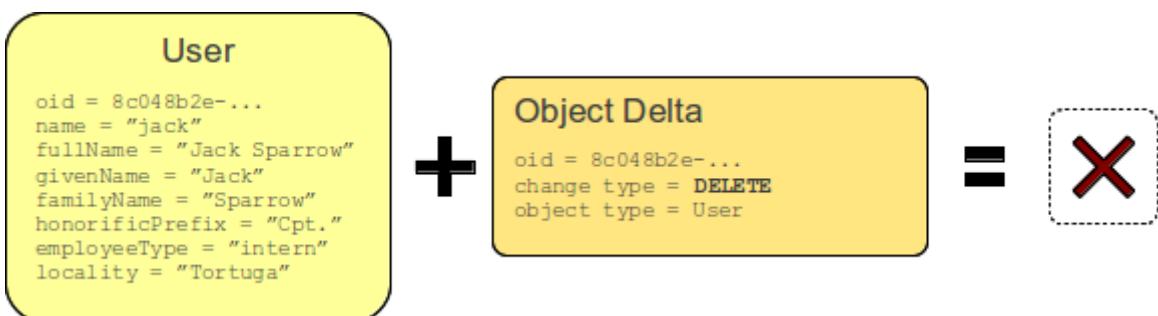
But there is also a better way. If we know that just one attribute was changed we can recompute that single attribute only. We do not need to care about other attributes. And if we know that attribute foo has changed in such a way, that there is a new value bar then it gets even better. We just need to recompute the value bar and do not care about any other values. This is what we like to call a *relative change*. We just care about the values that were changed. That is how midPoint works internally. We could say that MidPoint is *relativistic*.

This is where *delta* comes in. Delta is a data structure that describes the change of a single midPoint object. *Add* delta describes a new midPoint object that is about to be created. *Modify* delta describes existing midPoint object where some properties have changed. *Delete* delta describes an object that is going to be deleted. This is a very powerful mechanism. Just remember that everything in midPoint can be represented as an object: user, account, resource, role, security policy ... everything. Therefore delta can represent any change. It may be a change of user password, deletion of an account, change of connector configuration or introduction of a new password policy. If all the changes can be represented in a uniform way then they can also be handled in a uniform way. Therefore it is easy for midPoint to record all the changes in an audit trail – including configuration changes. It is easy to route any change through an approval process. And so on. MidPoint can create a relatively simple mechanisms to handle changes and then those mechanisms can be applied to changes of (almost) any object.

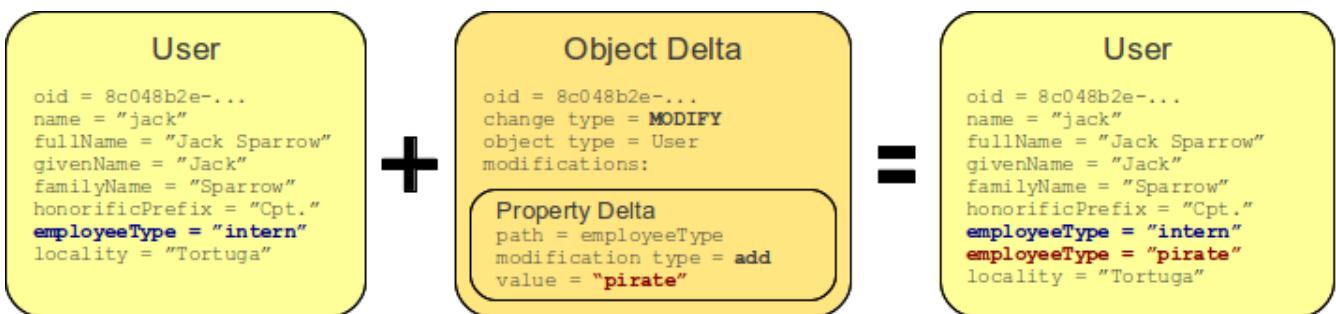
Let's have a closer look at an anatomy of a delta. There are three types of delta: *add*, *modify* and *delete*. *Add delta* is quite simple. It contains a new object to be created.



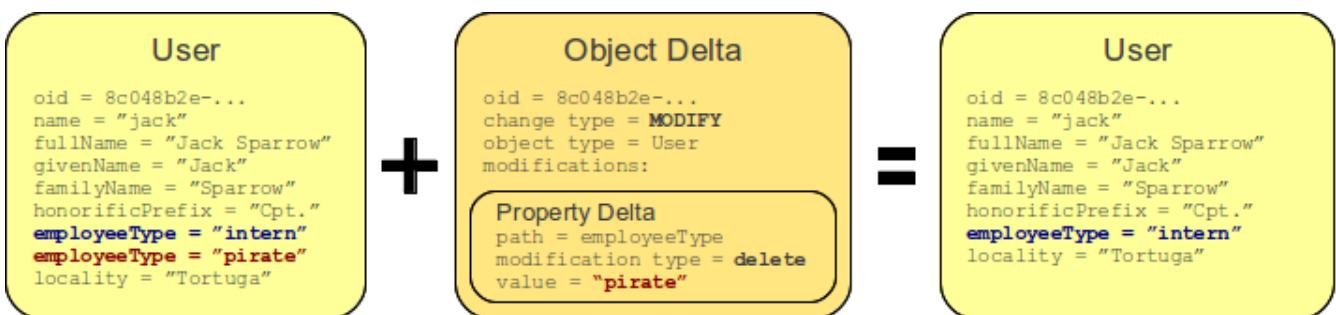
Delete delta is even simpler. It contains just object identifier (OID) of an object to be deleted.



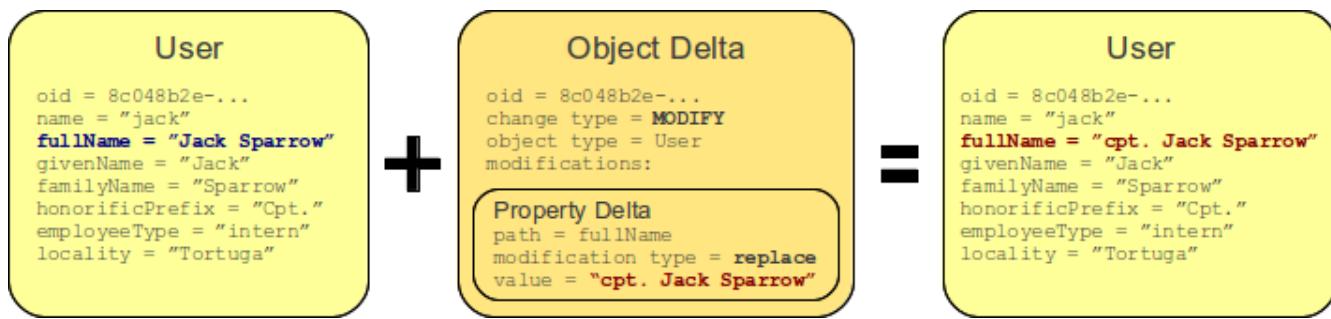
Last one is *modify delta*. This delta contains a description of modified properties of an existing object. But as the object can change in a variety of ways the modify delta is the most complex of the tree. Modify delta contains a list of *item deltas*. Each item delta describes how a particular part of an object changes. For example following delta describes that a new value **pirate** is added to a user property **employeeType**.



The item delta may have three modification types: *add*, *delete* and *replace*. *Add modification* means that new value or values are added to an item. *Delete modification* means that value or values are removed from an item.



In both *add* and *delete* cases the values that are not mentioned in the delta are not affected. However, *replace* modification is different. This means that all existing values of the item are going to be discarded and they are replaced with the value or values from the delta.



The deltas are designed to work with both single-valued and multi-valued items. In fact *add modification* and *delete modification* deltas are specifically designed with multi-value items in mind. Those deltas can work efficiently even in cases that there is a multi-valued attribute that has a very large number of values. And there is a good reason for this. Multi-valued properties are quite common in the identity management field. Just think about how roles, groups, privileges and access control lists are usually implemented. Everybody that ever managed a large group in LDAP server will surely remember that experience in vivid colors. But midPoint is designed to handle situations like those.

Everything in midPoint is designed to work with deltas: user interface, mappings, authorizations, auditing ... all the way down to the data storage components. Mappings are designed in a relativistic ways. That is the reason why we need to explicitly specify sources of the mapping. Mapping source definitions are matched with items in the delta to control execution of the mapping. Deltas permeate entire midPoint computation. Deltas are input to the mappings, but mapping produce other deltas as output. Therefore we can have a complete chain: deltas that are result of inbound mappings is applied to the user object, but those deltas are also input to outbound mappings. Everything is *relativistic* in midPoint.

This might seem to be a bit over-complicated at the beginning. But do not worry. You will get used to it. And clearly, this approach has major advantages. But a clever reader does not seems to be impressed. How can this relativistic approach conserve any significant portion of computational resources? We usually fetch the entire account from the resource anyway. Therefore there is no harm to recompute all the attributes. The computation itself is fast, it is the fetch operation that is slow. Isn't it? The clever reader is, of course, right. Or partially right at the very least. Most resources really fetch all the account attributes in a single efficient operation. And for those cases there is no big increase in efficiency if we go with the relativistic methods. But there are exceptions. For example some resources will not return all the values of big attributes, e.g. all the members of a large group. Additional requests are needed to fetch all the values – and there may be a lot of requests if the group is really large. Relativistic approach has significant benefit in those cases. And the benefits will be even more obvious when we get to the live synchronization in the next section. But performance is not the primary motivation for the relativistic approach. There is one extremely strong reason to go relativistic: data consistency. Consistency is something that brings ugly nightmares to many engineers that try to design distributed system. And identity management solution is in fact a distributed system. But it is a very loosely-coupled distributed system. There is no support for locking or transactions in the connector. And even if there was some support, vast majority of resource cannot provide those consistency mechanisms on their identity management

APIs. This means that midPoint cannot rely on traditional consistency mechanisms. And that is why relativistic approach is so useful. Relativistic computation has a very high probability of achieving correct result even without locking or transactions. This is more than acceptable for typical identity management deployments. And for those rare cases where relativistic computation can fluctuate there is always reconciliation as a last resort. But thanks to the relativistic nature of midPoint the need for reconciliation is significantly reduced.

That was a lot of long words, but clever reader seems to be satisfied now. At least for a while. But there is quite a simple summary: relativistic approach of midPoint can do miracles. For example, midPoint resource can be both sources and targets, even a single attribute can be both source and target of information. It is the relativistic approach that allows features like this. The principle of relativity is relatively simple. But its effect in midPoint is nothing short of being revolutionary.

Live Synchronization

MidPoint has a range of synchronization mechanisms. Slow, brutal but reliable reconciliation is at one end. Live synchronization is on the other. Live synchronization is a lightweight mechanism that can provide almost-real-time synchronization capabilities. Live synchronization is specifically looking for recent changes on a resource. When such changes are detected, live synchronization mechanisms process those changes immediately. The synchronization delay is usually in order of seconds or minutes if live synchronization is used properly.

Unlike reconciliation, live synchronization is not triggered manually. That would make very little sense. Live synchronization works in a long-running task repeatedly looking for fresh changes in short time intervals. If a resource is configured for synchronization then all that is needed to run live synchronization is to set up a live synchronization task. MidPoint user interface can be used to do that easily. And an example of live synchronization task was provided in the HR feed section above.

Live synchronization task wakes up at regular intervals. Each time the task wakes up it invokes the connector. Connectors that are capable of live synchronization have special operation that is used to get fresh changes from the resource. The connector can support any reasonable change detection mechanism – in theory. But two mechanisms are commonly used in practice:

- **Timestamp-based synchronization:** Resource keeps track of last modification timestamp for each account. The connector looks for all accounts that have been modified since last scan. This is very simple and relatively efficient method. But it has one major limitation: it cannot detect deleted accounts. If an account is deleted then there is no timestamp for that account and therefore the connector will not find it in the live synchronization scan.
- **Changelog-based synchronization:** Resource keeps a “log” of recent changes. The connector is looking at the log and it is processing all the changes that were added to the log since the last scan. This is a very efficient and flexible method. But it is not simple. And not many systems support it.

All live synchronization methods need to keep the track of what changes are “recent”, i.e. which changes were already processed by midPoint and which were not processed yet. There is usually some value that needs to be remembered by midPoint: timestamp of last scan, last sequence number in the change log, serial number of last processed change and so on. Each connector has a different value with a connector-specific meaning. MidPoint refers to those values as “tokens”. The

most recent token is stored in the live synchronization task. That is how midPoint keeps track of processed changes. There are (hopeful quite rare) cases when resource and midPoint token get out of alignment. This may happen in cases such as the resource database is restored from a backup, if network time gets out of synchronization and so on. If that happen then deleting the token from the live synchronization task is usually all it takes to get the synchronization running again.

Live synchronization is fast and very efficient. But it is not entirely reliable. MidPoint may miss some changes. This is quite a rare situation, but it may happen. Reconciliation will surely remedy the situation in such a case. Just remember, all the synchronization mechanism share the same configuration. And it is perfectly acceptable to run live synchronization and reconciliation on the same resource at the same time. But of course, it would be a good idea to run reconciliation less frequently than live synchronization.

Conclusion

Synchronization is one of the most important mechanisms in the entire identity management field. Primary purpose of synchronization is to get the data into midPoint. And that is really good approach when an identity management deployment begins: get your data into midPoint first. Get the data from the HR system. Correlate that with Active Directory. Connect all the major resources to midPoint and correlate the data. MidPoint does not need to make any changes at this stage. In fact it is perfectly good approach to make all the resource read-only at this stage. The point is to let midPoint see the data. But why do we need that?

- We will see what is the real quality of the data. Most system owners have at least some idea what data sets are there. But it is almost impossible to estimate data quality until the data are processed and verified. That is exactly what midPoint can do at this stage. This is essential information to plan data cleanup and sanitation.
- We will learn how many accounts and account types are there. It is perhaps quite obvious that there are employee accounts. But are there accounts for contractors, suppliers, support engineers? Are those accounts active? What is the naming convention? Do system administrators use employee accounts for administration. Or are they using dedicated high-privilege accounts? This information is crucial to set up provisioning policies.
- We will learn distribution of accounts and their entitlements. Do all employees have accounts in Active Directory? Are there any frequently-used groups? How does organizational structure influence the accounts? This information is very useful to design a role-based access control structures and other policies.
- We will surely learn some security vulnerabilities. Are there orphaned accounts that should have been deleted long time ago? Are there testing accounts that were left unattended after the last night-time emergency? Indeed, there is no security without identity management.

This is a good start. But even if this is all that you do in the first step of the deployment it is still a major benefit. You will get better visibility and with that comes better security. And you have the data to analyze your environment and plan next step of the identity management deployment. You won't be blind any longer. And that is really important. It is indeed a capital mistake to theorize before one has data.

Chapter 6. Schema

If you have built castles in the air, your work need not be lost, that is where they should be. Now put the foundations under them.

— Henry David Thoreau

So far we have been discussing the things that influence how midPoint interacts with the outside. Resource definitions, outbound and inbound mappings, even the roles - the primary purpose of those things is to control how data get into midPoint and out of midPoint. But now it is time to discuss how midPoint works *internally*.

Early IDM systems were little more than smart data transformers. They took data from data sources, modified them in some way, maybe applied a model such as RBAC and then pushed the data out. There was very little crucial information that was stored inside the IDM system itself. But that was a long time ago and the world is a different place now. The focus of IDM field has shifted towards identity governance. It is not enough to transform the data. Policies need to be applied. Compliance needs to be evaluated. There are processes to follow, paperwork to do, reports to compile, notifications, reviews and daily status reports. It is perhaps no big surprise that there is a good deal of *management* in Identity Management.

Many of the chapters that follow will deal with these management concepts. But we have to start from the basics. And the goal of this chapter is an explanation of the very foundation of midPoint: schema. The goal of midPoint is not just a mere data transformation. The goal is to *unify* the data. And midPoint schema plays a crucial part in that ambition.

MidPoint Schema

MidPoint is designed as a schema-aware system. For every bit of data that passes through midPoint we have a complete *definition*. We know whether this is string, integer or timestamp. We know whether it is single-valued or multi-valued. We know whether it is optional or mandatory. We know whether this is a sensitive piece of data that requires extra protection. We know whether it is part of technical meta-data that we usually do not want to show by default. And often we also know what label we should use when we are presenting the data and how that label translates to other languages. We know quite a lot about the data that we work with. All the objects that midPoint works with are completely defined by the schema. There is a schema for user, role, org, resource, system configuration and everything else.

Such awareness of the schema brings significant advantages to midPoint. The most obvious advantage is in data presentation. We know that we need to render a calendar selector because that particular data property is timestamp. We know that we need to render a text field with a plus button to add values because that particular property is a multi-valued string. And we know that some fields should be disabled because those properties are read-only. This behavior is not hard-coded. Vast majority of midPoint user interface is rendered by interpreting midPoint schema.

This approach is absolutely crucial for any serious IDM system. One of the reasons is that the IDM system works with data that are retrieved from other systems (resources). It is just not possible to hard-code midPoint user interface for all the various attributes that all the possible resources could

have. A different strategy is needed here, a strategy that is much more dynamic.

When midPoint connects to a new resource for the first time it attempts to retrieve *resource schema*. The schema specifies what object classes the resource supports, which attributes the object classes have, what types are those and so on. MidPoint transforms this schema to its own native format and stores that in the resource definition. This means that midPoint has the schema available anytime it is needed for dynamic interpretation. That schema is used to display resource data in the most natural and user-friendly way. It is also used by automatic data type conversions, which makes mapping configuration easier.

Data Unification

MidPoint schema is not just a nice way how to create user, role or organizational structure. It has a much deeper meaning. The primary purpose of a schema is integration, data translation and unification. A clever reader would certainly remember that we have already talked about *star topology* or *hub-and-spoke* integration pattern. MidPoint is like a hub of the wheel and all the resources connect to midPoint as spokes. MidPoint is actively discouraging direct resource-to-resource communication. Everything in midPoint is built for resource-to-midPoint and midPoint-to-resource communication. MidPoint is always the center – and for a good reason. All resource data need to be translated to and from a midPoint “data language”. Thus midPoint creates a common language that everybody can understand. And this is exactly the purpose of midPoint schema. The schema of user, role, org and service is designed to contain properties that are often used in identity-related integration scenarios. Therefore, an engineer who is designing a mapping is quite likely to find a suitable property in midPoint schema that is prepared to be used.

MidPoint schema forms a *lingua franca*, a common language that can be translating to various data dialects used by the resources. But it also provides a basic framework that can be reused for many midPoint deployments. Therefore, an engineer starting a new deployment does not need to start on a completely green field. The basic schema will always be there to provide a starting point.



Ever wondered why midPoint is called midPoint? Clever reader would have figured that out already.

Basic User Schema

When it comes to identity management field, there is one concept that is at the center of everything: a concept of *user*. MidPoint is no exception. User is undoubtedly the most important object in the entire midPoint schema. Therefore it is worth to have a closer look at how this object looks like. This is going to be a really educative lesson, as it will explain several fundamental principles of midPoint.

User is represented by schema datatype identified as [UserType](#). Adding the Type suffix to datatypes is a common convention in midPoint. There is [UserType](#), [RoleType](#), [OrgType](#), [ResourceType](#) and so on. This convention is partially historic, partially given by XML Schema conventions, partially a convenience to developers. Regardless of the origins, this convention is used for all the data types in midPoint schemas. You will get used to it eventually.

[UserType](#) is what we call an *object definition* in midPoint parlance. This means that [UserType](#) data

structure specifies a complete midPoint object with all the things that any self-respecting object needs. There is object identifier (OID), name that can be presented in different forms and languages, free-form description and so on. All midPoint objects have those things.

The [UserType](#) data structure has many additional *properties*, *containers* and *references*. Property is a primitive data item such as string, integer or a timestamp. Container is a complex data structure that contains a bunch of properties or other containers. Reference is a pointer to another midPoint object.



Properties are primitive. However, there may be properties that have internal structure, even quite a complex internal structure. This is sometimes given by historic reasons. But there are also properties that need to be complex, e.g. properties that require localizable presentation or properties that provide protection of data. Yes, this may be confusing. And even a clever reader is officially puzzled now. However, this distinction is not a big issue for now.

Definition of [UserType](#) is summarized in the following table:

Name	Type	Description
<code>name</code>	property	Human-readable, mutable name of the object. It is typically a username or some kind of application-level identifier. The value must be unique among all the users. Example: <code>jrandom</code>
<code>description</code>	property	Free-form textual description of the object. This is meant to be displayed in the user interface. Example: <code>Random account for testing.</code>
<code>extension</code>	container	A container for custom schema extensions. We will discuss that later.
<code>metadata</code>	container	Meta-data about object creation, modification, etc.
<code>lifecycleState</code>	property	Lifecycle state of the object. This property defines whether the object represents a draft, proposed definition, whether it is active, deprecated, and so on. Example: <code>active</code>

Name	Type	Description
assignment	container	<p>Set of object's assignments. Assignments define the privileges and "features" that this object should have, that this object is entitled to. Typical assignment will point to a role or define a construction of an account.</p> <p>Assignments represent what the object should have. The assignments represent a policy, a desired state of things.</p>
linkRef	reference	<p>Set of shadows (projections) linked to this focal object. E.g. a set of accounts linked to a user. This is the set of shadows that belongs to the focal object in a sense that these shadows represents the focal object on the resource. E.g. The set of accounts that represent the same midPoint user (the same physical person, they are "analogous").</p> <p>Links define what the object has. The links reflect real state of things.</p>
activation	container	Type that defines activation properties. Determines whether something is active (and working) or inactive (e.g. disabled).
jpegPhoto	property	Photo of a user (in a binary form).
costCenter	property	The name, identifier or code of the cost center to which the user belongs.
locality	property	Primary locality of the user, the place where the user usually works, the country, city or building that he belongs to. The specific meaning and form of this property is deployment-specific.

Name	Type	Description
preferredLanguage	property	Indicates user's preferred language, usually for the purpose of localizing user interfaces. The format is IETF language tag defined in BCP 47, where underscore is used as a subtag separator. This is usually a ISO 639-1 two-letter language code optionally followed by ISO 3166-1 two letter country code separated by underscore. Example: en_US
locale	property	Defines user's preference in displaying currency, dates and other items related to location and culture. It has the same format as preferredLanguage . Example: en_US
timezone	property	User's preferred timezone. It is specified in the "tz database" (a.k.a "Olson") format. Example: Europe/Bratislava
emailAddress	property	E-Mail address of the user, org. unit, etc. This is the address supposed to be used for communication with the user. Example: random@example.com
telephoneNumber	property	Primary telephone number of the user. Example: +421 123 456 789
fullName	property	Full name of the user with all the decorations, middle name initials, honorific title and any other structure that is usual in the cultural environment that the system operates in. This element is intended to be displayed to a common user of the system. Example: James W. Random, PhD.

Name	Type	Description
givenName	property	Given name of the user. It is usually the first name of the user, but the order of names may differ in various cultural environments. This element will always contain the name that was given to the user at birth or was chosen by the user. Example: James
familyName	property	Family name of the user. It is usually the last name of the user, but the order of names may differ in various cultural environments. This element will always contain the name that was inherited from the family or was assigned to a user by some other means. Example: Random
additionalName	property	Middle name, patronymic, matronymic or any other name of a person. It is usually the middle component of the name, however that may be culture-dependent. Example: Walker
nickName	property	Familiar or otherwise informal way to address a person. Example: Randy
honorificPrefix	property	Honorific titles that go before the name. Example: Sir
honorificSuffix	property	Honorific titles that go after the name. Example: PhD.
title	property	User's title defining a work position or a primary role in the organization. Example: CEO
employeeNumber	property	Unique, business-oriented identifier of the employee. Typically used as correlation identifier and for auditing purposes. Should be immutable, but the specific properties and usage are deployment-specific.

Name	Type	Description
organization	property	Name or (preferably) immutable identifier of organization that the user belongs to. The format is deployment-specific. This property together with organizationalUnit may be used to provide easy-to-use data about organizational membership of the user.
organizationalUnit	property	Name or (preferably) immutable identifier of organizational unit that the user belongs to. The format is deployment-specific. This property together with organization may be used to provide easy-to-use data about organizational membership of the user.
credentials	container	The set of user's credentials (such as passwords).

This is a basic outline of the schema for [UserType](#). This description is slightly simplified. Not all the items that are defined for [UserType](#) are shown in the table above. Deprecated items are not shown at all. Only some operational properties are shown. Some items are simplified or entirely omitted for clarity.

Following example illustrates the use of midPoint [UserType](#) schema:

```

<user>
    <name>alice</name>
    <activation>
        <administrativeStatus>enabled</administrativeStatus>
    </activation>
    <preferredLanguage>en_US</preferredLanguage>
    <assignment>
        <targetRef oid="aaa6cde4-0471-11e9-9b50-c743da469067" type="RoleType"/>
    </assignment>
    <assignment>
        <targetRef oid="4e73ed62-aef9-11e9-a7a8-57334ef1f991" type="RoleType"/>
    </assignment>
    <emailAddress>alice.anderson@example.com</emailAddress>
    <fullName>Alice Anderson, PhD.</fullName>
    <givenName>Alice</givenName>
    <familyName>Anderson</familyName>
    <honorificSuffix>PhD.</honorificSuffix>
    <title>Business Analyst</title>
    <employeeNumber>001</employeeNumber>
    <organizationalUnit>10010</organizationalUnit>
</user>

```

Operational, Experimental and Deprecated Items

Most of the items in midPoint schema are quite ordinary and they behave as expected. Such as the `fullName` property. The property can be set and changed by using midPoint user interface. But then there are some extraordinary items. Those are automatically determined and controlled by midPoint core. Those items are essential for correct operation of midPoint. Therefore they are called *operational* items. Operational items are usually not directly displayed in the user interface. They are either completely hidden, displayed indirectly or displayed only when user chooses to displays them.

MidPoint schema has grown and evolved over time. And it is still evolving. Therefore, it is quite expected that the schema will slightly change over time. However, we do not affect midPoint deployments by incompatible schema changes. Therefore items are usually not removed from midPoint schema without a warning. An item that we do not want is marked as *deprecated* first. At that point such item is still working as before. However, it is not displayed in the user interface to discourage use of that item. Deprecated items are removed in one of the subsequent midPoint releases. This gives enough time for midPoint users to adapt to schema changes.

There is also another kind of schema evolution. Development of most midPoint features is quick and straightforward. But then there are features that are quite complex or features that involve some degree of exploration do implement. Those features cannot be implemented in a single midPoint release. There are also features that are provided to the midPoint community as a "preview" to gather feedback for further development. All such features are marked as *experimental*. Those features are not officially supported, but you are free to use them at your own risk. Most new features require extensions of midPoint schema. This is also true for those experimental features. But when going experimental, there is a fair chance that something will change in the future.

Therefore, we are explicitly marking parts of the schema as *experimental*. This is a warning that those parts are likely to change. We are not promising any kind of compatibility for experimental parts of midPoint schema. They may change any time, they may even completely disappear. And there will be no deprecation or any other warning. Simply speaking: if you are dealing with experimental features, you are completely on your own. Do not come crying when those things stop working. You have been warned.

Activation

Time is cruel and everything that we do is in some way temporary. Except perhaps for stupidity, which seems to be utterly endless. But all other things have a beginning and an end. Employees have hiring date, contracts have end dates, users can be disabled, roles may get replaced and so on. We use the term activation to encompass all those things that deal with the questions of digital life and death of the objects.

The activation in itself is multi-dimensional and quite complex. It is composed from several properties that may change in somehow independent and somehow inter-dependent way. Following list provides a quick summary of activation properties:

- **Activation status** defines administrative state of the object, often manually set by system administrator.
- **Validity** properties specify when the object should be active. There is activation date and deactivation date.
- **Effective status** is a computed operational property that show the current effective status of the user. It is computed from other activation properties.
- **Lockout status** is used for temporary inactivation of user, e.g. in case of numerous failed authentication attempts.
- **Additional operational properties** provide (meta) data about the past changes of administrative status.

The best way to explain how activation work is to describe the meaning and behavior of individual properties.

Administrative status defines the "administrative state" of the object (user). I.e. the explicit decision of the administrator. If administrative status is set, this property overrides any other constraints in the activation type. E.g. if this is set to **enabled** and the user is not yet valid (according to *validity* below), the user should be considered active. If set to **disabled** the user is considered inactive regardless of other settings. Therefore this property does **not** necessarily define an actual state of the object. It is a kind of "manual override". In fact, the most common setting for this property is to leave it unset and let other properties determine the state. If this property is not present then the other constraints in the activation type should be considered (namely validity properties, see below).

Administrative Status Value	Description
<i>no value</i>	No explicit override. Other activation properties determine the resulting status.

Administrative Status Value	Description
<code>enabled</code>	The entity is active. It is enabled and fully operational.
<code>disabled</code>	<p>The entity is inactive. It has been disabled by an administrative action.</p> <p>This indicates temporary inactivation and there is an intent to enable the entity later. It is usually used for an employee on parental leave, sabbatical, temporarily disable account for security reasons, etc.</p>
<code>archived</code>	<p>The entity is inactive. It has been disabled by an administrative action.</p> <p>This indicates permanent inactivation and there is no intent to enable the entity later.</p> <p>This state is used to keep the user record or account around for archival purposes. E.g. some systems require that the account exists to maintain referential consistency of historical data, audit records, etc. It may also be used to "blocks" the user or account identifier to avoid their reuse. It is usually used for retired employees and similar cases.</p>

If the administrative status is not present and there are no other constraints in the activation type or if there is no activation type at all, then the object is assumed to be "enabled", i.e. that the user is active.

Validity refers to state when the object is considered legal or otherwise usable. In midPoint the validity is currently defined by two dates: the date from which the object is valid (`validFrom`) and the date to which an object is valid (`validTo`). When talking about users these dates usually represent the date when the contract with the user started (hiring date) and the date when the contract ends. The user is considered *valid* (active) between these two dates. The user is considered inactive before the `validFrom` date or after the `validTo` date.

It is perfectly OK to set just one of the dates or no date at all. If any date is not set then it is assumed to extend to infinity. E.g. if `validFrom` date is not set the user is considered active from the beginning of the universe to the moment specified by the `validTo` date.

The validity is overridden by the administrative status. Therefore, if administrative status is set to any non-empty value then the validity dates are not considered at all.

Activation is also influenced by *object lifecycle*. Object lifecycle specifies phases of object's life, such as `draft`, `proposed`, `active` and `deprecated`. There are some lifecycle states in which the object is considered to be active. And there are other states when the object is considered to be inactive. The

later states are important, because object lifecycle can completely override activation. This makes perfect sense. E.g. when an object is in **draft** state, it is just being prepared for use. Such object may have validity dates or administrative status that would normally activate it. But we do not want draft objects to be active yet. Such object may need a review and approval to transition to **active** lifecycle state. Only then it will really become active. This is just a rough overview of the lifecycle functionality that only scratches the surface. We will deal with object lifecycle details later in this book.

Activation is quite a complex matter that is spread out in several dimensions. Therefore it may not be entirely obvious which objects are active and which are not. For that reason midPoint provides an operational property **effectiveStatus** which shows the computed "effective state" of the object. Simply speaking it is a read-only property that tells whether the user should be considered active or inactive. The effective status is the result of combining several activation settings (administrative status, validity dates, etc.).

The effective status holds the result of a computation, therefore it is an *operational* property that is recomputed every time the status changes. The effective status should not be set directly. The effective status can be changed only indirectly by changing other activation properties.

Effective Status Value	Description
no value	Not yet computed. This should not happen under normal circumstances.
enabled	The entity is active.
disabled	The entity is inactive (temporary inactivation).
archived	The entity is inactive (permanent inactivation).

The effective status is the property that is used by majority of midPoint code when determining whether a particular object is active or inactive. This property should always have a value in a normal case. If this property is not present then the computation haven't taken place yet.

Similarly to effective status, there is yet another operational property **validityStatus**. This property reflects the state of validity constraints with respect to current time. The values are **before**, **in** and **after**, meaning the states before the validity intervals started, inside the validity interval and after the validity interval ended respectively.

Lockout status defines the state of user or account lock-out. Lock-out means that the account was temporarily disabled due to failed login attempts or a similar abuse attempt. This mechanism is usually used to avoid brute-force or dictionary password attacks and the lock will usually expire by itself in a matter of minutes.

This value is usually set by the resource or by midpoint internal authentication code. This value is mostly used to read the lockout status of a user or an account. This value is semi-writable. If the object is locked then it can be used to set it to the unlocked state. But not the other way around. It cannot be used to lock the account. Locking is always done by the authentication code.

Lockout Status Value	Description
<i>no value</i>	No information (generally means unlocked user or account)
normal	Unlocked and operational user or account.
locked	The user or account has been locked. Log-in to the account is temporarily disabled.

Please note that even if user or account are in the **normal** (unlocked) state they still be disabled by administrative status or validity which will make them efficiently inactive.

There is also an informational property `lockoutExpirationTimestamp` that provides information about the expiration of the lock. However, not all resources may be able to provide such information.

There are several *operational properties* in the activation data structure that provide operational data about user activation:

Name	Type	Description
<code>disableReason</code>	URI	URL that identifies a reason for disable. This may be indication that that identity was disabled explicitly, that the disable status was computed or other source of the disabled event.
<code>disableTimestamp</code>	dateTime	Timestamp of last modification of the activation status to the disabled state.
<code>enableTimestamp</code>	dateTime	Timestamp of last modification of the activation status to the enabled state.
<code>archiveTimestamp</code>	dateTime	Timestamp of last modification of the activation status to the archived state.
<code>validityChangeTimestamp</code>	dateTime	Timestamp of last modification of the effective validity state, i.e. last time the validity state was recomputed with result that was different than the previous recomputation. It is used to avoid repeated validity change deltas.

Those properties are *operational*, therefore from the user point of view they are read-only. The values are automatically computed by midPoint and stored in the database.

Let's see how that works on some examples. The simplest example is perhaps not even worth mentioning. A user without any activation data structure is considered to be active (enabled). When such user is stored in midPoint repository, midPoint will automatically compute effective status:

```
<user>
  <name>alice</name>
  ...
  <activation>
    <effectiveStatus>enabled</effectiveStatus>
  </activation>
</user>
```

Administrator can disable such user by using administrative status property:

```
<user>
  <name>alice</name>
  ...
  <activation>
    <administrativeStatus>disabled</administrativeStatus>
  </activation>
</user>
```

Once again, when such user object is stored after the modification, midPoint computes the value of effective status:

```
<user>
  <name>alice</name>
  ...
  <activation>
    <administrativeStatus>disabled</administrativeStatus>
    <effectiveStatus>disabled</effectiveStatus>
  </activation>
</user>
```

The use of administrative status is usually quite harsh. MidPoint deployments are often using validity constraints instead. For example, an employee that has employment contract for a year would look like this:

```

<user>
  <name>bob</name>
  ...
  <activation>
    <validFrom>2019-01-01T00:00:00Z</validFrom>
    <validTo>2019-12-31T23:59:59Z</validTo>
    <validityStatus>in</validityStatus>
    <effectiveStatus>enabled</effectiveStatus>
  </activation>
</user>

```

Given that this chapter was written in 2019, such user will be active. It will automatically switch to inactive state after the last day of 2019. However, if there is ever a need to explicitly disable the user, administrative status can still be used:

```

<user>
  <name>bob</name>
  ...
  <activation>
    <administrativeStatus>disabled</administrativeStatus>
    <validFrom>2019-01-01T00:00:00Z</validFrom>
    <validTo>2019-12-31T23:59:59Z</validTo>
    <validityStatus>in</validityStatus>
    <effectiveStatus>disabled</effectiveStatus>
  </activation>
</user>

```

In this case the user is still in its validity interval. Hence the `in` value of `validityStatus`. But the administrative status is explicitly set to `disabled`. Therefore the resulting effective status is also `disabled`.

The concept of activation is not limited to users. Many midPoint objects have activation. Roles can expire, organizational units can be disabled and so on. Activation is a concept that has a very broad application in midPoint. Even assignments have activation. Which is a crucial element in some configuration (e.g. multi-affiliation). Assignments are often used to model employment contracts, student affiliations, service contracts and similar concepts that have time boundaries. This is usually achieved by a clever use of assignment activation.

Schema Definition

So far we have talked mostly about the use schema (`UserType` data type). However, the entire midPoint schema is quite complex. There are many types of objects and there are thousands of data types overall. It would be almost impossible to manage such a big schema directly in midPoint code. Therefore the schema is defined in special definition files that are used by midPoint in several ways. It is used by the user interface to automatically render form fields. It is used by midPoint expression engine to automatically convert data types. It is even used by midPoint build process (compilation) to make sure that the schema is properly used in midPoint code. MidPoint is

completely schema-aware system from top to bottom.

Schema obviously plays a crucial role in everything that midPoint does. Therefore it may be interesting to have a look at schema definition. This can be particularly useful for engineers that are deploying midPoint professionally and that often needs to extend and customize the schema.

MidPoint schema is specified in XML Schema Definition (XSD) format. MidPoint schema is defined in several parts, but the most important is the "core" schema definition. The schema files reside in midPoint source code in schema component in the `infra` subsystem. Therefore schema files can be found in the `resources` part under the `infra/schema` subdirectory of midPoint source code. Schema files are also included in midPoint distribution package for convenience.



Why XSD? Why did we chose to use the XML Schema Definition format for midPoint schema? There are historic reasons and there are pragmatic reasons. Back in early 2010s when midPoint was born XML was perhaps the only sensible choice to build a complex system. Alternatives such as JSON were young and their schema languages ranged from "very limited" through "useless" to "non-existent". Therefore XML and XSD were a natural choice. However, quite early in midPoint development we have discovered limitations of XSD and especially limitations of Java libraries that work with XML and XSD. We had to extend XSD with custom features. But fortunately, XSD allowed that. We also had to rewrite parts of the XML/XSD-processing code. We have also invented a way how to use XSD to describe generic data structures (a.k.a. "Prism objects") that can be represented in XML, JSON and YAML. Therefore XSD does not really holds us back that much. And despite all the limitations, XSD worked for us quite well during all those years. We expect that we will replace XSD with something better in the future. The problem is that right now there is nothing that is significantly better. And it does not makes any sense for us to switch one buzzword for another only to remain cool. Therefore we will stick to XSD as long as it works for us.

Every deployment engineer that takes midPoint deployments seriously should be aware of the schema. Hardcore engineer will surely open the XSD files in their favorite text editor in the terminal and analyze the definitions line-by-line. Developers could open the XSD files in their IDEs and have a nice organized look at the schema. But even an ordinary engineer could benefit from learning the basics of XSD and having a look at a few important data types in midPoint schema.

Schema definition is not just about the properties, containers and data types. Crucial part of the schema definition is in-line documentation. Most of the data types and items are documented by using XSD in-line documentation mechanism. Therefore a huge amount of details about midPoint can be learned by exploring the schema. We have tried to make that process easier by developing `schemadoc` mechanism. Schemadoc is a process that takes raw midPoint schema and generates HTML documentation out of that. This task is part of midPoint build process and generated documentation is a result of midPoint build. Schemadoc is also available online. Just search for "schemadoc" in midPoint wiki.

Schema is not just a description how midPoint works. MidPoint schema is part of midPoint itself. It is used when midPoint is compiled. It is parsed when midPoint starts. It is used by midPoint core and user interface. MidPoint is complex and even the experts can be sometimes wrong. MidPoint documentation is quite extensive, therefore it may be misleading or out of date at places. But not

the schema. Schema is always right. Otherwise midPoint won't work. Schema is the law.

Schema Extensibility

MidPoint schema is quite rich. Many of the properties that are frequently used in IDM deployments are already part of midPoint schema. But reality has always a way to bring unexpected things. Therefore, midPoint deployments won't get far if midPoint schema cannot be extended.

Vast majority of midPoint schema is available at *compile-time*. This means that such schema is used during compilation (build) of midPoint. That "static" part of schema is somehow hardcoded into midPoint itself and it would very difficult to change. Therefore we have developed a mechanism to extend the schema at *deployment-time*. Small parts of the XSD definition can be provided when midPoint is deployed. MidPoint will read those definitions when it starts up. The static part of the schema is extended with those definitions. From that point on the extensions are part of midPoint schema. The extensions will be used by midPoint user interface, expression-processing code and all other parts of midPoint.

Our ExAmPLE company was quite happy with the progress of IDM deployment so far. Mappings were used to synchronize values of user names and all other common attributes. There is plenty of suitable properties for that in midPoint schema such as `givenName` and `fullName`. Even `employeeNumber` came very handy. But now they need to customize midPoint schema to better suit their very specific needs. The company management decided that the people look really cool in fancy hats. Therefore they will provide a hat for every employee. Which means that the IDM system needs to track hat size for all users. Hat size is not used in the IDM deployments very often, therefore it is not a part of standard midPoint schema. But fortunately, it is easy to extend the schema.

