Microsoft

Building an On-Demand Video Service with Microsoft Azure Media Services

David Britch
Martin Cabral
Ezequiel Jadib
Douglas McMurtry
Andrew Oakley
Kirpa Singh
Hanz Zhang

April 2014

patterns & practices

Copyright

Contents

Preface

Microsoft Azure Media Services allows you to build scalable, cost effective, end-to-end media distribution solutions that can upload, encode, package, and stream media to Windows, iOS, Android, Adobe Flash, and other devices and platforms.

The guide describes a scenario concerning a fictitious company named Contoso that has decided to use Azure Media Services to provide a video-on-demand service as an end-to-end solution.

In addition to describing the client applications, their integration with Azure Media Services, and the decisions made during the design and implementation, this guide discusses related factors, such as the design patterns used, and the ways that the application could be extended or modified for other scenarios.

The result is that, after reading this guide you will be familiar with how to design and implement applications that consume Azure Media services.

Who this guidance is for

This guidance is intended for architects, developers, and information technology professionals who design, build, or maintain video-on-demand or online video portal applications and services, particularly those that integrate with a content management systems. To understand the sample code provided with this guidance, you should be familiar with the Microsoft .NET Framework, the Microsoft Visual Studio development system, the Azure SDK for .NET, ASP.NET MVC, and the Microsoft Visual C# development language.

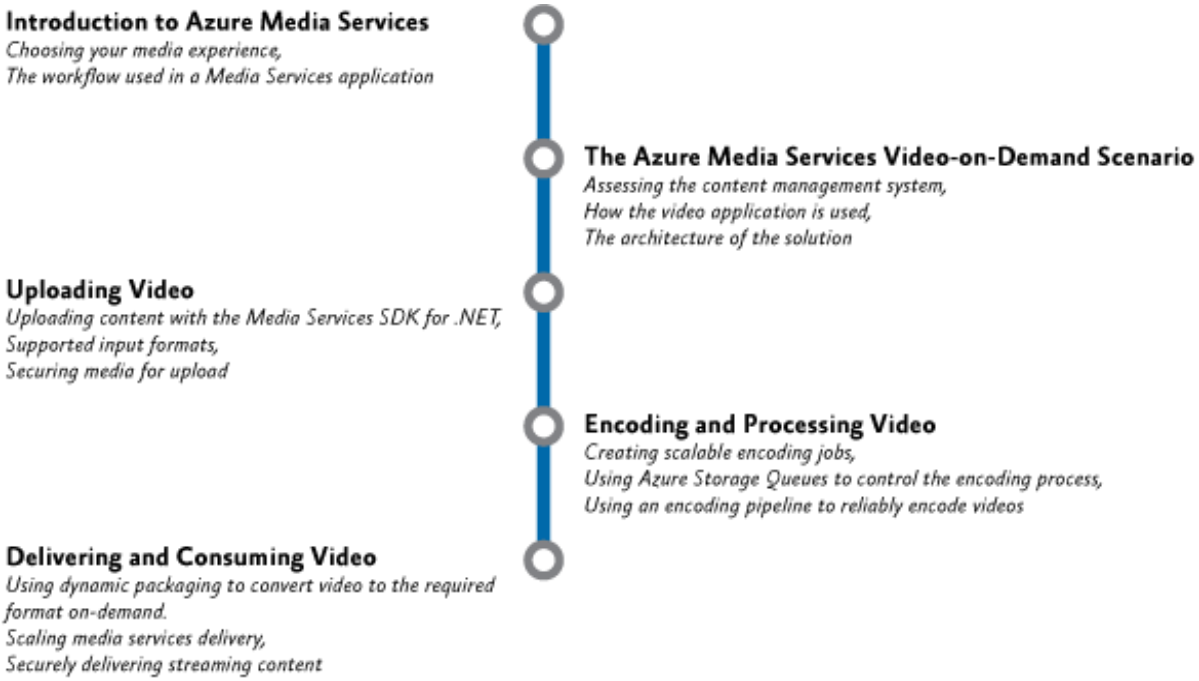
Why this guidance is pertinent now

Building the workflow for the creation, management, and distribution of media is problematic. It involves having to integrate multiple technologies and providers, some of which may be incompatible. In addition, it can require a huge investment in infrastructure, which may not always be fully utilized. These issues can result in a non-standardized workflow that is not easily scaled, and that requires coordination at different stages of the workflow.

Media Services provides everything you'll need to build and operate video-on-demand services to multiple devices and platforms, including all the tools and services you'll need to handle media processing, delivery, and consumption. In addition, Media Services will integrate with content management systems to help your platform scale by using the global footprint of Azure datacenters, without having to plan for capacity spikes or worry about idle datacenters. Together, this helps to reduce the costs that are associated with integrating multiple products and providers when building a media solution.

How this guidance is structured

The following figure shows the road map for the guide.



The guide structure

Chapter	Summary
Chapter 1, "Introduction to Microsoft Azure Media Services"	This chapter provides an overview of the workflow used by Media Services, and discusses how to decide what type of media experience users should have.
Chapter 2, "The Azure Media Services Video-on-Demand Scenario"	This chapter describes the video content management system developed by Contoso, and the business requirements of the video applications, and summarizes the architecture of the solution that Contoso built, based on a web service that's consumed by client applications.
Chapter 3, "Uploading Video into Microsoft Azure Media Services"	This chapter describes the input formats supported by Media Services, how to use the Media Services SDK for .NET to upload content, and how to secure media for upload.
Chapter 4, "Encoding and Processing Video in Microsoft Azure Media Services"	This chapter focuses on encoding media, examining how to encode media for efficient delivery, how to create scalable encoding jobs, and how to control the encoding process by using Azure Storage Queues.
Chapter 5, "Delivering and Consuming Video from Microsoft Azure Media Services"	This chapter provides describes how to use dynamic packaging to convert video to the required format on-demand, how to scale media services delivery, and how to securely deliver streaming content to the end user.

Note: This guide also includes appendices that describe how the web service works, and task presets you can use to configure the Azure Media Encoder.

What you need to use the code

These are the system requirements for building and running the sample solution:

- Microsoft Windows 8.1.
- Microsoft Visual Studio 2013 Ultimate, Premium, or Professional edition.
- Azure SDK for .NET.
- Windows Phone SDK 8.0.
- Microsoft Internet Information Server (IIS).
- A Media Services account in a new or existing Azure subscription.

You can download the sample code from http://aka.ms/amsg-code.

Who's who?

This guidance uses a sample application that illustrates consuming Media Services. A panel of experts comments on the development efforts. The panel includes a mobile app specialist, a software developer, a database specialist, and a cloud specialist. The delivery of the sample application can be considered from each of these points of view. The following table lists these experts.

	**Christine** is a mobile application specialist. She understands the special requirements inherent in applications designed to be used on mobile devices. Her expertise is in advising architects and developers on the way they should plan the feature set and capabilities to make the application usable and suitable for these types of devices and scenarios. *"To build successful applications that work well on the phone, you must understand the platform, the user's requirements, and the environment in which the application will be used."*
	**Markus** is a senior software developer. He is analytical, detail oriented, and methodical. He's focused on the task at hand, which is building a great cloud-based application. He knows that he's the person who's ultimately responsible for the code. *"For the most part, a lot of what we know about software development can be applied to different environments and technologies. But, there are always special considerations that are very important."*
	**Poe** is a database specialist. He is an expert on designing and deploying databases. Poe has a keen interest in practical solutions; after all, he's the one who gets paged at 03:00 when there's a problem. *"Implementing databases that are accessed by thousands of users involves some big challenges. I want to make sure our database performs well, is reliable, and is secure. The reputation of Contoso depends on how users perceive the applications that access the database."*
	**Bharath** is a cloud specialist. He checks that a cloud-based solution will work for a company and provide tangible benefits. He is a cautious person, for good reasons. *"The cloud provides a powerful environment for hosting large scale, well-connected applications. The challenge is to understand how to use this environment to its best advantage to meet the needs of your business."*

If you have a particular area of interest, look for notes provided by the specialists whose interests align with yours.

Community

This guide, like many patterns & practices deliverables, is associated with a community site. On this community site, you can post questions, provide feedback, or connect with other users for sharing ideas. Community members can also help Microsoft plan and test future guides, and download additional content such as extensions and training material.

Authors and contributors

This guide was produced by the following individuals:

- Program and Product Management: Andrew Oakley (Microsoft Corporation)
- Development: Martin Cabral (Southworks SRL), Ezequiel Jadib (Southworks SRL), Douglas McMurtry (Agilethought Inc.), Hanz Zhang (Microsoft Corporation)
- Test: Monika Jadwani (Tata Consultancy Services), Sumit Jaiswal (Tata Consultancy Services), Gurunath Navale (Tata Consultancy Services), Kirpa Singh (Microsoft Corporation)
- Documentation: David Britch (Content Master Ltd)
- Edit: RoAnn Corbisier (Microsoft Corporation)
- Illustrations and book layout: Chris Burns (Linda Werner & Associates Inc)
- Release Management: Nelly Delgado (Microsoft Corporation)

1 - Introduction to Microsoft Azure Media Services

Traditionally, building the workflow for the creation, management, and distribution of media is problematic. It involves having to integrate multiple technologies and providers, some of which may be incompatible. In addition, it can require a huge investment in infrastructure, which may not always be fully utilized. These issues can result in a non-standardized workflow that is not easily scaled, and that requires coordination at different stages of the workflow.

This chapter introduces Microsoft Azure Media Services, and discusses the typical Media Services workflow.

What is Microsoft Azure Media Services?

Azure Media Services allows you to build scalable, cost effective, end-to-end media distribution solutions that can upload, encode, package, and stream media to Windows, iOS, Android, Adobe Flash, and other devices and platforms.

The benefits that Media Services offers over the traditional approach to building a media workflow are as follows:

- An API that allows developers to easily create, manage, and maintain custom media workflows.
- A standardized workflow that improves coordination and productivity when there are multiple participants involved in the creation and management of content.
- Automatic scalability by using global data centers to transcode and deliver media assets, without having to plan for capacity spikes or worry about idle datacenters.
- Cost effectiveness by encoding media once, and then using dynamic packaging to deliver it in multiple formats.

Media Services provides everything you'll need to easily build and operate the three standard media solutions:

- **Video-on-demand (VOD) services**. Media Services provides everything you'll need to operate VOD services to multiple devices and platforms, including all the tools and services you'll need to handle media processing, delivery, and consumption.
- **Online video platforms (OVP)**. Media Services will integrate with your OVP and Content Management System (CMS) to help your platform gain scale by using the global footprint of Azure datacenters, without having to plan for capacity spikes or worry about idle datacenters.
- **End-to-end solutions**. Media Services can be used to easily build secure end-to-end media workflows entirely in Azure, from content ingestion through to encoding, packaging, and

protection. This helps to reduce the costs that are associated with integrating multiple products and providers.

Regardless of your development scenario, the first step in sharing video content is to choose your media experience.

To see how companies are using Media Services to stream video to their customers, read the following case studies.

- all3media
- blinkbox
- Xbox

Choosing your Azure Media Services video experience

The first step in sharing video content is deciding what type of experience you want your users to have. This can be answered by asking a number of questions:

- How will your users be viewing the video content?
- Will your users be connected to the internet?
- Will your users expect the video content to be in HD?
- Will your users be viewing the video content on a computer or a hand-held device?

Providing answers to these questions will help to give your users the best possible experience.

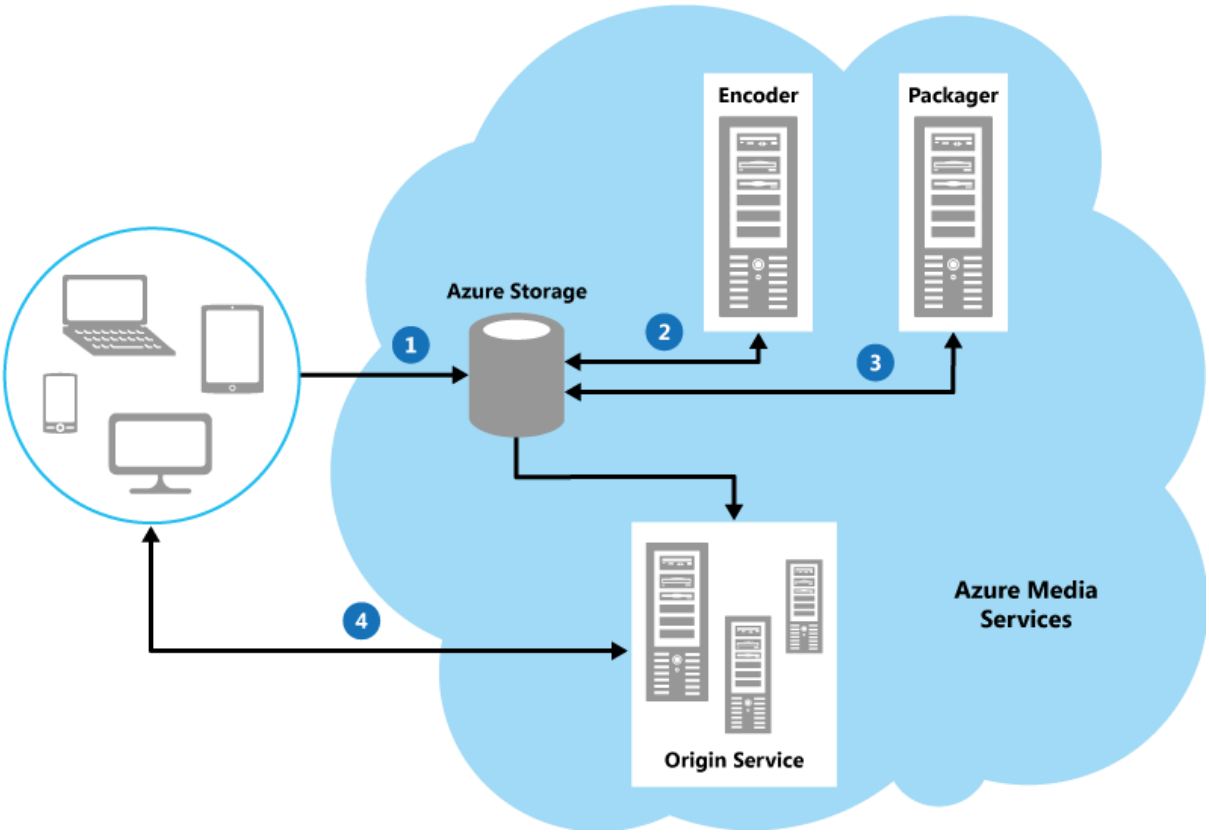
Another decision that must be made is the type of viewing devices that you will support. The following table outlines the viewing devices supported by Media Services, and the recommended viewing approaches for those devices.

Device	Description	Recommended viewing approaches
Web browsers	Web browsers can be run on desktop PCs, smart phones, and tablets. When running on desktop PCs you can take advantage of the large screen size and the large storage capacity, allowing you to stream HD videos.	Offline viewing, progressive downloading, and streaming.
Smart phones	Smart phones have small screens and small storage capacities.	Streaming.
Tablets	Tablets have larger screens than smart phones, but still typically have smaller storage capacity.	Streaming. Tablets with larger storage capacities can take advantage of offline viewing and progressive downloading.
Xbox	Xbox consoles have the benefit of large screens and large storage capacity.	Offline viewing, progressive downloading, and streaming.
Set-top boxes	These devices typically have large screens but	Streaming.

and connected TVs	minimal storage capacity.	

Organizing the Azure Media Services video processing workflow

The following figure shows a high-level overview of the standard workflow used when processing media with Media Services.



A high-level overview of the standard Media Services workflow

> Media Services supports on-demand media and live streams workflows. However, the live streams workflow is outside the scope of this guide. Therefore, the guide focuses on the on-demand media workflow.

The steps involved in the workflow are as follows:

1. Media is uploaded to Media Services and stored in Azure Blob Storage. service
2. Uploaded media is encoded using the Azure Media Encoder, with the encoded media being stored in Azure Storage.
3. Encoded media is packaged by the Azure Media Packager, with the result being stored in Azure Storage.
4. Client applications playback the media located at a URL, with the Origin Service processing the outbound stream from storage to client application.

Therefore, the typical Media Services workflow can be summarized as:

1. Media upload
2. Media processing
3. Delivery
4. Consumption

Each item will now be discussed in turn.

Uploading video into Azure Media Services

You must upload your content into Azure Media Services in order to be able to encode, manage, and consume it. Media Services uses Azure Storage to store your media for processing and viewing. Your content can be programmatically uploaded using the Media Services REST API or one of the available client SDKs. These APIs allow you to upload one file at a time or perform bulk upload operations. Media Services also allows you to perform secure uploading and storage of your content. Storage encryption will encrypt your content locally prior to uploading it to Azure Storage where it will be stored in an encrypted form.

The fundamental content in Media Services is an asset. An asset contains one or many files, such as video, audio, closed caption files, and metadata about the files. Each asset contains one or more asset files, with each asset file containing metadata about a specific media file. Once an asset has been created by uploading files it can be used in Media Services workflows such as encoding and streaming.

Each asset is mapped to a blob container in an Azure Storage account, with the files in the asset being stored as blobs in the container. A blob container groups a set of blobs, just as a folder groups a set of files. They are used in Media Services as a boundary point for access control. An Azure Storage account can contain an unlimited number of blob containers and a container can store an unlimited number of blobs.

Media Services accounts are associated with one or more Azure Storage accounts. Each account can contain an unlimited number of blob containers, and is only subject to the limits on the underlying account. Media Services provides SDK tooling to manage multiple storage accounts and perform load balancing of the distribution of assets during upload. For more information see "Managing assets across multiple storage accounts in Azure Media Services and defining load balancing strategy."

For more information about uploading content, see "Chapter 3 – Uploading Video."

Supported file types in Azure Media Services

Various video, audio, and image file types can be uploaded to a Media Services account, with there being no restriction on the types or formats of files that you can upload using the Media Services SDK. However, the Azure Management portal restricts uploads to the formats that are supported by the Azure Media Encoder. These import formats include MPEG-1, MPEG-2, MPEG-4, and Windows Media Video encoded video, MP3, WAVE, and Windows Media Audio encoded audio, and BMP, JPEG, and PNG encoded images. The Azure Media Encoder can export data as Windows Media Video, Windows Media Audio, MP4, and Smooth Streaming File Format.

For more information about the supported file formats see "Supported input formats" and "Introduction to encoding."

Processing media with Microsoft Azure Media Services

In Media Services, media processing involves obtaining a Media Processor instance, encoding, packaging, and protecting media files.

Obtaining a media processor instance

Media Services provides a number of *media processors* that enable video to be processed. Media processors handle a specific processing task, such as encoding, format conversion, encrypting, or decrypting media content. Encoding video is the most common Media Services processing task, and it is performed by the Azure Media Encoder.

Encoding video

Encoding is the process of taking a video and turning it into a format that can be consumed by users. Users could be using a variety of devices to watch your videos, including desktop computers, smart phones, tablets, Xbox consoles, set-top boxes, or Internet-connected TVs. These devices all have features that affect the required encoding. For instance, smart phones have small screens and little storage, while desktop computers have larger screens and larger storage. In addition, smart phones potentially have a more limited bandwidth than desktop computers. Therefore, when you choose how to encode a video you must bear in mind the variety of devices that users will consume the video on.

> In some cases you may want to have multiple encodings to enable the best possible experience on a range of devices.

The Media Encoder is configured using encoder preset strings, with each preset specifying a group of settings required for the encoder. Encoder presets are divided into two groups – general presets and device specific presets. Videos encoded with general presets can be used by any device that supports the required file formats. Videos encoded with device specific presets are designed to be used by a specific device, such as a smart phone. For a list of all the presets see "Appendix B –Azure Media Encoder Presets."

Encoding for a smart phone

When encoding video for a smart phone you should choose an encoding preset that matches the resolution, supported codecs, and supported file formats of the target device. For example, for a Windows Phone that supports H.264 video up to 1080p you should use the "H264 Smooth Streaming Windows Phone 7 Series" preset.

> Different smart phones, even those from the same company, can support different resolutions, bit rates, codecs, and file formats.

The iPhone 5 supports H.264 video up to 1080p in HLS format. However, Media Services does not support encoding video directly into HLS but you can encode to MP4 and then use static packaging to convert the video to HLS. Alternatively, you can encode to Smooth Streaming or MP4 and use dynamic packaging to convert the video to HLS in real-time. Therefore, if you wanted to encode

video to 1080p for an iPhone 5 you would use the "H264 Adaptive Bitrate MP4 Set 1080p for iOS Cellular Only" preset. Similarly, for an Android phone that supports H.264 video at 480x360 you could use the "H264 Adaptive Bitrate MP4 Set SD 4x3 for iOS Cellular Only" preset. Then dynamic packaging would be used to convert the video to HLS in real-time.

Encoding for Xbox

When encoding video for Xbox you can choose between VC-1 and H.264 smooth streaming at resolutions up to 1080p. For example, to encode a video to 720p using H.264 you would use the "H264 Smooth Streaming 720p Xbox Live ADK" preset.

Encoding for other devices and platforms

General presets can be used to encode for a variety of devices including desktop machines, tablets, and set-top boxes. To choose the appropriate encoding preset you must determine how users will view your content, and what resolutions, bit rates, codecs, and file formats are supported on their viewing devices.

The following table lists each type of device and the client technologies supported by Media Services.

Device	Technologies
Windows 8	Smooth streaming, progressive downloading, MPEG-DASH.
Windows RT	Smooth streaming and progressive downloading.
Windows Phone	Smooth streaming and progressive downloading.
Web browsers	Smooth streaming is supported through additional SDKs and Player Frameworks provided by Microsoft. Progressive download is supported in browsers through the HTML5 **video** element. Internet Explorer 11 and Chrome both support MPEG-DASH through the use of Media Source Extensions (MSE).
Xbox	Smooth streaming and progressive downloading.
Macintosh	Apple HLS and progressive download.
iOS	Smooth streaming, Apple HLS, and progressive downloading.
Android	Smooth streaming, progressive downloading, and Apple HLS.
Set-top box, connected TVs	Smooth streaming, progressive downloading, and Apple HLS.

For more information about encoding media with Media Services see "Chapter 4 – Encoding and Processing Video."

Packaging video with Azure Media Services

Once a video has been encoded it is placed in an output asset, which can then be placed into a variety of file containers. This process is referred to as *packaging*. For example, you could convert an MP4 file into smooth streaming content by using the Azure Media Packager to place the encoded content into a different file container.

Note: Packaging does not re-encode a video. Instead, it rearranges the encoding and places it in a different file container.

Media Services allow the user to decide if they will package video upfront with a media processor, known as *static packaging*, or package video on demand, known as *dynamic packaging*.

The Azure Media Packager is a media processor capable of performing static packaging. Static packaging involves creating a copy of your content in each format required by users. For example, an MP4 file could be converted into smooth streaming content if both formats are required by users. This would result in two copies of the content existing, each in a different format.

Dynamic packaging is not performed by a media processor, but by *origin servers*. An origin server packages the source media when a client application requests a specific video format, allowing you to encode your video just once, with it being converted in real time to the format requested by the client application. With dynamic packaging your video is typically stored as an adaptive bitrate MP4 file set. When a client application requests the video it specifies the required format. The origin server then converts the MP4 adaptive bitrate file to the format requested by the client in real time. This ensures that only one copy of your video has to be stored, therefore reducing the storage costs.

Dynamic packaging is the preferred method for publishing a video. For more information see "Dynamic packaging."

