# 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 focuses on RESTful error handling

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 valuable information for service clients.

API designers should ensure that error handling guidelines is contained in the associated API documentation. Such documentation provides information and exception handling guidance for application-specific errors and should be referenced as part of the API specification. This supports facilitating self-service enablement, thereby improving consumer and provider API understanding.

This document details various Exception Handling Artifacts and Strategies that provide the API developer sufficient background to ensure the developed API is sufficiently robust from an Exception Handling perspective.

These Artifacts and Strategies include:

* Exceptions and API Development
* HTTP Responses and API development
* JSON Response for API Service Errors Guidelines
* Transactional Exceptions and API Development
* Non-Transactional Exceptions and API Development
* Global Exceptions and API Development
* API Message Process Failure and Retries
* API Notification and Alerting Guidelines

## Exceptions and API Development

The discussion of the several types of exceptions and try/catch best practices in this focuses on the Java implementation although differences between Java and C# will be noted where appropriate.

An extensive description of Exception Handling, including the try/catch statement in both Java and C# programming languages, can be found <https://pclt.sites.yale.edu/exception-handling-java-and-c>

In Java, all exceptions (***java.lang.Exception)*** and errors (***java.lang.Error)*** extend from a common ***java.lang.Throwable*** parent class and only Throwable objects can be thrown and caught. In the .NET Framework, they are all descended from ***System.Exception***.

### Checked Exceptions

Checked Exceptions are thosethat are checked at compile time. If code within a method invokes a method that throws a Checked Exception, then the invoking method must either handle the exception or it must propagate the exception using a throws statement.

Checked Exceptions requirements instituted by API designers force API developers to deal with the exceptions. The expectation of the API designer is for the developers to be able to provide the appropriate logic to recover from those exceptions. Minimally this would include logging the exceptions and returning the appropriate response to the end user.

Java supports both Checked and Unchecked Exceptions.

### Unchecked Exceptions

Unchecked Exceptions are those that are not checked at compiled time although some component in the stack trace must accommodate the Unchecked Exception. This typically occurs at the top of the stack trace immediately before returning to the end user.

Unchecked Exceptions give developers fixability to ignore exceptions that are not recoverable but handle the ones that are.

Java exceptions inherited from ***java.lang.Error*** and ***java.lang.RuntimeException*** classes are Unchecked Exceptions. Everything else under ***java.lang.throwable*** is checked.

C# only supports Unchecked Exceptions thus all exception handling is purely optional although as stated above, in the Java example, some component in the stack trace must accommodate the Unchecked Exception.

### Try/Catch Exception Handling Best Practices

The system design artifact that addresses exceptions is implemented using the **try**/**catch** block statement. The following code is a typical **try**/**catch** statement with a **finally** statement,

public void tryCatchExample() {

try {

// Invoke a method that throws an Exception or Error

} catch (Exception e) {

// Perform some action to handle the Exception or Error

} finally {

// Perform some action outside of the catch

}

}

* The **try** block needs to catch all checked exceptions. Optionally unchecked exceptions can be caught here as well.
* The **catch** block defines the exception(s) to be caught.
* The **finally** block always executed and will happen either after the successful execution of the try block or after the exception has been handled in a catch block. The finally block accommodates any actions outside if the catch block for example, closing a File.

Defining Exception Handling best practices in either Java or C# in detail is not within the scope of this discussion.

The following link provides Java Exception Handling Best Practices:

<https://stackify.com/best-practices-exceptions-java/?utm_referrer=https%3A%2F%2Fwww.google.com%2F>

The following link provides C# Exception Handling Best Practices:

<https://stackify.com/csharp-exception-handling-best-practices/?utm_referrer=https%3A%2F%2Fwww.bing.com%2F>

These links provide descriptive narratives as well as code examples of these best practices.

## HTTP Responses and API Development

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.

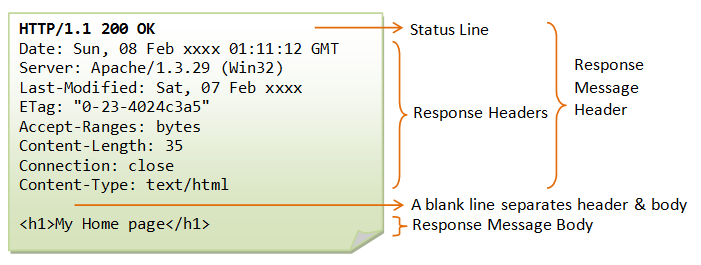
### HTTP Response Message Header Configuration

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

**Important Note:**

The HTTP Response Message descriptions in this section of the document go into detail regarding the configuration of the parts of the HTTP Response Message. The level of interest, from the perspective of an API developer, should be determined by how much responsibility the developer has in the creation of the API HTTP Response Message.

Figure 1 Sample HTTP Response Message



The above figure is a pictorial example of an HTTP Response Message. It will facilitate the explanation of the parts of the message in the descriptions that follow.

#### 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 brief 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. Guidelines for the use of HTTP status codes follow.