First step to extend midPoint schema is to prepare a small XSD file:

`example.xsd`

```
<xsd:schema targetNamespace="http://example.com/xml/ns/midpoint/schema">
    ...
    <xsd:complexType name="UserTypeExtensionType">
        <xsd:annotation>
            <xsd:appinfo>
                <a:extension ref="c:UserType"/>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="hatSize" type="xsd:string"
                         minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

This file defines a new data structure `UserTypeExtensionType`. The name of this data structure does not really matter. What matters is that it is bound to an extension of `UserType` in the annotation part of the type definition. When midPoint reads this file, it will extend the definition of `UserType` with

this data type.

The extension data type specifies just a single property: `hatSize`. This is an optional single-valued string property. Every user in midPoint will have this property. User interface will automatically display text input field for this property.

MidPoint administrator puts this XSD content into `example.xsd` file. Name of the file can be chosen arbitrarily as long as it has `.xsd` file extension. Administrator copies that file to `schema` subdirectory of midPoint home directory and restarts midPoint. From that point on the schema extension is active.

The users can now be extended with custom property:

```
<user xmlns:exmpl="http://example.com/xml/ns/midpoint/schema">
    <name>alice</name>
    <extension>
        <exmpl:hatSize>M</exmpl:hatSize>
    </extension>
    ...
</user>
```

There is a couple of important remarks to be made here. Firstly, all the extension properties are always placed in a special `extension` container in the objects. Even though the properties are placed inside a container, the user interface will present them in the same way as the static (native) midPoint properties.

Secondly, a clever reader surely noticed that we have used XML namespace here. We have omitted XML namespaces from majority of other examples as they are not that important when working with midPoint objects. But schema is different. Namespaces are handled quite a strict way when working with the schema. Namespaces must be declared and namespace prefixes must be properly used in all XSD definitions. The most important namespace in this case is the *target namespace* of the extended schema. The URI for this namespace should be chosen in such a way that it is globally unique. The use of your DNS domain is the recommended technique.

Namespaces also *should* be used when working with `extension` container in users and other midPoint objects. This requirement is not that strict as midPoint can usually figure out the namespace. However, this may be a problem in case that several schema extensions are combined. Such combinations are possible in midPoint. MidPoint will simply parse all the XSD files in the schema directory and apply all of them as extensions. The namespace is used to differentiate between them. Therefore, if there is an expectation that several schema extensions will be used in the same deployment then the use of namespaces in object extension is more than recommended.

Why is there an **extension** container? Why are not the properties simply mixed among other static properties? This is related to the intricacies of XML and XML schema. Theoretically, XML is completely extensible. However, when XML Schema is applied to XML, some extensibility scenarios do not work very well. And that is also the case for mixing of static XML elements and dynamic XML elements. We are hitting what is called “Unique Particle Resolution” limitation of XML schema. This was further amplified by limitations of Java XML libraries. The easiest and perhaps even most correct way to resolve this limitation was to create a dedicated XML element for schema extensions. That is what we have done in early midPoint versions. The schema processing code in midPoint has significantly improved since and now we are almost at the point where we could remove the **extension** element. But we are not yet there. And there is still an aspect of compatibility to consider. Therefore the **extension** element stays for now. But we are trying hard to hide its existence from the end user.

MidPoint schema does not just specify the "core" data model. MidPoint schema goes a bit further and it can also specify the details of data *presentation*. This means that the schema can specify a label that should be used for particular data item, help text and so on. The XML Schema (XSD) cannot do this. But fortunately, XSD schema can be extended by *annotations*. Those annotations can be used to define the presentation properties of the items:

```
...
<xsd:element name="hatSize" type="xsd:string"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
        <xsd:appinfo>
            <a:displayName>Hat size</a:displayName>
            <a:help>
                Your hat size in whatever mysterious units the hatters
                are using for measuring hats.
            </a:help>
        </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
...
```

This works fine if your system works for just a single localization environment. But this is not enough in case that you need more than one language. MidPoint was born in Europe and we know quite well all the pain that comes with multi-language environments. MidPoint is designed to be localizable. Therefore you can simply use localization keys instead of actual text:

```

...
<xsd:element name="hatSize" type="xsd:string"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
        <xsd:appinfo>
            <a:displayName>
UserTypeExtensionType.hatSize.displayName</a:displayName>
            <a:help>UserTypeExtensionType.hatSize.help</a:help>
        </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
...

```

The actual text to be used for label an help text is looked up in the localization catalog. However, using localization catalogs is a matter of its own. It will be covered by later chapters.

PolyString and Protected String

Majority of midPoint schema is pretty standard stuff. When you walk through the jungle of midPoint schema definition you can see all the usual wildlife: strings, integers, booleans, timestamps and binary values. But there are few species that are quite strange. However strange they might look, they are immensely useful. Their names are *PolyString* and *Protected String*.

PolyString is the stranger one of those two. Its name came from *polymorphic string*, which means a string that can take a variety of forms. In its simplest form PolyString is just a simple string that can be *normalized*. Normalization means that we convert the original string into some standard form, e.g. we are removing leading and trailing whitespace (trimming), we are converting all letters to lower case, simplifying national characters and so on.

Many ordinary midPoint properties are PolyStrings. Object `name` and user's `givenName`, `familyName` and `fullName` and all PolyStrings. And yet not even a clever reader haven't noticed anything suspicious about them so far. The reason for this is that normalization is almost transparent in midPoint PolyStrings. But now it is a time to have a peek inside. Let's import a user that looks like this:

```

<user>
    <name>semančík</name>
    ...
    <fullName>Radovan Semančík, PhD. </fullName>
    ...
</user>

```

What is really stored in midPoint repository is this:

```

<user>
  <name>
    <orig>semančík</orig>
    <norm>semancik</norm>
  </name>
  ...
  <fullName>
    <orig>Radovan Semančík, PhD.</orig>
    <norm>radovan semancik phd</norm>
  </fullName>
  ...
</user>

```

This all happens in a transparent way. PolyStrings are displayed as strings in the user interface. They are handled (almost completely) as strings in the mappings. Ordinary midPoint user has no idea that the normalization happens at all. But why we bother to normalize strings at all? PolyString normalization has many practical uses. However, two of them are embedded quite deep in the way how midPoint works.

Firstly, normalization is used to provide reliable uniqueness mechanism. Usually we do not want a user with username `semancik` and another user with username `Semancik` or even `Semančík`. This may lead to confusion. As midPoint has uniqueness constraints on both the `orig` and `norm` parts of the name then such situation is completely avoided. All those usernames have the same normalized form, therefore the uniqueness constraint on `norm` part of the name will prohibit the use of all those forms at the same time.

Secondly, normalization is simple and elegant way how to conveniently search for objects. When PolyStrings are searched, the value from the query is normalized. Then the norm part of the PolyString is searched. Therefore, whether the query contains `semancik`, `Semancik` or `semančík`, it will always find the user entry above.

Default normalization algorithm in midPoint should be a good fit for most environments. But there are always deployments that are different. For example, characters such as hyphens (-) are usually not considered to be significant. But some deployments will consider `aliceanderson` and `alice-anderson` to be two different usernames. The default midPoint normalization mechanism will remove hyphens, therefore attempt to have two such users will end up with an error. But fortunately the normalization algorithm is customizable. There are several algorithms to choose from and they can even be parameterized. In the extreme case there is a way to develop a completely custom algorithm. Therefore the PolyString normalization should fit pretty much every deployment scenario.

But PolyString still has more tricks to do. The normalization is not much of a polymorphism yet. PolyString becomes a real shape-shifter in fully localized environments. PolyString is designed to store values that can have individualized representations in national environments. E.g. in multi-national deployments we may want to provide localized role names. Like this:

```

<role>
  <name>
    <orig>System administrator</orig>
    <lang>
      <en>System administrator</en>
      <sk>Systémový správca</sk>
      <cz>Správce systémů</cz>
    </lang>
  </name>
  ...
</role>

```

This is a mechanism to display midPoint to end users in their own language, complete with localized *content* of midPoint. This functionality is only partially implemented in midPoint 4.0 and it is considered to be experimental (i.e. unsupported). But this is a glimpse of how the future of midPoint schema may look like.

The other strange animal in the midPoint jungle is *protected string* ([ProtectedStringType](#)). Identity management systems often work with sensitive data such as user passwords. All the identity-related data usually need protection, but those sensitive data items need even better safeguards. This usually means that some kind of cryptographic technique needs to be employed. E.g. we do not want to store passwords in the cleartext form. Want them to be either hashed or encrypted. And that is what protected string is for. Protected string is basically just a simple string, but it has extra cryptographic protection.

If you have ever had something to deal with cryptography you will probably know that cryptography is not simple. Even such a seemingly simple thing as password hash is quite complex when it comes to all the details. E.g. we do not want simple hash as that would not provide sufficient protection. We want *salted* hash. Which means that the salt value needs to be stored together with the string. Many algorithms are parametric and the parameters used during the hashing also need to be stored. And most importantly, we do not want to hard-wire midPoint to any specific algorithm. Cryptographic algorithms often do not age well and they need to be replaced. Therefore we also need to store algorithm identifiers with the value. If the value is encrypted, we also need to store key identifier, as several keys may be active at the same time. And so on. The cryptographic devil is in the tiny and often counter-intuitive details.

Protected string is a data structure that is designed to handle all those pesky cryptographic details and still pretend that the content of the data structure is just a string. Similarly to PolyString, the basic usage is quite simple. Data can be imported into midPoint by using [clearValue](#) element:

```

<user>
  <name>alice</name>
  ...
  <credentials>
    <password>
      <value>
        <clearValue>sup3rSECRET</clearValue>
      </value>
    </password>
  </credentials>
</user>

```

The data are automatically protected when the object is imported into midPoint:

```

<user>
  <name>alice</name>
  ...
  <credentials>
    <password>
      <value>
        <t:encryptedData>
          <t:encryptionMethod>
            <t:algorithm>http://www.w3.org/2001/04/xmlenc#aes128-
cbc</t:algorithm>
          </t:encryptionMethod>
          <t:keyInfo>
            <t:keyName>1z0N17tv6hNQh5CAJ+jWHWDXeBM=</t:keyName>
          </t:keyInfo>
          <t:cipherData>
            <t:cipherValue>
              g6Neg3ZEXY/ga00SpEa9w5M1J9/IR+M1vEjdcei6bM=</t:cipherValue>
            </t:cipherData>
          </t:encryptedData>
        </value>
      </password>
    </credentials>
</user>

```

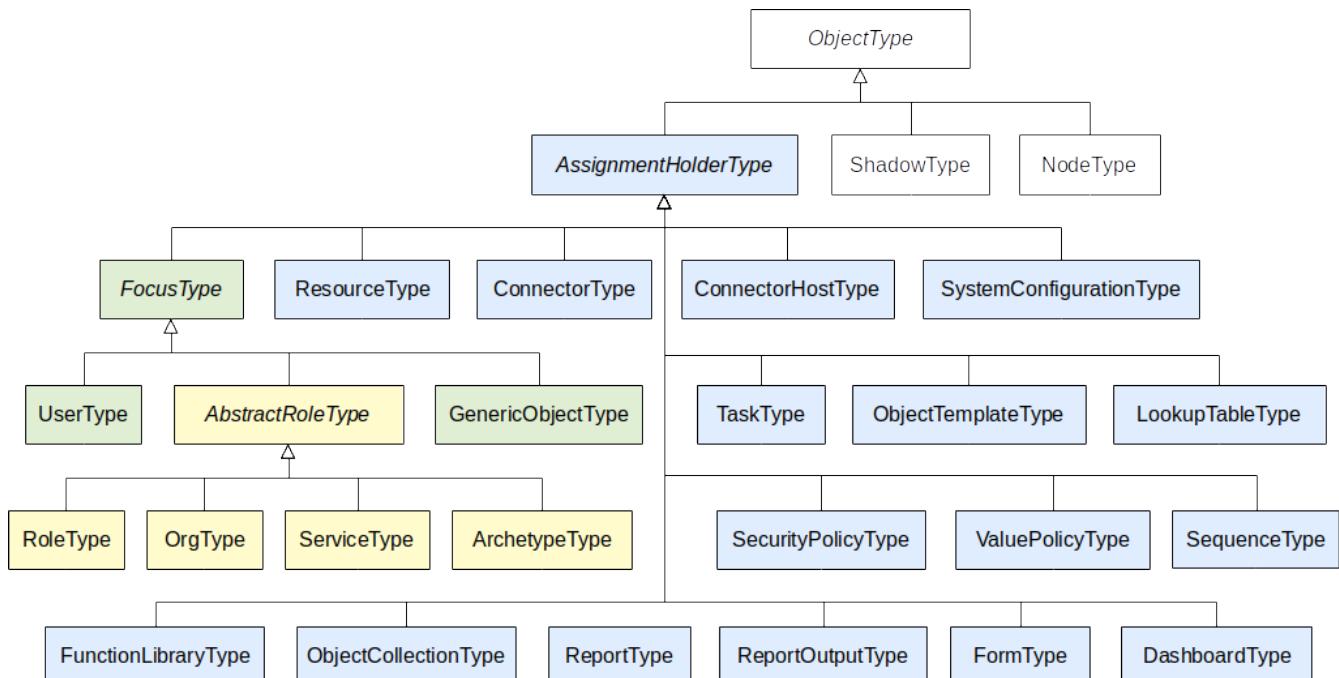
Protected string data type supports cleartext representation, encryption using a symmetric algorithm and hashing. However, the data type itself is just a mechanism for storing the data. Whether specific protected string in the schema gets encrypted or hashed and at which point that happens is not controlled by the protected string itself. It is controlled by midPoint configuration and policies. For example, whether user password is encrypted or hashed is determined by midPoint security policy.

Advanced Schema Concepts

This section describes schema concepts that goes deeper into midPoint mechanisms and implementation. Awareness of those concepts will provide insight into how midPoint works. However, we have already talked about the schema quite a lot. And this chapter was quite low on practical examples. Feel free to skip the rest of this chapter if you want to get your hands dirty as soon as possible. But please make sure to come back later. You will have to learn those schema concepts eventually to get the best of midPoint functionality.

Type Hierarchy

So far we have presented midPoint schema as a simple set of data types. There is [UserType](#) for users, [RoleType](#) for roles and so on. However, all the midPoint objects have something in common. For example, all of them have object identifier (OID), name, description and so on. We could simply copy definitions of those properties to all the data types. But that is not the best way how to do data modeling. The proper way is to create a type hierarchy. Therefore, there is an [ObjectType](#) data type that specifies all the items that all the object types share. However, midPoint schema is substantial and one common ancestor won't be enough. MidPoint type hierarchy was evolving during midPoint development and now it forms quite a rich structure.



Following table is summarizing midPoint data types and their purpose.

Data type	Description
ObjectType	Common (abstract) data type for all midPoint objects. Specifies basic items that all midPoint objects have: name, description, metadata and so on.
AssignmentHolderType	Abstract supertype for all object types that can have assignments.

Data type	Description
FocusType	Abstract supertype for all object types that can be focus of full midPoint computation. This basically means objects that have projections. But focal objects also have activation, they may have personas, etc.
UserType	User object represents a physical user of the system. Properties of User object typically describe the user as a physical person. Therefore, the user object defines handful of properties that are commonly used to describe users in the IDM solutions (employees, customers, partners, etc.)
AbstractRoleType	Abstract data type that contains the "essence" of a role. Roles and other objects that behave like roles are derived from this data type. All abstract roles may "grant" accounts on resources, attributes and entitlements for such accounts. The role can also imply (induce) organizational units, other roles or various IDM objects that can be assigned directly to user.
RoleType	A role in the Role-Based Access Control (RBAC) sense. The roles specify privileges that the user (or other object) should have. Roles are intended to give privileges to users and other objects.
OrgType	Organizational unit, division, section, object group, team, project or any other form of organizing things and/or people. The OrgType objects are designed to form a hierarchical organizational structure (or rather several parallel organizational structures). Orgs are intended to group objects. But as orgs are abstract roles, they can also behave as roles.
ServiceType	This object type represents any kind of abstract or concrete services or devices such as servers, virtual machines, printers, mobile devices, network nodes, application servers, applications or anything similar. The "service" is a very abstract concept.

Data type	Description
ArchetypeType	Archetype definition. Archetype defines custom object (sub)type. I.e. it defines specific behavior, look and feel of objects of a particular type, such as "employee", "project", "application", "business role" and so on.
ResourceType	Resource represents a system or component external to midPoint system which is managed by midPoint. It is sometimes called IT resource, target system, source system, provisioning target or by variety of their names. MidPoint connects to the resource to create accounts, assign accounts to groups, etc. But it also may be an authoritative source of data, database that contains organizational structure and so on.
ConnectorType	Description of a generic connector. Connector in midPoint is any method of connection to the resource. This usually describes a ConnId connector.
ConnectorHostType	Host definition for remote connector, remote connector framework or a remote "gateway". This usually specifies the detail of a ConnId remote connector server.
SystemConfigurationType	System configuration object. Holds global system configuration setting. There is just one object of this type in the system. It has a fixed OID.
TaskType	Object that contains information about a task. This can represent active running task. It may be a scheduled task waiting for execution. Or the object may contain results of a finished task.
ObjectTemplateType	An object that contains mappings and other configuration intended to apply to other object types. E.g. it may be used as "user template" to set up basic properties of new user objects.
LookupTableType	An object that represents lookup table. The lookup table can be used for two purposes: value enumerations (e.g. for GUI or validation) and value mapping (translation). Simply speaking it is a set of key-value pairs that can be efficiently stored and used in midPoint user interface, mappings and so on.

Data type	Description
<code>SecurityPolicyType</code>	System that contains definitions of overall security policy. It contains configuration of authentication mechanisms, credentials management (such as password resets) and so on.
<code>ValuePolicyType</code>	Policy for values of properties. This is almost always used to store password policies.
<code>FunctionLibraryType</code>	Object that contains a set of reusable functions. Those functions can be used in mappings and expressions in all parts of midPoint.
<code>ObjectCollectionType</code>	Object that specifies a collection of other objects. It is mostly just a named search filter that can be reused in other parts of midPoint. But there are also some advanced functions that can be used in dashboards, for compliance purposes and so on.
<code>ReportType</code> <code>ReportOutputType</code>	<p>Specification of midPoint report. This specification defines what the report should contain, how it should look like, output format and so on.</p> <p>This is a report definition. It is a report “template” that can be executed and it produces data. The output data are referred to by report output objects.</p>
<code>SequenceType</code>	Object that refers to specific output of the report. It also contains metadata, e.g. when the report was created, what definition was used, etc.
<code>FormType</code>	Definition of a sequence object that produces unique values. The sequence state is persistently stored in the repository, therefore it can efficiently produce unique identifiers in a controlled and predictable manner.
<code>DashboardType</code>	Form definition. Forms define how a certain user interface form or dialog is presented in the user interface. It is used for user interface customization.
	Object that specifies a look and a behavior of a dashboard. This is used for user interface customization. But it can also specify some aspects of midPoint reports.

Data type	Description
GenericObjectType	Generic type for any other object type that do not fit into any other category. However, support for this data type is extremely limited. We generally do not recommend to use it at all.
ShadowType	Shadow of a resource object. Local copy of any object on the provisioning resource that is related to provisioning. It may be account, group, role (on the target system), privilege, security label, organizational unit or anything else that is worth managing in identity management.
NodeType	Node describes a single installation of midPoint. MidPoint installations can work in cluster. The Node objects are the way how the nodes in cluster know about each other.

Type hierarchy is a principle that is used in many software systems. This principle will be quite obvious to all software developers, but it may need some time to get used to for other engineers. However, the basic idea is quite simple. E.g. [AbstractRoleType](#) has all the items that are needed for an object to behave like a role. [RoleType](#), [OrgType](#), [ServiceType](#) and [ArchetypeType](#) are subtypes of [AbstractRoleType](#). Therefore [RoleType](#), [OrgType](#), [ServiceType](#) and [ArchetypeType](#) can all behave like a role.

This may sound quite strange, why would we want an organizational unit to behave like a role. But the answer is quite obvious. Membership in an organizational unit may imply some privileges. Other IDM systems need complex rules in a form "if user belongs to organizational unit A then he will also have role X". But that is not needed in midPoint. Organizational unit is a role, therefore it can simply include all the roles that are needed. This means that the Role-Based Access Control (RBAC) principles can be applied to several object types. And this is a very typical trait of a midPoint philosophy: reuse of generic principles. We reuse existing principles instead of complicating the system by inventing a new single-purpose mechanism. As you will see later, this makes midPoint both elegant and powerful.

Item Path

MidPoint configuration often needs to reference a particular item in a particular object. For example, mapping sources and target are references to properties and containers. However, midPoint data structures can be quite complex. For example, password is stored in property value that is located in container password which is in container [credentials](#) defined in [UserType](#) data type. It may be difficult to find a way in this little maze. And there may be even some unambiguous situations. For example, user status is controlled by property [administrativeStatus](#) that is in the [activation](#) container. But assignment also has an [activation](#) container and there is an assignment [administrativeStatus](#). Therefore referencing an item by a simple name would not be enough. We need something more sophisticated here.

MidPoint is using the concept of *item path* to reference items in the schema. In its simplest form, item path is just a sequence of item names concatenated by slash characters. For example the path of user administrative status is

```
activation/administrativeStatus
```

whereas the path of assignment activation status is

```
assignment/activation/administrativeStatus
```

Item path provides an unambiguous reference to a specific item in midPoint schema. The path can be used in all the places where there is a need to reference a particular item. It is often used in mappings to specify sources and target. But it is also used in other places that we will mention in later chapters. The concept of path is deeply embedded in all midPoint operations. For example, modification deltas are using item path to precisely pinpoint the places in the object that are modified.

The path is used to locate a particular item in midPoint schema. But it is also used to reference a specific value in midPoint objects. In that case the path often looks exactly the same. As long as we are dealing only with single-value containers, the path can unambiguously point to a specific item. But we may get into trouble in case that multi-valued containers are used. And those are used in midPoint quite often. Assignment is one of those multi-valued container. User can have many assignments. If we want to disable one particular assignment, how do we do it? If we would use the path above then it is not clear which assignment should be disabled. Therefore, in case of multi-valued containers the path is extended with a container identifier in square brackets:

```
assignment[123]/activation/administrativeStatus
```

This path is unambiguously referencing `administrativeStatus` property in an `activation` container in a very specific assignment - an `assignment` container with identifier `123`. This form of the path is used mostly in the deltas and user should not need to ever enter those paths manually. However, this form is often recorded in midPoint log files and other diagnostic output. Therefore it is very useful to be familiar with it.

You might wonder why there is an identifier for `assignment` but there is no identifier for `activation`. Both are containers, aren't they? However, the clever reader already knows the answer. `assignment` is a multi-valued container. Therefore, identifier is needed to pinpoint a specific value of that container. But `activation` is a single-valued container. There is no danger of ambiguity. Therefore the identifier is not needed in this case.

This form of item path works fine if need to identify an item in a particular object. But sometimes we have a lot of objects and other data structures to choose from. For example a mapping can have several sources. And then there are expression variables. Therefore using simple paths would be ambiguous. In such case the path can start with an optional variable identifier:

```
$focus/activation/administrativeStatus
```

The path above explicitly states that it should be applied to the content of variable **focus**. Therefore there is no danger that this path could be applied to a shadow object which also has the **activation** container. This form of item path is often used in path expression evaluators.

Clever reader is surely wondering about QNames now. The XML schema defines the elements in a form of QNames, which basically means "names in a namespace". Therefore element names and QNames. And path should use QNames as well. But so far all the names in the path looks like simple strings. Yes, they are simple strings. But they point to elements in the schema. While the path is correct and unambiguous, midPoint does not need the namespaces. Simple string (known as *local part* of QNames) are enough to navigate through the schema and automatically determine the namespaces. This is the same principle used for parsing XML, JSON or YAML document without namespace definitions. However, there may be ambiguities in case that several custom schema extensions are used. Those extensions may have elements with conflicting local parts. In that case an alternative form of item path can be used:

```
declare namespace exmpl="http://example.com/xml/ns/midpoint/schema";  
extension/exmpl:foo
```



This alternative form is based on XPath specification, that was used in early midPoint versions and it was an inspiration for the concept of item path.

Clever reader may have also noticed that there are two types of namespaces that are often used in midPoint:

```
http://prism.evolveum.com/xml/ns/public/...
```

```
http://midpoint.evolveum.com/xml/ns/public/...
```

Indeed, the schema is divided into two big parts:

- **Prism schema** is used to express basic concepts that deal with objects, deltas, item paths, queries and similar mechanisms. Those are concepts of our data representation library that we dubbed Prism. Prism concepts are very generic mechanisms that have nothing to do with identity management. While currently Prism is an integral part of midPoint, it is supposed to be a general-purpose data representation library that can be reused to build other applications. The plan is to separate Prism from midPoint at some point in the future.
- **MidPoint schema** is used to express all the objects and data types that midPoint works with. All the concepts specific to identity management are there: user, role, org, assignment and many, many others. This is the data model of identity management as it is implemented in midPoint.

Conclusion

This is all about midPoint schema that you need to know right now. There is still much more to learn, as the entire midPoint schema is big and complex. And understanding of midPoint schema is absolutely crucial, as the schema is a foundation of everything that midPoint does. But the best way to do the learning is to do it on the go. You will learn more about midPoint schema as you will explore midPoint functionality.

Chapter 7. Role-Based Access Control

Simplicity is the most complex of all concepts.

— Mentat conundrum, Dune: House Corrino by Brian Herbert

Basic idea of Role-based access control (RBAC) is very simple: instead of assigning the same privileges to users over and over again, let's group such privileges into roles. Then assign roles to users. This often aligns with organizational roles such as *manager*, *assistant* or *analyst*. Therefore, roles are quite easy to understand even on an intuitive level. And RBAC should make your life easier - at least in theory.

Role-based access control principles are present in almost all identity management systems. Therefore it is no surprise that RBAC is one of the basic midPoint mechanisms to organize privileges in midPoint. MidPoint supports all the usual RBAC features such as role hierarchies, an automatic assignment of roles, entitlement definition etc. But midPoint goes beyond traditional RBAC. MidPoint roles can be smart. There may be dynamic expression inside midPoint roles, such as attribute mappings. The roles may be conditional, so one role is included in another role, but only in case that a specific condition is satisfied. The roles may be parametric, so the role can determine the specific set of entitlements based on the user data or a parameter of a role assignment.

But midPoint role dynamics goes even one step further. The RBAC system can be applied to the roles themselves, thus creating *meta-roles*. It is quite common that the roles are divided into several types: application roles, business roles, technical roles and so on. However, all the business roles have common characteristics such as common approval processes, common life-cycle policies etc. Instead of copying the common parts into each and every business role, the business roles may be assigned a common meta-role. The meta-role defines all the common characteristics of all business roles, therefore the RBAC system is much easier to maintain. And this concept is extended even further with *archetypes*. But more on that later.

Terminology.

The term *RBAC* is many things to many people. We use the term *RBAC* in quite a broad sense. We do not strictly mean NIST RBAC model. What me mean by *RBAC* is a generic mechanism that is based on the concept of roles. Although the basic principles of midPoint RBAC are very similar to NIST RBAC model, we take the liberty to deviate from NIST model when needed. And as you will see later, such deviation is really necessary.



Reality, Policy and Assignments

Previous chapters were focused on account provisioning and synchronization. Which means that the primary focus was an *account* (or a similar resource object). This is what we call *reality* in midPoint way of thinking. Accounts are objects that exist in the databases and files on the resources. In that aspect they are almost tangible things. Existence of an account allows user to access a particular system, to execute operations and so on. Therefore we consider an account to be something *real*.

But how do we know whether an account should exist or it should not exist? The situation would be quite clear if midPoint is the only source of truth. In that case if there is a linked shadow then account should exist. If there is no shadow, then account is illegal. But reality is almost never that simple. In real deployments MidPoint is not the only source of truth. It is usually human resource (HR) system that is the source of the truth – but only for some types of users, usually employees. Then there are external users, temporary workers, special personas for user administrators and so on. Some of them may have their own source systems similar to HR database. But for some users it may still be midPoint which is the ultimate source of truth. And that "truth" may be in fact only partial or compiled from several sources. To keep long story short: reality is messy and complicated. And it is often quite difficult to figure out which accounts particular user should have and which he should not have. Yet, for an IDM system this distinction is absolutely crucial. Various IDM systems came with broad range of mechanisms to handle this problem, and sadly, those mechanisms are often not very good. Fortunately, midPoint was designed from the beginning with a full awareness of this problem. Therefore there is a clean distinction between *reality* and *policy* in midPoint.

Accounts, shadows and links are what we refer to as *reality*. Those describe what *exists*, what *is*. And there is a separate mechanism to describe policy. Policy, in midPoint parlance, means definition of what *should be*. In the ideal world *reality* and *policy* should be in accord. They should describe the same state of things. But we do not live in ideal world. Perfectly good accounts may be deleted by mistake, illegal accounts may be created, entitlements may get mixed up, attribute values destroyed – there are many dangers in the big wild world out there. And then there are scenarios when we actually want reality to be different than policy for some period of time. Those may be migration scenarios when a new system is being connected to midPoint and the data needs to be cleaned up. *Reality* and *policy* do not match exactly in practice. We all know that only too well. Therefore, midPoint is designed in such a way that it can graciously handle the differences between *reality* and *policy*.

When it comes to policy, the most important concept is an *assignment*. Simply speaking, assignment is a data structure which specifies that a particular user should have something. The simplest case is *account assignment*. This type of assignment states that the user should have an account on a particular resource. When such assignment is added to a user, there is suddenly a discrepancy between reality and policy. The assignment states that a user *should have* an account. But there is no such account yet. It is a nature of midPoint to align policy and reality as much as possible (unless it is told otherwise). Therefore midPoint will try to create missing account. Once that account is created *reality* and *policy* are aligned once again.

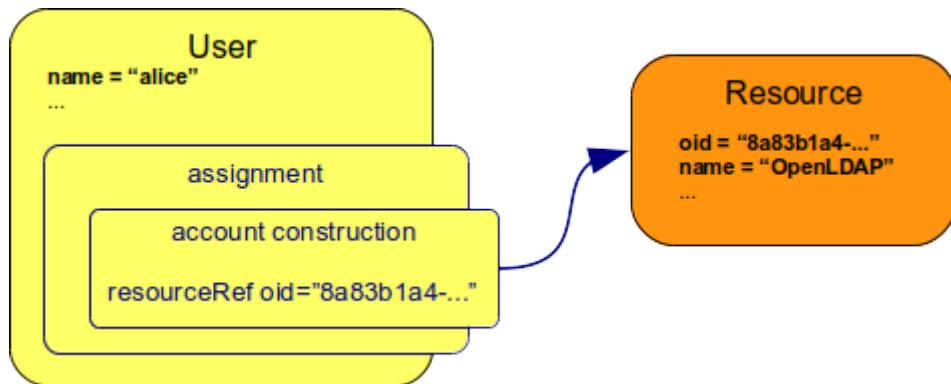
The mechanism that midPoint uses to define that a particular user needs an account, entitlement or other resource object is called *construction*. The simplest case is a *construction* that specifies to create an account (a.k.a *account construction*):

```

<user>
  ...
  <assignment>
    <construction>
      <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
      <kind>account</kind>
    </construction>
  </assignment>
  ...
</user>

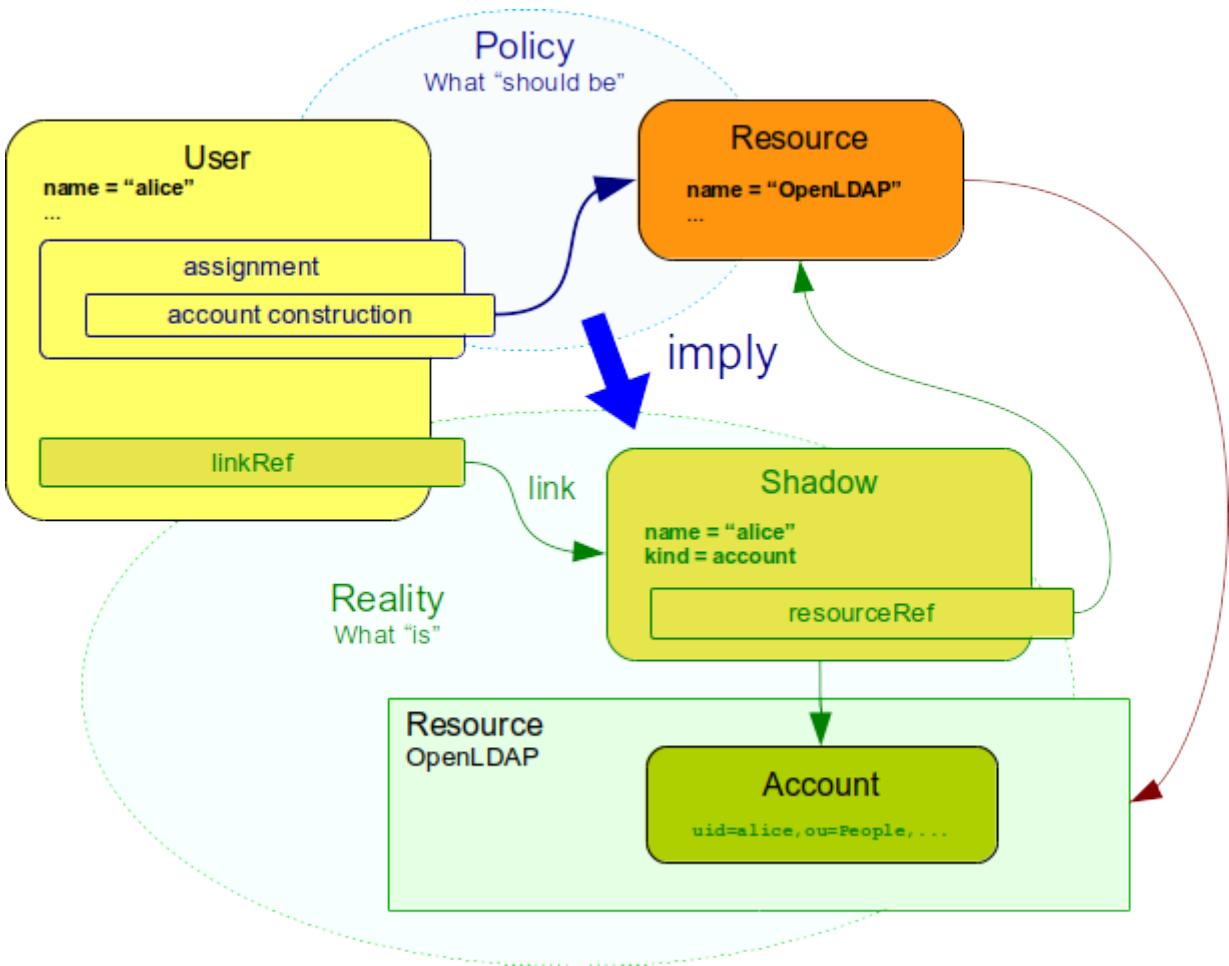
```

The term *construction* means that object on that particular resource should be constructed. In this case the object on OpenLDAP resource should be constructed for this particular user. If no construction parameters are specified then a *default account* will be constructed. Which means that outbound mappings in the OpenLDAP resource definition will be used to set up the account.



Construction can be quite a complex data structure describing object types, object classes, attributes and so on. However, it is unlikely that they will be placed directly in assignment like this. But more on that later. What is important for now is that assignments specify *policy*.

After the assignment is added to a user and all the processing and provisioning takes place the situation looks like this:



Assignment is a definition of policy which states that an OpenLDAP account should exist for Alice. But there is no such account. Therefore MidPoint aligns reality and policy by creating that account. As for any other account there is a shadow and a link to track account ownership.

This may look like a very complicated method to do something simple. But this kind of thinking is really necessary to handle complex cases. There may be several assignments that mandate the same account. There may be assignments for the same accounts, but each assignment mandates different attributes or values. The account that the assignments mandate may exist already, e.g. it may be linked by previous reconciliation with the resource. There may be several accounts for the same user on the same resource (e.g. “ordinary” account and “testing” account). And so on. We will deal with various cases in this book. But the basic principle is the same: assignments are policy and midPoint is trying to align reality to match the policy.

Roles

There is much more in the concept of an assignment than just the very simple account assignment that was introduced above. Assignment is a generic mechanism that is used in midPoint for wide variety of cases, from simple account provisioning to really complex identity governance policies. But one specific assignment type is particularly interesting with respect to the topic of this chapter: role assignment.

The basic idea of Role-Based Access Control (RBAC) is simple: Instead of assigning account to users directly, let us group all accounts that a particular group of users need into a *role*. Then assign the role to users. Later on you may add new application to your system and you want all the users to

have account there. In that case all that is needed is to add that account to a role and recompute the users. All the users that should have the account will get the account. This principle is reused for many purposes in midPoint: accounts, privileges, authorizations, policies ...

Role is a special type of object in midPoint. But as all midPoint objects it has a very familiar structure:

```
<role oid="aaa6cde4-0471-11e9-9b50-c743da469067">
    <name>Business Analyst</name>
    ...
</user>
```

Role object has its OID and name. The rest of the role usually specifies the privileges that the role gives to the users. But how do we give this roles to users? That is what role assignment is good for:

```
<user>
    <name>alice</name>
    ...
    <assignment>
        <targetRef oid="aaa6cde4-0471-11e9-9b50-c743da469067" type="RoleType"/>
    </assignment>
</user>
```

User `alice` has role `Business Analyst` assigned. The assignment is using the familiar style of object references in midPoint, referring to the role by its OID. This is very useful, as the assignment stays the same in case that the role or the user are renamed - and both of those events are much more frequent than one would think.

Provisioning Roles

Provisioning is the bread and butter of identity management. Therefore it is quite understandable that the most natural usage of roles in midPoint is to automate provisioning. Provisioning roles are usually combining several *construction* statements. The idea is that a provisioning role should specify all the privileges that users of that role need. Therefore a `Business Analyst` role may look like this:

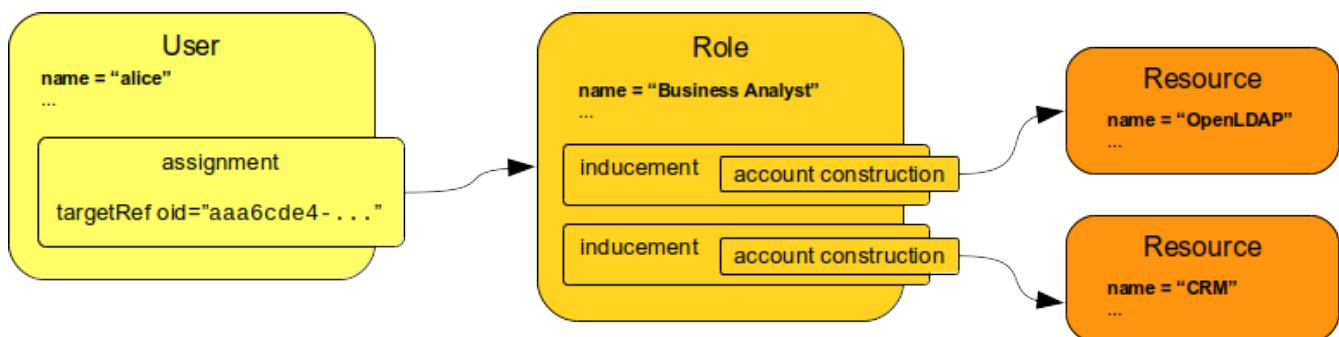
```

<role oid="aaa6cde4-0471-11e9-9b50-c743da469067">
  <name>Business Analyst</name>
  <inducement>
    <construction>
      <!-- OpenLDAP resource -->
      <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
      <kind>account</kind>
    </construction>
  </inducement>
  <inducement>
    <construction>
      <!-- CRM resource -->
      <resourceRef oid="04afeda6-394b-11e6-8cbe-abf7ff430056"/>
      <kind>account</kind>
    </construction>
  </inducement>
</user>

```

The usual case is that every employee need to have basic access to company functionality. In our case that access is granted by an account in central OpenLDAP directory. In addition to the basic LDAP account, business analysts need access to the CRM system. The role combines all the accounts that a business analyst needs. Assign that one role and the user has all that is needed to do the job.

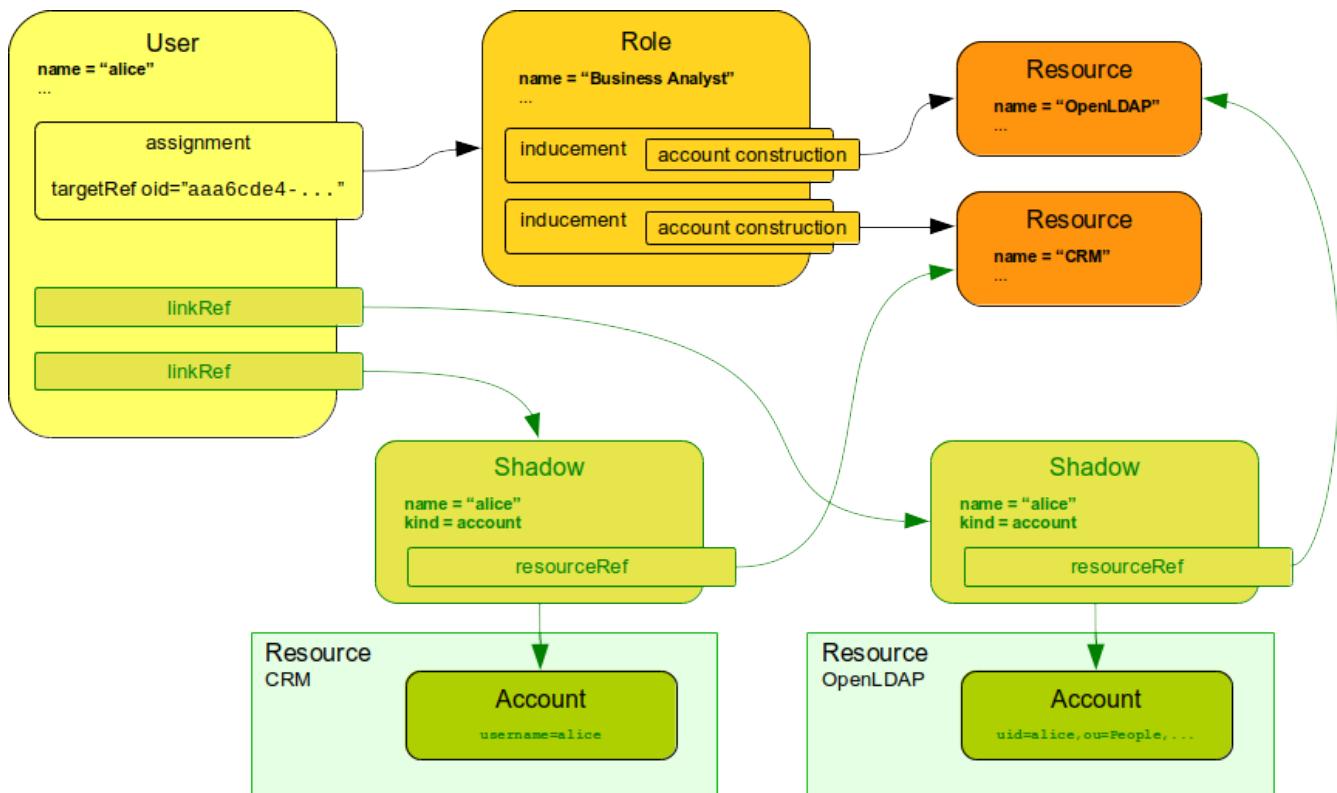
But, what is that mysterious *inducement* thing? Think of inducement as indirect assignment. Assignments give privileges directly to the object in which they are placed. The assignment in the previous section gave account to the user because it was placed in the user object. However, here we do not want the accounts to be created for a role. We want accounts to be created for all the users that have the role. That is one *order of indirection* down the line. Therefore we (usually) do not want to use assignments in roles. We want to use something that reflects this indirect relation. And that is exactly what inducement is. Inducement is very similar to assignment - in fact it has exactly the same structure. But while assignment is direct, inducement is indirect.



MidPoint user interface can show a nice summary of the inducements:

Name	Activation	More data
OpenLDAP	enabled	Kind: account
CRM	enabled	Kind: account

It is perhaps worth explaining what happens if this **Business Analyst** role is assigned to a user. When that role is assigned to a user, midPoint will process all the parts of role definition. MidPoint will take the inducements from the role and apply them to the user. In fact, midPoint will behave in almost the same way as if those construction statements were specified directly in user's assignment. And then we have the familiar principle: policy mandates that two accounts should exist, but in reality there are no such accounts. Therefore midPoint creates the accounts. MidPoint also creates appropriate shadow objects and links them to the user.



Many applications implement at least some aspects of RBAC, as RBAC is a very useful way how to organize the privileges. However, almost all the applications limit the applicability of RBAC to the application itself. I.e. roles in an application can contain only those privileges that apply to that particular application. The roles cannot have privileges from other application. But IDM systems are different. IDM systems such as midPoint are reaching out to many applications (resources). Therefore, a single midPoint role can give access to many applications at once. No other application

can do that.

Roles, Accounts and Attributes

We have already seen how outbound mappings can be used to set up account attributes. Roles can also contain outbound mappings, therefore they can be used for a similar purpose:

