Windows Azure Queue

December, 2008

Table of Contents

[1 Introduction 1](#_Toc217109670)

[2 Build Cloud Applications with Azure Queue 2](#_Toc217109671)

[3 Data Model 5](#_Toc217109672)

[4 Queue REST Interface 6](#_Toc217109673)

[5 Queue Usage Example 7](#_Toc217109674)

[5.1 A Producer-Consumer Scenario 7](#_Toc217109675)

[5.2 REST Request Examples 8](#_Toc217109676)

[5.2.1 REST PutMessage 8](#_Toc217109677)

[5.2.2 REST GetMessages 9](#_Toc217109678)

[5.2.3 REST DeleteMessage 10](#_Toc217109679)

[6 Best Practices 11](#_Toc217109680)

[6.1 Retry Timeouts and “Connection closed by Host” errors 11](#_Toc217109681)

[6.2 Tune Application for Repeated Timeout errors 11](#_Toc217109682)

[6.3 Error Handling and Reporting 11](#_Toc217109683)

[6.4 Choosing the Invisibility Time for GetMessage 12](#_Toc217109684)

[6.5 Deleting a Message from the Queue 12](#_Toc217109685)

[6.6 Adjusting Worker Roles Based on Queue Length 13](#_Toc217109686)

[6.7 Using Binary Format for Messages 13](#_Toc217109687)

# Introduction

 Windows Azure is the foundation of Microsoft’s Cloud Platform. It is an “Operating System for the Cloud”, which provides essential building blocks for application developers to write scalable and highly available services. Windows Azure provides:

* Virtualized Computation
* Scalable Storage
* Automated Management
* Rich Developer SDK

Windows Azure Storage allows application developers to store their data in the cloud, so the application can access its data from anywhere at any time, store any amount of data and for any length of time, and be confident that the data is durable and will not be lost. Windows Azure Storage provides a rich set of data abstractions:

* Windows Azure Blob – provides storage for large data items.
* Windows Azure Table – provides structured storage for maintaining service state.
* Windows Azure Queue – provides asynchronous work dispatch to enable service communication.

This document describes Windows Azure Queue, and how to use it. Windows Azure Queue provides a reliable message delivery mechanism. It provides a simple and asynchronous work dispatch mechanism, which can be used to connect different components of a cloud application. The Windows Azure Queues are highly available, durable and performance efficient. Its programming semantics ensure that a message can be processed at least once. Furthermore, Windows Azure Queue has a REST interface, so that applications can be written in any language and they can access the queue at anytime from anywhere in the Internet via the web.

# Build Cloud Applications with Azure Queue

Windows Azure Queue allows decoupling of different parts of a cloud application, enabling cloud applications to be easily built with different technologies and easily scale with traffic needs.



**Web Server FE**

**Backend Processing Server**

**Web Server FE**

**Windows Azure Cloud**

Request Queue

Table Store

Blob Store

**Backend Processing Server**

**Backend Processing Server**

Figure 1 Build Cloud Applications with Azure Queue

The figure above illustrates a simple but common scenario for cloud applications. There are a set of web servers hosting the frontend logic of handling web requests. There are a set of backend processing servers implementing the business logic of the application. The web server frontend nodes communicate with the backend processing nodes via a set of queues. Persistent state of the application can be stored in Windows Azure Blob storage and Windows Azure Table storage.

Consider an online video hosting service application as an example. Users can upload videos to this application; the application can then automatically convert the video file into different media formats and store them; in addition, the application will automatically index the description information of the video so that they can be easily searched on (e.g. based on keywords in the descriptions, actors, directors, title, and so on).

Such an application can use the architecture described earlier. The web frontends implement the presentation layer and handle web requests from users. Users may upload videos via the web frontends. The video media files can be stored as blobs inside the blob store. The application may also maintain a set of tables to keep track of the video files it has as well as maintaining the indexes used for search. The backend processing servers are responsible for converting the input video files into different formats and store them into the blob storage. The backend servers are also responsible for updating the tables for this application in the table storage. Once the frontend servers receive a user request (e.g. a video upload request), they can generate a work item and push it into the request queue. The backend servers can then take these work items off the queue and process them accordingly On successful processing of each work item, the backend server should delete it from the queue so as to avoid duplicate processing by another backend server.