Protecting video with Azure Media Services

To protect your media when it is published, Media Services supports PlayReady sample-based Common Encryption and AES 128-bit CBC Envelope Encryption. PlayReady is a Digital Rights Management (DRM) system developed by Microsoft. DRM allows you to control who has access to your content. When a user tries to watch PlayReady protected content, the client application requests the content from Media Services. Media Services then redirects the client to a licensing server that authenticates and authorizes the user's access to the content. The client application can then safely download the decryption key which will allow the content to be decrypted and viewed.

AES Envelope Encryption provides content encryption, but does not allow sophisticated digital rights management, or secure key delivery (which is provided only by SSL). Content owners should trust their clients if they choose AES 128-bit Envelope Encryption. It is much easier for an untrusted, malicious end user to acquire and redistribute keys.

For more information about content protection see "Protecting Assets with Microsoft PlayReady."

Delivering video from Azure Media Services

Media Services provides different mechanisms for delivering media assets that have been uploaded to Media Services. It can be used to deliver content that has simply been stored in Media Services, or it can also include content that has been processed or encoded in different ways.

There are typically four approaches that users can use to access videos:

- Offline viewing
- Progressive downloading
- Streaming

- Adaptive bitrate streaming

Offline viewing involves a user downloading an entire video onto their computer or device. Because videos can be quite large, it may take some time for the download to complete, and the device must have enough storage space to hold the entire video. However, the benefit of this approach is that you do not need a network connection to view the video once it has been downloaded.

Progressive downloading allows a user who is connected to the internet to start viewing a video before the entire video has been downloaded. However, it does require that the viewing device has enough storage space to hold the entire video.

Streaming also requires an internet connection, but differs from progressive downloading in that it only downloads a small amount of the video at once and discards it once it has been displayed. The benefit of this approach is that it requires little storage on the viewing device.

Adaptive bitrate streaming allows client applications to determine network conditions and adapt the data rate of the video content to the available network bandwidth. When network communication degrades, the client can automatically select a lower bitrate version of the content, therefore allowing the user to continue viewing the video, albeit at a lower quality. When network conditions improve the client can automatically switch back to a higher bitrate with improved video quality. The benefit of this approach is that the video player can automatically react to changes in bandwidth during playback, therefore providing a better user experience.

> **Note:** Some adaptive bitrate streaming technologies, such as Smooth Streaming, also monitor video rendering performance in order to determine the appropriate bitrate stream for the client.

Media Services supports three adaptive bitrate streaming technologies:

- Smooth Streaming. This is an adaptive bitrate streaming technology developed by Microsoft.
- HTTP Live Streaming (HLS). This is an adaptive bitrate streaming technology developed by Apple.
- MPEG DASH. This is an adaptive bitrate streaming protocol created by the Motion Picture Experts Group (MPEG), and is an international standard.

Processing outbound video from Azure Media Services

The Media Services Origin Service handles requests for content. It retrieves files from Azure Storage and provides them to Content Delivery Networks (CDN) or client applications directly. The origin servers have features that allow them to respond to several hundred requests per second, and provide dynamic encryption and dynamic packaging services.

Providing access to video within Azure Media Services

Accessing content in Media Services requires a *locator*, which provides an entry point to access the files contained in an asset. An access policy is used to define the permissions and duration that a client has access to a given asset. Multiple locators can share an access policy so that different locators can provide different start and stop times while all using the same permission and duration settings provided by the access policy.

There are two types of locators:

- Shared access signature locators
- On-demand origin locators

A shared access signature locator grants access rights to the underlying blob container of the media asset in Azure Storage. By specifying a shared access signature, you can grant users who have the URL access to a specific resource for a specified period of time. You can also specify what operations can be performed on a resource that's accessed via a shared access signature locator.

Note: Media Services enables the authoring of shared access signature locators to simplify complex workflows. However, it is not expected that end-users will consume these except in special cases.

An on-demand origin locator should be used to grant access to streaming content. On-demand origin locators are exposed by the Media Services Origin Service, which pulls the content from Azure Storage and delivers it to the client. The on-demand origin locators obfuscate the underlying asset's blob container and storage account URL. Instead, they always point to a Media Services Origin Service, therefore allowing advanced scenarios such as IP restriction, cache control, and CDN authentication. For more information about the Media Services Origin Service see "Origin Service."

Consuming video from Azure Media Services

Media Services provides support for creating media player applications that run on different devices and platforms including PCs, Macintosh, Windows Phone, iOS devices, and Android devices. Microsoft also provides many different SDKs and player frameworks that allow you to create applications that consume streaming media from Media Services. For more information see "Developing Azure Media Services Client Applications."

Summary

Media Services provides everything you'll need to build and operate video-on-demand services to multiple devices and platforms, including all the tools and services you'll need to handle media processing, delivery, and consumption. In addition, Media Services will integrate with content management systems to help your platform scale by using the global footprint of Azure datacenters, without having to plan for capacity spikes or worry about idle datacenters. Together, this helps to reduce the costs that are associated with integrating multiple products and providers when building a media solution.

More information

- The article "Managing assets across multiple storage accounts in Azure Media Services and defining load balancing strategy" is available on a blog site.
- You can find information about content protection at "Protecting Assets with Microsoft PlayReady."
- You can find more information about the SDKs and Player Frameworks that allow you to create client applications that can consume streaming media from Media Services at "Developing Azure Media Services Client Applications."

2 - The Microsoft Azure Media Services Video-on-Demand Scenario

Contoso is a startup ISV company of approximately 20 employees that specializes in developing solutions using Microsoft technologies. The developers at Contoso are knowledgeable about various Microsoft products and technologies, including the .NET Framework, Microsoft Azure, the Windows Runtime, and Windows Phone.

Contoso has been contracted to develop a video-on-demand service as an end-to-end solution. The service must work with multiple devices and platforms, with the client application for the Windows Runtime expecting the highest usage. While the primary purpose of the client applications is to consume videos whose details are stored in a Content Management System (CMS), the applications are also required to be able to capture and upload new videos for encoding by Azure Media Services, which can then be consumed from the CMS once encoding has completed. Access to the CMS and Media Services is through the Contoso web service.

This chapter describes the business requirements of the Contoso Video applications, and summarizes the architecture of their solution.

Using the Contoso Azure Media Services video application

The purpose of the Contoso video application is to enable users to consume video-on-demand, stored in the cloud, from a CMS. The customer's experience is paramount, so the designers at Contoso chose to implement the applications as a series of easy to use, mobile applications. The Contoso video application is available for the Windows Runtime, Windows Phone 8, iOS, Android, and the web.

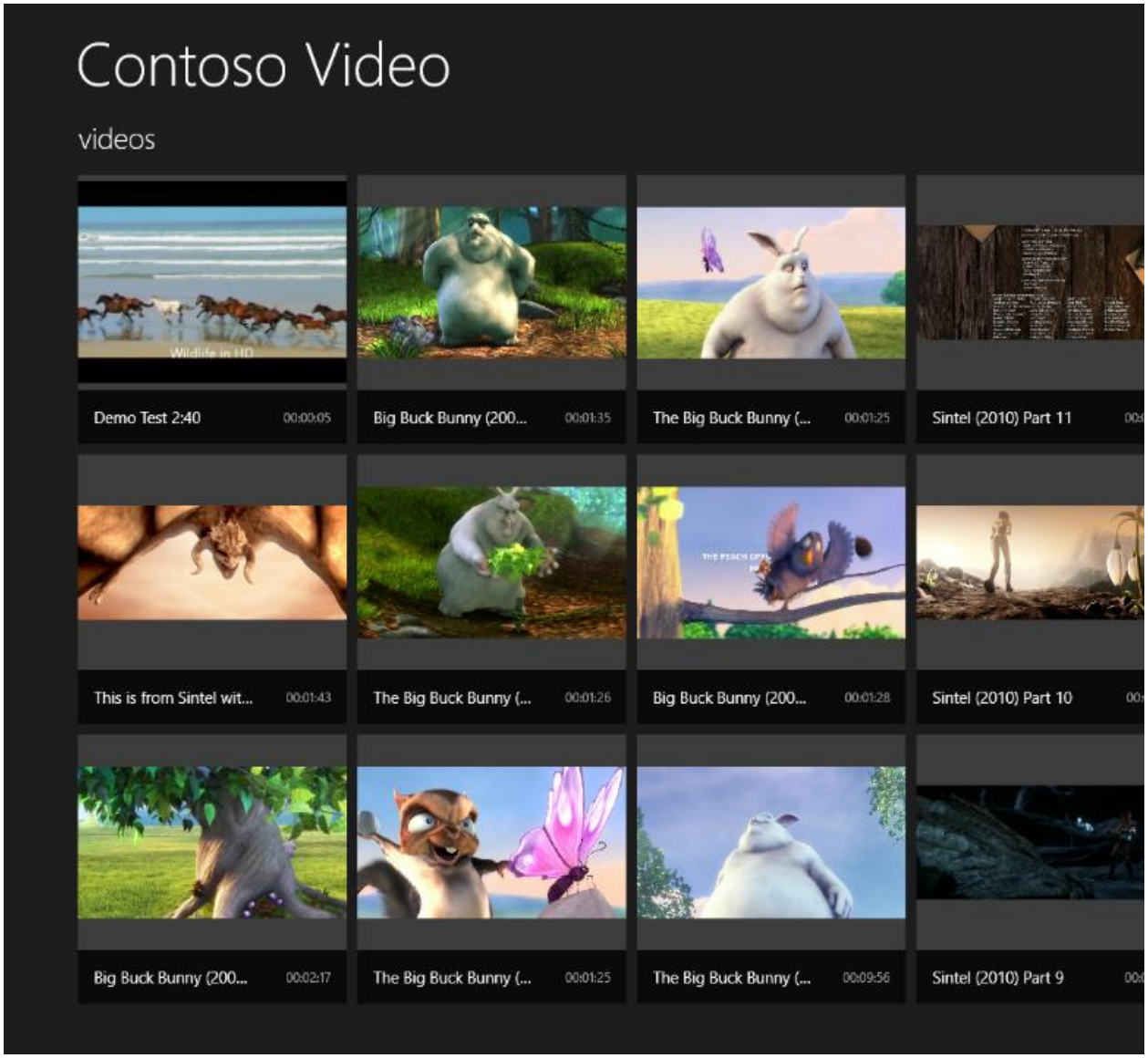
The video application implements the features typical of many video-on-demand services. It enables users to browse videos, and control the playback of selected videos. When viewing videos the application also suggests other related items that the user may want to view.

> Contoso expects that the Windows Store video application will gain the highest usage among their clients' customers. Therefore, this guide focuses on the Contoso Windows Store video application, and its interaction with the Contoso web service.

The following sections describe the primary business use cases of the Windows Store Contoso video application in more detail.

Browsing videos

When the user starts the application the first page displays thumbnails for each video that can be viewed, as shown by the following screenshot.



Thumbnails for each video that can be viewed

Users can use pointing devices or touch gestures to browse these thumbnails. Clicking on a thumbnail navigates to a new page from where the video can be viewed. Alternatively users can use the bottom app bar to capture a new video or upload an existing video.

Users prefer an application that fits well with the device's design and theme. You will also have to comply with certain UI design guidelines if you want to distribute your application through the Windows Store.

For information on how the browsing video use case is implemented, see "Chapter 5 – Delivering and Consuming Video."

Playing videos from Azure Media Services

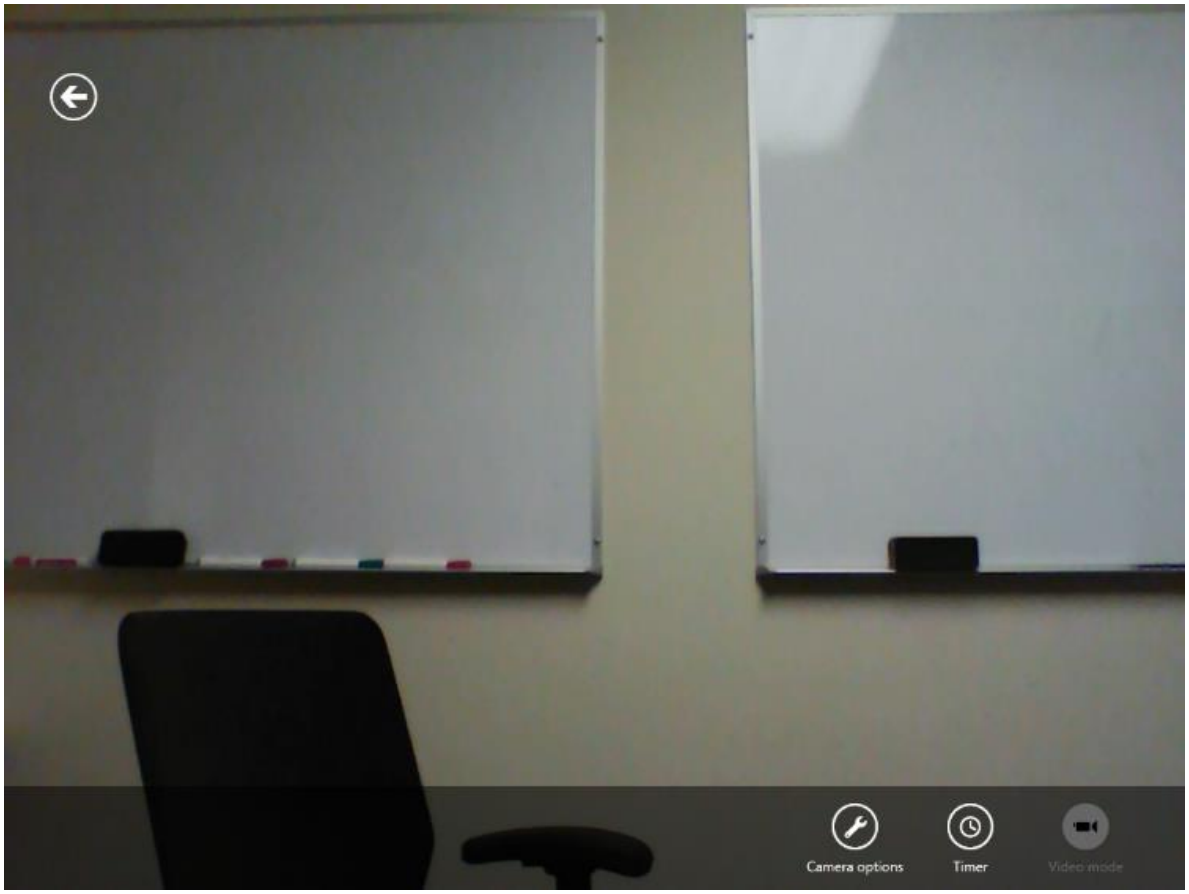
To play videos users must have selected a video on the initial page of the application. Users can then control the playback of videos using pointing devices or touch gestures, as shown in the following figure.

Video playback

As well as controlling the playback of the video, basic information is also displayed to users, including the duration of the video, a description of it, and a list of related videos.

Capturing videos

The video application also allows users to capture video, which they can then choose to upload to Media Services for encoding, prior to making it available for consumption for other users, as shown in the following figure.



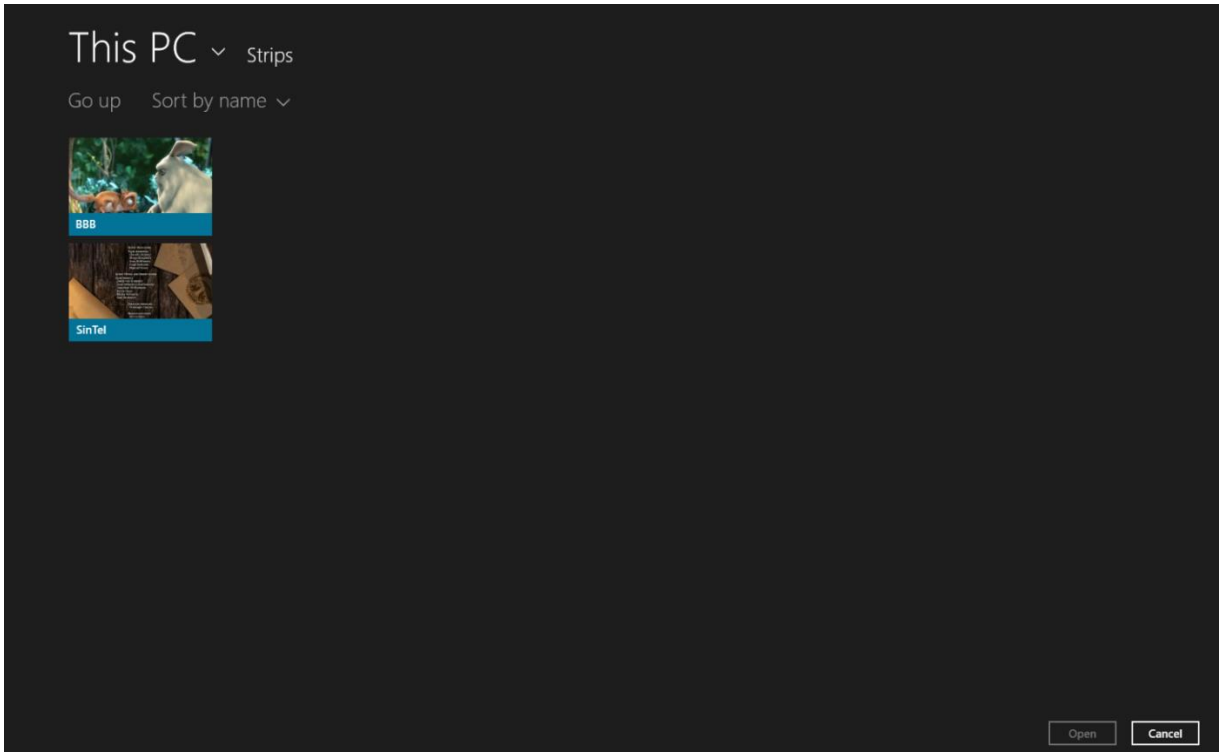
Capturing video

Users can configure their camera options, set a timed recording, and configure the recording mode of the camera.


You should always be aware of how your application consumes the resources on a device, such as bandwidth, memory, and battery power. These factors are far more significant on mobile devices than on the desktop.

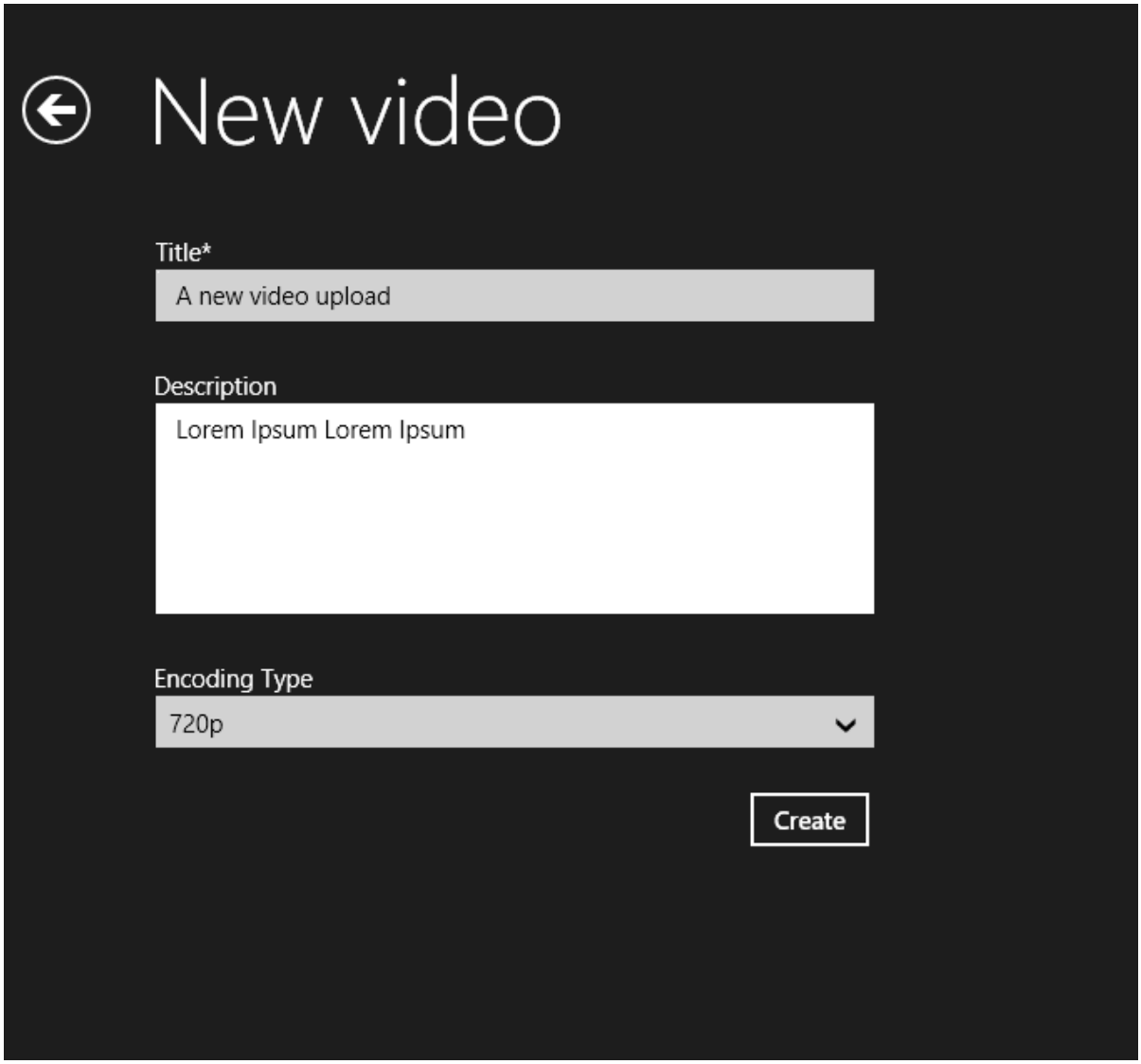
Uploading videos into Azure Media Services

User can choose to upload media stored on their device to Media Services for encoding, prior to making it available for consumption by all users. Videos can be stored anywhere in the file system, and can either have been captured on the device, or downloaded to the device from other locations. The following figure shows the page that allows users to choose a video to upload for encoding.



