CEL Functions

Theresa Henze - 2026-11-05 - Verschiedenes

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”)

> **Note**: This feature may need to be activated for your account and may not be included in your subscription. If you have any questions, please contact Customer Support.

General Helper Functions

There are macros from the CEL language definition as well as the following commonly used CEL extensions available:

Area	Functions
Strings	`charAt()`, `indexOf()`, `lastIndexOf()`, `trim()`, `replace()`, `reverse()`, `split()`, `substring()`, `join()`, `upperAscii()`, `lowerAscii()`, `quote()`
Lists	`slice()`, `flatten()`, `range()`, `distinct()`, `reverse()`, `sort()`, `sortBy()`, `last()`, `first()`
Sets	`contains()`, `intersects()`
Regex	`replace()`, `extract()`, `extractAll()`, `matches()`
Mathematical Functions	`greatest()`, `least()`, `bitAnd()`, `bitXor()`, `bitNot()`, `bitShiftLeft()`, `bitShiftRight()`, `ceil()`, `floor()`, `round()`, `trunc()`, `abs()`, `sign()`, `isInf()`, `isNaN()`, `isFinite()`, `sqrt()`
Optional	`has()`
Bind	`bind()`
Comprehensions	`all()`, `exists()`, `existsOne()`, `transformList()`, `transformMap()`, `transformMapEntry()`

Additionally, the following functions are available in all CEL expressions. These tables are automatically generated from the source code and are therefore always up to date — even newly added functions appear here automatically, without the need to update this page.

`base64.decode`

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

Decodes a standard base64 string back to its original UTF-8 string.

`base64.encode`

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

Encodes a UTF-8 string to standard base64.

`base64.encode(bytes) -> string`

Encodes a byte string to standard base64.

`csv.parse`

`csv.parse(string, string) -> list<map<string, string>>`

Parses CSV text into a list of maps with one map per line, keyed by the column names of the first (header) line. Values are trimmed, blank lines are skipped. The second argument is the field separator: a single character, or `tab`.

Example: `csv.parse('id;name\n1;Paul', ';')` → `[{'id': '1', 'name': 'Paul'}]`

`errorCase`

`errorCase(bool, string) -> cel.v1.ValidatorResponse`

Adds a single validation error with the given message key if the boolean condition is true.

> **Note:**: This function is only available for the “User Profile Validator”.

`errorCases`

`errorCases(map<string, bool>) -> cel.v1.ValidatorResponse`

Adds one validation error per map entry whose message-key entry key has a true boolean condition value.

`file.read`

`file.read(string, string) -> string`

Reads a text file from the server’s import directory, e.g. a mounted file share. The path is relative to the realm’s import directory; the second argument is the file’s charset, e.g. `UTF-8` or `windows-1252`. Fails if the file is not valid in that charset.

Example: `csv.parse(file.read('hr/employees.csv', 'UTF-8'), ';')`

`UserModel.getFirstAttribute`

`cel.v1.UserModel.getFirstAttribute(string) -> string`

Returns the first value of the given user attribute, or an empty string if the attribute is not set.

`groups.getById`

`groups.getById(string) -> optional_type<cel.v1.GroupModel>`

Looks up a single group by its ID. Returns an empty optional if no such group exists.

`groups.getFirstByName`

`groups.getFirstByName(string) -> optional_type<cel.v1.GroupModel>`

Returns the first group whose name contains the given substring, or an empty optional if none match.

`groups.listByName`

`groups.listByName(string) -> list<cel.v1.GroupModel>`

Searches groups (including subgroups) whose name contains the given substring.

`guid.toByteArray`

`guid.toByteArray(string) -> bytes`

Converts a UUID string to its 16-byte binary representation (RFC 4122 mixed-endian byte order used by Microsoft-style GUIDs).

`http.call`

```
http.call(cel.v1.HttpServiceCallRequest) -> cel.v1.HttpServiceCallResponse
```

Performs an outbound HTTP request (GET/POST/PUT/PATCH/DELETE) and returns the response status, headers, and body.