```
<role oid="aaa6cde4-0471-11e9-9b50-c743da469067">
    <name>Business Analyst</name>
    <inducement>
        <construction>
            <!-- OpenLDAP resource -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
            <kind>account</kind>
            <attribute>
                <ref>ri:title</ref>
                <outbound>
                    <expression>
                        <value>Business Analyst</value>
                    </expression>
                </outbound>
            </attribute>
        </construction>
    </inducement>
    ...
</role>
```

When the above role is assigned to a user, an account on OpenLDAP server will be created. The account will be provisioned in a usual way. All the outbound mappings from resource definition will be applied to set up the account. But there is one difference. The role specifies one additional outbound mapping for the account. This mapping will be included in the set of usual account mappings when the account will be provisioned. Therefore the account will have attribute **title** set to **Business Analyst**.

This is a very typical way how midPoint deployments are set up:

- Common and usual attribute values are specified by outbound mappings in the resource definition (in **schemaHandling**). Those are usual mappings that take user properties as their source. Many of those mappings do not even modify the value at all (**asIs** mappings).
- Attributes that are specific to roles are defined in the roles themselves. Those mappings often do not have any source at all. They just set a static value (literal **value** mappings).

At the time when midPoint is about to provision an account, all the mappings are merged and processed together. It is quite common than more than one role has a construction for the same account. All those constructions from all such roles are merged together and they are added to the mappings specified in resource definition. All those mappings are used to compute final values of account attributes.

Most account attributes are single-valued. Attempt to set more than one value for such an attribute will end up with an error. Therefore it does not make sense to specify more than one mapping for such an attribute. The mapping can be specified in a resource or in the role, but only one of those should be active at the time. Mapping conditions and strength can be used to selectively deactivate some mappings in more complicated cases. But it still means that only one mapping is active at a time.

However, some attributes are multi-value. In that case midPoint will merge the values from all the mappings. In that case several roles may contribute to the final set of attribute values, as can the mapping in resource definition. This is the usual case of attributes that specify privileges, such as permissions, authorization codes, access control list (ACL) entries and so on.

Merging of multi-value attributes is an easy way how to manage simple privileges in resources. However, midPoint contains a whole sophisticated mechanism for managing *entitlements* such as groups. There is an entire chapter in this book dedicated to entitlement management.

Additive principle.

MidPoint is built on a principle of merging. Assigned roles are merged together, that is merged with outbound mappings, entitlements are merged and so on. MidPoint always adds, it never subtracts. E.g. there is no simple way how one role can "eliminate" a value given by another role. If a role specifies that an account should have value A, that account will have value A. And that's it. It can also have values B and C given by other roles. But A will always be there, no matter what other roles do (unless those roles are involved in some really dark magic). This may seem quite limiting. But it is sufficient for vast majority of cases. It only needs a change in your way of thinking about privileges. Do not think about removing a privilege. Think about *not adding* a privilege. There are many ways how that can be achieved. There is a role hierarchy, mappings can be conditional and whole assignments and inducements can be conditional too. We are trying really hard to avoid concept of "removing" privileges, because that requires ordered processing. E.g. if role X adds something and role Y removes it, the final result depends on the order in which such roles are processed. This creates ambiguities, it limits parallelism and overall it is a huge complication. Therefore we try to avoid it. And so far we have been successful.



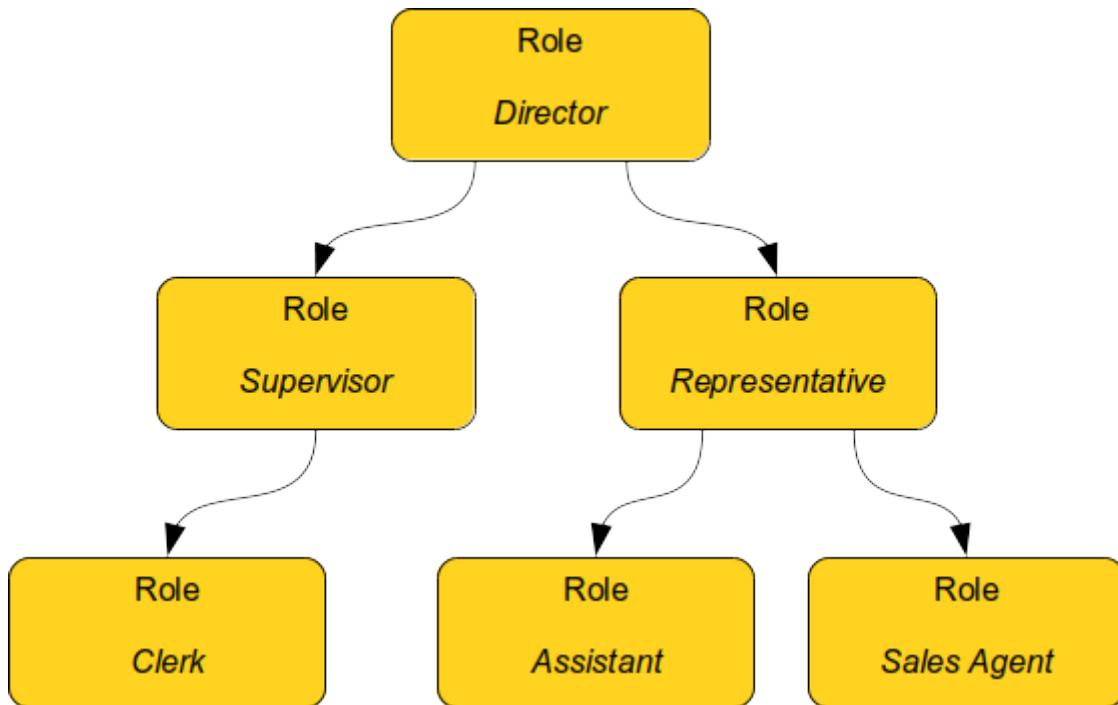
Role Hierarchy

Ability to group privileges into roles is quite useful. But it is still not good enough unless your access control policy is extremely simple. Most practical policies require to place roles into roles, thus creating role hierarchy.

Let's consider two work positions: clerk and supervisor. Clerk has some basic set of privileges. Supervisor can do everything that a clerk can do, but supervisor has some additional privileges. A naive way would be to simply copy all the clerk's privileges in supervisor's role. However, privileges are seldom static. Access control policies tend to change and evolve as much as the environment changes. It is likely that a clerk's privileges will change. In that case we will need to update the supervisor's role as well. This would be a maintenance burden. Now imagine hundreds or thousands of related roles that need constant maintenance. Any person maintaining such a

structure will need superhuman precision and patience to do that.

A more natural idea would be to include clerk's role into a supervisor's role. If clerk's privileges change, then also supervisor's privileges are automatically updated. Maintenance is much easier. And that is the basic idea of role hierarchy. Basic privileges are placed into low-level roles. Low-level roles are combined to create a higher-level roles. Then those roles are combined as well. The process is repeated until there are all the roles that are needed for assignment to users.



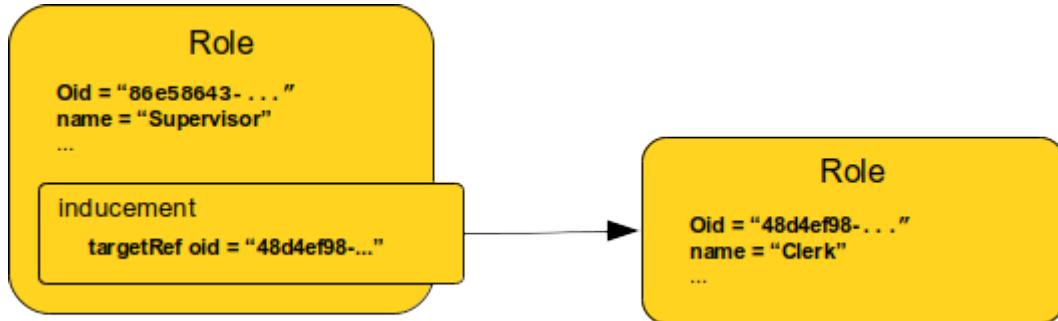
Creating role hierarchies in midPoint is quite easy. A clever reader would already expect that this has something to do with inducements. And clever reader would be absolutely right. Role hierarchy is nothing more than a set of inducements between roles:

```
<role oid="48d4ef98-20e3-46ab-cd78-548d38364a6b">
    <name>Clerk</name>
    <!-- Privileges needed to do clerk's work will be here. -->
</role>
```

```
<role oid="86e58643-d5e7-36a8-04f6-38dc3754f04e">
    <name>Supervisor</name>
    <!-- Privileges that are unique to supervisor's work will be here. -->
    <inducement>
        <!-- This "includes" all the clerk's privileges in this role -->
        <targetRef oid="48d4ef98-20e3-46ab-cd78-548d38364a6b" type="RoleType"/>
    </inducement>
</role>
```

The inducement includes **Clerk** role in **Supervisor** role. When midPoint evaluates the **Supervisor** role, it will get all the inducements from both the **Supervisor** and **Clerk** roles. This process is almost transparent, it works almost as if the clerk's privileges were copied in the supervisor's role. All the

constructions in all the inducements in both roles are processed. Therefore, supervisor will get all the accounts that a clerk would get, plus few extra privileges.



Both **Clerk** and **Supervisor** roles are likely to have construction for the same account. This is quite natural, as both clerk and supervisor would probably work with the same applications. However, their privileges will be different. This is where the merging mechanism becomes very useful. When a supervisor role is processed then privileges of clerk are merged with privileges of supervisor:

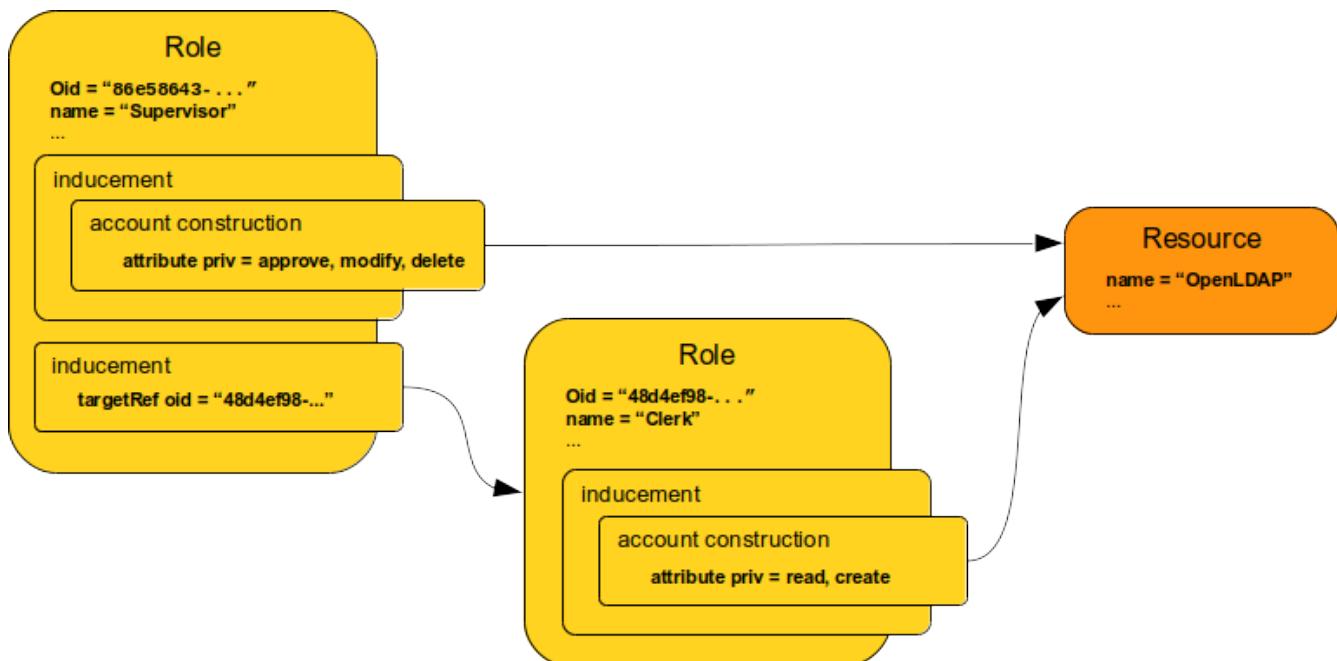
```
<role oid="48d4ef98-20e3-46ab-cd78-548d38364a6b">
    <name>Clerk</name>
    <inducement>
        <construction>
            <!-- Record management system -->
            <resourceRef oid="84de003e-014f-2040-efbc-482e009ed2bcf"/>
            <kind>account</kind>
            <attribute>
                <ref>ri:priv</ref>
                <outbound>
                    <expression>
                        <value>read</value>
                        <value>create</value>
                    </expression>
                </outbound>
            </attribute>
        </construction>
    </inducement>
</role>
```

```

<role oid="86e58643-d5e7-36a8-04f6-38dc3754f04e">
  <name>Supervisor</name>
  <inducement>
    <construction>
      <!-- Record management system -->
      <resourceRef oid="84de003e-014f-2040-efbc-482e009ed2bcf"/>
      <kind>account</kind>
      <attribute>
        <ref>ri:priv</ref>
        <outbound>
          <expression>
            <value>approve</value>
            <value>modify</value>
            <value>delete</value>
          </expression>
        </outbound>
      </attribute>
    </construction>
  </inducement>
  <inducement>
    <targetRef oid="48d4ef98-20e3-46ab-cd78-548d38364a6b" type="RoleType"/>
  </inducement>
</role>

```

When supervisor's role is processed, midPoint figures out that those two `construction` statements are referring to the same account. Therefore they will be merged together. Supervisor will get an account that will have `priv` attribute set to values `read, create, approve, modify` and `delete`.



Assignments in roles.

i

So far we have seen only inducement in the roles. But what about assignment? Assignment is indeed used in the roles, but it has different meaning. Inducement means that role A has to be included in role B. But assignment means that role A has to be applied to role B. In that case role A is in fact a meta-role. But more on that later. For now it is good to remember a rule of the thumb: role hierarchy is always created by inducements.

Role Universality

MidPoint roles are very useful kind of an animal, because they are used for almost everything in midPoint. MidPoint role be used as:

- **Provisioning role:** The role can include constructions that control provisioning and deprovisioning of accounts.
- **Entitlement management:** The constructions can include specification of groups, resource-side roles, privileges and so on.
- **Internal authorization:** The roles give access to data in midPoint itself. E.g. a role can allow reading some user properties. Authorizations in a role can also allow access to particular parts of midPoint user interface, remote network services and so on.
- **Policy specifications:** Roles (and especially meta-roles) are the places where important parts of policy management is specified. Roles include policy rules that can apply segregation of duties (SoD) policies, ownership and approval policies and so on.

All those aspects can be combined together into a single role. Therefore, such role can specify everything that is needed for the role holder to live a complete digital life: access to systems (accounts), entitlements, access to midPoint itself (e.g. for self-service), apply policy constraints and so on. Everything in one place.

Role universality may seem mundane and completely natural, but in fact it is quite unique and incredibly powerful idea. As you will see later, roles can be driven through approval process, lifecycle management can be applied to role, roles can be subject to policies, role compliance can be evaluated and so on. All of this applies to provisioning roles. But the same mechanism can be applied also to roles that govern the administration of midPoint itself. And even to meta-roles that specify high-level policies. Which means that in a strange post-modern way midPoint can be applied to itself. MidPoint can be its own manager.

i

Surprisingly, role universality is quite a unique concept in the IDM field. The common approach of older IDM systems is to separate provisioning roles, authorization roles, governance roles and so on. Each of them was different and it was managed in a different way. It was quite difficult to create a unified and consistent policy. This is one of many aspects where midPoint provides a seemingly simple mechanism, but that mechanisms simplifies a lot of things and provides an elegant solution to a difficult problem.

Role Hierarchy Structure

There are many ways how a role hierarchy can be structured. One way is to create all roles as "end user" roles that are supposed to be directly assigned to user. The clerk-supervisor example above is that case. But there are also other approaches that are often used. For example, the low level roles are often modeled as *application roles*. Those roles deal with access to a single application, but they are not meant to be assigned directly to users. They are supposed to be abstract, to be the base "material" used to create other roles. Those higher-level roles are often called *business roles*, as they reflect the needs of the business, such as specific job or responsibility in a business process. Those roles are assigned to users. However, those are just two of many conventions and recommendations on designing a role structure. We will not dive deep into role modeling topics in this book. There is a plenty of literature on that topic already. We will rather focus on a technical implementation of role hierarchy in midPoint.

However, there are few cases that are frequent pain points in RBAC deployments. The philosophy of midPoint is to make IDM deployment easier, therefore it is quite natural that midPoint tries to address those particular issues.

One of the big troubles are application roles. There is usually a huge number of them and they need to be maintained manually. It is usual practice that there is one application role for every privilege in the target system (resource), for every group for every organizational unit and so on. Application roles duplicate the information that is already present on the resource side. And as application roles are maintained manually, it is almost certain that this information will become inconsistent. This approach is sometime even recommended as a best practice. But the reality it is a maintenance nightmare.

This behavior is motivated by several factors. But perhaps the strongest factor is that it was very difficult to set up the privileges in older (first generation) IDM systems. Setup of an application role often required intimate knowledge of the entire IDM configuration as various tricks were used to implement entitlement management. Heavy connector support and customization were often necessary to provide even the very basic entitlement management. However, midPoint is different. There is a clean concept of *construction*, which is designed in such a way that a system (resource) administrator can understand. E.g. the construction refers to the native (non-mapped) names of resource attributes, it is referring to native object classes that are used on the resource, native identifiers and so on. The construction is built to use the language of the target system (resource), not the language of midPoint. Therefore, there is a good chance that system administrators can set up constructions easily. In addition to that, midPoint has a native support for entitlements such as groups and resource-side roles. Those are designed to be easy to use in constructions. Therefore there is very little need for application roles in midPoint. Higher-level roles can contain the constructions directly. Therefore the need for application roles in midPoint is significantly reduced.

However, application roles may still be needed and in some cases they may even be useful. For example, there may be a common combination of privileges that is always assigned together. In that case it makes a lot of sense to create an application role. There may still be a need for application roles if strong role lifecycle management is required. Or application roles might be a legacy from a previous IDM system. As always, the best recommendation would be to analyze the RBAC policies and to use pragmatic thinking. Be careful about generic RBAC recommendations and be even more careful about recommendations that are meant for older IDM systems. MidPoint is

different. Of course, midPoint can implement all those old fashioned RBAC models. But it will be a pain to maintain. MidPoint can do better. Try to understand how midPoint works with the roles first. And they apply those mechanisms to your RBAC policies. You might be surprised how midPoint can simplify the implementation of the policies.

In case that application roles are still needed, you can consider to automate the management of those roles. MidPoint synchronization mechanism is really powerful. It can synchronize users and accounts, but it is designed to synchronize almost anything with anything. Therefore, it can be used to automatically create application roles from all the LDAP groups. While this approach still have some drawbacks, it automates the most painful parts of application role's maintenance.

There is yet another practice that is quite common, but mostly wrong. Many IDM deployments create "login roles" or "default roles" for each application (resource). Those roles are supposed to define basic properties of the account. And in some cases they are supposed to keep account in existence when such account is unassigned. While this practice is very common, it is complicated, messy and very difficult to maintain. MidPoint was specifically designed in such a way that this practice is not necessary in midPoint deployments. Default account attributes are easy to set up by using resource definition. And even the ability to keep unassigned account is directly supported in midPoint by using existence mapping, which will be described later. Therefore, such "login roles" are not needed at all in midPoint deployments and they are generally considered to be a bad practice.

Assignment Gets Complicated

At the first sight, the concept of assignment may seem quite mundane, maybe even over-complicated. But in fact it is a very powerful concept and it has been a crucial part of midPoint design from the very beginning. Assignment is so much more than just a simple user-role connection:

- Assignments can have **validity period**. This can be used to assign roles for a temporary period of time. It can also be used to assign roles that will be activated in the future.
- Assignment have **administrative status** that can be used to manually disable or enable a particular assignment. This can be used to manage exceptions from the policies or it can be very useful in emergency situations.
- Assignments can contain **parameters** that are used to support parametric roles (see above).
- Assignments are subject to policies, governance and compliance mechanisms. Assignments have their lifecycle, they are subject to re-certification campaigns, there can be policy exception recorded for an assignment and so on. But more on that in later chapters of this book.

For example, assignment validity period can be used to assign a role only for a temporary period:

```

<user>
  <name>bob</name>
  ...
  <assignment>
    <!-- Deputy Cheerleader role -->
    <targetRef oid="0c87d8f8-c9a4-11e9-81b8-e7d43e9f9a2b" type="RoleType"/>
    <activation>
      <validTo>2019-12-31T23:59:59Z</validTo>
    </activation>
  </assignment>
</user>

```

As assignment and inducement are in fact the same data structure, similar approach can be used to disable parts of role hierarchy:

```

<role>
  <name>Marketing Research Undersecretary</name>
  ...
  <inducement>...</inducement>
  <inducement>...</inducement>
  ...
  <inducement>
    <description>
      Employee access to the lab is disabled because the lab burned down
      during an ugly accident. Will be re-enabled when the lab is rebuilt.
    </description>
    <!-- Experimental Research Lab Access role -->
    <targetRef oid="e8ef819c-c9a4-11e9-80a8-1bddb446391e" type="RoleType"/>
    <activation>
      <administrativeStatus>disabled</administrativeStatus>
    </activation>
  </inducement>
</user>

```

Many types and variants of assignments can be combined in a single user. Assignment validity periods may overlap, there may be disabled assignments and enabled assignments for the same role at the same time, there may be several assignments to the same role with various parameters and so on. All reasonable combinations are supported, which allows to model very complicated schemes such as multi-affiliation, multiple employment contracts and so on. Assignment is a crucial data structure and we will be dealing with it in almost every chapter in the book.

Dynamic Roles

RBAC is a nice and elegant way how to create and maintain access control policies. However, there is a serious danger: roles can be quite explosive. The role structure can easily get out of control and the roles may start to multiply. This is known as *role explosion* and it is one of the nastiest drawbacks of access control system based on static roles. It is not uncommon for an organization to

have much more roles than it has users. This creates a recurring maintenance nightmare. Fortunately, midPoint has a very powerful support for dynamic roles that can significantly reduce or even completely eliminate the impact of role explosion.

To understand dynamic roles, we first need to understand what is wrong with static roles. Many organizations have jobs that are very similar, they just differ in some small detail. For example, all bank tellers are similar, the difference is just their branch office. Similarly, all the assistant jobs are pretty much the same. The difference is the department or section that they work for. Therefore, there is **Sales Assistant**, **Engineering Assistant**, **Logistics Assistant** - and a hundred or so similar roles. Almost all the privileges in those roles are the same. Of course, we can create an (abstract) role **Assistant** that will have all the common privileges. But we still need those hundreds of specific assistant roles. And then it gets even worse, because there may be **Senior Sales Assistant**, **Trainee Sales Assistant**, **Senior Engineering Assistant**, ...

The key to the role explosion is a realization that those "exploded" roles are created in an algorithmic way. Maybe we do not need **Sales Assistant**, **Engineering Assistant** and **Logistics Assistant** roles at all. Maybe we need just one **Assistant** role. The organizational unit (sales, engineering or logistics) is just a parameter to that role. Then the number of roles can be significantly reduced. This is what we call *parametric roles*.

Parametric roles are not your ordinary garden-variety static roles that just contain a set of privileges. Parametric roles need to be much smarter. E.g. the Assistant role needs an algorithm, that takes the organization unit as an input and it outputs a privileges that are appropriate for that organizational unit. This may be a simple expression that determines correct group name based on organizational unit name. But it may also be quite a complex code that determines most efficient location of home directories and other resources based on office location. There is no free lunch. The algorithm that was used to generate the number of "exploded" roles will not magically disappear. In case of parametric roles that algorithm needs to be placed in the role itself. But it still may be much easier to maintain a couple of expression than to maintain thousands upon thousands of roles.

The usual problem with parametric roles is, quite obviously, the presence of the parameters. The parameters cannot be stored with the role, as they are different for each assignment of the role. The parameters also cannot be stored directly with the user, as the user may have the same role assigned with a different set of parameters. Fortunately, midPoint was designed with this problem in mind and this was one of the big motivations to create a concept of *assignment*. Assignment is the right place to store the parameters, as it is the data structure that associates user with a specific role.

ExAmPLE is a very progressive company. Similarly to other corporations they have functional organizational structure. But their employees are also organized in teams. Each team can have a manager and ordinary members. The team membership is represented by custom attributes in LDAP server. Each user has two custom multi-value attributes: **exampleTeamMember** and **exampleTeamManager**. Both attributes expect team name as their value.

The naïve way to handle this would be to create two roles for each team. But there are hundreds of teams and that would be a maintenance nightmare. A smarter solution is to use parametric roles. There will be two roles only: **Team Member** and **Team Manager**. Those roles will take custom property **teamName** as parameter. But where does this property come from? It comes from assignment

extension. Each time the team role is assigned there needs to be a parameter in the assignment:

```
<user>
  <name>alice</name>
  ...
  <assignment>
    <extension>
      <exmpl:teamName>x-force</exmpl:teamName>
    </extension>
    <!-- Team Manager role -->
    <targetRef oid="aaa6cde4-0471-11e9-9b50-c743da469067" type="RoleType"/>
  </assignment>
</user>
```

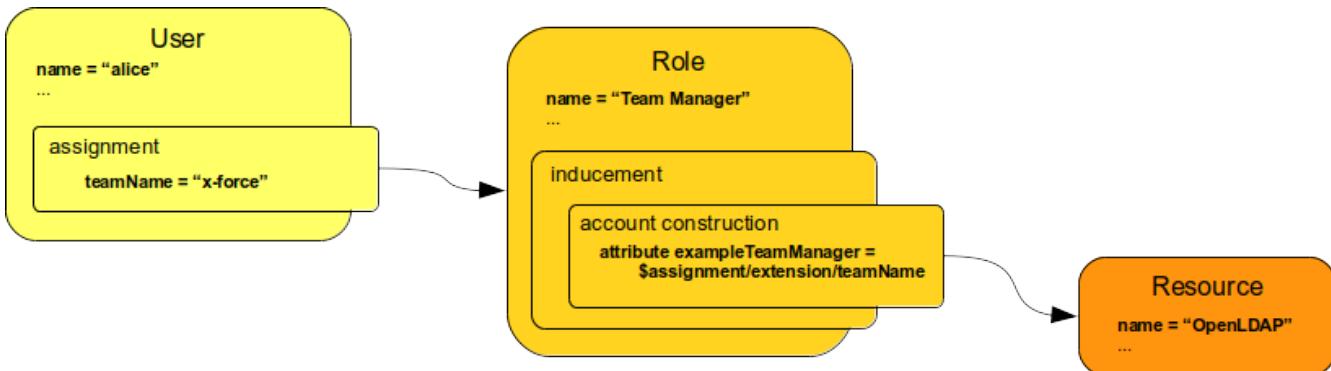
This is the first part of the solution. The second part are the roles. The roles needs to be a bit smarter to use the `teamName` parameter:

```
<role oid="aaa6cde4-0471-11e9-9b50-c743da469067">
  <name>Team Manager</name>
  ...
  <inducement>
    <construction>
      <!-- OpenLDAP resource -->
      <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
      <kind>account</kind>
      <attribute>
        <ref>ri:exampleTeamManager</ref>
        <outbound>
          <expression>
            <path>$assignment/extension/teamName</path>
          </expression>
        </outbound>
      </attribute>
    </construction>
  </inducement>
</user>
```

Resulting LDAP account looks like this:

```
dn: uid=alice,ou=people,dc=example,dc=com
objectclass: inetOrgPerson
...
exampleTeamManager: x-force
...
```

This setup is illustrated in the following diagram:



This is the basic mechanism of parametric roles. It is incredibly powerful mechanism. Unfortunately, current implementation of parametric roles in midPoint is quite limited. While midPoint was designed with parametric roles in mind, the implementation is not yet finished. Therefore support for parametric roles is quite limited. MidPoint core supports parametric roles quite well. Assignment parameters and mappings should work perfectly. However, the support for assignment parameters in midPoint user interface is very limited. In fact, the production-quality support is limited only to the couple of hardcoded parameters (`orgRef` and `tenantRef`) and even that leaves a lot to be desired. While we would like to improve support for parametric roles in midPoint, we have to listen to what midPoint subscribers are saying. Our development priorities are influenced by midPoint platform subscribers. So far platform subscribers prioritized other features and therefore there was not sufficient funding to completely finish user interface support for parametric roles.

i Roles may also explode due to other reasons. Application roles that were mentioned above may significantly contribute to role explosion. Also attempts to "atomize" the low-level roles as an attempt to create enough "material" to compose higher-level roles may lead to explosion. MidPoint has some mechanism that limit those effects. But perhaps the best approach for those cases could be summarized as "do not overdo it".

Metaroles

MidPoint roles are usually applied to users. But midPoint roles are universal. The roles can be applied to almost any midPoint object. Roles can be applied to users, organizations, services and even to roles themselves.

Simply speaking, meta-roles are roles applied to other roles. Ordinary role applies its characteristics to a user. Meta-role applies its characteristics to another role. This is perfectly possible in midPoint, as role can be applied to almost any midPoint object. Then why not apply a role to another role? This may seem like a pretty useless exercise, but the truth is that meta-roles are tremendously useful.

History is repeating, they say. And the fact is that repetition is a daily bread in almost all IDM deployments. E.g. many business roles have something in common. For example, the business roles have similar approval process. There may be role classes that have similar exclusion policies that are part of global segregation of duties (SoD) policy. There are roles that are tied to entitlements in a systematic way and so on. Roles, organizational units, services and other role-like objects tend to be quite similar. Therefore applying meta-roles to them can be very useful.

So far all the roles that we have seen were composed exclusively from inducements. And that made perfect sense, as all those things that were in the inducement did not apply to the role itself. Those things applied to users that the role was assigned to. But in this case we want to apply meta-role to a role. The effects of a meta-role should apply to the role, not to the user. Therefore assignment is used instead of inducement:

```
<role oid="6924fb9c-a184-11e9-840e-2feb476335f4">
    <name>Account Manager</name>
    <description>
        This is business role that corresponds to account manager job.
    </description>
    <assignment>
        <!-- Metarole assignment -->
        <targetRef oid="a3065910-a183-11e9-835c-0b6edc3d44c3" type="RoleType"/>
    </assignment>
    <inducement>
        <!--
            Privileges specific to account manager.
        -->
    </inducement>
</role>
```

```
<role oid="a3065910-a183-11e9-835c-0b6edc3d44c3">
    <name>Business metarole</name>
    <inducement>
        <!--
            Policies and constructions that should be applied to all
            business roles.
        -->
    </inducement>
</role>
```

This may seem similar to a role hierarchy. But it is a completely different animal. The crucial difference is that the meta-role is applied to the role and not to the user. The inducements in the meta-role often contain policies such as approval policy. Or construction clauses that create groups or organizational units. We usually do not want to create a group for each user. But we often want to create a group for a role. That's what meta-role can do.



Meta-roles are one of the stranger concepts of midPoint, but it goes well with midPoint philosophy. Meta-roles are roles that are applied to themselves. This is a reuse of an existing mechanism to create something new. This is very typical for midPoint. We always try to reuse an existing mechanism instead of reinventing a new one. And sometimes the result is quite unexpected and surprising. When we have designed the RBAC system for midPoint, we haven't thought about meta-roles at all. The meta-roles just appeared as a consequence of the design, a consequence that was absolutely unexpected. But we have quickly realized the potential that meta-roles have and we have put them to a full use.

A clever reader would probably notice that meta-roles can be used to set up different types of roles. We could have meta-role for application role, business role and so on. And clever reader, as always, would be right. However, there are few bits still missing here to create a full-featured type system. And those missing bits are implemented in a form of *archetypes*. Simply speaking, archetypes are meta-roles with some optimizations and improved user experience. But more on that later.

It may be difficult to understand the concept of meta-roles from such a short and very abstract description. But do not worry. As meta-roles are often used in midPoint, we will get back to the meta-roles on several occasions. Meta-roles often allow to simplify complex problems by creating a very elegant solutions. But for now it is enough to remember that roles can be applied to almost anything in midPoint, including themselves.

RBAC, ABAC And The Wildlife

This section is where we will get all thoughtful and philosophical. The people that are bored with philosophical questions should skip the rest of this chapter. We will also throw some dirt on almost every access control model in existence. Therefore the people that maintain dogmatic beliefs about IAM mechanisms should skip this section as well. On the other hand, open-minded people are quite likely to enjoy it.

Role-based Access Control (RBAC) is just one of many access control models. There many variants of RBAC and there are also other access control models that are based on a completely different paradigm. One such popular model is Attribute-Based Access Control (ABAC). ABAC is based on an idea that access to the systems can be determined dynamically just based on "attributes". Simply speaking, we can imagine ABAC as a one big algorithm that takes "attributes" as an input and decides whether access should be allowed or denied.

ABAC is very popular in the access management (AM) community because of its simplicity. And it all makes much sense as it is much simpler and faster to evaluate one expression than to sift through a mountain of roles. The problem with ABAC is manageability. ABAC assumes that all access control decisions could be based on algorithms and that they can be made anytime a decision is needed. However, that is almost never the case in larger practical deployments.

Many professionals responsible for identity management dream about complete automation of access control. It would be a marvel if an IDM system could automatically determine the privileges of every person simply based on the organizational unit and work responsibilities of that person. It would be perfect to get that information from an HR system, process it through a set of algorithms and automatically provision correct privileges to everybody. That is a very nice dream. But reality has a different idea. Such automated approach never really works in practice.

First problem is at the very start: HR data are almost never correct. There is very little motivation for the HR data to be completely correct. It is not a big issue if someone has a wrong job code or organizational unit code in the HR system. The business goes on, the salary is paid, everybody is happy. There is no really efficient feedback loop that would force corrections in HR data. Until the IDM system is deployed, that is. But it takes years or decades for a typical company to get to deployment of IDM system. At that point the HR data are beyond repair. The corrections that need to be done in the HR system are substantial. Even in small organizations it is very difficult to correct HR data manually. Bigger deployments absolutely require proper tooling to do that job. But

even with good tooling it can take a lot of time. Many IDM deployments were significantly delayed or even canceled because of data quality problems. This method does not work very well.

The fundamental problem here is in the overall approach. IDM system should not fail when the input data are wrong. There should be procedures how to correct those data. And the IDM deployment should not be delayed because of wrong input data. That would be like refusing to use your reading glasses because the text you are reading is wrong. IDM systems are essential tools that help you to clean up the data. The IDM system should be deployed and it should be used to manage data quality on day-to-day basis. It is naive to think that once the data are cleaned up they will stay clean. The processes that lead to data errors will continue, therefore data errors will appear all the time. The crucial insight is to accept that there will be data errors and to design the mechanisms to detect and correct them.

There are many manual and ad-hoc decisions that need to be made in practical IDM deployments. And not just during the deployment. Many ad-hoc decisions must be made during routine operation. Privileges need to be assigned manually to compensate for missing input data. Privileges need to be corrected, input data need to be temporarily overridden, policy exceptions has to be made. There are many things that need to do manually. Such decisions are made almost on day-to-day basis. For ABAC and similar systems this would mean that a policy needs to be updated on a day-to-day basis. And ABAC is not designed for that.

This leads to another big problem of ABAC and similar "flexible" access control models. Even if HR data are correct, the data usually do not provide all the information needed to completely provision the user with privileges. The HR data are often limited to organizational unit and formal code of the work position. However, this is often miles away from the job that user really does. The usual solution to this problem is that the user requests the privileges that are needed to do a job. Such request is then routed through appropriate approval process. And that *request* is a big problem for ABAC. What should the user request to get the privileges? Should the user request a change in ABAC policy? That would not be practical. Should the user request a new value for an attribute? Which attribute? And what value that should be? Can be somehow create a catalog of the things that a user can request? Once again, ABAC is not designed for this. But all those problems are very easy to solve in RBAC. User is expected to request a role. And it is quite easy to create a role catalog. But as ABAC does not have roles, there is nothing that a user can get a grip on. There is no "handle" that would allow the user to make sense from the ABAC policies.

This is all a consequence of yet another ABAC problem. While ABAC policy may be easy to set up, it is quite difficult to analyze and maintain. Which users are affected by this particular policy statement? How many users will be affected if I make this change to ABAC policy? ABAC systems would need a complex simulation algorithms to answer those questions. However, it is all quite trivial in RBAC. Policies are encapsulated into roles. Therefore only the users that have those roles are affected. The roles also divide the policy to a smaller, manageable pieces. Each of the roles can have its own state and lifecycle. Therefore it is not that difficult to work with two versions of the same role at the same time. Old version is still assigned to some users, but we are deprecating that and slowly migrating to a new version. Such continuous processes are difficult to do in ABAC.

And there is still one crucial problem when ABAC is used in provisioning scenarios. ABAC policies often benefit from the fact that complete data about the user accessing the system are available when the access control decision is made. The crucial part of that data is called *context*. This includes data such as time of day, network location of the user, recent events related to the user,

real-time estimate of the risk and so on. However, such data are simply not available in provisioning scenarios. Accounts are usually provisioned long before the first access to the account is made. Therefore, many of the advantages of ABAC are useless in identity management scenarios that rely on provisioning.

However, ABAC is not a complete failure. ABAC is very useful in customer-oriented identity and access management (CIAM). Customer identities are usually "lightweight" and the policies are simple. But when there is a need to manage employees, teachers, contractors and similar "heavyweight" identities then ABAC almost always fails.

The fact that ABAC fails in complex practical IDM deployments does not mean that RBAC is ideal. Quite the contrary. RBAC has problems of its own and the applicability of pure RBAC in practical IDM deployments is very limited. Many of the problems of RBAC model motivated engineers to develop ABAC and similar models. In fact, the "algorithmic" idea of ABAC is not entirely bad. Only if we had a way how to combine ABAC and RBAC ... Oh, but there is a way! We did it already.

MidPoint combines RBAC and ABAC by putting expressions into roles. We have seen that already. When role is assigned, the expressions in the role gets evaluated. And there can be any complex algorithm in the expression, even a complete ABAC policy. At least in theory. MidPoint expressions do not make access control decisions, because it is not the job of an IDM system to make such decisions. IDM system should set up the account. It provides the "material" for an authorization system to make a correct decisions. Therefore, midPoint goes as close to ABAC as a provisioning system can go. In an extreme case the entire ABAC policy can be implemented in outbound expressions in resource definition. But there is a good reason nobody does that.

Dividing the policy into smaller parts brings substantial advantage. Therefore many midPoint deployments are very RBAC-like. There are many roles and rich role hierarchies. Role expressions are used in moderation. But there are also deployments that are using parametric roles and role expressions extensively. In such cases there is a smaller number of roles, almost no role hierarchy, but the roles are smarter. Those are more ABAC-like deployments. But roles are still there. The roles act as "handles" for users to understand the policies, to give names to relevant parts of the policy. This combined approach works surprisingly well.

Of course, this idea is not new. Many RBAC-like systems use some kind of "smart" behavior inside the roles. However, so far we haven't seen anything as comprehensive as midPoint role-based access control model. Therefore we had to invent an impressive marketing name for our creation. Due to a momentary lapse of imagination we dubbed it Advanced Hybrid RBAC. But whatever you choose to call it, the fact is that this approach is very useful in practice. It can transform apparent chaos into something that can be efficiently managed.

Chapter 8. Object Templates

Scientists study the world as it is; engineers create the world that has never been.

— Theodore von Kármán

Identity management systems are often seen as integration engines that move data from one system to another system. This is indeed a very important part of the identity management functionality. But the things identity management systems do *internally* are crucial to all identity management deployments - especially those that deal with identity governance and compliance.

MidPoint does quite a lot of things that may not be entirely obvious on the outside. There are rules to apply, processes to drive, policies to enforce and so on. Those things are gaining utter importance at that strange boundary where identity management becomes identity governance. There are complex and very powerful mechanisms allowing midPoint to implement identity governance. But more on that later. We need to start with simple things. And the simplest of those internal mechanisms is the functionality of *object template*.

Object Templates

Data that come to midPoint are seldom complete and clean. Quite the contrary. The data that come from the "feeds" are often incomplete, they are not very precise and sometimes several sources may not even agree on a value for a particular data item. Inbound mappings can be used to sort out some of these problems. However, inbound mappings are designed to work in isolation. They work only for one particular resource. But it is often needed to gather data from several resources and then look at all of them at once. Inbound mappings cannot do that as they are tied to a particular resource. However, they can be used to gather all the relevant data in the user object. And then we can use some kind of a mechanism to have a look at user object when those data are gathered together. That is what object templates do.

Simply speaking, *object template* is a set of mappings that is applied to a particular midPoint object. For example, user template is applied to all user objects. The mappings in the object template can produce new values for the object. For instance, a very typical use of object template is computation of user's full name:

```

<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
    <name>User Template</name>
    <item>
        <ref>fullName</ref>
        <mapping>
            <strength>weak</strength>
            <source>
                <path>givenName</path>
            </source>
            <source>
                <path>familyName</path>
            </source>
            <expression>
                <script>
                    <code>
                        givenName + ' ' + familyName
                    </code>
                </script>
            </expression>
        </mapping>
    </item>
</objectTemplate>

```

The mapping above will compute the value of user's `fullName` property from `givenName` and `familyName` by using a simple Groovy expression. It is a weak mapping, therefore it will compute the full name only in case that it is not present already.

Object template can be used to do variety of things to all kinds of midPoint objects. This chapter will cover the most important functionality of object templates.

Importing an object template definition into midPoint will not do much. The template will not be used just by being imported. There can be several object templates for different types of objects, archetypes and even organizations. MidPoint will not know how to use the template. Therefore, the use of the template needs to be specified in a configuration. The simplest and most common way to use an object template is to configure its use in the system configuration.

```

<systemConfiguration>
    ...
    <defaultObjectPolicyConfiguration>
        <type>UserType</type>
        <objectTemplateRef oid="22f83022-b76d-11e9-8a30-6ffc11b23016"/>
    </defaultObjectPolicyConfiguration>
    ...
</systemConfiguration>

```

This configuration activates the object template for use by all object of `UserType` type. Therefore this template will be applied to all midPoint users.

User template is applied every time an object is changed or explicitly recomputed (e.g. on reconciliation). User template is applied after all the inbound mappings are processed. Inbound mappings often copy important data to the focal objects (e.g. user objects). Therefore, the template can work on a data that are summarized from all the resources.

Item Definitions In Object Template

We have already seen how object template can be used to apply mappings on particular items of midPoint objects. But object template can also do other tricks. We will have a look at some of them here.

The processing of an object template is almost always focused on particular items of an object. Therefore, almost all of the object template functionality is located in item element that references a particular item by its path:

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
    <name>User Template</name>
    <item>
        <ref>fullName</ref>
        ...
    </item>
    <item>
        <ref>assignment</ref>
        ...
    </item>
</objectTemplate>
```

The most common use of object template is to run mappings on items, such as the mapping to determine user's full name above. Target of the mapping is automatically set to the item for which it is specified. Sources of the mapping need to be defined explicitly. But the basic idea is, that the user template should take properties of user as inputs. In other words, user template works on the same user object both as input and output.

Object template mappings often use static (literal) values or a very simple expressions, but mapping conditions are used to control application of the value. The easiest way to explain this is to use a simple example:

```

<objectTemplate>
  ...
  <item>
    <ref>description</ref>
    <mapping>
      <source>
        <path>extension/hatSize</path>
      </source>
      <expression>
        <value>WARNING: Big brain!</value>
      </expression>
      <condition>
        <script>
          <code>hatSize &gt; 60</code>
        </script>
      </condition>
    </mapping>
  </item>
  ...
</objectTemplate>

```

This mapping works for **description** property of the user. The mapping sets a fixed warning text specified as text literal in the mapping by using value expression evaluator. However, the mapping is not setting that value for all the objects. The mapping is applied only for objects that satisfy the condition. The condition is set to trigger for all the users that have hat size larger than 60.

However, mappings are made to be relativistic. This means that mappings react to changes. The same principle applies to mapping conditions. They are also relativistic. Therefore the mapping reacts to changes in the condition state. When user's head grows and the hat size changes to a value over 60, then the mapping will add the warning. When the user's head shrinks, then the warning disappears.

It may look that this mechanism is too complicated, if you look at single-valued properties only. But it all starts to make sense in case of multi-value items. Such as the assignments. But more on that in the next section.

While mappings are the things that object template does almost all the time, the template can also do other interesting things. First of all, object template can tweak the schema. MidPoint comes with a rich schema that is prepared to be used. However, the schema is not a perfect fit for all the deployments. Previous chapter described a method to extend the schema. But what we should do if we want to change the built-in schema of midPoint? Yes, object template is the right answer. The **item** specification can be used to modify the way how midPoint applies the schema:

```

<objectTemplate>
...
<item>
    <ref>givenName</ref>
    <displayName>First Name</displayName>
</item>
<item>
    <ref>additionalName</ref>
    <displayName>Middle Name</displayName>
</item>
<item>
    <ref>familyName</ref>
    <displayName>Last Name</displayName>
</item>
...
</objectTemplate>

```

The "additional name" is a nice and generic term that can fit many cultural environment. But it is not very usual or intuitive in cultures that are not used to it (which means all the cultures). Therefore, almost all midPoint deployments that chose to use this property would like to rename it to something that feels more natural. Similarly for "given name" and "family name". We have expected that and object template can be used to modify some aspects of built-in schema.

Object template can also be used to override object multiplicity, especially to change mandatory item into optional. MidPoint insists on having name set for all the users. But we may be able to compute a name from other properties, such as other user names, employee number or other identifiers, if we don't want to present name item as mandatory in the user interface. We can compute the value of user's full name from given name and family name. Therefore user may leave full name blank in the user interface. But user interface is driven by the schema. As name is mandatory in the schema, also the user interface will insist that the name field should be filled in. But this can be changed in the object template:

```

<objectTemplate>
...
<item>
    <ref>name</ref>
    <limitations>
        <layer>presentation</layer>
        <minOccurs>0</minOccurs>
    </limitations>
    <mapping>
        ...
    </mapping>
</item>
...
</objectTemplate>

```

This configuration will make the name optional for the *presentation* purposes. This means that the

user interface will treat `name` as optional. But core midPoint engine will still require the `name` to have a value. This gives object template a chance to generate the value for `name`. However, this means that `name` will still be present as read-write item in the user interface. We do not want that as `name` is supposed to be immutable identifier. We want to present `name` as read-only item. This can also be achieved by object template by using access configuration:

```
<objectTemplate>
  ...
  <item>
    <ref>name</ref>
    <limitations>
      <layer>presentation</layer>
      <minOccurs>0</minOccurs>
      <access>
        <read>true</read>
        <add>false</add>
        <modify>false</modify>
      </access>
    </limitations>
    <mapping>
      ...
      </mapping>
    </item>
  ...
</objectTemplate>
```

Even immutable identifiers may need to change occasionally. There may be a bug in the identifier generator. Or some identifier has to change manually to adjust to the reality. Theoretically, every piece of the solution should play by the rules. But we know that rules have exceptions in the practice. Therefore, privileged users such as system administrator should be able to change the identifiers if really needed. The proper way how to do this would be to use authorizations and not object template. But we do not know how to use authorizations yet. Therefore this solution will have to do for now.



Object template can be used to adjust how midPoint user interface interprets the schema. But perhaps the most extreme measure is to eliminate certain item entirely. In fact, this happens quite often. MidPoint schema is rich and many deployments do not use all the items in midPoint schema. It makes little sense to present the items that are not used, therefore there is a way to tell midPoint that we want to completely ignore an item:

```

<objectTemplate>
...
<item>
    <ref>employeeNumber</ref>
    <limitations>
        <layer>presentation</layer>
        <processing>ignore</processing>
    </limitations>
</item>
...
</objectTemplate>

```

Object template can be used to do further tricks. It can be used to associate value enumeration with an item, e.g. to apply lookup table to a particular item. Object templates can be used to set up a validation expression for items. And a couple of other things. But more on that later.

