Department of Veterans Affairs

API Developer Playbook

API Developer Playbook Blueprint Documentation

Maps to Framework Set #1



Document Version Draft 01.00.04

Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
|  |  |  |  |

Table of Contents

[1. Exception Handling 1](#_Toc519761250)

[1.1. Exception Handling Approach 1](#_Toc519761251)

[1.1.1. HTTP Response Message Header Configuration 1](#_Toc519761252)

[1.1.1.1. HTTP Status Line 1](#_Toc519761253)

[1.1.1.2. HTTP Response Headers 2](#_Toc519761254)

[1.1.1.3. HTTP Response Message Body 2](#_Toc519761255)

[1.1.2. HTTP Response Message Body Guidelines for RESTful Services 2](#_Toc519761256)

[1.1.2.1. Enterprise Shared Services Fault Model 4](#_Toc519761257)

[1.2. Transactional Exception Processing 5](#_Toc519761258)

[1.2.1. BankAccount Use Case 5](#_Toc519761259)

[1.2.2. BankAccounts Use Case 6](#_Toc519761260)

[1.3. Non-Transactional Exception Processing 7](#_Toc519761261)

[1.4. Global Exception Guidelines 8](#_Toc519761262)

[1.4.1. Legacy Access Level Exceptions 8](#_Toc519761263)

[1.4.2. Middleware-Level Exception 9](#_Toc519761264)

[1.4.3. API Level Exception Guidelines 9](#_Toc519761265)

[1.5. Message Process Failure and Retries 10](#_Toc519761266)

[1.5.1. Transient Process Failures Message Retries 10](#_Toc519761267)

[1.5.1.1. Transient Process Failure Simple Retry 10](#_Toc519761268)

[1.5.1.2. Transient Process Failure Dynamic Retry 10](#_Toc519761269)

[1.5.2. Handling Message Failures 11](#_Toc519761270)

[1.5.2.1. Central Exception Message Handling 11](#_Toc519761271)

[1.5.2.2. Local Level Exception Message Handling 11](#_Toc519761272)

[1.6. Exception Logging Guidelines 12](#_Toc519761273)

[1.6.1. Enterprise Shared Services Logging Model 12](#_Toc519761274)

[1.7. Notification and Alerting Guidelines 16](#_Toc519761275)

[1.7.1. Asynchronous Error/Fault Message Channel 16](#_Toc519761276)

[1.8. Integration with Queuing 17](#_Toc519761277)

[1.8.1. RESTful Integration WebSphere MQ Messaging System 17](#_Toc519761278)

Table of Tables

[Table 1: ESS Fault Description Table 4](#_Toc519761279)

[Table 2: ESS Logging Description Table 13](#_Toc519761280)

**Table of Figures**

[Figure 1 ESS Fault Model 4](#_Toc519761281)

[Figure 2 ESS Fault Example 5](#_Toc519761282)

[Figure 3 Multi Level Processing Stack 8](#_Toc519761283)

[Figure 4 ESS Logging Model 12](#_Toc519761284)

[Figure 5 Logging Example 15](#_Toc519761285)

# Areas that need more content

* Explain Exception Handling patterns such as Try/Catch approaches; Actions to take when an Exception occurs
* Better explanation of Error Categories and their definitions: (EX: Infrastructure vs. Service vs. Resource)

# Exception Handling

The SOA web service interfacing style of SOAP tends to be centered around operations that are usually use-case specific and specialized. In contrast, RESTful is centered around business/data entities exposed as resources that are identified via URIs and can be manipulated via standardized CRUD-like methods using different representations.

This section of the document will focus on RESTful error handling although SOAP error handling may be included in a latter iteration.

## Exception Handling Approach

RESTful APIs should define the functional, business view and abstract from implementation aspects. Success and error responses are a vital part to define how an API is used correctly.

Defining all success and service specific error responses in an API specification is required. Both are part of the interface definition and provide important information for service clients.

