****

Zentity (Version 2.0)

Security User Guide

May 2009

Contents

[Introduction 3](#_Toc273432231)

[Overview 3](#_Toc273432232)

[Features 3](#_Toc273432233)

[User management 3](#_Toc273432234)

[Built in users and groups 4](#_Toc273432235)

[Permissions Model 4](#_Toc273432236)

[Permission types 4](#_Toc273432237)

[User and group permissions 4](#_Toc273432238)

[Grant/Revoke/Remove permissions 4](#_Toc273432239)

[Permissions hierarchy 4](#_Toc273432240)

[Read/Update/Delete permissions 5](#_Toc273432241)

[Owner Permission 5](#_Toc273432242)

[Denying vs. Removing Permissions 5](#_Toc273432243)

[Restricting unpublished resources 6](#_Toc273432244)

[Authorizing operations 6](#_Toc273432245)

[Components 6](#_Toc273432246)

[High Level Design Diagrams 7](#_Toc273432247)

[Application Configuration 9](#_Toc273432248)

[Code samples 11](#_Toc273432249)

[Known issues 15](#_Toc273432250)

[Glossary 16](#_Toc273432251)

[Core 16](#_Toc273432252)

[Authentication 16](#_Toc273432253)

[Authorization 16](#_Toc273432254)

[References 16](#_Toc273432255)

# Introduction

Security is fundamentally about protecting assets – assets could be files, databases, tables, rows, and so on, together with system-level resources such as registry keys and configuration data.

Security consists of many features, out of which two are most prominent – Authentication and Authorization.

**Authentication** addresses the question: who are you? It is the process of uniquely identifying the clients of your applications and services. These might be end users, other services, processes, or computers.

**Authorization** addresses the question: what can you do? It is the process that governs the resources and operations that the authenticated client is permitted to access.

More details about security fundamentals can be found [here](#FundamentalsReference)

Zentity security module aims at protecting ‘Resource’s and its derivatives through authentication and authorization.

# Overview

The Zentity Security module provides various components for authentication, user management and authorization. All Zentity applications and services have been made security aware by integrating the security APIs.

# Features

Zentity consists of the following security features which are common for all Zentity services and applications.

1. Authenticating users
2. User management
3. Authorizing operations requested by users
4. Permissions management

# User management

The Zentity Security module provides the following user management features.

1. User and group creation, update, and deletion.
2. Assigning or removing users from groups.
3. User profile update
4. User status management – the administrator can set a user account status to inactive/disabled/locked etc., for users not logging in to the Zentity installation for a long time, for invalid login attempts etc.
5. Change password.
6. Forgot password – users can get a new password when they provide the security question and answer to validate their identity. The security question and answer should match those provided at the time of user account creation.
7. Administrators can impose a password policy on the users. The policy is optional and configurable. It contains policies like minimum and maximum password length, password complexity, password expiry etc.

## Built in users and groups

1. Administrator. A built in administrator account is provided, which cannot be changed or removed.
2. Administrators group. More people can be assigned as repository administrators by adding their accounts to the administrators group.
3. Guest user. This is an anonymous login to the repository, and can be used to access public resources.

# Permissions Model

The Zentity Security module has the following permissions model.

## Permission types

1. There are five permissions available in the security module – Create, Read, Update, Delete and Owner.
2. Create is a repository level permission, while other permissions are granted on a resource.
3. The module supports three permission management operations –
   1. Grant – Grant permission to a user or group.
   2. Revoke – Deny permission to a user or group.
   3. Remove – Remove a granted and / or revoked permission from a user or group.

## User and group permissions

1. An administrator has full access to all resources in the repository. No permission needs to be granted to administrators. Also no permission can be revoked from administrators.
2. The guest user has Read permission to resources. The Read permission can be revoked from the guest, however, he cannot be granted Update/Delete/Owner permissions on any resource.
3. For granting or revoking permission from all users at once, grant to/revoke from the AllUsers group.
4. User inherits permissions granted to/revoked from his group(s), including the AllUsers group.

