Data Standards Body
Technical Working Group

Decision Proposal 119 – Enhanced Error Handling – Data Structure

Contact: *Mark Verstege*
Publish Date: *21st May 2020*
Feedback Conclusion Date: *19th June 2020*

Context

Currently, the CDR data standards and the CDR Register adopt elements of the JSON API error code structure. Within this structure, the data standards define a single core CDR Error Code. Whilst maintaining consistency across both is essential, there is a need to apply consistent conventions to identify ecosystem error codes. In conjunction, a review of whether the current error response structure is sufficient to support data recipients has been considered.

With an implementation base that has involved integration across initial data holders and data recipients, lessons have been learnt and scenarios identified where improved error handling will improve consistency across a wider implementation base. This is further supported by more non-major data holders and ecosystem participants designing their solutions.

Some of the issues that have been raised by the community include:

- the format of error responses is inconsistent across participants making it difficult to interpret errors
- there is no guidance on the format of error codes for implementers
- desire for increase consistency and uniformity across implementations
- recommendation to establish a core collection of error codes that apply to all participants
- defining consistent naming conventions for error codes to reduce ambiguity and avoid different participants using the same error code to mean completely different errors

This proposal specifically relates to the data structure of the payload for error responses within the Data Standards *and* CDR Register. It is considered in conjunction with a group of decision proposals under the Enhanced Error Handling problem space:

- **Decision Proposal 119 - Enhanced Error Handling Payload Conventions (this proposal)**
- Decision Proposal 120 - CDR Error Codes for Enhanced Error Handling
- Decision Proposal 121 - Application of existing HTTP Error Response Codes to Enhanced Error Handling
- Decision Proposal 122 - Extension of Supported HTTP Response Codes for Enhanced Error Handling

Decisions To Be Made

- Decide on a standard naming convention for error codes.
- Decide whether any other changes to the error data structure are required to facilitate enhanced error handling between participants.
- Decide whether data recipients need to know about the list of supported error codes via an error catalogue publishing solution.

Current recommendation

The current recommendation of the Data Standards Body is the adoption of a well-defined error code format that defines an error code namespace enumeration. This format will reserve a code space for core CDR errors and also designate bounded error code spaces for each participant.

No other changes are foreseen to the error code structure based on current community feedback. This position may change as part of consultation if there is clear need from data recipients to be more programmatically deterministic when errors are encountered.

The purpose of the error code namespace enumeration is to map to HTTP Response Codes and give explicit guidance when core errors are encountered that there is a well-known error code to handle that error.

Of the options represented below this includes:

- *Option 3 – Error Code Enumerations*. This option provides a better developer experience
- *Option 1 – No change*. Other than changing the error code format, the data model is left as is **unless** the is a clear need from data recipients to programmatically handle field-level error handling.
- *Option 1 – No Discovery*. If participants — particularly data recipients — see the need for a discovery service to better develop working applications against, this would be considered.

Given there is only one core CDR Error Code currently defined in the data standards, this would result in a minor change to the data standards but create greater long-term predictability in the ecosystem.

The DSB is seeking feedback on this recommendation. In particular, feedback from data recipients on what granularity of detail they require for errors to be handled gracefully and programmatically in their software.

Design Considerations

When reviewing error responses, a key consideration is given to the needs of the data recipients: what do ADRs need in order to handle error scenarios gracefully, convey sufficient meaning to the consumer and be able to resolve errors when they are encountered.

Further to this, consideration has been given to the approaches adopted by the UK Open Banking standards.

Design Principles

1. The design should result in consistency across the CDR ecosystem.
2. The design should cover all flows and interactions (e.g. authorisation flows, dynamic client registration, API consumption flows, registry interactions, etc.).
3. Error handling should be testable via conformance test suites.
4. Errors should convey enough meaning to allow clients to programmatically handle different errors.
5. Errors should allow clients to appropriately tailor the consumer experience.

Outcomes

- Consistency and robustness for all participants
- Reduced customisation for participant-specific error scenarios

- Reduced processing load on data recipients
- Flexibility to define rich errors that are descriptive and clearly communicated
- Reduced ambiguity in interpretation that leads to the chance of software issues

Requirements

1. Error handling should be defined consistently between the CDR Register and Data Standards
2. When errors are common to all implementations, they should be clearly identifiable as Data Standards error codes
3. When errors are specific to the participant, the errors should identify the participant
4. Participant-specific error codes should be consistent within a Data Recipient or Data Holder implementation.
   In other words, if a Data Recipient has multiple software products in market, they should use the same error codes consistently across
5. The specification of custom error codes should be done in a way that reduces the chance of collision across data holders.
   In other words, we want to avoid having situations such as Bank ABC using "ERR-002: Account Unavailable" whilst Bank Happy uses "ERR-002: Integer must be greater than zero(0)". This fails to avoid rules of collision avoidance and increases ambiguity in the ecosystem.
6. Where a Data Holder has multiple in-market brands, they should represent their error codes within a namespace that is consistent with how they model their data holder brands in the CDR Register.
   In other words, if the Data Holder maintains a sister/child brand as a separate data holder, their error codes should be represented under these unique contexts. However, if they represent their brands within one data holder, they should represent the error codes as a single data holder solution. This doesn't preclude having brand-specific or product-specific error codes where applicable.

