ESDL Language Reference

Boca Raton Documentation Team

ESDL Language Reference

Boca Raton Documentation Team

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com>

Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Version Number in the text of the message.

2026 Version 10.6.18-1

ESDL Language Overview

ESDL (Enterprise Service Description Language) is a methodology that helps you develop and manage web-based query interfaces quickly and consistently.

Dynamic ESDL takes an interface-first development approach. It leverages the ESDL Language to create a common interface "contract" that both Roxie Query and Web interface developers will adhere to. It is intended to allow developers to create production web services, with clean interfaces that can evolve and grow over time without breaking existing applications.

ESDL's built-in versioning support helps ensure compiled and deployed applications continue to operate while changes are made to the deployed service's interface for new functionality.

ESDL's ability to define and reuse common structures helps maintain consistent interfaces across methods.

The Dynamic ESDL service is built to scale horizontally, and hooks are provided to add custom logging and security to help create fully "productionalized" web services.

Once a service is deployed, application developers and end-users can consume the service using REST, JSON, XML, SOAP, or form encoded posts. Dynamic ESDL provides quick and easy access to a WSDL, live forms, sample requests and responses, and testing interfaces to allow developers to test logic changes, data changes, or new features, as well as to interact with the service directly using SOAP, XML, or JSON.

ESDL Structures

ESPstruct

ESPstruct is a set of elements grouped together under one name. These elements, known as *members*, can have different types and different lengths. Structures can be nested and support inheritance.

Example:

```
ESPstruct NameBlock

{
    string FirstName;
    string MiddleName;
    string LastName;
    int Age;
};

ESPstruct NameBlockExtended:NameBlock

{
    string SSN;
    string partyAffilition;
};
```

ESPrequest

The request structure for a method. ESPrequests can be nested and support inheritance.

Example:

```
ESPrequest  MyQueryRequest

{
    string FirstName;
    string MiddleName;
    string LastName;
    string Sortby;
    bool Descending(false);

};

ESPrequest MyQueryRequestExtended:MyQueryRequest

{
    string SSN;

};
```

ESPresponse

The response structure for a method. ESPresponses can be nested and support inheritance.

Example:

```
ESPresponse  MyQueryResponse

{
    string FirstName;
    string MiddleName;
    string LastName;

};

ESPresponse MyQueryResponseExtended:MyQueryResponse

{
    string SSN;
    string partyAffilition;
};
```

ESParray

A structure for unbounded arrays. Arrays support inheritance and can be nested.

Either [max_count_var(k)] or [max_count_var(k)] is required for an ESPArray when k>1.

Example:

```
ESPstruct NameBlock

{
    string FirstName;
    string MiddleName;
    string LastName;

    int Age;
};

[max_count(20)] ESParray <ESPstruct NameBlock, Name> Names;
```

This results in something like:

```
<Names>
  <Name>
    <FirstName>James</FirstName>
    <MiddleName>Joseph</MiddleName>
    <LastName>Deerfield</LastName>
    <Age>42</Age>
  </Name>
  <Name>
    <FirstName>Emily</FirstName>
    <MiddleName>Kate</MiddleName>
    <LastName>Constance</LastName>
    <Age>33</Age>
  </Name>
</Names>
```

See Also: max_count_var, max_count

ESPenum

A structure containing an enumerated value.

Example:

```
ESPenum EyeColors : string

{
  Brn("Brown"),
  Blu("Blue"),
  Grn("Green"),
};


ESPstruct Person

{
  string FirstName;
  string MiddleName;
  string LastName;

  ESPenum EyeColors EyeColor("Brown"); //provides a default value
};
```

ESPinclude

ESPinclude allows you to include an external ESDL file. This is similar to the #include statement.

Example:

```
ESPinclude(commonStructures);
```

ESPservice

This defines an ESP web service interface. Once defined, this interface definition can be assigned (bound) to a Dynamic ESDL-based ESP Service.

An ESPservice should contain one or more method definitions.