## Grant/Revoke/Remove permissions

1. Administrator grants create permission to the users.
2. When a user creates a resource he is granted Owner permission on the resource.
3. An owner of the resource or an administrator can grant/revoke permissions. Attempt to grant/revoke/remove fails if the user does not own the resource (and is not an administrator).

## Permissions hierarchy

1. The permissions priority in increasing order is as follows –
   1. Read
   2. Update
   3. Delete
   4. Owner
2. Whenever permission is granted all permissions which have lower priority than the given permission are also automatically granted to the user. That is, granting ‘Owner’ will grant Owner, as well as Delete, Update and Read. So if a user has Update permission on a resource it means he has Read permission as well.
3. In case of revoking permissions the sequence is reverse. That is, revoking permission also revokes all permissions which have higher priority than the given permission. For example, revoking ‘Read’ will also revoke Update, Delete and Owner permissions. Revoking Delete will revoke Owner permission as well.
4. The security module maintains consistent state of permissions all the time for a given resource and a given user or group. For example, granting ‘Update’ will remove deny-Update permission if any. Revoking ‘Delete’ will remove Delete permission, if already granted to the user/group.

## Read/Update/Delete permissions

1. All users have Read permission on all resources by default. That is, Read is an implicit permission. However Read permission may be revoked from users if the Owner of the resource or administrator wishes so.
2. Update and Delete permissions have to be granted to users/groups. Without a grant users do not get Update/Delete access on resources.
3. In case a user has permission but any of his groups is denied (revoked) the permission, then effectively user is not able to perform that operation on the resource. The same is true when a user is denied permission but his group has been granted the permission, or when the user belongs to two groups which have conflicting permissions on a resource. That is, denied permission takes precedence over allowed permission.

## Owner Permission

1. When a user creates a resource he is granted Owner permission on the resource.
2. If a group is granted Owner permission on a resource, users in the group also get ownership of the resource. This is similar to other permissions.
3. If a user is granted Owner permission, and his group is denied Owner permission on the same resource, the user gets Owner permission. This is designed to ensure that an owner gets full access to his resources all the time irrespective of grants and revokes which he does to other users and groups. This behavior of Owner permission is different than other permissions, which are described in the section above.

## Denying vs. Removing Permissions

1. ‘Deny’ is the highest priority operation which supersedes any granted permissions. Hence this operation should be used very judiciously. This is a tool in the administrator’s box which gives him guarantee that the resource is really restricted from unintentional/accidental/unauthorized access by members of the group to which permission has been denied.
2. It is recommended that administrator should create separate groups for denying permissions on resources, and add all the desired members to these groups. The reason being it is not possible to allow a user to access the resource if his group is denied access.
3. If the intention is to prevent group members from accessing the resource, but allow a few selected members to access the same, it can be achieved by ‘removing’ the required permission from the group, and ‘granting’ the same to the selected users. E.g. if a group does not have ‘update’, users of this group will not be able to update the resource. However administrator can grant ‘update’ to a member of this group who will be able to update the resource.
4. The above case works for update, delete and owner permissions, however, since read permission is implicitly given to all users, removing ‘read’ does not work in restricting resources from being viewed. In this case the administrator can create separate groups of users who are to be denied read permission, and deny read to this group. All other users continue to use the read permission.

## Restricting unpublished resources

It may be desirable to restrict unpublished resources from being viewed by guests, or even registered repository users. The following sample cases illustrate how to deal with restricting the unpublished resources.

1. No one except the resource owner(s) and administrator(s) should view the unpublished resource
   1. Using permission management pages, deny read to ‘AllUsers’ group for the resource. This will restrict all users except the resource owner and administrator(s) from viewing the resource.
2. No one except the resource owner(s), publishers group and administrator(s) should view the resource.
   1. Deny read to Guest user
   2. Deny read to all users/groups except the publishers group. If this is a frequent requirement, or if the number of users is large, create a group ‘NonPublishers’ and add all users to this group except those in publishers group. Deny read to ‘NonPublishers’ group on the resource.

The administrators group and resource owners cannot be prevented from viewing the resource.