Automatic Role Assignment in Object Template

Object templates are very flexible and they can be used for a lot of different things. But there is one particular usage of object template that appears in almost every deployment. It is an ability to automatically assign roles.

The basic idea is quite simple. An assignment is just an ordinary item. If we use object template mapping to populate that item with appropriate value, we will get automatic assignment of roles. Like this:

```

<objectTemplate>
...
<item>
    <ref>assignment</ref>
    <mapping>
        ...
    </mapping>
</item>
...
</objectTemplate>

```

The trick here is to set up the mapping correctly. The simplest case is an conditional assignment of a role. Let's suppose that we want to assign a Hatter role to everybody that has provided a hat size in user profile. We already know what to do, don't we? Let's use mapping condition:

```

<role oid="c38a5e6e-b783-11e9-b82f-ebb94fb5b6ec">
    <name>Hatter</name>
    ...
</role>

```

```

<objectTemplate>
    ...
    <item>
        <ref>assignment</ref>
        <mapping>
            <source>
                <path>extension/hatSize</path>
            </source>
            <expression>
                <value>
                    <targetRef oid="c38a5e6e-b783-11e9-b82f-ebb94fb5b6ec" type=
"RoleType">
                        ...
                    </value>
                </expression>
                <condition>
                    <script>
                        <code>hatSize as Boolean</code>
                    </script>
                </condition>
            </mapping>
        </item>
    ...
</objectTemplate>

```

Simple, isn't it? The `value` part in the expression is an inside of a new assignment to create. And the assignment will be created on a condition that `hatSize` has a non-null, non-empty and non-zero value (that is a built-in evaluation of booleans in Groovy). The real trick here is the relativity of mapping conditions. Assignments are multi-valued. Therefore it is important to know when to add a value and when to remove one. MidPoint evaluates the condition twice. The condition is evaluated for an object before a change is applied (old object) first. The condition is evaluated for an object after the change is applied (new object) once again. When the condition changes from `false` to `true`, `Hatter` role is assigned. When the condition changes from `true` to `false`, `Hatter` role is unassigned. Other assignments values are not changed by this mapping. Therefore many assignment mappings can happily coexist.

However, this is a very simple case. Typical midPoint deployments will have many roles. It is theoretically possible to create mappings like this for each and every role that has to be assigned automatically. But that would be a lot of repetitive work. And even worse, it is likely to become a major maintenance nightmare in the future. We are creative people and we do not really like repetitive work. And we really hate maintenance nightmares. Therefore it is perhaps no big surprise that there is a better way to do this.

The most common use case for automatic role assignment is to look up the role using some of its properties. For example, let's suppose that our HR system provides job codes for our employees. Therefore we have extended midPoint schema with a custom property `jobCode`:

```

<xsd:schema targetNamespace="http://example.com/xml/ns/midpoint/schema">
    ...
    <xsd:complexType name="UserTypeExtensionType">
        <xsd:annotation>
            <xsd:appinfo>
                <a:extension ref="c:UserType"/>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:sequence>
            ...
            <xsd:element name="jobCode" type="xsd:string"
                minOccurs="0" maxOccurs="1"/>
            ...
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

```

A user object that is created from an HR record looks like this:

```

<user>
    <name>bob</name>
    <extension>
        <exmpl:jobCode>S007</exmpl:jobCode>
    </extension>
    <fullName>Bob Brown</fullName>
    ...
</user>

```

The we do similar extension for roles. We extend role schema with custom `autoassignJobCode` property:

```

<xsd:schema targetNamespace="http://example.com/xml/ns/midpoint/schema">
    ...
    <xsd:complexType name="RoleTypeExtensionType">
        <xsd:annotation>
            <xsd:appinfo>
                <a:extension ref="c:RoleType"/>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:sequence>
            ...
            <xsd:element name="autoassignJobCode" type="xsd:string"
                minOccurs="0" maxOccurs="1"/>
            ...
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

```

And then we set up the roles:

```

<role oid="a1572de4-b9b9-11e9-af3e-5f68b3207f97">
    <name>Sales Manager</name>
    <extension>
        <exmpl:autoassignJobCode>S006</exmpl:autoassignJobCode>
    </extension>
    ...
</role>

```

```

<role oid="b93af850-b9b9-11e9-8c2c-dfb9a89635a0">
    <name>Sales Agent</name>
    <extension>
        <exmpl:autoassignJobCode>S007</exmpl:autoassignJobCode>
    </extension>
    ...
</role>

```

```

<role oid="b9d2b604-b9b9-11e9-bbc4-17d8e85623b4">
    <name>Sales Assistant</name>
    <extension>
        <exmpl:autoassignJobCode>S008</exmpl:autoassignJobCode>
    </extension>
    ...
</role>

```

We are almost there. The final part of this puzzle is an object template mapping that automatically assigns the roles according to job code. Naive solution would be to create one mapping for each job

code. But we do not want that. We want something smarter. We want a single mapping that can work for all these roles. Such mapping needs to dynamically look up the role when it is evaluated. It is certainly possible to create such mapping in Groovy script. But that is not entirely straightforward. And in fact, this use case is a very common one. Role autoassignment is a part of almost every IDM solution in one form or another. Therefore we have created a special expression evaluator to make this job easy. Enter **assignmentTargetSearch** expression evaluator:

```
<objectTemplate>
...
<item>
    <ref>assignment</ref>
    <mapping>
        <source>
            <path>extension/jobCode</path>
        </source>
        <expression>
            <assignmentTargetSearch>
                <targetType>RoleType</targetType>
                <filter>
                    <q:equal>
                        <q:path>extension/autoassignJobCode</q:path>
                        <expression>
                            <path>$jobCode</path>
                        </expression>
                    </q:equal>
                </filter>
            </assignmentTargetSearch>
        </expression>
    </mapping>
</item>
...
</objectTemplate>
```

Now, is this how "easy job" looks in midPoint? Yes, in fact, it is. But do not panic. Not yet. It all makes perfect sense once it is explained. The code above is a part of user template. It is a mapping that is producing assignments. This mapping has an ordinary source which is user's **jobCode**. This mapping also has an expression. But that expression is somehow extraordinary. It is not the usual **script**, **path** or **value**. The expression evaluator is **assignmentTargetSearch**. This is a special evaluator that looks for assignment target, creates a complete assignment from that target and provides that assignment as an output. The interesting part here is the way how **assignmentTargetSearch** looks for assignment target. First of all, there is a **targetType** clause, which tells the expression to look for roles as assignment target. And then there is a search filter. We have already seen midPoint search filters a couple of times, for example in a form of correlation expression. This is yet another use for search filters. In this case, the filter is used to look up appropriate role in midPoint repository. The filter looks up the roles by the **autoassignJobCode** value. But what value do we require here? Static value such as **S007** will not really help as that means that we will need one mapping for each role. We need to make this query dynamic and smarter. You probably know the answer already as the same approach was used in correlation expressions. We simply use an expression instead of static

value. In this case, we use path expression that point to `jobCode` variable. Job code is a source for this mapping, therefore it will be present as a variable in all the expressions of the mapping. So when the expression is processed for Bob then the final filter looks like this:

```
<filter>
  <q:equal>
    <q:path>extension/autoassignJobCode</q:path>
    <q:value>S007</q:value>
  </q:equal>
</filter>
```

That filter is used to look for a role. The `Sales Agent` role is found. And this roles is used to construct an assignment with the role as a target:

```
<assignment>
  <targetRef oid="b93af850-b9b9-11e9-8c2c-dfb9a89635a0" type="RoleType"/>
</assignment>
```

And that is it! That assignment is added to the user object. Which means that Bob has that role assigned now:

```
<user>
  <name>bob</name>
  <extension>
    <exmpl:jobCode>S007</exmpl:jobCode>
  </extension>
  <assignment>
    <targetRef oid="b93af850-b9b9-11e9-8c2c-dfb9a89635a0" type="RoleType"/>
  </assignment>
  <fullName>Bob Brown</fullName>
  ...
</user>
```

This single mapping will work for all the cases of all the current job codes and future job codes. That is how we like it.

Clever reader is surely amused at this point. As you can see, we have put an expression into an expression, so now midPoint can evaluate while it evaluates. There is an `assignmentTargetSearch` expression, inside that is a search filter and inside the search filter there is a path expression. This is one of the basic tenets of midPoint philosophy: we reuse and combine the mechanisms that we already have. The second expression could be any midPoint expression that makes sense here, e.g. it could be a script. The filter can be much more complex and it can have several expressions. Full power of midPoint is at your disposal here.



Of course, this could all have been done with one smart groovy script instead of

`assignmentTargetSearch` expression. And if you prefer it that way, you are free to do it. It will work. But assignment is a complex data structure and `assignmentTargetSearch` makes work with that structure easier. It can set up validity constraints, relation and other assignment details. It can also create the target on demand in case the target is not found. It is quite powerful. But perhaps the most important detail is that `assignmentTargetSearch` expression implements the use cases that are common in almost every IDM deployment. And that functionality is maintained and tested as a native part of midPoint. Therefore, you can simply reuse it in every midPoint deployment instead of copying, adapting, testing and bugfixing one big groovy script over and over again.

Autoassignment in Roles

Automatic assignment of roles in the object template is not the only option. This is midPoint, therefore there are usually several ways to do the same thing. For example, a mapping similar to that `assignmentTargetSearch` mapping used above can be used as an inbound mapping. And there is yet another way. Strictly speaking, this method has almost nothing to do with object template. But as we are talking about role autoassignment, this is a good opportunity to cover all the options here.

The statements that control automatic assignment of roles can be placed in the roles themselves:

```
<role oid="9f6add7c-b9bf-11e9-abf6-2348fc328f1">
    <name>Cook</name>
    ...
    <autoassign>
        <enabled>true</enabled>
        <focus>
            <mapping>
                <source>
                    <path>organizationalUnit</path>
                </source>
                <condition>
                    <script>
                        <code>
                            organizationalUnit?.norm == 'kitchen'
                        </code>
                    </script>
                </condition>
            </mapping>
        </focus>
    </autoassign>
</role>
```

The mapping in the `autoassign` part of the role will be evaluated approximately at the same time as other object template mappings. The mapping has no expression. There is no need to. MidPoint will prepare complete assignment data structure. The mapping just has to decide when to apply that assignment to the user and when not to apply it. That is what the condition is for. The expression in this mapping is optional. If an expression is specified, then such expression can be used to further set up the assignment. For example, it can set assignment activation, relation, parameters and so on.

But wait, why is there this strange `norm` thing in the condition? Remember about Polystrings? The `organizationalUnit` property is a polystring. Therefore it has `orig` part and `norm` part. In this case we want to compare the `norm` part, as the organizational unit name may be spelled as `Kitchen` or `KITCHEN`. But in all those cases the `norm` part will be `kitchen`.

If fact, there is little trap for the unwary here. The obvious way to specify the expression would be like this:

```
organizationalUnit == 'kitchen'           // This is wrong!
```

However, such expression will always return `false`. The reason is that different data types are being compared. The `organizationalUnit` property is polystring, while `'kitchen'` is a string literal. Polystring and string will never be equal regardless for their content. Therefore this form of the expression is wrong. Following forms may be used instead:

```
organizationalUnit?.orig == 'Kitchen'  
organizationalUnit?.norm == 'kitchen'  
basic.stringify(organizationalUnit) == 'Kitchen'
```

The later form is using `stringify()` method from basic midPoint function library. This method converts everything to string. Whatever data type is passed to this method the result is always a string that can be safely compared.

But let's get back to role autoassignment. When autoassign mappings are specified in the roles, midPoint will process in a way that is very similar to object template mappings. This has benefits, but there are also drawbacks.

The benefit of role autoassignment is manageability. The conditions are stored in roles themselves. Therefore they are bound to the object that they assign. It is there, right in front of administrator's eyes. It may also be a benefit if delegated administration is used. E.g. a role owner may manage role definition and the autoassignment condition in the same object. However, in that case beware of the expressions. MidPoint expressions are very powerful. In fact, they are way too powerful for secure delegated administration. Unconstrained midPoint expression can do pretty much anything. It can bring down the system, read memory, modify data, it can do whatever it likes to do. There are some safeguards that prohibit against accidental abuse, but a malevolent expression can easily circumvent them. If you allow a user to specify an expression, you are pretty much giving away keys to the kingdom. Therefore do not do it. At least not now. There is a code in midPoint that implements expression profiles. The goal of the profiles is to constraint expression to only allow safe operations. However, that functionality is not finished yet. If you are interested in this functionality, then midPoint platform subscription is the way to get it fully implemented.

Role autoassignment has another drawback and that is performance. All the autoassignment mappings need to be evaluated every time that user is recomputed. This means that all the roles that contain the mappings need to be retrieved from midPoint repository. This may not be a big deal for a small deployment with thousands of users and hundreds of roles. But the performance hit is likely to be significant as the number of users and roles grows. Therefore, roles autoassignment is not enabled by default. It has to be explicitly enabled in system configuration:

```

<systemConfiguration>
    ...
    <roleManagement>
        <autoassignEnabled>true</autoassignEnabled>
    </roleManagement>
    ...
</systemConfiguration>

```

But perhaps the most significant drawback of role autoassignment is that the mapping needs to be in every role. There is no way how to use this mechanism to handle autoassignment of many roles with just one mapping. But object template mappings can do that easily. Therefore, many deployments chose to implement automatic assignment of roles by the means of object template or inbound mappings.

Iteration

There are some use cases that pop out in IDM solutions all the time. One such case is the problem of finding a unique identifier. This is a concern for almost any identifier, but it is particularly painful when it comes to usernames. In midPoint world this means finding a value for name property. This property must be unique for almost all the data types that midPoint supports.

The rational way would be to base usernames on something that is already unique and immutable such as employee numbers or student identifiers. But those tend to be long numbers and people often hate them. Therefore, many deployments chose to base usernames on real names of the user. We can easily generate username for Alice Anderson. Maybe **aanderson** would be a good fit? And this can indeed work quite well. Until Albert Anderson is hired. Then we need to get creative. Obviously, **alander** will not work here. What about **alianderson** and **albanderson**? Oh no, we have this ancient system that allows only ten characters in the username. And **alianderson** is too long. What is even worse is that we would need to change Alice's username. She will get really mad about it. Not to mention changing usernames for pretty much everybody. That won't do. Let's go the usual way. Let's have **aanderson** and **aanderson1**. It is not elegant. But it will do the job. And Alice will not get mad. You know, she is really scary when she gets mad.

This use case is so common that even very early midPoint versions supported it. This feature is called *iteration* in midPoint terminology. The name suggests how the mechanism works. First step is an attempt to create a user object in a perfectly normal way. This means that username **aanderson** is created for Albert Anderson. The midPoint checks if that username is unique. In this case the username is not unique as it is already taken by Alice. That is the point when midPoint starts iterating. MidPoint creates *iteration token*. Iteration token is a short string that changes in every iteration. In our case, the iteration token will be set to **1**. Then midPoint re-evaluates all the object template mappings. Mappings that are supposed to create a unique values need to use that token. They should look like this:

```

<objectTemplate>
  ...
  <item>
    <ref>name</ref>
    <mapping>
      <source>
        <path>givenName</path>
      </source>
      <source>
        <path>familyName</path>
      </source>
      <expression>
        <script>
          <code>
            givenName?.norm[0] + familyName?.norm + iterationToken
          </code>
        </script>
      </expression>
    </mapping>
  </item>
  ...
</objectTemplate>

```

When this mapping is evaluated for the first time, the iteration token is empty. Therefore it will make no difference for the normal processing. But when the mappings are re-iterated, the token is set to **1**. Result of this mapping will be **aanderson1**. Which is unique username. Therefore iteration stops there and normal processing continues. In case that even **aanderson1** is not unique, the iteration continues. Usernames **aanderson2**, **aanderson3** and other variants are tried. The iteration continues until a unique username is found or until iteration limit is reached.

Iteration functionality is disabled by default. Therefore any conflict in username will result in hard error. This makes sense, as no amount of iterations will make any difference until the iteration token is used in the expressions. We also want to set maximum number of iterations. E.g. there may be a bug in the mappings that may cause endless iterations. The iteration functionality can be enabled by specifying **iterationSpecification** element and setting iteration limit:

```

<objectTemplate>
  ...
  <iterationSpecification>
    <maxIterations>5</maxIterations>
  </iterationSpecification>
  ...
</objectTemplate>

```

Iteration tokens are strings that are created from iteration number. It is the iteration number that really matters for midPoint. Iteration token can take variety of forms, it can be numeric, it may be alphanumeric, fixed length, variable length or anything else. Some mappings will not use the token at all. E.g. mappings that subsequently add letters from given name to the username. Therefore,

both iteration number and iteration token are exposed to the mappings. There are two variables:

- `iteration` variable contains iteration number. It is always numeric, starting with zero (`0`). Iteration zero means normal processing. Iteration one happens after the first conflict.
- `iterationToken` variable contains a string that is derived from the iteration number.

There is default algorithm that derives iteration tokens from iteration number. The algorithm is illustrated in following table.

Iteration	The value of <code>iteration</code> variable	The value of <code>iterationToken</code> variable
Normal processing	<code>0</code>	<code>"" (empty string)</code>
First iteration	<code>1</code>	<code>"1"</code>
Second iteration	<code>2</code>	<code>"2"</code>

The algorithm is designed to put empty value in the `iterationToken` during normal processing. The idea is that `iterationToken` variable can be safely used for both the normal processing and the iterations. This is just a default algorithm and it will not fit all the deployments. Therefore, a custom mechanism to derive iteration token can be specified. For example, we may not like to have `aanderson` and `aanderson1`. Which one of these is number one and which is number two anyway? Let's skip `aanderson1` and let's use `aanderson2` for the first iteration. The iteration number cannot be changed as the iteration sequence is fixed. But there is no problem for iteration 1 to produce iteration token `"2"`. This can be achieved by specifying a custom algorithm for the token:

```
<objectTemplate>
  ...
  <iterationSpecification>
    <maxIterations>5</maxIterations>
    <tokenExpression>
      <script>
        <code>
          if (iteration == 0) {
            return ''
          } else {
            return iteration + 1
          }
        </code>
      </script>
    </tokenExpression>
  </iterationSpecification>
  ...
</objectTemplate>
```

This algorithm will produce sequence of `aanderson`, `aanderson2`, `aanderson3` and so on.

Iteration number and iteration token is the same for the entire object template. All the mappings will see the same value and all the mappings are recomputed when there is a need to re-iterate.

This means that iteration token can be used in other mappings. For example, use of the token in e-mail address is a very common case:

```
<objectTemplate>
...
<item>
    <ref>emailAddress</ref>
    <mapping>
        <source>
            <path>givenName</path>
        </source>
        <source>
            <path>familyName</path>
        </source>
        <expression>
            <script>
                <code>
                    givenName?.norm + '.' + familyName?.norm
                    + iterationToken + '@example.com'
                </code>
            </script>
        </expression>
    </mapping>
</item>
...
</objectTemplate>
```

This mapping will produce a sequence of `albert.anderson@example.com`, `albert.anderson2@example.com` and so on (assuming that customized token expression is also applied). Shared value of `iterationToken` means that the values of e-mail address are consistent with the values of username. If username of `aanderson2` is generated then the e-mail address will be `albert.anderson2@example.com`. The same iteration token is used.

However, it all becomes interesting when it comes to e-mail addresses and other identifiers that are publicly exposed. It is one thing to have username `aanderson2`. That username is used to log into the system. But otherwise is it not very visible outside of the system. However, an e-mail address is exposed to a lot of people. It may be strange to have e-mail address of `albert.anderson2@example.com` while there is no `albert.anderson@example.com` in the company. This can be solved by making the mapping for e-mail address smarter. It can ignore the iteration token and try to create an e-mail address without the token. But in that case it needs to explicitly check for uniqueness. There are two ways to do that. First method is to check for e-mail address uniqueness inside the e-mail mapping. There is a `isUniquePropertyValue(...)` method in midPoint function library that is designed for this purpose:

```

<objectTemplate>
...
<item>
    <ref>name</ref>
    <mapping>
        <source>
            <path>givenName</path>
        </source>
        <source>
            <path>familyName</path>
        </source>
        <expression>
            <script>
                <code>
                    def plainAddress = givenName?.norm + '.' + familyName?.norm
                        + '@example.com'
                    if (midpoint.isUniquePropertyValue(focus, 'emailAddress',
                        plainAddress)) {
                        // Bingo! We have unique address
                    } else {
                        // Address not unique.
                        // We have to use iteration token here.
                    }
                </code>
            </script>
        </expression>
    </mapping>
</item>
...
</objectTemplate>

```

The problem with this approach is that there may be corner cases. We might need to force another iteration even if the username is unique. MidPoint checks only for uniqueness of username by default. But it is possible that even if `aanderson2` username is available, the `albert.anderson2@example.com` address is already taken. This may be an error in the data, administrator's mistake or it may be a remain of retired Albert Anderson senior that worked in the company years ago, but his e-mail address was never deprovisioned. The e-mail address mapping can detect this situation. But what it is supposed to do with that. It makes no sense to have username `aanderson2` and e-mail address `albert.anderson3@example.com` or `albert.anderson.X@example.com` or anything similar. What would make sense is to re-iterate and produce username `aanderson3` and e-mail address `albert.anderson3@example.com`. That would be consistent. But the mapping cannot do that. Therefore, there is another iteration expression for this purpose: *post-iteration condition*. It is a condition that will be executed after the iteration is completed. If the condition returns `true`, then the iteration will be accepted as valid and the generated values will be used. If the condition returns `false`, then midpoint will re-iterate and yet another iteration will be tried.

```

<objectTemplate>
...
<iterationSpecification>
...
<postIterationCondition>
<script>
<code>
    def email = ... // Code to generate or retrieve e-mail
    return midPoint.isUniquePropertyValue(focus,
        'emailAddress', email)
</code>
</script>
</iterationSpecification>
...
</objectTemplate>

```

The code above does not do much in case the e-mail address is unique. It returns `true`, the iteration is accepted and everything goes as usual. But in case that the e-mail address is not unique, the code returns `false`. In that case, midPoint will discard all the results of the iteration, increment iteration counter and re-try the iteration.

There is yet another mechanism that can be used here: *pre-iteration condition*. It is a condition that will be executed prior to iteration. If it returns `true`, then the iteration will continue. If it returns `false`, then midPoint will re-iterate. The difference here is that this condition will be executed before all the other mappings are evaluated. Therefore, it may be used to avoid evaluation of expensive mappings just to discard the values that they produce.

Finding identifier values and uniqueness checks are messy stuff. And they are not entirely reliable. There is a delay between the time when uniqueness is checked and the time when the record is actually written into database. Therefore strange things can happen. Duplicate identifiers may be generated or attempt to create a user may end up with an error. Especially under high loads. The delay between check and write cannot be entirely avoided. We could lock the data during that time, but that would have significant impact on system performance. What we can do to improve the situation is to check the uniqueness on database level and gracefully handle the errors. This is currently implemented only for usernames and even for that the implementation is not perfect. Implementation of strict uniqueness constraints for other properties is possible, but it is no easy endeavor. The values need to be normalized, this can influence database schema and so on. But it is feasible. In case you are interested, midPoint platform subscription is the best approach for you.

When it comes to human-friendly identifiers, there is yet another trouble. People tend to change their minds. And then they have all kinds of crazy ideas such as the urge to get married. The result is that the names of people change. And in fact they change surprisingly often. When user-friendly identifiers are used, change in user's name usually means a change in the identifiers. This is known as the *rename problem* and you can observe a glimpse of fear in the eyes of all experienced IDM engineers every time it is mentioned. Overall, midPoint handles renames very well. Primary identifier of any midPoint object is an OID, not a name. And OIDs do not change. Therefore, as long as midPoint is concerned, nothing special happens when user's name is changed. The change is picked up by mappings, recomputed and stored. However, iterations and uniqueness checks may

complicate the things here. MidPoint remembers the iteration number for all objects that went through an iteration process. This is necessary to get the same results from the mappings every time that they are recomputed. Otherwise the identifiers may get re-generated on every recompute. But there is a drawback to this approach. Let's suppose that Carol Cooper had username `ccooper2`. She got married and now her name is Carol Cunningham. Even though there is no `ccunningham` in the system, her generated user name will be `ccunningham2`. The iteration token is remembered. The rename scenarios can be very treacherous. We always recommend to test them thoroughly in any project where user renames are possible.

Another drawback of those iterating algorithms is scalability and performance. Every time there is a conflict the algorithm need to go through all the iterations. How many people named John Smith can be in a large user population? We can easily get to `jsmith42`. This means that the next John Smith will need to go through 42 iterations before the system figures out that the next available username is `jsmith43`. And this gets worse with every John Smith added to the system. Therefore, this iterative approach is not suitable for generating identifiers that are likely to require a lot of iterations. Generating UNIX user and group numbers is a good example for identifiers that would surely cause a disaster if an iterative approach is used. Fortunately, there is another mechanism in midPoint that can support generation of such identifiers: sequences. But more on that later.

Overall, the best strategy is to avoid using those generated human-friendly identifiers altogether. The best choice would be something that is already unique, immutable and reasonably short. Something like employee number, student identifier or partner ID are usually suitable. If that is not acceptable, then the second best approach is to keep the algorithms simple. The simpler it is the less likely it is to fail.

Includes

There are many ways to apply an object template to an object. The template can be set globally in system configuration, it can be set by an archetype or even by an organizational unit. However, only one object template can be active for any particular object at one time. Yet, there are often mappings that need to apply universally. For example, we may want to generate full name using the same algorithm for all users, regardless of their archetype. Or we may want to automatically assign some roles to all users regardless of their organizational units. For that reason there is mechanism to include one object template in another:

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
    <name>Default User Template</name>
    <item>
        <ref>fullName</ref>
        <mapping>
            ...
        </mapping>
    </item>
</objectTemplate>
```

```

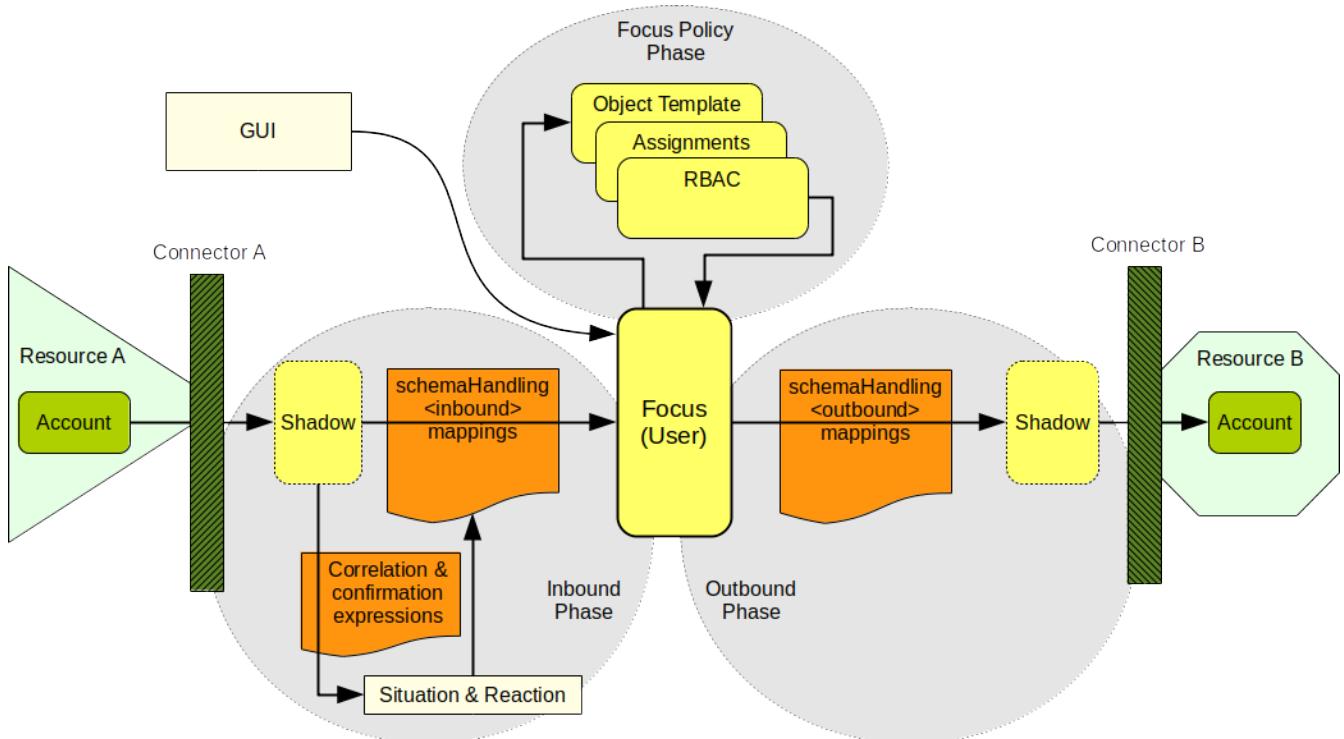
<objectTemplate oid="60eab6a8-ba87-11e9-b9a3-bbb8418de4d5">
  <name>Special User Template</name>
  <includeRef oid="22f83022-b76d-11e9-8a30-6ffc11b23016"/>
  <item>
    <ref>employeeNumber</ref>
    <mapping>
      ...
    </mapping>
  </item>
</objectTemplate>

```

In this case, the special user template includes all the mappings from default user template. Therefore both mapping for `employeeNumber` and mapping for `fullName` will be processed.

Combining the Ingredients

It is time to put all the bits and pieces together. So far we have been talking about provisioning, inbound synchronization, schema, RBAC and object templates. Let's see how all the parts fit together:



Everything starts with a synchronization process, whether it is reconciliation or live synchronization. Synchronization process invoked the connector for the source resource (Resource A). The connector retrieves the data from the source system. Shadow objects are created for all the source accounts as soon as the data set foot in midPoint. Correlation expression is evaluated for all new accounts to find their owners. Once we have owner of the account, we can execute inbound mappings. This is the way how account data are reflected to midPoint user object, which is a *focus* of the computation.

Next couple of steps is all about the *focus*. This is the part where object templates are executed,

assignment and roles are evaluated. Assignments and roles may contain *construction* statements. Those are just collected at this stage. They are not evaluated yet. This focus policy phase of computation is all about the focus. Which means that user object is both the input and output of this computation.

Outbound phase takes place next. In this phase the focus of the computation (user) is projected to accounts. This is the time when *constructions* are processed and the mappings inside them are evaluated. Those constructions were collected from the assignments in the previous phase. They are combined with outbound mappings specified in resource *schema handling*. All of that is mixed together, sorted to resource accounts, all the values are computed. This is also the time when attribute-level reconciliation takes place. We know what attributes the account should have, therefore we can compare that with the values that the attributes have in reality. When all of that is computed and processed, then a connector is used to update the target resource (Resource B).

This picture is still not entirely complete. It does not show policy rules, existence mappings, approval processes, hooks and good deal of other advanced features. But this picture is good enough for now. It is good enough to create a simple solution.

Complete Deployment Example

We have all that we need to create a simple but mostly complete identity management solution. Our environment and solution outline:

- HR system is a data input. It exports employee data into a CSV file. Employee number is a primary key, there is employee first and last name and job code. There is no username or password.
- We need to feed employee data into midPoint. Which means that we need to configure synchronization.
- We need to generate unique and user-friendly username, compute full name and generate a random initial password.
- We need to automatically assign roles based on job code from the HR system.
- We need to automatically provision account to LDAP server and CRM database table.

We can do that if we put together all that we have learned so far. Even though this is still quite a simple example, the complete configuration is too large to put all of it into this book. It will take too much space. And after all the detailed explanation in the previous chapters it will also get a bit boring. Therefore we will show only the interesting pieces of the configuration here. Complete configuration can be found in the usual place. Please see [Additional Information](#) chapter for details. These files represent the final configuration, the expected state at the end of this chapter. Therefore, if you want to follow instructions in this chapter step-by-step you have to choose appropriate parts of the files to import. Or you can just import everything and use the following text as an explanation of the effects that you see.

Let us start with an HR resource. This is mostly the same resource definition as we have seen in the [Synchronization](#) chapter. But there are few differences. First of all, the data feed is a bit different. We have a new **jobcode** column there. It looks like this:

hr.csv

```
"empno","firstname","lastname","jobcode"  
"001","Alice","Anderson","S006"  
"002","Bob","Brown","S007"  
...
```

Of course, the HR resource definition has to reflect those changes. We have defined a new custom user property **jobCode** in our extension schema:

extension-example.xsd

```
<xsd:schema ...>  
  <xsd:complexType name="UserExtensionType">  
    <xsd:annotation>  
      <xsd:appinfo>  
        <a:extension ref="c:UserType"/>  
      </xsd:appinfo>  
    </xsd:annotation>  
    <xsd:sequence>  
      ...  
      <xsd:element name="jobCode" type="xsd:string"  
                   minOccurs="0" maxOccurs="1">  
      ...  
    </xsd:element>  
  </xsd:sequence>  
</xsd:complexType>  
...  
</xsd:schema>
```

The jobcode column is mapped to jobCode extension property in HR resource inbound mapping:

```

<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
  ...
  <schemaHandling>
    <objectType>
      ...
      <attribute>
        <ref>ri:jobcode</ref>
        <displayName>Job code</displayName>
        <inbound>
          <target>
            <path>$focus/extension/jobCode</path>
          </target>
        </inbound>
      </attribute>
      ...
    </objectType>
  </schemaHandling>
  ...
</resource>

```

The rest of the mappings that are defined in the HR resource is a bit boring. The interesting thing is the mapping that is not there at all. The mapping for username (property name of the user object) is missing. We will not generate username in the inbound phase. We just do not have enough data to responsibly generate username just yet. Inbound phase is still running, user object is not fully populated yet. Let's postpone the decision about username for later.

Synchronization part of the HR resource definition is also a pretty standard one. This resource is an authoritative source. Accounts will be correlated by the `empno` column matching the `employeeNumber` user property. Linked accounts will be updated and new users will be created for unmatched accounts. It is all the same routine as we have already described in [Synchronization](#) chapter.

Perhaps the most interesting part of this setup is object template. The template has several responsibilities:

- Compute full name from first name and last name.
- Generate unique username.
- Generate e-mail address.
- Automatically assign basic employee role.
- Automatically assign the roles based on job code.

Let's start with the simple thing: generating full name. At this point this is probably a no-brainer:

object-template-user.xml

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
  ...
  <item>
    <ref>fullName</ref>
    <mapping>
      <source>
        <path>givenName</path>
      </source>
      <source>
        <path>familyName</path>
      </source>
      <expression>
        <script>
          <code>givenName + ' ' + familyName</code>
        </script>
      </expression>
    </mapping>
  </item>
  ...
</objectTemplate>
```

This is really simple. But it is much harder for username. We want to generate a user-friendly username. We could simply use user's last name. But this is very likely to create conflicts. Therefore let's combine last name with the first letter of first name. We will get nice usernames such as aanderson, bbrown and so on. But there is still a chance of username conflict. So let's add iteration tokens into the mix. Like this:

object-template-user.xml

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
  ...
  <item>
    <ref>name</ref>
    <mapping>
      <source>
        <path>givenName</path>
      </source>
      <source>
        <path>familyName</path>
      </source>
      <expression>
        <script>
          <code>
            if ( givenName == null && familyName == null ) {
              return null
            }
            if ( familyName == null ) {
              return givenName?.norm + iterationToken
            }
            if ( givenName == null ) {
              return familyName?.norm + iterationToken
            }
            givenName?.norm[0] + familyName?.norm + iterationToken
          </code>
        </script>
      </expression>
    </mapping>
  </item>
  ...
</objectTemplate>
```

This looks a bit more complicated than you have expected, doesn't it? The basic idea is simple, so why won't equally simple expression work? Maybe something like this?

```
givenName[0] + familyName + iterationToken
```

The devil is, as usual, in the details.

Firstly, good part of any programming is error handling. Hence all the if-then statements. It may look like those situations cannot happen in our little example. All the HR records have both first and last name set, anyway. Therefore they will never be `null` in the expression, will they? In fact, they will. This is one small peculiarity of midPoint expressions. MidPoint works in a relative way. Therefore midPoint often evaluates *old* values of attributes and properties to figure out which values to remove. The old values for any new user are `null`. Therefore it may happen that the expression is evaluated with `null` inputs. This may seem a bit annoying at the beginning. But you will be more than grateful that your expressions are properly sanitized and null-safe when you get

to work with real data. Reality always finds a way to bring surprises.

Secondly, midPoint is built with multi-national environment in mind. It is 21st century already and unicode is everywhere. *Almost* everywhere, that is. It is expected that the HR system stores names with full national characters, such as [Radovan Semančík](#). But it is still not a common practice to use national characters in usernames, e-mail addresses and so on. Therefore we usually want to normalize the national characters to their ASCII-7 equivalents. That is what PolyString is for and that is what the `norm()` methods are doing. The result is that the generated username will be [rsemancik](#) instead of [RSemančík](#).

But there is still one piece missing. We want to enable iteration to resolve naming conflicts. Otherwise poor Arnold Anderson won't have his accounts created because [aanderson](#) username is already taken by Alice. We can enable iterations like this:

`object-template-user.xml`

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
    ...
    <iterationSpecification>
        <maxIterations>5</maxIterations>
        <tokenExpression>
            <script>
                <code>
                    if (iteration == 0) {
                        return ''
                    } else {
                        return iteration + 1
                    }
                </code>
            </script>
        </tokenExpression>
    </iterationSpecification>
    ...
</objectTemplate>
```

The `maxIteration` part up there is quite straightforward. We want to have some limit on the number of iterations as we do not want to iterate forever. Most iteration sequences are short in practice. If the iterative approach cannot find a match in several steps, then perhaps the iteration is not a good method anyway. Therefore the limit is usually not a problem. But having a limit makes a huge difference for troubleshooting. Most infinite iteration loops are caused by configuration errors. And it is much better to get an error after a couple of seconds than to wait forever.

The second part of the iteration configuration is also quite clear for people that read this chapter carefully. The default iteration token sequence is `"", "1", "2", "3"` and so on. But that would give us [aanderson](#), [aanderson1](#), [aanderson2](#) and so on. We do not want to have [aanderson](#) and [aanderson1](#) as that would be confusing. Therefore we chose to skip the `"1"` token and start with `"2"`. The custom iteration token expression does just that.

As soon as we have the mapping for name in place we can start testing the configuration. Therefore go ahead and import the HR resource, import object template, set the object template in the

configuration and do not forget to replace the HR CSV file. If you did experiments with previous configuration, it can be helpful to clean up midPoint by using the “delete all identities” process. When everything is set up, you can try to manually import a single account from the HR resource by using the “import” button on the page where you can list resource accounts. Once the basic configuration works, you can test iterations by adding Arnold Anderson to the HR CSV file and importing the account. Do not forget to switch from repository to resource view by clicking on the *Resource* button on the top-left side of the page. There is no synchronization task running, therefore MidPoint haven’t see Arnold’s account yet. You have to instruct midPoint to explicitly look at the resource. Once Arnold’s account is imported a non-conflicting username should be selected for him:

The screenshot shows a table-based interface for managing users. The columns are labeled: Name, Given name, Family name, Full name, Email, and Accounts. The rows contain the following data:

	Name	Given name	Family name	Full name	Email	Accounts
<input type="checkbox"/>	aanderson	Alice	Anderson	Alice Anderson		1
<input type="checkbox"/>	aanderson2	Arnold	Anderson	Arnold Anderson		1
<input type="checkbox"/>	administrator	midPoint	Administrator	midPoint Administrator		0

At the bottom left are icons for adding (+), deleting (trash), and filtering (magnifying glass). At the bottom right are navigation buttons for page 1 of 3, and a settings gear icon.

