

Guide to Microsoft Enterprise Library

Solutions for Enterprise Development

Sateesh Arveti



Guide to Microsoft Enterprise Library:

Solutions for Enterprise Development

This free book is provided by courtesy of [C# Corner](#) and Mindcracker Network and its authors. Feel free to share this book with your friends and co-workers.

Please do not reproduce, republish, edit or copy this book.



Sateesh Arveti
Software Developer

Sam Hobbs
Editor, C# Corner

This book is a basic introduction to **Microsoft Enterprise Library** basically for beginners who want to learn complete basic with example of **Microsoft Enterprise Library**.

Table of Contents

1. Introduction of Enterprise Library

- 1.1 What is Enterprise Library
- 1.2 Benefits of Enterprise Library
- 1.3 Types of application Blocks
 - 1.3.1 Caching Application Block
 - 1.3.2 Cryptography Application Block
 - 1.3.3 Data Access Application Block
 - 1.3.4 Logging Application Block
 - 1.3.5 Policy Injection Application Block
 - 1.3.6 Security Application Block
 - 1.3.7 Unity Application Block
 - 1.3.8 Validation Application Block(VAB)

1.4 History of Enterprise Library

2. Validation Application Block

3. Attributes Based Validators

- 3.1 Property Comparison Validator
- 3.2 Regex Validators
- 3.3 Validator Composition

4. Enterprise Library Configuration Tool

5. Implementation of Validators using Custom Code

6. Adaptors provided by the Validation Application Block

7. Windows Forms Validation using VAB Adaptor

8. VAB Adaptor for WCF

9. Implementation of Custom Validators using VAB

10. Integration of VAB using Policy Injection Application Block(PIAB)

1 Introduction of Enterprise Library

Most of us might have heard about Enterprise library. We even used in our code also. Now I am going to explain about Enterprise library and its application blocks in brief. We will start with an introduction to Enterprise library and benefits of using it. In this series, I am going to use Enterprise Library 4.1. It's a free download and available at [here](#). Microsoft Patterns & Practices team will be taking care of the Enterprise library design.

1.1 What is Enterprise Library?

Enterprise library is a set of application blocks and services that will make our coding work easier. Application blocks are nothing but reusable software components, which can be plug-in into the projects for solving commonly occurring problems. It provides a high level of abstraction over existing class libraries.

1.2 Benefits of using Enterprise Library:

1. Consistency: By using it, we can make sure that code written for a particular task like caching or exception handling will be consistent across all developers and projects.
2. Time Saving: By using it, we can handle most of the common tasks of a project with minimal lines of code and configuration settings, which will save time in turn.
3. Reduced Errors: We can reduce errors count by using it.
4. Extensibility: We can extend any of the application blocks as per our requirements.
5. Easy Integration: All application blocks are based on plug-in pattern. So, we can easily integrate it into our code.
6. Additional Functionality: By using it, sometimes we can get extra functionality which is not provided by base libraries.

Based on the functionality provided, application blocks are classified as:

1.3 Types of Application Blocks:

1. Caching Application block: This can be used to handle tasks related to caching of data. It provides all functionality required for storing and retrieving data from cache either from in-memory or database.
2. Cryptography Application block: This can be used to incorporate security for the data with cryptographic algorithms and classes. It frees developers from understanding complex classes for implementing security.
3. Data Access Application block: This will provide classes based on ADO.NET for accessing external data easily with minimal lines of code. It is one of the blocks, which got high popularity.
4. Exception Handling Application block: It provides a consistent way of handling exceptions in all the layers of our code.
5. Logging Application block: This will help to log all kinds of messages easily either in the event log or external database. It provides a single interface by which we can log messages in any storage media without changing code.

6. Policy Injection Application block: It provides a mechanism for applying policies to objects. We will define a set of policies for the target classes and their members through configuration of the Policy Injection Application Block or by applying attributes to individual members of the target class.
7. Security Application block: It helps to implement authorization and authentication mechanisms into the system easily.
8. Unity Application block: It provides constructor, property, and method call injection.
9. Validation Application block: It provides simple and easy way of implementing validations.

Based on our requirement, we can use that application block only. Dependencies among application blocks got reduced a lot in the release of 3.0 and 4.0 versions.

Configuration block is the base for customizing any of the above application blocks.

1.4 History of Enterprise Library:

Now, we will have a small look into the past versions of Enterprise Library and its contents:

- First release of Enterprise Library is in 2005 targeting 1.1 Framework. This has been deprecated.
- Second release (version 2.0) was in 2006 targeting both 1.1 & 2.0 Frameworks.

This release is having following application blocks:

1. Caching Application Block
 2. Cryptography Application Block
 3. Data Access Application Block
 4. Exception Handling Application Block
 5. Logging Application Block
 6. Security Application Block
- Third release (3.1) was in 2007 targeting 2.0 and 3.0 frameworks. It had enhancements to existing blocks and additional ones apart from above.

Additional blocks in this release are:

1. Policy Injection Application Block
 2. Validation Application Block
- The fourth release (4.0) was in 2008. It is targeting 3.5 framework. It has Unity Application Block as a new entry.
 - Fifth and latest release was in Oct 2008. It is targeting 3.5 framework. It does not have any new blocks. But having enhancements to existing blocks and performance improvements.

2 Validation Application Block

After explaining Enterprise Library and its components. Now we will start our journey with a Validation Application block (VAB). We will go through a series of samples that make use of this block. Validation is a very common activity in any kind of application. Before looking into this block, the following explains why validation is necessary:

1. Store Correct information: By validation, we can make sure the data entered is proper. For example, an account balance should be numeric only.
2. Security: By validation, we can protect our code from threats, hacking techniques and SQL injection etc.
3. System crash: By validation, we can make sure our code won't crash due to invalid data like storing non-numeric data in a numeric variable, etc.

Because of the above reasons, validation became a quite an essential task in most of the projects.