## Authorizing operations

Whenever a resource is requested for Read/Update/Delete, the operation will be authorized before the user can perform the operation. The process of authorization is as follows.

1. If the user requesting the permissions is an administrator he gets the requested permission.
2. If the user requesting the permission is guest and the permission requested is not ‘Read’ permission then he is denied the permission.
3. If the user himself is owner of the resource he gets the requested permission.
4. If none of the above is the case it is checked whether the user and his groups are granted or revoked the permission. If the user or any of his groups have been granted a permission and none of them have been revoked the permission, authorization succeeds, otherwise it fails.
5. In case of associations the user needs Update permission on the subject side; however it is sufficient to have Read permission on the object side. For example, for associating book with a chapter the user should have Update access on the book but Read access on the chapter object.
6. The permission set of a user remains the same on a resource whether the resource is accessed through the UI or any service available in the Zentity installation.

# Components

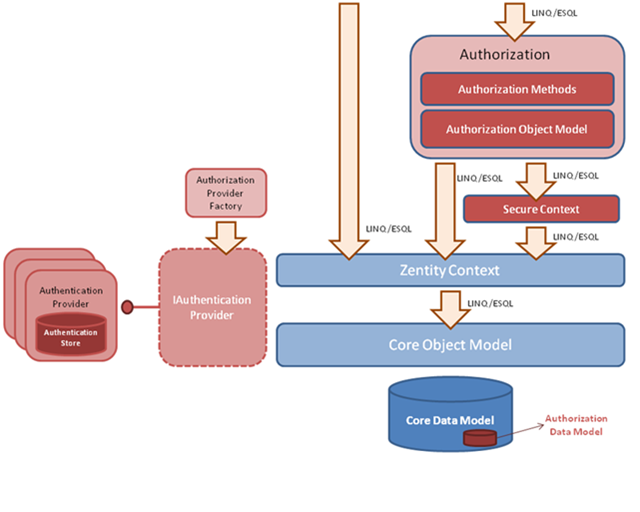
*[This section is designed for developers who want to build secure applications on top of Zentity Core and Zentity Security.]*

Zentity security module provides the following components.

1. Authentication contract which can be implemented to provide different types of authentication, like integrated windows authentication, windows live authentication, active directory authentication, http digest authentication etc.
2. Two authentication providers which implement the authentication contract.
   1. Zentity custom authentication provider
   2. HTTP digest authentication provider
3. User management API
4. Zentity services authentication which authenticates http requests by using the HTTP digest authentication provider
5. Zentity Core based security data model
6. Zentity Core and LINQ based authorization API
7. Security helper API - provides a layer on top of the user management and authorization APIs.
8. T-SQL authorization criteria which can be appended to T-SQL queries for obtaining resources with Read permission from the Core database.

# High Level Design Diagrams

The following diagrams describe the component relationships and interactions.



*Diagram – Authentication and Authorization API’s.*

Authorization Helper

Authorization

Permission Management

User Management

TSQL Authorization

Authorization API

Authentication API

Core Object Model

Core Data Model

Authorization Data Model

*Diagram – Authorization Helper API.*

# Application Configuration

*[This section is designed for developers who want to build secure applications on top of Zentity Core and Zentity Security.]*

Zentity Security provides the following configuration sections.

1. AuthenticationProviders – this section can be used to configure authentication providers for the application. The Zentity UI configures ZentityAuthenticationProvider for authentication, while Zentity services configure ZentityAuthenticationProvider for supporting basic authentication and HttpDigestAuthenticationProvider for supporting digest authentication.
2. PasswordPolicy – For configuring password policy this section needs to be added to the application configuration file. This section contains settings for password length, complexity and expiry.
   1. For enabling password policy an appSetting has been provided, which should be set to true.
3. Password encryption – for enabling password encryption there is a configuration setting which should be set to true. This setting value should be set only once when Zentity is installed. Changing this value afterwards will lead to errors in authenticating users.
4. Connection strings – the following sample configuration file also contains connection strings required for
   1. Using core + authorization data models
   2. Using core + authorization + ScholarlyWorks datamodels
   3. Making a SqlConnection to the core database.
   4. Connecting to the authentication database.