API designers should also considererror handling guidelines that should be part of the associated online API documentation. Such documentation provides information and exception handling guidance on application-specific errors and should be referenced as part of the API specification. This can help allow for more self-service enablement, thereby reducing service support tasks, and improving consumer and provider performance.

HTTP Response Messages

Restful APIs are typically based on the HTTP protocol for connecting their clients. To this end, the following sections detail HTTP Response Message population guidelines

### HTTP Response Message Header Configuration

RESTful API response implementations utilize the HTTP Response Message Header. A high-level overview of the HTTP Header and it communication of success and error conditions follows.

#### HTTP Status Line

The first line of the Response Message is called the status line and is followed by optional response header(s).

The HTTP Status Line has the following syntax:

*HTTP-version status-code reason-phrase*

* HTTP-version: The HTTP version used in this session. Either HTTP/1.0 and HTTP/1.1.
* status-code: a 3-digit number generated by the server to reflect the outcome of the request.
* reason-phrase: gives a **short** explanation to the status code.

Examples of status line are:

* HTTP/1.1 200 OK
* HTTP/1.0 404 Not Found
* HTTP/1.1 403 Forbidden
* HTTP/1.1 500 Internal Server Error

Use the standard-based HTTP status codes. There are over 70 HTTP status codes but it’s unnecessary to use more than is needed.

#### HTTP Response Headers

The response headers are in the form name: value pairs:

*response-header-name: response-header-value1, response-header-value2, ...*

Examples of response headers are:

* Content-Type: application/json
* Content-Length: 35
* Connection: Keep-Alive
* Keep-Alive: timeout=15, max=100

#### HTTP Response Message Body

The response message body is optional and should be used to supply any additional information required to understand and appropriately handle the HTTP Response. As an example, returning a 200 OK Status Code is often sufficient, in that the consumer of the API is only concerned with the success of the operation. It is, however, poor practice to return a 400 Bad Request without explicitly describing the faults in the Request Payload in the Response Message Body.

Where possible, the consumer of the API should be able to correct any issues with a failed call to your API entirely autonomously. As in the case of the 400 Bad Request, the consumer should be able to correct their model based on the instructions in the HTTP Response Message Body. For a 301 Moved Permanently, the consumer should be able to discover the new resource to be used and the impact the move has on their business practices.

When returning a 500 Internal Server Error, **only** the minimum data necessary to aid the consumer in handling the error should be provided. This may include a generic error message, a reference id to a logged error which the consumer may use to match their error to the API’s logs, or instructions on how to seek assistance. Under **no** circumstances should diagnostic data be released in a 500 Internal Server Error, as this information can be used to compromise VA systems and security.

### HTTP Response Message Body Guidelines for RESTful Services

References were made above to the Response Message Body JSON payloads. This section details a standardized layout that will facilitate the creation of JSON payload.

The Enterprise Shared Services (ESS) Exception Handling Guidelines document is pointed to by this imbedded document:



This document contains XML based messaging guidelines that were defined for SOAP services. From a consistency perspective, continuing these guidelines for API’s would be beneficial.

VA APIs should use the Status Code which best reflects the result of each API call. A complete list can be found at <https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml>, or a more friendly list at <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

While there is no prohibition for use of any HTTP Status Code, there are those which are more and those which are less useful. 200 OK is a ubiquitous code and its use is strongly encouraged. 203 Non-Authoritative Information is rarely used and only appropriate when it exactly describes the response and is useful to the consumer. The recommended VA status codes are as follows:

* 200 OK – When the API call was successful.
* 400 Bad Request – When the API call failed because the request contained incomplete or invalid data. Should be paired with meaningful instructions in the HTTP Response Message Body.
* 401 Unauthorized – When the API call was made by an actor with insufficient privileges to make use of the resource.
* 404 Not Found – When the call requested a resource which does not exist. This not only serves to handle calls to non-existent end-points. It is also a common industry practice to use this to communicate a bad ID. If the consumer sends a GET for an entity whose ID does not exist in your system, 404 is an appropriate response, particularly with a meaningful Response Message Body.
* 500 Internal Server Error – When the Request and its payload were valid and *should have succeeded*, but an unexpected error occurred in the API. This indicates that the resolution should be affected by the producers of the API, and **no** diagnostic data should be returned to the consumer.

