Knowledgebase > Miscellaneous > CEL functions

CEL functions

Theresa Henze - 2026-02-12 - Miscellaneous

What is CEL (Common Expression Language)?

CEL is a simple yet powerful language for defining expressions and rules. It is often used in applications to implement custom logic without the need to write complex code. CEL is especially useful in scenarios where flexibility and adaptability are required.

Bare.ID uses CEL in various areas:

- User Profile Validator (“CEL Program”)
- Login Provider Mapper (“CEL Attribute Mapper”)
- Application Mapper
  - for OIDC (“CEL claim”)
  - for SAML (“CEL claim”)

General Helper Functions:

- `base64.encode(input: string) -> string`

  Returns the Base64-encoded value of the input.

- `base64.decode(input: string) -> string`

  Returns the decoded value of the Base64-encoded input.

The following sections contain examples of using CEL in these different areas.

User Profile Validator

Validation functions use CEL expressions to check input values and report errors.

Input values:

- `value: string`: The input value to be validated.
- `field: string`: The name of the field being validated.

Return value:

- `ValidatorResponse`: An object containing a list of validation errors (`errors`). If the list is empty, the validation was successful.

Helper functions:

- `errorCase(condition: bool, message_key: string) -> ValidatorResponse`

  Returns a `ValidatorResponse` with a `ValidationError` if the condition is true, otherwise an empty `ValidatorResponse`.

- `errorCases(conditions: map<string, bool>) -> ValidatorResponse`

  Returns a list of `ValidationError` objects for the conditions that are true.

  Note: Validation functions throw errors with a translation key. This key must be defined in the translations so that the error message is displayed correctly for the available languages. See the manual, chapter Translations.

Examples:

- This triggers an `input.not.test` error if the input value is not `test`.

  ```
  ValidatorResponse{
      errors:
      [
          value == 'test' ? ValidationError{} : ValidationError{message_key:
  'input.not.test'}
      ]
  }
  ```

- Short notation: This triggers an `error.expected.not.test` error if the input value is `test`.

  ```
  errorCase(value == 'test', 'error.expected.not.test')
  ```

- Multiple errors: This triggers errors if the input value
  - is less than or equal to 5, or
  - is greater than or equal to 10.

  ```
  errorCases({'error.lesser.than.expected': int(value) <= int(5),
  'error.greater.than.expected': int(value) >= int(10)})
  ```

Login Provider Mapper

Pre-processing

This defines how the username and email address for login are selected from the data provided by the external provider. This ensures that the correct information is used for login.

This happens before the “First Broker Login” flow is called.

Input values:

- `brokered_identity_context: BrokeredIdentityContext`: The context object containing information from the external identity provider.

  ```
  BrokeredIdentityContext {
      string id
      string username
      string model_username
      string email
      string first_name
      string last_name
      string broker_session_id
      string broker_user_id
      String token
      google.protobuf.Struct context_data
  }
  ```

Return value:

- `IdentityProviderPreprocessorResponse`: An object containing the username (`username`) and email address (`email`) of the user.

  ```
  IdentityProviderPreprocessorResponse{
      string username
      string email
  }
  ```

Examples:

- Mapping the username and email address:

  ```
  IdentityProviderPreprocessorResponse{
      username: 'username',
      email: 'test@example.com'
  }
  ```

Mapping attributes and assigning roles and groups

Input values:

- `brokered_identity_context: BrokeredIdentityContext`: The context object containing information from the external identity provider.

  ```
  BrokeredIdentityContext {
      string id
      string username
      string model_username
      string email
      string first_name
      string last_name
      string broker_session_id
      string broker_user_id
      String token
      google.protobuf.Struct context_data
  }
  ```

Return value:

- `IdentityProviderMapperResponse`: An object containing the attributes to be mapped.

  ```
  IdentityProviderMapperResponse {
      AttributeEntry[] attributes
      string[] role_uuids
      string[] group_uuids
  }
  AttributeEntry {
      string key
      string[] values
  }
  ```

Helper functions:

- `groups.listByName(search: string) -> GroupModel[]`

  Returns a list of groups matching the search term.

  ```
  GroupModel {
      string id
      string name
      string full_path
  }
  ```

Examples:

- Mapping multiple attributes and assigning roles and groups:

  ```
  IdentityProviderMapperResponse{
      attributes: [
          AttributeEntry{key: 'my-attribute', values: ['val1', 'val2']},
  ```

```
        AttributeEntry{key: 'broker-attribute', values:
[brokered_identity_context.id]}
    ],
    role_uuids: ['role-uuid-1'],
    group_uuids: ['group-uuid-1']
}
```

- Notation with helper function:

  ```
  IdentityProviderMapperResponse{
      group_uuids: groups.listByName('editor').map(g, g.id)
  }
  ```

Application Mapper

OIDC

Mapper for OpenID Connect (OIDC) claims: This defines how user attributes are embedded in the OIDC tokens issued to applications. This allows applications to receive the information they need about the user.

Input values:

- `user_session: UserSessionModel`: The user session model containing information about the signed-in user.

  ```
  UserSessionModel {
      UserModel user
  }
  UserModel {
      string id
      string username

      GroupModel[] groups
  }
  GroupModel {
      string id
      string name
      string full_path
  }
  ```

Return value:

- `OIDCProtocolMapperResponse`: An object containing the value of the attribute (`claim_value`).

  ```
  OIDCProtocolMapperResponse{
      string claim_value
  }
  ```

Examples:

- Mapping a boolean value:

  ```
  OIDCProtocolMapperResponse{claim_value: has(user_session.user.username)}
  ```

- Mapping a numeric value:

  ```
  OIDCProtocolMapperResponse{claim_value: size(user_session.user.username)}
  ```

- Mapping a text value:

  ```
  OIDCProtocolMapperResponse{claim_value: user_session.user.id}
  ```

- Mapping a list of values:

  ```
  OIDCProtocolMapperResponse{claim_value: [user_session.user.id,
  user_session.user.username]}
  ```

- Mapping a map or JSON object:

  ```
  OIDCProtocolMapperResponse{claim_value: {'id': user_session.user.id, 'username':
  user_session.user.username}}
  ```

SAML

Mapper for SAML attributes and NameID: This defines how user attributes are embedded in the SAML assertions issued to applications. This allows applications to receive the information they need about the user.

Input values:

- `user_session: UserSessionModel`: The user session model containing information about the signed-in user.

  ```
  UserSessionModel {
          UserModel user
  }
  UserModel {
          string id
          string username

          GroupModel[] groups
  }
  GroupModel {
          string id
          string name
          string full_path
  }
  ```

Return value:

- `SAMLProtocolMapperResponse`: An object that contains
  - **either** the value of the attribute (`attribute_value`)
  - **or** the value for the NameID (`mapper_name_id`).

  ```
  SAMLProtocolMapperResponse{
      oneof(
          string attribute_value
          string mapper_name_id
      )
  }
  ```

Examples:

- Mapping a boolean value:

  ```
  SAMLProtocolMapperResponse{attribute_value: 'test'}
  ```

- Mapping the NameID:

  ```
  SAMLProtocolMapperResponse{mapper_name_id: 'test'}
  ```