Andrei Fangli
Class 248

Mini project 1 – Azure Blob Manager

1 OVERVIEW

Microsoft Azure comes with a lot of features, one of them is storage. Storage comes in different forms such as SQL Databases, File Storage, Queues, Tables and Blob storage. The last is of particular interest as it is the best tool to store pictures, videos and other types of data. If one has a website where file uploads are allowed (e.g.: for comments, or screenshots for an issue tracker) then Blob storage would be the best place to store that data.

Blob stands for **B**inary **L**arge **Ob**jects. Sad enough, Microsoft Azure Management Portal does not have a proper Blob Management tool, one can see Blobs per container but cannot upload or decently browse as this storage type is actually flat, there is no hierarchy of directories, there's just a list of names used to identify a particular blob, one can sacrifice a character to obtain a virtual directory hierarchy, but still all blobs are at the same level for Azure.

The goal is clear, in order to use Azure Blob Storage as a place to store files there is need for a proper management tool. One that allows browsing blobs by directory, downloading and uploading, removing them when the user is authenticated.

2 REQUIREMENTS

As a guest it would be useful to view uploaded files into public Azure Blob Containers organized into directories and the date they were uploaded. Items are sorted by their type (directory or blob) and then by name. **Note!** As Azure Blob Containers are flat, slash ('/') is used to delimit item paths. This will help in organizing blobs into virtual containers just as one would use directories in a file system.

As a guest it would be useful to see the path from the root directory to the current directory with the option to navigate to any of the directories listed in the path to ease navigation towards containing directories.

As a guest it would be useful to be able to download files by clicking them or to click on an adjacent button to ease navigation.

As an administrator it would be useful to do all that a guest can do and also to be able to upload new blobs and delete existing ones for maintenance.

As an administrator it would be useful to be able to move files rather than gave to delete them and re-upload them.

As an administrator it would be useful to be able to add new containers and delete existing ones to not involve the Azure Management Portal. Containers can be either public or private.

As an administrator it would be useful to see all containers and whether they are public or not and from there to go to each container and manage it to ease navigation.

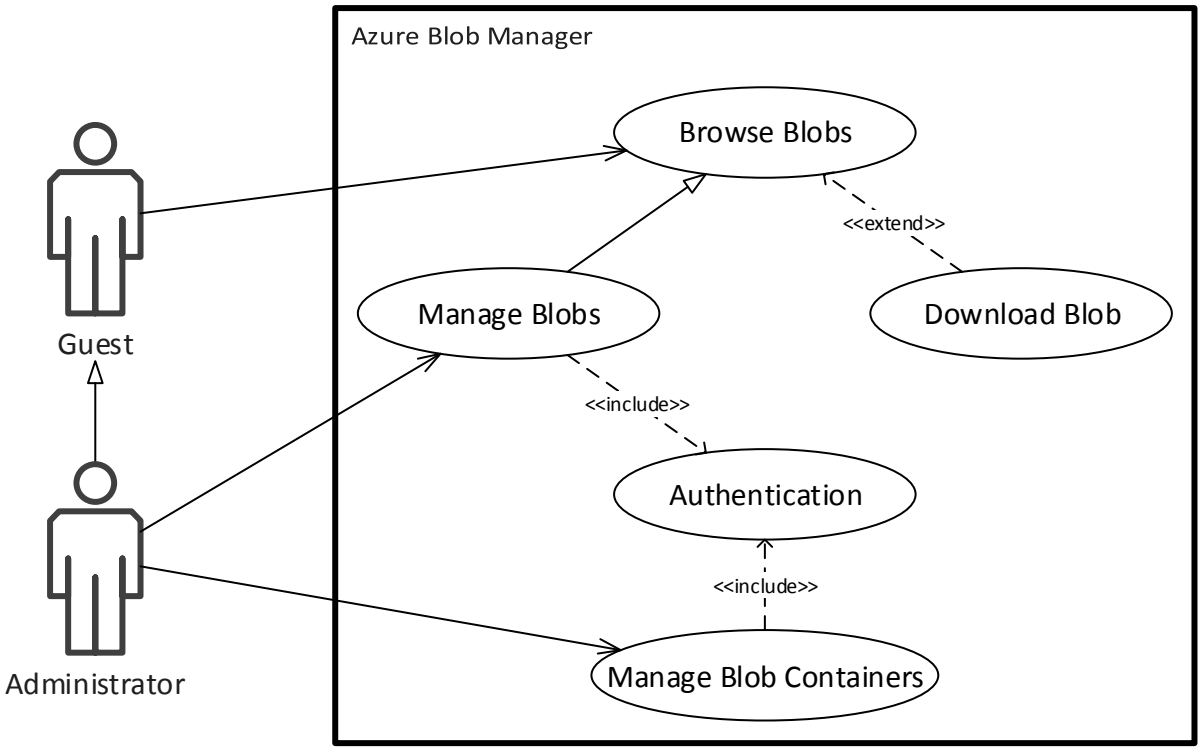
Authentication is made based on Microsoft Azure Storage Account credentials that a guest must enter in order to confirm his rights to change blob containers. A guest is any user that browses public containers.