References were made above to the Response Message Body JSON payloads. This section details a standardized layout that will facilitate the creation of JSON payload.

The Enterprise Shared Services (ESS) Exception Handling Guidelines document, pointed to by this imbedded document:



This document contains XML based messaging guidelines that were defined for SOAP services. From a consistency perspective, continuing these guidelines for RESTful based services would be beneficial.

#### Enterprise Shared Services Fault Model

Figure 1 ESS Fault Model



The format and encoding of faults varies with the protocol and transport, but the meaning of the following information will be consistent across all interfaces.

Table 1: ESS Fault Description Table

| **Element** | **Type** | **Required** | **Example** | **Description** |
| --- | --- | --- | --- | --- |
| code | String | Yes | 404 | A descriptor of the fault condition, and generated by the service. It could be an application generated error code or it could map to the HTTP Response Code Guidelines detailed in **Section** Error! Reference source not found. above. |
| text | String | Yes | Not Found | A textual description of the fault condition, and generated by the service. It could map to the HTTP Response Code Guidelines detailed in **Section** Error! Reference source not found. above. |
| essCode | String | Yes | gov.va.ess.fault.io.ResourceNotFound | A code representing the mapping of the fault condition a representative taxonomy. The taxonomy included in the (ESS) Exception Handling Guidelines document – Appendix B may be a candidate. |
| essText | String | No | “Unable to locate resource XXX” | A user interface friendly text description of the fault condition. The service generating the fault may populate this field. |
| nested Fault | ESS Fault | No | See JSON example below | Nested structure of faults bubbled up from lower in the chain of service calls.  Must be populated if fault is triggered by another fault. Is not populated if fault is triggered by an internal exception. |

The following example contains a minimized set of HTTP Response Message Header name-value pair parameters. The JSON response is contained in the HTTP Response Message body.

Figure 2 ESS Fault Example

{

"ESSFault": {

"code": "400" "text": "NotFound",

"essCode": "gov.va.ess.fault.io.ResourceNotFound",

"essText": "UnabletolocateresourceXXX",

"nestedFault": {

"code": "400",

"text": "NotFound",

"essCode": "gov.va.ess.fault.io.ResourceNotFound",

"essText": ""UnabletolocateresourceXXX"

}

}

}

## Transactional Exception Processing

Consider the following use case and how to implement it using an API service philosophy, keeping in mind to not comprise the state of conceptual API model. The API model is predicated on correctly representing the state related transition(s) of the API resource. The processing of exceptions should include not only notifying the end client of errors but just as important, ensuring that the state of the resource is not negatively impacted by the error scenario.

The use case is updating multiple resources. For example, the transfer of $100 from one customers bank account into another customer’s bank account. The actual detailed transactional semantics of this use case will not be detailed in this section as these details are better served by their inclusion in a section that discusses transactional behaviors using a API service or services.

An Exception Processing discussion for this scenario will need some assumptions made as to the makeup of the RESTful services needed to implement this scenario. There are potentially many RESTful service solutions that this use case, but two will be discussed in this section. These proposals may not be the most reasonable or elegant solution, but the point is to detail the Exception Processing of each. It should be noted here that the use cases and subsequent RESTful interactions that can be applied to a domain object such as a Bank Account can be created using elegant restful principles such as *Hypermedia As The Engine Of Application State* (HATEOAS).

The first is a RESTful service that updates the state of two instances of a resource named BankAccount identified by a unique identifier for that resource (i.e. account number).

The second is a RESTful service that updates the state of a collection resource named BankAccounts identified by unique identifiers for each resource in the collection (i.e. account number).