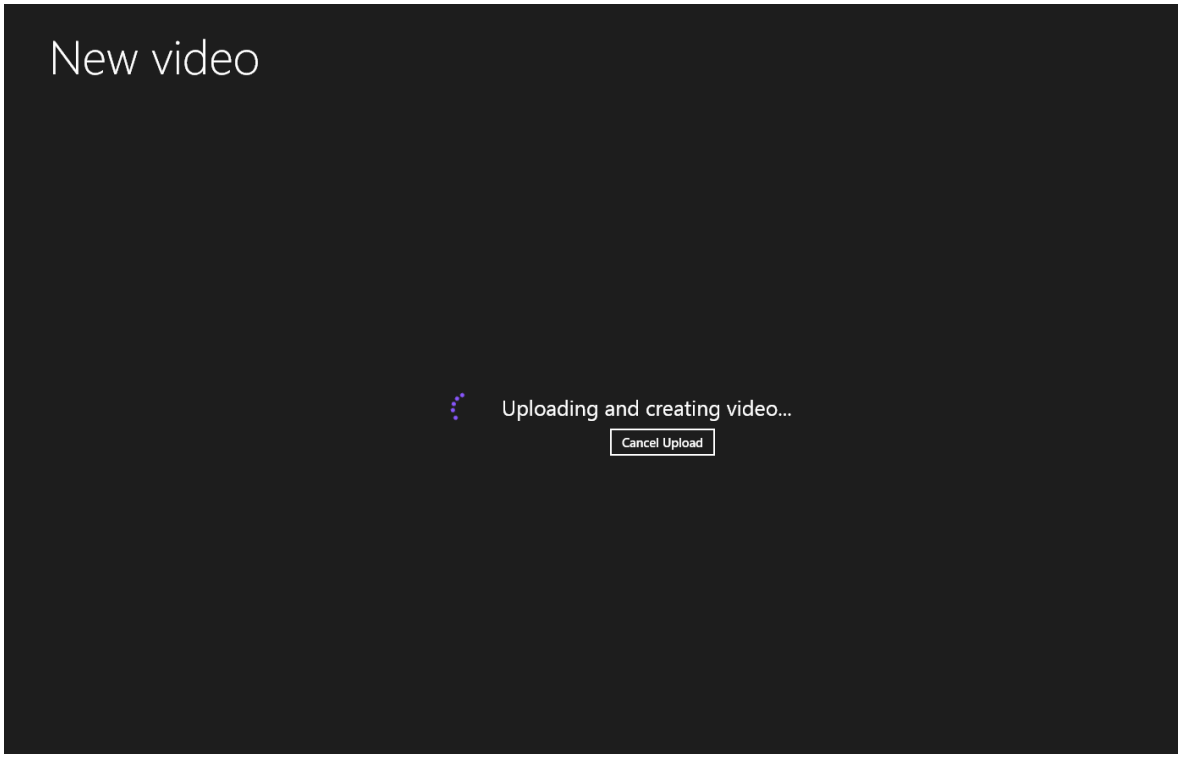
Choosing a video to upload to Media Services

When the user selects a video, using either a pointing device or a touch gesture, the **Open** button should be selected to begin the upload process. The first step in the upload process is to enter basic video details, as shown in the following figure.



The video details that must be provided prior to upload

The user must enter a title, description (optional), and select the resolution they'd like the video to be encoded to. When the **Create** button is selected the upload process begins, as shown in the following figure.



A video uploading for processing by Media Services

The user can choose to cancel the upload at any point, or allow the upload to proceed until it completes. A failure message is displayed to the user if the upload process fails.

When a video has been successfully uploaded the video details are saved to the CMS, and the encoding process begins. The Contoso has opted to always encode videos to adaptive bitrate MP4s, and then uses dynamic packaging to convert the adaptive bitrate MP4s to smooth streaming, HLS, or MPEG DASH, on demand. For more information about dynamic packaging see "Dynamic packaging."

Once a video has been successfully encoded it can be selected for viewing from the initial page of the application.

For information on how the uploading video use case is implemented, see "Chapter 3–Uploading Video."

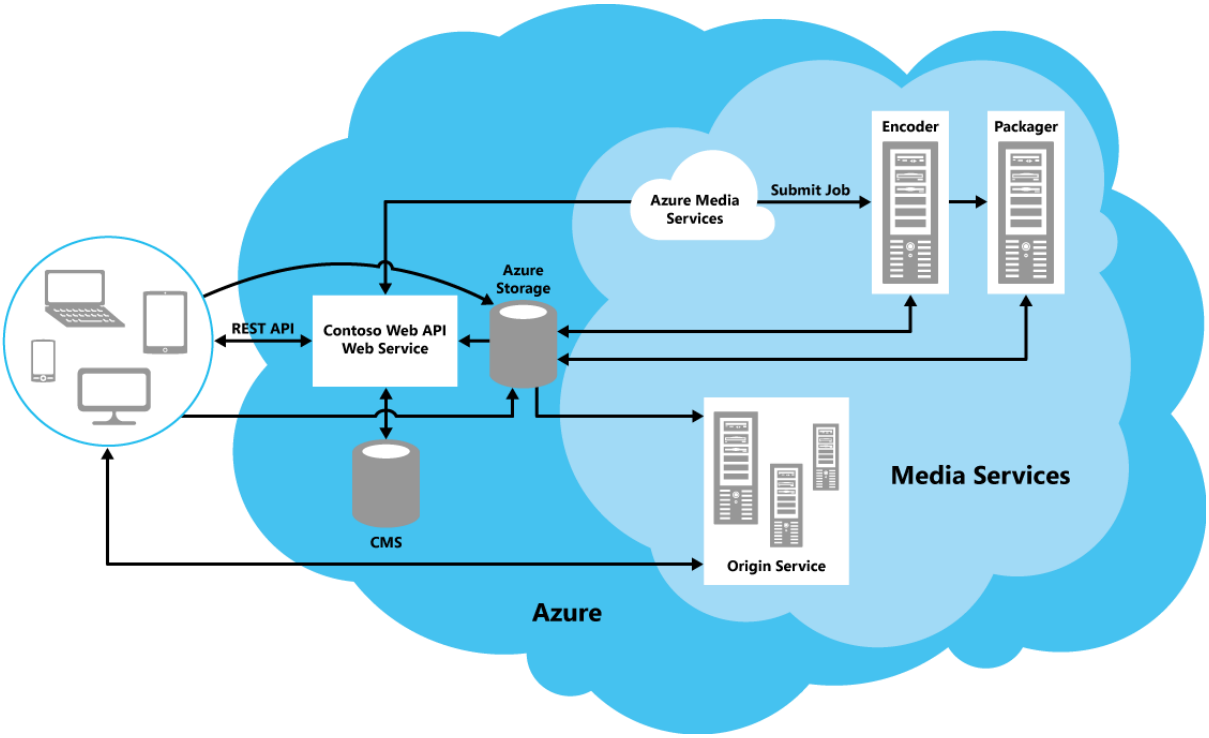
Understanding the Contoso Azure Media Services application architecture

The developers at Contoso are knowledgeable about various Microsoft products and technologies, including the .NET Framework, the Entity Framework, Azure, and the Windows Runtime, so they decided to implement the solution using these technologies.

Building and hosting a video-on-demand service is a major undertaking and can require a significant investment in hardware, management, and other infrastructure resources. Connectivity and security are also major concerns because users require timely and responsive access, and at the same time the system must maintain the integrity of the data and the privacy of users' information. To support a potentially large number of concurrent users against an ever-expanding collection of videos, the staff at Contoso chose to implement the video-on-demand service by using Azure Media Services. Media Services allows you to build scalable, cost effective, end-to-end media distribution solutions

that can upload, encode, package, and stream media to a variety of devices and platforms. In addition, the Azure environment provides the necessary scalability, reliability, security, and performance necessary for supporting a large number of concurrent, distributed users.

The following figure shows a high-level overview of the solution.



A high-level overview of the Contoso Media Services solution

Client applications communicate with the video-on-demand service through a REST web interface. This interface allows applications to retrieve, upload, and publish videos. When a video is uploaded for processing by Media Services it is stored in Azure Storage, with the video details being stored in the CMS. It's then encoded to a set of adaptive bitrate MP4s, which can be converted by dynamic packaging to smooth streaming, HLS, or MPEG-DASH, on demand. For more information about dynamic packaging see "Dynamic packaging."

When a video is consumed by applications, its URL is retrieved from the CMS and returned to the application. The application then requests the URL content from the Media Services Origin Service, which processes the outbound stream from storage to client app. For more information about the Origin Service, see "Origin Service."

The solution comprises three main components:

- The user facing client applications, implemented for the Windows Runtime, Windows Phone, Web, iOS, and Android. These applications allow users to browse videos, and control the playback of videos. In addition the applications allow users to capture and upload new videos for encoding by Media Services, which can then be consumed from the CMS once encoding has completed.

The Contoso Windows Store application is used to demonstrate the Media Services functionality, which is consumed through a REST interface. However, several other apps are also provided for different devices and platforms, which all formulate the appropriate REST requests and consume REST responses.

- The business logic, implemented as a web service. The Contoso web service exposes the data and operations that it supports through a REST (Representational State Transfer) interface. Separating the business logic in this way decouples it from the client applications, minimizing the impact that any changes to the implementation of the applications will have on this business logic.


Using REST enables you to invoke operations and consume the responses by using any web-enabled system that can formulate REST queries, providing great flexibility in building similar apps for different devices and platforms.

- Data storage, provided by an Azure SQL database, and by Azure Storage. The CMS, which stores details of videos and encoding jobs, is implemented as a Azure SQL database. However, uploaded videos and encoded videos output by Media Services are stored in Azure Storage.


Using Media Services minimizes the hardware and support investment that needs to be made when providing a video-on-demand service. You can monitor the volume of traffic to the web service, and if necessary simply scale the solution to use additional resources.

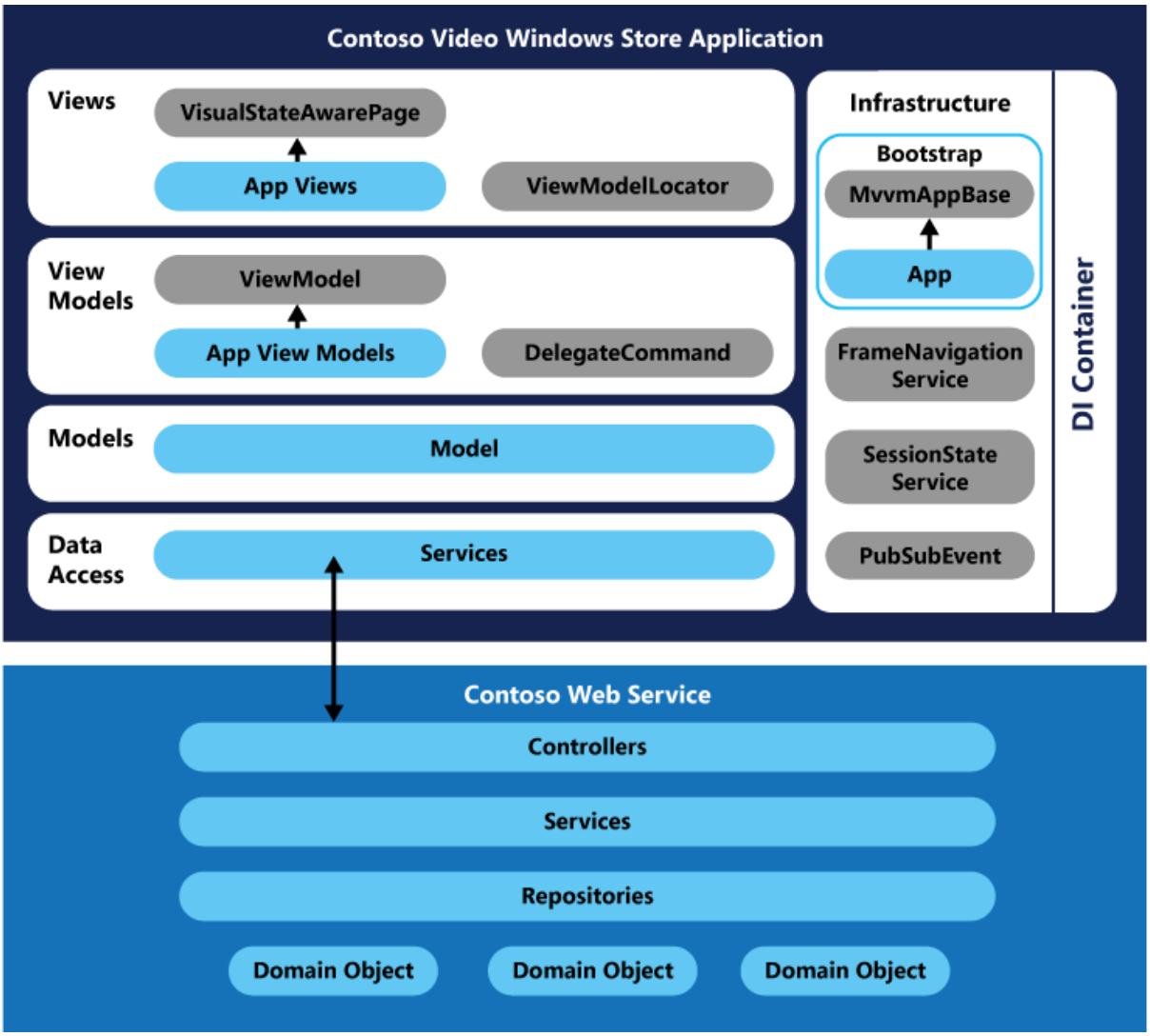
For more information about building and deploying applications to the cloud by using Azure, see the patterns & practices guide "Developing Multi-tenant Applications for the Cloud, 3rd Edition" available from MSDN.

Understanding the Windows Store application architecture

Developers of Windows Store applications face several challenges. Application requirements can change over time. New business opportunities and challenges may present themselves. Ongoing customer feedback during development may significantly affect the requirements of the application. Therefore it's important to build an application that has a flexible architecture and can be easily modified or extended over time.

The Contoso developers used a modified version of Prism for the Windows Runtime to accelerate the development of their Windows Store application. Prism includes components that provide support for MVVM and the core services required in Windows Store applications. This allowed the Contoso developers to focus on developing the user experiences for their video application. For more information about Prism for the Windows Runtime, see "Developing a Windows Store business app using C#, XAML, and Prism for the Windows Runtime."

The following figure shows the architecture of the Contoso Windows Store video application and web service in more detail. Grey items are provided by the modified version of Prism for the Windows Runtime, with blue items being created by the Contoso development team. For clarity, the diagram does not show all the class names.



The architecture of the Contoso Windows Store video application and web service

The advantage of this architecture is that it helps to produce flexible, maintainable, and testable code, by addressing common Windows Store application development scenarios, and by separating the concerns of presentation, presentation logic, and entities through support for MVVM.

For information on how to bootstrap a Windows Store application that uses Prism for the Windows Runtime, see "Bootstrapping an MVVM Windows Store app Quickstart using C#, XAML, and Prism."

Using a dependency injection container

The Contoso developers use a dependency injection container to manage the instantiation of many of the classes.


Dependency injection enables decoupling of concrete types from the code that depends on these types. It uses a container that holds a list of registrations and mappings between interfaces and abstract types and the concrete types that implement or extend these types.

The Contoso Windows Store video application uses the Unity dependency injection container to manage the instantiation of the view model and service classes in the application. The **App** class instantiates the **UnityContainer** object and is the only class that holds a reference to this object. Types are then registered in the **OnInitialize** method in the **App** class.


You should consider carefully which objects you should cache and which you should instantiate on demand. Caching objects improves the application's performance at the expense of memory utilization.

For more information about using Unity, see "Unity Container."

Understanding the Visual Studio solution

The Visual Studio solution organizes the source code and other resources into projects. All of the projects use Visual Studio solution folders to organize the source code and other resources into categories. The following table outlines the projects that make up the Contoso web service and Contoso video applications.

Project	Description
Contoso.Infrastructure.ReusableComponents	This project contains classes and interfaces from Prism for the Windows Runtime, which are used by the Contoso.Infrastructure.WindowsPhone and Contoso.Infrastructure.WindowsStore projects.
Contoso.Infrastructure.WIndowsPhone	This project contains Windows Phone specific classes and interfaces from Prism for the Windows Runtime.
Contoso.Infrastructure.WindowsStore	This project contains Windows Store specific classes and interfaces from Prism for the Windows Runtime.
Contoso.Api.Test	This project contains unit tests for the Contoso.Api project.
Contoso.Services.Test	This project contains unit tests for the Contoso.Domain.Services.Imply project.
Contoso.Test.Shared	This project contains unit test helper methods used by the Contoso.Api.Test and Contoso.Services.Test projects.
Contoso.UILogic.Tests	This project contains unit tests for the Contoso.UILogic project.
Contoso.UILogic.Tests.Mocks	This project contains mocks used by the Contoso.UILogic.Tests project.

Contoso.WindowsStore.Tests	This project contains unit tests for the Contoso.WindowsStore project.
Contoso.Api	This project contains the code for the Contoso web service.
Contoso.Azure	This project defines the roles for deploying the application to Azure, along with the service configuration. The Contoso.Api and Contoso.WebClient projects are deployed as web roles, with the Contoso.EncodingWorker being deployed as a worker role.
Contoso.Azure.Shared	This project defines the **CloudConfiguration** and **TraceHelper** classes. The **CloudConfiguration** class is used to retrieve Media Services account credentials from configuration.
Contoso.Azure.Stores	This project defines classes used to manage the video encoding process through Azure Storage Queues.
Contoso.Domain	This project defines the domain entity objects that remove the dependencies that controller classes in the Contoso.Api project might otherwise have on the way that data is stored.
Contoso.Domain.Services	This project defines the interfaces for the domain services that are implemented by the Contoso.Domain.Services.Impl project.
Contoso.Domain.Services.Impl	This project contains the classes that implement the domain services that perform encoding and interact with the repository classes.
Contoso.EncodingWorker	This project contains the worker role code that interacts with Azure Storage Queues in order to manage the encoding process.
Contoso.Repositories	This project defines the interfaces for the repository classes that retrieve and modify data in the Contoso Content Management System (CMS).
Contoso.Repositories.Impl.Sql	This project contains the repository classes and the data types that the repository classes use to retrieve and modify data from the Contoso CMS.
Contoso.Shared	This project defines the **ContosoEventSource** class which creates events for Event Tracing for Windows (ETW). This class is used by the Contoso.Api and Contoso.Services.Test projects.
Contoso.UILogic	The project contains the shared business logic for the Contoso video Windows Store and Windows Phone implementations.
Contoso.WebClient	This project contains the web client implementation of the Contoso video application.

Contoso.WindowsPhone	This project contains the Windows Phone specific code for the Windows Phone client implementation of the Contoso video application.
Contoso.WindowsStore	This project contains the Windows Runtime specific code for the Windows Store client implementation of the Contoso video application.

For information about the structure of the projects that implement the Contoso video applications see Appendix C – Understanding the Contoso Video Applications.

Developing the content management system

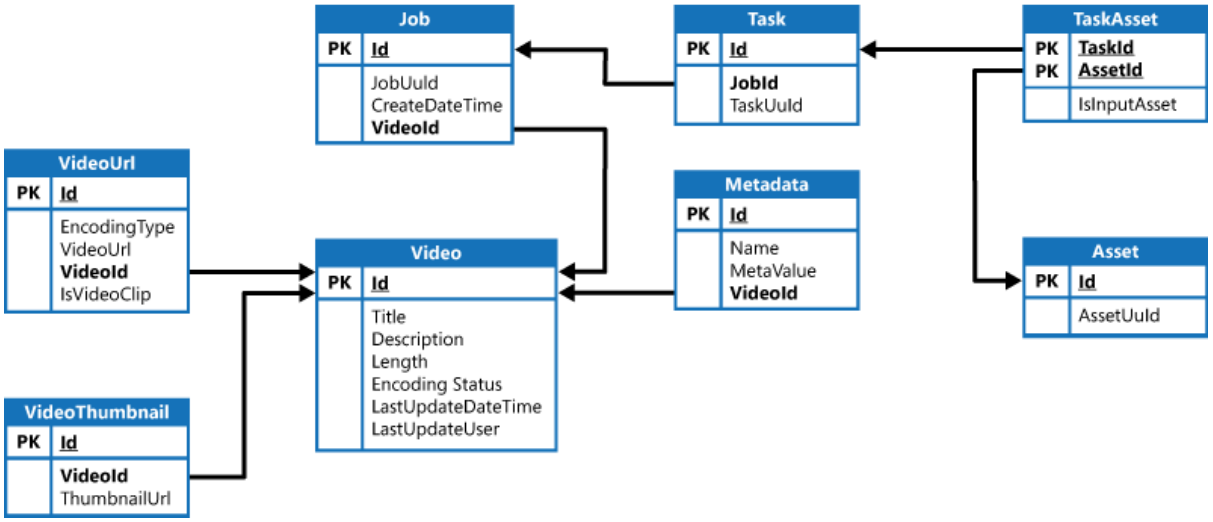
A video CMS enables you to upload, store, process, and publish media. They generally store files in a database and allow for metadata tagging and searching.

Media Services is not a CMS but it does enable you to implement a video processing workflow. You can upload and store your content in Azure Storage, encode and package media into a variety of popular formats, and stream your videos online.

The requirements for the Contoso CMS were as follows:

- Ability to store details of videos that can be consumed by client applications.
- Ability to store video metadata.
- Ability to store thumbnail images that represent each video.
- Ability to store details of encoding jobs.

The developers at Contoso decided to store this information in a series of tables in third normal form. This structure helps to reduce the probability of duplicate information, while optimizing many of the common queries performed by the client applications. The following figure shows the table structure of the database.



The table structure of the CMS

Cloud-based databases are increasingly popular because they remove the need for an organization to maintain its own infrastructure for hosting a database. They offer elasticity that enables a system to quickly and easily scale as the number of requests and hence the volume of work increases. An additional advantage is that Azure SQL databases maintain multiple copies of the database, running on different server. Therefore, if the primary server fails, all requests are transparently switched to another server. Therefore, the developers at Contoso chose to implement the database as a Azure SQL database.

A complete list of features available in a Azure SQL database is available at "General Guidelines and Limitations (Windows Azure SQL Database)" on MSDN.

For best practices about designing and developing a relational database, see the patterns & practices guide "Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence" available from MSDN.

Accessing the content management system

A relational database stores data as a collection of tables. However, the Contoso video applications process data in the form of entity objects. The data for an entity object might be constructed from one or more rows in one or more tables. In the Contoso video applications, the business logic that manipulates objects is independent of the format of the data for that object in the database. This offers the advantage that you can modify and optimize the database structure without affecting the code in the applications, and vice versa.

This approach requires the use of an object-relational mapping layer (ORM). The purpose of an ORM is to act as an abstraction of the underlying database. The Contoso video applications create and use objects, and the ORM exposes methods that can take objects and use them to generate relational create, retrieve, update, and delete (CRUD) operations, which it then sends to the database server. Tabular data is then returned from the database and converted into a set of objects by the ORM.

The Contoso developers chose to use the Microsoft Entity Framework as the ORM, and also used the Fluent API to decouple the classes in the object model from the Entity Framework. In the Entity Framework the database is interacted with through a *context* object. The context object provides the connection to the database and implements the logic performing CRUD operations on the data in the database. The context object also performs the mapping between the object model of your application and the tables defined in the database.

The Contoso video apps send REST requests to the Contoso web service that validates these requests and converts them into the corresponding CRUD operations against the CMS. All incoming REST requests are routed to a controller based on the URL that the client application specifies. The controllers indirectly use the Microsoft Entity Framework to connect to the CMS database and retrieve, create, update, and delete data. The developers implemented the Repository pattern to minimize dependencies that the controllers have on the Entity Framework.

The purpose of the Repository pattern is to act as an intermediary between the object-relational mapping layer (implemented by the Entity Framework) and the data mapping layer that provides the objects for the controller classes. In the Contoso web service, each repository class provides a set of APIs that enable a service class (invoked by a controller class) to retrieve a database-neutral object from the repository, modify it, and store it back in the repository. The repository class has the responsibility for converting all the requests made by a service class into commands that it can pass to the Entity Framework. As well as removing any database-specific dependencies from the business logic of the controller and service classes, this approach provides flexibility. If the developers decided to switch to a different data store, they can provide an alternative implementation of the repository classes that expose the same APIs to the service classes.


Avoid building dependencies on a specific data access technology into the business logic of an application. Using the Repository pattern can help reduce the chances of this happening.

For more information about the Contoso web service and its use of the Repository pattern, see Appendix A – The Contoso Web Service.

Summary

This chapter has introduced the video application and web service built by Contoso. The Windows Store video application is built using XAML and C#, and consumes a web service that provides a REST interface to the CMS and Media Services. Media Services is used to encode and package video into the required formats for consumption across a variety of platforms and devices.

