Article
Istvan Hahn · Oct 6, 2016 4m read

# RESTful Exception Handling

A beginner ' s guide to Exception Handling in RESTful web services. The article gives an example how the various error conditions during processing a service request can be handled.

We expect our client – server communication working in a flawless operational condition, running error free software. But we are prepared to handle exceptions. Are we? So far in the examples of the previous sessions were not. We did not care about exceptions. The result? In any error incident it took ages to figure out what the problem is and more importantly how to fix it.

Remember, REST leans on HTTP. HTTP explicitly defines how error conditions must be handled. HTTP requires to return a status in the response header and optionally supplement it by a human or robot readable error explanation in the message body. RESTful communication is not different.

When an exception caught at server side, which prevents the server to serve the client request fully, an error condition has to be reported. Reporting means setting the HTTP response status to other than 200 (OK), and dump details into the message body. It is also possible that the status code tells that the response is incomplete (206) and the body contains the fragment available. Or in other situation when the service processes the request asynchronously for the initial communication the server might return 202 (Accepted). Or as in the example class, when Ensemble fails to create the Business Service instance it returns 503 (Service is unavailable).

The next example is a variant of our ResourceMap class. We created that during the " Creating RESTful Web Service in Ensemble " or any later session. What it does is basically the following.

- The UrlMap is a jump table. It defines which static class method is to be called on receiving a request with an URL patch matching a pattern.
- The static method InvokeEnsembleService calls the service and dumps the response to the HTTP message body.

The real new in the source is the " try-catch " decoration. To those who are mostly COS programmers: it is a beginner ' s guide. I do not want to dispute neither programming styles nor performance of particular techniques.

```
Class h2.createrestfulservice.ResourceMap Extends %CSP.REST
{

///
/// The UrlMap determines how a Url should map to a HTTP Method and a Target ClassMet
hod
/// indicated by the 'Call' attribute. The call attribute is either the name of a met
hod
/// or the name of a class and method seperated by a ':'. Parameters within the URL p
receded
/// by a ':' will be extracted from the supplied URL and passed as arguments to the n
amed method.
///
/// In this Route Entry GET requests to /class/namespace/classname will call the GetC
lass method
///
/// <Route Url="/class/:namespace/:classname" Method="GET" Call="GetClass"/>
```

```
///
XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <Route Url="/:service" Method="GET" Call="InvokeEnsembleService"/>
  <Route Url="/:service/:p1" Method="GET" Call="InvokeEnsembleService"/>
  <Route Url="/:service/:p1/:p2" Method="GET" Call="InvokeEnsembleService"/>
  <Route Url="/:service" Method="PUT" Call="InvokeEnsembleService"/>
</Routes>
}

ClassMethod InvokeEnsembleService(service, argv...) As %Status
{
  try {
    set status = ##class(Ens.Director).CreateBusinessService(service, .instace)
    throw:$$$ISERR(status) ##class(NoProduction).%New()
    #dim response as %DynamicObject
    set status = instace.ProcessInput(.argv, .response)
    throw:$$$ISERR(status) ##class(%Exception.StatusException).%New(status)
    throw:'$data(response) ##class(NoResponse).%New()
    if $isObject(response) {
      if
 response.%Extends(##clas
s(%DynamicObject).%ClassName(1)) { write response.%ToJSON() }
      elseif response.%Extends(##class(%ZEN.proxyObject).%ClassName(1)) {
        do ##class(%ZEN.Auxiliary.jsonProvider).%ObjectToJSON(response)
      }
      elseif response.%Extends(##class(%XML.Adaptor).%ClassName(1)) {
        do response.XMLExportToString(.ret)
        write ret
      }
      else { throw ##class(Serialization).%New() }
    }
    else {
      write response
    }
  }
  catch ex {
    set httpErrorCode = $case(ex.%ClassName(1),
                ##class(h2.createrestfulservice.NoProduction).%ClassName(1):503,
                                                              :500)
    do ..ReportHttpStatusCode(httpErrorCode,ex.AsStatus())
  }
  quit $$$OK
}

}
```

If you are over with reading you might recognize that there are three custom exceptions. Please find the code of them right below.

```
Class h2.createrestfulservice.Serialization extends %Exception.AbstractException {

Method OnAsStatus() as %Status
{
  quit $$$ERROR(5001,"No known serialization method")
```

```
}

}

Class h2.createrestfulservice.NoProduction extends %Exception.AbstractException {

Method OnAsStatus() as %Status
{
  quit $$$ERROR(5001,"Production is down")
}

}

Class h2.createrestfulservice.NoResponse extends %Exception.AbstractException {

Method OnAsStatus() as %Status
{
  quit $$$ERROR(5001,"No response is received")
}

}
```

As you expect Ensemble Business Operations using EnsLib.HTTP.OutboundAdapter are catching return status other then 200 (OK). That allows to stay with the standard error handling inside Ensemble at the service consumer side. There is no real need for Business Operation example. Any of those created during any earlier session is just fine.

Stay tuned, I ' ll be back soon with further reading on Ensemble RESTful web services. The next is " RESTful API " .

#Beginner #Error Handling #REST API #Ensemble