Example:

```
ESPservice [auth_feature("AllowMyService")] MyService
{
  ESPmethod MyMethod1(MyMethod1Request, MyMethod1Response);
  ESPmethod MyMethod2(MyMethod2Request, MyMethod2Response);
};
```

ESPmethod

This defines a method definition you can reference in an ESPservice structure. The method definition should contain references to a previously defined ESPrequest and ESPresponse.

Example:

```
ESPservice MyService
 {
  ESPmethod MyMethod1(MyMethod1Request, MyMethod1Response);
  ESPmethod
   [
    auth_feature("AllowMyMethod2"),
    description("MyMethod Two"),
    help("This method does everything MyMethod1 does plus a few extra features"),
    min_ver("1.2")
   ]
  MyMethod2(MyMethod2Request, MyMethod2Response);
};
```

ESDL Datatypes

boolean / bool

A boolean or logical data type having one of two possible values: true (1) or false (0).

Example:

```
boolean includeFlag;
bool includeMore;
```

string

A data type consisting of sequence of alphanumeric characters.

Example:

```
string FirstName;
```

int

An integer value.

Example:

```
int Age;
```

int64

A 64-bit integer value

Example:

```
int64 Iterations;
```

float

A 4-byte floating point or real number.

Example:

```
float Temperature;
```

double

An 8-byte floating point or real number.

Example:

```
double Temperature;
```

binary

A data type containing binary data, similar to a BLOB .

Example:

```
binary RetinaScanSample;
```

ESDL, XSD, and ECL Type Mapping

ESDL	XSD	ECL
Bool boolean	bool	boolean
Binary	Base64Binary	String (base64 encoded)
Float	float	REAL4
Double	double	REAL8
Int	int	INTEGER
Int64	long	INTEGER8
String	string	String

ESDL Attributes

You can use ESDL attributes to extend and override the default behavior of an ESDL definition. For example, adding a max_len(n) to a string defines the string will only need to store a certain number of characters.

Many attributes are treated as hints that may have more effect on some implementations than others. For example, max_len(n) will affect generated ECL code, but is ignored when generating C++.

max_len (n)

The *max_len* attribute specifies the field length for ECL string field.

Example:

```
[max_len(20)] string City;
```

It means that in ECL, City field is a fixed length of 20 chars. For integer type, the max_len means the maximum size in bytes for the integer (8*max_len bits integer).

Example:

```
[max_len(3)] int Age;
```

This generates ECL code:

```
integer3 Age{xpath('Age')};
```

This attribute works for ESPenum type, too. The ECL type is also string.

```
[max_len(2)] ESPenum StateCode State;
```

Here the StateCode is 2-char state code enumeration.

This attribute can also be used for ESPstruct, ESPrequest, ESPresponse:

```
ESPstruct [max_len(1867)] IdentitySlim : Identity
  {
     ...
  };
```

This generates ECL code:

```
export t_MyQuery := record (share.t_Name), MAXLENGTH(1867)
  {
  };
```

The ECL option *MAXLENGTH* helps ECL engine better manage memory.

This does not affect the type in the XSD/WSDL. ESP ignores this attribute when generating the schema. The type for a string is xsd:string which has no length limit. Therefore, the schema stays the same if the field length changes in the Roxie query.

ecl_max_len (n)

This *ecl_max_len* attribute tells the ECL generator to use ECL *maxlength* instead of the regular field length.

Example:

```
[ecl_max_len(50)] string CompanyName;
[max_len(6)] string Gender;
```

The generated ECL code is:

```
string CompanyName { xpath("CompanyName"),maxlength(50) };
   string6 Gender { xpath("Gender") };
```

Note: when both *max_len* and *ecl_max_len* are specified, *ecl_max_len* is used and *max_len* is ignored.

ecl_name ("name")

The *ecl_name* attribute specifies the field name in generated ECL code. By default, the field name in ECL is the same as the name defined in ECM. However, in some cases, the name could causes issues in ECL. For example keywords in ECL cannot be used as a field name.