We need to determine e-mail address next. In our case the mapping for e-mail address is quite similar to username mapping:

object-template-user.xml

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
  ...
  <item>
    <ref>emailAddress</ref>
    <mapping>
      <source>
        <path>givenName</path>
      </source>
      <source>
        <path>familyName</path>
      </source>
      <expression>
        <script>
          <code>
            if ( givenName == null && familyName == null ) {
              return null
            }
            if ( familyName == null ) {
              return givenName?.norm + iterationToken +
                '@example.com'
            }
            if ( givenName == null ) {
              return familyName?.norm + iterationToken +
                '@example.com'
            }
            givenName?.norm + '.' + familyName?.norm +
              iterationToken + '@example.com'
          </code>
        </script>
      </expression>
    </mapping>
  </item>
  ...
</objectTemplate>
```

This mapping should be quite understandable now. There are the same checks for special cases. Then the main expression at the end combines given name and family name in a slightly different way to get an e-mail address. This is just a simple example for e-mail address that is a good fit for a book. However, dealing with e-mail address is a bit more difficult in practice. A clever reader can surely discover a couple of obvious issues. Firstly, the expression is using the same iteration token than the username mapping is using. Therefore the e-mail address for Arnold Anderson will be **arnold.anderson2@example.com**.

	Name	Given name	Family name	Full name	Email	Accounts	Actions
<input type="checkbox"/>	 aanderson	Alice	Anderson	Alice Anderson	alice.anderson@example.com	1	Actions
<input type="checkbox"/>	 aanderson2	Arnold	Anderson	Arnold Anderson	arnold.anderson2@example.com	1	Actions
<input type="checkbox"/>	 administrator	midPoint	Administrator	midPoint Administrator		0	Actions

[+ New](#) [Edit](#) [Delete](#) [Import](#) [Export](#)
1 to 3 of 3 [<<](#) [<](#) [1](#) [>](#) [>>](#) [Settings](#)

This is not exactly what we want. Ideally, we would like to use much simpler versions [arnold.anderson@example.com](#). In this case there is no conflict with [alice.anderson@example.com](#). But midPoint does not consider e-mail address to be an identifier, therefore it does not check for its uniqueness. Also, there is only one iteration token that is reused for all the expressions in all object template mappings. There are also primary e-mail accounts and account aliases, dealing with account renames and temporary assignment of e-mail aliases and so on. Overall, dealing with e-mail addresses is far from easy. Some of those issues can be solved with pre-iteration or post-iteration conditions. But it is quite likely that a completely custom code will be needed for a more complex cases.

Role autoassignment is the next step. Let's start with something simple. All the records that come from the HR resource are employee records. Therefore let's assign Employee role to all of them. The easiest way to do that is to use inbound mapping of HR resource:

```

<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
  ...
  <schemaHandling>
    <objectType>
      ...
      <attribute>
        <ref>ri:empno</ref>
        ...
        <inbound>
          <expression>
            <value>
              <!-- Employee role -->
              <targetRef oid="86d3b462-2334-11ea-bbac-13d84ce0a1df"
                type="RoleType"/>
            </value>
          </expression>
          <target>
            <path>assignment</path>
          </target>
        </inbound>
      </attribute>
      ...
    </objectType>
  </schemaHandling>
  ...
</resource>

```

This mapping is quite straightforward. Its expression produces a static `targetRef` value that is placed in user's assignment. The strange thing here is the placement of this mapping. It is placed in the section that corresponds to `empno` attribute. This is inbound mapping and it just has to be placed somewhere. Any reasonable attribute would do. It does not really matter into which attribute it is placed as it ignores attribute value anyway.

That was a very simple static mapping. It is perhaps too simple for practical use as there not many cases when a role is assigned both automatically and unconditionally. The mapping above can be slightly improved by adding a *condition* to the mapping. But that still has some limitations. For example, inbound mappings cannot have more than one input. And being an inbound mapping, this can only work with account attributes and therefore its behavior cannot be influenced by user data changed in the user interface.

Autoassignment in inbound mapping is still useful and in fact it is also used quite often. But there is another way that is much more popular: autoassignment in object template mapping. That is what we are going to do next. We are going to handle role autoassignment based on job code.

We want to demonstrate autoassignment in object template. Our object template works with user object both as input and output. It cannot (or rather *should not*) reach out to the HR account to get the value of `jobcode` attribute. We have to do that the other way around. We have to map HR account attribute `jobcode` to midPoint user property `jobCode` by using an inbound mapping:

resource-csv-hr.xml

```
<resource oid="03c3ceea-78e2-11e6-954d-dfdfa9ace0cf">
  ...
  <schemaHandling>
    <objectType>
      ...
      <attribute>
        <ref>ri:jobcode</ref>
        <inbound>
          <target>
            <path>$focus/extension/jobCode</path>
          </target>
        </inbound>
      </attribute>
      ...
    </objectType>
  </schemaHandling>
  ...
</resource>
```

We can easily use the job code in object template mapping now:

object-template-user.xml

```
<objectTemplate oid="22f83022-b76d-11e9-8a30-6ffc11b23016">
  ...
  <item>
    <ref>assignment</ref>
    <mapping>
      <strength>strong</strength>
      <source>
        <path>extension/jobCode</path>
      </source>
      <expression>
        <assignmentTargetSearch>
          <targetType>RoleType</targetType>
          <filter>
            <q:equal>
              <q:path>extension/autoassignJobCode</q:path>
              <expression>
                <path>$jobCode</path>
              </expression>
            </q:equal>
          </filter>
        </assignmentTargetSearch>
      </expression>
    </mapping>
  </item>
  ...
</objectTemplate>
```

This is the same principle as we have used earlier in this chapter. The mapping is using `assignmentTargetSearch` expression to look for roles where user's `jobCode` and role's `autoassignJobCode` match. This mapping is strong as we want to recompute the mapping and set the value all the times. If the mapping would be normal-strength, then the values are recomputed only when `jobCode` changes. Which actually might be enough during normal operation of the system. But making this mapping strong makes things much easier during testing. That is all for the mapping. Now we need to prepare the roles for this mapping to work. We need to extend role schema first:

extension-example.xsd

```
<xsd:schema ...>
  ...
  <xsd:complexType name="RoleExtensionType">
    <xsd:annotation>
      <xsd:appinfo>
        <a:extension ref="c:RoleType"/>
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="autoassignJobCode" type="xsd:string"
                   minOccurs="0" maxOccurs="1">
        ...
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  ...
</xsd:schema>
```

Then we need a couple of roles with job codes in their extension:

```
<role oid="a1572de4-b9b9-11e9-af3e-5f68b3207f97">
  <name>Sales Manager</name>
  <extension>
    <exmpl:autoassignJobCode>S006</exmpl:autoassignJobCode>
  </extension>
  ...
</role>
```

```
<role oid="b93af850-b9b9-11e9-8c2c-dfb9a89635a0">
  <name>Sales Agent</name>
  <extension>
    <exmpl:autoassignJobCode>S007</exmpl:autoassignJobCode>
  </extension>
  ...
</role>
```

```
<role oid="b9d2b604-b9b9-11e9-bbc4-17d8e85623b4">
  <name>Sales Assistant</name>
  <extension>
    <exmpl:autoassignJobCode>S008</exmpl:autoassignJobCode>
  </extension>
  ...
</role>
```

That is all the configuration needed for autoassignment to work. The roles should be automatically assigned to users when the users are recomputed. Just make sure that the users have their `jobCode` properly set in the user object. If they do not have it then re-import them or run a reconciliation task. Then go ahead and create some more roles for the missing job codes. No change in any of the mappings is needed to support more job codes. Just create the roles and recompute. That is the beauty of this solution. It is easy to maintain.

So far we have tackled the inbound phase and focus policy phase. But we have not talked about the outbound (provisioning) phase much. Now it is the right time to have a look at that.

We are going to reuse the LDAP and CRM resources from previous chapters. Those resources are used here pretty much unchanged. There is no need to change them. Outbound mappings in the resource definitions specify the basic framework of the account. The key to provisioning flexibility is usually not in the resource definition. The key is in the roles. But let's start in the simplest way possible with the `Employee` role. Example company policy states that every employee should have a very basic LDAP account. Therefore all we need is a very simple LDAP account *construction* that we place into an *inducement* in the `Employee` role:

`role-employee.xml`

```
<role oid="86d3b462-2334-11ea-bbac-13d84ce0a1df">
    <name>Employee</name>
    <inducement>
        <construction>
            <!-- OpenLDAP -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c" />
            <!-- just basic account. Nothing special here. -->
        </construction>
    </inducement>
</role>
```

All employees get this role by the means of inbound mapping on HR resource. Therefore all employees will automatically get basic LDAP account. As simple as that. Put the construction in the role and the reconcile HR resource or just recompute the users. LDAP accounts will be created.

But we want something that is a bit more fancy. Salespeople are a bit sensitive when it comes to their professional image. Therefore, they insist on having proper titles set up in company directory. Not a problem. We can do that easily in their "job" roles. This is how it looks like for a sales manager:

role-sales-manager.xml

```
<role oid="a1572de4-b9b9-11e9-af3e-5f68b3207f97">
    <name>Sales Manager</name>
    ...
    <inducement>
        <construction>
            <!-- OpenLDAP -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c" />
            <attribute>
                <ref>ri:title</ref>
                <outbound>
                    <expression>
                        <value>Sales Manager</value>
                    </expression>
                </outbound>
            </attribute>
        </construction>
    </inducement>
</role>
```

This construction refers to the same account as the Employee role. MidPoint knows that, therefore it does not attempt to create a new account. It just updates existing account with appropriate title. But that is still not enough. We need to provide access to CRM system for the salespeople. Should we create a new role for that? Absolutely not. We do not want to have too many roles as every role is a maintenance burden. Let's just add new construction to an existing job role:

```
<role oid="a1572de4-b9b9-11e9-af3e-5f68b3207f97">
    <name>Sales Manager</name>
    ...
    <inducement>
        <construction>
            <!-- OpenLDAP -->
            ...
            </construction>
    </inducement>
    <inducement>
        <construction>
            <!-- CRM -->
            <resourceRef oid="04afeda6-394b-11e6-8cbe-abf7ff430056" />
            <attribute>
                <ref>ri:accesslevel</ref>
                <outbound>
                    <expression>
                        <value>MANAGER</value>
                    </expression>
                </outbound>
            </attribute>
        </construction>
    </inducement>
</role>
```

This is the only role that gives access to the CRM system for a sales manager. MidPoint knows that and it automatically creates a new CRM account when the role is assigned. Outbound mappings from the CRM resource definition are used to set basic properties of CRM account, such as account identifiers and password. In addition to that, the Sales Manager role sets appropriate access level to the CRM system.

Our setup is almost complete now. We have inbound synchronization, object template, roles and outbound mappings. This is the right time to test everything. Select few representative HR accounts and try to import them. Check that everything is provisioned correctly. If it works, then it is the time for roll-out. Set up a synchronization task for the HR resource and we are done. We have running system:

The screenshot shows a user management interface with the following columns: Name, Given name, Family name, Full name, Email, and Accounts. There are also icons for search, filter, and advanced search at the top right.

	Name	Given name	Family name	Full name	Email	Accounts	
<input type="checkbox"/>	aanderson	Alice	Anderson	Alice Anderson	alice.anderson@example.com	3	
<input type="checkbox"/>	aanderson2	Arnold	Anderson	Arnold Anderson	arnold.anderson2@example.com	2	
<input type="checkbox"/>	administrator	midPoint	Administrator	midPoint Administrator		1	
<input type="checkbox"/>	bbrown	Bob	Brown	Bob Brown	bob.brown@example.com	3	
<input type="checkbox"/>	ccooper	Carol	Cooper	Carol Cooper	carol.cooper@example.com	3	
<input type="checkbox"/>	ddavies	David	Davies	David Davies	david.davies@example.com	2	

This is nice little identity management deployment. But there is still a lot of things to do here. Maybe we want to set up a formalized organizational structure. Maybe we need delegated administration. Maybe we have several object types and we want to set up archetypes for them. We almost certainly want to manage groups, privileges and other entitlements. This is still just a beginning.

Conclusion

This chapter concludes one whole part of the book. If you have followed the book so far, you should be able to set up a simple working IDM deployment at this point. We have covered all the basic mechanisms: resources, mappings, roles, schema and object templates. This is a good time to stop reading and get your hands dirty. Take the examples from this book and play a bit with them. Explore the examples that come with midPoint distribution. Now it is time for experiments. You will surely do a lot of things that are suboptimal or even outright wrong. But never mind. This is part of the learning process. If you get to dead end, just scrap everything and start over. Or maybe rework everything from the ground up. MidPoint is designed for this. Evolutionary approach is deeply embedded in midPoint philosophy and design. Just go ahead, have fun, conduct experiments and explore. Such experience will help a lot when you get back and read through following chapters.

Chapter 9. Organizational Structures

If life is going to exist in a Universe of this size, then the one thing it cannot afford to have is a sense of proportion.

— Douglas Adams, The Restaurant At The End Of The Universe

Organizations come in all shapes and sizes. Unless your organization is extremely unusual, there is always some form of recognizable internal structure. There may be the usual corporate divisions, departments and sections. Or there may be dynamic teams, projects, work groups and task forces. Some kind of *organizational structure* is always there. And it matters. While many organizational structures are quite far from the ideal, organizational structure is seldom completely useless. Membership in some organizational units is a reason to automatically grant privileges. Managers can usually access quite a wide set of data about employees in organizational units that they are managing. Team leaders and project manager often exercise elevated privileges over their team and projects. And all of that is not limited just to users. Roles are often organized into *role catalog*. Services and devices may be organized by applications, by geographical location and so on. There are many things that need to be organized and there are many ways to organize them.

Organizational structure affects almost every part of the identity management deployment. We have realized that in very early stages of midPoint development. Therefore organizational structure is an integral part of midPoint. It permeates almost every part of midPoint functionality. Unlike most other systems, organizational structure in midPoint is a very flexible and almost universal concept. It can be used to build functional organizational structures with divisions, departments and sections. It can also be used to create a flat project-based organizational structure. The same mechanism can be used to sort roles in a role catalog or to manage devices by geographical location. And all of those organizational structures may co-exist at the same time in the same system.

The concept of organizational structure is a very powerful one, but it is implemented by just a handful of simple components. Let us have a look at those building blocks now.

Organizational Units

Basic building block of all organizational structures is just one simple object type. Due to the lack of poetic talent and because of critical shortage of abstract words in our dictionary, we have decided to call that object simply an *org*. It is a nice and short name. *Org* can represent any kind of organizational unit: companies, division, department, section, project, team, role category, geographical location or anything else. Orgs can be used to create hierarchical structures. For example, a top-level org may represent a company. A couple of other orgs can represent divisions. Those orgs can be put "inside" the company org. Yet another orgs can represent departments, these can be put "inside" the division orgs. Repeat the process until complete organizational structure is formed.

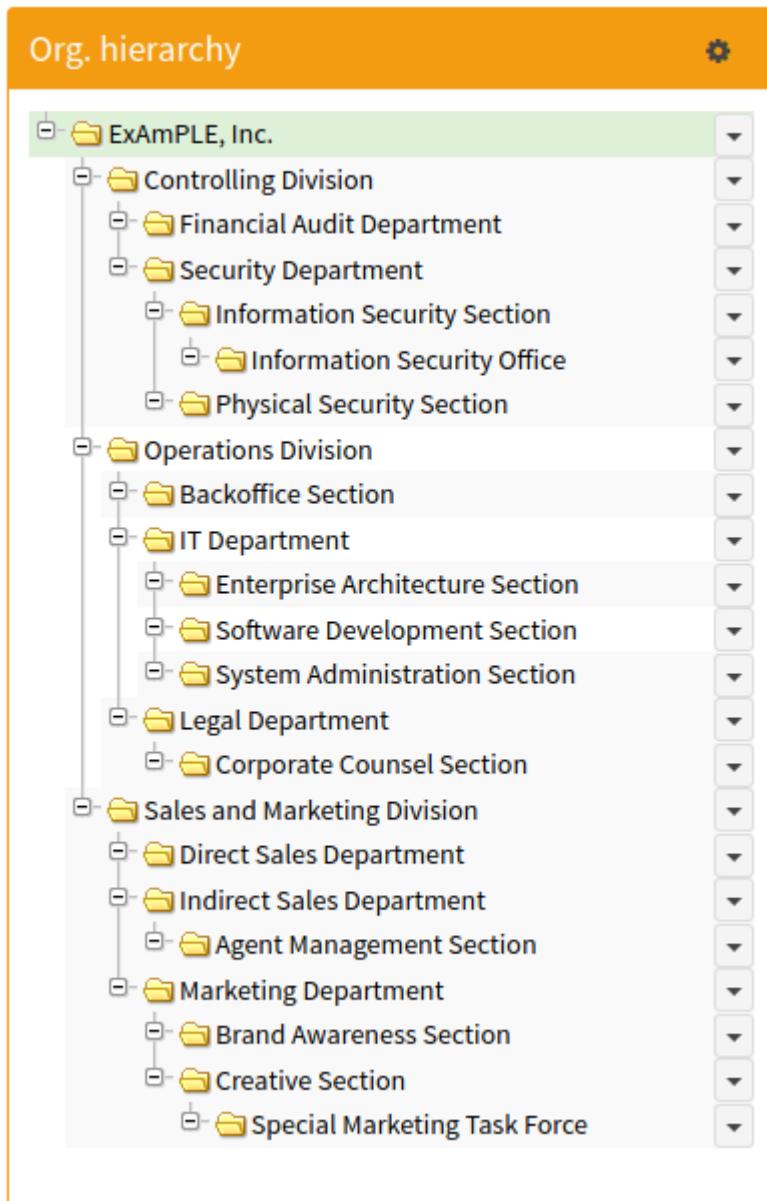


Figure 1. Organizational tree

Org is quite a basic thing with a very simple anatomy:

`org-example-top.xml`

```

<org oid="4d12c1ac-440c-11ea-80af-2b314d06ba95">
    <name>F10000</name>
    <display_name>ExAmPLE, Inc.</display_name>
</org>
```

Strictly speaking, the only things that an org really needs are name and OID. The example above adds `displayName` to make the presentation of the org nicer. As all regular midPoint objects, a name of an org must be unique. This often leads to a practice that org names are in fact identifiers or that they are generated automatically. This is also our case. We have decided to set `F1000` as the name of this org. The value `1000` is an identifier of the company in our HR system. As we are building a functional organizational structure of the company, we have prefixed the identifier with `F` which stands for *functional*. However, names such as `F1000` are not very friendly. Therefore there is a

mechanism to set nicer display name that does not need to be unique. The display name will be used instead of the ordinary **name** whenever this org is displayed to a user.

The screenshot shows the 'Org. hierarchy' and 'Managers' sections. In the 'Org. hierarchy' section, there is a tree view with a single node 'ExAmPLE, Inc.'. In the 'Managers' section, there is a search interface for finding users. The search filters include 'Type: User', 'Relation: Choose one', and 'Scope: One level'. The search results table has columns: 'Name', 'Identifier/Description', and 'Relation'. A message at the bottom right says 'No matching result found.'

Figure 2. Top-level organizational unit

We have an organizational unit now. But how do we put users in it? Clever reader already knows the answer: assignment. All that is needed is to assign the org to a user. This is done in almost the same way as you would assign a role, just select the **Org** tab in assignment dialog.

The screenshot shows the 'Assignments' tab for 'Erin Evans (elevans)'. The 'Organization' tab is selected. The table lists two assignments: 'Employee' and 'ExAmPLE, Inc.', both of which are enabled. The 'Options' section includes checkboxes for 'Force', 'Reconcile', 'Execute after all approvals' (which is checked), and 'Keep displaying results'. Buttons at the bottom include 'Back', 'Preview changes', 'Save', and 'Edit raw'.

Figure 3. Assignment of an organizational unit

Or, in XML form:

```

<user>
  <name>eevans</name>
  ...
  <assignment>
    <targetRef oid="4d12c1ac-440c-11ea-80af-2b314d06ba95" type="OrgType"/>
  </assignment>
  ...
</user>

```

The user is a part of our minimalistic organizational unit now:

The screenshot shows the Organelle application interface. On the left, there is a sidebar titled "Org. hierarchy" with a tree view. A node labeled "ExAmPLE, Inc." is highlighted with a red oval. On the right, there are two main panels: "Managers" (which is empty) and "Members". The "Members" panel has a search bar at the top with filters for "Type: User", "Relation: Choose one", "Indirect: false", and "Scope: One level". Below the search bar is a table with a single row. The table columns are "Name", "Fullscreen/Display name", "Identifier/Description", and "Relation". The row contains a checkbox, the name "eevans", the identifier "erin.evans@example.com", and the relation "default". The entire screenshot is framed by a light gray border.

Figure 4. Assigned top-level organizational unit

Organizational Structure Hierarchy

There is very little structure in our tiny organizational structure yet. Orgs would not be very useful unless they can be placed inside each other, creating a hierarchy. It is this hierarchy that makes organizational structures attractive. Therefore let us go corporate and create some hierarchy now. It is a well-known fact that all self-respecting corporations need sales and marketing division:

org-sales-and-marketing-division.xml

```

<org oid="7a1feb50-471f-11ea-8aab-1b2627541f15">
  <name>F11000</name>
  <description>Expensive people that make money.</description>
  <displayName>Sales and Marketing Division</displayName>
  <identifier>11000</identifier>
</org>

```

We have pimped up this organizational unit a little. We have seen **name** and **displayName** before. The **description** is no stranger either. Then there is an **identifier**. The value of the **identifier** is usually an official "code" of the organizational unit assigned by HR people. But why do we need yet another

identifier? OID is an identifier, name is an identifier of sorts, why do we need another one? For now let's just say that the identifier will be very useful later on, when we will be synchronizing organizational structures.

If we import the org above into midPoint it will become the *top* org. MidPoint will put it at the same level as the ExAmPLE company org. We do not want that. We want to create a hierarchy. We want to tell midPoint to put the department inside the company. How do we do that? We use an assignment, of course:

org-sales-and-marketing-division.xml

```
<org oid="7a1feb50-471f-11ea-8aab-1b2627541f15"
      xmlns='http://midpoint.evolveum.com/xml/ns/public/common/common-3'
      xmlns:org='http://midpoint.evolveum.com/xml/ns/public/common/org-3'>
    <name>F11000</name>
    <description>Expensive people that make money.</description>
    <display_name>Sales and Marketing Division</display_name>
    <identifier>11000</identifier>
    <assignment>
      <targetRef oid="4d12c1ac-440c-11ea-80af-2b314d06ba95" type="OrgType"/>
    </assignment>
</org>
```

Now we have our little hierarchy:

The screenshot shows the midPoint interface with two main tabs: 'Org. hierarchy' and 'Members'. The 'Org. hierarchy' tab on the left displays a tree structure with 'ExAmPLE, Inc.' expanded, showing its children 'Sales and Marketing Division' and 'Marketing Department'. The 'Members' tab on the right is active, showing a list of users assigned to the 'Sales and Marketing Division' unit. The list includes a single entry for 'eevans' (Erin Evans), with details: Fullname/Display name: Erin Evans, Identifier/Description: erin.evans@example.com, Relation: default. There are buttons for adding new members and navigating through the list.

Figure 5. Small organizational structure tree

This makes perfect sense, doesn't it? User become part of organizational units when the units are assigned to them. Therefore also organizational units become part of other organizational units when they are assigned to them. And this applies to everything: roles, services, tasks, resources and other object types. Every object type that can be *assignment holder* can be placed in organizational structure.

Assignment holders

Majority of midPoint object types are *assignment holders* and therefore they can be placed into organizational structure. Theoretically. However, midPoint user interface has some limits. Convenient management of the assignments is currently possible only for *focal* types: user, role, org and service. Other objects can be placed in organizational structure and they should behave up to the expectations. But that cannot be done by few convenient clicks in midPoint user interface Not yet. You have to use a different approach. You either add the assignment manually in the XML/JSON/YAML form. Or you may try to use mappings to create the assignments automatically. Or perhaps use the REST interface to do that. Or maybe send some money in the direction of midPoint development team to motivate them to add this functionality to user interface.



We know how to create a simple organizational hierarchy. All we need to do now is to repeat the process *ad nauseam* to create something that resembles real corporate organizational structure. Let us add marketing department to our division:

org-marketing-department.xml

```
<org oid="a0c7d92c-4722-11ea-bc8d-d79a6cefb1bf"
    xmlns='http://midpoint.evolveum.com/xml/ns/public/common/common-3'
    xmlns:org='http://midpoint.evolveum.com/xml/ns/public/common/org-3'>
    <name>F11300</name>
    <description>Creative bunch that spends money to get more money.</description>
    <displayName>Marketing Department</displayName>
    <identifier>11300</identifier>
    <assignment>
        <targetRef oid="7a1feb50-471f-11ea-8aab-1b2627541f15" type="OrgType"/>
    </assignment>
</org>
```

It is the same process over and over again. But there so many organizational units, there are so many files to import. We like to be efficient in all the things that we do. Therefore let's put the entire organizational structure into a single file:

<objects>

```
<!-- Functional organizational structure of ExAmPLE company -->

<org oid="4d12c1ac-440c-11ea-80af-2b314d06ba95">
    <name>F10000</name>
    <displayName>ExAmPLE, Inc.</displayName>
</org>

<org oid="7a1feb50-471f-11ea-8aab-1b2627541f15">
    <name>F11000</name>
    <description>Expensive people that make money.</description>
    <displayName>Sales and Marketing Division</displayName>
    <identifier>11000</identifier>
    <assignment>
        <targetRef oid="4d12c1ac-440c-11ea-80af-2b314d06ba95" type="OrgType"/>
    </assignment>
</org>

...
</objects>
```

It would be no big surprise to find out that laziness was a driving force behind many improvements in life, would it? Now, let us use this convenient approach to create a nice and rich corporate organizational tree:

The screenshot shows a software interface for managing an organizational structure. On the left, there is a tree view labeled "Org. hierarchy" showing the "ExAmPLE, Inc." organization with various divisions and sections. On the right, there are two tabs: "Managers" (which is currently selected) and "Members". The "Managers" tab displays a table with one row for "eevans" (Erin Evans), with columns for Name, Fullname/Display name, Identifier/Description, and Relation. The "Members" tab has filter and search options at the top.

Name	Fullname/Display name	Identifier/Description	Relation
eevans	Erin Evans	erin.evans@example.com	default

Figure 6. Organizational tree



Of course, you can create and manage organizational structure in midPoint user interface. In fact, people do that quite often. But now we are talking about the *initial* organizational structure. It is the structure that gets created in midPoint at the beginning of the deployment. There is usually a lot of *trial and error* until you get your midPoint configuration right. It is quite likely you will have to purge all midPoint configuration and start clean. In that case, it is very convenient to have organizational structure in one file that can be easily imported after the clean up. Also, it is a common practice to have several environments: development, testing and production. You probably want the same organizational structure in all of them. Having organizational structure in a file makes that job easy. Of course, you can also create organizational structure in the user interface and then export it into a file. But according to our experience, many engineers prefer text editor to graphical user interfaces.

Orgs in the Database

Organizational structures tend to form *hierarchies* - data structures that look like trees. But databases are usually designed to store *relational* data - data structures that look like tables. If you ever tried to express hierarchical data in a spreadsheet application you know that these paradigms are not entirely easy to align. It is not entirely easy to express tree-like data structure in relational tables. Moreover, hierarchical data tend to have some specific requirements. For example, we usually want to look for people in *Operations Division* and all the departments and sections that belong to it. This is known as *subtree searches*, and it is usually not possible to execute them directly on data that are stored in relational form.

This is further complicated by the fact that midPoint *assignment* is a very flexible data structure. Assignments can be valid from a specific time to a specific time. Assignments can be parametric and conditional. Assignment is just too complex for the database to understand and use efficiently.

MidPoint is solving these problems with **parentOrgRef** operational data item. As the name suggests, **parentOrgRef** is an object reference that points to *parent org*. Any *assignment holder* in midPoint can have **parentOrgRef** and it points to the org (or orgs) that the object belongs to. This somehow duplicated the data in the assignment. But there are several crucial differences.

Firstly, **parentOrgRef** points to the orgs that the object is *currently* member of. I.e. it only reflects those assignments that are currently active and valid. Therefore there will be no **parentOrgRef** value for assignment that is expired or not valid yet.

Secondly, **parentOrgRef** represents all organizational assignments, both direct and indirect. Orgs that are directly assigned to users will be present in **parentOrgRef**. Orgs that are induced in a role that is assigned to the user will also be present in **parentOrgRef**. Everything will be there.

Thirdly, **parentOrgRef** is a very simple data structure. This simplicity allows efficient *indexing* of the **parentOrgRef** values in the database (repository) layer. The indexes are designed to allow efficient subtree searches over organizational structure hierarchies.

This is our trick how to fit hierarchical data into flat data tables. The details may be quite complicated, but it usually works quite well. The **parentOrgRef** is automatically maintained by

midPoint under the hood. Therefore it is usually completely transparent. The user does not even notice that there is a special mechanism working in the background.

However, there are also downsides to this approach. The index that is build on `parentOrgRef` is designed to work even if organizational structure is re-organized. The index has to be continually maintained. Maintenance overhead of the index is usually very low for small or mid-sized structures that do not change often. But maintenance of massive organizational structures can be painful. Similarly, it may be problematic to maintain organizational structures that change very frequently. Therefore it is perhaps a good idea to prototype the design of organizational structure before putting the system into production. Also, the `parentOrgRef` is in fact a copy of the primary data (assignment). As it is a copy, there is a risk that it may get out of synchronization. MidPoint is designed to keep `parentOrgRef` and all the indexes strictly consistent during normal operations. However, midPoint allows systems administrators to do a lot of non-standard things. Some of those things may lead to data inconsistencies. Therefore it is a good idea to check whether the values of `parentOrgRef` make sense in case you notice that organizational structures are behaving strangely.

Overall, organizational structures work very well in midPoint and you usually do not need to care about the mechanisms under the hood. However, management of organizational structures is much more complex than it seems. Therefore if you try to do strange and unusual things, you should better be sure you fully understand what you are doing.

Orgs and Roles

Organizations and roles have many things in common. Roles are granting privileges to its members. Usually, people that are members of an organization are granted privileges too. People that have the same role usually have the same set of privileges. People in an organization often have the same privileges too. In fact, organizations behave in almost the same way as roles.

MidPoint has fully embraced this similarity. Orgs are designed to behave in almost the same way as roles. Orgs may have *inducements*, there may be *constructions* in them, orgs may contain *authorizations* and so on. Org can do everything that a role can do.

Therefore there is no need to set up complicated configurations that assign a particular role to all members of an organization. The organization itself acts as a role. All the privileges that organization members need can be simply added as *inducements* in the organization itself. This is very simple, elegant and mostly fool-proof solution.

We have *Indirect Sales Department* in ExAmPLE, Inc. We want to make things simple, and therefore we want to grant access to CRM system to all the members of this department. It is very easy to do:

```

<org oid="8887e0b0-4726-11ea-96b0-5f5ced221e42">
  <name>F11200</name>
  <description>Suits that talk to other suits that talk to customers.</description>
  <displayName>Indirect Sales Department</displayName>
  <identifier>11200</identifier>
  <assignment>
    <!-- Assignment of parent organizational unit -->
    <targetRef oid="7a1feb50-471f-11ea-8aab-1b2627541f15" type="OrgType"/>
  </assignment>
  <inducement>
    <!-- Inducement that grants CRM privileges to all members of this department
-->
    <construction>
      <!-- CRM resource -->
      <resourceRef oid="04afeda6-394b-11e6-8cbe-abf7ff430056"/>
      ...
    </construction>
  </inducement>
</org>

```

Clever reader certainly wonders whether the CRM privileges apply also to *Agent Management Section*, which is located below *Indirect Sales Department* in our organizational structure. However, clever reader is clever enough to figure out that the privileges are not "inherited" in this case. To follow the thoughts of clever reader, you have to think about orgs in the same way as you would think about roles. There is an assignment from *Agent Management Section* to *Indirect Sales Department*. But there is no *inducement*. Role hierarchies are built using *inducements*. Therefore privileges of *Agent Management Section* are not included in *Indirect Sales Department*. This may seem to be counter-intuitive, but in fact it is completely correct. Orgs and roles form separate hierarchies (see note below). However, if you want to "inherit" privileges of a parent org, there is a very simple way how to do it: add explicit *inducement*. For example, this is how we can "inherit" the CRM privileges in *Agent Management Section*:

```

<org oid="f5e619a6-4726-11ea-888c-ab25c098d8b3">
  <name>F11210</name>
  <description>People that deal with agents (no James Bond here).</description>
  <displayName>Agent Management Section</displayName>
  <identifier>11210</identifier>
  <assignment>
    <!-- Assignment of parent organizational unit. This creates organizational
hierarchy. -->
    <targetRef oid="8887e0b0-4726-11ea-96b0-5f5ced221e42" type="OrgType"/>
  </assignment>
  <inducement>
    <!-- Inducement to parent organizational unit. This creates "inheritance" of
privileges. -->
    <targetRef oid="8887e0b0-4726-11ea-96b0-5f5ced221e42" type="OrgType"/>
  </assignment>
</org>

```

This has to be done for every org that needs to inherit privileges from parent, which may be quite daunting for large organizational structures. There is a clever way how to avoid placing inducements everywhere. The solution involves the concept of *metaroles*, as parent org is technically a metarole for child orgs. However, this involves an advanced thinking about application of assignment and inducements. Even a clever reader may not be ready for such abstract thoughts yet. This has to come later when the basic principles have enough time to sink in.

Org and role hierarchies

Both orgs are roles are *hierarchical* in a way. BUt it is not the same hierarchy. Org hierarchy is used to model organizational trees. Role hierarchy is used to build RBAC structures. Those hierarchies have completely different purpose. They are also built using different mechanism. Role hierarchy is used to group privileges and therefore it is built using *inducements*. Org hierarchy is used to group subjects (users) and it is built using *assignments*. There are also different internal mechanisms, indexing and data storage properties. For example, role hierarchy is not using `parentOrgRef`, therefore there is much lower overhead as compared to org hierarchy. However, this means that the capabilities to query role hierarchy is limited. Both hierarchies are their respective place and purpose. Even though the difference may not be very apparent now, it is quite substantial. Hopefully, this will get much more clear later when there will be more examples for both org and role structures.



Managers

Placing people in organizational structures has a significant value on its own. However, all the people usually do not have the same *relation* to the organizational unit. Most people will usually be ordinary *members* of organizational unit. And then there are people that are somehow special: departmental managers, team leaders, project managers, supervisors and similar life forms.

How do we designate a manager of an organizational unit? You probably guessed it already. In a typical midPoint fashion, we are re-using *assignment*, of course. There is just one small detail. We are specifying *relation* in assignment target reference:

```
<user>
  <name>aanderson</name>
  ...
  <assignment>
    <!-- Direct Sales Department -->
    <targetRef oid="832f409a-4726-11ea-b0be-8b8eab99c1ed" type="OrgType" relation
="manager"/>
  </assignment>
  ...
</user>
```

This assignment makes Alice a manager of direct sales department. It is as simple as that. And all the power of assignment is at your disposal. Therefore it is easy to assign a manager for a temporary time period, suspend a manager and so on.

Manager assignment is created in the user interface in almost the same way as normal assignment is created. The only difference is selection of *manager relation* at the bottom of the assignment target dialog:

	Name	Display Name	Description	Identifier
<input type="checkbox"/>	F10000	ExAmPLE, Inc.		
<input type="checkbox"/>	F11000	Sales and Marketing Division	Expensive people that make money.	11000
<input checked="" type="checkbox"/>	F11100	Direct Sales Department	Suits that talk to customers directly.	11100
<input type="checkbox"/>	F11200	Indirect Sales Department	Suits that talk to other suits that talk to customers.	11200
<input type="checkbox"/>	F11210	Agent Management Section	People that deal with agents (no James Bond here).	11210
<input type="checkbox"/>	F11300	Marketing Department	Creative bunch that spends money to get more money.	11300
<input type="checkbox"/>	F11310	Brand Awareness Section	People that get all mad about missing (TM) in our logo.	11310
<input type="checkbox"/>	F11320	Creative Section	We are all mad here.	11320
<input type="checkbox"/>	F11321	Special Marketing Task Force	We have twisted mind and we are not afraid to use it!	11321
<input type="checkbox"/>	F12000	Operations Division	People that make this company work.	12000

1 to 10 of 23 << < 1 2 3 > >>

Figure 7. Assign organizational unit manager

MidPoint now knows that Alice is a manager of Direct Sales Department. This is also displayed in the organizational tree:

The screenshot displays the midPoint web interface. On the left, the 'Org. hierarchy' sidebar shows a tree structure of the organization: ExAmPLE, Inc. has a Controlling Division, Operations Division, Sales and Marketing Division (which contains Direct Sales Department, Indirect Sales Department, and Marketing Department). The Direct Sales Department is currently selected. The main panel is divided into two sections: 'Managers' and 'Members'. In the 'Managers' section, Alice Anderson (aanderson) is listed as a Manager of the Direct Sales Department. She is shown with a red profile icon and her details: Name (Alice Anderson), Identifier (alice.anderson@example.com), and Relation (Manager). In the 'Members' section, a search form allows filtering by Type (User), Relation (Any), Indirect (unchecked), and Scope (One level). A search result table shows one entry: aanderson (Alice Anderson, alice.anderson@example.com, Manager). Below the table are standard CRUD buttons (+, edit, delete, etc.) and navigation controls (1 to 1 of 1, <<, <, 1, >, >>).

Figure 8. Organizational unit with a manager

But, wait a minute! Alice has been promoted to be a *manager* of direct sales department. But she should be located in sales and marketing division as a direct *member* of that division. E.i. she is *member* of one organizational unit and *manager* of another organizational unit. How do we do that? This is all perfectly clear to a clever reader by now. Alice has assignments. First assignment is an ordinary assignment that targets sales and marketing division. Second assignment is a manager assignment that targets direct sales department. This is perfectly normal in midPoint. Users may have any number of assignments to any objects (unless it is explicitly constrained by policy rules or archetypes). Which means that users may be members of any number of organizational units at the same time.

Therefore there is no problem for Alice to be a member of one unit and manager of another. MidPoint can support all kind of bizarre organizational arrangements. MidPoint was deliberately designed in this way. Because reality has an annoying habit to bring surprises, especially when organizational structures are involved.

Well, Alice is a manager now. Good for her. But she still has the same access rights as ordinary workers. That is not right! Managers wear suits and ties. Which means that they need to have more privileges than mere mortals. As managers usually control funding of software development, it is perfectly understandable that midPoint has a way to set up privileges that apply to managers:

```

<org oid="832f409a-4726-11ea-b0be-8b8eab99c1ed">
  <name>F11100</name>
  <display_name>Direct Sales Department</display_name>
  ...
  <inducement>
    <construction>
      ... Privileges exclusive to managers are specified here ...
    </construction>
    <orderConstraint>
      <order>1</order>
      <relation>manager</relation>
    </orderConstraint>
  </inducement>
</org>

```

This is a way how a manager of the direct sales department gets special privileges. The **orderConstraint** makes sure that only the users that have **manager** relation to this organization units will get the privileges.

But wait a minute! Clever reader does not like that. This approach to manager privileges is not going to be very practical. Managers usually do not have special privileges in each organizational unit. In most organizations, managers have the same privileges regardless of the unit they manage.

One way to implement this approach is to create a **Manager** role, put the special privileges there, and assign the role to every manager of every organizational unit. But that creates redundancy. We have to make sure this role is assigned whenever a person becomes manager and that it is unassigned when the person is no longer manager. This is the way how this problem is solved in many IDM deployments. But it is quite a fragile mechanism. And this is not a way how we do things in midPoint.

The **orderConstraint** data structure in our example looks suspiciously complex. And it indeed is quite a complex concept. What we see here is the first glimpse at high-order "assignment algebra" that is a working horse of complex midPoint deployment. It is often employed when working with meta-roles and archetypes. As organizational units are in fact roles and organizational structures are just trees formed by *assignments*, they technically form meta-role structures.

Therefore the right way how to set up manager privileges is to move privilege definition to a central place. It may be top-level organizational unit, or it may be an *archetype*. In that case our *inducement* can apply to all the managers, regardless of organizational unit. However, the exact configuration is a bit complex, and we still need to learn more about midPoint to be able to use it. Therefore we leave the details for later chapters.



MidPoint assigns managers to organizational units. That is the right way how to do it. However, we have often seen a different approach. In such cases the manager is "assigned" to users. I.e. each user has a reference to his or her manager. This approach is wrong. Organizational structures change. People come and go. Everything is changing all the time. It is very easy to change one assignment in organizational structure in case that a manager is replaced. But it is extremely difficult to replace a manager in the direct user-manager data structure. Maybe the former manager was managing several organizational units and now we are replacing him with two managers. Maybe there is a re-organization going on at the same time. The result is going to be a mess. Avoid the direct user-manager approach whenever possible.

Relation

In midPoint, we like to design generic re-usable mechanisms. You did not think that we made the concept of *manager* in a way that would be hardcoded to organizational structure, did you? As you have got so far through this book, you would probably suspect there is more to this *relation* thing that we have seen so far.

The *relation* specifies the nature of a relation between two objects. For example a user may be a member of an organizational unit, manager of a project, owner of a role or approver of role assignment requests. In such cases, *member*, *manager*, *owner* and *approver* are relations that a user can have to an object.

The most common way how to use relation is to specify it in `targetRef` in an assignment. The following example illustrates the usual way how to assign an owner for a role:

```
<user>
    <name>aanderson</name>
    ...
    <assignment>
        <!-- Business Analyst role -->
        <targetRef oid="aaa6cde4-0471-11e9-9b50-c743da469067" type="RoleType"
relation="owner"/>
    </assignment>
    ...
</user>
```

There are several built-in relations in midPoint:

Relation	Usually used for	Description
<code>default</code>	Everything	<p>This is the most common, non-specific relation to an object. When used with a role, it simply means that the user <i>has</i> the role. Usually interpreted as <i>member</i> when used with organizational units. It is the usual, normal relation.</p> <p>As the name suggests, this is the default relation. If no other relation is specified, this relation is used.</p>
<code>manager</code>	Orgs	<p>Manager of an organizational unit, project manager, teamleader, etc. Usually entitles a person (or a group) that have leading position in an org. This usually specifies executive or operational privileges (cf. owner).</p>
<code>owner</code>	Roles, Orgs	<p>Person responsible for governance of the object. Often used to nominate role owners that are responsible for role definition and maintenance. May be used with organizational units to specify project sponsor or business owner. Specifies a person responsible for governance and high-level policy decisions rather than day-to-day management (cf. manager).</p>

Relation	Usually used for	Description
approver	Roles, Orgs	Person responsible for deciding <i>membership</i> in roles and orgs. A gatekeeper or moderator. Approvers usually decide whether someone can have a role, or may be a member of organizational unit. Unlike owners, approvers do not create or modify role definition. They cannot change the role. They can only decide who can have that role and who cannot.
meta	Metaroles	<p>Special-purpose relation that is sometimes used with metaroles. Metarole structures can be complex and confusing. However, such structures and especially policies that govern them may sometimes be simplified, if role-metarole relations are marked in a special way. This relation is designed specifically for that purpose.</p> <p>The <code>meta</code> relation is not mandatory. Metarole functionality will work just fine without it. In fact, almost all of the metarole configuration are not using this relation. But it may come handy if the situation becomes too complicated.</p>

Those are built-in relations. There are some pre-configured policies that work with them. However, you are free to specify and use your own relations. But that is quite an advanced topic and majority of the deployments are perfectly fine using just the built-in relations.

As you can see, the built-in relations do not have overly strict specifications. There is a lot of *usually*, *often* and *almost* in the description of relations. The reason is that the relations do not do anything just by themselves. They just specify how one object relates to another object. There are no strict *policies* or *behavior* associated with them.

The policies are specified elsewhere. Assignment and inducements may behave differently for different relations, as we have seen in previous section. Similarly, *policy rules* are often sensitive to relations. For example the policy that assignment of some roles has to be approved is implemented

by a policy rule that is aware of **approver** relation. Authorizations are often sensitive to relations. Archetypes influence how the system behaves based on relations. User interface may behave differently for some relations. And so on. Relations do nothing just by themselves. However, good part of the system is usually configured to recognize relations and behave accordingly. It is a matter of that *configuration* that determines how exactly will the system behave. This is also the reason for such vague definition of relations, even those built-in relations. They will do what you make them do.

Multiple Organizational Structures

Tree is a simple and very elegant structure in many ways. But it is a rare sight to see a lone tree growing in the field. When we think of trees, we usually think about a forrest. It takes a lot of trees to make a forrest.

This is also the case when it comes to organizational structures. It is a very rare sight when an entire organizational structure of an organization can be modeled in a single tree. There is always the usual *functional* organizational structure with divisions, departments, sections, companies, branches, schools and faculties. But then there is a *project* organizational structure that is often completely orthogonal to functional organizational structure. This is sometimes spiced up with workgroups, task forces, focus groups, research teams, interest groups, clubs and similar collective life forms. There are the many trees that make a forrest.

Fortunately, midPoint is not very picky when it comes to organizational structures. You can have as many organizational trees as you like. MidPoint is not limited to a single organizational tree. You can have functional organizational tree, as we have seen in previous sections. Then you can have independent project organizational structure. Just create new root *org* for projects and place the projects under it:

org-tree-project.xml

```
<org oid="832e37e4-edfd-11ea-9f8c-ef736d6646a2">
    <name>Projects</name>
</org>

<org oid="9c1b8464-edfd-11ea-87b8-db467c5ae301">
    <name>PBD2020</name>
    <description>Make money fast.</description>
    <displayName>Big Deal</displayName>
    <identifier>BD2020</identifier>
    <assignment>
        <targetRef oid="832e37e4-edfd-11ea-9f8c-ef736d6646a2" type="OrgType"/>
    </assignment>
</org>

<org oid="22dc2bd4-edfe-11ea-a904-5be54dda2e46">
    <name>PLS</name>
    <description>Make sure our marketing message gets across.</description>
    <displayName>Loudspeaker</displayName>
    <identifier>LS</identifier>
    <assignment>
        <targetRef oid="832e37e4-edfd-11ea-9f8c-ef736d6646a2" type="OrgType"/>
    </assignment>
</org>

<org oid="1954d496-f6ad-11ea-a96a-8bfa569f5fff">
    <name>PWL2</name>
    <description>Second generation wonderland. We are all mad here.</description>
    <displayName>Wonderland 2.0</displayName>
    <identifier>WL2</identifier>
    <assignment>
        <targetRef oid="832e37e4-edfd-11ea-9f8c-ef736d6646a2" type="OrgType"/>
    </assignment>
</org>
```

We have two organizational trees now. Each neatly stowed under its own tab:

Figure 9. Project organizational tree

This is a nice *project* organizational structure. But our users are members of *functional* organizational structure already. How can I add my users to the projects? The answer is *assignment*, of course. User can belong to any number of organizational units at the same time. It makes no difference whether they are in the same organizational tree or in different trees. Simply assign the projects to the users. The same *manager* relation works for projects as well. In fact, midPoint does not even recognize the difference between *functional* and *project* organizational structures. They look all the same to midPoint and midPoint treats them in the same way. If there is a need for the structures to behave differently, it has to be explicitly configured. Which is usually done by using *archetypes*. We will talk about *archetypes* later.

There are two organizational structures now. You can have three organizational structures if you want to. Of five of them. Any number you like - as long as all the tabs for organizational structures fit on the screen. The structures can be deep and trees with many branches. Or they can be completely flat. The structure may not even be a tree. As long as it is an *acyclic directed graph* it will work just fine. It can have multiple roots, it may have alternate paths, it can do all the crazy stuff. Just avoid cycles. Cycles break the maths which is the foundation of organizational structure indexing and evaluation. Cycles won't work. But pretty much all the other arrangements are perfectly fine.

Organizational structure may have almost any form. A user can be a member of many organizational units. Which also means that a user may *manage* many organizational units. That also applies the other way around: an organizational unit may have many managers. MidPoint fully supports all such cases. By default, MidPoint is very flexible when it deals with organizational structure. However, you may not like all this liberalism in organizational management. *Ordnung muss sien!* If you want to constraint organizational management to allow only a single manager for each organizational unit, you can do it. But you have to explicitly specify a policy by setting up *policy rules*. *Policy rules* provide a very generic and very powerful mechanism how to constraint and control midPoint in many ways. But that is a topic for its own chapter.

Beyond Users

MidPoint organizational structure can do a lot of crazy stuff. Organizational structures are usually build to contain people. Whereas midPoint organizational structure can contain a broad range of

object types. Users, roles and services are the most common object types, but almost any other midPoint object can be placed in organizational structure.

Role catalog is a common use of organizational structure that does not (directly) involve people. Role catalog is used to sort the roles into categories, much like a catalog in electronic shop is used to sort the products. The catalog is used to present roles to users in organized form, so users may easily find the roles when request them in self-service interface.

MidPoint role catalog is simply an organizational structure. It does not have divisions, sections or projects, but it has categories. Categories are (almost) ordinary orgs that form the hierarchy.

The screenshot shows the MidPoint user interface with the 'Role catalog' tab selected. On the left, there is an 'Org. hierarchy' sidebar with a tree view. The root node is 'Role catalog', which has three children: 'Client acquisition' and 'Customer support'. 'Customer support' is highlighted with a green background. At the top right, there are tabs for 'Projects' and 'Role catalog'. Below the sidebar, the main area is divided into two sections: 'Managers' (red header) and 'Members' (blue header). The 'Members' section contains search filters for 'Type' (Role), 'Relation' (Any), 'Indirect' (unchecked), and 'Scope' (One level). A search bar with 'Name' and 'Advanced' buttons is at the top. Below the filters is a table with columns: 'Name', 'Fullname/Display name', 'Identifier/Description', and 'Relation'. Three rows are listed: 'Call center agent' (Member), 'Call center manager' (Member), and 'Tech support specialist' (Member). At the bottom of the table are navigation buttons for adding (+), deleting (-), and sorting (refresh).

Figure 10. Role catalog

Primary use of the catalog is related to request-and-approval process. The catalog makes it easier for a user to find appropriate role when requesting its assignment in self-service part of midPoint user interface. However, the catalog can also be used to apply policies to a whole group of roles. Owner of the category may be considered to be a default approver for all the roles in the category. Category owner may be authorized to modify roles in the category. And so on.

Similar approach can be applied to most objects in midPoint. Organizational structure can be used to organize roles, services, resources, function libraries and other objects. Orgs are also crucial mechanism in supporting midPoint multi-tenancy. Not everything is perfectly supported in user interface yet. Yet, the organizational structure is a powerful mechanism to systematically and consistently apply policies and organize the system.

Organizational Structure Synchronization

MidPoint can manage organizational structure. But where that structure comes from? Back in 20th century there were entire teams dedicated to drawing organizational chart on paper. It is 21st century now, we do not use paper any more. We are using computers to manage organizational

charts now. Which means that dedicated teams are drawing organizational charts in Excel and distributing them by e-mail.

Fortunately, there are some organizations that have truly progressed into 21st century. Such organizations store their organizational structures in a structured form, usually in database tables. When exported to a CSV file, the structure may look like this:

org.csv

