**Custom Validation and Role Authorization**

* **principal** object encapsulates both an identity object and a role.

The principal object represents the security context under which code is running. Applications that implement role-based security grant rights based on the role associated with a principal object. Similar to identity objects, the .NET Framework provides a [GenericPrincipal](https://msdn.microsoft.com/en-us/library/system.security.principal.genericprincipal(v=vs.110).aspx) object and a [WindowsPrincipal](https://msdn.microsoft.com/en-us/library/system.security.principal.windowsprincipal(v=vs.110).aspx) object. You can also define your own custom principal classes.

**Custom Principal**

Another option is to supply your custom IPrincipal implementation to WCF. This gives you the chance to implicitly run code after the authentication stage of each request. For this you have to create your own custom principal and return it to the WCF plumbing. The custom principal will then be available from Thread.Cur­rentPrincipal to the service code. Custom principals allow full customization of role-based security and expose specialized security logic for the service developer to use.

* **The identity** object encapsulates information about the user or entity being validated. At their most basic level, identity objects contain a name and an authentication type. The [IIdentity](https://msdn.microsoft.com/en-us/library/system.security.principal.iidentity(v=vs.110).aspx) interface defines properties for accessing a name and an authentication type, such as Kerberos V5 or NTLM.
* **ClaimSet Class** -- Represents the collection of claims that are associated with an entity. A claim is a piece of information that can be associated with an entity in your system. This is most commonly a user but could also be a service or some resource. A claim consists of three pieces of information: a claim type, the claim content, and whether the claim describes the identity of the subject or a capability of the subject. This data structure is represented by a class known as System.IdentityModel.Claims.Claim. Furthermore, Claim is a DataContract, which makes it serialization-friendly (important when you plan to transmit claims over service boundaries):
* **evaluation context**, which basically represents the authorization context while it is still in the process of being built.
* **service authorization manager** is a class that derives from System.ServiceModel.ServiceAuthorizationManager. You can override the CheckAccessCore method to run custom authorization code for each request. In CheckAccess, you have access to the current security context as well as the incoming message, including the headers. When you return false from CheckAccess, WCF will create the Access Denied fault message and send that back to the client. A return value of true will grant access to the service operation.

The ServiceAuthorizationManager is part of the WCF Identity Model infrastructure. The Identity Model enables you to create custom authorization policies and custom authorization schemes

We tell the WCF framework that we are going to use my CustomValidator.cs class to validate incoming message to our service. My CustomValidator.cs inherits from UserNamePasswordValidator class to specify how a username and password is validated. This can be done by deriving a class from UserNamePasswordValidator and override the [Validate](https://msdn.microsoft.com/en-us/library/system.identitymodel.selectors.usernamepasswordvalidator.validate(v=vs.110).aspx) method.

By setting config values below we are telling the service to validate/authenticate and authorize each incoming message to the service using UserName authentication policy and redirect to My classes to check authentication and Authorization/Roles against my database.

The soap message contains the user name and password that is passed from the client to the service.

* 1 My CustomValidator.Validate override method from UserNamePasswordValidator class is executed. Here we check if the passed in credentials match what I have in the database. If they don't then a fault exception is thrown.
* Next My Custom AuthorizationPolicy.cs class is called. It inherits IAuthorizationPolicy and implements Evaluate(), getter System.IdentityModel.Claims.ClaimSet Issuer and getter Id.

My class uses custom authentication, it creates a class that derives from **IAuthorizationPolicy**. In this class, I retrieve the principal from the cache that was created by the store based on the username, so that WCF can authorize the user. After I get the principal, I assign it to **EvaluationContext.Properties[“principal”]** and the identity to the **EvaluationContext.Properties["Identities"]**

In order to integrate a custom Principal with WCF, we need to set the “PrincipalPermissionMode” property to “Custom”. In addition, we also need to set the authorization policy that will be used to create the custom principal objet and supply it to WCF plumbing.

I added

customUserNamePasswordValidatorType="ToothPickService.CustomValidator,ToothPickService" />

**<userNameAuthentication userNamePasswordValidationMode="Custom"**

customUserNamePasswordValidatorType="ToothPickService.CustomValidator,ToothPickService"

<authorizationPolicies>

<add policyType="ToothPickService.AuthorizationPolicy,ToothPickService"/>

</authorizationPolicies>