`json.decode`

```
json.decode(string) -> google.protobuf.Value
```

Parses a JSON string into a dynamic value (object, array, string, number, bool, or null).

Example: `json.decode('{"key": "value"}')` → `map('key': 'value')`

`json.encode`

```
json.encode(dyn) -> string
```

Serializes a value (map, list, string, number, bool, or null) to a compact JSON string.

Example: `json.encode(map('key': 'value'))` → `'{"key": "value"}'`

`query`

```
cel.v1.SqlDatasource.query(string) -> cel.v1.SqlQueryResult
```

Runs a SQL query against the datasource returned by `sql()` and returns the result rows.

Example:

```
sql("myDatasource").query("SELECT id, name FROM my_table WHERE status = 'active'")
```

```
cel.v1.SqlDatasource.query(string, list<dyn>) -> cel.v1.SqlQueryResult
```

Runs a parameterized SQL query (positional `?` placeholders bound from the given list) against the datasource returned by `sql()` and returns the result rows.

Example:

```
sql("myDatasource").query("SELECT id, name FROM my_table WHERE status = ?",
["active"])
```

`roles.getById`

```
roles.getById(string) -> optional_type<cel.v1.RoleModel>
```

Looks up a single realm or client role by its ID. Returns an empty optional if no such role exists.

`roles.getFirstByName`

```
roles.getFirstByName(string) -> optional_type<cel.v1.RoleModel>
```

Returns the first realm or client role whose name contains the given substring, or an empty optional if none match.

`roles.listByName`

```
roles.listByName(string) -> list<cel.v1.RoleModel>
```

Searches realm and client roles whose name contains the given substring.

`secret`

```
secret(string) -> string
```

Returns the value of the named CEL secret (configured per-realm via the Bare.ID app-frontend), or fails if no secret with that name is configured.

`secretOrDefault`

```
secretOrDefault(string, string) -> string
```

Returns the value of the named CEL secret, or the given fallback if no secret with that name is configured.

`sql`

```
sql(string) -> cel.v1.SqlDatasource
```

Resolves a named SQL datasource (configured on the Keycloak server) so it can be queried with `query()`. Fails if no active datasource with that name exists.

Example:

```
sql("myDatasource").query("SELECT id, name FROM my_table WHERE status = ?",
["active"])
```

`url.decode`

```
url.decode(string) -> string
```

Decodes percent-encoded (%XX) sequences. A + is left as-is; use `url.decodeQuery` to decode form-encoded query parameters.

`url.decodeQuery`

```
url.decodeQuery(string) -> string
```

Decodes a query parameter name or value: both %XX sequences and + as a space.

`url.encode`

```
url.encode(string) -> string
```

Percent-encodes a value so it can be used as a single component of a URL (RFC 3986), e.g. a path segment. A space becomes `%20`.

`url.encodeQuery`

```
url.encodeQuery(string) -> string
```

Percent-encodes a value for use as a query parameter name or value (application/x-www-form-urlencoded, i.e. a space becomes +).

Example: `url.encodeQuery('a b')` → `'a+b'`

`users.getById`

```
users.getById(string) -> optional_type<cel.v1.UserModel>
```

Looks up a single user by its ID. Returns an empty optional if no such user exists.

`users.getByLoginName`

```
users.getByLoginName(string) -> optional_type<cel.v1.UserModel>
```

Looks up a single user by their login name (email if the realm uses email as username, otherwise username). Returns an empty optional if no such user exists.

`validatedAttributeEntry`

```
validatedAttributeEntry(string, list<string>) -> cel.v1.AttributeEntry
```

Shortcut helper function to use in CEL Validators.

Runs the configured user-profile validators for the given attribute key against the provided values, throwing on failure, and returns a validated AttributeEntry.

> **Warning:** Do not encode API keys or other secrets directly in CEL expressions, because mapper and validator configurations are readable by everyone with read access to the instance, and they end up in exports and support tickets. Store them as named secrets under “System / Advanced → Secrets” instead and reference them with `secret()` / `secretOrDefault()`. See the manual, chapter Secrets.

