Guidelines for Response Status Codes and Resource Exceptions

Introduction

This page provides guidelines for correct handling of response status codes when implementing or consuming resources. This is a very important aspect of the REST-ful remote API for the ABAP Development Infrastructure and the guidelines should be followed by all resource implementations.

In general, we try to stick to the HTTP standard as much as possible. Therefore, the standard HTTP status codes should be used as they are defined in RFC2616. The section below provides a summary of the relevant status codes in our context.

Modern programming languages usually handle error situations by raising exceptions. We follow this approach and have defined an exception hierarchy which should be used by resource implementations. For every relevant error status, there is a corresponding exception class. The resource framework handles resource exceptions raised by resource implementations as follows:

1. the response status code of the exception instance is set in the response
2. the exception instance data is serialized in a specific format and set in the response body

On the client-side, the resource framework handles the exception case according to the response status and deserializes the response body, so that the detailed exception information can be evaluated.

Resource implementations should either use the exception classes of the framework directly and only set domain-specific messages or create domain-specific exception classes which must inherit from the correct framework exception class. Exception classes must always be used in a way that the corresponding HTTP response status code matches the specific error case.

Response Status Overview

The HTTP Standard defines the status code as a 3-digit integer where the first digit defines the status category. The categories are defined as follows:

1. 1xx - informational; not used for AIE
2. 2xx - success
3. 3xx - redirection; not (yet) used for AIE
4. 4xx - client error
5. 5xx - server error

Constants for all HTTP Status codes are defined in

- class *cl_rest_status_code* for the ABAP back-end
- class *HttpUrlConnection* for the Eclipse client

> The **minimum** requirement for resource exception handling is to distinguish correctly between **client** errors and **server** errors.

Some of the standard HTTP status codes refer to rather technical aspects which are already handled by the low-level frameworks or not used at all (such as the 401 status used for the user authentication hand-shake). The following table lists the status codes which *are* currently relevant for AIE and the corresponding exception classes.

Status Code	Description	Equivalent server exception class	Equivalent client exception class
200	OK - Success!		

201	Created - should be used as response to POST requests on a collection resource; response must contain a *Location* header in this case		
400	Bad Request: The request was invalid, e.g. query parameters or request body with invalid syntax. Clients should *not* repeat such a request.	cx_adt_res_bad_request	ResourceBadRequestException
403	Forbidden: The request is understood, but it has been refused. Examples are missing permissions or an object is read-only for any other reason.	cx_adt_res_no_access	ResourceForbiddenException
404	Not Found: The URI requested is invalid or the requested resource, such as a user, does not exist.	cx_adt_res_not_found	ResourceNotFoundException
405	Method not allowed: The resource does either not support this request method at all or is in a state where this method cannot be used (e.g. resource with that name already exists)	cx_adt_res_meth_not_supported, cx_adt_res_already_exists	ResourceMethodNotAllowedException
412	Precondition failed: used to indicate that a precondition for one of the request headers is not fulfilled. One use-case example is optimistic locking with an etag sent in an *If-Match* header.	cx_adt_res_invalid_etag	ResourcePreconditionFailedException
500	Internal Server Error: Something is broken. Please open CSN message. Depending on the use-case, you should consider to rather let it dump (so that you have something to analyze) in this case. Do *not* catch *cx_root* and raise *cx_adt_res_int_server_error* instead.	cx_adt_res_int_server_error Short Dump	ResourceException SystemFailureException

Inspired by Twitter API

Raising exceptions in resource implementations

It is important to understand that the exception handling in the resource framework does not happen within the context of one programming language. It happens on the integration layer which is used to bind different technologies together (In our case a Java based client and an ABAP backend). Thereby developers of a resource should keep the following things in mind when using service or resource exceptions:

- Exceptions are thrown on the API level. A developer who remotely calls a resource usually does not have the ability to debug the API implementation therefore clear exception messages are crucial.
- Exceptions are never allowed to leak implementation details. Potential attackers use such information like stack traces, etc. to gain knowledge about the implementation, its weaknesses and thereby can improve their attacks.
- Exceptions types are clearly connected to the domain. For example when calling the save operation on a resource a valid exception would be *"Salesorder with Id <xy> already exists"* compared to *"Duplicated key <xy>"*. As you can see the first exception is clearly connected to the sales domain compared to the second example which returns a cryptic technical exception which even reveals implementation details.

Attributes of a resource exception

The base class for all resource exceptions is *cx_adt_rest*. This class is also used as subclass for exceptions raised by the framework. Resource controller implementations should rather use the more specific exception classes outlined above (named *cx_adt_res*_*).