Example:

```
[ecl_name("_export")] string Export;
 [ecl_name("_type")] string Type;
```

Here, both **EXPORT** and **TYPE** are ECL keywords and cannot be used as ECL field names. We use *ecl_name* to tell the esdl2ecl process to generate acceptable names.

counter and count_val

These two attributes are used to help ESP calculate the record count of the response.

counter counts the number of children of the nodes. When the node is an array, it is the same as the number of items in the array.

count_val will use the value of the node as record count. Field **RecordCount** is implicitly marked as *count_val*.

When an response has multiple counter, count_val, the sum of the values is returned as record-count.

Example:

```
[counter] ESParray<MyRecord, Record> Records;
[count_val] int TotalFound;
```

max_count

The max_count attribute is used to specify the expected max items in a dataset (ESParray).

Example:

```
[max_count(20)] ESParray <ESPstruct MYRecord, Record> Records;
```

See Also: ESParray, max_count_var

max_count_var

The max_count_var attribute is used to specify a variable (an ECL Attribute) containing the value of the expected max items in a dataset (ESParray).

Example:

```
[max_count_var("iesp.Constants.JD.MaxRecords")] ESParray <ESPstruct MYRecord, Record> Records;
```

The ECL developer defines the constant iesp.Constants.JD.MaxRecords rather than hard coding the max count value in the ESDL.

See Also: ESParray, max_count

ecl_null (n | string)

The *ecl_null* attribute tells ESP to remove the field altogether if the field's value is *n* or *string*. This provides a means to remove a field completely when there is no data for it.

Example:

```
[ecl_null(0)] int Age;
[ecl_null("false")] bool IsMatch;
```

Age 0 means there is no Age data for this person. So, if Age is 0, the <Age> tag is not returned.

Without this attribute, **<Age>0</Age>** would be returned.

For the second example, a bool value of *false*, returned as a string, is treated as null and therefore the tag is not returned.

leading_zero(n)

The *leading_zero* attribute adds zero(s) to the field value to so that the total length is *n*.

Example:

```
ESPstruct Date
 {
    [leading_zero(4)] Year;
    [leading_zero(2)] Month;
    [leading_zero(2)] Day;
 };
```

So the Date will always have a 4-digit Year and a 2-digit Month and a 2-digit Day.

ecl_hide

The *ecl_hide* attribute hides the field from ECL (that is, the field is removed when generating the ECL code). This is used for some special cases.

Example:

```
ESPstruct Relative
 {
    [ecl_hide] ESParray<ESPstruct Relative, Relative> Relatives;
    "
 };
```

In this case, the Relative structure is defined in a recursive manner, and ECL does not support such a construct. Therefore, we use *ecl_hide* to avoid the recursive definition in ECL.

Sometimes a field is hidden from ECL for other reasons. In these cases, *ecl_hide* is not needed.

ecl_type ("type")

The *ecl_type* attribute defines the field type in ECL.

Example:

```
[ecl_type("Decimal10_2")] double RetailPrice;
```

ESDL does not have a monetary type, so we use *ecl_type* to define it.

ecl_keep

The *ecl_keep* attribute keeps the field in the generated ECL even though this field would have been hidden without this attribute.

min_ver

The min_ver attribute allows you to define the minimum (earliest) version where a field is visible. Requests using a prior version will not have access to the field.

Example:

```
[min_ver("1.03")] bool IsValid;
```

max_ver

The max_ver attribute allows you to define the maximum (latest) version where a field is visible. Requests using a later version will not have access to the field.

Example:

```
[max_ver("1.04")] bool IsValid;
```

ping_min_ver

Dynamic ESDL services automatically add a ping method to your service for monitoring connectivity to the service.

The ping_min_ver attribute on a service allows you to define the minimum (earliest) version where the ping method is visible. The ping method is not visible to versions lower than the ping_min_ver

Example:

```
[ping_min_ver("1.03")] ;
```

depr_ver

The depr_ver attribute allows you to declare a field's end of life version. The field is deprecated at the specified version number. Requests using that version or any subsequent version will not have access to the field.