This architecture has a number of advantages due to the use of Windows Azure Queue:

1. **Scalability** - The application can more easily scale according to traffic needs. The benefits are of several-folds:

First of all, the queue length directly reflects how well the backend processing nodes are catching up with the overall workload. A growing queue length indicates that the backend servers cannot process the work fast enough. In this case, the application may want to increase the number of backend nodes so that the work can be consumed more quickly. If the queue length always stays near zero, this means there is more processing power at the backend than what the traffic needs. In this case, the application may reduce the number of backend nodes to save resources. Observing the queue length and tuning the number of backend nodes accordingly, applications can effectively scale smoothly accordingly to the amount of traffic.

Second, the use of queues decouples different parts of the application, making it easier to scale different parts of the application independently. In this example, the frontends and the backend servers are decoupled, and they communicate via the queues. This allows the number of backend servers and the number of frontend servers to be adjusted independently without affecting the application logic. This allows an application to easily scale out the critical components by adding more resources/machines to them.

Third, the use of queues allows the flexibility of efficient resource usage within an application, allowing the application to scale more efficiently. That is, separate queues can be used for work items of different priorities and/or different weights, and separate pools of backend servers can process these different queues. In this way, the application can allocate appropriate resources (e.g. in term of the number of servers) in each pool, thereby efficiently use the available resources to meet the traffic needs of different characteristics. For example, work items that are mission critical can be put into a separate queue, so that they can be processed earlier without having to wait for other work to complete. In addition, work items that will consume a large amount of resources (such as video conversion) may use their own queue. Different pools of backend servers can be used to process work items in each of these queues. The application can adjust the size of each of these pools independently according to the traffic it receives.
2. **Decoupling Front-End Roles from Back-End Roles** - Different parts of the application are decoupled due to the use of queues, which allows significant flexibility and extensibility of how the application can be built. The messages in the queue can be in a standard and extensible format, such as XML, so that the components communicating at both ends of the queue do not have dependency on each other as long as they can understand the messages in the queue.

Different technologies and programming language can be used to implement different parts of the system with maximum flexibility. For example, the component on one side of the queue can be written in .NET framework , and the other component can be written in Python.

Furthermore, changes within a component are transparent to the rest of the system. For example, a component can be re-written using a totally different technology or programming language, and the system still works seamlessly without changing the other components, since the components are decoupled using queues.

In addition, this also allows your application to smoothly transition to newer technologies, since different implementations of the same component can coexist in such a system due to the use of queues. In the example above, one can gradually phase out the components built with legacy technologies and replace them with new implementations. The old implementation and the new implement can run on different servers at the same time, and process work items off the same queue. All these are transparent to the other components in the application.
3. **Traffic Bursts** - Queues provide buffering to absorb traffic bursts and reduce the impact of individual component failures. In the earlier example, there can be occasions where a burst of requests arrive in a short interval. The backend servers cannot quickly process all the requests. In this case, instead of dropping the requests, the requests are buffered in the queue, and the backend servers can process those at their own pace and eventually catch up. This allows the application to handle bursty traffic without losing availability.

Furthermore, use of queues also mitigate the impact of individual component failures. In the earlier example, if a few backend servers crashed, instead of losing all the work items, the queues can buffer all the work items sent while the backend servers are down. When the backend servers come back up, they can continue processing the work items off the queue and eventually catch up to arrival rate of the data into the queue, and these failures are transparent to the other components. Remember that the work items being processed by the back end servers when they crashed are also not lost as they reappear in the queue after the VisibilityTimeout thus ensuring there is no data loss due to component failures. This allows the application to tolerate failures without losing availability.

In summary, the queue model ensures that applications are insured from data loss and availability loss even if the components underneath the applications fail regularly. For this model to work correctly, the application writer should ensure that the backend processing of the queue work items are idempotent. This allows a work item to be partially processed many times, due to failures, before finally being completely processed and then deleted from the queue.

# Data Model

Windows Azure Queue has the following data model.