#### HTTP Response Message Header

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.

Following is an example of HTTP Response returning a 400 Bad Request. In this GET request, the HTTP-version was misspelled, resulted in bad syntax. The server returns an error "400 Bad Request".

Figure 2 HTTP Response returning a 400 Bad Request Example

GET /index.html HTTTTTP/1.1

(enter twice to create a blank line)

HTTP/1.1 400 Bad Request

Date: Sun, 11 Feb 2018 01:29:40 GMT

Server: Apache/1.3.29 (Win32)

Connection: close

Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">

<HTML><HEAD>

<TITLE>400 Bad Request</TITLE>

</HEAD><BODY>

<H1>Bad Request</H1>

Your browser sent a request that this server could not understand. <P>

The request line contained invalid characters following the protocol string. <P><P>

</BODY></HTML>

Wherever possible, the consumer of the API should be able to correct any issues with a failed call to the API entirely autonomously. As in the case of the 400 Bad Request, the consumer should be able to correct the request based on the instructions in the HTTP Response Message Body.

The above 400 Bad Request example definitively identified the issue in this case a misspelled HTTP literal. The important point here is defining a best practice to avoid a 400 Bad Request apart from always ensuring that the API invocation request is flawless, is to ensure that to the error message returned is unambiguous. If this situation arises the responding messaging should be clear and provide enough information so the API invocation request is correctly configured.

##### 500 Internal Server Error Guidelines

The 500 response codes indicate non-programmatic failures, such as failure to connect to a database. The expectation from a client perspective is that the client can re-submit the same request in the future and expect a successful response

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.

Ultimately the response generated because of an internal server error should ensure that the error returned is clear enough for the API user to take the appropriate action.

Figure 3 HTTP Response returning a 500 Internal Server Error Example

GET /index.html HTTP/1.0

(enter twice to create a blank line)

HTTP/1.1 500 Internal Server Error

Date: Sun, 11 Feb 2018 01:29:40 GMT

Server: Apache/1.3.29 (Win32)

Connection: close

Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">

<HTML><HEAD>

<TITLE> 500 Internal Server Error Database Connection Failure</TITLE>

</HEAD><BODY>

<H1>Bad Request</H1>

Unable to connect to the "Application Specific" Data Base. <P>

Necessary system alerting had been accomplished. <P><P>

</BODY></HTML>

### HTTP Response Message Body Guidelines for RESTful Services

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.

## JSON Response for API Service Errors Guidelines

This section details a proposed standardized layout that will facilitate the creation of JSON payload specifically as it applies to the generation of a JSON Response for API errors. The following XML API Fault Class Diagram is used to define the data elements that will be returned, in a JSON format, to the API consumer. The JSON response will be contained in the HTTP Response Message body.

Figure 4 API Fault Class Diagram



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. |
| text | String | Yes | Not Found | A technical textual description of the fault condition, and generated by the service. |
| apiText | 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 | API 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.

Figure 5 JSON API Fault Response Example

{

"APIFault": {

"code": "400"

"text": "Not Found"

"apiText": "UnabletolocateresourceXXX",

"nestedFault": {

"code": "400",

"text": "NotFound",

"apiText": ""Unable to locate resourceXXX"

}

}

}

## Transactional Exceptions and API Development

The definition of Transactional Processing per Wikipedia:

*Transaction processing is information processing in computer science that is divided into individual, indivisible operations called transactions. Each transaction must succeed or fail as a complete unit; it can never be only partially complete.*

Because RESTful is typically defined as an architecture, the implementation of transactional semantics in a Restful API implementation should be considered an architectural responsibility.

Although this document is meant to be a set of guidelines for API Developers including an example use case detailing various solutions, the fact remains that the solutioning for API transactional behavior should be defined by architectural personnel.

The following URL contains a detail White Paper discussion of Restful Transactional challenges:

<http://ws-rest.org/2014/sites/default/files/wsrest2014_submission_4.pdf>

This White Paper contains among other things, tables that define requirements and architecture considerations for Transactions.

An excerpt from the document abstract states:

*The goal of this paper is to analyze the state-of-the-art RESTful transaction models and identify the current challenges.*

### Bank Account Use Case Overview

Consider the following use case and how it might be implemented with 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 customer’s 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 an API service or services.

An Exception Processing discussion for this scenario will require some assumptions made as to the makeup of the RESTful services needed to implement this scenario. There are potentially many RESTful service solutions for 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).

Restful Service Solutions Detailed:

* The first is a RESTful service that updates the state of two instances of a resource named BankAccount, containing a unique identifier for that resource (i.e. account number).
* The second is a RESTful service that updates the state of a collection of Bank Account references contained in a single BankAccount resource using the unique identifier for each BankAccount 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. Simply stated this Bank Account Transaction resource acts as a Transactional Processing (TP) monitor and has the responsibilities of managing the various banking resources under its control ensuring transactional integrity.