Example:

```
[depr_ver("1.04")] bool IsValid;
```

get_data_from

The get_data_from attribute allows you to specify that a field gets its data from another variable. This supports code reuse when complex versioning changes are made.

Example:

```
ESPresponse RoxieEchoPersonInfoResponse
{
  ESPstruct NameInfo Name;
  string Var1;
  [get_data_from("Var1")] string Var2;
};
```

In the example above, the query returns the data in Var1 then the service puts the data into the Var2 field and sends that in the response to the client.

In this example both Var1 and Var2 are in the response to the client. Typically, Var1 and Var2 are in non-overlapping versions so only one will be in the response depending on the version specified.

Since the get_data_from attribute supports complex data types, such as an ESPstruct, the fields do not have to be limited to string types.

optional

The optional attribute allows you to specify that a field is optional and is hidden or not depending on the absence or presence of a URL decoration.

When a field has an optional attribute, the field is visible only when the option appears on the URL. But when the option starts with an exclamation point (!), then the field is hidden only if the option is in the URL.

Example:

```
ESPrequest RoxieEchoPersonInfoRequest
{
     ESPstruct NameInfo Name;
                            string First;
                            string Middle;
                            string Last;
     [optional("dev")]      string NickName;
     [optional("!_NonUS_")] string SSN;
};
```

Assuming the service is running on a server with the hostname of example.com and the service binding is set to 8003:

If the URL is

```
http://example.com:8003/
```

then SSN is visible and NickName is hidden;

If the URL is

```
http://example.com:8003/?dev
```

then SSN and NickName are both visible

If the URL is

```
http://example.com:8003/?dev&_NonUS_
```

then NickName is visible and SSN is hidden.

If the URL is

```
http://example.com:8003/?_NonUS_
```

then both NickName and SSN are hidden.

help

The help attribute (valid only for an ESPMethod) allows you to specify some additional text to display on the form that is automatically generated to execute a method.

Example:

```
ESPservice MyService

 {
   ESPmethod MyMethod1(MyMethod1Request, MyMethod1Response);
   ESPmethod
    [
     description("MyMethod Two"),
     help("This method does everything MyMethod1 does plus a few extra features"),
     min_ver("1.2")
    ]
   MyMethod2(MyMethod2Request, MyMethod2Response);
 };
```

description

The description attribute (valid only for an ESPMethod) allows you to specify some additional text to display on the form that is automatically generated to execute a method.

Example:

```
ESPservice MyService
{
  ESPmethod MyMethod1(MyMethod1Request, MyMethod1Response);
  ESPmethod
   [
    description("MyMethod Two"),
    help("This method does everything MyMethod1 does plus a few extra features"),
    min_ver("1.2")
   ]
  MyMethod2(MyMethod2Request, MyMethod2Response);
};
```

version and default_client_version

The **version** and **default_client_version** attributes (valid only for an ESPService) allow you to specify the version to use when a version is not explicitly specified in the request.

The **default_client_version** is used for API requests in SOAP format if the client doesn't specify the version. The **version** is used for requests coming from a web browser without a version decoration in the URL.

These attributes provide better API backward compatibility while allowing API developers to see the latest interface using a browser.

If **default_client_version** is higher than **version**, the service uses **default_client_version** for all requests that don't specify a **version**.

Even though defaults can be specified for a service, you should still encourage API developers to specify a version in requests to ensure compatibility between their application and the service.

Example:

```
 ESPservice [version("0.02"), default_client_version("0.01")] EsdlExample
{
    ESPmethod EchoPersonInfo(EchoPersonInfoRequest, EchoPersonInfoResponse);
    ESPmethod RoxieEchoPersonInfo(RoxieEchoPersonInfoRequest, RoxieEchoPersonInfoResponse);
};
```

auth_feature

The *auth_feature* attribute configures feature-level security in ESDL by associating features and access levels to `ESPservice` and `ESPmethod` elements. In the formal syntax, these correspond to `scope-name` values such as `ESDLSERVICE` and `ESDLMETHOD` when evaluating precedence and overrides. Additional scope names may relate to integration-specific bindings. This provides fine-grained, declarative access control for ESDL services and methods.