Options Identified

Error Codes

Option 1 – No format is defined

This is the situation today where there is no strict format imposed for error codes. Instead, error codes are defined on an ad-hoc basis when they are required. Each participant can define their own error code scheme. Typically, it would be assumed that solutions would gravitate towards a few common patterns, but consistency isn’t guaranteed.

Option 2 – Numeric Error Ranges

Errors are defined as a set of numeric codes. A range is reserved for all core CDR error codes however all participant-specific error codes are unbounded outside the reserved range. It would be expected that an auto-incrementing approach would be likely.

For example:

- Reserved namespace format: <code 000 to 500>
- Custom namespace format: <code 501 to 999>

Option 3 – Error Codes Enumerations

One key issue with error codes as they are represented today is there is high likelihood of multiple implementations choosing the same errors codes but describing different errors. Adopting enumeration along with reserved namespaces for core CDR error codes and each participant would make it possible to distinguish error codes.

Error codes are described using a participant prefix, an error category and error code. The error category is used to represent a categorisation of errors which has the advantage of grouping the types of error - this conveys additional meaning and in future may make it possible for the ACCC to request reporting of errors be broken down into error categories for more detailed understanding of ecosystem issues and trends.

The advantage of this format is that it closely aligns to the UK OBIE approach. Format:

- Reserved namespace format: <reserved_namespace>.<error_category_code>.<code>
- Custom namespace format: <participant_id>.<error_category_code>.<code>

Core CDR Error Code Format Fragment Definitions

The format proposed below is a decimal separated set of defined namespace segment that each describe an aspect of the error encountered.

Segment	**Format**	**Requirement**	**Description**
Reserved Namespace	AU.CDR	Mandatory. Denotes a CDR Error	Describes the error falls within the CDR regime of the AU jurisdiction. Mandatory for reserved CDR error codes that apply to all participants. These are the official CDR Error codes.
Error Category Code	string (40)	Conditional	Category for the type of error. • Optional for participant-specific error codes. • Mandatory for core CDR error codes. • Error Category Code of "Register" is reserved for use by the CDR Register only
Code	String (40)	Mandatory	Human readable code denoting the error encountered. • Codes must be descriptive. • Codes must be max 40 characters • If a numeric code is used, it should be 4-digit numeric value with leading zeros.

Participant-specific Code Format Fragment Definitions

The format proposed below is a decimal separated set of defined namespace segment that each describe an aspect of the error encountered.

Segment	Format	Requirement	Description
Participant ID	string(20)	Mandatory	The participant common name or short name defined by the participant. Assume that public entities would align to their ASX Issuer Name or similar. Examples: "WBC", "ANZ", "XERO", "ABC1". But other examples are valid: "Frollo", "Intuit", "Westpac".
Error Category Code	string (40)	Conditional	Category for the type of error. • Optional for participant-specific error codes. • Mandatory for core CDR error codes. • Error Category Code of "Register" is reserved for use by the CDR Register only
Code	String (40)	Mandatory	Human readable code denoting the error encountered. • Codes should be descriptive. • If a textual code is used, it must be max 40 characters • If a numeric code is used, it should be 4-digit numeric value with leading zeros.

Worked Examples

```
AU.CDR.Register.InvalidADRStatus
AU.CDR.Missing.Header
AU.CDR.Entitlements. AccountConsentWithdrawn
AU.CDR.ConstraintViolation.PageSizeTooLarge
AU.CDR.Register.MetadataUpdated
Westpac.HeaderValidationFailure.999
ANZ.FieldType.IncorrectAccountIdFormat
TMBL.UnexpectedError.057
XERO.ERR-010
Moneytree.112
```

Data model

In reviewing enhanced error handling, it is reasonable to look at the error response structure itself and whether it conveys the detail required for participants to interpret errors, to be able programmatically handle errors and triage issues.