* **Storage Account** – All access to Windows Azure Storage is done through a storage account.
	+ This is the highest level of the namespace for accessing queues and their messages. To use Windows Azure Storage a user needs to create a storage account. This is done via the Windows Azure portal web interface. The user will receive a 256-bit secret key once the account is created. This secret key is then used to authenticate user requests to the storage system. Specifically, a HMAC SHA256 signature for the request is created using this secret key. The signature is passed with each request to authenticate the user requests by verifying the HMAC signature.
	+ An account can have many queues
* **Queue** – A queue contains many messages. The queue name is scoped by the account.
	1. There is no limit on the number of messages stored in a queue.
	2. A message is stored for at most a week. The system will garbage collect the messages that are more than a week old.
	3. Queues can have metadata associated with them. Metadata is in the form of <name, value> pairs, and they are up to 8KB in size per queue.
* **Messages** – Messages are stored in queues. Each message can be up to 8KB in size. To store larger data, one can store the data in Azure Blob store or Azure Table store, and then store the blob/entity name in the message. Note that when you put a message into the store, the message data can be binary. But when you get the messages back from the store, the response is in XML format, and the message data is returned as base64 encoded. **There is no guaranteed return order of the messages from a queue, and a message may be returned more than once.** Definitions of some parameters used by Azure Queue Service are
1. **MessageID:** A GUID value that identifies the message in the queue
2. **VisibilityTimeout:** An **i**nteger value that specifies the message's visibility timeout in seconds. The maximum value is 2 hours. The default message visibility timeout is 30 seconds.
3. **PopReceipt**: A string which is returned for every message retrieved getting a message. This string, along with the MessageID, is required in order to delete a message message from the Queue. This should be treated as opaque, since its format and contents can change in the future..
4. **MessageTTL**: This specifies the time-to-live interval for the message, in seconds. The maximum time-to-live allowed is 7 days. If this parameter is omitted, the default time-to-live is 7 days. If a message is not deleted from a queue within its time-to-live, then it will be garbage collected and deleted by the storage sytem.

The URI for a specific queue is structured as follows:

http://<account>.**queue**.core.windows.net/<QueueName>

The storage account name is specified as the first part of the hostname followed by the keyword “queue”. This sends the request to the part of Windows Azure Storage that handles queue requests. The host name is followed by the queue name. Accounts and queues have naming restrictions (see the [Windows Azure SDK](http://msdn.microsoft.com/en-us/library/dd179349.aspx) document for details), for example, the queue name cannot contain a “/”.

# Queue REST Interface

All access to Windows Azure Queue is done through an HTTP REST interface. Both HTTP and HTTPS are supported.

The HTTP/REST commands at the account-level include:

* List Queues – List all queues under the given account.

The HTTP/REST commands at the queue-level include:

* Create Queue - Create a queue under the given account.
* Delete Queue - Permanently delete the specified queue and its contents.
* Set Queue Metadata - Set/update the user-defined queue metadata. Metadata is associated with the queue as name-value pairs. This will overwrite all existing metadata with the new metadata. Get Queue Metadata - Retrieve the user-defined queue metadata as well as the approximate number of messages in the specified queue.

The queue-level operations can be done with the following URL:

http://<account>.**queue**.core.windows.net/<QueueName>

The HTTP/REST commands supported to implement the message-level operations include:

* PutMessage (QueueName, Message, MessageTTL) - Add a new message to the back of the queue. MessageTTL specifies the time-to-live interval for this message. When storing a message it can be in text or binary format, but when getting a message it is base64 encoded.
* GetMessages(QueueName, NumOfMessages N VisibilityTimeout T) - Retrieve N messages from the front of the queue and make these messages invisible for the given VisibilityTimeout T. This operation will return a message ID for the message returned along with a PopReceipt. **There is no guaranteed return order of the messages from a queue, and a message may be returned more than once.**
* DeleteMessage(QueueName, MessageID, PopReceipt) - Delete the message associated with the given PopReceipt which is returned from an earlier GetMessage call. Note that if a message is not deleted, it will reappear on the queue after its VisibilityTimeout..
* PeekMessage(QueueName, NumOfMessages N) - Retrieve N messages from the front of the queue without making the messages invisible. This operation will return a message ID for each of the message returned.
* ClearQueue - Delete all the messages from the given queue. Note that the caller should retry this operation until it returns success in order to ensure that all the messages in the queue are deleted.

 The message-level operations can be done with the following URL:

http://<account>.**queue**.core.windows.net/<QueueName>/messages

See the Windows Azure SDK document for the complete definition of the REST APIs.

# Queue Usage Example

## A Producer-Consumer Scenario

The following figure shows an example to illustrate the semantics of Windows Azure Queue.