Higher-precedence scopes may override requirements from ancestor scopes. For service/method scopes defined in .ecm files, precedence is `ESDLSERVICE`, then `ESDLMETHOD`. In DESDL deployments that include binding scopes, precedence then continues with `BINDINGSERVICE`, then `BINDINGMETHOD`.

Examples:

Service-level authentication

Both `GetData` and `ListData` require `SecAccess_Read` access for `DataAccess`.

```
ESPservice [auth_feature("DataAccess:Read")] DataService
{
  ESPmethod GetData(GetDataRequest, GetDataResponse);
  ESPmethod ListData(ListDataRequest, ListDataResponse);
};
```

Combined service and method authentication

`GetData` requires `SecAccess_Read`; `UpdateData` requires both `SecAccess_Read` and `SecAccess_Write` for `DataAccess`.

```
ESPservice [auth_feature("DataAccess:Read")] DataService
{
  ESPmethod GetData(GetDataRequest, GetDataResponse);
  ESPmethod [auth_feature("DataAccess:Write")]
      UpdateData(UpdateDataRequest, UpdateDataResponse);
};
```

Method bypassing service authentication

`GetData` requires `SecAccess_Full` access; `GetPublicData` requires no access for `DataAccess`.

```
ESPservice [auth_feature("DataAccess:Full")] DataService
{
  ESPmethod GetData(GetDataRequest, GetDataResponse);
  ESPmethod [auth_feature("DataAccess:NONE")]
      GetPublicData(GetPublicDataRequest, GetPublicDataResponse);
};
```

An *auth_feature* string is a comma-delimited list of tokens, interpreted in order from service (lowest precedence) to method (highest precedence), and processed in reverse order.

`ESPservice` values take effect at the `EsdlService` scope, while `ESPmethod` values take effect at the `EsdlMethod` scope and may extend or override service scope requirements. If *auth_feature* is set on both service and method, combine both token sets into the resulting security requirement, with method-level tokens taking precedence where conflicts arise.

See the syntax and parsing details for precise semantics.

Token Semantics

Each *auth_feature* may include universal tokens and feature-specific tokens, each with well-defined semantics and support.

Feature-Independent Tokens

The following tokens are universally supported and do not reference any specific feature:

`NONE`	No feature access required. Suppresses all lower-precedence tokens (including feature requirements), but does not override requirements already applied from higher-precedence scopes/tokens. Supported by `esdllib` and `hidl`.
`DEFERRED`	Indicates that authorization may be necessary, but leaves enforcement to the method handler. Because no feature is identified, generic ESP has no context to pre-fetch authorization state. Supported by `esdllib` and `hidl`.
`!X` (where X is empty, a scope, or a feature selector)	Exclusion token. Use `!` to exclude all lower-precedence settings; use `!scope` to exclude a lower-precedence scope; use `!::feature` or `!scope::feature` to exclude a lower-precedence feature. Supported by `esdllib` and `hidl`.

`hidl` applies to .ecm scopes (`ESDLSERVICE`, `ESDLMETHOD`), while `esdllib` additionally supports DESDL binding scopes (`BINDINGSERVICE`, `BINDINGMETHOD`).

Permitted Feature-Specific Tokens

For a feature named `MyFeature`, permitted tokens include:

Token	SecAccess Required
`MyFeature`	SecAccess_Full
`MyFeature:Access`	SecAccess_Access
`MyFeature:Read`	SecAccess_Read
`MyFeature:Write`	SecAccess_Write
`MyFeature:Full`	SecAccess_Full
`MyFeature:NONE`	No access required for this feature; removes it from authorization consideration
`MyFeature:DEFERRED`	Authorization state is obtained for this feature, but it is not enforced for request access unless checked conditionally by the method handler; this supports conditional authorization without another round trip to the security manager

Forbidden Tokens

The following tokens are not permitted for feature `MyFeature`:

`MyFeature:`	Not permitted; invalid token; error because the access level is missing.
`MyFeature:X` (where X is not listed above)	Invalid token; results in an error.