User Profile Validator

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

Available Variables and Return Value:

ValidatorRequest — User Profile Field Validator

Field	Type	Description
value	string	
field	string	

ValidatorResponse

Field	Type	Description
errors	ValidationError[]	

ValidationError

Field	Type	Description
message_key	string	

The helper `errorCase`/`errorCases` are listed in the section “General Helper Functions”.

> **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:

Full notation: throws error `input.not.test` if the input value is not `test`.

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

Shorthand notation: throws error `error.expected.not.test` if the input value is `test`.

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

Multiple errors: throws if the input value is less than or equal to 5, or 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.

Available Variables and Return Value:

IdentityProviderPreprocessorRequest

Field	Type	Description
`brokered_identity_context`	`BrokeredIdentityContext`	
`claims`	`google.protobuf.Struct`	The upstream claims merged from the access token, the ID token and the

UserInfo claim set - the same three sources Keycloak's own `oidc-user-attribute-idp-mapper` reads. On a conflicting top level claim the access token wins over the ID token, which wins over UserInfo. (Keycloak resolves each claim path individually, so for a nested path it may fall back to a lower precedence source where this merge lets the higher one win.) |

IdentityProviderPreprocessorResponse

Field	Type	Description
`username`	`string`	evaluated in preprocessFederatedIdentity before first broker login flow
`email`	`string`	evaluated in preprocessFederatedIdentity before first broker login flow

BrokeredIdentityContext

Field	Type	Description
`id`	`string`	
`username`	`string`	
`model_username`	`string`	
`email`	`string`	
`first_name`	`string`	
`last_name`	`string`	
`broker_session_id`	`string`	
`broker_user_id`	`string`	
`token`	`string`	
`context_data`	`google.protobuf.Struct`	IdentityProviderModel idp_config = 10;

IdentityProvider idp = 11; The raw context data Keycloak keeps on the brokered identity, e.g. `VALIDATED_ACCESS_TOKEN`, `VALIDATED_ID_TOKEN`, `UserInfo`, `FEDERATED_ACCESS_TOKEN_RESPONSE` or `user.attributes.<name>`. Entries whose value cannot be represented as JSON (such as a SAML assertion) are omitted. |

Beispiele:

Maps the username and email address.

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

Mapping attributes and assigning roles and groups

Available Variables and Return Value:

IdentityProviderMapperRequest — Identity Provider Mapper

Field	Type	Description
`brokered_identity_context`	`BrokeredIdentityContext`	roles and groups are provided as functions
`claims`	`google.protobuf.Struct`	The upstream claims merged from the access token, the ID token and the

UserInfo claim set - the same three sources Keycloak's own `oidc-user-attribute-idp-mapper` reads. On a conflicting top level claim the access token wins over the ID token, which wins over UserInfo. (Keycloak resolves each claim path individually, so for a nested path it may fall back to a lower precedence source where this merge lets the higher one win.) |

IdentityProviderMapperResponse

Field	Type	Description
`attributes`	`AttributeEntry[]`	

Field	Type	Description
role_uuids	string[]	
group_uuids	string[]	

BrokeredIdentityContext

Field	Type	Description
id	string	
username	string	
model_username	string	
email	string	
first_name	string	
last_name	string	
broker_session_id	string	
broker_user_id	string	
token	string	
context_data	google.protobuf.Struct	IdentityProviderModel idp_config = 10;

IdentityProvider idp = 11; The raw context data Keycloak keeps on the brokered identity, e.g. `VALIDATED_ACCESS_TOKEN`, `VALIDATED_ID_TOKEN`, `UserInfo`, `FEDERATED_ACCESS_TOKEN_RESPONSE` or `user.attributes.<name>`. Entries whose value cannot be represented as JSON (such as a SAML assertion) are omitted. |

AttributeEntry

Field	Type	Description
key	string	
values	string[]	

Helper functions for looking up groups (`groups.*`), roles (`roles.*`), and users (`users.*`) are listed above in the section “General Helper Functions”.