Reasons for adopting new parameters may include:

- Pointing to the source of an error such as body parameter or URL field
- Client software needs a more structured format to describe where the error was encountered, the nature of the error and the allowable values
- Client software is primarily designed for real-time programmatic handling of errors (e.g. retry) rather than offline triage and code release
- The level of detail currently catered for is insufficient to differentiate errors or insufficient to triage

The key candidate for changing the error payload is the JSONAPI "source" parameter which is used to provide a reference to the source of the error, optionally including a pointer and URI parameter

Examples where using the "source" parameter may prove useful include:

- The recipient requests a negative version for the Get Products API (e.g. x-v=-56)
- The accountId requested for the Get Account Balance API is not a legal ASCII string
- The recipient makes a call to the Get Payees API with a non-numeric page (e.g. page=foo)

In these instances, a data holder can direct the client to the source of the problem that resulted in an error where it may not otherwise be obvious.

Error "source" is particularly useful for data sent in a message body such as a POST or PUT operation. Especially where a request has multiple parameter validation issues. In these instances, knowing what data caused issues can be used for programmatic handling of the error (e.g. for retry scenarios) and for general logging and issue identification on the consumer system.

Option 1 – No change

Keep as-is and consider the introduction of a source parameter when moving into write operations for resources. This would allow for use cases to be documented and tested against the need for this level of detail when requirements are better defined. Introducing the source parameter later on is still possible including any optional/mandatory consideration for write operations.

In this option, the "description" field is considered sufficient and it can also be used by a data holder to provide this level of description, in a less structured approach, thus avoiding data model changes.

If the source parameter is required for read-only operations not just body data, then later adoption will require more change management and a transition plan.

Option 2 – Adopt JSONAPI error "source" parameter

Using the "source" parameter defined in the JSONAPI spec (https://jsonapi.org/format/#error-objects) and require Data Holders to specify. a pointer the specific field that encountered the error. In this option, the "description" field can be used to describe the error encountered and the "source" parameter can be used to define the field in a structured format. This would allow for better handling of errors within the consumer experience.

Example

Current error

```
{
  "errors": [
    {
      "code": "0001 - Account not able to be found",
      "title": "Invalid account",
      "detail": "00284ae747",
      "meta": {}
    },
    {
      "code":   "422",
      "title": " Invalid field format",
      "detail": "is-owned must be a boolean value"
    },
    {
      "code":   "406",
      "title": "Unsupported version",
      "detail": "Version ‘6’ is not supported.
                 Maximum version supported is ‘2’."
    }
  ]
}
```

Using error “source”

```
{
  "errors": [
    {
      "code":  "0001 - Account not able to be found",
      "title": "Invalid account",
      "detail": "00284ae747",
      "source": { "pointer": "/data/accountIds/[1]" }
    },
    {
      "code":   "422",
      "title": "Invalid field format",
      "detail": "Must be a boolean value but received a number",
      "source": { "parameter": "is-owned" }
    },
    {
      "code":  "406",
      "title": "Unsupported version",
      "detail": "Version ‘6’ is not supported.
                 Maximum version supported is ‘2’",
      "source": { "parameter": "x-v" },
    }
 ]
}
```

Error code discovery

Another pain point raised by the community is not knowing what errors will be encountered and how each Data Holder will represent there set of errors. As the catalogue of errors that a participant caters for grows, there may be benefit for each implementation to publish the list of error codes they cater for via some form of discovery service.

Allowing participants to know ahead of time the custom errors that could be encountered when connecting to another participant would produce more robust and reliable solutions.

As participants expand their implementations into the competitive space having each participant publish the catalogue of errors their software produced would provide for less reactive development to handle new error conditions.

This would allow user experiences to be designed to either avoid the errors altogether, or to be handled in a way that would improve the user experience when the only other option would be to fail and possibly not complete the action.

Option 1 – No discovery service

In this option, it is deemed that there is no ecosystem benefit for participants to publish their supported error code catalogue. Instead, new errors would be known only when the scenario is encountered in production.

With this option, it would not be any different to today. Data recipients could only cater for new errors after they are encountered in production, or they are defined as core CDR Error Codes.

Option 2 – Discoverable via CDR Register Participant Metadata

Participants supply their error catalogue in their software metadata held by the ACCC. Participants would do so via a set of management APIs hosted by the CDR Register allowing the participant to maintain a current list of error codes.

Alternatively, a low-tech option where participants upload their error list periodically to the CDR Register may be desirable. In this respect, the CDR Register may offer a way for other participants to download error lists.

Option 3 – Discoverable via Participant API

A discovery service is still provided but the onus shifts to participants hosting a discoverability API allowing any other participant to call and obtain the list of know participant error codes that could be encountered.

It would only allow for ADR to DH or DH to ADR discoverability (e.g. a Data Holder couldn't call another Data Holder) whereas Option B would allow any participant to collect the error code list of any other participant.