Now, let's see the benefits of using VAB; they are:

1. Less Code: By using VAB, the amount of code needed for validations get reduced.
2. Code Readability: It's easy to understand the code by using VAB.
3. Easy to maintain: It's easy to modify code without side-effects.
4. Centralization of code: By using VAB, we can place validation logic in a single place for easy maintenance.

VAB is based on a set of classes called validators. These validators provide the basic validation functionality like null checking, string length checking etc. By combining these built-in validators and using and/or conditions, we can create complex validators.

We can combine validators by using Rule sets. For the time being, assume a rule set combines two or more validators and make sure that all validations are carried out.

By using VAB, we can create validators in three ways:

1. using Attributes
2. using Configuration
3. using Code.

VAB comes with a set of adapters for working with the following technologies:

1. ASP.NET
2. Windows Forms and
3. WCF

Let's start with an example showing an Attribute-based validation implementation. Open VS 2008 with Enterprise Library 4.1 installed.

Create a new console application and name it AttrBasedVAB. Add a reference to the following assemblies present in the installation folder of Enterprise Library:

Microsoft.Practices.EnterpriseLibrary.Common.dll
Microsoft.Practices.EnterpriseLibrary.Validation.dll

And import the following namespaces:

Microsoft.Practices.EnterpriseLibrary.Validation.Validators
Microsoft.Practices.EnterpriseLibrary.Validation

Create an employee class in program.cs as shown below:

```
class Program
{
    static void Main(string[] args)
    {
    }
}
class Employee
{
    public int EmpID { get; set; }
    public string EmpName {get; set;}
    public double Salary {get; set;}
}
```

Now, we will add some validations like empid should be less than 2000; empname should not be more than 10 characters by using VAB attributes as shown below:

```
class Employee
{
    [RangeValidator(0, RangeBoundaryType.Inclusive, 2000, RangeBoundaryType.Inclusive, Ruleset="Group1", MessageTemplate="ID should be below 2000")]
    public int EmpID { get; set; }
    [StringLengthValidator(10, Ruleset = "Group1", MessageTemplate="Name should be less than 10 characters")]
    public string EmpName {get; set;}
    public double Salary {get; set;}
}
```

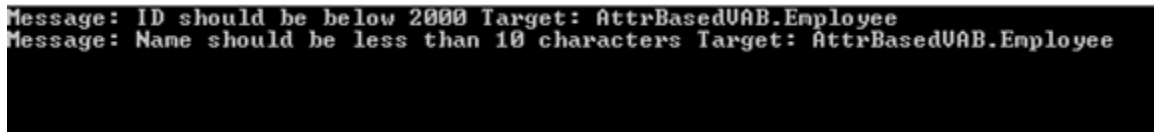
Here, rangevalidator validates empid between 0-2000 inclusive and stringlength validator validates empname should be less than 10 characters. By default, all validators will have a default error message. The MessageTemplate property is used to override default error message and set a custom error message.

Now, go to the Main method and add the following code:

```
static void Main(string[] args)
{
    Employee objEmp = new Employee();
    objEmp.EmpID = 3000;
    objEmp.EmpName = "CSharp Corner";
    objEmp.Salary = 3000;
    //Create a Validator instance using ValidationFactory.
    Validator empValidator = ValidationFactory.CreateValidator<Employee>("Group1");
    ValidationResult results = empValidator.Validate(objEmp);
    if (!results.IsValid)
    {
        //Loop through each validation result and display it on console.
        foreach (ValidationResult result in results)
        {
            Console.WriteLine("Message: " + result.Message + " Target: " + result.Target);
        }
    }
    Console.ReadLine();
}
```

Here, we are creating an Employee and Validator instance and validating an objEmp object using the validator's Validate Method. This method will return a list of validation messages. We are looping through this validation results and displaying on the console.

Run the application, we will see the validation messages on the console as shown below:



```
Message: ID should be below 2000 Target: AttrBasedVAB.Employee
Message: Name should be less than 10 characters Target: AttrBasedVAB.Employee
```

In this way, we can implement an attribute-based validation using VAB. My suggestion is to use attribute-based validation, if it is a new project starting from scratch. Since, it's hard to implement this validation in an existing code base, because it requires adding this kind of validation in many places. The main benefit of attribute-based validation is that modifying the validation logic is easy, since it is present in a single place.

I experienced the real benefit of this kind of validation in my application. In my project, Job Number should not be more than 7 characters. The current requirement is that it should not be more than 9 characters. I saw they are doing this 7 character validation in many places. Finally, I had gone through all modules and modified it to 9 characters. If I had used this attribute validation, then I need to change only in one place.

3 Attributes Based Validators

Now, we look into some more attribute validators and end up with rule sets. We will discuss about PropertyComparisonValidator, RegexValidator and ValidatorComposition. Add new properties to Employee class as shown below:


```
class Employee
{
    [RangeValidator(0, RangeBoundaryType.Inclusive, 2000, RangeBoundaryType.Inclusive, Ruleset="Group1",
        MessageTemplate="ID should be below 2000")]
    public int EmpID { get; set; }

    [StringLengthValidator(10, Ruleset = "Group1", MessageTemplate="Name should be less than 10 characters")]
    public string EmpName { get; set; }

    public double Salary { get; set; }

    //Email should be in Valid Format.
    [RegexValidator(@"^[a-zA-Z]([w\.-])*[a-zA-Z0-9]@[a-zA-Z0-9]([w\.-])*[a-zA-Z0-9]\.[a-zA-Z]([a-zA-Z\.-])*[a-zA-Z]$",
        Ruleset="Group1", MessageTemplate="Not a Valid E-Mail.")]
    public string EmpMailID { get; set; }

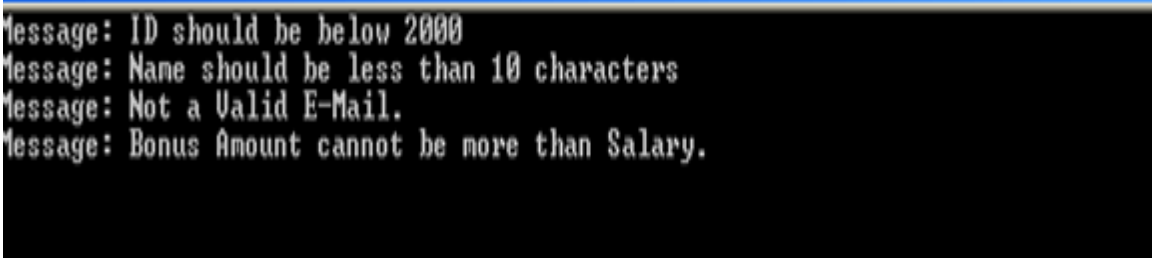
    //Bonus should be less than Salary.
    [PropertyComparisonValidator("Salary", ComparisonOperator.LessThan, Ruleset="Group1",
        MessageTemplate="Bonus Amount cannot be more than Salary.")]
    public double Bonus { get; set; }
}
```

