Configurator Services Interface

It is possible to work with Configurator Services without using JSON/AML serialization. This is provided by a public interface IConfiguratorServices implemented in the Aras.Server.Core.Abstractions namespace.

The interface provides the following functionality described in section **Error! Reference source not found. Error! Reference source not found**:

- alternate for the cfg_GetScopeStructure method (without output_builder_method parameter, see 5.2.1 cfg_GetScopeStructure);
- alternate for the cfg_GetValidCombinations method (see 5.2.2 cfg_GetValidCombinations)

IConfiguratorServices interface allows working with objects from Configurator Services without using JSON/AML. This can be beneficial in some cases from a performance perspective.

This section describes:

- Getting an IConfiguratorServices interface
- Interface description
- Usage example

Getting an IConfiguratorServices interface

IConfiguratorService interface object can be received from a call context in a server method.

```
CCO.GetService<Aras.Server.Core.Abstractions.IConfiguratorServices>();
```

Interface description

The interface definition is:

```
public interface IConfiguratorServices
{
    ReadOnlyScope GetScope(Item targetScopeItem);
    IScopeSolver CreateSolver(ReadOnlyScope scope);
}
```

Where:

ReadOnlyScope	Class representing a Scope object with list of rules and list of variables. It cannot be modified.
IScopeSolver	Interface that represents a scope solver. This interface is responsible for setting the scope and getting combinations.

Both interface methods are described below.

GetScope

```
public ReadOnlyScope GetScope(Item targetScopeItem);
```

> **Important**
>
> GetScope method provides similar functionality as cfg_GetScopeStructure (refer to5.2.1 cfg_GetScopeStructure) except the output_builder_method.

Method returns ReadOnlyScope with immutable list of rules and list of variables. ReadOnlyScope class defined as:

```
public class ReadOnlyScope
{
    public string Id;
    public string Name;
    public ReadOnlyCollection<Rule> RuleList;
    public ReadOnlyCollection<Variable> VariableList;
}
```

> **Important**
>
> RuleList and VariableList contain references to the actual objects. Changing referenced Rules and Variables would change cached Scope Rules and Variables.

`targetScopeItem` parameter can be defined like (see *Input Item Format*)

```
Item targetScopeItem = this.newItem("Method", "%builder method name%");
targetScopeItem.setAttribute(
    "%builder method attribute name%",
    "%builder method attribute value%");
targetScopeItem.setProperty(
    "%builder method property name%",
    "%builder method property value%");
```

Where:

%builder method name%	Name of the server method that builds the Scope object model based on the business logic being used.
%builder method attribute name%	Builder method attribute name.
%builder method attribute value%	Builder method attribute value.
%builder method property name%	Builder method property name.
%builder method property value%	Builder method property value.

CreateSolver

```
public IScopeSolver CreateSolver(ReadOnlyScope scope);
```

IScopeSolver interface is declared as following:

```
public interface IScopeSolver
{
    ReadOnlyScope Scope { get; }
    ExpressionBase Condition { get; set; }

    IEnumerable<IEnumerable<ExpressionTerm>> FindAllCombinations(
        IList<Variable> selectVariables);

    IEnumerable<IEnumerable<ExpressionTerm>> FindAllCombinations(
        IList<Variable> selectVariables,
        ExpressionBase additionalCondition);
}
```

Important

FindAllCombinations method provides similar functionality as cfg_GetValidCombinations (refer to cfg_GetValidCombinations).

FindAllCombinations

Method returns all possible valid combinations. Returned object is a collection of `IEnumerable<ExpressionTerm>` objects. A single `IEnumerable<ExpressionTerm>` is a unique combination of all variables and their values.

ExpressionTerm class consists of two fields:

- OperandLeft – represents a certain variable in combination.
- OperandRight – represents a value (named-constant) corresponding to the variable.

Both operands contain an Id and Value of a variable (for the OperandLeft) and of named-constant (for the OperandRight).

Usage example

This section provides an example of using the public interface: There is a need to check if any valid combination will exist when certain variable options (values) are selected. Therefore, it needs to be checked if such selected variable options are valid in the Scope.

The corresponding server method code could be the following.

```
// Get IConfiguratorServices object.
Aras.Server.Core.Abstractions.IConfiguratorServices configuratorModule =
    CCO.GetService<Aras.Server.Core.Abstractions.IConfiguratorServices>();

// Set targetScope.
Item targetScopeItem = this.newItem("Method", "scopeBuilderMethod");
string businessItemId = "6B9B49B827C44F38A97BBFB524CA5BD5";
targetScopeItem.setID(businessItemId);

// Get scope.
Aras.Server.Core.Configurator.ReadOnlyScope scope =
    configuratorModule.GetScope(targetScopeItem);

// Create ScopeSolver.
Aras.Server.Core.Configurator.IScopeSolver scopeSolver =
    configuratorModule.CreateSolver(scope);

string selectedVariable1Id = "62419B0CF8B1464CA73FF72B072924E5";
string selectedOption1Id = "3021A89854F24F38918BCF2CFC27CAD3";
string selectedVariable2Id = "0C6562E7925C4215ABE6E31EF0ED1BD2";
string selectedOption2Id = "B574E44C807545C2B74BA8A30CA8B319";

// Create selected variable options as a condition and set them to the solver.
scopeSolver.Condition = new Aras.Server.Core.Configurator.ExpressionLogical(
    Aras.Server.Core.Configurator.ExpressionType.AND);
scopeSolver.Condition.InnerExpressions.AddRange(
    new List<Aras.Server.Core.Configurator.ExpressionBase>
    {
        // SelectedVariableId is the Id of the selected variable.
        // SelectedOptionId is the Id of corresponding option(value) of the selected variable.
        new Aras.Server.Core.Configurator.ExpressionTerm(
            Aras.Server.Core.Configurator.TermOperator.Equal,
            new Aras.Server.Core.Configurator.TermOperandVariable(selectedVariable1Id),
            new Aras.Server.Core.Configurator.TermOperandNamedConstant(selectedOption1Id)),
        new Aras.Server.Core.Configurator.ExpressionTerm(
            Aras.Server.Core.Configurator.TermOperator.Equal,
            new Aras.Server.Core.Configurator.TermOperandVariable(selectedVariable2Id),
            new Aras.Server.Core.Configurator.TermOperandNamedConstant(selectedOption2Id)),
    });

// Null parameter for FindAllCombinations means that all scope variables will be used.
return scopeSolver.FindAllCombinations(null).Any();
```