An additional approach to the above resource solution is the use of a Bank Account Transaction resource which raises the process level of abstraction. This solution will not be detailed in this discussion

### BankAccount Use Case

The implementation solution for this use case will consist of two BankAccount RESTful HTTP PUT invocations one for the first account and one for the second. Because there is a requirement to implement stateful semantics on the server, the inclusion of a unique Transaction Id which could be included in the HTTP Request Header of both invocations will be used. The dollar amount of the transfer will be included in the Request Message Body JSON payload of the first invocation.

The unique Transaction Id will be used on the server implementation to marry the two RESTful invocations.

The happy path scenario consists of the second RESTful invocation returning a response that states that the transfer from one account to another was successful. The client application service invocation orchestration would then have to reconcile the business rules whereby the two invocations were successful.

In the case of a non-happy path, one of two failures can occur:

* Within the first service invocation

The non-happy path processing if a failure occurred in the first BankAccount invocation, (e.g. there wasn’t sufficient funds in the account to do the transfer) would consist of an error response sent back alerting the client that the service call was not successful. The appropriate error verbiage would be included in the JSON response payload. The client application service invocation orchestration would then cancel the submission of the second service.

* Within the second service invocation

The non-happy path processing if a failure occurred in the second BankAccount invocation would consist of an error response sent back alerting the client that the service call was not successful. The appropriate error verbiage would be included in the JSON response payload. The result of the first service would be nullified by whatever transaction recovery scheme was used (a pending database commit for the first bank account was cancelled, database roll back etc.)

### BankAccounts Use Case

The implementation solution for this use case will consist of one BankAccounts RESTful HTTP PUT invocation. The dollar amount of the transfer and the two bank account numbers will be included in the Request Message Body JSON payload.

The happy path scenario consists of the BankAccounts RESTful invocation returning a response that states that the transfer from one account to another was successful.

In the case of a non-happy path, one of two failures can occur:

* Processing the first account in the first BankAccounts collection.

The non-happy path processing if a failure occurred in the first BankAccounts processing, (e.g. there wasn’t sufficient funds in the account to do the transfer) would result in an error response sent back alerting the client that the service call was not successful. The appropriate error verbiage would be included in the JSON response payload. The result of the first account processing would be nullified by whatever transaction recovery scheme was used (a pending database commit for the first bank account was cancelled, etc.) plus the second account processing would be terminated.

* Processing the first account in the second BankAccounts collection.

The non-happy path processing if a failure occurred in the second account would result in an error response sent back to the client alerting the client, utilizing the JSON response payload, that the service call was not successful. The result of the first service would be nullified by whatever transaction recovery scheme was used (a pending database commit for the first bank account was cancelled, etc.)

## Non-Transactional Exception Processing

As was alluded to in in **section 1.2 Transactional Exception Processing** RESTful web service processing, even those of the non-transactionalvariety should keep in mind to not compromising the conceptual RESTful model. The RESTful model is predicated on correctly representing the state related transition(s) of the RESTful resource. The processing of exceptions should include not only notifying the end client of errors but just as important, ensuring that the state of the resource is not negatively impacted by the error scenario.

The processing of the scenarios defined in **section 1.2 Transactional Exception Processing** even though they involved multiple state transition processing, should be applied to RESTful web services that perform more simplistic single resource processing. There is no need to re-iterate the processing step as they are applicable to non-transactional scenarios.

## Global Exception Guidelines

The following diagram depicts the Multi-Level Processing Stack that will be used to describe the Error Processing philosophy as exceptions occur and propagate up to the Consuming Application Level. The diagram is a conceptual view of the processing stack, actual physical components within each level are not detailed as to not complicate the discussion.

Additionally, this section will describe service orchestration and how multiple layers of service calls can affect the handling of exceptions and management of faults.

Figure 3 Multi Level Processing Stack



### Legacy Access Level Exceptions

For the purposed of brevity, the Legacy Access Level layer will incorporate invocation and/or integration with legacy application infrastructures as well as database access utilizing existing data store access mechanisms (JDBC, ODBC, Relational Mapping products such as Hibernate etc.).