This chapter also discussed why Contoso chose to store the database in the cloud by using a Azure SQL database. It summarized how the client applications connect to the database by using the Entity Framework, and how the Repository pattern is used to abstract the details of the Entity Framework from the business logic of the system.

In the following chapters you will see how the developers at Contoso designed and implemented the end-to-end solution that uploads, encodes, delivers, and consumes media.

More information

- You can find the General Guidelines and Limitations (Windows Azure SQL Database) page on MSDN.
- The patterns & practices guide "Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence" is available from MSDN.
- You can find information about the Entity Framework in the Data Developer Center, available on MSDN.
- The Repository pattern is described on MSDN.
- The patterns & practices guide *"Developing Multi-tenant Applications for the Cloud, 3rd Edition"* is available on MSDN.
- For information about Prism for the Windows Runtime, see "Developing a Windows Store business app using C#, XAML, and Prism for the Windows Runtime."
- For information about using Unity, see "Unity Container."
- For information on how to bootstrap a Windows Store application that uses Prism for the Windows Runtime, see "Bootstrapping an MVVM Windows Store app Quickstart using C#, XAML, and Prism."

3 - Uploading Video into Microsoft Azure Media Services

In order to manage, encode, and stream your videos, you must first upload your content into Microsoft Azure Media Services. Once uploaded, your content is stored in the cloud for further processing and streaming.

When deciding upon the media content to upload and store as an asset there are restrictions that apply. Each asset should only contain a unique instance of media content, such as a single TV episode or ad. Therefore, each asset should not contain multiple edits of a file, in order to reduce difficulties submitting encoding jobs, and streaming and securing the delivery of the asset later in the workflow. For example, an incorrect usage of an asset would be storing both the trailer and the feature-length movie within a single asset: you may want the trailer to have wide viewership, but restrict viewing of the movie.

This chapter describes how the Contoso developers incorporated Media Services' uploading functionality into their web service and Windows Store client application. It summarizes the decisions that they made in order to support their business requirements, and how they designed the code that performs the upload process.

For more information about the Contoso web service see "Appendix A – The Contoso Web Service."

> Chapter 2, "The Azure Media Services Video-on-Demand Scenario," describes the primary business functions of the video application.

Uploading content

Media Services is an OData-based REST service that exposes objects as entities that can be queried in the same way as other OData entities. Media Services is built on OData v3, which means that you can submit HTTP request bodies in atom+pub or verbose JSON, and receive your responses in the same formats. For more information about ingesting assets using the REST API see "Ingesting Assets with the Media Services REST API" and "Ingesting Assets in Bulk with the REST API."

Uploading content with the Media Services SDK for .NET

The Media Services SDK for .NET is a wrapper around the REST APIs. The SDK provides a simple way to accomplish the tasks that are exposed by the REST API.

Uploading content with the Azure Management Portal

Video, audio, and images can be uploaded to a Media Services account through the Azure Management Portal. However, this approach limits uploads to the formats that are supported by the Azure Media Encoder. For more information see "Supported Codecs and File Types for Microsoft Azure Media Services."

There are several limitations to consider when uploading content through the Management Portal:

- You can't upload multiple files in a single upload.

- You can't upload a file larger than 200MB. However, there's no file size limit if you're uploading from an existing storage account.
- You can't upload all the file formats that are supported by Media Services. You can only upload files with the following extensions: .asf, .avi, .m2tf, .m2v, .mp4, .mpeg, .mpg, .mts, .ts, .wmv, .3gp, .3g2, .3gp2, .mod, .dv, .vob, .ismv, .m4a.

If you need to upload content into Media Services at high speed you can take advantage of high speed ingest technology offered by third-party providers. For more information see "Uploading Large Sets of Files with High Speed."

Managing assets across multiple storage accounts within Azure Media Services

Media Services accounts can be associated with one or more storage accounts, with each storage account being limited to 200TB. Attaching multiple storage accounts to a Media Services account provides the following benefits:

- Load balancing assets across multiple storage accounts.
- Scaling Media Services for large amounts of storage and processing.
- Isolating file storage from streaming or DRM protected file storage.

For more information see "Managing Media Services Assets across Multiple Storage Accounts."

Ingesting content with the Media Services SDK for .NET

To get content into Media Services you must first create an asset and add files to it, and then upload the asset. This process is known as ingesting content.

The content object in Media Services is an **IAsset**, which is a collection of metadata about a set of media files. Each **IAsset** contains one or more **IAssetFile** objects. There are two main approaches to ingesting assets into Media Services:

- Create an **Asset**, upload your content to Media Services, and then generate **AssetFiles** and associate them with the **Asset**.
- Bulk ingest a set of files by preparing a manifest that describes the asset and its associated files. Then use the upload method of your choice to upload the associated files to the manifest's blob container. Once a file is uploaded to the blob container Media Services completes the asset creation based on the configuration of the asset in the manifest.

The first approach is the preferred approach when working with a small set of media files, and is the approach adopted by the Contoso development team. For more information about ingesting assets in bulk see "Ingesting Assets in Bulk with the Media Services SDK for .NET."


To create an asset you must first have a reference to the Media Services server context.

Supported input formats for Azure Media Services

Various video, audio, and image file types can be uploaded to a Media Services account, with there being no restriction on the types or formats of files that you can upload using the Media Services SDK. However, the Azure Management portal restricts uploads to the formats that are supported by the Azure Media Encoder.

Content encoded with the following video codecs may be imported into Media Services for processing by Azure Media Encoder:

- H.264 (Baseline, Main, and High Profiles)
- MPEG-1
- MPEG-2 (Simple and Main Profile)
- MPEG-4 v2 (Simple Visual Profile and Advanced Simple Profile)
- VC-1 (Simple, Main, and Advanced Profiles)
- Windows Media Video (Simple, Main, and Advanced Profiles)
- DV (DVC, DVHD, DVSD, DVSL)

The following video file formats are supported for import:

File format	File extension
3GPP, 3GPP2	.3gp, .3g2, .3gp2
Advanced Systems Format (ASF)	.asf
Advanced Video Coding High Definition (AVCHD)	.mts, .m2tf
Audio-Video Interleaved (AVI)	.avi
Digital camcorder MPEG-2 (MOD)	.mod
Digital video (DV) camera file	.dv
DVD transport stream (TS) file	.ts
DVD video object (VOB) file	.vob
Expression Encoder Screen Capture Codec file	.xesc
MP4	.mp4
MPEG-1 System Stream	.mpeg, .mpg
MPEG-2 video file	.m2v
Smooth Streaming File Format (PIFF 1.3)	.ismv
Windows Media Video (WMV)	.wmv

Content encoded with the following audio codecs may be imported into Media Services for processing by Azure Media Encoder:

- AC-3 (Dolby Digital audio)
- AAC (AAC-LC, HE-AAC v1 with AAC-LC core, and HE-AAC v2 with AAC-LC core)
- MP3
- Windows Media Audio (Standard, Professional, and Lossless)

The following audio file formats are supported for import:

File format	File extension
AC-3 (Dolby digital) audio	.ac3
Audio Interchange File Format (AIFF)	.aiff
Broadcast Wave Format	.bwf
MP3 (MPEG-1 Audio Layer 3)	.mp3
MP4 audio	.m4a
MPEG-4 audio book	.m4b
WAVE file	.wav
Windows Media Audio	.wma

The following image file formats are supported for import:

File format	File extensions
Bitmap	.bmp
GIF, Animated GIF	.gif
JPEG	.jpeg, .jpg
PNG	.png
TIFF	.tif
WPF Canvas XAML	.xaml

For more information on the codecs and file container formats that are supported by Azure Media Encoder see "Supported input formats" and "Introduction to encoding."

Securing media for upload into Azure Media Services

Media Services allows you to secure your media from the time it leaves your computer. All media files in Media Services are associated with an **Asset** object. When creating an **Asset** for your media by calling **Asset.Create**, you must specify an encryption option as a parameter by using one of the **AssetCreationOptions** enumeration values. Each file added to the **Asset** will then use the asset creation options specified when the asset is created.

The **AssetCreationOptions** enumeration specifies four values:

- AssetCreationOptions.None
- AssetCreationOptions.StorageEncrypted
- AssetCreationOptions.CommonEncryptionProtected
- AssetCreationOptions.EnvelopeEncryptionProtected

Media can be uploaded without any protection by specifying **AssetCreationOptions.None**. This is not recommended as the content will not be protected during the upload, or in storage. However, media could be uploaded over an SSL connection to protect the transmission process, prior to it being stored unprotected in Azure Storage.

If you have unencrypted media that you wish to encrypt prior to upload you should specify **AssetCreationOptions.StorageEncrypted** when creating the asset. This encrypts media locally prior to uploading it to Azure storage where it will be stored encrypted.

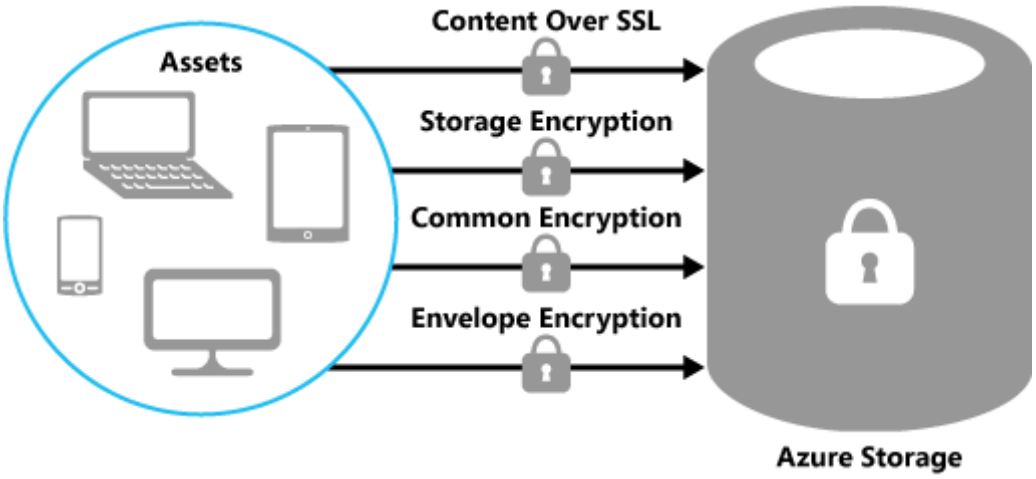
> Assets protected with storage encryption will be automatically unencrypted and placed in an encrypted file system prior to encoding. In addition, any storage encrypted content must be decrypted before being streamed.

If you have pre-encoded Smooth Streaming content that is already protected with PlayReady Digital Rights Management (DRM) you should specify **AssetCreationOptions.CommonEncryptionProtected** when creating the asset. This enumeration value specifies that an assets files are protected using a common encryption method. Therefore your content is already protected in transit and in storage.

If you have pre-encoded HLS content with AES encryption you should specify **AssetCreationOptions.EnvelopeEncryptionProtected** when creating the asset. This enumeration value specifies that an assets files are protected using an envelope encryption method, such as AES-CBC. Therefore your content is already protected in transit and in storage.

>  Media Services only provides on-disk storage encryption, not over the wire encryption like a Digital Rights Management (DRM) solution.

The following figure summarizes how media can be protected during the upload process.



The options for protecting media when at rest and in transit

The Contoso video application does not secure the web service with Secure Sockets Layer (SSL), so a malicious client could impersonate the application and send malicious data. In your own application you should protect any sensitive data that you need to transfer between the application and a web service by using SSL.

For more information about securing your media at rest and in transit, see "Securing Your Media."

Connecting to Azure Media Services

Before you can start programming against Media Services you need to create a Media Services account in a new or existing Azure subscription. For more information see "How to Create a Media Services Account."

At the end of the Media Services account setup process you will have obtained the following connection values:

- Media Services account name.
- Media Services account key.

These values are used to make programmatic connections to Media Services. You must then setup a Visual Studio project for development with the Media Services SDK for .NET. For more information see "Setup for Development on the Media Services SDK for .NET."

Media Services controls access to its services through an OAuth protocol that requires an Access Control Service (ACS) token that is received from an authorization server.

To start programming against Media Services you must create a **CloudMediaContext** instance that represents the server context. The **CloudMediaContext** includes references to important collections including jobs, assets, files, access policies, and locators. One of the **CloudMediaContext** constructor overloads takes a **MediaServicesCredentials** object as a parameter, and this enables the reuse of

ACS tokens between multiple contexts. The following code example shows how the **MediaServicesCredentials** object is created.

> **Note:** You can choose not to deal with ACS tokens, and leave the Media Service SDK to manage them for you. However, this can lead to unnecessary token requests which can create performance issues both on the client and server.

C#

```
private static readonly Lazy<MediaServicesCredentials> Credentials =
    new Lazy<MediaServicesCredentials>(() =>
    {
        var credentials = new MediaServicesCredentials(
            CloudConfiguration.GetConfigurationSetting("ContosoAccountName"),
            CloudConfiguration.GetConfigurationSetting("ContosoAccountKey"));

        credentials.RefreshToken();
        return credentials;
    });
```

The **Credentials** object is cached in memory as a static class variable that uses lazy initialization to defer the creation of the object until it is first used. This object contains an ACS token that can be reused if hasn't expired. If it has expired it will automatically be refreshed by the Media Services SDK using the credentials given to the **MediaServicesCredentials** constructor. The cached object can then be passed to the **CloudMediaContext** constructor in the constructor of the **EncodingService** class.

C#

```
public EncodingService(IVideoRepository videoRepository,
    IJobRepository jobRepository)
{
   ...
   this.context = new CloudMediaContext(EncodingService.Credentials.Value);
}
```

When the **CloudMediaContext** instance is created the **Credentials** object will be created. Using lazy initialization to do this reduces the likelihood of the **MediaServicesCredentials** object having to refresh its ACS token due to expiration. For more information about lazy initialization see "Lazy Initialization."

If you don't cache your Media Services credentials in a multi-tenant application, performance issues will occur as a result of thread contention issues.

For better scalability and performance, the **EncodingService** constructor uses the constructor overload of the **CloudMediaContext** class that takes a **MediaCredentials** object.

The Contoso developers store connection values, including the account name and password, in configuration. The values in the <ConfigurationSettings> element are the required values obtained during the Media Services account setup process.

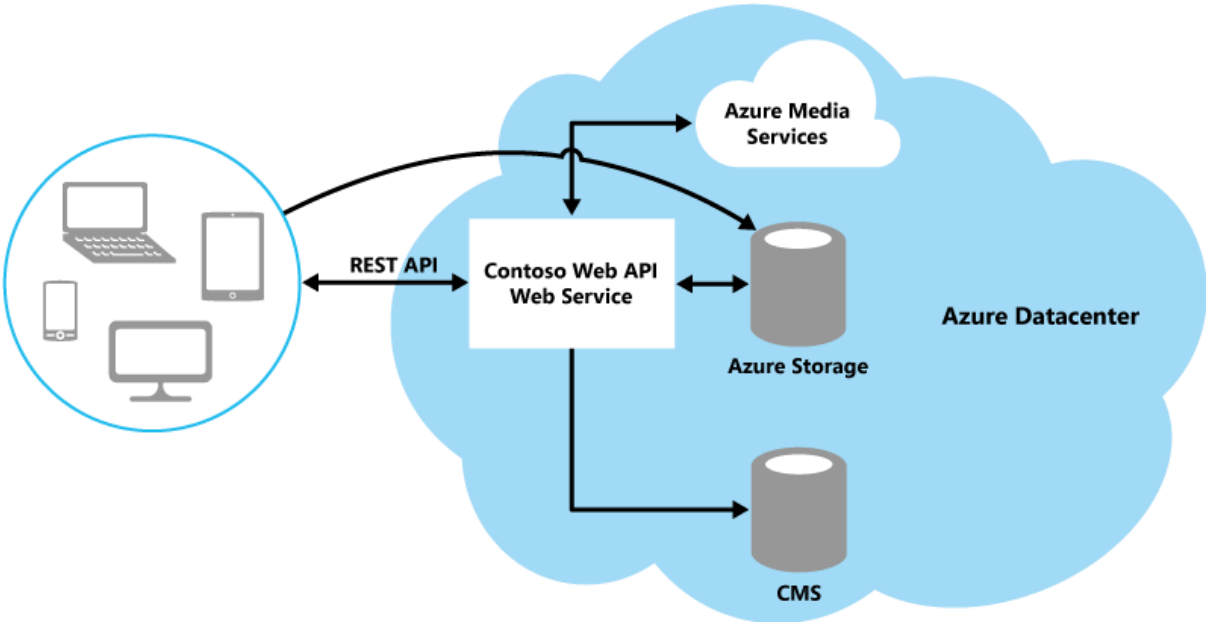
XML

```
<ConfigurationSettings>
     ...
     <Setting name="ContosoAccountName" value="Media_Services_Account_Name" />
     <Setting name="ContosoAccountKey" value="Media_Services_Account_Key" />
</ConfigurationSettings>
```

Configuration files can be encrypted by using the Windows Encrypting File System (EFS). Or you can create a custom solution for encrypting selected portions of a configuration file by using protected configuration. For more information see "Encrypting Configuration Information Using Protected Configuration."

Upload process in the Contoso Azure Media Services applications

The following figure shows a high-level overview of the Contoso media upload process.



A high-level overview of the Contoso media upload process

Client apps communicate with the Contoso web service through a REST web interface, which allows them to upload media assets. When a new video is uploaded a new asset is created by Media Services, and the asset is uploaded to Azure Storage before the assets details are published to the Content Management System (CMS).

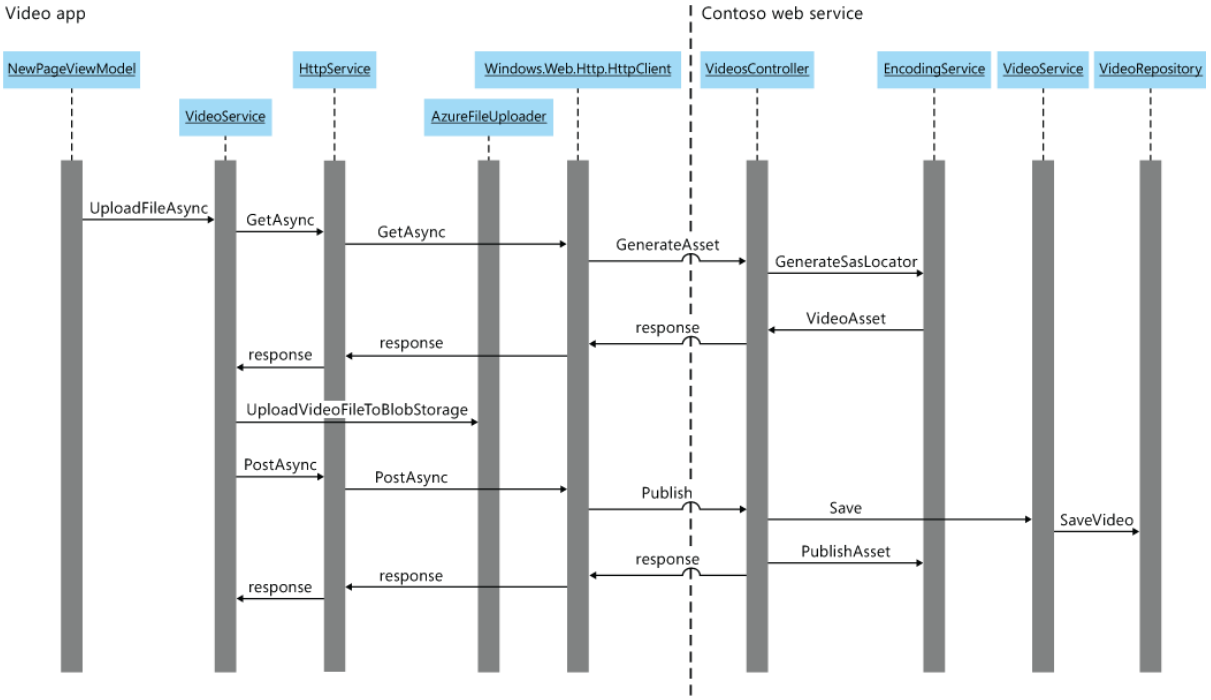
This process can be decomposed into the following steps for uploading content into Media Services:

1. Create a new empty **Asset**.
2. Create an **AccessPolicy** instance that defines the permissions and duration of access to the asset.

3. Create a **Locator** instance that will provide access to the asset.
4. Upload the file that's associated with the **Asset** into blob storage.
5. Publish the **Asset**.
    - Save the **Asset** details to the CMS.
    - Generate an **AssetFile** for the **Asset**.
    - Add the **Asset** to the encoding pipeline.

The Contoso web service does not contain an authentication mechanism. In your own application you should implement a secure authentication mechanism so that it's possible to link videos to the users who uploaded them.

The following figure shows the interaction of the classes in the Contoso Windows Store Video application that implement uploading a video for processing by Media Services.



The interaction of the classes that upload a video to Media Services

For information on how the upload process works in the Contoso video web application, see "Appendix C – Understanding the Contoso Video Applications."

The upload process is managed by the **VideoService** class in the Contoso.UILogic project. In the **OnInitialize** method in the **App** class, the **VideoService** class is registered as a type mapping against the **IVideoService** interface with the Unity dependency injection container. Then when a view model

class such as the **NewVideoPageViewModel** class accepts an **IVideoService** type, the Unity container will resolve the type and return an instance of the **VideoService** class

The upload process is invoked when a user selects the **Create** button on the **NewVideoPage**. Through a binding this executes the **CreateVideo** method in the **NewVideoPageViewModel** class. This creates a new **Video** object and initializes the object with data from the file chosen for upload. Then, the upload process begins by calling the **UploadFileAsync** method in the **VideoService** class. A **CancellationToken** is passed to the **UploadFileAsync** method so that the upload process can be cancelled by the user if required.

C#

```
private async void CreateVideo()
{
   ...
   await this.videoService.UploadFileAsync(this.videoFile, video,
       this.cancellationTokenSource.Token);
   ...
}
```

The **UploadFileAsync** method manages the upload process in the client by invoking methods of the **HttpService** and **AzureFileUploader** classes.

C#

```
public async Task UploadFileAsync(VideoFile file, Video video, CancellationToken
cancellationToken)
{
    var requestUri = new Uri(string.Format("{0}/{1}/?filename={2}",
        this.videosBaseUrl, "generateasset", file.Name));

    var responseContent =
        await this.httpService.GetAsync(requestUri, cancellationToken);

    var videoUploadInfo =
        JsonConvert.DeserializeObject<VideoUpload>(responseContent);

    await this.azureFileUploader.UploadVideoFileToBlobStorage(file,
        videoUploadInfo.SasLocator, cancellationToken);

    video.AssetId = videoUploadInfo.AssetId;
    var videoUpload = JsonConvert.SerializeObject(video);

    var uploadVideoUri = new Uri(string.Format("{0}/{1}", this.videosBaseUrl,
        "publish"));
    await this.httpService.PostAsync(uploadVideoUri, videoUpload,
        cancellationToken);
}
```

This method creates a Uri that specifies that the **GenerateAsset** method will be called on the web service, with the filename of the file to be uploaded being passed as a parameter. The **HttpService** class, which implements the **IHttpService** interface, is used to make the call to the web service.