Examples:

Maps multiple attributes and assigns 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']
}
```

Using a helper function to look up groups by name.

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

LDAP Attribute Mapper

Mapper for LDAP-connected directories: This defines how a single attribute is synchronized between the LDAP directory and the local Bare.ID user. There are two directions, each configurable with its own CEL expression.

LDAP → Local

Evaluated when importing or reconciling a user from the LDAP directory to determine the value for the local user attribute.

Available Variables and Return Value:

LdapUserAttributeToLocalMapperRequest

Field	Type	Description
attribute_value	string	
realm	RealmModel	
is_create	bool	

LdapUserAttributeToLocalMapperResponse

Field	Type	Description
value	string	

RealmModel

Field	Type	Description
id	string	
name	string	

Example:

Copies the LDAP value unchanged.

```
LdapUserAttributeToLocalMapperResponse{value: attribute_value}
```

Local → LDAP

Evaluated when writing a local user attribute back to the LDAP directory.

Available Variables and Return Value:

LdapUserAttributeFromLocalMapperRequest

Field	Type	Description
attribute_value	string	
local_user	UserModel	
realm	RealmModel	

LdapUserAttributeFromLocalMapperResponse

Field	Type	Description
value	string	

UserModel

Field	Type	Description
id	string	
username	string	
attributes	AttributeEntry[]	
groups	GroupModel[]	

AttributeEntry

Field	Type	Description
key	string	
values	string[]	

GroupModel — OIDC and SAML Protocol Mappers

Field	Type	Description
id	string	
name	string	
full_path	string	

RealmModel

Field	Type	Description
id	string	
name	string	

Example:

Copies the local value unchanged.

```
LdapUserAttributeFromLocalMapperResponse{value: attribute_value}
```

Application Mapper

OIDC

Mapper for OpenID Connect (OIDC) claims: This defines how user attributes are embedded in the OIDC token issued to applications. This ensures that applications receive the necessary information about the user.

Available Variables and Return Value:

OIDCProtocolMapperRequest

Field	Type	Description
user_session	UserSessionModel	ProtocolMapperModel mappingModel = 1;

ClientSessionContext clientSessionCtx = 4; |

OIDCProtocolMapperResponse

Field	Type	Description
claim_value	google.protobuf.Value	Number value means long value, no floating point values possible

UserSessionModel

Field	Type	Description
user	UserModel	

UserModel

Field	Type	Description
id	string	
username	string	
attributes	AttributeEntry[]	
groups	GroupModel[]	

GroupModel — OIDC and SAML Protocol Mappers

Field	Type	Description
id	string	
name	string	
full_path	string	

AttributeEntry

Field	Type	Description
key	string	

Field	Type	Description
`values`	`string[]`	

Examples:

Maps a boolean value.

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

Maps a numeric value.

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

Maps a text value.

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

Maps a list of values.

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

Maps a map/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 attribute statement issued to applications. This ensures that applications receive the necessary information about the user.

`SAMLProtocolMapperResponse` contains **either** the value of the attribute (`attribute_value`) **or** the value for the NameID (`mapper_name_id`).

Available Variables and Return Value:

SAMLProtocolMapperRequest

Field	Type	Description
`user_session`	`UserSessionModel`	string nameIdFormat = 1;

ProtocolMapperModel mappingModel = 2; KeycloakSession session = 3;

AuthenticatedClientSessionModel clientSession = 5; \| |

SAMLProtocolMapperResponse

Field	Type	Description
`attribute_value`	`string`	
`mapper_name_id`	`string`	

UserSessionModel

Field	Type	Description
user	UserModel	

UserModel

Field	Type	Description
id	string	
username	string	
attributes	AttributeEntry[]	
groups	GroupModel[]	

GroupModel — OIDC and SAML Protocol Mappers

Field	Type	Description
id	string	
name	string	
full_path	string	

AttributeEntry

Field	Type	Description
key	string	
values	string[]	

Examples:

Maps an attribute value.

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

Maps the NameID.

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