The following are the processing steps with positive and negative procedures for handling a fault thrown by a component below the Legacy Access Level

* The Legacy Access Level **can** **handle** the fault thrown by a component in the Legacy Access Level. The Legacy Access Level has the option to propagate the error up to the Middleware Level. Additionally, the Legacy Access Level can optionally log the error returned by the legacy system component because these components are not expected to use an Enterprise Logging Service.
* The Legacy Access Level **cannot handle** the fault thrown by any component in the Legacy Access Level and continue processing. The Legacy Access Level will propagate the fault up to the Middleware Level. Additionally, the Legacy Access Level will log the error returned by the legacy system component because these components are not expected to use an Enterprise Logging Service.

### Middleware-Level Exception

The following are the processing steps with positive and negative procedures for handling a fault thrown by the Middleware Level.

* The Middleware Level **can** **handle** the fault thrown by the Legacy Access Level. The Middleware Level has the option to propagate the error up to the API Level.
* The Middleware Level **cannot handle** the fault thrown by the Legacy Access Level and continue processing. The Middleware Level will propagate the fault up to the API level.

### API Level Exception Guidelines

The following are the processing steps with positive and negative procedures for handling a fault thrown by the API Level.

* The API Level **can** **handle** the fault thrown by the Middleware Level. The API Level has the option to propagate the error up to the Consuming Application.
* The API Level **cannot handle** the fault thrown by the Middleware Level and continue processing. The API Level will propagate the fault up to the Consuming Application.

## Message Process Failure and Retries

This section will discuss retry guidelines for failures that are caused by either transient (short duration) failures and long duration failures. The specific HTTP codes that are used to define these types of failure are not included as part of this discussion as the HTTP code and error response philosophy for a service depends on the Error Response design for that service.

### Transient Process Failures Message Retries

Transient process failures (short duration) occur while communicating to external components or services that are not available. If the service again is called again, it is possible that it be will be successful hence the term transient failure.

The initial step in handling transient process failures is to identify the transient process fault by checking if the fault is something that the target service is sending and has provided error message context from the application perspective. If this is the case then this is not a transient fault since the service is sending a fault. But if the fault is not coming from the service, there may be other reasons like infrastructure failure issues and it is something that cannot be resolved by simply the re-calling service again, this is potentially a transient process fault.

#### Transient Process Failure Simple Retry

Once a transient fault has been identified, the simplest solution is the implementation of some retry logic so that the issue will get resolved by calling the service again. A typical way to implement the retry is as follows

1. Define the maximum retry count.
2. Retry the service call and increment the retry count.
3. If the call succeeds, return the result to the caller.
4. If the same fault persists, keep retrying until the maximum retry count is hit.
5. If the call is failing even after maximum retries, communicate that target service is unavailable

#### Transient Process Failure Dynamic Retry

Using the Retry Process identified above could cause potential issues. These issues may be a result of the service being overloaded or some type of throttling that was implemented at the service end. This service is rejecting new calls and there is a possibility that the retry requests are further adding to the overload of service.

A work around for this problem is the use of a Dynamic Retry that instead of retrying after waiting for a fixed amount of time, the waiting time is increased. Incrementally increasing the wait time between the consecutive retry requests after each failure gives the service some “breathing time’. A typical way to implement the dynamic retry is as follows

1. Define the maximum retry count.
2. Retry the service call and increment the retry count.
3. If the call succeeds, return the result to the caller.
4. If the same fault persists, increase the delay period for next retry.
5. Keep retrying and keep increasing the delay period until the maximum retry count is hit.
6. If the call is failing even after maximum retries, communicate that target service is unavailable.

### Handling Message Failures

The distributed nature of composite applications and service orchestrations requires message handling capabilities to support effective diagnosis and resolution of errors. Centralized as well as localized error handling augmented by error logging enables efficient access to error information and consistent tracking of exceptions and faults.

#### Central Exception Message Handling

Exceptions that do not result in a fault message, but do affect the behavior of service orchestrations or service consumers must be handled and logged centrally. Information such shutdowns due to scheduled maintenance should be handled and logged centrally. Any exception that results in a fault message must be handled and logged centrally.