Figure 2 Queue Usage Example

In this example, producers(P1 and P2) and consumers(C1 and C2) communicate via the message queue shown above. Producers generate work items, and put the work items as messages onto the queue. Consumers take the messages/work items off the queue and process them. There can be multiple producers and multiple consumers. Consider the following sequences of operations:

1. C1 dequeues a message off the queue. This operation will return message 1, and make message 1 invisible in the queue for 30 seconds (we assume in this example that the default VisibilityTimeout is used, which is 30 seconds).
2. Then C2 comes in and dequeues another message off the queue. Since message 1 is still invisible, this operation will not see message 1 and return message 2 back to C2.
3. When C2 completes processing of message 2, it calls Delete to remove message 2 from the queue.
4. Now let us assume that C1 crashes and does not complete processing message 1 before it dies, and therefore the message was also not deleted by C1.
5. After message 1 has passed its VisibilityTimeout, it will reappear on the queue.
6. After message 1 reappears on the queue, a later dequeue call from C2 will be able to retrieve it. It will then process message 1 to completion, and then delete it from the queue.

As shown in the example, the semantics of the queue API ensures that every message in the queue will have the chance to be processed to completion at least once. That is, if a consumer crashes after it dequeues a message and before it deletes the message, the message will re-appear in the queue after the VisibilityTimeout. This allows another consumer to come along and process the message to completion.

## REST Request Examples

This section shows the REST requests used by Windows Azure Queue. In these examples, the queue is named “myqueue” and is under the account named “sally”.

### REST PutMessage

Below is an example of a REST request for an enqueue operation. Note that the PUT HTTP verb is used. Then an optional "messagettl" option is specified. This specifies the time-to-live interval for the message in seconds. The maximum time-to-live allowed is 7 days. If this parameter is omitted, the default value is 7 days. When this time-to-live is reached, the message will be garbage collected by the system. A Content-MD5 can be provided to guard against network transfer errors and insure integrity. The Content-MD5 in this case is the MD5 checksum of the message data in the request. The content length specifies the size of the message data contents. There is also an authorization header inside the HTTP request header as shown below. Note that the message data is in the HTTP request body. It can be in text or binary format, but when getting the message it is returned as base64 encoded.

 PUT http://sally.queue.core.windows.net/myqueue/messages

 ? messagettl=3600

 HTTP/1.1 Content-Length: 3900

 Content-MD5: HUXZLQLMuI/KZ5KDcJPcOA==

 Authorization: SharedKey sally: F5a+dUDvef+PfMb4T8Rc2jHcwfK58KecSZY+l2naIao=

 x-ms-date: Mon, 27 Oct 2008 17:00:25 GMT

 ……… Message Data Contents ………

### REST GetMessages

Below is an example of a REST request for a dequeue operation. Note that the GET HTTP verb is used. Two optional parameters are specified. "numofmessages" specifies the number of messages to retrieve from the queue, up to a maximum of 32. By default, a single message is retrieved from the queue. In the example below, up to 2 messages will be retrieved. "visibilitytimeout" specifies the message's visibility timeout in seconds; the message will remain invisible in the queue during this timeout period, and will reappear in the queue if it has not been deleted by the end of the visibility timeout period. The maximum value of this timeout is 2 hours, and the default value is 30 seconds. In the example below, the visibility timeout is set to 60 seconds. There is also an authorization header inside the HTTP request header as shown below. Note that the response is in XML, and the message data will be base64 encoded in the response (between the <MessageText> </MessageText> tags shown in the example below).

 GET http://sally.queue.core.windows.net/myqueue/messages

 ?numofmessages=2 &visibilitytimeout=60

 HTTP/1.1

 Authorization: SharedKey sally: QrmowAF72IsFEs0GaNCtRU143JpkflIgRTcOdKZaYxw=

 x-ms-date: Thu, 13 Nov 2008 21:37:56 GMT

This call will have a response as in the following example:

HTTP/1.1 200 OK

Transfer-Encoding: chunked

Content-Type: application/xml

Server: Queue Service Version 1.0 Microsoft-HTTPAPI/2.0

x-ms-request-id: 22fd6f9b-d638-4c30-b686-519af9c3d33d

Date: Thu, 13 Nov 2008 21:37:56 GMT

<?xml version="1.0" encoding="utf-8"?>

<QueueMessagesList>

 <QueueMessage>

 <MessageId>6012a834-f3cf-410f-bddd-dc29ee36de2a</MessageId>

 <InsertionTime>Thu, 13 Nov 2008 21:38:26 GMT</InsertionTime>

 <ExpirationTime>Thu, 20 Nov 2008 21:36:40 GMT</ExpirationTime>

 <PopReceipt>AAEAAAD/////AQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2VydmljZS5RdWV1ZU1hbmFnZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49bnVsbAUBAAAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuUXVldWVNYW5hZ2VyLlhBQy5SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nRmllbGQgPFZpc2liaWxpdHlTdGFydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3///8LU3lzdGVtLkd1aWQLAAAAAl9hAl9iAl9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAjSoEmDP8w9Bvd3cKe423ipfNapL7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=</PopReceipt>

 <TimeNextVisible>Thu, 13 Nov 2008 21:38:26 GMT</TimeNextVisible>

 <MessageText>......</MessageText>

 </QueueMessage>

 <QueueMessage>

 <MessageId>2ab3f8e5-b0f1-4641-be26-83514a4ef0a3</MessageId>

 <InsertionTime>Thu, 13 Nov 2008 21:38:26 GMT</InsertionTime>

 <ExpirationTime>Thu, 20 Nov 2008 21:36:40 GMT</ExpirationTime>

 <PopReceipt>AAEAAAD/////AQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2VydmljZS5RdWV1ZU1hbmFnZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49bnVsbAUBAAAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuUXVldWVNYW5hZ2VyLlhBQy5SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nRmllbGQgPFZpc2liaWxpdHlTdGFydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3///8LU3lzdGVtLkd1aWQLAAAAAl9hAl9iAl9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAuX4syrxsEFGviaDUUpO8KNfNapL7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=</PopReceipt>

 <TimeNextVisible>Thu, 13 Nov 2008 21:38:26 GMT</TimeNextVisible>

 <MessageText>.....</MessageText>

 </QueueMessage>

</QueueMessagesList>

### REST DeleteMessage

Below is an example of a REST request for a DELETE message operation. The HTTP verb DELETE is used in this case. The "popreceipt" parameter specifies the message to be deleted. The "popreceipt" is retrieved from a previous dequeue operation as shown in the earlier example.

DELETE /sally/myqueue/messages/6012a834-f3cf-410f-bddd-dc29ee36de2a?popreceipt=AAEAAAD%2f%2f%2f%2f%2fAQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2VydmljZS5RdWV1ZU1hbmFnZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49bnVsbAUBAAAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuUXVldWVNYW5hZ2VyLlhBQy5SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nRmllbGQgPFZpc2liaWxpdHlTdGFydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3%2f%2f%2f8LU3lzdGVtLkd1aWQLAAAAAl9hAl9iAl9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAjSoEmDP8w9Bvd3cKe423ipfNapL7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%3d&timeout=30

HTTP/1.1

Content-Type: binary/octet-stream

x-ms-date: Thu, 13 Nov 2008 21:37:56 GMT

Authorization: SharedKey sally:M/N65zg/5hjEuUS1YGCbVDHfGnI7aCAudkuTHpCDvZY=

# Best Practices

When designing an application for use with Windows Azure Storage, it is important to handle errors appropriately. This section describes issues to consider when designing your application.

## Retry Timeouts and “Connection closed by Host” errors

Requests that receive a Timeout or “Connection closed by Host” response might not have been processed by Windows Azure Storage. For example, if a PUT request returns a timeout, a subsequent GET might retrieve the old value or the updated value. If you see either of these responses, retry the request again with exponential back-off and increasing timeout

Furthermore, some delete operations, such as clear queue messages, are expensive and may take a while to complete. If it cannot complete within the user-specified timeout, the user may get a timeout error back. In these cases, the user should retry the request until it succeeds.

## Tune Application for Repeated Timeout errors

Timeout errors can occur if there are network issues between your application and data center. Over the wide area network, it is recommended to break a single large transfer into a series of smaller calls, and design your application to handle timeouts/failures in this case so that it is able to resume after an error and continue to make progress.

We designed the system to scale and be able to handle a large amount of traffic. However, extremely high rate of requests may lead to request timeouts. In that case, reducing your request rate may decrease or eliminate errors of this type. Generally speaking, most users will not experience these errors regularly; however, if you are experiencing high or unexpected Timeout errors, contact us via Windows Azure forums on MSDN to discuss how to optimize your use of Windows Azure Storage and prevent these types of errors in your application.