Formal Syntax:

auth_feature tokens follow the grammar and precedence as demonstrated below:

```
token ::= ( exclusion | suppression | deferral | assignment )

exclusion ::= ( exclude-all | exclude-scope | exclude-feature | exclude-fea-
ture-in-scope )
```

Do not treat any form of exclusion as an affirmative security specification.

```
exclude-all ::= '!'
```

Ignore all lower-precedence tokens, including any lower-precedence affirmation of security.

```
exclude-scope ::= '!' scope-name
```

Ignore all lower-precedence tokens specified by the named scope, including any resulting affirmation of security.

```
exclude-feature ::= '!' '::' feature-name
```

Ignore any lower-precedence token that specifies the named feature without a scope qualifier (for example, `!::FeatureName`), including any resulting affirmation of security.

```
exclude-feature-in-scope ::= '!' scope-name '::' feature-name
```

Ignore any lower-precedence token that specifies the named feature in the named scope, including any resulting affirmation of security.

```
suppression ::= ( suppress-all | suppress-feature )

suppress-all ::= 'NONE'
```

Ignore all lower-precedence tokens (including feature requirements) and affirm the absence of security, but does not override requirements already applied from higher-precedence scopes/tokens.

```
suppress-feature ::= feature-name ':NONE'
```

Ignore any lower-precedence token that specifies the named feature, remove that feature from lower-precedence authorization consideration, and discard any resulting lower-precedence requirement to authorize it.

```
deferral ::= ( 'DEFERRED' | feature-name ':DEFERRED' )
```

Standalone `DEFERRED` indicates that authorization may be necessary, but leaves enforcement to the method handler. Because no feature is identified, generic ESP has no context to pre-fetch authorization state.

`feature-name:DEFERRED` preserves authorization state for the named feature without enforcing it for request access, allowing the method handler to perform later conditional authorization without another round trip to the security manager.

```
assignment ::= ( assign-default-level | assign-default-feature | assign-fea-
ture-and-level )

assign-default-level ::= feature-name
```

Ignore any lower-precedence token that specifies the named feature, require full access-level for that feature, and affirm the security state.

```
assign-default-feature ::= ':' access-level
```

Ignore any lower-precedence token that specifies the feature-name equivalent to `${service}Access`, require the given access-level for that feature, and affirm the security state.

```
assign-feature-and-level ::= feature-name ':' access-level
```

Ignore any lower-precedence token that specifies the named feature, require the given access-level for that feature, and affirm the security state.

```
scope-name ::= ( 'DEFAULT' | 'ESDLSERVICE' | 'ESDLMETHOD' | 'BINDINGSERVICE'
| 'BINDINGMETHOD' | ... )
```

For features defined in .ecm files, use `ESDLSERVICE` and `ESDLMETHOD`. In Dynamic ESDL (DESDL) deployments, additional binding scopes may be used: `BINDINGSERVICE` and `BINDINGMETHOD`. Treat `scope-name`, `access-level`, `reserved-word`, and `variable-name` values as case-insensitive, and match `feature-name` values case-insensitively.

`hidl` applies to .ecm scopes (`ESDLSERVICE`, `ESDLMETHOD`), while `esdllib` additionally supports DESDL binding scopes (`BINDINGSERVICE`, `BINDINGMETHOD`).

```
access-level ::= ( 'ACCESS' | 'READ' | 'WRITE' | 'FULL' )

reserved-word ::= scope-name | access-level | 'NONE' | 'DEFERRED'

feature-name ::= feature-name-part+

feature-name-part ::= feature-name-char+ | '${' variable-name '}'
```

Define `feature-name-char` as any character except whitespace, `,`, `"`, `!`, or `:`. We also require `feature-name` to not be a reserved-word.

```
variable-name ::= feature-name-char+
```

Variable names are case-insensitive and must be defined by the invoking tool. In ESP/DESDL contexts, the commonly defined variables are `SERVICE` and `METHOD`.

See Also: ESPservice, ESPmethod