The following configuration file describes the above settings and sections.

<?xml version="1.0" encoding="utf-8" ?>

<configuration>

<configSections>

<!-- This section adds reference to the assemblies containing security configuration sections. -->

<section name="Authentication" type="Zentity.Security.Authentication.AuthenticationConfigurationSection,

Zentity.Security.Authentication"/>

<section name="PasswordPolicy" type="Zentity.Security.AuthenticationProvider.PasswordManagement.PasswordPolicy,

Zentity.Security.AuthenticationProvider"/>

</configSections>

<!--

Authentication configuration section

Add one provider entry for each authentication implementation.

name = Choose a unique name for authentication provider.

type = Name of the type implementing IAuthenticationProvider, Fully qualified assembly name

-->

<Authentication>

<Providers>

<add name="ZentityAuthenticationProvider" type="Zentity.Security.AuthenticationProvider.ZentityAuthenticationProvider,

Zentity.Security.AuthenticationProvider"/>

</Providers>

</Authentication>

<!--

Password policy configuration section. Please add only one policy entry in this section. Otherwise the behavior of the application cannot

be predicted.

MinimumLength = Desired minimum length of the password. Value must be >= 6

MaximumLength = Desired maximum length of the password. Value must be <= 100.

ExpiresInDays = Number of days after which a user must change his password. Value must be between 7 and 100.

StartWithAlphabet = Set to True if password must start with an alphabet.

MustContainDigit = Set to True if password must contain a digit.

MustContainSpecialCharacter = Set to True if password must contain a special character - ~`!@#$%^&\*()\_-+={[}]|\\:;\"'<,>.?/

-->

<PasswordPolicy>

<CurrentPolicy

MinimumLength="7"

MaximumLength="20"

ExpiresInDays="90"

StartWithAlphabet="true"

MustContainDigit="true"

MustContainSpecialCharacter="true"/>

</PasswordPolicy>

<connectionStrings>

<!--Connection string for the authentication and authorization databases-->

<!-- This connection string is required for accessing core and authorization data models. -->

<add name="ZentityContext" connectionString="metadata=res://Zentity.Security.Authorization;

provider=System.Data.SqlClient;

provider connection string=&quot;Data Source=.;

Initial Catalog=Zentity;

Integrated Security=True;

MultipleActiveResultSets=True&quot;"

providerName="System.Data.EntityClient"/>

<!-- This connection string is required for accessing core, authorization and scholarlyworks data models. -->

<add name="ScholarlyWorksConnection" connectionString="metadata=res://Zentity.ScholarlyWorksAndAuthorization;

provider=System.Data.SqlClient;

provider connection string=&quot;Data Source=.;

Initial Catalog= Zentity;

Integrated Security=True;

MultipleActiveResultSets=True&quot;"

providerName="System.Data.EntityClient"/>

<!-- This connection string is required for accessing the authentication database. -->

<add name="AuthenticationConnection" connectionString="Data Source=.;Initial Catalog=ZentityAuthentication; integrated security=true"/>

<!-- This connection string is required for opening a SqlConnection with the core database. (This database should also have authorization

data model installed. Zentity installer installs the core database with scholarlyworks and authorization data models.) -->

<add name="CoreSqlConnection" connectionString="Data Source=.;

Initial Catalog= Zentity;

Integrated Security=True;

MultipleActiveResultSets=True"/>

</connectionStrings>

<appSettings>

<!--

This configuration setting decides whether the password policy is applied. If this is set to true

the new passwords entered while changing password or creation of user account will be checked against the password policy.

-->

<add key="ApplyPasswordPolicy" value="False"/>

<!--

IMPORTANT It is recommended that this setting should be set to true from the point of view of ensuring secure storage of passwords.

This configuration setting can be changed only once right after Zentity is installed, before creating any

user accounts. If this is changed afterwards it will lead to errors in authenticating users.

Also the built in users - administrator and guest have their passwords stored in encrypted form. You need to set them in clear text in

the authentication database, if this setting is made false.