C#

```
public async Task<string> GetAsync(Uri requestUri)
{
    return await this.GetAsync(requestUri, CancellationToken.None);
}

public async Task<string> GetAsync(Uri requestUri, CancellationToken
cancellationToken)
{
    using (var httpClient = new HttpClient())
    {
        var response =
            await httpClient.GetAsync(requestUri).AsTask(cancellationToken);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

This method asynchronously retrieves data from the web service by using the **HttpClient** class to send HTTP requests and receive HTTP responses from a URI. The call to **HttpClient.GetAsync** sends a GET request to the specified URI as an asynchronous operation, and returns a **Task** of type **HttpResponseMessage** that represents the asynchronous operation. The **Task** is cancellable, and will complete after the content from the response is read. For more info about the **HttpClient** class see "Connecting to an HTTP server using Windows.Web.Http.HttpClient."

When the **UploadFileAsync** method calls **HttpService.GetAsync**, this calls the **GenerateAsset** method in the **VideosController** class in the Contoso.Api project.

C#

```
public async Task<HttpResponseMessage> GenerateAsset(string filename)
{
    ...
    var videoAsset = await encodingService.GenerateSasLocator(filename);
    var result = new VideoAssetDTO();

    Mapper.Map(videoAsset, result);

    return Request.CreateResponse(HttpStatusCode.Created, result);
    ...
}
```

This method calls the asynchronous **GenerateSasLocator** method in the **EncodingService** class. The **EncodingService** class is registered as a type mapping against the **IEncodingService** interface with the Unity dependency injection container. When the **VideosController** class accepts an **IEncodingService** type, the Unity container will resolve the type and return an instance of the **EncodingService** class.

When a **VideoAsset** object is returned from the **GenerateSasLocator** method a **VideoAssetDTO** object is created, with the returned **VideoAsset** object being mapped onto the **VideoAssetDTO**, which is then returned in an **HttpResponseMessage** to the **HttpService.GetAsync** method.

The **GenerateSasLocator** method uses Media Services types to return a new **VideoAsset** that contains a new asset id to represent the asset being uploaded, and a shared access signature locator to access the asset.

C#

```
public async Task<VideoAsset> GenerateSasLocator(string filename)
{
    var duration = int.Parse(
        CloudConfiguration.GetConfigurationSetting("SasLocatorTimeout"));

    IAsset asset = await this.context.Assets.CreateAsync(
        "NewAsset_" + Guid.NewGuid() + "_" + filename, AssetCreationOptions.None,
        CancellationToken.None).ConfigureAwait(false);

    IAccessPolicy writePolicy = await this.context.AccessPolicies.CreateAsync(
        "writePolicy", TimeSpan.FromMinutes(duration), AccessPermissions.Write)
        .ConfigureAwait(false);

    ILocator destinationLocator = await this.context.Locators.CreateLocatorAsync(
        LocatorType.Sas, asset, writePolicy).ConfigureAwait(false);

    var blobUri = new UriBuilder(destinationLocator.Path);
    blobUri.Path += "/" + filename;

    // return the new VideoAsset
    return new VideoAsset()
        { SasLocator = blobUri.Uri.AbsoluteUri, AssetId = asset.Id };
}
```

The method creates a new asset using **AssetCreationOptions.None** that specifies that no encryption is used when the asset is in transit or at rest. An access policy named **writePolicy** is then created that specifies that the asset can be written to for 30 minutes. A shared access signature locator is then created, using the access policy. The locator returns an entry point that can be used to access the files contained in the asset. Finally, a Uri is created to which the video file will be uploaded to in blob storage, before the **VideoAsset** object is created and returned.

The maximum number of assets allowed in a Media Services account is 1,000,000.

Back in the **UploadFileAsync** method in the **VideoService** class the **UploadFileToBlobStorage** method of the **AzureFileUploader** class is used to upload the file to blob storage using the shared access storage locator returned by the **GenerateSasLocator** method.

C#

```
public async Task UploadVideoFileToBlobStorage(VideoFile file, string sasLocator,
CancellationToken cancellationToken)
{
    var blobUri = new Uri(sasLocator);
    var sasCredentials = new StorageCredentials(blobUri.Query);
```

```
    var blob = new CloudBlockBlob(new
Uri(blobUri.GetComponents(UriComponents.SchemeAndServer | UriComponents.Path,
UriFormat.UriEscaped)), sasCredentials);
    StorageFile storageFile = null;

    if (string.IsNullOrEmpty(file.FutureAccessToken))
    {
        storageFile = await
StorageFile.GetFileFromPathAsync(file.Path).AsTask(cancellationToken);
    }
    else
    {
        storageFile = await
StorageApplicationPermissions.FutureAccessList.GetFileAsync(file.FutureAccessToken
).AsTask(cancellationToken);
    }
    cancellationToken.ThrowIfCancellationRequested();
    await blob.UploadFromFileAsync(storageFile);
}
```

This method uploads the video file to blob storage using a URI specified by the shared access scheme locator. The file is uploaded by using the **UploadFromFileAsync** method of the **CloudBlockBlob** class.

The final step of the upload process is to publish the file for processing by the encoding pipeline. To do this the **UploadFileAsync** method creates a Uri that specifies that the **Publish** method will be called on the web service, with the address of the asset being passed as a parameter. The **PostAsync** method of the **HttpService** class is used to make the call to the web service.

C#

```
public async Task<string> PostAsync(Uri requestUri, string stringifyJsonPostData,
CancellationToken cancellationToken)
{
    using (var httpClient = new HttpClient())
    {
        var postData = new HttpStringContent(stringifyJsonPostData,
            UnicodeEncoding.Utf8, "application/json");
        var response = await httpClient.PostAsync(requestUri,
            postData).AsTask(cancellationToken);

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

This method asynchronously sends data to the web service by using the **HttpClient** class to send HTTP requests and receive HTTP responses from a URI. The call to **HttpClient.PostAsync** sends a POST request to the specified URI as an asynchronous operation, passing data that represents a **Video** instance that contains the metadata for the content to be published, and returns a **Task** of type **HttpResponseMessage** that represents the asynchronous operation. The returned **Task** will complete after the content from the response is read.

When the **UploadFileAsync** method calls **HttpService.PostAsync**, this calls the **Publish** method in the **VideosController** class in the Contoso.Api project.

C#

```
public async Task<HttpResponseMessage> Publish(VideoSaveDTO video)
{
    ...
    var newVideo = MapSaveDTO(video);
    newVideo.EncodingStatus = EncodingState.NotStarted;
    ...
    var videoDetail = await videoService.Save(newVideo);
    ...
    await encodingService.PublishAsset(videoPublish);

    return Request.CreateResponse(HttpStatusCode.Created);
    ...
}
```

The method first uses the **MapSaveDTO** method to convert the **VideoSaveDTO** object to a **VideoDetail** object, before setting its **EncodingState** to **NotStarted**. The **Save** method of the **VideoService** class is then called, followed by the **PublishAsset** method of the **EncodingService** class. In turn, the **Save** method in the **VideoService** class calls the **SaveVideo** method in the **VideoRepository** class, which creates a **VideoEntity** object to persist the video details to the CMS database in Azure Storage. The **PublishAsset** method in the **EncodingService** is shown in the following code example.

C#

```
public async Task PublishAsset(VideoPublish video)
{
    var inputAsset = this.context.Assets.Where(
        a => a.Id == video.AssetId).SingleOrDefault();
    if (inputAsset != null)
    {
        if (inputAsset.AssetFiles.Count() == 0)
        {
            await inputAsset.GenerateFromStorageAsync().ConfigureAwait(false);
        }

        var videoEncodingMessage = new EncodeVideoMessage()
        {
            AssetId = video.AssetId,
            VideoId = video.VideoId,
            IncludeThumbnails = true,
            Resolution = video.Resolution
        };

        ...
        IAzureQueue<EncodeVideoMessage> queue =
            new AzureQueue<EncodeVideoMessage>(
            Microsoft.WindowsAzure.CloudStorageAccount.Parse(
            CloudConfiguration.GetConfigurationSetting(
            "WorkerRoleConnectionString")),
```

```
CloudConfiguration.GetConfigurationSetting("ContosoEncodingQueueName"),
            TimeSpan.FromSeconds(300));
        queue.AddMessage(videoEncodingMessage);
    }
}
```

This method performs two tasks. The first task is to generate an **AssetFile** for the **Asset**, using the **GenerateFromStorageAsync** extension method, and associate it with the **Asset**. It is important to note that the **AssetFile** instance and the media file are two distinct objects. The **AssetFile** instance contains metadata about the media file, whereas the media file contains the actual media content. The second task is to create an **EncodeVideoMessage** instance and add it to the **AzureQueue** instance to begin the encoding process. For more information about the encoding process see "Chapter 4 – Encoding and Processing Video."

Summary

This chapter has described how the Contoso developers incorporated Media Services' uploading functionality into their web service and Windows Store client application. It summarized the decisions that they made in order to support their business requirements, and how they designed the code that performs the upload process.

In this chapter, you saw how to connect to Media Services and ingest content with the Media Service SDK for .NET. The chapter also discussed how to secure your content during the upload process, both when it's in transit and at rest.

The following chapter discusses the next step in the Media Services workflow – encoding and processing uploaded media.

More information

- The page, "Ingesting Assets with the Media Services REST API" describing how to ingest assets into Media Services using the REST API, is available on MSDN.
- You can find the page, "Ingesting Assets in Bulk with the REST API" describing how to use the REST API to ingest assets into Media Services in bulk, on MSDN.
- For information about high speed ingest technology, see "Uploading Media" on MSDN.
- You can find the page, "Managing Media Services Assets across Multiple Storage Accounts" on MSDN.
- For information about ingesting assets in bulk, see the page "Ingesting Assets in Bulk with the Media Services SDK for .NET" on MSDN.
- For information about creating a Media Services account and associate it with a storage account, see "How to Create a Media Services Account" on MSDN.
- The page, "Setup for Development on the Media Services SDK for .NET" describes how to set up a Visual Studio project for Media Services development, is available on MSDN.

- For more information about lazy initialization, see "Lazy Initialization" on MSDN.
- For a detailed description of how to encrypt configuration information see "Encrypting Configuration Information Using Protected Configuration" on MSDN.
- For information about how to connect to a web service from a Windows Store application, see "Connecting to an HTTP server using Windows.Web.Http.HttpClient" on MSDN.

4 - Encoding and Processing Media in Microsoft Azure Media Services

Microsoft Azure Media Services enables you to encode your video to a variety of devices, ranging from desktop PCs to smartphones. To do this you create processing jobs which enable you to schedule and automate the encoding of assets.

This chapter describes how the Contoso developers incorporated Media Services' encoding and processing functionality into their web service. It summarizes the decisions that they made in order to support their business requirements, and how they designed the code that performs the encoding process.

Introduction to video encoding

Uncompressed digital video files can be large and would be too big to deliver over the Internet without first compressing them. Encoding is the process of compressing video and audio using codecs. The quality of the encoded content is determined by the amount of data that is thrown away when the content is compressed. There are many factors that affect what data is thrown away during the compression process, but generally the more complex the data is and the higher the compression ratio, the more data is thrown away. In addition, people watch videos on a variety of devices including TVs with set top boxes, desktop PCs, tablets, and smartphones. Each of these devices has different bandwidth and compression requirements.

Codecs both compress and decompress digital media files. Audio codecs compress and decompress audio, while video codecs compress and decompress video. Lossless codecs preserve all of the data during the compression process. When the file is decompressed the result is a file that is identical to the input file. Lossy codecs throw away some of the data when encoding, and produce smaller files than lossless codecs. The two main codecs used by Media Services to encode are H.264 and VC-1.

Encoders are software or hardware implementations that compress digital media using codecs. Encoders usually have settings that allow you to specify properties of the encoded media, such as the resolution, bitrate, and file format. File formats are containers that hold the compressed media as well as data about the codecs that were used during the compression process. For a list of the codecs and file formats supported by Media Services for import see "Supported input formats." The following table lists the codecs and file formats that are supported for export.

File format	Video codec	Audio codec
Windows Media (*.wmv, *.wma)	VC-1 (Simple, Main, and Advanced profiles)	Windows Media Audio (Standard, Professional, Voice, Lossless)
MP4 (.mp4)	H.264 (Baseline, Main, and High profiles)	AAC-LC, HE-AAC v1, HE-AAC v2, Dolby Digital Plus
Smooth Streaming (PIFF 1.1) (*.ismv, *.isma)	VC-1 (Advanced profile) H.264 (Baseline, Main, and High profiles)	Windows Media Audio (Standard, Professional) AAC-LC, HE-AAC v1, HE-AAC v2

For information about additional supported codecs and filters in Media Services, see "Codec Objects" and "DirectShow Filters."

Resolution specifies how many lines make up a full video image. Typically resolutions are 1080p and 720p for high definition, and 480p for standard definition. The bitrate of a video is the number of bits recorded per sec, and is usually specified as kilobits per second (kbps). The higher the bitrate the higher the quality of video. Videos can be encoded using a constant bitrate or a variable bitrate.

In constant bitrate encoding (CBR) a maximum bitrate is specified that the encoder can generate. If the video being encoded requires a higher bitrate then the resulting video will be of poor quality. CBR encoding is useful when there's a requirement to stream a video at a predictable bit rate with a consistent bandwidth utilization.

While CBR encoding aims to maintain the bit rate of the encoded media, variable bit rate (VBR) encoding aims to achieve the best possible quality of the encoded media. A higher bitrate is used for more complex scenes, with a lower bitrate being used for less complex scenes. VBR encoding is more computation intensive, and often involves multiple passes when encoding video.

Encoding for delivery using Azure Media Services

Media Services provides a number of *media processors* that enable video to be processed. Media processors handle a specific processing task, such as encoding, format conversion, encrypting, or decrypting media content. Encoding video is the most common Media Services processing operation, and it is performed by the Azure Media Encoder. The Media Encoder is configured using encoder preset strings, with each preset specifying a group of settings required for the encoder. For a list of all the presets see "Appendix B –Azure Media Encoder Presets."

Media Services supports progressive download of video and streaming. When encoding for progressive download you encode to a single bitrate. However, you could encode a video multiple times and have a collection of single bitrate files from which a client can choose. When a client chooses a bitrate the entire video will be displayed at that bitrate. However, if network conditions degrade playback of the video may pause while enough data is buffered to be able to continue.

To be able to stream content it must first be converted into a streaming format. This can be accomplished by encoding content directly into a streaming format, or converting content that has already been encoded into H.264 into a streaming format. The second option is performed by the Azure Media Packager, which changes the container that holds the video, without modifying the video encoding. The Media Packager is configured through XML rather than through string presets. For more information about the XML configuration see "Task Preset for Azure Media Packager."

There are two types of streaming offered by Media Services:

- Single bitrate streaming
- Adaptive bitrate streaming

With single bitrate streaming a video is encoded to a single bitrate stream and divided into chunks. The stream is delivered to the client one chunk at a time. The chunk is displayed and the client then requests the next chunk. When encoding for single bitrate streaming you can encode to a number of different bitrate streams and clients can select a stream. With single bitrate streaming, once a bitrate stream is chosen the entire video will be displayed at that bitrate.

When encoding for adaptive bitrate streaming you encode to an MP4 bitrate set that creates a number of different bitrate streams. These streams are also broken into chunks. However, adaptive bitrate technologies allow the client to determine network conditions and select from among several bitrates. When network conditions degrade, the client can select a lower bitrate allowing the video to continue to play at a lower video quality. Once network conditions improve the client can switch back to a higher bitrate with improved video quality.

Media Services supports three adaptive bitrate streaming technologies:

- Smooth streaming, created by Microsoft
- HTTP Live Streaming (HLS), created by Apple
- MPEG-DASH, an ISO standard

Media Services enables you to encode and stream video to a variety of devices. The following table summarizes the streaming technology supported by different device types.

Device type	Supports	Example presets
Windows	Smooth streaming MPEG-DASH	H264 Broadband 1080p H264 Adaptive Bitrate MP4 Set 1080p H264 Smooth Streaming 1080p
Xbox	Smooth streaming	H264 Smooth Streaming 720p Xbox Live ADS
iOS	HLS Smooth streaming (with the Smooth Streaming Porting Kit)	H264 Broadband 1080p H264 Adaptive Bitrate MP4 Set 1080p H264 Smooth Streaming 1080p
Android	Smooth Streaming via the OSMF plug-in, when the device supports Flash HLS (Android OS 3.1 and greater)	H264 Broadband 720p H264 Smooth Streaming 720p

Smooth streaming is the preferred adaptive bitrate streaming technology for Microsoft platforms. There are a number of approaches to creating smooth streaming assets:

- Encode your single video file using one of the H.264 smooth streaming task presets to encode directly into smooth streaming. For more information see "Appendix B –Azure Media Encoder Presets."
- Encode your single video file using one of the H.264 adaptive bitrate task presets using the Azure Media Encoder, and then use the Azure Media Packager to convert the adaptive bitrate MP4 files to smooth streaming.
- Encode your video locally to a variety of bit rates, and then create a manifest file describing the video files. After uploading the files to the Azure Storage account associated with your Azure Media account, use the Azure Media Packager to convert the MP4 files into smooth streaming files.

- Encode your video to MP4 and using dynamic packaging to automatically convert the MP4 to smooth streaming. For more information about dynamic packaging see "Dynamic packaging."

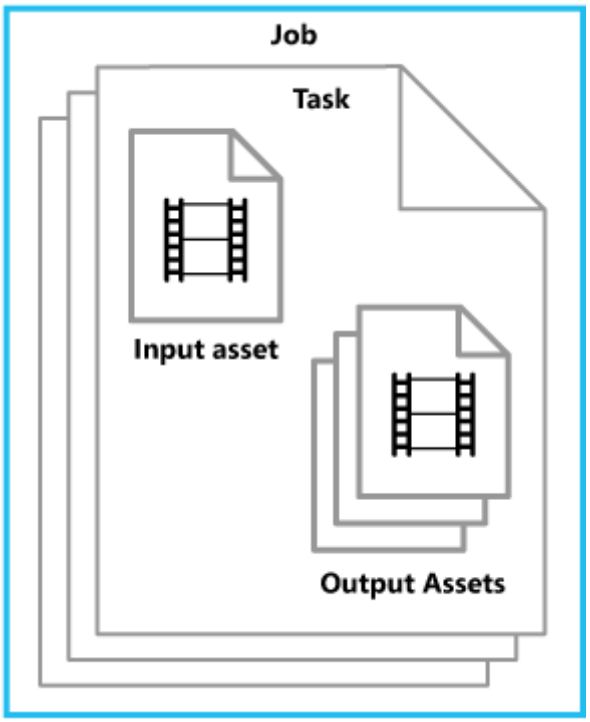
If you intend to protect your content with PlayReady you should use the Azure Media Encoder to encode directly to Smooth Streaming, and then use the Azure Media Packager to protect your media.

Note: To convert WMV files to Smooth Streaming you must first encode your files from WMV to H.264. WMV is a video codec that typically has an ASF container format, with H.264 being a video codec that can be used with the MP4 container format. Smooth Streaming uses a variant of MP4 called fragmented MP4, or F-MP4. Smooth Streaming is an adaptive streaming format that requires a set of files of different bitrates, all encoded with fragments that are aligned across the bitrates. Therefore, a set of MP4 files that are encoded with aligned fragments can be converted to F-MP4 without requiring a re-encode. However, this is not the case with WMV files. For more information see "Smooth Streaming Technical Overview."

Creating encoding jobs in Azure Media Services

After media has been uploaded into Media Services it can be encoded into one of the formats supported by the Media Services Encoder. Media Services Encoder supports encoding using the H.264 and VC-1 codecs, and can generate MP4 and Smooth Streaming content. However, MP4 and Smooth Streaming content can be converted to Apple HLS v3 or MPEG-DASH by using dynamic packaging. For more information about dynamic packaging see "Dynamic packaging." For information about the input and output formats supported by Media Services see "Supported input formats" and "Introduction to encoding."

Encoding jobs are created and controlled using a **Job**. Each **Job** contains metadata about the processing to be performed, and contains one or more **Tasks** that specify a processing task, its input **Assets**, output **Assets**, and a media processor and its settings. The following figure illustrates this relationship.



The relationship between jobs, tasks, and assets

Tasks within a **Job** can be chained together, where the output asset of one task is given as the input asset to the next task. By following this approach one **Job** can contain all of the processing required for a media presentation.

> The maximum number of **Tasks** per **Job** is 50.
>
> The maximum number of **Assets** per **Task** is 50
>
> The maximum number of **Assets** per **Job** is 100. This includes queued, finished, active, and canceled jobs. However, it doesn't include deleted jobs.

Accessing Azure Media Services media processors

A standard task that's required for most processing jobs is to call a specific media processor to process the job.

>  A media processor is a component that handles a specific processing task such as encoding, format conversion, encryption, or decrypting media content.

The following table summarizes the media processors supported by Media Services.

Media processor name	Description
Azure Media Encoder	Allows you to run encoding tasks using the Media Encoder.
Azure Media Packager	Allows you to convert media assets from MP4 to Smooth Streaming format. In addition, allows you to

	convert media assets from Smooth Streaming format to HLS format.
Azure Media Encryptor	Allows you to encrypt media assets using PlayReady Protection.
Storage Decryption	Allows you to decrypt media assets that were encrypted using storage encryption.

To use a specific media processor you should pass the name for the processor into the **GetLatestMediaProcessorByName** method, which is shown in the following code example.

C#

```
private IMediaProcessor GetLatestMediaProcessorByName(string mediaProcessorName)
{
    var processor = this.context.MediaProcessors.Where(p => p.Name ==
mediaProcessorName)
        .ToList().OrderBy(p => new Version(p.Version)).LastOrDefault();

    if (processor == null)
    {
        throw new ArgumentException(string.Format("Unknown media processor: {0}",
mediaProcessorName));
    }

    return processor;
}
```

The method retrieves the specified media processor and returns a valid instance of it. The following code example shows how you'd use the **GetLatestMediaProcessorByName** method to retrieve the Azure Media Encoder processor.

C#

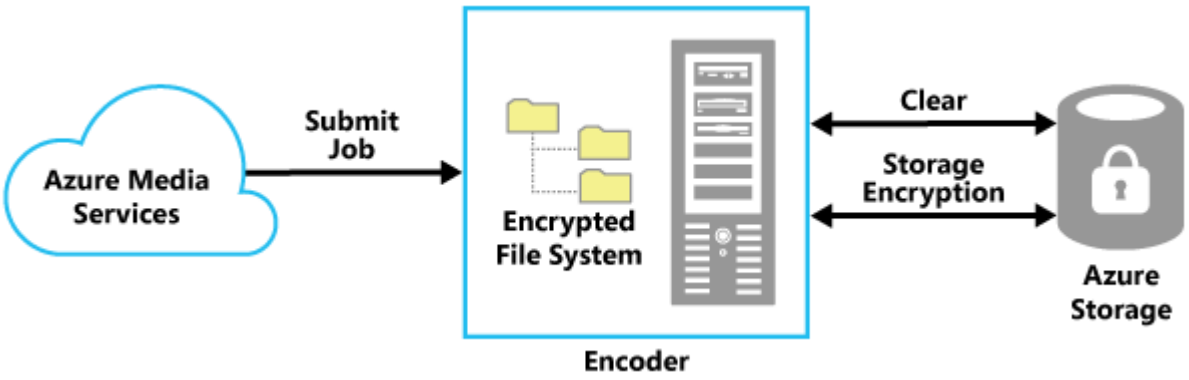
```
IMediaProcessor mediaProcessor =
this.GetLatestMediaProcessorByName(MediaProcessorNames.WindowsAzureMediaEncoder);
```

Securely encoding media within Azure Media Services

When encoding encrypted assets you must specify the encryption option when adding the output asset to the processing task. The encryption of each asset created by a job is controlled by specifying one of the **AssetCreationOptions** enumeration values for each task in the job.