#### Local Level Exception Message Handling

Exceptions that do not return a fault and do not influence the behavior of service orchestrations and service consumers would only require locally supported logging. Exceptions of the informational variety would only require locally supported logging

## Exception Logging Guidelines

Centralized and Localized Error procedures are very similar to the Centralized and Localized Message Handling Failures detailed in section 1.5 Message Process Failure and Retries. Refer to that section for details.

Actual Logging Procedures and implementation will be detailed in the Baseline Logging section of this document.

The Enterprise Shared Services (ESS) Exception Handling Guidelines document, pointed to by this imbedded document:



This document contains XML based logging guidelines that were defined for SOAP services. From a consistency perspective, continuing these guidelines for RESTful based services would be beneficial.

### Enterprise Shared Services Logging Model

Figure 4 ESS Logging Model



Table 2: ESS Logging Description Table

| Element | **Type** | **Required** | **Example** | **Description** |
| --- | --- | --- | --- | --- |
| transaction ID | String | Yes | a20014d3e17ff4fcb578-b7a6-41f9-b974-cef684d5ec62 | A unique identifier passed to the service in the message header. |
| Code | int | Yes | 32154 | A descriptor of the fault condition, and generated by the service. It may be unique to the service, but is typically a coded entry defined by the protocol used on the service interface. |
| Text | String | Yes | gov.va.ess.service.AccessServiceLegacySystemError | A textual description of the fault condition, and generated by the service. It may be unique to the service, but is typically defined by the protocol used on the service interface. It should be human readable. |
| essCode | String | Yes | gov.va.ess.fault.io.FileCannotBeFound | A code representing the mapping of the fault condition to the standard ESS taxonomy. |
| essText | String | No | Requested action is denied due to insufficient authorization | A user interface friendly text description of the fault condition. The service generating the fault may populate this field. |
| nestedFault |  | No |  | Nested structure of faults bubbled up from lower in the chain of service calls.  Only populated if fault is triggered by another fault. |
| severity | String | Yes | Error | Enumeration:  **Fatal** – Severe error that causes premature termination of service execution. The service would not able to continue processing the request, and a response message would not be sent. An example would be an unrecoverable system level error.  **Error** – Significant error that causes the service to return a fault. An exception occurred that prevented the service from completing the request normally and a fault was returned. Examples might include a database constraint error, or disk full.  **Warning** – Significant error that does not result in the return of a fault, but is otherwise important to understanding service behavior. A recoverable exception that does not result in a fault message. Abnormal retries on a database connection for example. May also include interesting events not generated by exceptions, but useful in forensic analysis  **Information** – Not an error, but information that is otherwise important to understanding service behavior. Example might be shutting the service down for maintenance. |
| timestamp | Timestamp | Yes | 2013-05-05T08:15:30-05:00 | Timestamp of the time the error occurred. |
| serviceName | String | Yes | gov.va.ess.util.FileAccessService | Unique name of the service. This is the service namespace. |
| serviceInstance | String | Yes | <http://101.12.34.97:443 /eligibility/claimsService/Benefits/eligibilityStatus>/v1 | This is the service endpoint |
| hostName | String | No | vaaacmhvapp12 | Name of the individual host within the cluster hosting the service instance. |
| hostIP | String | No | 101.34.28.134 | IP address of the individual host. |
| userId | String | No | CN=Test User, OU=ESS,DC=OIT,DC=VA,DC=GOV | Fully qualified identifier of the user passed on the request. |
| subjectId | String | No | <Id>1234567890V654321<Id> <IdType>NI<IdType>  <AssigningLocation>123VA</AssigningLocation>  <IdSource>200M</IdSource> | Fully qualified identifier of the subject of the request. This would typically be the veteran or beneficiary, but may be any entity that is the primary subject of the request and/or important to understanding the error. |
| codePackage | String | No | gov.va.ess.resource.fileAccess | Identifier of the code package or deployment unit originating the exception. This should be unique. |
| serviceDomain | String | No | gov.va.ess.resources.v2.2.3 | Name of the service domain |
| businessDomain | String | No | Education Eligibility | Name of the business domain |
| correlationId | String | No | c1d2398c3-1449-7654-9793-29373a9b9c53c1d2398c3-1449-7654-9793-29373a9b9c53 | Correlation ID of the current request |
| sessionId | String | No | 8839272636511237 | Session ID of the user session |
| Fault Trace  or equivalent | String | No | java.io.IOException: Unable to find file 'topicalIndex.csv' ... | Detailed information on the state of the execution of service code package when the error occurred. |