We are using a PropertyComparison validator to compare two properties of a class. Here, I assumed a condition, that bonus should be less than salary. So, I used this validator and compared it with salary property as shown above. Regular expression validator is used to validate data against a pattern like email, phone number etc. Just give the pattern for which the property's format should satisfy.

Now, add the code to Main method to validate these new conditions as shown below:

```
static void Main(string[] args)
{
    Employee objEmp = new Employee();
    objEmp.EmpID = 3000;
    objEmp.EmpName = "CSharp Corner";
    objEmp.Salary = 3000;
    objEmp.Bonus = 4000;
    objEmp.EmpMailID = "abc@domain";
    //Create a Validator instance using ValidationFactory.
    Validator empValidator = ValidationFactory.CreateValidator<Employee>("Group1");
    ValidationResult results = empValidator.Validate(objEmp);
    if (!results.IsValid)
    {
        //Loop through each validation result and display it on console.
        foreach (ValidationResult result in results)
        {
            Console.WriteLine("Message: " + result.Message);
        }
    }
    Console.ReadLine();
}
```

Run the application, the output will be like this:



```
message: ID should be below 2000
message: Name should be less than 10 characters
message: Not a Valid E-Mail.
message: Bonus Amount cannot be more than Salary.
```

There are lot more validators present in the VAB. I will list all those for future reference.

Not Null - NotNullValidator	The value must not be NULL
Contains Characters - ContainsCharactersValidator	For checking, whether a string contains specified characters or not like special characters
Regular Expression - RegexValidator	For checking, Format of the data
Range - RangeValidator	For checking, data's valid values like age between 0 - 100
Relative DateTime - RelativeDateTimeValidator	For comparing difference between a specific date and today's date.
String Length - StringLengthValidator	For checking character count in a string
Domain - DomainValidator	For checking the domain for a value like IsManager property should have YES or NO only.
Enum Comparison - EnumConversionValidator	For checking, Conversion of a string to Enum type is possible or not
Type Conversion - TypeConversionValidator	For checking, Conversion of a string to a specified type is possible or not
Property Comparision - PropertyComparisonValidator	For comparing two properties of a class
Negated	This attribute is applicable to above all validators for negating the condition like Value should be NULL by making Negated= true for Not Null validator

ValidatorComposition is used to combine two or more validators. When we add two or more validators for a property; by default, both validators should pass [Since, it uses internally logical AND on the validators]. We can change the behavior by using ValidatorComposition to OR. Add below validator to empname as shown below:

```
[ValidatorComposition(CompositionType.Or, Ruleset="Group1")]
[StringLengthValidator(10, Ruleset = "Group1", MessageTemplate="Name should be less than 10 characters")]
[DomainValidator(new object[] { "E1", "E2", "E3" }, MessageTemplate = "Not a valid Employee", Ruleset = "Group1")]
public string EmpName {get; set;}
```

The above validator will fail only, if both empname more than 10 characters and empname not in E1, E2 and E3 validators fails. So, we can use ValidatorComposition to aggregate validators using either logical AND or OR.

Few restrictions on using Attribute based Validators:

1. You should have to source code to implement these validators.
2. Need to recompile the code, whenever we change the validator's properties.

Finally, we look into Rule sets and its use. By using rule sets, we can group validators and make it to work as a whole based on a specific environment. For example, in Employee class, by setting rule set = "TEST" we can run only those validators having rule set as TEST. We can do this assigning it to TEST rule set and passing that rule set name to CreateValidator method.

4 Enterprise Library Configuration Tool

Now, we look into validation through config files. We will discuss about Enterprise Library Configuration tool followed by an example. Create a new Console Application in VS 2008 and name it as ConfigBasedVABClient.

Add new properties to Employee class and add it to a new class Library ConfigBasedVABServer as shown below:

```
class Employee
{
    public int EmpID { get; set; }

    public string EmpName { get; set; }

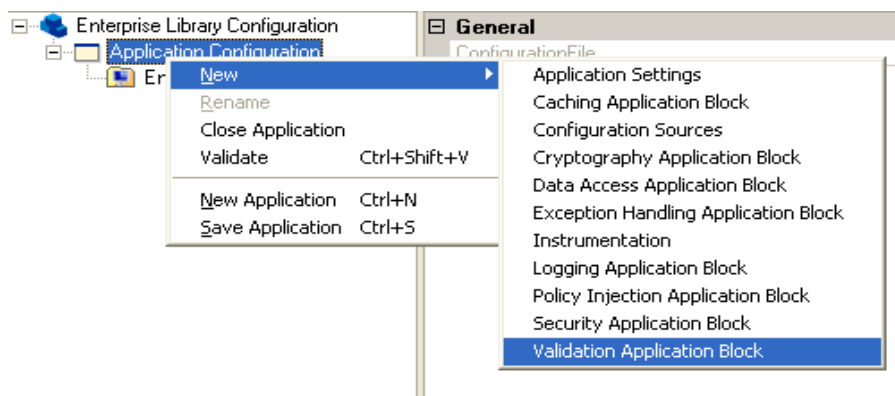
    public double Salary { get; set; }

    public string EmpMailID { get; set; }

    public double Bonus { get; set; }
}
```

Build the application. Go to Enterprise Library Configuration tool present in installation folder or Start → Programs → Microsoft patterns & practices → Enterprise Library 4.1 - October 2008.