```
"orgnum","name","description","parentOrgNum"  
"10000","ExAmPLE","ExAmPLE, Inc.",  
"11000","Sales and Marketing Division","Expensive people that make money.", "10000"  
"11100","Direct Sales Department","Suits that talk to customers directly.", "11000"  
"11200","Indirect Sales Department","Suits that talk to other suits that talk to  
customers.", "11000"  
...
```

In this case each organizational unit has a unique identifier, such as **11000**. Each organizational unit has a reference to parent organizational unit. When all the lines are processed, they form a complete organizational tree.

This is a very good information source. Of course, we would like to automatically pull the data from this source instead of managing organization tree manually. How could we do it? Clever reader is smiling, remembering that we like to create generic re-usable mechanisms in midPoint. There is a way how to synchronize *user* records from the HR system. Of course, the same mechanisms can be reused to synchronize *organizational unit* records.

MidPoint synchronization mechanism can work with almost any object. It can synchronize HR records to users, organizational unit records to orgs, printer database to services, Active directory groups to roles or pretty much anything to anything. This is what we call *generic synchronization*.

Similarly to ordinary synchronization, we need to start with a *resource*. However, this resource will not contain accounts, it will contain organizational units.

resource-csv-org.xml

```
<resource oid="81ec779e-13b2-11eb-8e47-dfbfd542db3e">  
  
    <name>Organizational Chart</name>  
  
    <connectorRef type="ConnectorType">  
        <filter>  
            <q:equal>  
                <q:path>c:connectorType</q:path>  
                <q:value>com.evolveum.polygon.connector.csv.CsvConnector</q:value>  
            </q:equal>  
        </filter>  
    </connectorRef>  
  
    <connectorConfiguration>
```

```

<icfc:configurationProperties
    xmlns:icfccsvfile=
    "http://midpoint.evolveum.com/xml/ns/public/connector/icf-
1/bundle/com.evolveum.polygon.connector-
csv/com.evolveum.polygon.connector.csv.CsvConnector">
    <icfccsvfile:filePath>
        /opt/midpoint/var/resources/org.csv</icfccsvfile:filePath>
        <icfccsvfile:encoding>utf-8</icfccsvfile:encoding>
        <icfccsvfile:fieldDelimiter>,</icfccsvfile:fieldDelimiter>
        <icfccsvfile:multivalueDelimiter>;</icfccsvfile:multivalueDelimiter>
        <icfccsvfile:uniqueAttribute>orgnum</icfccsvfile:uniqueAttribute>
    </icfc:configurationProperties>
</connectorConfiguration>

<schemaHandling>

    <objectType>
        <displayName>Organizational unit</displayName>
        <default>true</default>
        <objectClass>ri:AccountObjectClass</objectClass>
        <kind>generic</kind>
        <intent>orgunit</intent>
        <attribute>
            <ref>ri:orgnum</ref>
            <inbound>
                <target>
                    <path>$focus/identifier</path>
                </target>
            </inbound>
        </attribute>
        <attribute>
            <ref>ri:name</ref>
            <inbound>
                <target>
                    <path>$focus/displayName</path>
                </target>
            </inbound>
        </attribute>
        <attribute>
            <ref>ri:description</ref>
            <inbound>
                <target>
                    <path>$focus/description</path>
                </target>
            </inbound>
        </attribute>
    </objectType>

</schemaHandling>

<projection>

```

```

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

<synchronization>
  <objectSynchronization>
    <enabled>true</enabled>
    <objectClass>AccountObjectClass</objectClass>
    <kind>generic</kind>
    <intent>orgunit</intent>
    <focusType>OrgType</focusType>
    <correlation>
      <q:equal>
        <q:path>identifier</q:path>
        <expression>
          <path>$projection/attributes/orgnum</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#deleteFocus</handlerUri>
      </action>
    </reaction>
    <reaction>
      <situation>unlinked</situation>
      <synchronize>true</synchronize>
      <action>

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

<handlerUri>http://midpoint.evolveum.com/xml/ns/public/model/action-
3#addFocus</handlerUri>
      </action>
    </reaction>
    </objectSynchronization>
  </synchronization>

```

```
</resource>
```

This should all look very familiar by now. It is almost the same resource as we have seen in the synchronization chapter. However, there are few differences. We will describe them in next sections.

Kind and Intent

The definition of "Organizational unit" resource object contains specification of *kind* and *intent*:

resource-csv-org.xml

```
...
<objectType>
  <displayName>Organizational unit</displayName>
  <default>true</default>
  <objectClass>ri:AccountObjectClass</objectClass>
  <kind>generic</kind>
  <intent>orgunit</intent>
...
...
```

Kind and intent identify the type of resource object for use by midPoint. There are three possible values for *kind*:

Kind	Description
account	Resource object that represents identity of a person, either physical such as computer user or virtual such as <code>administrator</code> , <code>root</code> , <code>daemon</code> or similar special-purpose identities. Accounts are usually linked to the User objects.
entitlement	Resource object that represents groupings or privileges of an account. Entitlement resource objects represent groups, resource-specific roles or access permissions. Entitlements are meant to be associated to an account. For example a <code>group</code> entitlement may have accounts as its members.
generic	Any other type of resource object. This is used for resource objects that cannot be classified as <i>account</i> or <i>entitlement</i> .

You can choose any of these accounts for your resource objects. Values of *kind* are pre-defined in midPoint, as midPoint will make some assumptions about them. For example, midPoint will expect that accounts can be associated with entitlements, for example accounts may be members of groups. Therefore, it is recommended to properly categorize your resource objects to kinds. This also helps to make the configuration understandable for mere mortals.

Then there is *intent*. There are no pre-defined values for *intent* in midPoint, perhaps except for

value `default` that will be used in case there is no explicit definition of *intent*. You can choose any value that you like. However, it is recommended to choose a value that describes the intended use of the resource object. For example:

Kind	Example intent	Description
account	default	Default account. This usually means the usual, very ordinary user account.
account	admin	Administration account. Used in situations where administrators get dedicated accounts with administrator privileges.
account	test	Testing account. Used when testers are given special-purpose accounts to use for testing, to avoid interference with their usual accounts.
entitlement	group	The usual, most ordinary, boring group of users. It can have accounts or other groups as its members.
entitlement	posixGroup	LDAP <code>posixGroup</code> , used to assign UNIX group membership in LDAP. It has a different structure than ordinary group, hence we would like to use a different <i>kind</i> for it.
entitlement	privilege	Resource object that represents system privilege. Can be "given" to an account.
generic	locality	Resource object that represents physical location in our organization. Such as branch office, campus building or meeting room. It has no formal association to the account.
generic	orgunit	Resource object that represents organizational unit. In our case, it represents one record in the organizational chart database.

These are just examples. You can choose any kind/intent combination that makes sense for your deployment. For example, you may choose to use `entitlement` kind for organizational units instead of `generic`. MidPoint would work fine even in that case. However, our `Organizational Chart` resource does not have any accounts, therefore the organizational units cannot really be associated

to anything in this resource. Also, membership in an organizational unit does not really look like an entitlement. Therefore we have chosen to use **generic** kind here. You are free to make your own choices.

Why **AccountObjectClass**?

Why is there **AccountObjectClass** in the configuration? We are not working with accounts here, we work with organizational units. So, why **AccountObjectClass**? The reason is the CSV connector that we are using. The CSV connector is quite simplistic, it considers everything to be an account. Object classes are set by the connector, hence we need to use **AccountObjectClass** here. However, the *kind* and *intent* definition is "overriding" the notion that this is an account, making it more understandable for the users.

Kind and *intent* behave like coordinates when midPoint has to identify resource object that is assigned to a user. A user may have at most one *default account* on a resource (which means *kind=account*, *intent=default*). That same user may also have *admin account* on the same resource (*kind=account*, *intent=admin*). A role may be represented by at most one *LDAP group* on a resource (*kind=entitlement*, *intent=ldapGroup*). This is heavily used by midPoint logic. Whenever there are two *constructions* that have the same combination of *kind+intent*, midPoint assumes that they are describing the same resource object. MidPoint automatically merges the constructions. If the constructions have different combination of *kind+intent*, midPoint assumes that they are describing different resource objects and the constructions are processed separately. Correct configuration of *kind* and *intent* is crucial for midPoint to work correctly.

Tag (a.k.a. "multiaccounts")

The requirement that there may be at most one resource object for each *kind+intent* combination works very well in most cases. However, there are also cases when more than one resource object is needed. *Kind* and *intent* has to be specified in the configuration, therefore this mechanism will not work for resource objects that appear and disappear dynamically. Therefore, new concept of *tag* was introduced in midPoint 4.0. The *tag* can supplement the *kind+intent* combination with a dynamic value, thus allowing multiple resource objects to exist for any particular *kind+intent* combination. This feature is colloquially known as "multiaccounts".

Names and Identifiers

Synchronization of organizational structure is the same as synchronization of users. Theoretically. However, there are some subtle differences in practice, mostly caused by the differences of **User** and **Org** schemas.

The first difference originates from the fact, that **name** of the org has to be unique. This uniqueness is usually not a problem for users, as username is naturally unique among the entire user base. However, this is slightly different for orgs, as there may be several parallel organizational structures. There may be **Security** department, **Security** project and **Security** workgroup at the same time. This may be partially solved by using identifiers instead of names. However, this still does not solve the problem of department **123** and project **123**. The simple solution is to prefix the

identifier with a code of the organizational tree that it belongs to, thus creating department 0123 and project P123. However, users looking for project 123 may have difficulty finding it, as the P prefix in P123 name is usually just a deliberate decision of IDM administrator. Therefore, we still want to store the original identifier value (123) into the **identifier** property of the org object. There is no uniqueness constrain on the **identifier** property, therefore both department 123 and project 123 can co-exist and both can be easily discovered by searching the identifier. The result is that we need two inbound mappings for the `` attribute:

resource-csv-org.xml

```
<attribute>
    <ref>ri:orgnum</ref>
    <inbound>
        <target>
            <path>$focus/identifier</path>
        </target>
    </inbound>
    <inbound>
        <expression>
            <script>
                <code>'0'+input</code>
            </script>
        </expression>
        <target>
            <path>$focus/name</path>
        </target>
    </inbound>
</attribute>
```

Storing original organizational unit identifier in the **identifier** property makes it easier to correlate organizational units. The **identifier** property can be used in the correlation query. If the identifier is reasonably persistent, this is a huge benefit.

Changes in organizational structure can be quite nasty. Organizational units are often renamed or moved in organizational trees. Simplistic synchronization configuration may not be able to interpret such events correctly. It may look like new organizational unit was created, and the old unit was deleted. This is likely to wreak havoc to organizational unit assignments, especially if there were special privileges configured for this organizational unit. Even worse, organizational tree data are sometimes acquired from a different source than the user data, which is causing timing problems. If organizational tree is updated first, there will be new empty organizational unit, old unit will be deleted, and user assignments will become invalid. If user data are updated first, the synchronization routines may not be able to update the assignments as the new organizational unit does not exist yet. This is going to cause a whole lot of problems, most of them will need to be fixed by manual intervention of IDM administrator. Additionally, reorganizations usually happen in cycles, each cycle changing a number of units at the same time. Which means that every few months the organizational structure is going to break down, everybody will be complaining, and it will take days to fix all the problems manually.

All of that can be avoided if there are reasonably persistent organizational unit identifiers. Which

means that every organizational unit has an identifier that does not change when the unit is renamed or moved. In that case midPoint can reliably detect the rename, change organizational unit name and keep the assignments intact. MidPoint can also detect that the unit was moves, change the parent unit and still keep all the assignments. Organizational unit identifiers make everything so much easier. Therefore, try really hard to use the identifier when setting up synchronization of organizational structure. If there is no such identifier, talk to the business people to add it. This is usually not an easy discussion, as the solution often involves changes in business processes. However, it is absolutely essential to get it right. All the effort will be repaid many times over during the course of IAM program.

Nesting Organizational Units

We can synchronize the organizational units into midPoint. We can set up the properties of organizational units. However, organizational structures are usually hierarchical. How to do we nest organizational units to create organizational tree?

In midPoint, organizational tree is formed by *assignments*. Therefore the answer is quite simple: create the right assignments. Clever reader is not paying attention any more, re-reading the sections on automatic role assignments in object template. However, even a clever reader should pay attention here, as there is an easier way how to do it. The assignments can be set up in the inbound mappings:

resource-csv-org.xml

```
<attribute>
    <ref>ri:parentOrgNum</ref>
    <inbound>
        <expression>
            <assignmentTargetSearch>
                <targetType>OrgType</targetType>
                <filter>
                    <q:equal>
                        <q:path>identifier</q:path>
                        <expression>
                            <path>$input</path>
                        </expression>
                    </q:equal>
                </filter>
            </assignmentTargetSearch>
        </expression>
        <target>
            <path>$focus/assignment</path>
        </target>
    </inbound>
</attribute>
```

This mapping automatically sets up an assignment to parent organizational unit. We are lucky, we have almost ideal source of organizational data. Our CSV file contains an identifier of a parent organizational unit in the **parentOrgNum** column. All we need to do is to look for midPoint org that

has that particular value in its `identifier` property. This is done by the `assignmentTargetSearch` that we have already used for automatic assignment of roles in object template. The same mechanism is reused here.

All that remains is to set up a synchronization task. Make sure that you specify `kind` and `intent` in the synchronization task. This is important, otherwise the tasks will not work. Setting the right `kind` and `intent` was not emphasize before when we were synchronizing accounts. The `account` `kind` is the default and midPoint is usually smart enough to use default `intent`. However, the defaults will no longer work when we go beyond the accounts.

Troubleshooting



Generic synchronization can be confusing and there may be non-obvious configuration complexities. When mis-configured, the synchronization mechanism often does nothing. There is no error or any other obvious indication of an error. Logging is your best friend in that case. Enable logging of *synchronization service* (`com.evolveum.midpoint.model.impl.sync`) at debug level. MidPoint will log a reasonable amount of information about the synchronization process. That information is very likely to lead you to the solution.

The configuration above will work for simple cases, but there is still a room for improvement. The search filter in `assignmentTargetSearch` expression is quite simplistic. It will match orgs from several trees if they have the same identifier. The same problem is in the correlation query. However, clever reader would surely find a way how to improve it.

Also, this method will work only if the data feed is correctly ordered. Everything will work as long as parent organizational units are synchronized before child organizational units. However, that is not always the case. If ordering is wrong, child organizational units will not be able to find parent units, and the tree will disintegrate. We are living in a networked concurrent world, data ordering is usually difficult to guarantee.

MidPoint has a mechanism to handle unordered data source. There is a way how to create parent organizational units *on demand*. When a child organizational unit looks for a parent that is not there yet, the parent object can be created at that moment. Of course, this can only create *stub* parent, a very minimal object that has only the essential data. Yet, even such a *stub* object is sufficient to create an organizational hierarchy. The *stub* will be updated later, when the details about parent organizational unit are retrieved from the data feed. The details of this *create on demand* mechanism is beyond the scope of this chapter. We will get back to it later.

Adding Users To Units

We have a nice hierarchical organizational structure now. Yet, something is still missing. The organizational structure is all about the people, but there are no people in our organizational tree. Let's fix this.

The people data are coming from HR resource in our ExAmPLE case. We need to modify the HR resource to automatically assign people to the organizational tree. To do that, we need to add information about organizational units into our HR feed. After several phone calls, tens of e-mails and a quick 3-hour meeting, the HR department agreed to add a new `orgnum` column to the CSV file:

```
"empno","firstname","lastname","jobcode","orgnum"  
"001","Alice","Anderson","S006","11100"  
"002","Bob","Brown","S007","11210"  
"003","Carol","Cooper","S008","11310"  
...
```

The 'orgnum' column contains an identifier of the organizational unit the person belongs to. This looks quite familiar, and clever reader is working on the configuration already. Of course, we can use the same approach we have used to build up organizational hierarchy. We just need to apply it to the users instead of organizational units. Therefore we are going to add new inbound mapping to the HR resource:

```
<attribute>  
    <ref>ri:orgnum</ref>  
    <inbound>  
        <expression>  
            <assignmentTargetSearch>  
                <targetType>OrgType</targetType>  
                <filter>  
                    <q:equal>  
                        <q:path>identifier</q:path>  
                        <expression>  
                            <path>$input</path>  
                        </expression>  
                    </q:equal>  
                </filter>  
            </assignmentTargetSearch>  
        </expression>  
        <target>  
            <path>$focus/assignment</path>  
        </target>  
    </inbound>  
</attribute>
```

The **assignmentTargetSearch** expression looks for the right organizational unit. Then the mapping creates an assignment to that unit. And we are done. Run the HR synchronization task, and all the users are going to be neatly organized in the tree.

Get your data structures right at the beginning.

That 3-hour meeting with HR was in fact really useful and necessary. The result was that ExAmPLE HR department did the right thing. They put *identifier* of organizational unit in the HR feed, instead of organizational unit *name*. Having organizational unit *identifier* makes everything much more stable.



There is a lesson to be learned. Every hour spent designing the data structures will be repaid many times over. Getting it wrong will cost you days or months dealing with data inconsistencies. It is also very difficult to change data formats in the future, as they effectively become data integration interfaces. Take your time and get it right at the beginning.

Provisioning Organizational Structure

We have seen how we can synchronize organizational structure into midPoint. We are talking midPoint here. What goes in, can also go out. It is very simple to provision organizational structure to an ordinary target system, such as database table. However, we have already learned a thing or two, and doing that would be almost boring. Therefore let's do something more challenging. Let us synchronize the organizational structure into an LDAP directory tree.

The basic principles of organizational structure provisioning are the same as for users. We need to set up *outbound mappings* for organizational units. We already know how to do that. There are just few little differences:

- We will use `organizationalUnit` object class instead of `inetOrgPerson`.
- We will use special kind/intent combination.
- We have to be a bit smarter about creating LDAP distinguished names (DNs) for the entries, as we want them to create a hierarchical data structure.

Everything else is essentially the same as for users and accounts. However, let us go over all the details step by step.

First of all, we need to add new `objectType` definition to the LDAP resource:

`resource-openldap.xml`

```
<objectType>
    <kind>generic</kind>
    <intent>ou</intent>
    <display_name>Organizational Unit</display_name>
    <objectClass>ri:organizationalUnit</objectClass>
    <attribute>
        <ref>ri:dn</ref>
        <display_name>Distinguished Name</display_name>
        <limitations>
            <minOccurs>0</minOccurs>
            <maxOccurs>1</maxOccurs>
        </limitations>
```

```

<outbound>
    <name>ldap-ou-dn</name>
    <trace>true</trace>
    <source>
        <path>$focus/name</path>
    </source>
    <expression>
        <script>
            <code>
                import javax.naming.ldap.Rdn
                import javax.naming.ldap.LdapName

                // We will collect names of the org units in the
                orgpath list
                // We cannot add them to dn yet as we need their order
                to be reversed
                def orgpath = []
                def node = focus
                while (true) {
                    log.debug("processing node {}", node)
                    orgpath.add(node.displayName.orig)
                    if (node.parentOrgRef == null ||
node.parentOrgRef.isEmpty()) {
                        break
                    } else {
                        node =
midpoint.resolveReference(node.parentOrgRef[0])
                    }
                }

                log.debug("orgpath={}", orgpath)
                def dn = new LdapName('ou=org,dc=example,dc=com')
                orgpath.reverse().each { ouname -> dn.add(new
Rdn('ou',ouname)) }
                return dn.toString()
            </code>
        </script>
    </expression>
</outbound>
</attribute>
<attribute>
    <ref>ri:ou</ref>
    <limitations>
        <maxOccurs>1</maxOccurs>
    </limitations>
    <outbound>
        <source>
            <path>$focus/displayName</path>
        </source>
    </outbound>
</attribute>

```

Except for that big piece of Groovy code, this configuration is relatively simple. The `objectType` definition specifies `organizationalUnit` value for object class. This is a standard LDAP object class for "ou" entries. There is also specification of kind (`generic`) and intent (`ou`). These are midPoint "coordinates" for this object type. Then we have two *outbound* mappings, one for LDAP distinguished name (`dn`), the other for naming attribute (`ou`). The `ou` mapping is very simple, using a value of org's `displayName`. On the other hand, the `dn` mapping looks somehow scary. There is no need to be afraid. We are going to explain everything, and there are some really interesting parts here.

The purpose of this mapping is to construct LDAP distinguished name in a hierarchical manner. We want to put the organizational tree under the `ou=org,dc=example,dc=com` entry, with the entry for ExAmPLE company at the top. Therefore the `dn` of Operations Division need to be `ou=Operations Division,ou=ExAmPLE,ou=org,dc=example,dc=com`. IT Department goes under the Operations Division, therefore we need its `dn` to be `ou=IT Department,ou=Operations Division,ou=ExAmPLE,ou=org,dc=example,dc=com`. We need to process the tree from the organizational unit all the way through all the parent units to the very top of organizational structure. That is exactly the thing that the Groovy expression does.

Let's skip the `import` statements for now. The first thing that the expression has to do is to figure out the "path" from the current organizational unit to the top of the tree. The expression gets the current organizational unit in the `focus` variable. However, the `org` object does not contain its complete path in the tree. All it has is a reference to its parent organizational unit (`parentOrgRef`). Therefore the expression has to iterate over all the levels in the tree until it gets to the top. The top organizational unit does not have any parent, that is where the iteration stops. Display names of each organizational unit at the "path" is collected in the `orgpath` list. As the `org` contains only a reference to the parent, we need to explicitly read the parent object from midPoint repository. We will do that with an explicit call to `midpoint.resolveReference(...)` method. This method reads the object from the database and returns it. When the loop stops, `orgpath` contains all the display names that we want in our `dn`. Now we need to encode the names in LDAP DN format. This can be done by simple string operations. However, there are some intricate details about escaping the names as they are encoded. It would be nice if someone else could do the encoding for us. Turns out, there is someone else to do it. Java platform comes with Java Naming and Directory Interface (JNDI), which is supposed to be a generic library to access broad range of directory services. JNDI is not the best library that the world has ever seen, but it is part of Java platform and it can do formatting of LDAP DN. It will be good enough for us. We will take advantage of `LdapName` and `Rdn` classes to encode the DN. The `import` statements at the beginning made use of these classes quite convenient. The last detail is the ordering. We want our names in the DN to be in a different order, therefore we just reverse `orgpath` before processing.

Clever reader has noticed a couple of interesting things in that mapping. Especially the `<trace>` element looks very useful, which it is. It turns on detailed tracing of the mapping. MidPoint will record the details of mapping evaluation in the log file. Similar `<trace>` element can also be applied at the expression level. Then there are the `log` statements in the Groovy code, such as `log.debug("processing node {}", node)`. These are explicit logging statements. The `processing node ...` message is recorded to the log file at debug level. This is a very useful tool for diagnosing execution of complex expressions.

Our outbound mappings are ready to go. But nothing happens yet. MidPoint does not know that it is supposed to create LDAP objects for our organizational units. MidPoint does not automatically create accounts for all the users either. We need a *construction* to do that. The orgs need to have an assignment with an construction, similar to these that we have used to create accounts. However, we will need to use the right *kind* and *intent*. We want to archive something like this:

```
<org oid="7a1feb50-471f-11ea-8aab-1b2627541f15">
    <name>F11000</name>
    <display_name>Sales and Marketing Division</display_name>
    ...
    <assignment>
        <construction>
            <!-- LDAP resource -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
            <kind>generic</kind>
            <intent>ou</intent>
        </construction>
    </assignment>
</org>
```

You can test your configuration by creating this assignment manually in the GUI, or by editing the XML/JSON/YAML version of the org. However, we need to create these assignments automatically. There are several ways to do it.

Perhaps the most obvious way would be to add a mapping to create this assignment in the object template. As there is an object template for users ([UserType](#)), there may be an object template for any other midPoint object. Therefore, you can create object template for orgs ([OrgType](#)) and configure the mapping there. This would be an acceptable solution.

An alternative way would be to add *inbound* mapping to create this assignment. That would be in fact quite easy, as it would is very similar to the mapping that we have used to create organizational hierarchy. However, that would not be an ideal configuration, as would would mix the *concerns* here. The mapping will be an *inbound* mapping in the [Organizational Chart](#) resource. It would not be entirely appropriate for this *inbound* mapping to control provisioning (i.e. *outbound* flow) of organizational structure. It will work, but such configuration will be difficult to understand and maintain.

Clever reader is not thinking about the *metaroles* that we have already mentioned in the RBAC chapter. As usual, cleaver reader is right, or perhaps very close to the right solution. Creating a *metarole* that could be applied to all the orgs would be almost an ideal solution:

role-meta-orgunit.xml

```
<role oid="b8c3ccba-1dfb-11eb-9031-27e1d26ca36e">
    <name>Functional Organizational Unit Metarole</name>
    <inducement>
        <construction>
            <!-- LDAP resource -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
            <kind>generic</kind>
            <intent>ou</intent>
        </construction>
    </inducement>
</role>
```

This metarole can be assigned to every org that we want to provision. In such case, it is perfectly good to have the mapping to assign the meta role even in the *inbound* part of the [Organizational Chart](#) resource. The metarole specifies type of the organizational unit. Therefore it is desirable to set the metarole when the org is created for the first time, which happens during organizational structure synchronization.

This would be almost ideal solution. However, there is a better way still. We can use *archetype* instead of metarole. Archetype will do the same trick as metarole does here.

archetype-orgunit.xml

```
<archetype oid="475106e4-1dfe-11eb-8429-534869969212">
    <name>Functional Organizational Unit</name>
    <inducement>
        <construction>
            <!-- LDAP resource -->
            <resourceRef oid="8a83b1a4-be18-11e6-ae84-7301fdab1d7c"/>
            <kind>generic</kind>
            <intent>ou</intent>
        </construction>
    </inducement>
</archetype>
```

Archetype can do even more tricks than metarole. Archetype can specify an icon and a color for the objects. It can control the assignments to these objects. Archetypes give a structure to midPoint objects. We will get back to archetypes in a dedicated chapter.

For now, just choose any of the methods above to trigger provisioning of organizational structure. We will go for the archetype, assigning it automatically in inbound mappings of [Organizational Chart](#) resource:

```
...
<objectType>
  ...
    <objectClass>ri:AccountObjectClass</objectClass>
    <kind>generic</kind>
    <intent>orgunit</intent>
    <attribute>
      <ref>ri:orgnum</ref>
      ...
      <inbound>
        <expression>
          <value>
            <targetRef oid="475106e4-1dfe-11eb-8429-534869969212"
type="ArchetypeType"/>
          </value>
        </expression>
        <target>
          <path>$focus/assignment</path>
        </target>
      </inbound>
    </attribute>
    ...
  </objectType>
  ...

```

All we need to do now is to re-run the synchronization task. The result is a nice organizational structure in LDAP directory:



Figure 11. Organizational structure in LDAP

This is a nice result. However, there are still several remarks to make.

If you are going to try this scenario with a real LDAP server, you will need to create a root entry for the organizational structure (`ou=org,dc=example,dc=com`). You will also need to update access control lists (ACLs). However, be warned that this is not a very ideal way how to maintain organizational structure in LDAP directory. We are using display names here, which are part of LDAP identifiers (DNs). Therefore even a minor correction in organizational unit name will trigger LDAP rename operation. It will change the identifier of the organizational unit, and also all the organizational units below it. Perhaps the only thing that can make it worse is to place *users* in that structure as well. Which some people actually do. The only real reason to put organizational structure in this form is to satisfy the needs of legacy applications. Avoid this approach whenever you can.

The clever reader have surely noticed that the big Groovy expression above is explicitly fetching objects from the database. This may cause a performance problem in case that such expression is evaluated often or if the strucutre is very deep. This is usually not the case with ordinary organizational structures, therefore this approach is usually not problematic. However, it is always a good thing to keep performance in mind, especially if your user population is bigger than few thousands of users.

Finally, we still depend on ordering of the data feed. We want to create "higher" entries in LDAP first, otherwise an attempt to create "lower" entries would fail. This can be solved by using the "create on demand" approach mentioned above. However, it is going to be problematic and cumbersome when organizational unit display names are used instead of indetifiers. It does not matter that much anyway, as it is likely you will suffer for many different reasons if organizational display names are used. Always use organizational unit identifiers if you can.

Focus and Projection

Synchronization of an organizational structure is an application of a *generic synchronization* principle. Almost any resource object can be synchronized with almost any midPoint object - and vice versa. Principle of the synchronization is essentially the same as in user-account case that we have seen in previous chapters. In that case the accounts were linked to user, midPoint synchronized the data from account to user and from user to accounts. The synchronization follows user-account links.

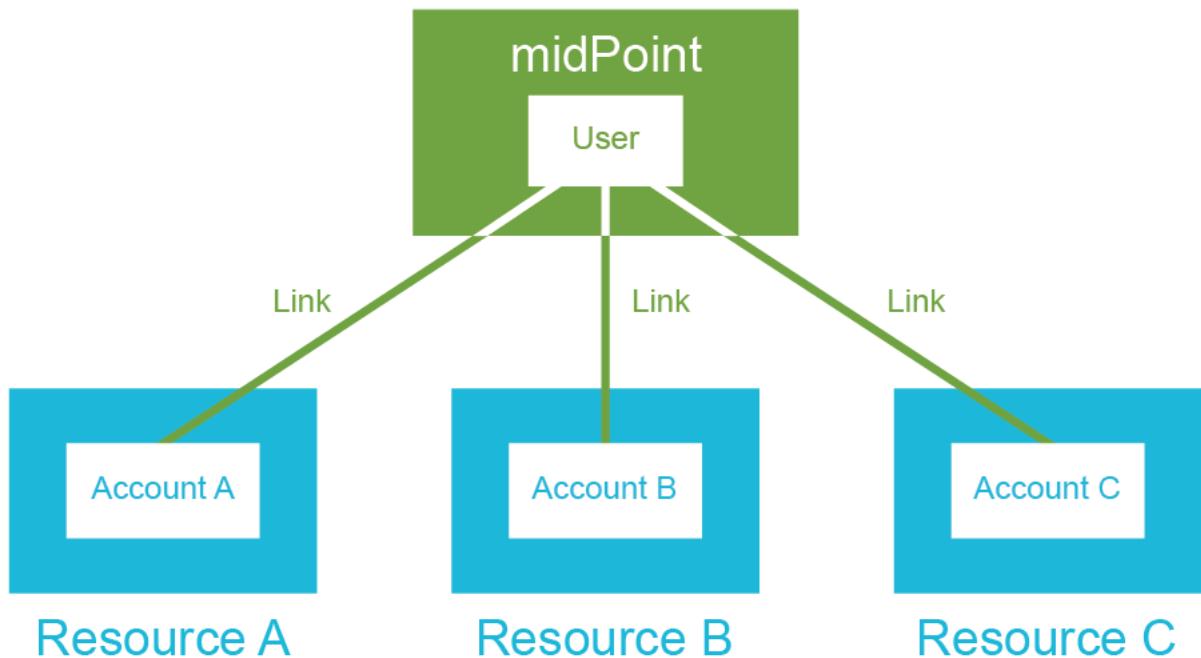


Figure 12. User and accounts

Synchronization of an organizational structure is based on the same principle. However, there is an org instead of user, and there are various resource objects instead of accounts. It may look like this:

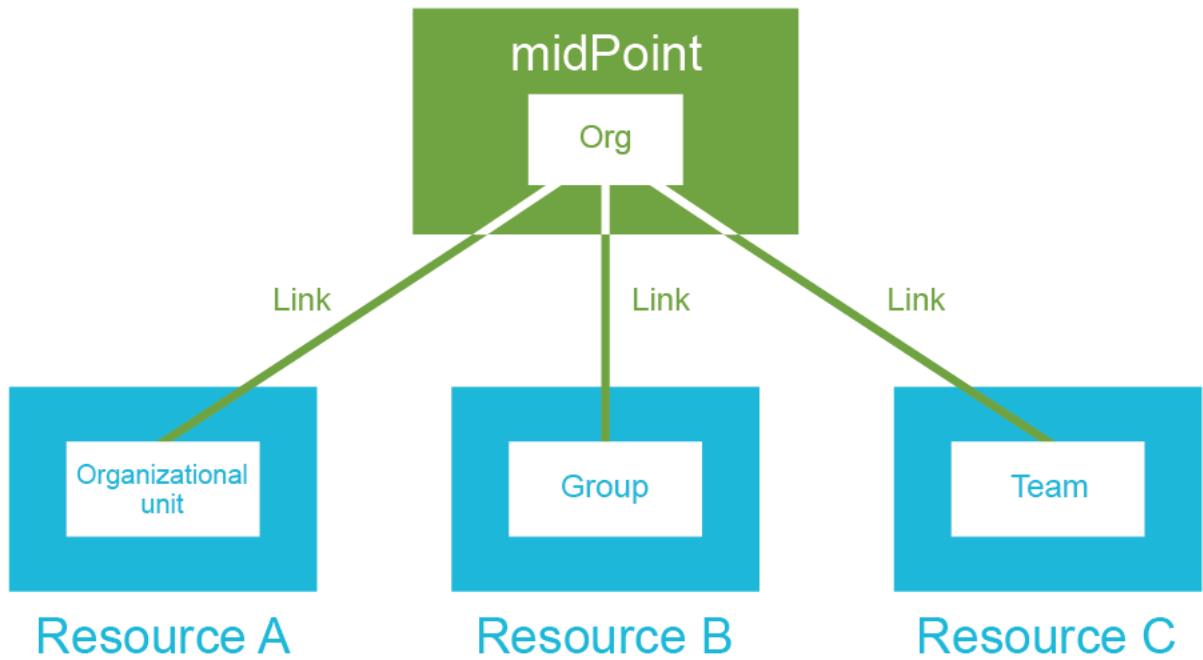


Figure 13. Org and projections

There may be user, org, service, role or almost any midPoint object on the midPoint side. That may be synchronized with account, group, role, privilege, organizational unit or almost any resource object on resource side. As you can see, the terminology becomes quite cumbersome. Saying "almost any midPoint object" and "almost any resource object" is not very natural or precise. Therefore we have decided to use *focus* and *projection* terms:

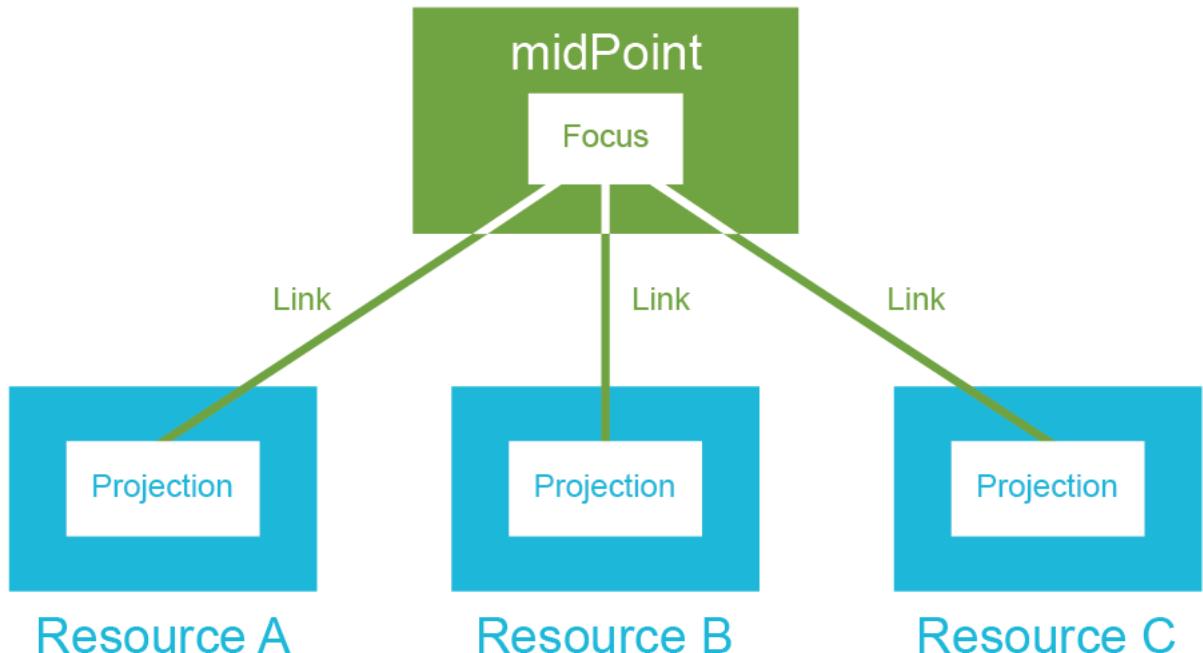


Figure 14. Focus and projections

The object that is "in the middle" is called *focus* or *focal object*. It is in the centre of the synchronization, it is its focal point. Every relevant piece of data is reflected onto the focal object by the synchronization mechanisms.

User is a typical focal object. Other midPoint objects can be focal objects as well, most notably org, role and service.

The state of focal object is *projected* back to the resources. Therefore the objects that reside on the resources are called *projections*. An account is a typical projection object, but there is wide variety of other object classes such as groups, organizational units, (resource-side) roles, privileges, access control lists, teams and so on.

Focus and projection



Focus and projection may sound like strange words to use for identity management concepts. Many identity management systems work with just *user* and *account*. However, midPoint is more flexible, more generic. When we designed the generic synchronization mechanism, we needed to find good names for the generalization of *user* and *account* concepts. We tried hard, but we could not find anything better than *focus* and *projection*. There are only two hard problems in computer science, after all.

There is always one *focus*, one focal object in the center. There may be any number of *projections* linked to the *focus*. There may be several *projections* on one resource. However, each *projection* has to be unique, it needs to have a unique combination of *kind* and *intent*. This is the reason that we consider *kind* and *intent* to be coordinates, as they uniquely identify a *projection* on a particular resource.

This is a very flexible concept that can be used for various purposes. We have already seen how it can be used for synchronization of organizational structure. Similar principle can be used to synchronize Active Directory groups, automatically creating application role for each Active Directory group. This principle provides a mechanism to create multiple accounts for one user on one resource. For example, it can be used to create administration accounts for some users. In this case we would use explicit *kind* and *intent* in the construction:

```
<role oid="0e9c448c-1f87-11eb-9703-b3d28c537192">
    <name>System Administrator</name>
    <inducement>
        <construction>
            <!-- Active Directory resource -->
            <resourceRef oid="1e1e5a1c-1f87-11eb-8ace-1fdb338f61c5"/>
            <kind>account</kind>
            <intent>admin</intent>
        </construction>
    </inducement>
</user>
```

The **System Administrator** role above specifies that a special **admin** account should be created for system administrators. When this role is combined by an ordinary **Employee** role, the administrator

will get two accounts: the usual employee account (`intent=default`) and a special-purpose administration account (`intent=admin`).

Conclusion

MidPoint organizational structure is a versatile and powerful mechanism. It can be used to organize users in units, teams and projects. It can be used to group variety of other midPoint objects. However, organizational structure becomes incredibly powerful when combined with other midPoint mechanisms. Authorizations can take advantage of organizational structures to implement delegated administration schemes. Organizational structures are used in recertification campaigns. MidPoint multi-tenancy mechanism also relies on organizational structures. Organizational structure is a universal mechanism to organize midPoint objects.

There are many universal mechanisms in midPoint, synchronization mechanism being one of the prominent ones. Synchronization was designed to work with many types of midPoint objects, including organizational structure. Expressions and mappings give bring the power to transform organizational structure data during synchronization, supporting diverse set of use cases. Similar mechanisms can be used to synchronize roles and services, granting midPoint enormous flexibility in identity management deployment.

Yet, we are just starting to uncover the full power of midPoint.

Chapter 10. Troubleshooting

The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.

— Theodore Rubin

MidPoint is one big and comprehensive system. Generally speaking, identity management systems tend to be big and complex. They have a lot of things to do. There is a lot of data mapping, synchronization, policies, access control models, expressions and all such stuff. While each individual mechanism in itself is relatively simple, the combination of those mechanisms often creates something that would put the famous labyrinth of Knossos to shame. Especially junior engineers tend to create configurations that are unnecessarily complex. Those configurations may work for common cases. But then there comes a corner case and the result is a puzzled look at junior engineer's face and a mysterious smile of his senior colleagues. At that point junior engineers tend to panic and switch to high-energy trial-and-error mode. Which usually does even more harm. What is really needed here is to stand back, to think about the situation and start to troubleshoot the problem in a systematic way.

But even senior engineers often get into trouble. This is no weakness. MidPoint is really flexible and sometimes it is hard to figure out what exactly is going on. In fact, midPoint has surprised even its authors on more than one occasion. When midPoint misbehaves, then the cause is almost always a configuration problem. But there are usually thousands upon thousands of users, roles, policies and expressions. There is a lot of places where a little treacherous problem can hide and live happily ever after.

All of that was perfectly clear to midPoint developers even before midPoint project started. MidPoint developers are more than just programmers. Writing midPoint code takes a good part of developer's day. But there is also testing, diagnostics of bugs and helping our colleagues and partners to figure out what is going on when the things get really tough. The developers would be lost without an efficient method to diagnose the behavior of midPoint. Therefore diagnostic systems were an integral part of midPoint design.

This chapter will provide an overview of the diagnostics mechanisms in midPoint. But even more importantly, it will describe the method that can help to find the problem in a systematic and reliable way. That is what every engineer should learn and use every day. In fact, troubleshooting is perhaps the most important skill for smooth deployment of any software system.

Designed for Visibility

MidPoint is designed, developed and maintained by a very unique team. What is really unique in midPoint core team is the presence of identity management engineers very early in midPoint design. Several key people that took part in midPoint design were involved in IDM projects since early 2000s. That experience was crucial – especially bad experience from the use of the technologies that were available at the time. One of the most important lessons that can be learned from first-generation IDM systems is a lesson of visibility. All those systems were closed, their vendors jealously guarding all the secrets of inner workings of those systems. This made

troubleshooting a very demanding task. Engineers often had to resort to desperate measures, such as calling vendor's support help desk. But even such drastic actions usually did not help much. It was a real struggle.

When midPoint project started, the team agreed to take a different approach. MidPoint is radically open. It is an open source project from the day one. There is a reasonable documentation, even including architectural documentation and design notes. All of that is public. And most importantly, the visibility goes deep down in midPoint implementation. Great attention was paid for proper logging as that is the primary troubleshooting mechanism. There are various diagnostic mechanisms in midPoint user interface, performance metrics and so on. And all of that is improved in every midPoint release.

However, the most important thing is to know how to use those mechanisms properly. Even the best diagnostics mechanism will not do any good for you if it shows the data that you do not need at the moment. It is important to know where to start, where to look and what to look for. And that are the questions that this chapter should answer.

Systematic Approach

Where to start? That is a question that troubles most engineers that are new to midPoint. Senior engineers would probably think that this is a silly question and the answer is obvious. But it is not. And it gets even harder to figure out how to follow the trace of the problem deeper into midPoint configuration.

Start with the obvious. Maybe there is an error message right in front of your eyes. MidPoint usually provides a lot of details that come with error message. Then there are log files. MidPoint logs a huge amount of information, it just needs to be enabled.

The usual troubleshooting sequence goes like this:

1. **What exactly is the problem?** What are the symptoms? Does the operation fail? Is there an error message? Does it crash? Did it produce wrong results? Or are there no results at all? What was the supposed result? Can this be an error in the input data?
2. If there is an **error message**, what is that message saying? Is there any additional information in the message or in the operation result that comes with the message? Where is the error coming from? Is it an error from a connector? Or an error from an expression?
3. Is there any additional information in the **log file**? Are there any errors or warnings in the log file that may cause the problem?
4. **What was midPoint doing**, exactly? Is there any information about the operation progress when log level is set to DEBUG? What are the intermediary results of the processing? Are those correct? At which point in the operation are data getting wrong?
5. **Where exactly is the problem?** Which component, configuration, mapping or expression are causing it? What are the details of the operation? Are there any hints if you set logging level of that component to **TRACE**?

The first step is perhaps obvious – even though too many people fail to see what is right in front of their eyes. But once you have opened your eyes and checked for the obvious causes, then there is

time to go deeper.

Once you have ruled out the obvious cases you will need to have a look at midPoint log files. You will need to follow the trace and examine the operations that midPoint was doing. The best strategy here is one of *divide and conquer*. In other words, start in the middle. Find a convenient starting point in the middle of the processing. Where exactly that middle is depends on the nature of the problem. If the problem is related to a connector, then the best starting point is probably a connector framework. Does the operation make sense? Are the values correct? If the answer is "yes", then the problem is probably in the connector. If the answer is "no" then the problem is in midPoint configuration. Either way you know where you should focus your investigation. If that problem is in midPoint, then in which part it is? Are data in the user object correct? If they are, then the problem is probably in the outbound mappings. Have a look at those. And so on. Start with a big thing. And then follow the clues and dive into the details.

Error Messages and Operation Results

Error messages are the most obvious troubleshooting mechanisms. MidPoint error messages usually provide enough information to diagnose and fix trivial problems right away.

MidPoint error messages are in fact part of a more complex system of *operation results*. Operation result is a data structure that accompanies every midPoint operation. The operation records important points during the processing in the operation result. Operation result is hierarchical. There is one big operation at the top. But that usually consists of smaller operations which in turn consist of even smaller operations. Connector operations are somewhere at the bottom. If the top-level message does not provide information about the problem, then dig deeper. Expand the sub-operations until you find the root cause.

The screenshot displays three separate error message windows, likely from a Java application's error log or a browser-based interface. Each window has a yellow header bar with a warning icon and the text 'Error communication with the connector ConnectorInstancecfImpl(connector:004e6e8c-df2a-4116-909e-1fbfd55e6bd[ConnId com.evolveum.polygon.connector.ldap.LdapConnector v2.4]): Connection failed: org.identityconnectors.framework.common.exceptions.ConnectionFailedException(Unable to connect to LDAP server localhost:389: ERR_04110_CANNOT_CONNECT_TO_SERVER Cannot connect to the server: Connection refused)->org.apache.directory.ldap.client.api.exception.InvalidConnectionException(ER...'. Below this, the 'Operation Message' section shows the same error message again. The 'Parameters' section lists 'options: [ObjectOperationOptions(resourceRefResolve,readOnly), ObjectOperationOptions(/:resolveNames)]', 'oid: [b4137237-a2c7-489d-b54a-997ced234c96]', and 'class: [http://midpoint.evolveum.com/xml/ns/public/common-common-3#ShadowType]'. The third window is identical to the second, showing the same error message, operation details, parameters, and context/error sections.

Operation result can be used to get a rough idea what midPoint was doing and what are the results. Each of the operations in the result has a status:

Status	Meaning
SUCCESS	Operation completed. The operation has finished successfully. There are no errors.
WARNING	Operation completed. But there are warnings.
PARTIAL_ERROR	Operation completed. Some parts of the operation were successful, other parts of the operations resulted in an error. However, the operation was not stopped and the execution continued despite the errors. Partial error is often indicated in case that user modification is successful but account modification fails.
FATAL_ERROR	Operation was interrupted due to an error. The error prohibited completion of the operation.
HANDLED_ERROR	Operation completed. An error was experienced during execution of the operation. However, the error was handled and the system was able to compensate the effects of the error. The results should be equivalent to a successful operation.
NOT_APPLICABLE	Operation was not even started because the operation is not applicable to the inputs.
IN_PROGRESS	Operation is in progress. The operation was started, but it was not yet finished. This status is seen for operations that execute “in the background”, in running tasks and so on. But it may also be used for operations that are waiting for an external event, such as approval operations or operation retries.
UNKNOWN	Status of the operation is not known. This status code should not be normally seen. However, it may happen under special circumstances. For example, if a bug in midPoint or a connector leaves the operation in an uncertain state or in case that unforeseen error appeared and it was not handled properly. This status usually indicates a programming bug.

Operation result is a very useful data structure. It has many purposes. For example, it is stored in the tasks and provides data for later diagnostics. It can summarize the operations, provide performance data and so on. But that means that the operation result cannot be too big. And therefore the granularity of the operation result is usually quite rough to pinpoint serious issues.

Issues that are really tricky usually cannot be resolved by examination of operation results. For that we need to go deeper still.

Logging

Logging is the best tool in the troubleshooting toolbox. Logging provides information about the important things that happen in the system. But it also goes very deep and it can provide very fine details about midPoint operations. Logging is universal and very powerful. It is the best hope to find the cause even for the trickiest of problems.

As for every other tool, the most important thing is to know how to use it properly. This is especially important for midPoint logging. The default logging setting logs only a very little information. This is a reasonable detail for a production system that has been properly configured and tested. But it is not enough in case that you are chasing a configuration. In that case the logging system needs to be reconfigured to log more information. But beware, if logging system is set to its full power you will get a huge stream of data that is likely to completely overwhelm you. The important information will surely be there. But they will be lost in the flood of other data. Therefore logging need some skill and experience to manage it correctly.

MidPoint is using logging approach that is well established in the industry. MidPoint logging principles should be quite familiar to any deployment engineer. However, it is perhaps worth to provide a quick overview of those basic principles.

MidPoint log files are located in midPoint home directory. This is usually var sub-directory of midPoint installation directory. MidPoint home directory contains sub-directory logs and midPoint logfiles are there. There are usually several files:

- `midpoint.log` is the primary midPoint log file. Almost all log messages from midPoint will be recorded here. This is the right file to examine. The truth is in there.
- `midpoint.out` is file where the standard output of midPoint process is recorded. Only a very few things are usually logged here. Those are the things that happen before midPoint logging system is enabled, therefore midPoint cannot control and redirect logging of those messages. Which that the messages usually describe events that happen before midPoint starts and after it stops. This file does not need to be inspected routinely. However, it is a useful place to look while experiencing startup issues.

Log files are usually rotated. Which means that when there is too much data in one log file the file is renamed. Oldest files are removed. Otherwise the log files would fill all available disk space.

Log messages have a structured format. They look like this:

```
2019-08-16 16:40:25,863 [PROVISIONING] [main] INFO  
(com.evolveum.midpoint.provisioning.impl.ProvisioningServiceImpl): Discovered local  
connector connector: ConnId com.evolveum.polygon.connector.ldap.LdapConnector v2.3  
(OID:268678c0-b5b3-4b13-a399-c2fdb903e51d)
```

The fields are described in the following table.

Field	Description	Example
Timestamp	Timestamp of a moment when the message was generated.	2019-08-16 16:40:25,863
Component name	Name of the component where the message originated.	[PROVISIONING]
Thread name	Name of thread in which the message originated.	[main]
Log level	Severity level of the message.	INFO
Logger name	This is usually package name or a fully-qualified name of the class that produced the message. But there may be special-purpose loggers with special names.	com.evolveum.midpoint.provisioning.impl.ProvisioningServiceImpl
Message	Content of log message. This is usually single-line message. But multi-line messages are common on finer log levels.	Discovered local connector connector: ConnId com.evolveum.polygon.connector.ldap.LdapConnector v2.3 (OID:268678c0-b5b3-4b13-a399-c2fdb903e51d)

As midPoint takes advantage of parallel processing, the thread name is often useful to filter out messages that belong to a single operation. The logger name is sometimes abbreviated, therefore `com.evolveum.midpoint.provisioning.impl.ProvisioningServiceImpl` becomes `c.e.m.provisioning.impl.ProvisioningServiceImpl`. Otherwise the log message format is similar to message formats of other products and it should be quite familiar to most system engineers. This is the default log message format. The log format can be customized if needed.

The most important aspect of efficient usage of logging as a diagnostic tool is to control granularity of logging. This is both a science and an art. It requires some instincts and experience. The granularity can be controlled in two "dimensions":

- **Log level** determines the level of details that is logged. `INFO` log level will log only the important events. `TRACE` level will log huge amount of information that is primarily interesting for developers. This controls *depth* of logging.
- **Package** determines which midPoint components will log their messages. Setting a log level for a particular package also enables logging of all sub-packages and classes. This controls the *breadth* of logging.

Logging setting is a combination of a package and level. Therefore, it is possible to get a very detailed logging from a single package while keeping logging of other packages at a very rough-grained level. And this is exactly what is needed when chasing a bug. We want to have a very precise look at the component where the problem occurs without being overwhelmed by a flood of data from other parts of midPoint. The trick is to know which package and level to use.

Let's start with log levels. Each level has a precise definition of the amount of details it provides.

Level	Circumstances	Description
FATAL	Critical errors. The system cannot continue operation, will crash or stop working	This is bad. The system is going down. There is no way that the system can run. Big problem. This should not happen.
ERROR	Error that seriously affects the system, but the system as a whole can recover.	Typically caused by errors in the data, network errors and so on. This is not good. Sometimes the system can recover just by itself. But manual intervention of system administrator is usually needed.
WARNING	Suspicious situation. System may be able to operate normally, but there may be a hidden or temporary problem or an indication of future error.	Important messages that should not occur in a well-configured and tuned systems. If they appear they should be investigated. However, the investigation can wait. Immediate action is usually not required.
INFO	Important changes in system state, start/stop of important system tasks, etc.	Those events occur normally in almost all running systems. Unless you have a very busy system this log level can be enabled all the time in production.
DEBUG	Execution messages, state changes, expression evaluation messages and similar messages for system administrator.	This log level is dedicated for system administrators to debug the configuration. It will provide reasonable amount of messages that can be used to find configuration problems.
TRACE	Fine-grained messages about execution details.	This log level will provide a lot of data, a lot of details. Its primary purpose is to allow developers to find a bug in production systems. However, this can also provide precious details for system administrators when troubleshooting really tricky problems.

The levels are organized in a hierarchy. When **DEBUG** level is set for a particular package, the package will also log all messages with higher (rough-grain) levels. The usual log level to start with

when chasing a bug is **DEBUG** log level. This may be too much for some packages or too little for other packages. But it is a good overall starting point.

Log levels are simple and well defined. However, figuring out proper package name is much harder. Engineers that understand midPoint architecture and source code have a huge advantage here. Logging package names are directly derived from Java packages and classes used in midPoint source code. But even non-developers can learn how to use the package names efficiently.

The first thing to keep in mind is that midPoint is composed from subsystems and components. Each subsystem and component has its own package name. Those can be used to control logging of individual parts of midPoint. Following table provides an overview of those architectural blocks.

Subsystem/component	Package name	Description
GUI	<code>com.evolveum.midpoint.gui</code> <code>com.evolveum.midpoint.web</code>	User interface. This subsystem drives all interaction with the user. Note: the <code>*.web</code> package is legacy, but it is still used by a lot of code. Both packages are needed for complete GUI logging.
Model	<code>com.evolveum.midpoint.model</code>	This subsystem implements most of the IDM logic in midPoint. User processing, RBAC, organizational structure, policies – everything is processed here.
Provisioning	<code>com.evolveum.midpoint.provisioning</code>	Communication with external systems. This subsystem is responsible for communication with the connectors, management of shadow objects, driving live synchronization, manual connectors, operation retries, management of resources and connectors and so on.
ConnId	<code>org.identityconnectors.framework</code>	ConnId connector framework. This is responsible for running the connectors and passing operation to the connectors.

Subsystem/component	Package name	Description
Repository	<code>com.evolveum.midpoint.repo</code>	Primary responsibility is to store midPoint objects in the database. But there is also task management, authorization processing, expression evaluation and so on.
Schema	<code>com.evolveum.midpoint.schema</code>	Definition of midPoint data model and various utilities.
Prism	<code>com.evolveum.midpoint.prism</code>	Library that is parsing and storing objects in representation data formats (XML/JSON/YAML).

Those package names can provide rough boundaries for logging. Enabling **TRACE** level on an entire subsystem can still provide a lot of data, but it is better than enabling **TRACE** for whole midPoint. The best approach here involves a look at midPoint source code. But there is a still a way to do it without a source code:

1. Enable **DEBUG** level on the entire subsystem. This is likely to provide a log of data, but it should not be overwhelming.
2. Look at the log file and try to figure out in which part the interesting things happen. Where the things are getting out of control. Observe name of the packages that are used in those messages.
3. Set **TRACE** level only for those packages or classes where interesting things happen. You will get much more details.
4. Optionally mute some packages that show too much data by setting their log level to **INFO**.

This is a good overall approach. But there are few very specific packages that tend to attract problems. Therefore they are often set to finer log levels. Below is a list of those packages.

Component	Package name	Description
Clockwork	<code>com.evolveum.midpoint.model.impl.lens.Clockwork</code>	The "controller" that drives computation of all changes in midPoint. Change of every object is passing through clockwork (unless it is "raw"). Enabling logging on clockwork will provide rough overview of the processing.

Component	Package name	Description
Projector	<code>com.evolveum.midpoint.model.impl.lens.projector.Projector</code>	The "brain" that computes all effects of the change. It is invoked as part of the clockwork. Enabling logging will provide overview of the computation.
Change Executor	<code>com.evolveum.midpoint.model.impl.lens.ChangeExecutor</code>	The "hand" that executes all the computed changes. Enabling logging will provide an overview of computed changes and the result of their application (success or failure).
Lens	<code>com.evolveum.midpoint.model.impl.lens</code>	Sub-component responsible for computing and processing all ordinary changes on objects. It includes clockwork, projector and executor. Setting TRACE level here will provide all the gory details about the processing. Lots of data. Use only in deep despair.
Mappings	<code>com.evolveum.midpoint.model.common.mapping.Mapping</code>	Code that is processing mappings. Enabling logging will provide a short overview of mapping inputs and outputs with some insights into the inner processing.
Expressions	<code>com.evolveum.midpoint.model.common.expression.Expression</code>	Expression evaluation code. Enabling logging will provide a lot of details about expression evaluation. This is likely to produce a log of data.
Script expressions	<code>com.evolveum.midpoint.model.common.expression.script.ScriptExpression</code>	Logs a lot of details about script expression evaluation (Groovy, JavaScript, ...). Provides a lot of details.

Component	Package name	Description
ConnId API Operations	<code>org.identityconnectors.framework.api.operations</code>	Special package used to log summary of all connector operations that go through ConnId framework. This is the API side of the framework (midPoint-ConnId boundary).
ConnId SPI Operations	<code>org.identityconnectors.framework.spi.operations</code>	Special package used to log summary of all connector operations that go through ConnId framework. This is the SPI side of the framework (ConnId-connector boundary).
Security	<code>com.evolveum.midpoint.security</code>	Package that contains security-related components. This is especially useful for debugging authorizations.

Setting **DEBUG** log level to clockwork, projector or change executor is a good starting point for diagnostics of problems related to mappings and assignments. The ConnId operation log is a good starting point for connector related problems. And security package is perhaps the only really efficient mechanism to debug misbehaving authorizations.

Logging is the best troubleshooting tool to handle even the most complex issues. Therefore make sure you take full advantage of this tool. The importance of logging can hardly be overstated. Make sure you know how to set up logging properly and how to interpret log messages. This is a skill that takes some time to learn. But it is a crucial investment to make. The time will be repaid many times over. Therefore, if you have any problem that looks strange, remember one simple rule: **always look at the log files**. The answer will be there.

Auditing

Purpose of the auditing mechanism is to record all operations in midPoint for accountability purposes. Auditing will be used by security officers to inspect system activity, it may be used for forensic purposes or it can simply provide a data for statistical analyses. However, the fact that auditing records all operations in the system can be a significant benefit for troubleshooting.

MidPoint user interface is quite comprehensive. Despite that complexity, most operations of the user interface should be easily understandable in an intuitive way. However, there are cases when it is not clear what exactly is user interface trying to do. And some operations can even be quite counter-intuitive. For example, designation of a deputy is an operation on a different user than most people would intuitively expect. It is always an advantage to see all the details of an operation that user interface tries to initiate. And that is where auditing mechanism comes in. The auditing subsystem records all the operations in a precise, structured way. Maybe midPoint does unexpected thing just because the operation request itself does not make sense. Audit trail can be examined to

make sure that the operation is correct. Also, the results of the operation are recorded in the audit trail. Audit may be a quick and efficient way to get an overview about the operation as a whole.

Audit may also be very handy when exploring bulk operations, such as results of synchronization or reconciliation runs. The tasks in which those operations run will provide overview of the results, e.g. they will provide the number of errors. But the task data structure cannot hold the details of each operation. Such data structures would be huge. However, there are audit records for each of those operations. Those records can be used to figure out what went wrong exactly: which objects have failed, what operations were attempted, what exactly is the outcome.

Audit trail is one of the few places in midPoint where historical data are kept. All other parts of midPoint are concerned about *here and now*, historical data are usually kept only for informational purposes. But audit log is different. Audit log stores historical data, therefore it can be used to get an overview of midPoint operations in the past. Common use of audit trail is to get overview of daily operations. For example, audit records can provide data on how many operations were processed during the past day, which operations have failed during last few hours and so on. There is a special type of report to show such information. There is also a dashboard widget that is designed to show such audit-based information for monitoring purposes.

However, please keep in mind that midPoint is an identity management system. It is not a SIEM system or a data warehouse. MidPoint is not designed to keep and process massive amount of historical data. Therefore even the use of audit trails has its limits. Keeping audit trail in midPoint database for a short period of time is usually perfectly acceptable. However, a more suitable system should be used for a long-term storage and processing of audit trail data.

Typical midPoint deployment records audit trails in the database table. This is the right method to use for production deployment. However, there is another option. Audit records can be also recorded in the log files. This is not something that is recommended for a production deployment. In that case it is quite likely to flood the logs and it may even disclose sensitive data. However, directing audit records into system logs may provide interesting benefits in development environments. For example, in case that a detailed debug logging is used, audit records will provide summary of operation and the outcome in the same log together with all the details. It makes it easier to analyze the log files. As audit data is recorded close to the operation start and operation end, audit log entries also provide a “frame” for the operation. It may be easier to find start and end of the operation in the logfile.

Recording audit messages can be enabled in user interface, in the part where ordinary logging is configured. There is an “Audit” section on that page. Audit log message looks like this:

```
2019-08-19 15:02:05,367 [MODEL] [pool-3-thread-6] INFO
(com.evolveum.midpoint_audit.log): 2019-08-19T15:02:05.367+0200 eid=1566219725367-0-1,
et=MODIFY_OBJECT, es=REQUEST, sid=DF97547B47BC6795D941B8C28AFB6089, rid=d0c90fdf-d101-
457b-baf9-ea0371637a1d, tid=1566219725331-0-1, toid=null, hid=localhost,
nid=DefaultNode, raddr=127.0.0.1, I=FocusType:00000000-0000-0000-0000-
000000000002(user), T=PRV(oid=df2210ad-3eec-4f59-9b11-46479b9ebc7c,
targetType={.../common/common-3}UserType, targetName=alice, relation={.../common/org-
3}default), T0=null, D=[df2210ad-3eec-4f59-9b11-46479b9ebc7c:MODIFY],
ch=http://midpoint.evolveum.com/xml/ns/public/gui/channels-3#user, o=null, p=null, m=
```

This is a semi-structure message that provides summary of significant fields of the audit record. Detailed audit logging can also be enabled. In that case the deltas will be dumped in the log files.

Troubleshooting Clockwork and Projector

MidPoint has many components that have diverse responsibilities. But there is one set of components that can be described as a heart (or rather a brain) of midPoint. It is the set of components known as "lens". Clockwork and Projector are two most prominent classes in that set. Projector is responsible for computing the values, running the mappings, processing assignment and almost anything else related to the computation of identity data. Clockwork is responsible for controlling the process. It invokes Projector as many times as is needed to complete the computation. Clockwork also invokes `ChangeExecutor` to carry out the changes.

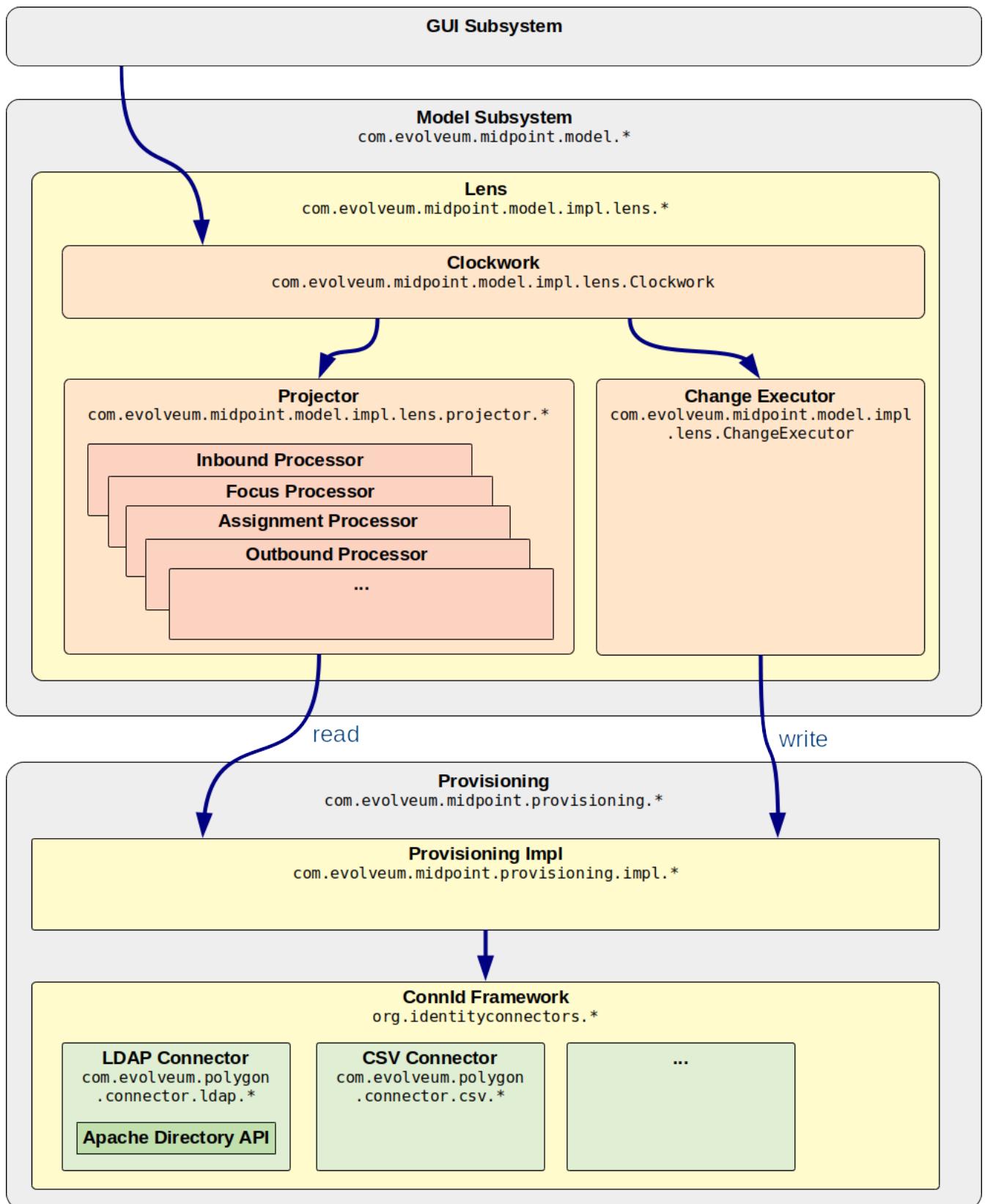
The overall request processing in midPoint works like this:

1. User clicks on *Save* button in midPoint user interface. User interface code computes what operation needs to be done.
2. Operation on `ModelService` is invoked. This is usually `executeChanges(...)` operation. Deltas that describe requested changes are passed as a parameter of this operation.
3. The operation is passed to `Clockwork` to control the operation.
4. `Projector` is invoked to compute all the changes. The changes are recorded in *model context*. The changes are computed, but not executed yet. Mappings and expressions are evaluated at this point.
5. `Clockwork` figures out what to do with the operation. There may be a need to drive the operation through approval process. Or a special hooks may be invoked. Maybe the operation violates the policy rules therefore it needs to be stopped. `Clockwork` does what needs to be done.
6. `ChangeExecutor` is invoked to carry out the changes. Changes to users, roles and other focal objects are carried out by changing the data in midPoint repository. That is quite straightforward. However, changes to projections (objects that reside in resources) are much more complicated.
7. `Provisioning` service is invoked to carry out changes to projections. The changes are expressed as changes to shadow objects (`ShadowType`). Some of those changes are recorded in midPoint database, such as changes in identifier or metadata. However, most of the changes need to be carried out on a resource by using a connector.
8. Connector framework (ConnId) is invoked to initiate resource operation by using appropriate connector.
9. Connector is invoked. Connector initiates the operation on resources and gets the results.
10. Operation results get back to `ChangeExecutor` and then to `Clockwork`. Results are summarized. If there is an error it is decided whether to continue or whether to stop the operation. At the end `Clockwork` records the final audit record and returns control back to the caller. Operation is finished.

Sometimes it is not possible to compute everything in a single pass. There may be dependencies between resources, result of one operation may be an input to another operation. For that reason clockwork and projector work in *waves*. Therefore several steps described above may be repeated

in each wave. Clockwork and projector exchange control in each wave until the operation is done.

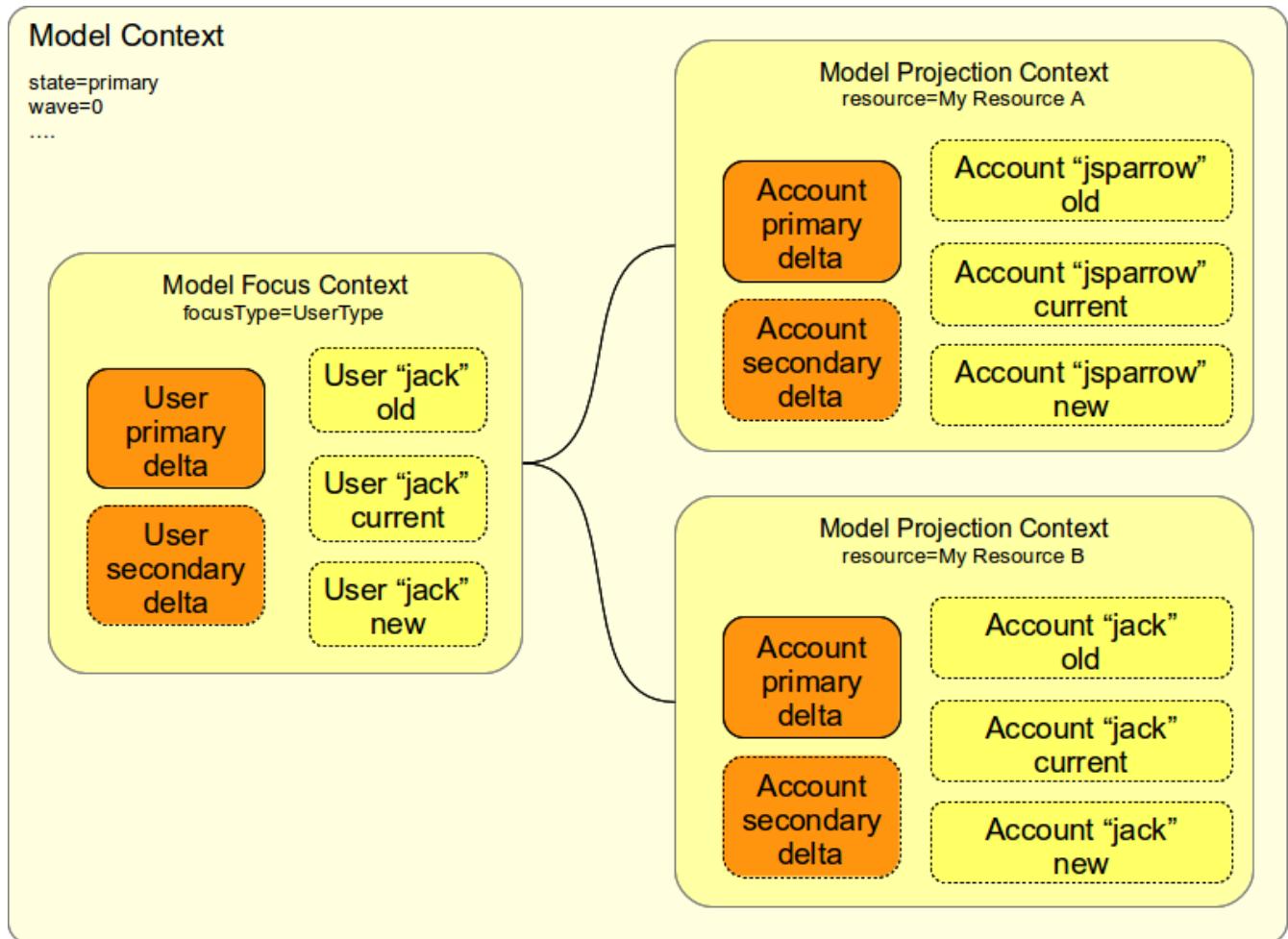
Following picture provides a structural view of this setup.



This process is moving around a lot of data. Those data are recorded in *model context*. It is a data structure that describes the operations, it holds all related objects, intermediary computation results and all other important data. This data structure is absolutely crucial for the entire process. But it also provides valuable troubleshooting information. When appropriate log packages and

levels are enabled, the model context is dumped to log files at important moments during the computation. Often the best way how to find a problem is to watch how the model contest changes during the computation.

The model context is putting together the *focus* and *projections* that belong together. Focus is usually a user, projections are accounts. In such case, the model context groups together a user with all the accounts are associated with that accounts. These are usually accounts that are linked to the user. But it also may be a new account that was not yet created, an old account that was recently deleted, etc. MidPoint groups all these objects together to allow efficient computation of assignments and mappings and other policies.



Model Context has three parts:

- The **context** itself contains information about the entire computation (such as computation state and wave number).
- **Focus** part which contains information about focal object. There is at most one focus.
- **Projection** part which contains information about each projection. There may be multiple projections.

Focus and projection parts have similar structure. Both of these parts contain:

- **Old object:** the object (focus or projection) as it was before the computation. This means really the beginning of computation. Please note that the computation can take several days e.g. if the request waits for approval.

- **Current object:** the object as it was last time the Projector loaded the object. This is usually quite recent information (at most few seconds old).
- **New object:** the expected form of new object after the computation. This item is here mostly for informational purposes and for diagnostics. The actual value of the result may be slightly different (e.g. if two operations are carried out over the same object in parallel).
- **Primary delta:** The request delta. This is the delta that was explicitly entered in the GUI, supplied to the web service or otherwise specified in IDM Model Interface invocation. This is the "command" that midPoint should execute. This defines what user wants. This delta will be executed exactly as it was specified.
- **Secondary delta:** The computed delta. Secondary delta originates from execution of mappings or hooks or other automated mechanisms. This describes what midPoint has computed. This delta will be executed, but it can be recomputed several times during the process.
- **Synchronization delta:** The detected delta. The delta that was detected by synchronization. MidPoint assumes that this delta was already executed and all it can do is to react to this. It is used as an input to the computation. This delta will not be executed again.

This description is slightly simplified. The real thing is more complex. However, this description should be sufficient to understand the overall process.

Model context is dumped at strategic places during clockwork and projector computation. The dumps look like this:

```
---[ PROJECTOR (INITIAL) context projection values and credentials of
resource:10000000-0000-0000-0000-000000000204(Dummy Resource Blue)(default)
]-----
LensContext: state=INITIAL, Wave(e=0,p=0,max=0), focus, 1 projections, 2 changes,
fresh=true
  Channel: null
  Options: null
  Settings: assignments=FULL
  FOCUS:
    User, oid=c0c010c0-d34d-b33f-f00d-111111111116, syncIntent=null
      User old:
        user: (c0c010c0-d34d-b33f-f00d-111111111116, v5, UserType) name: guybrush
....
```

Those dumps provide crucial information for troubleshooting. Their importance for diagnosing really hard problems cannot be overstated. It is more than recommended to get used to read and follow those dumps through the computation process. It will save a huge amount of time.

Why is it so important to know all of this? You need to know this to find your way through this labyrinth. MidPoint provides quite a lot of diagnostic data for each step and each sub-step of each step. You can try to enable full logging to get all the details. But what you get is a digital equivalent of a flash flood and you are very likely to get drowned. The information that you are looking for will be almost certainly there, but it will be lost among all the innocent-looking data. This is a good way how to spend a lot of time and lose your sanity in the process. But it is not an efficient debugging method.

The efficient debugging method is to proceed in steps. Start with high-level information. Then focus your eyes a bit deeper. Try to figure out which steps of the processing are working well and which steps are wrong. Then have a closer look at those steps by enabling more detailed logging. Then look deeper and deeper until the problem is found.

The process usually goes like this:

1. Have a look at input and output of the operation as a whole. There are several ways to do that. You can use "preview" operation of user interface to see the input. You can examine the operation result in the user interface to see the outcome of the operation. Or you may have a look at the audit trail. Enabling auditing to a log files may also help.
2. Have a look at clockwork. Clockwork can provide a summary of the operation when **DEBUG** log level on `com.evolveum.midpoint.model.impl.lens.Clockwork` package is enabled. The summary is an important branch in the troubleshooting process. The summary may suggest whether the problem with the operation is in the computation (Projector) or whether it is in the connectors and resources (Provisioning).
3. Have a detailed look at the Clockwork process. Clockwork summary provides only a brief summary in a very compact form. That information may not be detailed enough to figure out what is going on. Therefore you may need to go deeper. Enabling **TRACE** log level on Clockwork will provide detailed data about all the stages of clockwork processing. Model context is dumped in each step. And it is this dump of model context that provides valuable troubleshooting data. Have a look at those dumps and try to figure out where exactly the operation goes wrong. After that you should be able to decide whether the problem is in computation (Projector) or execution (Provisioning).
4. In case you suspect provisioning issues but you are not sure it may be helpful to have a detailed look at ChangeExecutor. This component is responsible to carry out all the changes from Projector and Clockwork. Enabling **DEBUG** or **TRACE** logging on `com.evolveum.midpoint.model.impl.lens.ChangeExecutor` will provide details about each operation and its outcome.

Overall, clockwork summary is a good starting point. The summary looks like this:

```
###[ CLOCKWORK SUMMARY ]#####
Triggered by focus primary delta ObjectDelta(UserType:c0c010c0-d34d-b33f-f00d-111111111111,MODIFY: PropertyDelta( / {.../common/common-3}organizationalUnit, REPLACE), PropertyDelta(metadata / {.../common/common-3}modifyTimestamp, REPLACE))
Focus: focus(user:c0c010c0-d34d-b33f-f00d-111111111111(jack))
Projections (1): account(ID {.../connector/icf-1/resource-schema-3}uid = [ jack ], type 'default', resource:1000000-0000-0000-0000-00000000104(Dummy Resource Red)): KEEP
Executed:
ObjectDelta(UserType:c0c010c0-d34d-b33f-f00d-111111111111,MODIFY: PropertyDelta( / {.../common/common-3}organizationalUnit, REPLACE), PropertyDelta(metadata / {.../common/common-3}modifyTimestamp, REPLACE)): SUCCESS
ObjectDelta(ShadowType:a3ebbe89-227b-42ff-9d00-f42bee3cf151,MODIFY: PropertyDelta(attributes / {.../resource/instance-3}ship, REPLACE), PropertyDelta(metadata / {.../common/common-3}modifyTimestamp, REPLACE)): SUCCESS
#####
```

At this point you should know the rough outline of the problem. At the very least you should know whether the problem is in:

1. **Projector:** the problem is that midPoint does not compute the values correctly.
2. **Provisioning:** the values are computed correctly, but there is a problem when midPoint tries to execute the operation.

If the problem is in the computation then you need to have a closer look at Projector. The first step should be to enable **DEBUG** logging of `com.evolveum.midpoint.model.impl.lens.projector.Projector`. The output of this logging is similar to the output of Clockwork trace. Each step of Projector computation is recorded and model context is dumped. Watch the changes in model context closely and try to figure out where wrong results occur. This is usually all that is needed to figure out the nature of the problem. The problem is often in the mappings. In that case, follow the instructions in the next section to debug the mappings. If you still cannot figure out what is going on there are still finer details that can be enabled. The projector consists of many "processor" classes, such as `ActivationProcessor` or `AssignmentProcessor`. Each of those are responsible for one part of the computation. Enabling **TRACE** logging on them provides a very fine details about the computation. Package names of those classes can be found in midPoint source code. But it is usually more convenient to enable **TRACE** logging on the entire `com.evolveum.midpoint.model.impl.lens` package at this point. Doing so will also show all the names of all the "processors" that take place during the computation. Those names can be used to focus the logging output only to specific parts of computation.

The things may get quite messy if the problem is in the execution of the operation. There are many components to consider. There is midPoint provisioning code, ConnId connector framework, connector and then the target system itself. There is almost uncountable number of combinations, configurations, network conditions and other circumstances where things may go wrong. The first step should be to figure out whether the problem is on midPoint side or on the resource side. Once again the best strategy is to find a suitable point in the entire process and to check operation status here. Connector framework is such a suitable point. There is special logger that can be used to record summary of all the operations of connector framework:

`org.identityconnectors.framework.spi.operations`. Enabling `TRACE` logging on this logger will record all the operation requests and results that pass between ConnId framework and the connector. Some connectors provide similar facility that can provide even more details. For example, LDAP and AD connectors can log details of all LDAP operations by enabling `TRACE` logging on `com.evolveum.polygon.connector.ldap.OperationLog`. This method is usually preferable as it can clearly indicate whether the problem is caused by wrong operation request or whether the problem occurs on the LDAP or AD server. If case that the problem is in the connector or somewhere on the resource side there is a separate troubleshooting guide below. In case that the problem is in midPoint, then the best strategy would be to enable logging of midPoint provisioning components: `com.evolveum.midpoint.provisioning`. Setting the logging to `DEBUG` level should provide enough information to locate the problem. Desperate engineers can try to use `TRACE` level here. But in that case, it is perhaps a good idea to leave some provisioning classes to `DEBUG` level (such as `ResourceManager`) as they are usually too loud at `TRACE` level.



Lens, Clockwork and Projector. Where do those names come from? Naming is a notoriously hard thing. Software engineers create things that are not alike anything in the real world. Therefore it is often very hard to find good names for components. The recommended practice is to find appropriate metaphor for the system. In other words: find something in the real world that something similar as you do. When midPoint was young we were implementing a component that mapped user data to accounts. However, this component was supposed to be generic. It mapped user to accounts, but also role to entitlements, org to OUs and so on. The obvious name "Mapper" was problematic, as we had a concept of "mapping" already. Such name would be confusing. Therefore, we have chosen a concept of *projecting* user data to accounts in the same way as movie is projected to a screen in a theater. That was also a reason for "focus" and "projection" and to use name "lens" for the whole package. Some time later we needed a name for a controller that will drive the projector. We thought about a planetarium or a telescope driven by a clockwork mechanism. MidPoint has evolved since then and those names may no longer be a perfect metaphor. But they are there and we got used to them.

Troubleshooting Mappings and Expressions

Mappings and expressions often contain custom scripting code. This means that midPoint is very flexible and can satisfy diverse requirements. But it also means that mappings and expressions are often the source of problems. Especially some scripting expressions tend to be quite complex. Creating, testing and maintaining those expressions would be almost impossible without any debugging and troubleshooting facilities.

MidPoint contains code that can be used to trace execution of mappings and expressions on a very detailed level. The trace shows inputs and outputs and deltas that are taken into consideration when the expression or mapping is evaluated. There are two options how to enable this tracing.

First option is to enable the tracing globally for all expressions and mappings by setting one or more of the following loggers to `TRACE` level:

Component	Package name	Description	Verbosity
Mappings	<code>com.evolveum.midpoint.model.common.mapping.Mapping</code>	<p>Code that is processing mappings.</p> <p>Enabling logging will provide a short overview of mapping inputs and outputs with some insights into the inner processing.</p>	Medium
Expressions	<code>com.evolveum.midpoint.model.common.expression.Expression</code>	<p>Expression evaluation code.</p> <p>Enabling logging will provide a lot of details about expression evaluation. This is likely to produce a log of data.</p>	High
Script expressions	<code>com.evolveum.midpoint.model.common.expression.script.ScriptExpression</code>	<p>Logs a lot of details about script expression evaluation (Groovy, JavaScript, ...).</p> <p>Provides a lot of details.</p>	Very high

Setting logger levels to **TRACE** will log all execution of mappings and expressions. However, this may be a huge amount of information, especially in complex deployments with many mappings and expressions. Therefore, there is an alternative way that can be used to trace mappings and expressions individually. There is a special-purpose trace property in the mapping:

```
<mapping>
  ...
    <trace>true</trace>
  ...
</mapping>
```

And there is a similar property in expression:

```

<mapping>
  ...
    <expression>
      <trace>true</trace>
    ...
  </expression>
  ...
</mapping>

```

This is a nice method to look at one particular troublesome mapping without flooding the log files with traces of all the mappings in the system. However, it still may not be entirely easy to locate the dump of the mapping in the log files. Therefore it is a good practice to name your mappings:

```

<mapping>
  <name>my-pretty-mapping</name>
  ...
</mapping>

```

Mapping name will be recorded in the mapping and expression dumps, therefore it can be easily located in the log files. Mapping names are also used in error messages and they are likely to be used in diagnostic outputs that will be developed in midPoint in the future. Therefore it is a very good practice to put names to mappings. It is probably an overkill to name all the mappings in the system. However, naming complex mappings can make a lot of difference in troubleshooting.

Dumping a mapping or expression will provide overview of inputs and outputs. But that alone may not be enough to figure out what is wrong inside the expression. Therefore script expressions can explicitly invoke a logging facility. MidPoint has script expression functions that can be used to log messages from the scripting code. It works like this:

```

<mapping>
  ...
    <expression>
      <script>
        <code>
          ...
            log.error('The {} is broken, {} is to blame', thing, reason)
          ...
        </code>
      </script>
    </expression>
  ...
</mapping>

```

Such messages are recorded in the system log using special-purpose logger `com.evolveum.midpoint.expression` and the appropriate level. The message itself is composed from several parts using `{}` placeholders.

Troubleshooting Connectors

MidPoint is using the ConnId connector framework to manage identity connectors. All ordinary connectors are running under the control of this framework. It means that midPoint calls the ConnId framework and the framework calls the connector. Therefore, when it comes to troubleshooting connector problems there are several places where a problem can occur and also several places where you can get diagnostic data:

1. **MidPoint:** midPoint may invoke wrong operation at the first place. This may be caused by a misconfiguration or a bug. We have already covered most of those cases.
2. **ConnId:** the framework may misinterpret the operation. The framework also simulates some operations and it may post-process the results.
3. **Connector:** this is the tricky part of the story. Each connector is different. Very different. But there are ways. More on this below.
4. **Resource:** it is possible that the problem is caused by resource misconfiguration. E.g. the connector is not allowed to see all data, there are some limits, etc. We will not go into details here. See the documentation that goes with the resource for troubleshooting details.

The ConnId connector framework stands between midPoint and the connectors. It knows about every operation that midPoint invokes on every connector and it knows about all the return values. This can be easily enabled by using the following log configuration:

```
org.identityconnectors.framework.api.operations: TRACE  
org.identityconnectors.framework.spi.operations: TRACE  
org.identityconnectors.framework.common.objects.ResultsHandler: TRACE
```

The ConnId operation traces look like this:

```
TRACE (org.identityconnectors.framework.api.operations.SearchApiOp): method: search  
msg:Enter: search(ObjectClass: inetOrgPerson, null,  
com.evolveum.midpoint.provisioning.ucf.impl.ConnectorInstanceIcfImpl$2@643dc940,  
OperationOptions:  
{ALLOW_PARTIAL_ATTRIBUTE_VALUES:true,PAGED_RESULTS_OFFSET:1,PAGE_SIZE:20})  
...  
TRACE (org.identityconnectors.framework.api.operations.SearchApiOp): method: search  
msg:Return: org.identityconnectors.framework.common.objects.SearchResult@a90221a
```

This is a very useful mechanism. It will log every operation of every connector. If you suspect that the connector is not executing the right operation this is the right place to check it. You can see what is the operation that midPoint is passing to the connector. If that operation looks good then the problem is most likely in the connector (see below). If the operation does not make sense, then the problem is usually in the provisioning (see above).

However, the operation is logged by the ConnId framework on relatively high level and the operation is still quite abstract. If you need more details about what really gets executed you have to rely on the connector logging.

Please note that the ConnId framework has two "faces": API and SPI. The API is facing midPoint. MidPoint invokes ConnId API operations. The SPI is facing the connector. Connector implements SPI operations and ConnId framework is invoking them. You can see the distinction in the class names that are written in the logfiles, e.g. `SearchApiOp` vs `SearchOp` (if there is no `Api` or `Spi` in the operation name then it is assumed to be SPI). There is also similar distinction in the package name of the logger. Most API and SPI operations are direct equivalents. But there may be subtle differences. E.g. The get API operation is executed as search (`executeQuery`) SPI operation.

Most connector operations are "pure" request-response operations: there is one request and one response. These are operations such as create, modify, delete. In this case you will see one request in the log files and one response. And that is the whole operation. Like this:

```
2017-02-01 10:44:16,622 [main] TRACE (o.i.framework.api.operations.CreateApiOp):  
method: create msg:Enter: create(ObjectClass: inetOrgPerson, [Attribute: {Name=uid,  
Value=[will]}, Attribute: {Name=__NAME__,  
Value=[uid=will,ou=People,dc=example,dc=com]}, Attribute: {Name=cn, Value=[Will  
Turner]}, Attribute: {Name=sn, Value=[Turner]}, Attribute: {Name=givenName,  
Value=[Will]}], OperationOptions: {})  
2017-02-01 10:44:16,623 [main] TRACE (o.i.framework.spi.operations.CreateOp): method:  
create msg:Enter: create(ObjectClass: inetOrgPerson, [Attribute: {Name=uid,  
Value=[will]}, Attribute: {Name=__NAME__,  
Value=[uid=will,ou=People,dc=example,dc=com]}, Attribute: {Name=cn, Value=[Will  
Turner]}, Attribute: {Name=sn, Value=[Turner]}, Attribute: {Name=givenName,  
Value=[Will]}], OperationOptions: {})  
...  
2017-02-01 10:44:16,641 [main] TRACE (o.i.framework.spi.operations.CreateOp): method:  
create msg:Return: Attribute: {Name=__UID__, Value=[675f7e48-c0ee-4eaf-9273-  
39e67df4cd2c]}  
2017-02-01 10:44:16,641 [main] TRACE (o.i.framework.api.operations.CreateApiOp):  
method: create msg:Return: Attribute: {Name=__UID__, Value=[675f7e48-c0ee-4eaf-9273-  
39e67df4cd2c]}
```

The above example illustrates a very common `create` operation. It should be interpreted like this:

1. `...api.operations.CreateApiOp Enter`: MidPoint invokes ConnId API. The object to create is logged as an operation parameter. This is what midPoint sends.
2. `...spi.operations.CreateOp Enter`: ConnId invokes the connector. This is what the connector receives.
3. Connector executes the operation. Logs from the connector will be here (if connector logging is enabled).
4. `...spi.operations.CreateOp Return`: Connector operation is finished. The connector returns the result to ConnId.
5. `...api.operations.CreateApiOp Return`: Operation is finished and post-processed by the framework. Framework returns the result to midPoint.

This is quite straightforward and it applies to vast majority of connector operations. However, there are some peculiarities. For example, there are four update operations:

- `update(...)` in `UpdateOp`: This replaces attribute values. It is (roughly) an equivalent to midPoint modify/replace deltas.
- `addAttributeValues(...)` and `removeAttributeValues(...)` in `UpdateAttributeValueOp`. This adds or deletes attribute values. It is (roughly) an equivalent to midPoint modify/add and modify/delete deltas.
- `updateDelta(...)` in `UpdateDeltaOp`: This operation allows complex combinations of add, delete and replace values. This is a new operation designed to replace older operations above.

New connectors implement `updateDelta(...)` operation only. Other update operations are considered to be obsolete. However, they are still used by many connectors.

Search operations are also a bit strange. First of all, the SPI provides only one operation for all search and get operation and that operation is `executeQuery(...)`. Then the results of each object found by the search operations is passed back to midPoint by using a callback method: `handle(...)`. Therefore interpreting search operations takes a keen eye and a bit of practice.

ConnId framework logs should indicate whether the problem is on the "connector-side". Which means that the problem is either in the connector or that the resource itself is not behaving according to expectations. The next step should be to have a look inside the connector. But each connector is different. The connectors have to adapt to the resource communication protocol and therefore they are expected to use variety of client and protocol libraries. Each library may have its own method of troubleshooting. Therefore there is no universal way troubleshoot a connector. However, there is (almost) always some way. Connector documentation should provide some details about troubleshooting. But unfortunately, most connectors do not. The best way is to have a look at connector source code. Enabling logging by using the connector package name is usually quite a safe bet. The logger name is usually the same as the package name of the connector classes. Look in the documentation or directly inside the connector JAR file to find out the package name. You may also need to enable logging of the libraries that come with the connector. You can examine these if you look in the `lib` directory inside the connector JAR file.

Some connectors have really good logging, such as the connectors in the LDAP connector family. The LDAP connector will log all the LDAP operations if you set the `com.evolveum.polygon.connector.ldap.OperationLog` logger to `DEBUG` level.