Any encrypted assets will be decrypted before a processing operation and stored in the encrypted file system on the Azure Compute node that is processing the task. The media processors then perform the required operations on the media stored in the encrypted file system and the output of each task is written to storage.

The following figure summarizes how media can be protected during the encoding and packaging process.



Media encryption options during the encoding and packaging process

The Contoso web service does not use any encryption because videos are encoded for progressive download and streaming. However, when developing a commercial video-on-demand service you should encrypt the content both in transit and at rest.

If you want to encode a video and secure it for storage you should specify **AssetCreationOptions.StorageEncrypted** when creating the output asset for the encoding task. When a storage encrypted asset is downloaded using one of the Media Services SDKs the SDK will automatically decrypt the asset as part of the download process.

If you want to encode and package a video for streaming or progressive download you should specify **AssetCreationOptions.None** when creating the output asset for the encoding task.

Scaling Azure Media Services encoding jobs

By default each Media Services account can have one active encoding task at a time. However, you can reserve encoding units that allow you to have multiple encoding tasks running concurrently, one for each encoding reserved unit your purchase. New encoding reserved units are allocated almost immediately.

The number of encoding reserved units is equal to the number of media tasks that can be processed concurrently in a given account. For example, if your account has 5 reserved units then 5 media tasks can run concurrently. The remaining tasks will wait in the queue and will be processed sequentially as soon as a running task completes.

If an account doesn't have any reserved units then tasks will be processed sequentially. In this scenario the time between one task finishing and the next one starting will depend on the availability of system resources.

The number of encoding reserved units can be configured on the Encoding page of the Azure Management Portal.

By default every Media Services account can scale to up to 25 encoding reserved units. A higher limit can be requested by opening a support ticket. For more information about opening a support ticket see "Requesting Additional Reserved Units."

For more information about scaling Media Services see "How to Scale a Media Service."

Accessing encoded media in Azure Media Services

Accessing content in Media Services always requires a locator. A locator combines the URL to the media file with a set of time-based access permissions. There are two types of locators – shared access signature locators and on-demand origin locators.


You cannot have more than five unique locators associated with a given asset at one time. This is due to shared access policy restrictions set by Azure Blob Storage service.

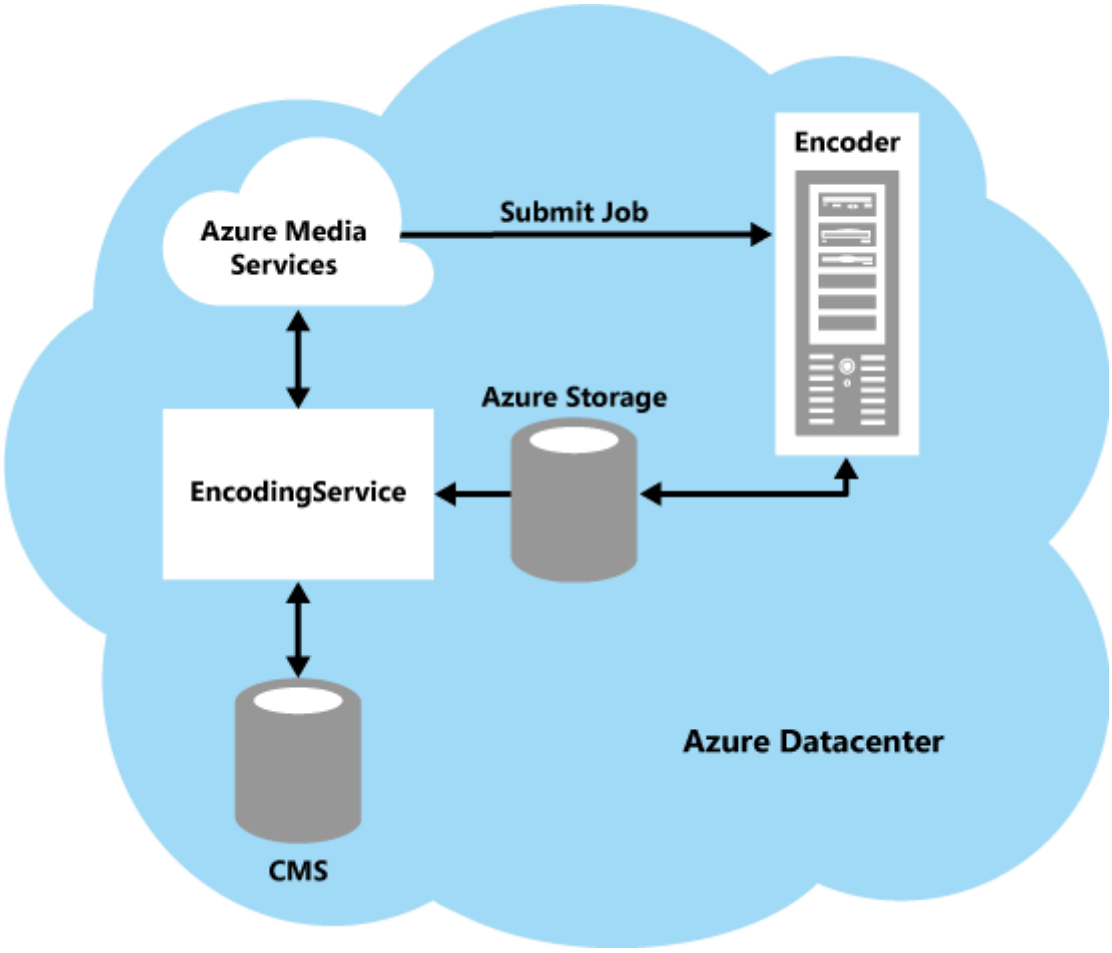
A shared access signature locator grants access rights to a specific media asset through a URL. You can grant users who have the URL access to a specific resource for a period of time by using a shared access signature locator, in addition to specifying what operations can be performed on the resource.

On-demand origin locators are used when streaming content to a client application, and are exposed by the Media Services Origin Service which pulls the content from Azure Storage and delivers it to the client. An on-demand origin locator URL will point to a streaming manifest file in an asset. For more information about the origin service see "Origin Service."


Locators are not designed for managing per-user access control. To give different access rights to different individuals, use Digital Rights Management (DRM) solutions.

Encoding process in the Contoso Azure Media Services web service

The following figure shows a high-level overview of the Contoso encoding process. The encoding process is managed by the **EncodingService** class in the Contoso.Domain.Services.Impl project.



A high-level overview of the Contoso encoding process

The **EncodingService** class in the Contoso web service retrieves the asset details from the CMS database and passes the encoding job to Media Services, where it's submitted to the Azure Media Encoder. The encoding job and video details are saved to the CMS database while the Media Encoder processes the job, retrieving the input asset from Azure Storage, and writing the output assets to Azure Storage. The Contoso web service always encodes videos to adaptive bitrate MP4s, and then uses dynamic packaging to convert the adaptive bitrate MP4s to Smooth Streaming, HLS, or MPEG-DASH, on demand. When encoding is complete Media Services notifies the **EncodingService** class, which generates locator URLs to the output assets in Azure Storage, and updates the encoding job and video details in the CMS database. For more information about dynamic packaging see "Dynamic packaging."

This process can be decomposed into the following steps for encoding content with Media Services:

1. Create a new **VideoEncodingMessage** and add it to the **ContosoEncodingQueue**.
2. Poll the **ContosoEncodingQueue** and convert the received **VideoEncodingMessage** to an **EncodingRequest**.
3. Delete the **VideoEncodingMessage** from the **ContosoEncodingQueue**.
4. Process the **EncodingRequest**.
    a. Create a new **Job**.

b. Retrieve Azure Media Encoder media processor to process the job.

c. Create a new **EncodingPipeline** to encode the video.

d. Add a **VideoEncodingPipelineStep** to the **EncodingPipeline**.

e. Add a **ThumbnailEncodingPipelineStep** to the **EncodingPipeline**, if required.

f. Add a **ClipEncodingPipelineStep** to the **EncodingPipeline**, if required.

g. Configure the **Job**.

   i. Create a **Task** in the **VideoEncodingPipelineStep** and specify input and output assets for the **Task**.

   ii. Create a **Task** in the **ThumbnailEncodingPipelineStep** and specify input and output assets for the Task.

   iii. Create a **Task** in the **ClipEncodingPipelineStep** and specify input and output assets for the **Task**.

h. Submit the **Job** to Azure Media Services.

   i. A new **JobNotificationMessage** is added to the **ContosoJobNotificationQueue**.

i. Create a new **EncodingJob** and populate it with job information, before storing it in the CMS database.

j. Update the **EncodingStatus** of the **Job** from **NotStarted** to **Encoding**.

5. Poll the **ContosoJobEncodingQueue** and convert the received **JobNotificationMessage** to a **JobNotification**.

6. Delete the **JobNotificationMessage** from the **ContosoJobEncodingQueue**.

7. Process the **JobNotification**.

   a. When the **JobState** is **Finished**, retrieve the job and video details from the CMS database.

   b. Process the output assets from the **Job**.

      i. Process the **VideoEncodingPipelineStep** output assets.

         a. Create on-demand origin and shared access signature locators for the output asset.

         b. Generate URIs for smooth streaming, HLS, MPEG-DASH, and progressive download versions of the output asset.

         c. Add the URIs to **VideoPlay** objects, and add the **VideoPlay** objects to the **VideoDetail** object.

      ii. Process the **ThumbnailEncodingPipelineStep** output assets.

         a. Create on-demand origin and shared access signature locators for the output asset.

b. Generate URIs for the thumbnail images.

c. Add the URIs to **VideoThumbnail** objects, and add them to the **VideoDetail** object.

iii. Process the **ClipEncodingPipelineStep** output assets.

a. Create on-demand origin and shared access signature locators for the output asset.

b. Generate URIs for the video clip assets.

c. Add the URIs to **VideoPlay** objects, and add the **VideoPlay** objects to the **VideoDetail** object.

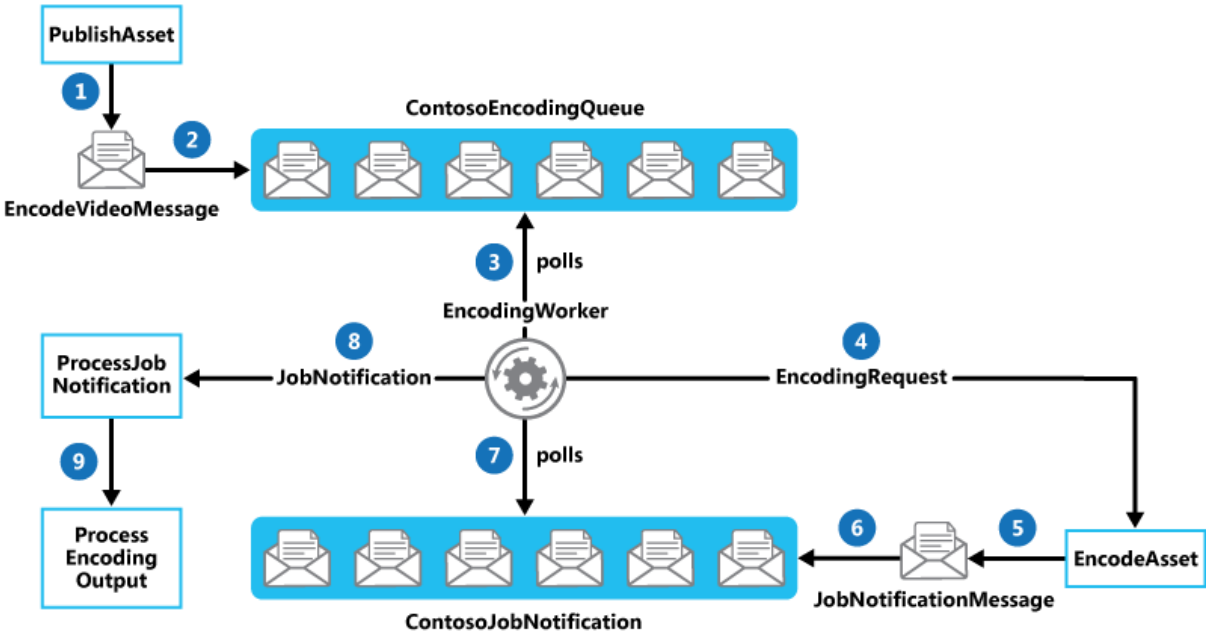
c. Save the updated video details and job details to the CMS database.

Media Services has the ability to deliver notification messages to the Azure Storage Queues when processing media jobs. The Contoso developers decided to use Media Services notifications during the encoding process. The advantages of this are that it provides an easy mechanism for managing the encoding jobs submitted by multiple clients, and encoding progress can be monitored through job notification messages, if required.

When a video is uploaded and published it's added to a queue named **ContosoEncodingQueue** which stores encoding jobs, and then moved to the **ContosoJobNotificationQueue** for encoding. When the encoding job completes the next step in the content publishing workflow is triggered, which is to process the assets output from the encoding job, and to update the CMS database.

Azure Storage Queues must be polled – they are not a push service.

The following figure shows a high level overview of how the **ContosoEncodingQueue** and **ContosoJobNotification** queue are used in the encoding process. The diagram shows the method names in the **EncodingService** class that initiate and manage the encoding process.



Use of queues in the encoding process

> **Note:** Azure Storage Queues do not provide a guaranteed first-in-first-out (FIFO) delivery. For more information see "Azure Queues and Azure Service Bus Queues – Compared and Contrasted."

As mentioned in the previous chapter, the **PublishAsset** method in the **EncodingService** class is responsible for starting the encoding process.

C#

```
public async Task PublishAsset(VideoPublish video)
{
    ...
    var videoEncodingMessage = new EncodeVideoMessage()
    {
        AssetId = video.AssetId,
        VideoId = video.VideoId,
        IncludeThumbnails = true,
        Resolution = video.Resolution
    };

    ...

    IAzureQueue<EncodeVideoMessage> queue = new AzureQueue<EncodeVideoMessage>(
        Microsoft.WindowsAzure.CloudStorageAccount.Parse(
        CloudConfiguration.GetConfigurationSetting("WorkerRoleConnectionString")),
        CloudConfiguration.GetConfigurationSetting("ContosoEncodingQueueName"),
        TimeSpan.FromSeconds(300));
    queue.AddMessage(videoEncodingMessage);
}
```

The method creates an **EncodeVideoMessage** instance and adds it an **AzureQueue** named **ContosoEncodingQueue**. Every video that will be encoded or otherwise processed by Media Services

must be added to this queue. Each **EncodeVideoMessage** instance contains properties that specify the details of the video to be encoded.

The Contoso.Azure project specifies a worker role named Contoso.EncodingWorker that is responsible for managing the two queues used in the encoding process. When a video is published it's added to the **ContosoEncodingQueue** for encoding, and once the encoding is complete it's moved to the **ContosoJobNotificationQueue**. The Contoso.EncodingWorker project contains the classes that make up the worker role.

The **Run** method in the **WorkerRole** class in the Contoso.EncodingWorker project is responsible for managing the two queues.

C#

```
public override void Run()
{
    ...
    var contosoEncodingQueue =
        this.container.Resolve<IAzureQueue<EncodeVideoMessage>>("Standard");
    var contosoEncodingCompleteQueue =
        this.container.Resolve<IAzureQueue<JobNotificationMessage>>("Standard");

     BatchMultipleQueueHandler
         .For(contosoEncodingQueue, GetStandardQueueBatchSize())
         .Every(TimeSpan.FromSeconds(GetSummaryUpdatePollingInterval()))
         .WithLessThanTheseBatchIterationsPerCycle(
              GetMaxBatchIterationsPerCycle())
         .Do(this.container.Resolve<EncodeVideoCommand>());

     BatchMultipleQueueHandler
         .For(contosoEncodingCompleteQueue, GetStandardQueueBatchSize())
         .Every(TimeSpan.FromSeconds(GetSummaryUpdatePollingInterval()))
         .WithLessThanTheseBatchIterationsPerCycle(
             GetMaxBatchIterationsPerCycle())
         .Do(this.container.Resolve<JobNotificationCommand>());
     ...
}
```

This method sets up two **BatchMultipleQueueHandler**s to process the **ContosoEncodingQueue** and the **ContosoJobNotificationQueue**. The **BatchMultipleQueueHandler<T>** class implements the **For**, **Every**, and **Do** methods. The **Do** method in turn calls the **Cycle** method, which calls the **PreRun**, **Run**, and **PostRun** methods of the batch command instance (any command which derives from **IBatchCommand**). Therefore, the first **BatchMultipleQueueHandler** polls the **ContosoEncodingQueue** every 10 seconds and for every **EncodingVideoMessage** on the queue, runs the **PreRun**, **Run**, and **PostRun** methods of the **EncodeVideoCommand** instance. The second **BatchMultipleQueueHandler** polls the **ContosoJobNotificationQueue** every 10 seconds and for every **JobNotificationMessage** on the queue, runs the **PreRun**, **Run**, and **PostRun** methods of the **JobNotificationCommand** instance. The 10 second time interval is set by the **SummaryUpdatePollingInterval** constant stored in configuration, and is retrieved by the **GetSummaryUpdatePollingInterval** method in the **WorkerRole** class. After the **Run** method of a batch command has executed the message is deleted from the appropriate queue.

Therefore, when the **PublishAsset** method of the **EncodingService** class places an **EncodeVideoMessage**, containing the details of the video to be encoded, onto the **ContosoEncodingQueue**, when the queue is polled the **Run** method of the **EncodeVideoCommand** class is invoked.

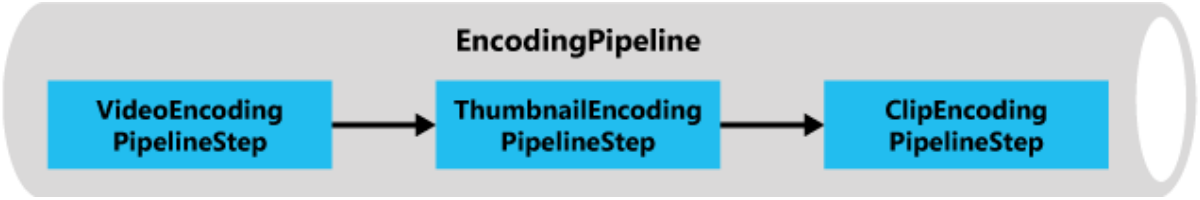
C#

```
public bool Run(EncodeVideoMessage message)
{
    var encodingRequest = new EncodingRequest()
    {
        AssetId = message.AssetId,
        ClipStartTime = message.ClipStartTime,
        ClipEndTime = message.ClipEndTime,
        IncludeThumbnails = message.IncludeThumbnails,
        Resolution = message.Resolution,
        VideoId = message.VideoId
    };

    this.encodingService.EncodeAsset(encodingRequest);
    return true;
}
```

This method converts the **EncodeVideoMessage** to a new **EncodingRequest** instance and then calls the **EncodeAsset** method of the **EncodingService** class. After the **Run** method has executed the **EncodeVideoMessage** is deleted from the **ContosoEncodingQueue**.

Creating the video encoding pipeline for Azure Media Services

The **EncodeAsset** method retrieves the media asset to be encoded and then creates a new **IJob** instance and gets the media encoder from the context, before creating a new instance of the **EncodingPipeline** class. The **EncodingPipeline** is used to place video encoding steps into a pipeline. The following figure shows an overview of the steps in the **EncodingPipeline**.



An overview of the steps in the EncodingPipeline

The pipeline consists of three steps:

1. A **VideoEncodingPipelineStep**.
2. A **ThumbnailEncodingPipelineStep**.
3. A **ClipEncodingPipelineStep**.

When an encoding job completes there is no information in the tasks or output assets associated with the job that identify what it is. By using an encoding pipeline, and pipeline steps, you are able to append a suffix to each output asset and then match them up when the encoding completes.

The following code example shows how the **EncodeAsset** method in the **EncodingService** class creates the encoding pipeline.

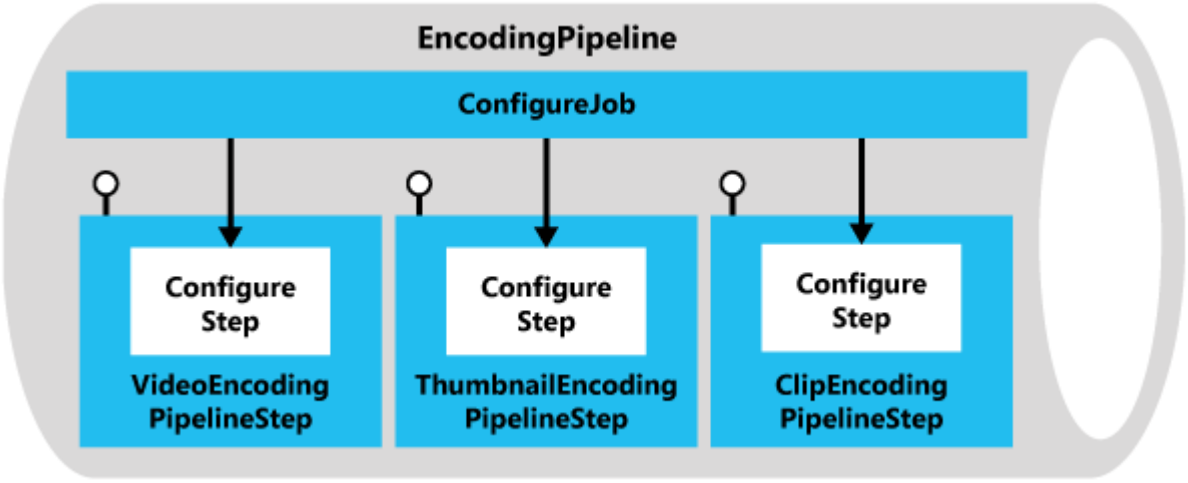
C#

```
public async Task EncodeAsset(EncodingRequest encodingRequest)
{
    ...
    // create a new instance of the encoding pipeline
    EncodingPipeline encodingPipeline = new EncodingPipeline();

    // add the video to the encoding pipeline
    VideoEncodingPipelineStep videoEncodingStep =
        new VideoEncodingPipelineStep(inputAsset, encodingRequest.Resolution);
    encodingPipeline.AddStep(videoEncodingStep);

    if (encodingRequest.IncludeThumbnails)
    {
        // add the thumbnails to the encoding pipeline
        ThumbnailEncodingPipelineStep thumbnailEncodingStep =
            new ThumbnailEncodingPipelineStep(inputAsset);
        encodingPipeline.AddStep(thumbnailEncodingStep);
    }

    if(encodingRequest.ClipEndTime.Ticks > 0)
    {
        ClipEncodingPipelineStep clipEncodingStep = new ClipEncodingPipelineStep(
            inputAsset, encodingRequest.ClipStartTime,
            encodingRequest.ClipEndTime, encodingRequest.Resolution);
        encodingPipeline.AddStep(clipEncodingStep);
    }

    // configure the job; adds the steps as tasks to the job
    encodingPipeline.ConfigureJob(job, mediaProcessor);
    ...
}
```

An **EncodingPipeline** instance will always have a **VideoEncodingPipelineStep**. However, a **ThumbnailEncodingPipelineStep** will only be added to the **EncodingPipeline** if the **IncludeThumbnails** property of the **EncodingRequest** is set to **true**. Similarly, a **ClipEncodingPipelineStep** will only be added to the **EncodingPipeline** if the **ClipEndTime.Ticks** property of the **EncodingRequest** is greater than zero.