The application needs to be accessed easily by any user therefore it should be a Web Application.

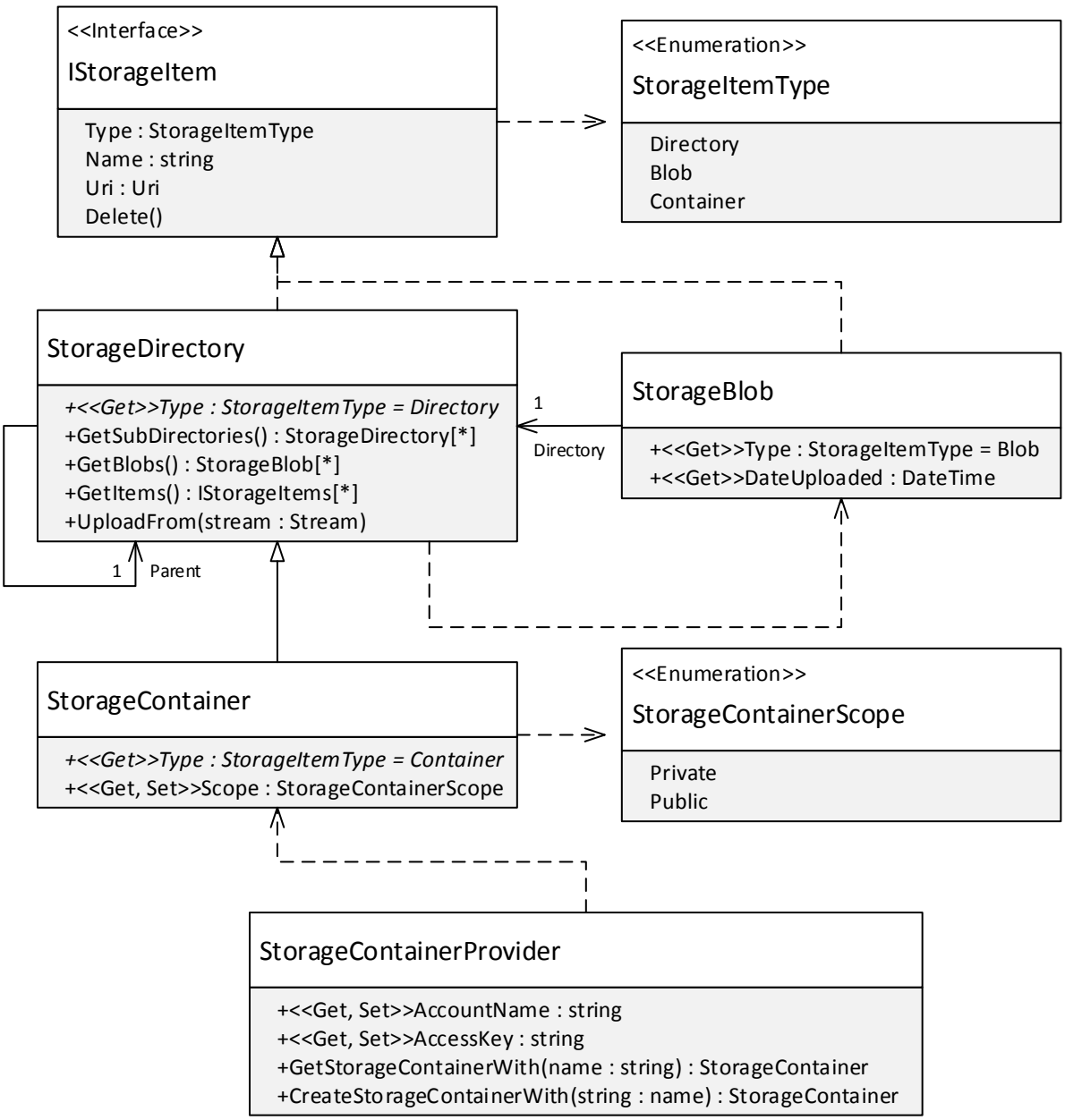
3 ARCHITECTURAL VIEWS

The views selected are *Scenarios* to better illustrate what guests can and cannot do and *Logical View* to illustrate the static structure of the system. The system itself is not very complex as it mostly forwards requests to Microsoft Azure and organizes results. From a behavior point of view the application isn't very rich as all the hard work is done by Azure Web Services.

3.1 SCENARIOS (USE CASES)



3.2 LOGICAL VIEW (CLASS DIAGRAM)



3.3 Design Elements Descriptions

IStorageItem
Represents the interface of any storage item where the type describes what sort of storage item it actually is, the name resembles the directory, container or blob (file) name and the Uri is the web location where the file can be accessed publicly (for download purposes).

StorageItemType
This is a simple enumeration for listing all possible storage item types.

StorageBlob
Represents the actual file. Its Uri provides the web location from which the file can be accessed and downloaded. In case of private containers the file cannot be accessed because Microsoft Azure restricts access to it.

StorageDirectory
Represents a virtual directory. It contains information about how to obtain data from within the current directory such as subdirectories and blobs as well as a reference to its parent directory. In case of root directories these will actually point towards the StorageContainer instance in which they are found.

StorageContainer
Represents the actual root for a set of blobs. Each container is isolated from any other container. Containers have no parent meaning that Parent will hold no reference.

StorageContainerScope
This is a simple enumeration listing all possible scopes for a container (private and public). Instead of an enumeration a Boolean flag could have been used however it would not illustrate the possible scope values as clear as this enumeration.

StorageContainerProvider
This is the entry point. In order to obtain any storage information an account name and access key must be provided. When new StorageContainers are created they are private by default.

Authentication and unauthorized access checks are done by the framework as such mechanisms are already built into ASP.NET MVC.

4 Portability and Performance

With today's devices the application must address the mobile platform one way or another. Smartphones are quite limited on resources and the most important of them all is the battery, on the other hand the way that smart phone operating system are design suggest different approaches than one would have in case of Web Applications. For instance requests to a server are processed in parallel by the web server, a response cannot be sent until all data has been obtained. Besides once a request is sent and the user does not want it anymore it is up to the browser to cancel the request. For mobile devices this is different, in the middle of a request an incoming call may come which will cancel the data transfer, or maybe the user miss tapped and decides to cancel the request. Regardless of the reason for a cancellation, it is a problem that needs to be addressed properly and in time if not immediately.

The mobile version of the application has to be responsive, one approach is to use multiple threads, or to use asynchronous programming.