Click on New Application and select Validation Application Block as shown below:



1. Select New → Type → Load From File → Select ConfigBasedVABServer dll path → Select Employee → Ok.
2. Select New Rule Set → Name as Group1
3. Right Click Group1 → New → Choose Field → Check EmpID.
4. Right Click EmpID → New Range Validator → Set required properties as shown below:

Enterprise Library Configuration	
Application Configuration	
Environments	
Validation Application Block	
Employee	
Group1	
EmpID	
Range Validator	
Self	

[Name]	
Name	Range Validator
General	
LowerBound	0
LowerBoundType	Ignore
Negated	False
Tag	
UpperBound	2000
UpperBoundType	Inclusive
Message	
MessageTemplate	Emp ID should be below 2000
MessageTemplateResourceName	
MessageTemplateResourceTypeName	

5. Now save it as ConfigurationVAB.

Similarly, we can add other validators on Employee properties.

Go to ConfigBasedVABClient and add reference to ConfigBasedVABServer and required VAB dlls .

Add app.config to the client project and paste the contents of ConfigurationVAB.config into it as shown below:

```

App.config Program.cs Start Page
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="validation" type="Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.Enter
  </configSections>
  <validation>
    <type assemblyName="ConfigBasedVABServer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      name="ConfigBasedVABServer.Employee">
      <ruleset name="Group1">
        <properties>
          <property name="EmpID">
            <validator lowerBound="0" lowerBoundType="Ignore" upperBound="2000"
              upperBoundType="Inclusive" negated="false" messageTemplate="Emp ID should be below 2000"
              messageTemplateResourceName="" messageTemplateResourceType=""
              tag="" type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.RangeValidator, Microsoft.Practices.EnterpriseLibrary.Va
              name="Range Validator" />
          </property>
        </properties>
      </ruleset>
    </type>
  </validation>
</configuration>

```

Add the below code to Main method:

```
class Program
{
    static void Main(string[] args)
    {
        Employee objEmp = new Employee();
        objEmp.EmpID = 3000;
        Validator empvalidator = ValidationFactory.CreateValidator<Employee>("Group1");
        ValidationResult results = empvalidator.Validate(objEmp);
        foreach (ValidationResult result in results)
        {
            Console.WriteLine(result.Message);
        }
        Console.ReadLine();
    }
}
```

Run the application, the output will be like this:



```
Emp ID should be below 2000
```

In this way, we can add our validators to existing assemblies using config files without the need of making changes to existing code.

Benefits of using Config based Validators:

1. No need of source code for implementing the validation.
2. No need to recompile after making changes to validation logic in config files.
3. No need to change existing code.

5 Implementation of Validators using Custom Code

Here, we will look into validator's implementation using custom Code and end up with Message templates. Sometimes, the built-in validators might not serve our purpose. In those cases, we can create our own validating methods.

Open your Visual Studio 2008 and create a new console application, name it as CodeBasedVAB. Add required VAB dlls. Now, Go to program.cs and add this below class definition:

```
[HasSelfValidation]
class Employee
{
    public int EmpID { get; set; }
    public string EmpName { get; set; }
    public double Salary { get; set; }

    //Validator for checking -ve Salary
    [SelfValidation(Ruleset = "Group1")]
    public void CheckSalary(ValidationResults results)
    {
        if (Salary < 0)
        {
            results.AddResult(new ValidationResult("Salary cannot be Negative.", this, null, null, null));
        }
    }

    //Validator for checking -ve Salary
    [SelfValidation(Ruleset = "Group1")]
    public void isValidEmpID(ValidationResults results)
    {
        List<int> validEmpIds = new List<int> { 100,200,300,400};
        if (!validEmpIds.Contains(EmpID))
        {
            results.AddResult(new ValidationResult("Not a Valid Emp ID.", this, null, null, null));
        }
    }
}
```

Validation through custom code is called as self-validation. We need to decorate the class having self-validation with [HasSelfValidation]. We had defined validation methods for checking negative salary and valid emp Ids with a Self Validation attribute as shown above.

Now, we will validate this Employee object in Main method as shown below:

```
static void Main(string[] args)
{
    Employee objEmp = new Employee();
    objEmp.EmpID = 201;
    objEmp.EmpName = "Testing";
    objEmp.Salary = -2000;
    Validator empvalidator = ValidationFactory.CreateValidator<Employee>("Group1");
    ValidationResult results = empvalidator.Validate(objEmp);
    if (!results.IsValid)
    {
        foreach (ValidationResult res in results)
        {
            Console.WriteLine(res.Message);
        }
    }
    Console.ReadLine();
}
```

Run the application, the output will be like this:



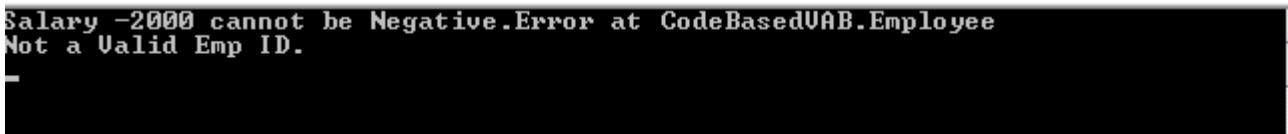
```
Salary cannot be Negative.
Not a Valid Emp ID.
```

Now, we look into MessageTemplate attribute. All the validators discussed so far are having a MessageTemplate attribute. This attribute is used to set the validation message to be displayed on fail of its validation. By default, all validators will be having an in-built Message template. Like, default MessageTemplate for Range validator is "The length of the value must fall within the range {0} (Ignore) - {2} (Inclusive)."