The following URL contains a alternative architectural solutions to the Bank Account use case:

<https://jcalcote.wordpress.com/2009/08/06/restful-transactions/>

#### Bank Account Use Case

The implementation solution for this use case consists 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 join the two RESTful invocations.

The normal course 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 an alternate course path, one of two failures can occur:

* Within the first service invocation

The alternate course processing if a failure occurred in the first BankAccount invocation, (e.g. there are insufficient funds in the account to do the transfer) would consist of a returned error response alerting the client that the service call was unsuccessful. 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 alternatate course 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 unsuccessful. 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.)

#### Bank Accounts 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 normal course 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 alternate course, one of two failures can occur:

* Processing the first account in the first BankAccounts collection.

The alternate course processing if a failure occurred in the first BankAccounts processing, (e.g. there are insufficient funds in the account to do the transfer) would result in a returned error response 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 alternate course 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 unsuccessful. 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 Exceptions and API Development

As was alluded to in in section 1.4 TransactionalExceptionsand API Development **[hyperlink here]** RESTful web service processing, even those of the non-transactionalvariety should not compromise 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 notifying the end client of errors, and, just as important, ensure the state of the resource is not negatively impacted by the error scenario.

(Editor’s Note: Standards Review stops here)

The processing of the scenarios defined in section1.4TransactionalExceptionsand API Developmenteven 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 steps as they are applicable to non-transactional scenarios.

## Global Exceptions and API Development

Global Exception processing is necessitated when multi components of an API configured in different layers of a processing stack. This section will discuss Global Exception processing using the Multi-Level API Processing Stack defined by the diagram below.

The following diagram depicts the Multi-Level Processing Stack that will be used to describe the Error Processing philosophy as faults/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 6 Multi Level API Processing Stack



A brief description of the levels in the above diagram follow:

* The Experience API Layer enables users (Consuming Applications) to interact with an IT ecosystem of a business.
* Process API Layer, which pulls content from System API Layer, applies business logic to process the content and pushes the content in any desired format asked by the Experience API Layer.
* The System API Layer houses the core systems in an IT ecosystem.

### System API Level Exceptions

For the purposed of brevity, the System API Level layer will incorporate invocation and/or integration with legacy application infrastructures, Software as A Service component, 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 System API Level

* The System API Level **can** **handle** the fault thrown by an integration component in the System API Level. Based on integration design behavior of the throwing component,
* The System API Level has the option to propagate the error up to the Process API Level.
* The Process API Level has the option to propagate the error up to the Experience API Layer
* Additionally, the System API Level can optionally log the error returned by the System API Level integrating component because these components are not expected to use an Enterprise Logging Service.
* If the System API Level **cannot handle** the fault thrown by any integrating component in the System API Level (e.g. database connectivity failure) and continue processing, the System API Level will propagate the fault up to the Process API Level. Additionally, the System API Level will log the error returned by the legacy system integrating component because these components most like will not use a logging service visible to the API infrastructure.

### Process API Level Exception

The Process API Level must react and possibly process error propagated upwards from the System API Level as well potentially having to propagate those error upward to the Experience API Level. This section details the error processing responsibilities that are required to be implemented by this layer

The following are the processing steps with positive and negative procedures for handling faults within the Process API Level.

* The Process API Level **can** **handle** the fault thrown by the System API Level. The Process API Level has the option to propagate the error up to the API Level.
* The Process API Level **cannot handle** the fault thrown by the System API Level and continue processing. The Process API Level will log appropriately and propagate the fault up to the Experience API level.

### Experience API Level Exception Guidelines

The Experience API Level must react and possibly process error propagated upwards from the Process API Level as well potentially having to propagate those error upward to the Consuming Application. This section details the error processing responsibilities that are required to be implemented by this layer

The following are the processing steps with positive and negative procedures for handling faults within the Experience API Level.

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

## API 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. For example, if the target service is responding that is has exceeded its processing time interval.

#### Transient Process Failure Simple Retry

The decision to retry or not to retry in a transient failure scenario is not a simple yes or no determination. A working knowledge of the Service Level Agreement (SLA) of the target service is required. An educated determination based on the SLA may then be made. There may be other characteristics of the target service that can be used to make to make the retry or not to retry decision.

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 which would most likely be determined by something like investigation the SLA of the target service before hand
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.

## API Error Response Notification and Alerting

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. This Error/Fault Message Channel should have the capability to segregate different Error/Fault types by categories.

### API Error/Fault Category Placement Guidelines

A requirement of the Error/Fault Message Channel, as alluded to above. Is the capability to segregate different Error/Fault types by categories. Following are some examples segregated Error/Fault categories:

Error/Fault types needing attention by a Middleware Administrator

Investigate errors emanating from middleware components

Error/Fault types needing attention by a Database Administrator

Investigate errors emanating from database components

Error/Fault types needing attention by a API Application on call SME

Investigate errors emanating from API application components

Error/Fault types needing attention by a Software as a Service Administrator

Investigate errors emanating from SAAS components

### Asynchronous Error/Fault Message Channel

An 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.