-->

<add key="EncryptPassword" value="True"/>

<add key="encryptionKey" value="877D3C2437D7B9DF4F31E9AE6CA97C3F8AE99B02BF66D137F865A8651C1A36D2"/>

</appSettings>

</configuration>

# Code samples

*[This section is designed for developers who want to build secure applications on top of Zentity Core and Zentity Security.]*

#### Security API usage

This sample below shows how to use create users and groups.

* Open Microsoft Visual Studio,
* Create new console application,
* Add references to
  + System.Data.Entity
  + System.IdentityModel
  + Zentity.Core
  + Zentity.Security.Authentication
  + Zentity.Security.AuthenticationProvider
  + Zentity.Security.Authorization
  + Zentity.Security.AuthorizationHelper
* Add an application configuration file similar to the one given [here](#SecurityConfig).
* Update Program.cs as shown below
* Replace administrator password with the correct one for your installation.

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using Zentity.Security.AuthorizationHelper;  using Zentity.Core;  using Zentity.Security.AuthenticationProvider;  using Zentity.Security.Authentication;  using System.IdentityModel.Tokens;  using Zentity.Security.Authorization;  namespace SecuritySamples  {  class Program  {  static void Main(string[] args)  {  //An administrator can create new users. Get an instance of authentication provider and login as administrator.  //The 'ZentityAuthenticationProvider' is configured in the application configuration file. Please refer to the sample  //application configuration file provided in the guide.  IAuthenticationProvider provider = AuthenticationProviderFactory.CreateAuthenticationProvider("ZentityAuthenticationProvider");  AuthenticatedToken adminToken = provider.Authenticate(new UserNameSecurityToken("Administrator", "XXXX")); //Supply correct password    //Create a ZentityUser instance and set its properties.  ZentityUser user = new ZentityUser  {  LogOnName = "John", //mandatory  FirstName = "John", //mandatory  LastName = "King",  City = "New York",  Country = "USA",  Email = "john@abc.com" //mandatory  };  user.SetPassword("john@123");//mandatory  user.SetSecurityQuestion("??");//mandatory  user.SetAnswer("\*\*");//mandatory  //You need an instance of ZentityContext for creating the user in authorization (core) database.  using (ZentityContext context = new ZentityContext())  {  if (UserManager.CreateUser(user, adminToken))  {  Console.WriteLine("User John created");  }  else  {  Console.WriteLine("Errors while creating user.");  }  //Create a group  Group users = new Group { GroupName = "Users", Title = "Users", Description = "This is a users group." };  if (UserManager.CreateGroup(users, adminToken))  {  Console.WriteLine("Group created.");  }  else  {  Console.WriteLine("Errors while creating group.");  }  //Add John to users group.  Identity john = UserManager.GetIdentity("John", context);  if (UserManager.AddIdentityToGroup(john, users, adminToken))  {  Console.WriteLine("John added to users group.");  }  else  {  Console.WriteLine("Errors while adding John to users group.");  }  }  }  }  } |

This sample below shows how to grant permissions and authorize operations.

* Open Microsoft Visual Studio 2008,
* Create new console application,
* Add references to
  + System.Data.Entity
  + System.IdentityModel
  + Zentity.Core
  + Zentity.Security.Authentication
  + Zentity.Security.AuthenticationProvider
  + Zentity.Security.Authorization
  + Zentity.Security.AuthorizationHelper
* Add an application configuration file similar to the one given [here](#SecurityConfig).
* Update Program.cs as shown below
* Replace administrator password with the correct one for your installation. It appears in two places in this sample.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Zentity.Security.AuthorizationHelper;

using Zentity.Core;

using Zentity.Security.AuthenticationProvider;

using Zentity.Security.Authentication;

using System.IdentityModel.Tokens;

using Zentity.Security.Authorization;

namespace SecuritySamples

{

class Program

{

static void Main(string[] args)

{

CreateUsersAndGroups();

//Login as Admin

IAuthenticationProvider provider = AuthenticationProviderFactory.CreateAuthenticationProvider("ZentityAuthenticationProvider");

AuthenticatedToken adminToken = provider.Authenticate(new UserNameSecurityToken("Administrator", "XXXX"));//Supply correct password

//Grant create access to Mary

using (ZentityContext context = new ZentityContext())

{

context.MetadataWorkspace.LoadFromAssembly(typeof(Identity).Assembly);

Identity Mary = UserManager.GetIdentity("Mary", context);

if (Mary.GrantCreate(adminToken, context))

{

Console.WriteLine("Create access granted to Mary.");

}

//Security APIs do not save context changes. Hence the following call is necessary after calling grant/revoke/remove.

context.SaveChanges();

}

//Create resources and grant permissions on them.