We will see how to pass substitution values to MessageTemplate. Add this code as shown below:

```
[SelfValidation(Ruleset = "Group1")]
public void CheckSalary(ValidationResults results)
{
    if (Salary < 0)
    {
        results.AddResult(new ValidationResult(String.Format( "Salary {0} cannot be Negative.Error at {1}",Salary,this.ToString()),this,null,
            null,null));
    }
}
```

Now, run the application. The output will be like this:



```
Salary -2000 cannot be Negative.Error at CodeBasedVAB.Employee
Not a Valid Emp ID.
```

6 Adaptors Provided By the Validation Application Block

Now, we will look into the adaptors provided by the VAB. VAB is having adaptors for ASP.NET, Win Forms and WCF technologies. Here we will start with ASP.NET adapter.

Create a new ASP.NET Web application in VS 2008 and name it as ASPAdapterVAB. Now add the following dlls present in Enterprise Library installation folder:

- Microsoft.Practices.EnterpriseLibrary.Common,
- Microsoft.Practices.EnterpriseLibrary.Validation.Validators and
- Microsoft.Practices.EnterpriseLibrary.Validation.Integration.AspNet

Now, add a class Employee to default.aspx as shown below:


```
class Employee
{
    [RangeValidator(0, RangeBoundaryType.Inclusive, 2000, RangeBoundaryType.Inclusive, Ruleset = "Group1",
    MessageTemplate = "ID should be below 2000")]
    public int EmpID { get; set; }

    [StringLengthValidator(10, Ruleset = "Group1", MessageTemplate = "Name should be less than 10 characters")]
    public string EmpName { get; set; }

    [RangeValidator(0, RangeBoundaryType.Inclusive, 20000, RangeBoundaryType.Inclusive, Ruleset = "Group1",
    MessageTemplate = "Salary should be below 20000")]
    public int Salary { get; set; }

    [RegexValidator(@"^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\w\.-]*[a-zA-Z0-9]\.[a-zA-Z][a-zA-Z\.-]*[a-zA-Z]$",
    Ruleset = "Group1", MessageTemplate = "Not a Valid E-Mail.")]
    public string EmpMailID { get; set; }
}
```

Create a UI on default page as shown below:

Enter Emp ID:

cc1:propertyproxy...#EmpIDValidator
Employee ID is not Valid.

Enter Emp Name:
Employee Name is Not Valid.

Enter Salary:
Salary is not Valid.

Enter E-Mail:
Employee Mail ID is not Valid.

☐ Show Detailed Validations

We have few labels, textboxes and PropertyProxyValidators [present in Microsoft.Practices.EnterpriseLibrary.Validation.Integration.AspNet.dll] on the page.

We can add this dll to ToolBox, go to Tool Box → Choose Items → Select Microsoft.Practices.EnterpriseLibrary.Validation.Integration.AspNet.dll → Check PropertyProxyValidator → Drag & Drop it onto the page.

We need to set following properties for each Validator:

Text → Validation Message that need to be displayed.

ControlToValidate → ID of the control to validate.

PropertyName → Property Name for doing validations.

RuleSetName → Name of the Rule set.

SourceTypeName → Complete path of the class [Namespace.ClassName].

SpecificationSource → Both [default option]. We can set this to either Attributes or Configuration based on the location of validation code.

Now, add the code to click event of Save button as shown below:

```
protected void Button1_Click(object sender, EventArgs e)
{
    Employee objEmp = new Employee();
    objEmp.EmpID = Convert.ToInt32(txtEmpID.Text.Trim());
    objEmp.EmpName = txtEmpName.Text.Trim();
    objEmp.Salary = Convert.ToInt32(txtSalary.Text.Trim());
    // Run all Validators
    EmpIDValidator.Validate();
    EmpMailIdValidator.Validate();
    EmpSalaryValidator.Validate();
    EmpNameValidator.Validate();
    //Update Validation Messages on UI.
    if (!EmpIDValidator.IsValid)
    {
        if (chkDetails.Checked)
            EmpIDValidator.Text = EmpIDValidator.ErrorMessage;
    }
    if (!EmpMailIdValidator.IsValid)
    {
        if (chkDetails.Checked)
            EmpMailIdValidator.Text = EmpMailIdValidator.ErrorMessage;
    }
    if (!EmpSalaryValidator.IsValid)
    {
        if (chkDetails.Checked)
            EmpSalaryValidator.Text = EmpSalaryValidator.ErrorMessage;
    }
    if (!EmpNameValidator.IsValid)
    {
        if (chkDetails.Checked)
            EmpNameValidator.Text = EmpNameValidator.ErrorMessage;
    }
}
```

Here, Validate method will do the validations and sets IsValid property based on it. Later, we are displaying the error messages coming from validators present in Employee class if chkDetails is checked. If chkDetails is not checked, then we are displaying error message present in the Text property of each Validator.

Now, run the application. The output will be like this:

Enter Emp ID: ID should be below 2000

Enter Emp Name: Name should be less than 10 characters

Enter Salary: Salary should be below 20000

Enter E-Mail: Not a Valid E-Mail.

☒ Show Detailed Validations

I will just outline the steps in using this VAB adapter for ASP.NET.

1. Create validation code using either attributes or configuration.
2. Import VAB adapter for ASP.NET into client application.
3. Create PropertyProxyValidator and set attributes discussed above.
4. Run Validate methods of the validators.
5. Check IsValid property for each validator and display error message.