The **VideoEncodingPipelineStep**, **ThumbnailEncodingPipelineStep**, and **ClipEncodingPipelineStep** classes all implement the **IEncodingPipelineStep** interface, which specifies that implementing classes must provide the **ConfigureStep** and **ProcessOutput** methods.

The following figure shows the methods involved in configuring the **EncodingPipeline** steps.



The methods involved in configuring the EncodingPipline steps

The **ConfigureJob** method of the **EncodingPipeline** class is shown in the following code example.

C#

```
public void ConfigureJob(IJob job, IMediaProcessor mediaProcessor)
{
    ...
    foreach(IEncodingPipelineStep step in this.steps)
    {
        step.ConfigureStep(job, mediaProcessor);
    }
}
```

This method simply calls the **ConfigureStep** method of any of the added **IEncodingPipelineStep**s. The overall effect is to add the steps as tasks to the job.

Configuring the video encoding pipeline step

An **EncodingPipeline** will always contain a **VideoEncodingPipelineStep**, which is responsible for encoding a video.

The **VideoEncodingPipelineStep** class defines a dictionary that contains the encoding presets that can be chosen when uploading a video using the client apps. The dictionary specifies four encoding presets:

Preset	Encoder task preset
1080p	H264AdaptiveBitrateMP4Set1080p
720p	H264AdaptiveBitrateMP4Set720p

480p 16x9	H264AdaptiveBitrateMP4SetSD16x9
480p 4x3	H264AdaptiveBitrateMP4SetSD4x3

These presets produce assets at different resolutions and aspect ratios for delivery via one of many adaptive streaming technologies after suitable packaging. If no encoding preset is specified the pipeline defaults to using the 720p preset.

> Not all videos can be encoded using these presets, for example low bitrate videos. In such cases you should create custom encoding presets.

The following code example shows the **ConfigureStep** method of the **VideoEncodingPiplineStep** class.

C#

```
public void ConfigureStep(IJob job, IMediaProcessor mediaProcessor)
{
    ITask encodingTask = job.Tasks.AddNew(
        this.inputAsset.Name + EncodingTaskSuffix,
        mediaProcessor,
        this.encodingPreset,
        TaskOptions.ProtectedConfiguration);
    encodingTask.InputAssets.Add(this.inputAsset);
    encodingTask.OutputAssets.AddNew(this.inputAsset.Name + EncodingOuputSuffix,
        AssetCreationOptions.None);
}
```

The method declares a task, passing the task a name made up of the input asset name with "_EncodingTask" appended to it, a media processor instance, a configuration string for handling the processing job, and a **TaskCreationOptions** setting that specifies that the configuration data should be encrypted. The task is then added to the **Tasks** collection of the job. An input asset is then specified for the task, along with an output asset whose filename is made up of the input asset name with "_EncodingOutput" appended to it.

>  By default, all assets are created as storage encrypted assets. To output an unencrypted asset for playback you must specify **AssetCreationOptions.None**.

Configuring the thumbnail encoding pipeline step

The **ThumbnailEncodingPipelineStep** class is responsible for producing thumbnail image files from a video file. In the Contoso video apps these thumbnail images are used to represent each video on the main page.

A **ThumbnailEncodingPipelineStep** will only be added to the **EncodingPIpeline** if the **IncludeThumbnails** property of the **EncodingRequest** is set to **true**. In the Contoso web service this property is always set to **true**.

The following code example shows the **ConfigureStep** method of the **ThumbnailEncodingPipelineStep** class.

C#

```
public void ConfigureStep(IJob job, IMediaProcessor mediaProcessor)
{
    ITask thumbnailTask = job.Tasks.AddNew(
        this.inputAsset.Name + ThumbnailTaskSuffix,
        mediaProcessor,
        this.thumbnailPresetXml,
        TaskOptions.ProtectedConfiguration);
    thumbnailTask.InputAssets.Add(this.inputAsset);
    thumbnailTask.OutputAssets.AddNew(this.inputAsset.Name +
        ThumbnailOutputSuffix, AssetCreationOptions.None);
}
```

This method declares a task, passing the task a name made up of the input asset name with "_ThumbnailTask" appended to it, a media processor instance, a custom configuration XML preset for handling the processing job, and a **TaskCreationOptions** setting that specifies that the configuration data should be encrypted. The custom configuration XML preset specifies the settings to use when creating the task. The task is then added to the **Tasks** collection of the job. An input asset is then specified for the task, along with an output asset whose filename is made up of the input asset name with "_ThumbnailOutput" appended to it. In order to output an unencrypted asset the **AssetCreationOptions.None** enumeration value is specified.

The following code example shows the XML configuration preset used to create thumbnails.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Thumbnail Size="50%,*" Type="Jpeg"
    Filename="{OriginalFilename}_{Size}_{ThumbnailTime}_{ThumbnailIndex}_{Date}
        _{Time}.{DefaultExtension}">
    <Time Value="10%" /></Thumbnail>
```

There are two primary elements:

- The **<Thumbnail>** element that specifies general settings for the thumbnail image that will be generated.
- The **<Time>** element that specifies the time in the source video stream from which a thumbnail will be generated.

The **<Thumbnail>** element specifies that the generated thumbnail should be a JPEG that's 50% of the height of the video, with the aspect ratio maintained. A template is also specified for producing the thumbnail filename. The **<Time>** element specifies that the thumbnail will be generated from the video data 10% of the way through the video stream. For more information about customizing the settings of a thumbnail file see "Task Preset for Thumbnail Generation."

Although we only generate one thumbnail image per video, the CMS allows multiple thumbnail URLs for each video to be stored in the database.

Configuring the clip encoding pipeline step

The **ClipEncodingPipelineStep** class is responsible for producing a clip (a short segment of video) from the video being encoded.

The Contoso Video web client is the only client that demonstrates producing clips from a video.

A **ClipEncodingPipelineStep** will only be added to the **EncodingPipeline** if the **ClipEndTime.Ticks** property of the **EncodingRequest** is greater than zero.

The **ClipEncodingPipelineStep** class defines a dictionary that contains the encoding presets that can be chosen when uploading a video using the client apps. The dictionary specifies the same four encoding presets that are used by the **VideoEncodingPipelineStep** class. Therefore, when a user selects an encoding preset it is used by both the **VideoEncodingPipelineStep** class and the **ClipEncodingPipelineStep** class, with the **ClipEncodingPipelineStep** class also defaulting to using the 720p preset if no encoding preset is specified.

The following code example shows the **ConfigureStep** method of the **ClipEncodingPipelineStep** class.

C#

```
public void ConfigureStep(IJob job, IMediaProcessor mediaProcessor)
{
    var clipXml = this.clipPresetXml.Replace("%startTime%",
        clipStartTime.ToString( @"hh\:mm\:ss"));
    clipXml = clipXml.Replace("%endTime%", this.clipEndTime.ToString(
        @"hh\:mm\:ss"));

    ITask clipTask = job.Tasks.AddNew(
        this.inputAsset.Name + ClipTaskSuffix,
        mediaProcessor,
        clipXml,
        TaskOptions.ProtectedConfiguration);

    clipTask.InputAssets.Add(this.inputAsset);
    clipTask.OutputAssets.AddNew(this.inputAsset.Name + ClipOuputSuffix,
        AssetCreationOptions.None);
}
```

This method updates the start and end time in the clip XML preset data, with the times specified by the user. The clip XML preset data was retrieved by the **ClipEncodingPipelineStep** constructor. The method then declares a task, passing the task a name made up of the input asset name with "_ClipTask" appended to it, a media processor instance, a configuration string for handling the processing job, and a **TaskCreationOptions** setting that specifies that the configuration data should be encrypted. The task is then added to the **Tasks** collection of the job. An input asset is then specified for the task, along with an output asset whose filename is made up of the input asset name with "_ClipOutput" appended to it. In order to output an unencrypted asset the **AssetCreationOptions.None** enumeration value is specified.

Handling job notifications from Azure Media Services

Once the **EncodingPipline**, and hence the job, has been configured it's added to the **ContosoJobNotificationQueue**, as shown in the following code example.

C#

```
public async Task EncodeAsset(EncodingRequest encodingRequest)
{
    ...
    // create a NotificationEndPoint queue based on the endPointAddress
    string endPointAddress = CloudConfiguration
        .GetConfigurationSetting("ContosoJobNotificationQueueName");

    // setup the notificationEndPoint based on the queue and endPointAddress
    this.notificationEndPoint =
        this.context.NotificationEndPoints.Create(Guid.NewGuid().ToString(),
            NotificationEndPointType.AzureQueue, endPointAddress);

    if (this.notificationEndPoint != null)
    {
        job.JobNotificationSubscriptions
            .AddNew(NotificationJobState.FinalStatesOnly,
```

```
                this.notificationEndPoint);
        await job.SubmitAsync().ConfigureAwait(false);

        // save the job information to the CMS database
        var encodingJob = new EncodingJob()
        {
            EncodingJobUuId = job.Id,
            EncodingTasks = new List<EncodingTask>(),
            VideoId = encodingRequest.VideoId
        };

        foreach(var task in job.Tasks)
        {
            var encodingTask = new EncodingTask() { EncodingTaskUuId = task.Id };
            foreach(var asset in task.InputAssets)
            {
                encodingTask.AddEncodingAsset(new EncodingAsset()
                    { EncodingAssetUuId = asset.Id, IsInputAsset = true });
            }

            encodingJob.EncodingTasks.Add(encodingTask);
        }

        await this.jobRepository.SaveJob(encodingJob).ConfigureAwait(false);
        await this.UpdateEncodingStatus(job,EncodingState.Encoding)
            .ConfigureAwait(false);
    }
}
```

This code first retrieves the endpoint address for the **ContosoJobNotificationQueue** from the configuration file. This queue will receive notification messages about the encoding job, with the **JobNotificationMessage** class mapping to the notification message format. Therefore, messages received from the queue can be deserialized into objects of the **JobNotificationMessage** type. The notification endpoint that is mapped to the queue is then created, and attached to the job with the call to the **AddNew** method. **NotificationJobState.FinalStatesOnly** is passed to the **AddNew** method to indicate that we are only interested in the final states of the job processing.

> If **NotificationJobState.All** is passed you will receive all the state changes (Queued -> Scheduled -> Processing -> Finished). However, because Azure Storage Queues don't guarantee ordered delivery it would be necessary to use the **Timestamp** property of the **JobNotificationMessage** class to order messages. In addition, duplicate notification messages are possible, so the **ETag** property on the **JobNotificationMessage** can be used to query for duplicates.
>
> It is possible that some state change notifications will be skipped.

> **Note:** While the recommended approach to monitor a job's state is by listening to notification messages, an alternative is to check on a job's state by using the **IJob.State** property. However, a notification message about a job's completion could arrive before the **IJob.State** property is set to **Finished**.

The job is then asynchronously submitted, before the job information is saved to the CMS database, with an **EncodingJob** object (which contains **EncodingTask** and **EncodingAsset** objects) representing the job information. Finally, the **UpdateEncodingStatus** method is called to update the **EncodingState** for the video from **NotStarted** to **Encoding** (the **Publish** method in the **VideosController** class was responsible for setting the **EncodingState** for a newly uploaded video to **NotStarted**). For more information about how the repository pattern is used to store information in the CMS database, see "Appendix A – The Contoso Web Service."

The **EncodingState** enumeration is defined in the Contoso.Domain project and has four possible values – **NotStarted**, **Encoding**, **Complete**, and **Error**.

The **ContosoJobNotificationQueue** is polled every 10 seconds to examine the state of the job. This process is managed by the **Run** method in the **WorkerRole** class in the Contoso.EncodingWorker project. When the queue is polled and a **JobNotificationMessage** is received the **Run** method of the **JobNotificationCommand** class is invoked.

C#

```
public bool Run(JobNotificationMessage message)
{
    var encodingJobComplete = new JobNotification()
    {
        EventTypeDescription = message.EventType,
            JobId = (string)message.Properties.Where(j => j.Key ==
            "JobId").FirstOrDefault().Value,
            OldJobStateDescription = (string)message.Properties.Where(j =>
            j.Key == "OldState").FirstOrDefault().Value,
            NewJobStateDescription = (string)message.Properties.Where(j =>
            j.Key == "NewState").FirstOrDefault().Value
    };

    this.encodingService.ProcessJobNotification(encodingJobComplete);

    return true;
}
```

This method converts the **JobNotificationMessage** to a new **JobNotification** instance and then calls the **ProcessJobNotification** method of the **EncodingService** class. After the **Run** method has executed the **JobNotificationMessage** is deleted from the **ContosoJobNotificationQueue**.

The following code example shows the **ProcessJobNotificationMethod** in the **EncodingService** class.

C#

```
public async Task ProcessJobNotification(JobNotification jobNotification)
{
    if (jobNotification.EventTypeDescription != "JobStateChange")
    {
        return;
    }
```

```
    JobState newJobState = (JobState)Enum.Parse(typeof(JobState),
        jobNotification.NewJobStateDescription);

    var job = this.context.Jobs.Where(j =>
              j.Id == jobNotification.JobId).SingleOrDefault();
    if(job == null)
    {
        return;
    }

    switch (newJobState)
    {
        case JobState.Finished:
            await this.ProcessEncodingOutput(job).ConfigureAwait(false);
            break;
        case JobState.Error:
            await this.UpdateEncodingStatus(job, EncodingState.Error)
                .ConfigureAwait(false);
            break;
    }
}
```

When this method is first called the **EventTypeDescription** property of the **JobNotification** instance will be set to **NotificationEndPointRegistration**. Therefore the method will return.

The **JobState** enumeration is defined in the Microsoft.WindowsAzure.MediaServices.Client namespace and has seven possible values – **Queued**, **Scheduled**, **Processing**, **Finished**, **Error**, **Canceled**, and **Canceling**.

When the **JobState** of the encoding **Job** changes, a new **JobNotificationMessage** is added to the **ContosoJobNotificationQueue**. When the queue is polled and the message is received the **Run** method of the **ProcessJobNotification** method of the **EncodingService** class is invoked again. In turn this calls the **ProcessJobNotification** method of the **EncodingService** class again. On this call the **EventTypeDescription** property of the **JobNotification** instance will be set to **JobStateChange**. Therefore the job details are retrieved from the CMS database and the **ProcessEncodingOutput** method will be called, provided that the **JobState** is **Finished**. Alternatively, the **UpdateEncodingStatus** method is called to update the **EncodingState** for the video to **Error**, if an error has occurred during the job processing.

Prior to the encoding job being submitted to the **ContosoJobNotificationQueue** the **NotificationJobState.FinalStatesOnly** was passed to the **AddNew** method to indicate that we are only interested in the final states of the job processing. This avoids the **ProcessJobNotification** method being called for every single **JobState** change.

The following code example shows the **ProcessEncodingOuput** method in the **EncodingService** class.

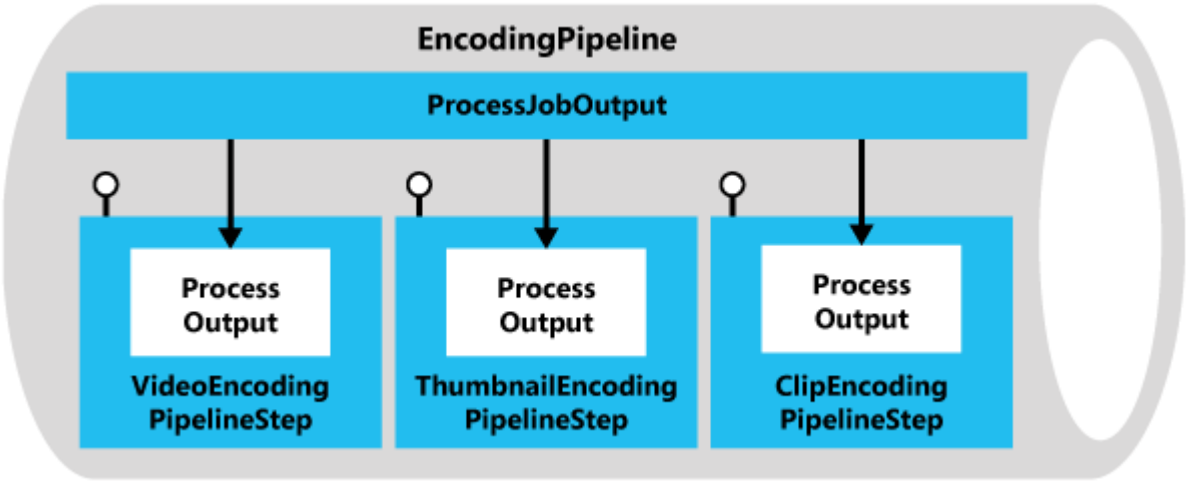
C#

```
private async Task ProcessEncodingOutput(IJob job)
{
    // retrieve the job from the CMS database
    var encodingJob = await this.jobRepository.GetJob(job.Id)
        .ConfigureAwait(false);

    // retrieve the video detail from the CMS database
    var videoDetail = await this.videoRepository.GetVideo(encodingJob.VideoId)
        .ConfigureAwait(false);
    EncodingPipeline pipeline = new EncodingPipeline();

    // process the output from the job
    pipeline.ProcessJobOutput(job, this.context, encodingJob, videoDetail);

    // save the into to the CMS database
    await this.videoRepository.SaveVideo(videoDetail).ConfigureAwait(false);
    await this.jobRepository.SaveJob(encodingJob).ConfigureAwait(false);
}
```

This method retrieves the job and video details from the CMS database, before creating a new instance of the **EncodingPipeline** class in order to process the output assets from the job. Finally, the updated job and video details are saved to the CMS database.

Processing the output assets from the Azure Media Services encoding job

The following figure shows the methods involved in processing output assets from Media Services in each step of the **EncodingPipeline**.



The methods involved in processing output assets in each step of the EncodingPipeline

The **ProcessJobOutput** method in the **EncodingPipeline** class is responsible for processing the encoding job retrieved from the CMS database. The following code example shows this method.

C#

```
public void ProcessJobOutput(IJob job, CloudMediaContext context,
    EncodingJob encodingJob, VideoDetail videoDetail)
{
    ...
    foreach(var task in job.Tasks)
    {
        var encodingTask = encodingJob.EncodingTasks.SingleOrDefault(
            t => t.EncodingTaskUuId == task.Id);
        if(encodingTask != null)
        {
            foreach(var outputAsset in task.OutputAssets)
            {
                encodingTask.AddEncodingAsset(new EncodingAsset()
                    { EncodingAssetUuId = outputAsset.Id, IsInputAsset = false });

                foreach(var step in pipelineSteps)
                {
                    if(outputAsset.Name.Contains(step.Key))
                    {
                        var pipelineStep = (IEncodingPipelineStep)Activator
                            .CreateInstance(step.Value);
                        pipelineStep.ProcessOutput(context, outputAsset,
                            videoDetail);
                    }
                }
            }
        }
    }
    videoDetail.EncodingStatus = EncodingState.Complete;
}
```

Each encoding job contains a number of encoding tasks, with each encoding task potentially resulting in a number of assets being output. Therefore, this code loops through each task in the job and then each output asset in the task, and then each step in the pipeline to perform string matching on the suffix (_EncodingOutput, _ThumbnailOutput, or _ClipOutput) that is appended to each output asset. Then, the **ProcessOutput** method is called on the newly instantiated pipeline step that matches up to the output asset suffix.

Finally, the **EncodingStatus** of the video is set to **Complete**.

Processing the video encoding pipeline step output assets

As previously mentioned, the **VideoEncodingPipelineStep** class is responsible for encoding a video. Therefore, its **ProcessOutput** method, which is shown in the following code example, is responsible for processing the adaptive bitrate encoded video assets.

C#

```
public void ProcessOutput(CloudMediaContext context, IAsset outputAsset,
    VideoDetail videoDetail)
{
    context.Locators.Create(LocatorType.OnDemandOrigin, outputAsset,
```

```
        AccessPermissions.Read, TimeSpan.FromDays(30));
    context.Locators.Create(LocatorType.Sas, outputAsset, AccessPermissions.Read,
        TimeSpan.FromDays(30));

    var mp4AssetFiles = outputAsset.AssetFiles.ToList().Where(
        f => f.Name.EndsWith(".mp4", StringComparison.OrdinalIgnoreCase));
    var xmlAssetFile = outputAsset.AssetFiles.ToList().SingleOrDefault(
        f => f.Name.EndsWith("_manifest.xml",
        StringComparison.OrdinalIgnoreCase));

    Uri smoothStreamingUri = outputAsset.GetSmoothStreamingUri();
    Uri hlsUri = outputAsset.GetHlsUri();
    Uri mpegDashUri = outputAsset.GetMpegDashUri();

    foreach (var mp4Asset in mp4AssetFiles)
    {
        ILocator originLocator = outputAsset.Locators.ToList().Where(
            l => l.Type == LocatorType.OnDemandOrigin).OrderBy(
            l => l.ExpirationDateTime).FirstOrDefault();
        var uri = new Uri(string.Format(CultureInfo.InvariantCulture,
            BaseStreamingUrlTemplate, originLocator.Path.TrimEnd('/'),
            mp4Asset.Name), UriKind.Absolute);

        videoDetail.AddVideo(new VideoPlay()
        {
            EncodingType = "video/mp4",
            Url = uri.OriginalString,
            IsVideoClip = false
        });
    }

    videoDetail.AddVideo(new VideoPlay()
    {
        EncodingType = "application/vnd.ms-sstr+xml",
        Url = smoothStreamingUri.OriginalString,
        IsVideoClip = false
    });
    videoDetail.AddVideo(new VideoPlay()
    {
        EncodingType = "application/vnd.apple.mpegurl",
        Url = hlsUri.OriginalString,
        IsVideoClip = false
    });
    videoDetail.AddVideo(new VideoPlay()
    {
        EncodingType = "application/dash+xml",
        Url = mpegDashUri.OriginalString,
        IsVideoClip = false
    });

    this.ParseManifestXml(xmlAssetFile.GetSasUri().OriginalString, videoDetail);
}
```


The Windows Store and Windows Phone Contoso Video apps both play smooth streaming assets. The Android and iOS Contoso video apps play HLS assets. However, all apps will fall back to playing the first available multi-bitrate MP4 URL if streaming content is unavailable.

This method creates an on-demand origin locator, and a shared access signature locator, to the output asset, with both locators allowing read access for 30 days. The shared access locator provides direct access to a media file in Azure Storage through a URL. The on-demand origin locator provides access to smooth streaming or Apple HLS content on an origin server, through a URL that references a streaming manifest file.


When you create a locator for media content there may be a 30-second delay due to required storage and propagation processes in Azure Storage.

A list of the multi-bitrate MP4 files produced by the encoding job is then created, along with the XML file that references the MP4 collection. Extension methods then generate URIs to smooth streaming, HLS, MPEG-DASH, and MP4 progressive download versions of the output asset, with the smooth streaming, HLS, and MPEG-DASH content being packaged on demand. For more information about packaging see "Dynamic packaging."

Media Services uses the value of the **IAssetFile.Name** property when building URLs for streaming content. Therefore, the value of the Name property cannot have any of the percent-encoding reserved characters (!#$&'()*+,/:;=?@[]). In addition, there must be only one '.' for the filename extension. For more information see "Percent-encoding."

The URIs to the different versions of the content (smooth streaming, HLS, MPEG-DASH, progressive MP4s) are then stored in new **VideoPlay** instances, which also specify the **EncodingType** of the content. The **VideoPlay** instances are then added to the **VideoDetail** object. Through this mechanism encoded content can be played back in client apps across a variety of devices.

Note: The URIs for the encoded assets can be very long. In this guide the URIs have been left unaltered so that you can understand how Media Services functionality works. However, in your own application you may choose to have a mechanism for handling long URIs, such as retrieving a base URI for an asset and then retrieving relative URIs for each asset file.

Processing the thumbnail encoding pipeline step output assets

As previously mentioned, the **ThumbnailEncodingPipelineStep** class is responsible for producing thumbnail images from a video file. Therefore, its **ProcessOutput** method, which is shown in the following code example, is responsible for processing the thumbnail images produced from a video file.

C#