Figure 5 Logging Example

Some elements in this figure are truncated for readability.

{

"ESSError": {

"transaction ID": "a20014d3e17ff4fcb578-b7a6-41f9-b974-cef684d5ec62",

"code": "32154",

"text": "gov.va.ess.service.AccessServiceLegacySystemError",

"essCode": "gov.va.ess.fault.io.ResourceNotFound",

"essText": "UnabletolocateresourceXXX",

"nestedFault": {

"code": "400",

"text": "NotFound"

"essCode": "gov.va.ess.fault.io.ResourceNotFound",

"essText": "UnabletolocateresourceXXX",

}

"severity": "Information",

"timestamp": "2013-05-05T08:15:30-05:00",

"serviceName": "gov.va.ess.util.FileAccessService",

"serviceInstance": "http://101.12.34.97:443...,

"hostName": "vaaacmhvapp12",

"hostIP": "101.34.28.134",

"userID": "CN=Test User, OU=ESS,DC=OIT,DC=VA,DC=GOV",

"subjectID": "Id>1234567890V654321<Id> <IdType>NI<IdType>…>",

"codePackage": "gov.va.ess.resource.fileAccess",

"serviceDomain":"gov.va.ess.resources.v2.2.3",

"businessDomain": "Education Eligibility",

"Correlation ID": "c1d2398c3-1449-7654-9793-29373a9b9c53c1d2398c3…”, "Session ID": "8839272636511237",

"faultTrace": "java.io.IOException: Unable to find file 'topicalIndex.csv' ..."

}

}

## Notification and Alerting Guidelines

The functionality needed to implement Notification and Alerting will be implemented by an Enterprise Error/Fault Message Channel platform either built, purchased or implemented as a Software as a Service.

### Asynchronous Error/Fault Message Channel

The Error/Fault Message Channel will provide the capability for the services that subscribe to be able to receive fault and error messages facilitated by the channel. It is anticipated that errors and faults will be published on one or more asynchronous message channels, and subscribed to by the following

* Logging Services

Will would centrally capture errors needed to provide support to services and service consumers. It would associate relevant error and fault information, and provide mechanisms to query this information, and to run analytical reports.

* Monitoring and Triage Services

Will enable automated diagnostics and handling of errors, creation of trouble tickets, integration with service desk, SLA management, etc. Design and specification of such services will have to be defined.

## Integration with Queuing

This section will give an overview of a API service that will integrate with a message Queue Messaging system such as IBM’s WebSphere MQ. WebSphere MQ is the current on-premise supported message system in the VA and because of this the IBM supplied functionality will be used to detail RESTful API that will integrate with a message Queue Messaging system functionality. It should be noted that that the IBM solution is obviously not vendor agnostic, but Queue Messaging systems are always vendor specific so solution involving queueing can’t be agnostic.

### RESTful Integration WebSphere MQ Messaging System

The messaging REST API comes as standard with IBM MQ from IBM MQ Version 9.0.4 and is enabled by default. It supplies the capability to use the messaging REST API to send and receive IBM MQ messages in plain text format.

Applications can issue an HTTP POST to send a message to IBM MQ, or an HTTP DELETE to destructively get a message from IBM MQ. Support is provided for many different HTTP headers which can be used to set common message properties.

A detailed discussion of the RESTful/WebSphere MQ functionality can be found on the using following URL:

<https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.dev.doc/q130940_.html>