Sometimes, we might input invalid data like entering non-numeric values for salary. In order to handle that, PropertyProxyValidator is having ValueConvert event. Now, we handle invalid salary using EmpSalaryValidator as shown below:

```
protected void EmpSalaryValidator ValueConvert(object sender,
    Microsoft.Practices.EnterpriseLibrary.Validation.Integration.ValueConvertEventArgs e)
{
    int salary;
    if (Int32.TryParse(e.ValueToConvert.ToString(), out salary))
    {
        e.ConvertedValue = salary;
    }
    else
    {
        e.ConvertedValue = null;
        e.ConversionErrorMessage = "Salary is not valid Numeric Data";
    }
}
```

If the data is invalid, it will display an error message and set ConvertedValue to null.

7 Windows Forms Validation using VAB Adaptor

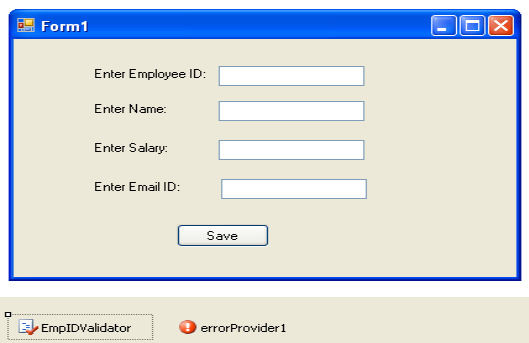
Here, we will discuss about the Windows Form Validation using VAB Adaptor.

Create a new Win Forms application in VS 2008 and name it as WinFormsAdapterVAB. Now add the following dlls present in Enterprise Library installation folder:

Microsoft.Practices.EnterpriseLibrary.Common,
 Microsoft.Practices.EnterpriseLibrary.Validation.Validators and
 Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms

I am using same old Employee class definition.

Create UI on Form1 as shown below:



We have few labels, textboxes and ValidationProvider [present in Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms dll] on the page.

We need to set following properties for ValidationProvider named as EmpIDValidator:

©2013 C# CORNER.

SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

Enabled → true.

ErrorProvider → errorprovider1.

RuleSetName → Name of the Rule set [Group1].

SourceTypeNames → WinFormsAdapterVAB.Employee, WinFormsAdapterVAB.

SpecificationSource → Both [default option]. We can set this to either Attributes or Configuration based on the location of validation code.

Then, we need to set following properties for each textbox:

ValidatedProperty on EmpIDValidator → should be name of the property of the control from where the value needs to be extracted for validating like Text property.

SourcePropertyName on EmpIDValidator → should be name of the property on the source type for which validation information should be retrieved to validate value for the control.

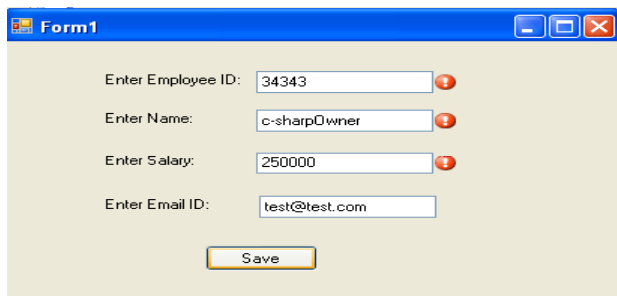
PerformValidation on EmpIDValidator → should be true/false, if it is true automatic validation will be performed when the validating event is fired like tab out.

Now, add the code to click event of Save button as shown below:

```
private void button1_Click(object sender, EventArgs e)
{
    //Validate Emp ID
    EmpIDValidator.PerformValidation(txtEmpID);
    //Validate Emp Name
    EmpIDValidator.PerformValidation(txtEmpName);
    //Validate Emp Salary
    EmpIDValidator.PerformValidation(txtSalary);
    //Validate Emp Mail ID
    EmpIDValidator.PerformValidation(txtEmailID);
    if (EmpIDValidator.IsValid)
    {
        MessageBox.Show("Data is correct..");
    }
}
```

Here, PerformValidation method will do the validation on the control passed to it and displays error message using ErrorProvider.

Now, run the application. The output will be like this:



I will just outline the steps in using this VAB adapter for Win Forms.

1. Create validation code using either attributes or configuration.
2. Import VAB adapter for Win Forms into the application.
3. Create ValidationProvider and set attributes discussed above.
4. Set properties for each textbox discussed above.
5. Run validation on the controls using provider's PerformValidation.

Sometimes, we might input invalid data like entering non-numeric values for salary. In order to handle that, ValidationProvider is having ValueConvert event. Now, we handle invalid salary and empid using EmpIDValidator as shown below:

```
private void EmpIDValidator_ValueConvert(object sender, Microsoft.Practices.EnterpriseLibrary.Validation.Integration.ValueConve
{
    if (e.SourcePropertyName == "Salary" || e.SourcePropertyName == "EmpID")
    {
        int sal;
        if (Int32.TryParse(e.ValueToConvert.ToString(), out sal))
        {
            e.ConvertedValue = sal;
        }
        else
        {
            e.ConvertedValue = 0;
            e.ConversionErrorMessage = e.SourcePropertyName+" is not numeric";
        }
    }
}
```

If the data is invalid, it will display an error message and set ConvertedValue to 0.

In certain cases, we might need to run some code after validation is performed. In order to handle that, ValidationProvider is having ValidationPerformed event. Now, we display an alert after validating empid as shown below:

```
private void EmpIDValidator_ValidationPerformed(object sender, Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms.Val
{
    if (e.ValidatedControl == txtEmpID)
    {
        if (e.ValidationResults.Count == 0)
        {
            MessageBox.Show("Emp Id is correct...");
        }
    }
}
```

8 VAB Adaptor for WCF

Now, we will About the VAB Adaptor for WCF.