//Login as Mary

AuthenticatedToken marysToken = provider.Authenticate(new UserNameSecurityToken("Mary", "Mary@123"));

using (ZentityContext context = new ZentityContext())

{

context.MetadataWorkspace.LoadFromAssembly(typeof(Identity).Assembly);

//Before allowing Mary to create new resources, the application should check whether Mary has create access.

if (marysToken.HasCreatePermission(context))

{

Resource r = new Resource { Title = "Resource 1", Description = "Mary created this resource" };

context.AddToResources(r);

context.SaveChanges();

//Make Mary owner of the resource. This is an explicit step which the application needs to take when

//it allows a user to create a resource.

r.GrantDefaultPermissions(context, marysToken);

context.SaveChanges();

//Now Mary wants to grant update on this resource to his collegue Patrick.

//Since he is owner of the resource he can grant / revoke / remove permissions of other users on his resource.

Identity patrick = UserManager.GetIdentity("Patrick", context);

r.Grant("Update", patrick, context, marysToken);

//Security APIs do not save context changes. Hence the following call is necessary after calling grant/revoke/remove.

context.SaveChanges();

}

}

//Now Patrick wants to update the resource on which Mary has granted him update.

//Login as Patrick.

AuthenticatedToken patricksToken = provider.Authenticate(new UserNameSecurityToken("Patrick", "patrick@123"));

using (ZentityContext context = new ZentityContext())

{

context.MetadataWorkspace.LoadFromAssembly(typeof(Identity).Assembly);

Resource r = context.Resources.Where(res => res.Title.Equals("Resource 1")).FirstOrDefault();

if (r != null)

{

//Check whether the logged on user has update access.

if (r.Authorize("Update", context, patricksToken))

{

Console.WriteLine("Patrick has update access on Resource 1.");

}

else

{

Console.WriteLine("Patrick does not have update access on Resource 1.");

}

}

}

}

private static void CreateUsersAndGroups()

{

//An administrator can create new users. Get an instance of authentication provider and login as administrator.

//The 'ZentityAuthenticationProvider' is configured in the application configuration file. Please refer to the sample

//application configuration file provided in the guide.

IAuthenticationProvider provider = AuthenticationProviderFactory.CreateAuthenticationProvider("ZentityAuthenticationProvider");

AuthenticatedToken adminToken = provider.Authenticate(new UserNameSecurityToken("Administrator", "XXXX")); //Supply correct password

//Create a ZentityUser instance and set its properties.

ZentityUser mary = new ZentityUser

{

LogOnName = "Mary", //mandatory

FirstName = "Mary", //mandatory

LastName = "King",

City = "New York",

Country = "USA",

Email = "Mary@abc.com" //mandatory

};

mary.SetPassword("Mary@123");//mandatory

mary.SetSecurityQuestion("??");//mandatory

mary.SetAnswer("\*\*");//mandatory

//Create a ZentityUser instance and set its properties.

ZentityUser patrick = new ZentityUser

{

LogOnName = "Patrick", //mandatory

FirstName = "Patrick", //mandatory

LastName = "Smith",

City = "New York",

Country = "USA",

Email = "patrick@abc.com" //mandatory

};

patrick.SetPassword("patrick@123");//mandatory

patrick.SetSecurityQuestion("??");//mandatory

patrick.SetAnswer("\*\*");//mandatory

//You need an instance of ZentityContext for creating the user in authorization (core) database.

using (ZentityContext context = new ZentityContext())

{

context.MetadataWorkspace.LoadFromAssembly(typeof(Identity).Assembly);

if (UserManager.CreateUser(mary, adminToken))

{

Console.WriteLine("User Mary created");

}

else

{

Console.WriteLine("Errors while creating user.");

}

//Create a group

Group users = new Group { GroupName = "Group1", Title = "Group1", Description = "This is a users group." };

if (UserManager.CreateGroup(users, adminToken))

{

Console.WriteLine("Group created.");

}

else

{

Console.WriteLine("Errors while creating group.");

}

//Add Mary to users group.