## Error Handling and Reporting

The REST API is designed to look like a standard HTTP server and interact with existing HTTP clients (e.g., browsers, HTTP client libraries, proxies, caches, and so on). To ensure the HTTP clients handle errors properly, we map each Windows Azure Storage error to an HTTP status code.

HTTP status codes are less expressive than Windows Azure Storage error codes and contain less information about the error. Although the HTTP status codes contain less information about the error, clients that understand HTTP, but not the Windows Azure Storage errors, will usually handle the error correctly.

Therefore, when handling errors or reporting Windows Azure Storage errors to end users, use the Windows Azure Storage error code instead of the HTTP status code as it contains the most information about the error. Additionally, when debugging your application, you should also consult the human readable <ExceptionDetails> element of the XML error response.

## Choosing the Invisibility Time for GetMessage

Choosing the invisibility time is a tradeoff between expected processing time and application recovery time.

When a message is dequeued, the application specifies the amount of time for which the message is invisible to workers dequeueing messages from the same queue. This time should be large enough to complete the operation specified by the queue message.

If the timeout is too large, the time it takes to finish processing the message is affected when there are failures. For example, if the invisibility time is set at 30 minutes , and the application crashes after 10 minutes, the message will not have a chance of being started again for another 20 minutes.

If the invisibility time is too small, the message may become visible when someone is still processing it. Thus, multiple workers could end up processing the same message, and one may not be able to delete the message from the queue (see the next section). The application could address this as follows

1. If the amount of time to process a message is predictable, set the invisibility timeout large enough so that a message can be completed within that time.
2. Sometimes the processing time for different types of messages may vary significantly. In that case, one can use separate queues for different types of messages, where messages in each queue take a similar amount of time to be processed. Appropriate invisibility timeout value can then be set to each queue.
3. Furthermore, ensure that the operations performed on the messages are idempotent and resume-able. The following can be done to improve efficiency
	1. The processing should be stopped before the invisibility time is reached to avoid redundant work.
	2. The work for a message can be done in small chunks, where a small invisibility time may be sufficient. In this way, the next time the work is picked up from the queue after it becomes visible again, the work can be resumed from where it is left off.
4. Finally, if the message invisibility time is too short and too many dequeued messages are becoming visible before they can be deleted, applications may want to dynamically change the invisibility time that is being set for new messages put onto the queues. This could be detected by counting at the worker roles how many times message deletes are failing due to messages becoming visible. Then based on a threshold communicate that back to the front-end web roles, so they can increase the invisibility time for new messages put into the queue if the invisibility time needs to be tuned.

## Deleting a Message from the Queue

Once the message is processed, it should be deleted from the queue. If the processing completed within the invisibility time, the delete should succeed.

The message should only be deleted after the message has been successfully processed. If it is deleted earlier than this, it is possible that the work for a message will not be completed if the application crashes.

The delete operation could fail if the invisibility time has passed and another process has dequeued the same message. In this case, the application will get an error code back indicating that the invisibility time has passed (it will be a 400 HTTP status code, and the extended error code will be regarding "PopReceipt" not match). The worker role with that stale PopReceipt should then ignore that message and continue, since the message will be deqeued and processed again. The main point here is that the message has to be deleted before its invisibility time has run out.

## Adjusting Worker Roles Based on Queue Length

As described earlier, the queue length directly reflects how well the backend processing nodes are catching up with the overall workload. A growing queue length indicates that the backend servers cannot process the work fast enough. In this case, the application may want to increase the number of backend nodes so that the work can be consumed more quickly. If the queue length always stays near zero, this means there is more processing power at the backend than what the traffic needs. In this case, the application may reduce the number of backend nodes to save resources. Observing the queue length and tuning the number of backend nodes accordingly, applications can effectively scale smoothly accordingly to the amount of traffic.

## Using Binary Format for Messages

Some applications may require storing binary data in the Queue messages. Such applications can call the PutMessage API and send the binary data in the request. However when applications call ‘GetMessages’ or ‘PeekMessages’ to get the messages back, the response containing the Message data, MessageID, as well as the PopReceipt are base64 encoded. Therefore, the application has to decode the response from GetMessages and PeekMessages before using the Message data.