Create a new WCF Service Library project in VS 2008 and name it as WCFAdapterVAB. Now add the following dlls present in Enterprise Library installation folder:

- Microsoft.Practices.EnterpriseLibrary.Common,
- Microsoft.Practices.EnterpriseLibrary.Validation.Validators and
- Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF

Now, delete all the existing class files in the project. We need to add DataContract, ServiceContract for Service. Add a class file named as Employee.cs for DataContract with below code:

```
using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
using Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF;

namespace WCFAdapterVAB
{
    [DataContract]
    public class Employee
    {
        [DataMember]
        public int EmpID { get; set; }
        [DataMember]
        [StringLengthValidator(1, RangeBoundaryType.Inclusive, 10, RangeBoundaryType.Inclusive,
MessageTemplate = "Name must be between 1 and 10.")]
        public string EmpName { get; set; }
        [DataMember]
        public double Salary { get; set; }
        [DataMember]
        public string EmpMailID { get; set; }
    }
}
```

Add another class named as IEmpService.cs for ServiceContract with below code:

```
namespace WCFAdapterVAB
{
    [ServiceContract]
    [ValidationBehavior]
    public interface IEmpService
    {
        [OperationContract]
        [FaultContract(typeof(ValidationFault))]
        void AddEmployee(Employee objEmp);
    }
}
```

```
[OperationContract]
[FaultContract(typeof(ValidationFault))]
Employee GetEmpDetails([RangeValidator(1,RangeBoundaryType.Inclusive,100,
RangeBoundaryType.Inclusive,MessageTemplate="ID should be between 1-100")]int id);
}
}
```

We need to specify FaultContract property for methods throwing validation errors.

Now, we will implement the service methods in another class file named as EmpService.cs with below code:

```
namespace WCFAdapterVAB
{
    public class EmpService : IEmpService
    {
        #region IEmpService Members
        private List<Employee> empList = new List<Employee>();
        public void AddEmployee(Employee objEmp)
        {
            empList.Add(objEmp);
        }

        public Employee GetEmpDetails(int id)
        {
            return empList.Find(e => e.EmpID.ToString().Equals(id.ToString()));
        }

        #endregion
    }
}
```

Here, we are validating parameter range for GetEmpDetails and attribute-based validation on EmpName.

Build the solution and run/deploy the service. Now, we need to create a client for testing this service. So, create a new console application and name it as WCFAdapterVABClient. Add service reference of the WCFAdapterVAB and write the below code in Program.cs:

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
using Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF;
using System.ServiceModel;

namespace WCFAdapterVABClient
{
    class Program
```



```
{
    static void Main(string[] args)
    {
        try
        {
            EmpService1.EmpServiceClient service1 = new
                WCFAdapterVABClient.EmpService1.EmpServiceClient();
            EmpService1.Employee objemp =
                new WCFAdapterVABClient.EmpService1.Employee();
            objemp.EmpID = 15;
            objemp.EmpName = "c-sharpcorner";
            service1.AddEmployee(objemp);
            EmpService1.Employee emp = service1.GetEmpDetails(15);
            EmpService1.Employee objemp1 = service1.GetEmpDetails(150);
        }
        catch (FaultException<ValidationFault> ex)
        {
            ValidationFault fault = ex.Detail;
            foreach (ValidationDetail validationResult in fault.Details)
            {
                Console.WriteLine(validationResult.Message);
            }
            Console.ReadLine();
        }
    }
}
```

Run the client application, the output will be like this:



Suggestion:

When we debug WCF service having VAB validation, sometimes we may get unhandled user exceptions; to handle that goto VS → Debug → Exceptions → Uncheck User-unhandled checkbox under CLR Exceptions for ServiceModel and other required classes.

9 Implementation of Custom Validators using VAB

Now, we will look into creation of custom validator.

Create a new Class Library in VS 2008 and name it as CustomValidators. Now add the following dlls present in Enterprise Library installation folder:

- Microsoft.Practices.EnterpriseLibrary.Common,
- Microsoft.Practices.EnterpriseLibrary.Validation.Validators and finally
- System.Configuration.

Now, delete existing class file in the project. We need to add two classes to the project, one is for the validator definition and other is validator's attribute definition. Add a class file named as EmployeeValidator.cs with below code:

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
using Microsoft.Practices.EnterpriseLibrary.Validation.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using System.Collections.Specialized;

namespace CustomValidators
{
    [ConfigurationElementType(typeof(CustomValidatorData))]
    public class EmployeeValidator:Validator<string>
    {
        private string joinedDt;

        public EmployeeValidator(NameValueCollection attributes) : base(null, null)
        {
            joinedDt = attributes.Get("JoinedDate").ToString();
        }

        public EmployeeValidator(string joinedDt)
            : this(joinedDt, null, null)
        {
        }

        public EmployeeValidator(string joinedDt, string messageTemplate)
            : this(joinedDt, messageTemplate, null)
        {
        }

        public EmployeeValidator(string joinedDt, string messageTemplate, string tag)
            : base(messageTemplate, tag)
        {
            this.joinedDt = joinedDt;
        }
    }
}
```

```
protected override void DoValidate(string relievedDt, object currentTarget, string key,
Microsoft.Practices.EnterpriseLibrary.Validation.ValidationResults validationResults)
{
    if (DateTime.Parse(relievedDt) < DateTime.Parse(joinedDt))
    {
        string message = string.Format(this.MessageTemplate, relievedDt, joinedDt);
        this.LogValidationResult(validationResults, message, currentTarget, key);
    }
}

protected override string DefaultMessageTemplate
{
    get { return "Relieved Date {0}is not valid for Joined Date {1}"; }
}
}
```

Here, we had implemented Validator class in our code. We need to add [ConfigurationElementType(typeof(CustomValidatorData))] to the class. Then, we added few constructors followed by implementation DoValidate method. In this method, we need to write validation code and return results back. This method checks whether the relievedDate is greater than JoinedDate or not for an employee.