Identity user = UserManager.GetIdentity("Mary", context);

if (UserManager.AddIdentityToGroup(user, users, adminToken))

{

Console.WriteLine("Mary added to Group1.");

}

else

{

Console.WriteLine("Errors while adding Mary to Group1.");

}

//Create user Patrick

if (UserManager.CreateUser(patrick, adminToken))

{

Console.WriteLine("User Patrick created");

}

else

{

Console.WriteLine("Errors while creating user.");

}

}

}

}

}

# Known issues

*[This section is designed for developers who want to build secure applications on top of Zentity Core and Zentity Security.]*

1. Authorize extension methods from the authorization helper API does not work with parameterized object queries.
   1. Workaround – Instead of adding parameters embed parameter value in the e-sql query text, by using string.Format().

The following code samples demonstrate the issue and its workaround.

//The following piece of code gives exception.

using (ZentityContext context = new ZentityContext())

{

    string esql = @"SELECT VALUE TOP(@cnt) Res FROM ZentityContext.Resources as Res WHERE Res.Title LIKE 'My%'

ORDER BY Res.Title";

    ObjectQuery<Resource> resources = new ObjectQuery<Resource>(esql, context);

    ObjectParameter p = new ObjectParameter("cnt", 2);

    resources.Parameters.Add(p);

    AuthenticatedToken userToken = LogIn("user", "password");

    ObjectQuery<Resource> authorizedResources = resources.Authorize("Read", context, userToken);

    //Enumerating through authorizedResources fails with exception 'A parameter named 'cnt' already exists...'

}

//The workaround is as follows - the following code works fine.

using (ZentityContext context = new ZentityContext())

{

    string staticEsql = @"SELECT VALUE TOP({0}) Res FROM ZentityContext.Resources as Res WHERE Res.Title LIKE 'My%'

    ORDER BY Res.Title";

    int cnt = 2; //get the result set size at runtime.

    string runtimeQuery = string.Format(staticEsql, cnt);

    ObjectQuery<Resource> resources = new ObjectQuery<Resource>(runtimeQuery, context);

    AuthenticatedToken userToken = LogIn("user", "password");

    ObjectQuery<Resource> authorizedResources = resources.Authorize("Read", context, userToken);

//Enumerating through authorizedResources works fine...'

}

# Glossary

## Core

Core is the ADO.Net entity framework based data model on top of the Zentity database which stores the resources metadata, binary content, associations between the resources and categorization information of the resources

## Authentication

Authentication is the act of establishing or confirming something (or someone) as *authentic*, i.e., that claims made by or about the thing are true. This might involve confirming the identity of a person, the origins of an artifact, or assuring that a computer program is a trusted one

With respect to software, authentication is the process of obtaining identification credentials from a user and validating those credentials against some authority. If the credentials are valid, the entity that submitted the credentials is considered an authenticated identity. The credentials may be username and password, or a token or a certificate.

## Authorization

Authorization is the concept of allowing access to resources only to those permitted to use them. Authorization process determines whether an identity should be granted to perform a specific operation on a specific resource

# References

<http://www.microsoft.com/technet/prodtechnol/sql/2005/multisec.mspx> (Implementing Row- and Cell-Level Security in Classified Databases Using SQL Server 2005)

[http://msdn.microsoft.com/en-us/library/aa302417.aspx#c01618429\_002](http://msdn.microsoft.com/en-us/library/aa302417.aspx" \l "c01618429_002) (Web Application Security Fundamentals)

<http://msdn.microsoft.com/en-us/library/yh26yfzy.aspx> (Introduction to Membership)