```

2016-08-30 17:14:20,043 [main] DEBUG
[](c.evolveum.polygon.connector.ldap.OperationLog): method: null
msg:ldap://localhost:10389/ Add REQ Entry:
Entry
dn: uid=jack,ou=People,dc=example,dc=com
objectClass: inetOrgPerson
uid: jack
userPassword: deadmentellnotales
sn: Sparrow
cn: Jack Sparrow
description: Created by IDM
givenName: Jack
l: Black Pearl
displayName: Jack Sparrow

2016-08-30 17:14:20,091 [main] DEBUG
[](c.evolveum.polygon.connector.ldap.OperationLog): method: null
msg:ldap://localhost:10389/ Add RES uid=jack,ou=People,dc=example,dc=com: Ldap
Result
    Result code : (SUCCESS) success
    Matched Dn : ''
    Diagnostic message : ''

```

This logging can be used in two connectors that are used in majority of midPoint deployments: LDAP connector and Active Directory connector. This logging is much more natural and easier to understand than the ConnId framework logging. Therefore, a look at this log should be the first thing to do when there are problems with LDAP and AD connectors.

However, not all connectors are built with troubleshooting in mind. Some connectors will barely log anything. This is all connector-dependent. If the connector author did a good job you will get what you are looking for. If the author did a poor job, you are mostly out of luck. But one way or another, this is the best chance to learn what the connector is doing. If that fails, you have to resort to packet sniffer and similar tools.

Troubleshooting Authorizations

MidPoint authorizations provide a very powerful mechanism for a fine-grained access control. This mechanism is quite simple in principle. But the configuration can get very complex especially if a sophisticated RBAC structure is in place. Setting the authorization up is not entirely easy task. It is often quite difficult to tell why the user is not authorized for a specific action or why the user can access more than he is supposed to. Therefore this page describes basic mechanisms how to troubleshoot authorizations.

The basic troubleshooting steps are simple in theory:

1. Enable logging of authorization processing.
2. Repeat the operation.
3. Figure out which authorization is wrong.

4. Fix it.
5. Rinse and repeat.

Yet, the practice is much more complex. As always.

The authorizations are processed in midPoint security component. The processing of every authorization is logged. Therefore, to see the authorization processing trace simply enable the logging of security component:

```
com.evolveum.midpoint.security: TRACE
```

However, please keep in mind that this is quite intense logging. It can easily impact the system performance and flood the logs on a busy system with a lot of authorization. It is better to troubleshoot the configuration in a development or testing environment.

When the security logging is enabled then you can see following messages in the logs:

```
2017-01-23 14:32:37,824 [main] TRACE (c.e.m.security.impl.SecurityEnforcerImpl): AUTZ: evaluating security constraints principal=MidPointPrincipal(user:c0c010c0-d34d-b33f-f00d-111111111111(jack), autz=[[http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#read]]), object=user:c0c010c0-d34d-b33f-f00d-111111111111(jack)
2017-01-23 14:32:37,824 [main] TRACE (c.e.m.security.impl.SecurityEnforcerImpl): Evaluating authorization 'read-some-roles' in role:7b4a3880-e167-11e6-b38b-2b6a550a03e7(Read some roles)
....
2017-01-23 14:32:37,824 [main] DEBUG (c.evolveum.midpoint.security.api.SecurityUtil): Denied access to user:c0c010c0-d34d-b33f-f00d-111111111111(null) by jack because the subject has not access to any item
```

There is a set of similar messages for every operation that midPoint attempts. First message describes the operations and its "context": who has executed it (principal), what was the object and target of the operation (if applicable). The last line usually summarizes the decision: allow or deny. The lines between describe the processing of each individual authorization. If you examine that part carefully, then you can figure out which authorizations were used. The result of authorization evaluation can be one of these:

- Authorization **denies** the operation. That's a dead end. If any authorization denies the operation then the operation is denied. No other authorizations need to be evaluated.
- Authorization **allows** the operation. That's a green light. However, other authorizations are still evaluated to make sure that there is no other authorization that denies the operation.
- Authorization is **not applicable**. The authorization does not match the constraint for object or target. Or it is not applicable for other reasons. Such authorization is skipped.

If there is a single *deny* then the evaluation is done. The operation is denied. Deny is also a default decision. I.e. if there is no decision at the end of the evaluation, then the operation is denied. At least one explicit *allow* is needed to allow the operation.

All authorization processing is recorded in the log. There is a record of processing of every operation, every authorization, evaluation of every authorization clause, whether the authorization is applicable or not, whether it denies or allows the operation.

Authorization of operations such as *add* or *delete* is quite easy. The result is simple: the operation is either allowed or denied. But it is a bit different for *get* operations. The entire get operation can still be denied if the user does not have any access to the object that he is trying to retrieve. But the common case is that the user has some access to the object, but not all the items (properties). In such a case the operation must be allowed. But the retrieved object needs to be post-processed to remove the fields that are not accessible to the user. This is done in two steps.

Firstly, the set of *object security constraints* is compiled from the authorizations. The *object security constraints* is a data structure that describes which properties of an object are accessible to the user. There is a map ([itemConstraintMap](#)) with an entry for every item (property) which is explicitly mentioned in the authorizations. This entry contains explicit decisions given by the authorizations for every basic access operation (read, add, modify, delete). And then there are default decisions ([actionDecisionMap](#)). These decisions are applied if there is no explicit entry for the item. This *object security constraints* data structure is usually logged when it is compiled.

Secondly, the *object security constraints* are applied to the retrieved object. The constraints are used to remove the properties that are not accessible to the user. This process is not easy to follow in the logs. Therefore it is better to inspect the *object security constraints* structure. If it is correct then also the resulting object will be most likely correct.



Object security constraints and the user interface.

The *object security constraints* has much broader application than just authorization of the read operations. This data structure is (indirectly) used by midPoint user interface when displaying edit forms for objects. The data from this structure are used to figure out which fields are displayed as read-write, which fields are read only and which fields are not displayed at all. The *object security constraints* structure is always produced by the same algorithm in the security component. Therefore, the interaction of authorizations and GUI forms can be diagnosed in the same way as the *get* operations.

Search operations are very different than common operations such as *add* or *delete* and they are also different than *get* operations. Search operations will **not** result in an access denied error (except for few special cases). If the user that is searching has access only to some objects, then only those objects will be returned. There is no error, because this is a perfectly normal situation. The extreme case is that user has access to no object at all. But although this situation is not entirely "normal" it is also not in any way special. The search will simply return empty result and there is also no error. You need to keep this in mind when troubleshooting the *read* authorizations. Attempt to *get* inaccessible objects will result in a security violation error. But *searching* for them will simply return empty result and there is no error.

The *search* operations are interesting for another reason. Operations such as *get*, *add* or *delete* have precise specification of object: the object that is being retrieved, added or modified. But it is entirely different for search operations. The *object* is a result of the search operation, not the parameter. We cannot examine the *object* before the search and decide whether we allow or deny the operation.

There is no *object* before search operation. Also, we cannot simply search all objects and then filter out those that are denied. That would be totally inefficient and it will completely ruin paging mechanisms. A completely different strategy is needed for search operations.

Search authorizations work the other way around: at first, the authorizations statements are compiled to a search filter. For example, if the authorization allows access only to active roles the authorization is compiled to a filter `activation/effectiveStatus=enabled`. Then this filter is appended to the normal search filter and the search operation is performed. This approach ensures that the search returns only the objects that the user is authorized to see. It also makes the search as efficient as possible and maintains page boundaries. But that is not all. Another round of post processing is needed to filter out only the items that are not visible to the user. This is the same filter as is applied to get operations.

Finally, there is a couple of things to keep in mind:

- The authorizations are designed to be **additive**. Each role should allow the minimum set of operations needed for the users to complete their job. MidPoint will "merge" all the authorizations from all the roles. Use allow operations, avoid deny operations if possible. It is much better not to allow an operation than to deny it.
- **Deny always wins.** If there is a single deny in any applicable authorization in any roles, then the operation is denied. It does not matter if there are thousands of allow authorizations, deny always wins. What was once denied cannot be allowed again. We need this approach because we do not have any way how to order the authorization in many roles. Do not use deny unless really needed.
- There are two phases: **request and execution.** The operation needs to be allowed in both phases to proceed. Please keep in mind that object may be changed between request and execution due to mappings, metadata and properties that are maintained by midPoint. This is also the reason why we have separate authorizations for request and execution.
- **Name** the authorizations. Each authorization statement can have an optional name. Specify a reasonably unique name there. Then use that name as a string to find the appropriate trace in the log files.

Authorization traces are quite verbose and there is quite a lot of them. Many traces need to be examined to figure out what exactly is going on. Troubleshooting is a hard work. This mechanism of recording authorization processing in the log is the best way that we have figured out to troubleshoot the authorizations. But we know that it is not ideal. If you have any better idea we are more than open to suggestions.

Reporting a Bug

MidPoint is perfect. There are no bugs. Therefore you can skip this section entirely.

No, that is not really the case. MidPoint is a real software deployed in a real world. And while we spend a huge amount of time and effort to maintain midPoint, test it and fix the bugs, the bugs have a way to always get in. This is also given by the very nature of midPoint. MidPoint is flexible and comprehensive. It is nearly impossible to test midPoint for all the conceivable configurations and use cases. Whenever you deploy midPoint there is a chance that some parts of your configuration

or usage patterns are unique. Those parts might not be used by anybody else yet. Therefore there may be bugs that nobody ever experienced yet. That is the fate of all flexible software products that live in the real world.

In case that you find a bug that needs to be fixed you have several options:

1. **Fix the bug yourself.** MidPoint is an open source product. We will gladly accept bugfixes. However, midPoint is also a substantial product and we need to keep it maintainable. Therefore all contributions including the bug fixes have to be reviewed. The fixes need to be good enough to be accepted. Writing an automated test for the bug is usually required as part of the bugfix contribution.
2. **Report the bug and wait.** The bug will be fixed by someone eventually. However, there is no telling how long you will need to wait. If it is a security-related bug, then it will be fixed as soon as possible. If the bug is severe and it affects a lot of users then it is likely to be fixed soon. However, such bugs are very rare. It is likely that your bug will be quite exotic and it can only be reproduced in your deployment. In that case, it is almost certain that you will need to wait for a very long time. Several years may pass before someone finds the time to have a look at your problem. Those issues are known as *community issues* and they are usually seen at the very tail of developer's work queues. Except for one case: security issues. Security issues are always prioritized regardless of who the reporter is.
3. **Purchase midPoint support from Evolveum.** This will dramatically increase the priority of your bug report. Software development cannot be easily predicted, therefore we still cannot guarantee precise time period to fix the bug. However, typical fix time will be counted in days, weeks or in very rare and complicated cases in months. Those issues are known as *subscriber issues* and they are always prioritized over community issues.

In any of those cases, there is a recommended procedure for bug diagnostics and reporting. There are bug reporting standards that apply to anybody, even midPoint subscribers. Evolveum provides 3rd-line support only. This means that it is expected that the issues has already passed through 1st and 2nd lines of support. The bug report should be really a report of midPoint bug. It should not be a configuration issue. Proper diagnostics techniques should be employed to investigate the issue before it is reported. The rest of this section will describe the recommended procedure.

Diagnostics is the first step that is absolutely mandatory. Troubleshooting techniques described in this chapter should be used to find out what is going on. MidPoint is a very flexible product and vast majority of midPoint issues are caused by wrong configuration. Therefore, please make sure that the problem is not one of those. Simple mis-configuration may easily look like a bug. Try to go through midPoint documentation and understand how midPoint works. Re-read relevant parts of this book. MidPoint development team invests a huge amount of time and effort to make the error messages and log entries are understandable. Then please make sure you use those facilities. Please follow the troubleshooting guides above. They are usually very helpful.

Reproducing the problem is a second step. The easiest way for us to fix a problem is being able to reproduce it. In such case, we do not just blindly fix the problem but we can also make sure it is really gone. In most cases we create a test case in our automated test suite to make sure the problem will gone and it will not appear again. Therefore, your best strategy to make sure that the problem is fixed quickly and does not appear again is to show us how to reproduce the problem in our environment. Saying that "this and that does not work" usually does not help as the same use

case will work perfectly in other configurations. Please describe your configuration in the report.

Try to figure out what is the minimal configuration necessary to reproduce the problem. We appreciate if you could reproduce the problem using our sample resources and objects with minimal customization. This saves you a lot of time describing your environment to us and it also saves us a lot of time to try to re-create your environment in our lab. This approach also helps you to check your configuration and to make sure you are not reporting mis-configuration as a bug.

In a very rare cases the problem cannot be easily reproduced using samples or similar simple setup. In that case we need to work with what we have. Therefore, we either need to know quite a lot about your environment to be able to set it up in our lab. Or we need access to your environment or your cooperation with diagnosing the problem. In such case, please use your common sense in what comes into the bug report. Please keep in mind that midPoint is open source project and the bug reports are public, therefore please be careful when providing sensitive information in bug reports.

When you are sure that the problem is not caused by mis-configuration, it is time to **report the issue**. The best way to submit a bug report is to use Evolveum bug tracking system. The registration is open to everybody. This is also the only bug reporting method available for community issues. Using Evolveum tracker allows you to track the progress of issue resolution, add additional information, etc. Just please keep in mind that the tracker is public and open to anyone following the spirit of open source. Therefore, be careful about submitting a sensitive information.

Please note that security issues revealing a potential security vulnerability should **not** be reported by using the tracker. Information in the tracker is public and this may lead to unintentional disclosure of sensitive information. Special e-mail address is provided for responsible disclosure of security-related issues:

`security@evolveum.com`

Typical bug report contains following information:

- What operation have you tried or what do you want to achieve. Some "bugs" may be caused by trying to achieve something using the wrong mechanism. Having a broader perspective helps us to help you.
- If there is a form or other input to the operation, then please describe how it was set up or filled in. E.g. an XML snippet used to import, data entered into input field, request deltas retrieved from an audit log and so on.
- What kind of resource definition was used, how it was modified, etc. We need to know only the relevant parts. We prefer if you reproduce the problem with the simplest configuration possible (see above).
- Any other special configuration that you feel can influence the outcome, such as custom schema, strange things in expressions, etc.
- If the operation produced an error message in GUI, include that error message as well.
- If there is an exception in the log files, please make sure that you include full stack trace of the exception. The exception stack trace is usually a very efficient pointer to likely cause of the

problem.

- Relevant part of the log files. You may want to have a look at the list of useful loggers above to correctly setup your logging to get the most useful data in the logs.
- Your environment: operating system, Java platform version, target system version. You do not need to bother with this if the bug is obviously not environment-specific.
- Indication of midPoint version (release) or git branch/revision that was used.

Not all of the above is required in a bug report. Use your common sense. As a rule of the thumb too much information is usually better than too little information. But sometimes too much non-relevant information may obscure the tiny problem that would be obvious if just the right amount of information is provided.

Useful Troubleshooting Tips

Finally, there are some generic troubleshooting tips. Those are not specific to midPoint and those tips should be a second nature to any experienced engineer. Some of those were already covered. However, it may still be useful to summarize.

First of all, try to keep your troubleshooting effort methodical and systematic. It makes very little sense to just randomly poke around and hope that the bug will show its ugly head. Even though such random methods may work occasionally, they will require a lot of effort in the end. Try to follow a divide-and-conquer method. Find a boundary in the middle of midPoint, e.g. Clockwork component. Examination of clockwork traces will show you whether the problem is in the mappings or it is in the provisioning. Is the problem in provisioning? Then select another boundary. LDAP connector operation traces may be a good bet in that case. That will show you whether the problem is in midPoint or it is in the LDAP server. Are operation parameters wrong? It means that problem is in midPoint, somewhere between model and the connector. Have a look at debug logs of provisioning component. Maybe there is wrong object class in the resource definition. Have a look at debug logs on the connector. Maybe connector configuration is wrong? The log files will guide you to the problem.

MidPoint log files may look like a maze. But it all makes sense. MidPoint has good component structure and the log files reflect that. You just need to understand how midPoint works and you will not get lost. Just make sure that you never forget to look at the log files. Always look at the log files. When it comes to troubleshooting logfiles are your best friends.

There is one troubleshooting method that is universally applicable to almost any problem. The method involves some specialist equipment. However, this methods provides surprising, almost unbelievable results. To make this method work you have to strictly follow those steps:

1. Describe your problem to a rubber duck.

Yes, a rubber duck. That strange object that usually floats in bubble baths. The duck is a good listener. Therefore just go ahead and describe you problem to the duck. Step by step. Talk about every detail that you have explored. Every possible solution that you have tried. Do not hurry. The duck has unlimited patience. You have to literally talk to the duck. Doing it just in your head does not work. Talk to the duck. The duck will help. It is a wise animal.

As ridiculous as this process might sound, it really works. It does wonders. It is known as *rubber duck debugging* method. Of course, it does not have to be a rubber duck. Any object will work as long as you really talk to it. However, choosing an object with eyes make you feel less stupid while you talk to it. Rubber duck is a popular choice.

That is it. Troubleshooting is not an easy work to do. Although it may sometime resemble witchcraft, it is in fact a science. And an art. It needs some time to find your way. But it is a time well spent. It will be repaid many times over.

Chapter 11. MidPoint Development, Maintenance and Support

Those who do nothing but observe from the shadows cannot complain about the brightness of the sun.

— Frank Herbert

MidPoint is a *professional open source project*. This means that midPoint is developed by using professional methods, but the product is still available under open source license.

Professional Development

MidPoint is developed by a professional developers. The development is lead by senior developers in the midPoint core team that have decades of software engineering experience. There are also few younger developers in early stages of their careers. MidPoint development team is first of all a community of developers that enjoy working together on a next-generation software. Professionalism is a strict requirement for all midPoint development, but it is mostly the engineering passion that really moves the project forward.

All midPoint core developers work for Evolveum. Evolveum is the company that created midPoint. Evolveum also maintains midPoint. Vast majority of work on midPoint is being done by midPoint core team. All the core developers are paid for their work on midPoint. The developers can pay their bills from the income that midPoint generates. Evolveum income from midPoint is necessary to make sure that the developers have all their time available for midPoint development. This means that midPoint can be properly maintained.

Professional development also means that professional software engineering methods are used to develop and maintain midPoint. MidPoint development is firmly founded on principles of continuous integrations. There are literally thousands of automated integration tests that are part of midPoint build process. Thousands of additional automated tests are running every day. There are tests that closely reflect real-world configurations and use cases. There are tests with real resources. All of that is an integral part of midPoint development. MidPoint is a comprehensive and very flexible system. Professional quality assurance is essential for midPoint to work reliably.

Open Source

MidPoint is an open source project. All of midPoint source code is available under two open source licenses. We have chosen Apache License 2.0 as this is one of the most liberal licenses out there. We are also based in European Union and therefore midPoint is also under the terms of European Union Public License (EUPL). But there is much more to open source than just a license. Evolveum is fully committed to the open source approach. MidPoint is completely developed in public. Entire history of midPoint source code is public. Every commit of every developer is immediately available to anyone. Complete midPoint source code is available. There are no private parts that are held back by purpose. There are no private branches with extra features. Even all the support branches are completely public. When it comes to the source code midPoint is as true to the open

source methods as it gets.

Even though vast majority of midPoint development is done by Evolveum, open source approach is absolutely critical for the success of midPoint. Open source is the only way that allows midPoint users to understand midPoint completely. All non-trivial software needs to be customized in some way and open source brings the ultimate power of customization. Open source allows participation. Open source is great approach to avoid vendor lock-in. Open source brings longevity to the project. Open source has so much advantages. Evolveum is completely committed to the open source approach.

MidPoint has started as an open source project. MidPoint source code was available from the day one. And as far as we have something to say about it midPoint will remain open source forever.

MidPoint Release Cycle

MidPoint has stable development cycle. There are two *feature releases* every year. As the name suggests, those releases are bringing new features and major improvements.

In addition to that, there are several *maintenance releases*. Those releases bring bugfixes and minor improvements. Maintenance releases are published as needed, there is no strict schedule. Timing of maintenance releases is influenced by midPoint subscribers.

Every couple of years there is a special long-term support (LTS) release. This release is supported for a longer time than usual. This release is ideal for people that prefer stability over new features.

MidPoint Support and Subscriptions

MidPoint support and subscription is a service provided by Evolveum. There are several service offerings with different scope and service level. But generally speaking, the most common service is 3rd-line support. Which basically means that we will fix midPoint bugs. Obviously this includes assistance with diagnostics of difficult issues where it is not entirely clear whether it is a bug or configuration issue. Simply speaking, midPoint support service is a way how to make sure that your midPoint deployment will run without any problems. There are also subscription offerings designed to help you deploy midPoint in the first place. Some subscription offerings also contain feature development and improvements (a.k.a. 4th-line support). Those subscriptions are ideal way to make sure midPoint will be able to do anything that you need for your project.

MidPoint support and subscription services provide significant funding for midPoint development and maintenance. Therefore, it is perfectly natural that midPoint subscribers get high priority for resolution of their issues, feature requests and so on. This limits the time that midPoint core team has available for other tasks. Therefore there are some rules:

- Every new midPoint feature must be *sponsored*. This means that a customer with an active midPoint subscription has endorsed the feature. Of course, this has to be a high-end subscription that includes feature development. Or, alternatively, someone have to pay for the development cost of that feature. However, direct feature sponsoring is very limited as most of the midPoint development capacity is reserved for subscribers.
- MidPoint architecture and quality is the primary responsibility of Evolveum team. Part of

Evolveum income is reserved to maintain midPoint - to keep the architecture up to date, to make systemic quality improvements, to maintain midPoint in the long run and so on. In some cases Evolveum will sponsor feature development. Those are usually strategic features that lead midPoint development in the right direction. Or those may be experimental features aimed at exploring a particularly interesting functionality. However, Evolveum will not sponsor any "customer" features. Those need to be covered by subscriptions.

- Evolveum will eventually fix any bugs in midPoint. Those bugfixes will be committed to midPoint primary development branch (*master branch*). The fixes that make it into development branch will be part of the next feature release. However, as midPoint release cycle is fixed, not all of the bugs will be fixed in each release. The bugs that were reported as part of subscription or support service will be fixed first. If there is still some time, then other (non-subscriber) bugs will be fixed as well. But there are no guarantees for that. If the time before the release runs out, bugs reported by non-subscribers will not get fixed. In fact, such non-subscriber fixes may have to wait for several releases until they finally get fixed.
- Every feature release has a *support branch*. This is where the maintenance releases come from. However, every bugfix or improvement is developed on master branch first. It has to be backported to the support branch. Which takes time. Therefore, there are very strict rules for backporting. Any bugfix, improvement or any other update will go to the support branch only if:
 - Backport to support branch is explicitly requested by customer with active support or subscription service.
 - The target release is still in its active support period (i.e. it is not after "end of life").
 - It is a security issue. Security issues have absolute priority. Those will be fixed immediately regardless of who reported them (subscriber or non-subscriber). Fixes for severe security issues will also get automatically backported to all active support branches.
 - It fixes a severe issue that affect large number of users.

Simply speaking, if you want to make sure that midPoint works for you, then purchase a support or subscription. Those services will help you, that is what they are for. But the money from support and subscription services also enable long-term midPoint maintenance and new feature development. Getting midPoint support or subscription is the right thing to do.

MidPoint Community

MidPoint is a proper open source project. And as all good open source projects midPoint has a vibrant community. This is both engineering community and business community. The primary communication channel of the engineering community is midPoint mailing list. Mailing list is used to discuss midPoint futures, announce new releases, discuss configuration issues, provide feedback to the development team and so on. MidPoint community is open to anyone.

Business community is formed mostly from Evolveum partners. Evolveum partners deliver midPoint solutions, provide 1st-line and 2nd-line support services, provide professional services, customized solutions based on midPoint and so on. The possibilities are endless. Even the business community is open. Entry-level partnership is open to anyone. However, there are several partnership levels and it takes some effort for a partner to level up. There is a rich (and growing) network of midPoint partners. The partners can deliver solutions based on midPoint almost

anywhere on planet Earth.

Chapter 12. Additional Information

Logic's useless unless it's armed with essential data.

— Leto II, Children of Dune by Frank Herbert

We have tried to make this book as comprehensive as possible. But no book can possibly include all the information that an IDM engineer would ever need. Therefore, this chapter describes the sources of additional information about midPoint.

MidPoint Wiki

MidPoint wiki is the most comprehensive information about midPoint. This is where all the midPoint documentation is stored. But there is much more. There is connector documentation, midPoint architecture, developer documentation, midPoint internals and all kinds of information including midPoint release planning and roadmap. Everything is public.

However, the wiki is so comprehensive that it is often not entirely easy to find the right page. We have done what we could to organize the information. The pages are organized hierarchically. Many pages have "See Also" section that points to additional information. Yet, the practice shows that if you want to find something in the wiki, you need to have at least a faint idea what you are looking for. This book should give you that idea. If you know what you are looking for, then the wiki search bar is your friend. If you enter the correct search term, then there is high probability that you will quickly find the right page.

URL: <https://wiki.evolveum.com/>

Samples

The midPoint projects maintains quite a rich collection of samples. These are sample resource definitions, role and organizational structure examples and other various samples. They are usually provided in the XML form. The samples are maintained in a separate project on GitHub. Those samples are also part of midPoint distribution package.

URL (latest version): <https://github.com/Evolveum/midpoint-samples/tree/master/samples>

Book Samples

This book contains many examples and configuration snippets that are taken from various places. Some of the smaller snippets are taken from midPoint wiki or from the sample files (see above).

Some chapters contain mostly complete configurations of the midPoint deployment. These configurations have a separate folder in the midPoint samples. Look for a folder **book** in midPoint samples. All the important files used in this book are there, sorted by chapter number.

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

Story Tests

MidPoint developers like to create and maintain complete end-to-end automated tests. These tests are usually inspired by real-world midPoint deployments. We call them story tests. These tests are important to maintain midPoint quality and continuity. However, they are also excellent source of inspiration and they have often proved useful as examples of midPoint configuration.

Wiki description: <https://wiki.evolveum.com/display/midPoint/Story+Tests>

Code and configuration: <https://github.com/Evolveum/midpoint/tree/master/testing/story>

MidPoint Mailing List

MidPoint project attracted a vibrant community during the years. The main community communication channel is midPoint mailing list. The mailing list is used for announcements, user suggestions and also what we at Evolveum call *community support*. The mailing list is used to ask questions about midPoint. Experienced community members usually answer these questions and provide pointers to additional information. The whole midPoint development team is also subscribed to the mailing list and they provide answers when needed. However, this is a best effort service. Please do not abuse this communication channel and try to keep the following community guidelines:

1. **Be polite.** Mailing list is a best effort service. Nobody is (directly) paid to answer mailing list questions. The engineers that answer the questions are doing that in addition to their day-to-day responsibilities and they are doing that because they want to help the community. Therefore, if you are asking for help, do so politely. If you are answering a question please respect other members. Everybody started somewhere and it is natural that novice users do not know everything. Please tolerate the differences in skill sets.
2. **Do some research before asking a question.** Do not ask trivial question that can be easily answered by googling the question, by searching for it in the midPoint wiki or mailing list archive. If you are getting an error try to read error message very carefully and try to think about the possible causes. Try to experiment with the configuration a bit. Look at troubleshooting section of this wiki. Spend at least a couple of minutes to make your own research before asking the question. If that research does not provide the answer, then it is a good question for the mailing list.
3. **Provide context.** If your post looks like "*my midpoint is broken, please help*" then it is very unlikely that you will get any answers. Try to describe your problem in more details. Make sure to describe relevant bits of your configuration. Be sure to include error message. Look in the log files if necessary. And most importantly: describe what you are trying to achieve. Maybe the root of your problem is that you are using completely wrong approach. The community may point your nose in the right direction - but only if they know what is your goal.
4. **Give back.** Mailing list is not one way communication channel where users ask questions and developers answer them. There is already a significant body of knowledge distributed among community members that are not midPoint developers. If you adhere to these guidelines and ask a question it will most likely be answered. But for that there needs to be someone who is answering. Therefore do not just ask the questions. If you know the answer to the question that someone else asks then please go ahead and answer it. Do not worry that your answer may not

be perfect. Even a partial answer will be greatly appreciated by any novice user. Simply speaking: Do not only take from the community. Try to repay what the community gave you.

You may also be tempted to send your questions directly to Evolveum or midPoint developers. However, the developers have many midPoint users, partners, customers and contributors to deal with in their day-to-day job. The first responsibility of any midPoint core developer is to make sure that midPoint development will continue. The developers naturally prefer to spend time doing tasks that bring funding to the midPoint project. Therefore, the developers will strictly prioritize the communication. Answers to midPoint subscribers are highest priority, mailing list is second and answers to private messages from the community are absolutely lowest priority. We prefer efficient spread of knowledge about midPoint. Mailing list is good for that, but private communication is not. That's the primary reason for this priority setup. Besides, if you contact a developer directly, then only that developer can answer your question. But if you send the question to the mailing list there are more people that can potentially answer the question. Therefore, unless you have active subscription the mailing list is your best option.

Mailing list URL: <http://lists.evolveum.com/mailman/listinfo/midpoint>

Evolveum Blog

Vast majority of midPoint development and maintenance is conducted by Evolveum team. The people of Evolveum present their professional opinions by the means of Evolveum blog. The blog is very technology-friendly. The information provided on the blog goes often quite deep. This is also a channel how Evolveum shares information about midPoint development plans and business activities related to midPoint. It is a very valuable resource for anyone that has a professional interest in midPoint.

URL: <https://evolveum.com/blog/>

To Be Continued

Hanc marginis exiguitas non caperet.

(There is not enough space in the margin to write it.)

— Pierre de Fermat

This is it then. That was the last real chapter of this book. Yet, we have not yet covered all capabilities of midPoint. In fact, we are not even close. We have only scratched the surface of what midPoint can provide. The other chapters are not written yet. There is still so much to write about:

- **Authorizations:** MidPoint works with sensitive data and there is a need to strictly control access to that data. Therefore, midPoint has a fine-grained authorization system for controlling access to itself. Authorization mechanisms are very powerful allowing many scenarios from delegated administration to partial multi-tenancy.
- **Archetypes:** Users, roles, orgs and services are the four basic object types of midPoint. They are quite powerful. Yet, there is often a need to define special behavior for employees, agents, students, academic staff, partners and other types of persons. We would also like to see business roles, application roles, system roles. The enterprises would be lost without their divisions, departments and sections. There are always shades, flavors and subtypes to objects. That is what the *archetypes* do. They can be used to create subtle flavors of functionality suitable for individual object types.
- **MidPoint queries:** MidPoint often needs to find objects in the repository. There is a special-purpose query language that is used in many places in midPoint.
- **Security:** MidPoint works with quite a sensitive data, therefore it is quite important to keep midPoint secure. There are many security-related settings, ranging from the usual network security mechanisms, through authentication to a midPoint-specific settings.
- **Identity management miscellanea:** There are various interesting features that were not mentioned yet: iteration, password policies, notifications, auxiliary object classes, provisioning dependencies, deputies, constants, function libraries, provisioning scripts and so on. Those may be little features, but they are essential pieces of the puzzle. It is almost impossible to have a complete identity management solution and not to use any of those features.
- **Requests and approvals:** Browse available roles, select role, request role, approve role, assign role, provision accounts. That is the basic mantra of many identity management deployments. Of course this is easy to do in midPoint. But there is much more: multi level approvals, optional approval steps, dynamic approver selection, escalation and so on. MidPoint has most of the features already built-in, you just need to configure them.
- **Entitlements:** Managing accounts is fine. Yet, it is not the whole story. There is huge difference between a regular account and an administrator account. Fortunately, midPoint can easily manage membership in groups, roles, assignment of privileges and other entitlements. In this case we really mean entitlements on the resources, such as Active Directory groups, distribution lists, Unix groups and so on. MidPoint is designed to this quite easily.
- **Manual resources:** Obviously, you want most of the resource to be connected to midPoint by a connector so they can be automatically managed. But there are always few bothersome resources that just won't comply. Maybe they are too small to justify the cost of building a

connector. Maybe there is just no good way for the connector to manage the resource. But midPoint one again comes to the rescue. MidPoint has a concept of *manual resource* where the work is done by system administrator instead of connector. There is even a way how to create semi-manual resource that can read the data, but provisioning is still manual. And there is a way how to integrate this with ITSM system.

- **Auditing and history:** No identity management system can be complete without an auditing facility. MidPoint can store every operation to the audit trail: changes in users, accounts, roles - even internal configuration changes. This is stored in a format that can be used to integrate midPoint with a data warehouse or a SIEM system. Also midPoint user interface has a facility to display the audit trail. And it can even look into the past: it can reconstruct the objects as they were at a certain point in time.
- **Policy rules:** MidPoint is much more than just an identity management system. The identity governance features of midPoint are based on a powerful and universal concept of *policy rules*. The rules can be used to express role exclusions, thus defining a segregation of duties (SoD) policy. The rules can be used to define policy-based approval. The rules can control role lifecycle. The rules can define compliance policies. The rules can do it all. The rules are here to govern the identities.
- **Access certification:** This is known by many names: certification, re-certification, attestation, ... but whatever the name is it is still the same process. Simply speaking, this is a method to review roles assigned to the users to make sure the users still need the roles that they have. This is a method how to get a grip on the *principle of least privilege* even in environments that are naturally inclined to ad-hoc operation. But it is very useful mechanism in almost all environments. And midPoint provides many flavors of certification mechanisms from a scheduled mass recertification campaigns focused on roles assignments to an ad-hoc recertification of a single user after he is moved to a new organization.
- **Data protection:** Identity management is no longer a wild west where anybody can do anything. Now there are strict data protection rules, regulations and legislation. Being a good identity governance system midPoint can assist in managing the data protection and privacy policies. MidPoint can be really helpful in managing compliance to the data protection regulations such as European GDPR legislation.
- **User interface customizations:** MidPoint has a general-purpose user interface that can be used for user self-service, identity administration and system configuration. The user interface is designed to be dynamic. It will automatically adapt to resource schemas, extension of midPoint schema, authorizations and so on. Therefore, usually there is no need to customize user interface at all. But there are cases when the deployment need to deviate from the default behavior. And midPoint is prepared to that. There are many ways how to customize user interface: colors, stylesheets, localization, custom forms, tabs, whole new custom pages. In extreme cases midPoint can be customized beyond recognition.
- **Integration with midPoint services:** MidPoint is a great system. But even great software does not live in isolation. There is always need to integrate the systems together. Integration runs through midPoint veins, because that is what the connectors really do. But often there is a need to integrate midPoint with other systems that is beyond the capabilities of a connector. Maybe there is a password reset application that needs to interact with midPoint. Maybe there is a analytic software that needs to get midPoint data. MidPoint was designed from the day one to be a service-based application. Therefore there are REST and SOAP services packed with

features. Actually almost anything that midPoint does can be controlled by using those services.

- **Advanced concepts:** There are still some features that were not explained in previous chapters: consistency mechanisms, personas, multi-connector resources and so on. Some of those features are seldom used, but they may save your project. Other features are used all the times, but they are a natural part of midPoint and therefore they are almost invisible. But all those features deserve explanation. And there is also a need to describe how midPoint itself is developed – as there is a lot of experimental and incomplete features. But, as this is midPoint, even those features may be extremely useful. This chapter may also be interesting to people who would like to extended midPoint in an unusual way or those that want to contribute to midPoint development.
- **MidPoint Deployment:** There are many path from downloading midPoint packages to a working system. Some of those paths are easier than other. MidPoint design was build on many years of practical identity management experience. Therefore, midPoint has mechanisms that can be used to efficiently overcome some of the notorious problems in identity management – provided that midPoint is used correctly. This chapter aims at giving advice how midPoint should be used in practical projects. How to plan the project, what information to gather, how to design the deployment, how to prepare the environment, plan the migration, handle project extensions and changes and so on.
- **Management of IAM program:** Identity management is very similar to information security: Identity management has no end. Identity management is not a project. It is a program. It is an endless cycle of gathering data, planning and execution. The environment around identity management is always changing, therefore identity management must change as well. But midPoint is designed for this kind of longevity. This chapter will describe how to handle this endless cycle. How to make midPoint configuration open to extensions. How to gather data. How to handle new feature requests. How to do upgrades. How to keep identity management solution sustainable.
- **Deployment examples:** This book uses a lot of examples in all the chapters. But those are examples designed to demonstrate one specific aspect of midPoint functionality. This chapter will be different. There will be complete examples of practical midPoint solutions. After all, the way that copying and pasting is one the best ways how to learn.
- **Glossary:** Identity management and governance parlance may sound like an alien language to a newcomer. Therefore, perhaps preparing IDMimsh-to-english dictionary might be a good idea.

Those chapters are still missing. They are not written yet. Obviously, the best people to write those chapters are the people from the Evolveum team: people that designed and implemented midPoint, people that support midPoint deployments, people that work with midPoint every day, people that eat, breathe and sleep midPoint. But those people are just engineers. They need to pay their bills. They cannot put away their day-to-day responsibilities to work on this book. Obviously, funding is needed to finish the book. As this book is available for free there is no direct income that could provide the funding for next chapters. There is only one way: sponsoring.

If you like this book, then please consider sponsoring some of the next chapters. The market economy is, unfortunately, quite ruthless. Therefore, it is pretty straightforward: if there are no sponsors, it is very unlikely that there will be any new chapters. Therefore, please sponsor this book if you can. If you cannot afford to sponsor this book, then please at least help us to spread the word: a word about midPoint and a word about this book. Any form of help is more than

appreciated.

Conclusion

A conclusion is the place where you get tired of thinking.

— Steven Wright

This book is not finished yet. In fact, it is just a beginning. The books is written continually in an incremental and iterative fashion as fast as time and money allow.

Your contributions and donations will surely speed up completion of this book. Also consider getting midPoint subscription. That is the source of funding for midPoint development. Evolveum is no mega-corporation that can fund open source development from huge profits in other areas. There are no other areas. Evolveum is open-source-only company. Everything we do is focused on open source projects. Evolveum is not a start-up either. We are not funded by venture capital, and we do not have millions to spend. Evolveum is a self-funded company. We can spend only what we earn on subscriptions, sponsored features and services. There is no other income. MidPoint development can only go as fast as the money allow. The same principle applies to midPoint documentation and this book. It will grow proportionally to the Evolveum income. Therefore, if you liked this book please consider supporting us. Both money and your time are more than appreciated.

We hope that you enjoyed reading this book at least as much as we enjoyed writing it – and as we enjoy creating midPoint in the first place. MidPoint is quite a unique software project. It would not be possible to maintain and develop this project without you. The whole Evolveum team would like to thank all past, present and future midPoint supporters for making this exciting project a reality. Together we have created interesting and useful software product. We hope that together we can make midPoint even better.

Thank you all.