Asynchronous programming does not necessarily mean multithreading, the bottom line in this programming technique is that the caller does not implicitly wait on an operation to complete thus control is returned to the caller. GUI frameworks often tend to dedicate one thread to the UI and use a queue based execution. As events occur the operating system adds messages to the queue and the UI thread consumes these events. Asynchronous programming can work the same way, any time an operation is called it is in fact placed in the queue and when such operations are awaited the method that contains the await is actually split up in two parts, the part before the await which is placed in the UI execution queue and the second part which is after the await and is enlisted after the asynchronous operation in the UI execution queue. With this approach there is no need for more threads than there already, on the other hand there is no need for synchronization, everything executes synchronously (one operation or part of an operation after another), the order in which operations execute may not be deterministic, however it is still synchronous (a bit of ironic).

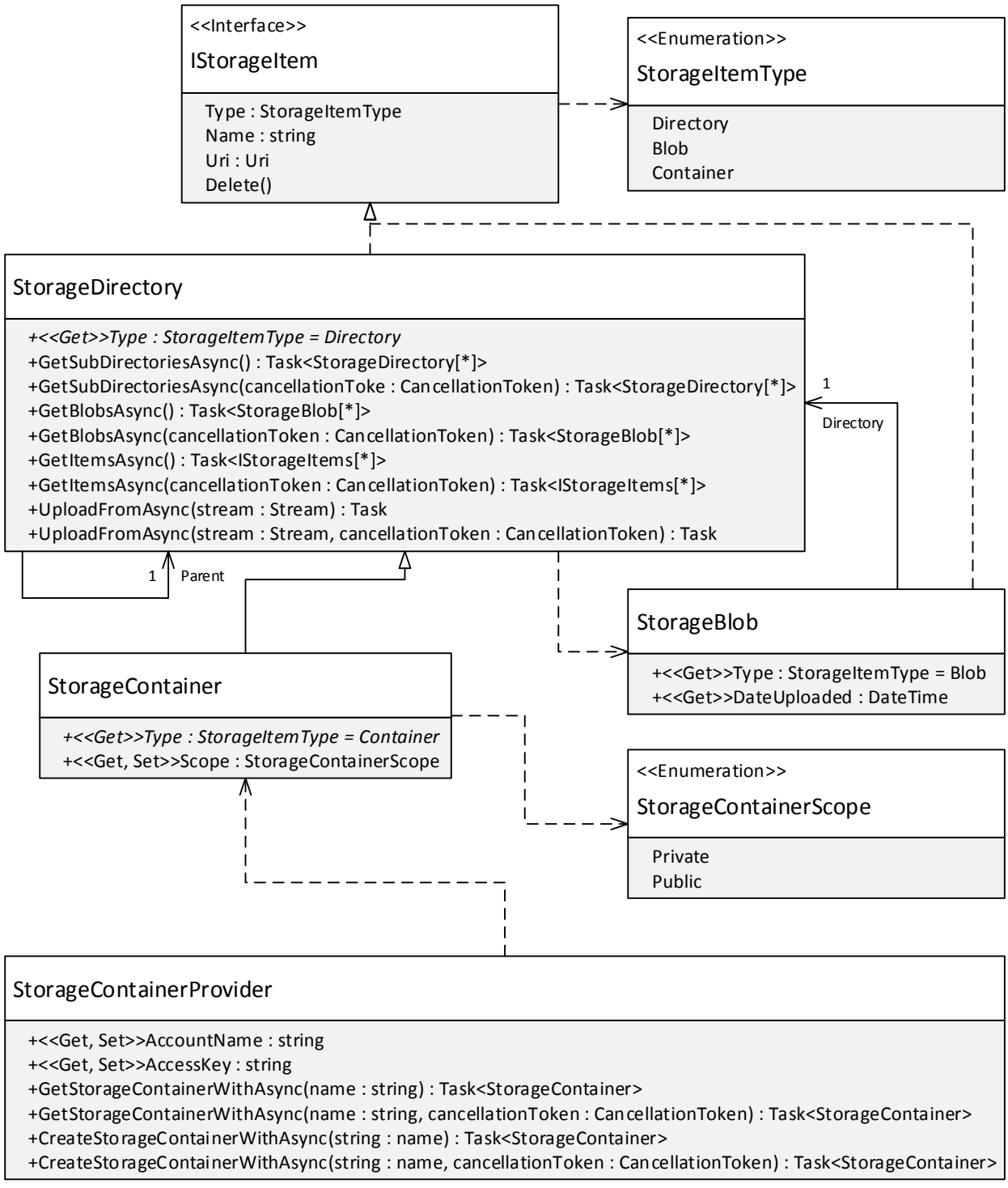
With the asynchronous approach performance is also addressed, assuming the underlying phone framework does not create more threads to fulfil asynchrony then both memory and CPU are used efficiently as there is no need to switch between thread contexts and thread scheduling.

