Web Role Service

# Overview:

The clients should be able to upload the photos and the story configuration files to the cloud. We store all those files in blob storage. Since the video encoding may take some time to finish, we use the standard Web Role forefront Worker Role background model.

# Goals:

* Create a service to allow clients upload files.

# None Goals:

* Encode the video.

# Design:

First we must decide which component is responsible for uploading the files. There’re several options:

1. On the phone, use REST API to upload the files to blob storage directly. This requires embedding the storage account key in client applications.
2. Upload the files to the Web Role service, and then store them in blob. This requires more time compared to #1, but does not require the client to know storage account key.
3. Refine #2, use streamed transfer mode. This may take some effort to implement.
4. Use the Web Role service to generate an SAS, and allow the phone clients to upload the files to blob storage directly. This option requires some fine controls over the SAS.

At a minimum, we should not embed our storage account keys in the Windows Phone client applications. So #1 is out of question. The remaining options all have advantages and disadvantages. For small files, such as the configuration file, #2 does not introduce too much latency, especially if the Web Role and the blob storage service are in the same data center. For large files, such as photos, #3 and #4 seems to be nice choices.

In this version, we use #2 to upload the xml configuration file. We don’t require the client to upload a physical file. Instead, client simply puts the xml file content in the request body. We use #4 to upload the photos. To provide better security, we choose a short lived SAS. Currently the decision is 30 minutes, which should be enough to upload all photos inside a typical story.

The web service itself uses WCF Web API, as it is the best solution to create REST services.

Below is an overview of the uploading process:

* Client posts the xml configuration file to the service.
* Service parses the configuration file. For each photo, it generates a blob URI with SAS.
* Service modifies the configuration file to embed the blob URI for each photo (without SAS), and stores it in blob storage, using a unique ID that represents the story.
* Service sends back a response that contains the blob URI for each photo (with SAS) and the unique ID.
* Client uploads the photos to blob storage.
* After all photos are uploaded, client notifies the service it’s done, using the unique ID for the story. This commits the story.
* Service creates a message in queue storage. The message contains the unique ID of the story.
* Worker Role queries the queue and handles the encoding job.

We design the service contract as below:

To upload the configuration file:

Request:

POST http://.../stoies

The request body is the xml configuration file. For example:

<Story PhotoCount="2">

<Photo Name="Blue hills.jpg" PhotoDuration="3">

<Transition Name="Fade Transition" Duration="20" />

</Photo>

<Photo Name="Test.jpg" PhotoDuration="7">

<Transition Name="Fade Transition" Duration="30" />

</Photo>

</Story>

Response:

If the operation succeeds, return 201. The response body is a simple xml file:

<Story ID=”some GUID”>

<Photo Name="Blue hills.jpg" Uri="A blob URI with SAS"/>

<Photo Name="Test.jpg" Uri="A blob URI with SAS "/>

</Story>

If the operation fails, return a proper error code, and the response body contains the error message.

To commit the story:

Request:

PUT http://.../stories/GUID?commit=true

Request body: Empty.

Response: If the operation succeeds, return 204 with an empty body. If it fails, return a proper error code, and the response body contains the error message.

To list the encoded videos:

Request:

GET <http://.../stories/>

Response: If the operation succeeds, return 200 with a JSON response body that describes the videos. If it fails, return a proper error code, and the response body contains the error message. Currently we only support JSON. But in the future, we may need to change the design so that it returns XML or JSON according to the Accept request header.