Every resource exception has the following attributes:

Attribute / Method	Description
get_http_status	Response status code
namespace	A namespace qualifier for the exception type; should be set to a domain-specific value when defining domain-specific exception subclasses
type	Type of the exception - could be mapped to an Exception Class on client side
subtype	More specific identifier for the reason of failure - could be mapped to a status code (of interface *org.eclipse.core.runtime.IStatus*)
message	Contains the exception message in English.

localized message	Contains the localized version of the exception message.
properties	The properties of an exception is a simple map of key/value entries which allows the developer to transfer any kind of interesting data to the receiver of the exception.

Exception format

All resource exceptions follow the same format.

Exception Format Example

```
<?xml version="1.0" encoding="UTF-8"?>
<exception>
  <namespace id="com.sap.adt"/>
  <type id="ExceptionResourceSaveFailure"/>
  <message>Class ZCL_TEST could not be saved</message>
  <localizedMessage lang="de">Klasse ZCL_TEST konnte nicht gesichert werden</localizedMessage>
  <properties>
    <entry key="URI">/sap/bc/adt/oo/classes/ZCL_TEST/includes/definitions</entry>
  </properties>
</exception>
```

The exception classes of the resource framework provide a generic text that can be overwritten by your own message. The following examples show you how you can raise resource exceptions with more specific texts than the generic ones.

Raise Exceptions with Message Parameters Using the System Variables SY-MSG*

The following example shows how you can raise the exception *CX_ADT_RES_ALREADY_EXISTS* by using the filled values of the system variables *SY-MSG**. Method *CX_ADT_REST=>CREATE_TEXTID_FROM_MSG_PARAMS( )* creates the correct T100-text-id for all T100 sub classes of exception class *CX_ADT_REST*. It uses the default values *SY-MSGID* and *SY-MSGNO* for the parameters *MSGID* and *MSGNO*.
The default values for the parameters *MSGV1* to *MSGV4* of the constructor of class *CX_ADT_REST* are *SY-MSGV1* to *SY-MSGV4*.

```
* Message statement that fill the system variables SY-MSGID, SY-MSGNO, SY-MSGTY and SY-MSGV1 to
SY-MASGV2
  message s002(OO) with 'CX_ROOT'.

* Raise exception and use the SY-MSG\*-fields as default input for the parameters TEXTID and MSGV1 to
MSGV4
  raise exception type cx_adt_res_already_exists
    exporting
      textid = CX_ADT_REST=>create_textid_from_msg_params( ).
```

If the message text of the exception is raised, its text refers to the specific message id and message number (in the example above it is message id *OO* and message number *002*). This information can be used to set an appropriate break point or to find the source code position by using the where used list of this message. Therefore it is recommended to use this way to raise exceptions with specific text instead of raising exceptions with message parameters.

The same concept can be used to forward classical exceptions which have set the values for the system variables *SY-MSG**.

```
* Call a function module that supports classical exceptions only
  call function 'MY_FUNCTION_MODULE'
    exceptions
      runtime_error = 1
      others        = 2.
  if ( sy-subrc <> 0 ).

*  Forward the message of the function modules exception
   raise exception type cx_adt_res_already_exists
     exporting
       textid = CX_ADT_REST=>create_textid_from_msg_params( ).
  endif.
```

Raise Exceptions with Message Parameters

The following example shows how you can raise the exception *CX_ADT_RES_ALREADY_EXISTS* with the text of message *OO 002* with message parameters *CX_ROOT*.

```
* Raise exception for MESSAGE 002(OO) WITH 'CX_ROOT'
  raise exception type cx_adt_res_already_exists
    exporting
      textid = CX_ADT_REST=>create_textid_from_msg_params( msgid = 'OO' msgno = '337' )
      msgv1  = 'CX_ROOT'.
```

Raise Exceptions using the text of the previous exception

You can also forward the text of another exception class (it doesn't matter whether it is a T100- or an OTR-exception class). To create a correct T100-text-id you have to call method *CX_ADT_REST=>CREATE_TEXTID_FROM_EXC_TEXT( )* and pass the exception caught by the catch-block as parameter. This method can be used for all T100-sub-classes of exception class *CX_ADT_REST*.

```
data:
    exception type ref to cx_oo_class_already_exists.

  try.
*     Calls some coding that raises CX_OO_CLASS_ALREADY_EXISTS exception if the class already exists
      class_api->create( 'CX_ROOT' ).
    catch cx_oo_class_already_exists into exception.

*     Catch the exception and forward its text in exception CX_ADT_RES_ALREADY_EXISTS
      raise exception type cx_adt_res_already_exists
        exporting
          textid   = CX_ADT_REST=>create_textid_from_exc_text( exception )
          previous = exception.        "The parameter Previous does not have to be set!
  endtry.
```

 If method *CX_ADT_REST=>CREATE_TEXTID_FROM_EXC_TEXT( )* is called, the values for the system variables *SY-MSGID* and *SY-MSGNO* will be reset to to the values before the method has been called. This method might change the values of *SY-MSGV1* ... *SY-MSGV4* because they are used by the constructor parameters *MSGV1* ... *MSGV4* of class *CX_ADT_REST*.