Now, add another class file named as EmployeeValidatorAttribute.cs with below code:

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;

namespace CustomValidators
{
    public class EmployeeValidatorAttribute : ValidatorAttribute
    {
        private string joinedDt;

        public EmployeeValidatorAttribute(string joinedDt)
        {
            this.joinedDt = joinedDt;
        }

        protected override Validator DoCreateValidator(Type targetType)
        {
            return new EmployeeValidator(joinedDt);
        }
    }
}
```

```

    }
}

}

```

Here, we had implemented ValidatorAttribute class in our code. Finally, we had implemented DoCreateValidator method; this will create an instance of the validator with passed parameters.

Now, we need to test this validator. So, create a new Console application named as CustomValidatorsClient with below code in program.cs:

```

using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
using CustomValidators;
namespace CustomValidatorsClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Employee objEmp = new Employee();
            objEmp.EmpID = 10;
            objEmp.RelievedDt = "01-JAN-2008";
            Validator<Employee> validator = ValidationFactory.CreateValidator<Employee>();
            ValidationResult results = validator.Validate(objEmp);
            foreach (ValidationResult res in results)
            {
                Console.WriteLine(res.Message);
            }
            Console.ReadLine();
        }
    }
    class Employee
    {
        public int EmpID { get; set; }
        [EmployeeValidator("02-AUG-2008")]
        public string RelievedDt { get; set; }
    }
}

```

We need to add reference to VAB dlls and our custom Validator dll also. Run the application, the output will be like this:

Relieved Date 01-JAN-2008 is not valid for Joined Date 02-AUG-2008

Steps to create our own Validator:

- Write a class derived from `Validator<T>` or existing validators like `DomainValidator`.
- Include required properties and constructors.
- Override `DoValidate` method and call `LogValidationResult` method on fail of validations.
- Override `DefaultMessageTemplate` property and include the message need to be displayed, if `MessageTemplate` attribute is not defined by the client.
- Apply `ConfigurationElementType` to the class having reference to `typeof(CustomValidatorData)` in it.
- Write a class derived from `ValidatorAttribute` with desired constructors and properties.
- Override `DoCreateValidator` method. This should return an instance of the validator with required attribute values.
- Build the Validator.
- Add the reference of the custom validator in the client.

10 Integration of VAB using Policy Injection Application Block

Now, we look into integration of VAB with Policy Injection Application Block [PIAB]. PIAB is a collection of handlers for implementing common scenarios like caching, validation, logging etc.

Create a new console application in VS 2008 and name it as `VABAndPIABIntegration`.

Add a reference to below Enterprise Library dlls present in its Installation folder:

Microsoft.Practices.EnterpriseLibrary.Common.dll
Microsoft.Practices.EnterpriseLibrary.PolicyInjection.dll
Microsoft.Practices.EnterpriseLibrary.PolicyInjection.CallHandlers.dll
Microsoft.Practices.EnterpriseLibrary.Validation.dll
Microsoft.Practices.Unity.Interception.dll

Add a new class `Employee` with below definition in `Program.cs`:

```
class Employee : MarshalByRefObject
{
    private static List<Employee> empList = new List<Employee>();
    public int EmpID { get; set; }
    public string EmpName { get; set; }
    public double Salary { get; set; }
    public void AddEmployee(Employee emp)
    {
        empList.Add(emp);
    }
}
```

```
[ValidationCallHandler]
public Employee
GetEmpDetails([RangeValidator(1,RangeBoundaryType.Inclusive,100,RangeBoundaryType.Inclusive,MessageTemplate="Emp ID should be below 100.")] int empid)
{
    foreach (Employee emp in empList)
    {
        if (emp.EmpID == empid)
        {
            return emp;
        }
    }
    return null;
}
```

Here, we are implementing MarshalByRefObject class. In order to use PIAB, it's mandatory to implement MarshalByRefObject. Then, we add ValidationCallHandler attribute to GetEmpDetails method. When we add this attribute, it will intercept the calls to that method and validates the parameters. If validation fails, it will throw ArgumentException and returns back without calling the method.

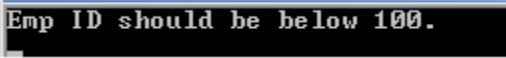
Now, we need to test this method. So, add the below code to Main method:

```
using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.PolicyInjection;
using Microsoft.Practices.EnterpriseLibrary.PolicyInjection.CallHandlers;
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
```

```
static void Main(string[] args)
{
    Employee e1 = PolicyInjection.Create<Employee>();
    e1.EmpID = 10;
    e1.EmpName = "Test";
    e1.Salary = 1000;
    e1.AddEmployee(e1);
    Employee e2 = PolicyInjection.Create<Employee>();
    e2.EmpID = 20;
    e2.EmpName = "CSharp";
    e2.Salary = 2000;
    e2.AddEmployee(e2);
    try
    {
        Employee e3 = e1.GetEmpDetails(150);
```

```
if (e3 != null)
{
    Console.WriteLine("Emp ID" + e3.EmpID);
    Console.WriteLine("Emp Name" + e3.EmpName);
    Console.WriteLine("Employee Salary" + e3.Salary);
}
}
catch (ArgumentValidationException ex)
{
    foreach (ValidationResult rs in ex.ValidationResults)
    {
        Console.WriteLine(rs.Message);
    }
}
Console.ReadLine();
}
```

After importing necessary namespaces, we are creating instances of Employee class using PolicyInjection's Create method. It's mandatory to use Create method for instance creation by which PIAB can intercept the calls. Now, run the application, the output will be like this:



Finally, I will outline the steps of the Integration:

- VAB can be used with PIAB to automatically validate parameters of a method.
- Validation Rules can be specified within Parameter types or as attributes in the parameter definition.
- Instances of the class should be created using PolicyInjection's Create method.
- Validation handler can be applied using either attributes or Configuration.
- If validation fails, an ArgumentValidationException will be thrown. This Exception will be having all validation results in it.