For starters, the application will target only Windows Phone, C# has built in support for asynchronous programming making it easy and also standard to program this way however the provided types hardly support asynchronous programming making it a nightmare to use for Windows Phone applications.

The preferred way of addressing asynchronous programming in C# at this moment is the TAP (Task-based Asynchronous Programming) pattern, meaning that any operations that returns a Task (or any of it subclasses such as Task<TResult>) can be awaited, all the scheduling is done for us through the SynchronizationContext class (in case of GUI it schedules operations one by one in an execution queue like presented earlier). This is good news, to support asynchronous programming operations need only return Tasks, however this is half the solution. CancellationToken in .NET enables cancellation (supported by Tasks), one can use such tokens to probe for cancellation intentions. This means that every operation returning a Task will be overloaded to accept a CancellationToken parameter (TAP guidelines that are followed throughout the .NET Framework). Also every asynchronous operation needs to be suffixed with *Async* (TAP guidelines).

Asynchronous programming is also supported by ASP.NET MVC meaning that the updated design should be available for both Web Applications and Mobile Applications without any migration or code duplication.

The updated design is on the next page.


<<Interface>>
IStorageItem
Type : StorageItemType
Name : string
Uri : Uri
Delete()
<<Enumeration>>
StorageItemType
Directory
Blob
Container
StorageDirectory
+<<Get>>Type : StorageItemType = Directory
+GetSubDirectoriesAsync() : Task<StorageDirectory[*]>
+GetSubDirectoriesAsync(cancellationToke : CancellationToken) : Task<StorageDirectory[*]>
+GetBlobsAsync() : Task<StorageBlob[*]>
+GetBlobsAsync(cancellationToken : CancellationToken) : Task<StorageBlob[*]>
+GetItemsAsync() : Task<IStorageItems[*]>
+GetItemsAsync(cancellationToken : CancellationToken) : Task<IStorageItems[*]>
+UploadFromAsync(stream : Stream) : Task
+UploadFromAsync(stream : Stream, cancellationToken : CancellationToken) : Task
1
Directory
1
Parent
StorageBlob
+<<Get>>Type : StorageItemType = Blob
+<<Get>>DateUploaded : DateTime
StorageContainer
+<<Get>>Type : StorageItemType = Container
+<<Get, Set>>Scope : StorageContainerScope
<<Enumeration>>
StorageContainerScope
Private
Public
StorageContainerProvider
+<<Get, Set>>AccountName : string
+<<Get, Set>>AccessKey : string
+GetStorageContainerWithAsync(name : string) : Task<StorageContainer>
+GetStorageContainerWithAsync(name : string, cancellationToken : CancellationToken) : Task<StorageContainer>
+CreateStorageContainerWithAsync(string : name) : Task<StorageContainer>
+CreateStorageContainerWithAsync(string : name, cancellationToken : CancellationToken) : Task<StorageContainer>