```
public void ProcessOutput(CloudMediaContext context, IAsset outputAsset,
    VideoDetail videoDetail)
{
```

```
    context.Locators.Create(LocatorType.OnDemandOrigin, outputAsset,
        AccessPermissions.Read, TimeSpan.FromDays(30));
    context.Locators.Create(LocatorType.Sas, outputAsset, AccessPermissions.Read,
        TimeSpan.FromDays(30));

    foreach (var assetFile in outputAsset.AssetFiles)
    {
        videoDetail.AddThumbnailUrl(new VideoThumbnail()
            { Url = assetFile.GetSasUri().OriginalString });
    }
}
```

This method creates a shared access signature locator to the output asset that allows read access for 30 days. The shared access locator provides direct access to a media file in Azure storage through a URL. A **VideoThumbnail** instance is then created for each thumbnail image that uses the shared access signature locator to specify a URL to a thumbnail image, and is added to the **VideoDetail** instance.

> **Note:** The URIs for thumbnails can be very long. In this guide the URIs have been left unaltered so that you can understand how Media Services functionality works. However, in your own application you may choose to have a mechanism for handling long URIs, such as retrieving a base URI for an asset and then retrieving relative URIs for each asset file.

Processing the clip encoding pipeline step output assets

As previously mentioned, the **ClipEncodingPipelineStep** class is responsible for producing a clip from the video being encoded. Therefore, its **ProcessOutput** method, which is shown in the following code example, is responsible for processing the clip produced from a video file.

C#

```
public void ProcessOutput(CloudMediaContext context, IAsset outputAsset,
    VideoDetail videoDetail)
{
    context.Locators.Create(LocatorType.OnDemandOrigin, outputAsset,
        AccessPermissions.Read, TimeSpan.FromDays(30));
    context.Locators.Create(LocatorType.Sas, outputAsset, AccessPermissions.Read,
        TimeSpan.FromDays(30));

    var mp4AssetFiles = outputAsset.AssetFiles.ToList().Where(
        f => f.Name.EndsWith(".mp4", StringComparison.OrdinalIgnoreCase));
    List<Uri> mp4ProgressiveDownloadUris = mp4AssetFiles.Select(
        f => f.GetSasUri()).ToList();

    mp4ProgressiveDownloadUris.ForEach(v => videoDetail.AddVideo(new VideoPlay()
    {
        EncodingType = "video/mp4",
        Url = v.OriginalString,
        IsVideoClip = true
    }));
}
```

This method creates a shared access signature locator to the output asset that allows read access for 30 days. The shared access locator provides direct access to a media file in Azure Storage through a URL. The collection of multi-bitrate MP4 clips produced by the encoding job are then retrieved, before URIs to each file are generated by the **GetSasUri** extension method. Finally, the URIs to each MP4 file are stored in separate **VideoPlay** instances, which also specifies the **EncodingType** of the content. The **VideoPlay** instances are then added to the **VideoDetail** object.

> **Note:** The URIs to the encoded assets can be very long. In this guide the URIs have been left unaltered so that you can understand how Media Services functionality works. However, in your own application you may choose to have a mechanism for handling long URIs, such as retrieving a base URI for an asset and then retrieving relative URIs for each asset file.

Summary

This chapter has described how the Contoso developers incorporated Media Services' encoding and processing functionality into their web service. It summarized the decisions that they made in order to support their business requirements, and how they designed the code that performs the encoding process.

In this chapter, you saw how to use the Azure Media Encoder to encode uploaded media for delivery. The chapter also discussed how to scale encoding jobs by reserving encoding units that allow you to have multiple encoding tasks running concurrently.

The following chapter discusses the final step in the Media Services workflow – delivering and consuming media.

More information

- For information about additional support codecs and filters in Media Services, see "Codec Objects" and "DirectShow Filters" on MSDN.
- For information about how to configure the Azure Media Packager, see "Task Preset for Azure Media Packager" on MSDN.
- You can download the "IIS Smooth Streaming Technical Overview" paper from the Microsoft Download Center.
- The page, "Azure Management Portal" explains tasks that can be accomplished by using the Azure Management Portal, and is available on MSDN.
- You can find the article "How to Scale a Media Service" on MSDN.
- The article "Azure Queues and Azure Service Bus Queues – Compared and Contrasted" is available on MSDN.
- For information about customizing the settings of a thumbnail file see "Task Preset for Thumbnail Generation" on MSDN.
- The page, "Percent-encoding" explains the mechanism for encoding URI information, and is available on Wikipedia.

5 - Delivering and Consuming Media from Microsoft Azure Media Services

Microsoft Azure Media Services content can be delivered in numerous application scenarios. Content can be downloaded, or directly accessed by using locator URLs. In addition, content can be sent to another application or to another content provider. Content to be delivered can include media assets that are simply stored in Media Services, or media assets that have been encoded and processed in different ways.

This chapter describes how the Contoso developers incorporated Media Services' delivery and consumption functionality into their web service and Windows Store client application. It summarizes the decisions that they made in order to support their business requirements, and how they designed the code that requests and consumes encoded media.

Delivering media from Azure Media Services

Media Services can be used to deliver content that has been stored in Media Services, and it can also deliver content that has been processed in different ways with the results stored in Media Services.

There are a number of approaches that can be used to deliver Media Services content:

- Direct download from Azure Storage.
- Access media in Azure Storage.
- Stream media to a client application.
- Send media content to another application or to another content provider.

Media content can be downloaded directly from Azure Storage as long as you have the Media Services credentials for the account that uploaded and encoded the asset. Downloading may be necessary for archival purposes or for further processing outside the Media Services environment. Content can be downloaded by creating a shared access signature locator, which contains a URL to the asset that contains the requested file. The URL can then be used by anyone to download the asset. When downloading a storage encrypted file with the Media Services SDKs, the encrypted asset is automatically decrypted after download. For more information about downloading a file from storage see "Create a SAS Locator to On Demand Content."

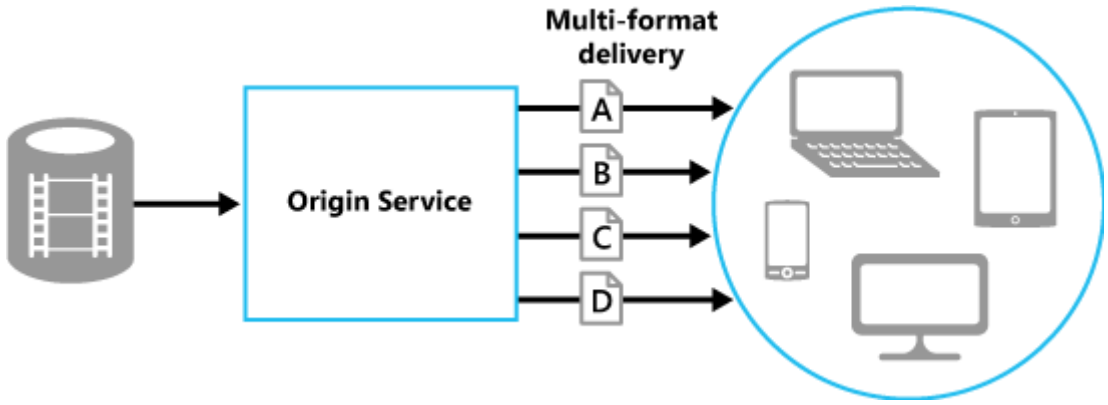
> A download that has not completed within 12 hours will fail.

You may want to give users access to content stored in Azure Storage. To do this you must create a full shared access signature URL to each file contained in the media asset. This is achieved by appending the file name to the shared access signature locator. For more information see "Processing the output assets from the encoding job."

Media Services also provides a way to directly access streaming media content. This can be accomplished by creating an on-demand origin locator, which allows direct access to Smooth Streaming or Apple HTTP Live Streaming (HLS) content. With on-demand origin locators, you build a

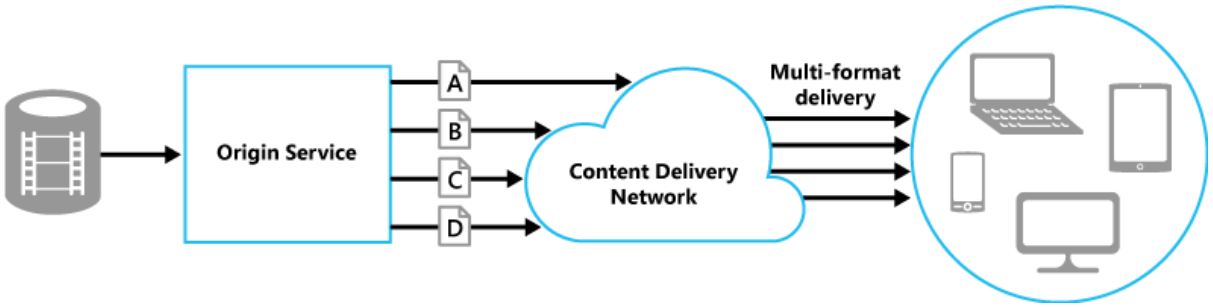
full URL to a streaming manifest file in an asset. You can then provide the URL to a client application that can play streaming content.

When streaming media using on-demand origin locators you can take advantage of dynamic packaging. When using dynamic packaging, your video is stored in one encoded format, usually an adaptive bitrate MP4 file set. When a video player requests the video it specifies the format it requires, and the Origin Service converts the MP4 file to the format requested by the player. This allows you to store only one format of your videos, therefore reducing storage costs. The following figure illustrates dynamic packaging in action, whereby a video stored in one encoded format is converted by the Origin Service into multiple formats, as requested by the client applications.



A video is converted into multiple formats on-demand

Content can also be delivered by using a Content Delivery Network (CDN), in order to offer improved performance and scalability when streaming media with Origin Services. For more information see "How to Manage Origins in a Media Services Account." The following figure illustrates a content delivery network delivering video to multiple client applications after being converted into multiple formats by the Origin Service.



A video is converted into multiple formats on-demand and delivered through a CDN

The Contoso web service is responsible for creating locators to give access to media assets, during the encoding/processing phase. The full URLs for media assets are then stored in the CMS database. Asset playback then simply involves retrieving the URL from the database and passing it to the appropriate control in the client application.

Azure Media Services Origin Service

The Origin Service handles requests for content by processing the outbound stream from storage to content delivery network or client application. It contains a content server which pulls the content

from storage and delivers it, caching it as required in order to reduce the load on delivery channels and storage. It also provides content encryption and decryption in order to keep content protected. In addition it provides the ability to use dynamic packaging by storing media in one multi-bitrate format and converting it to the format requested by the client application in real-time.

Each Media Service account has at least one streaming origin called **default** associated with it. Media Services enables you to add multiple streaming origins to your account and to configure the origins. Examples of configurations include:

- Setting the maximum caching period that will be specified in the cache control header of HTTP responses.
- Specifying IP addresses that are allowed to connect to the published streaming endpoint.
- Specifying configuration for g2o authentication requests from Akamai servers.

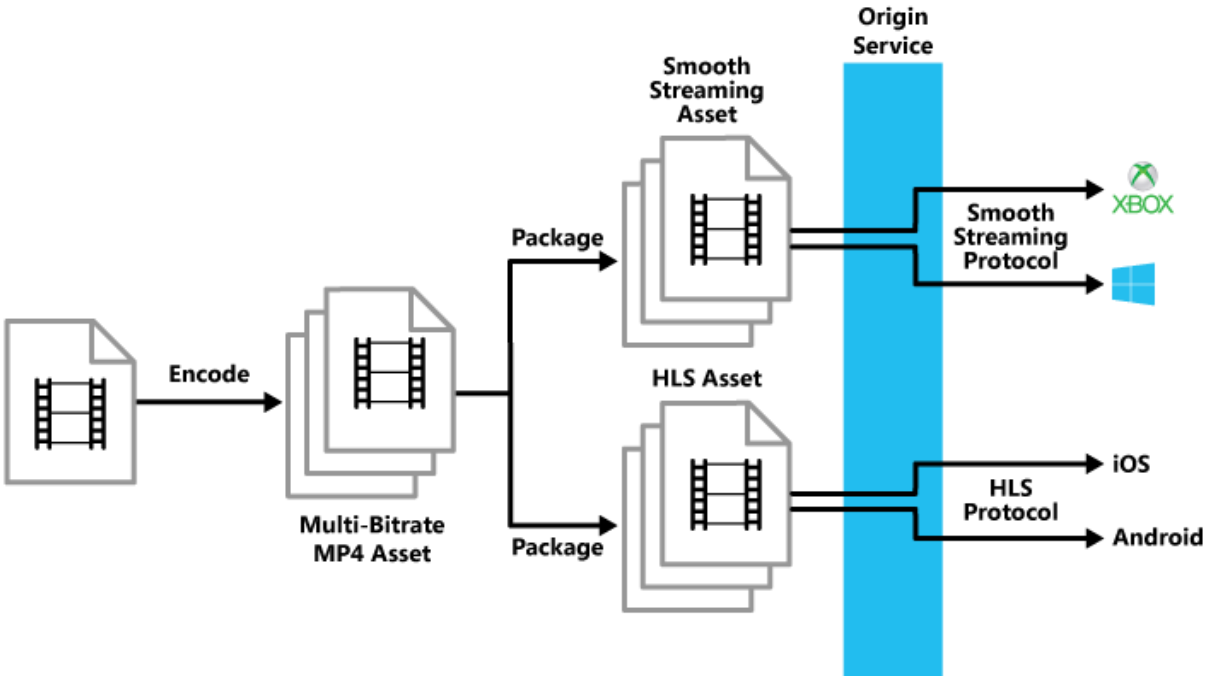
Akamai servers are a scalable network of servers around the world that make up a content delivery network. Server capacity is rented to customers who want their content to be delivered quickly from locations close to the user. When a user navigates to the URL of content belonging to an Akamai customer, they are transparently redirected to one of Akamai's copies of the content. Users then receive content from an Akamai server that is close to them, or which has a good connection, leading to faster download times.

For more information see "How to Manage Origins in a Media Services Account."

Azure Media Services dynamic packaging

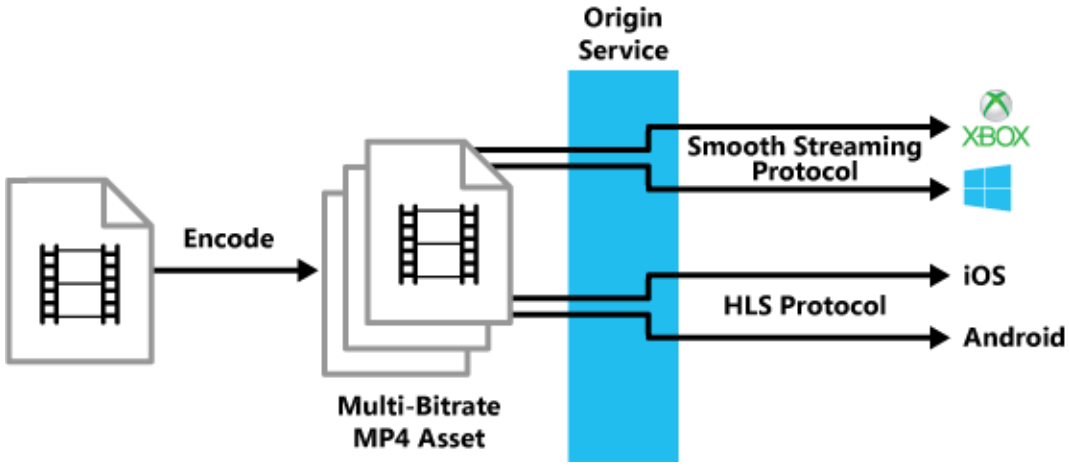
When a video has been encoded it can be placed into a variety of file containers. This process is referred to as *packaging*. For example, you could convert an MP4 file into smooth streaming content by using the Azure Media Packager to place the encoded content into a different file container.

The Azure Media Packager is capable of performing *static* packaging and *dynamic* packaging. Static packaging involves creating a copy of your content in each format required by users. For example, an asset could be converted into smooth streaming content and HLS content if both formats are required by users. This would result in multiple copies of the content existing, in different formats. The following figure shows an example of an encoding and packaging workflow that uses static packaging.



An encoding and packaging workflow that uses static packaging

Dynamic packaging enables video packaging to be performed when a client application requests a specific video format, allowing the video to be encoded just once, with it being converted in real-time to the format requested by the client. With dynamic packaging your video is typically stored as an adaptive bitrate MP4 file set. When a client application requests the video it specifies the required format. The Origin Service then converts the MP4 adaptive bitrate file to the format requested by the client in real-time. This ensures that only one format of your video has to be stored, therefore reducing the storage costs. The following figure shows an example of an encoding and packaging workflow that uses dynamic packaging.



An encoding and packaging workflow that uses dynamic packaging

To use dynamic packaging you must create an asset that contains a set of multi-bitrate MP4 files or multi-bitrate smooth streaming files. The on-demand streaming server will then ensure that clients receive the stream in the requested protocol, based on the specified format in the manifest.

Dynamic packaging requires that you purchase on-demand streaming reserved units. For more information see "Dynamic packaging."


Dynamic packaging only supports unencrypted source files and produces unencrypted output streams.

The following source file formats are not supported by dynamic packaging:

- Source files containing the following codecs:
  - Dolby digital in MP4 files.
  - Dolby digital in Smooth Streaming files.
- Protected content:
  - Storage encrypted content.
  - PlayReady protected Smooth Streaming content.
  - AES-128 bit CBC protected HLS content.
- HLS content:
  - HLS v4.
  - PlayReady protected HLS.
  - IIS MS HLS archives.
  - IIS MS HLS presentations from transform manager or the Media Services packager.
  - Segmented HLS.

Scaling Azure Media Services delivery

You can scale Media Services delivery by specifying the number of on-demand streaming reserved units that you would like your account to be provisioned with.

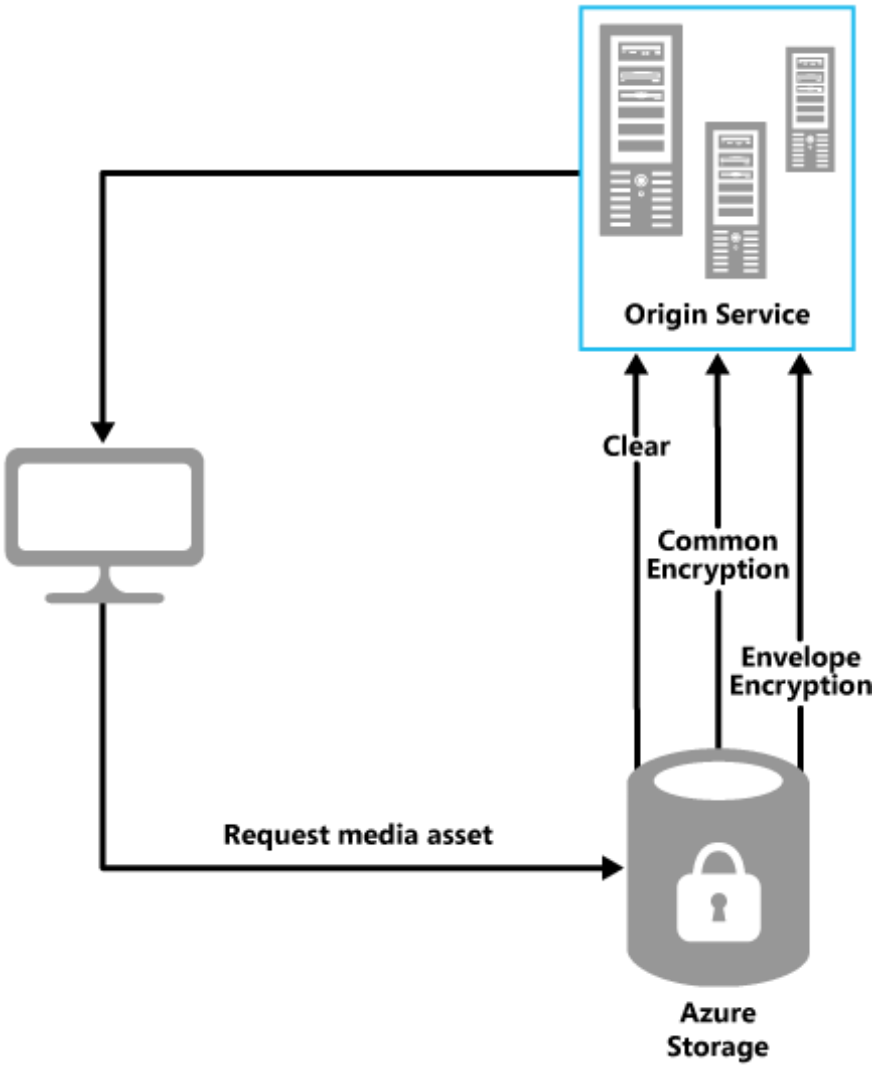
By default on-demand streaming is configured in a shared-instance mode in which server resources are shared with all users. On-demand streaming reserved units can be purchased to improve on-demand streaming throughput, with the units being purchased in increments of 200 Mbps. The allocation of any new on-demanding streaming reserved units takes around 20 minutes to complete.

The number of on-demand streaming reserved units can be configured on the Scale page of the Azure Management Portal.

By default every Media Services account can scale to up to 5 on-demand streaming reserved units. A higher limit can be requested by opening a support ticket. For more information about opening a support ticket see "Requesting Additional Reserved Units."

Securely delivering streaming content from Azure Media Services

The following figure summarizes how Media Services can deliver adaptive bitrate media assets using a number of techniques.



Options for delivering adaptive bitrate media assets

Media assets can be delivered in the clear, or securely by using common or envelope encryption. Each technique will now be discussed in turn.

Progressive download of storage encrypted content

Progressive download allows you to start playing media before the entire file has been downloaded, and is only supported with ISO standard MP4 files. To use progressive download you must create an on-demand locator and point your client application at the full URL of the MP4 file to play. However, on-demand locators do not support dynamic decryption of storage encrypted assets. Therefore, you must decrypt any storage encrypted assets that you wish to stream from the Origin Service for progressive download.

For more info see "Create a SAS Locator to On Demand Content."

Smooth Streaming content and MPEG-DASH

Smooth Streaming assets can be protected using common encryption and PlayReady DRM. Common encryption protects a media stream during storage and download by using Advanced Encryption Standard (AES) 128-bit elementary stream encryption, which provides content protection through to a secure decoder. PlayReady DRM protects a media stream during playback by using a license server that protects the decryption key required to decrypt the stream. Client apps that use PlayReady must provide a secure and robust playback environment that meets the compliance rules for PlayReady. When a client application attempts to access a PlayReady protected asset it must pass the player ID and device information to a license server. The licensing server will then determine if the user has permission to access the stream and if the device is trusted to decrypt the stream. For more info see "Microsoft PlayReady."

> **Note:** Microsoft does not provide a license delivery service for PlayReady as part of Media Services. You must implement your own or use a third-party provider.

To use dynamic packaging to deliver MPEG-DASH encrypted with PlayReady DRM you are required to convert your video into smooth streaming format first, and then protect it with PlayReady DRM. For more info see "Task Preset for Azure Media Encryptor."

Apple HLS content

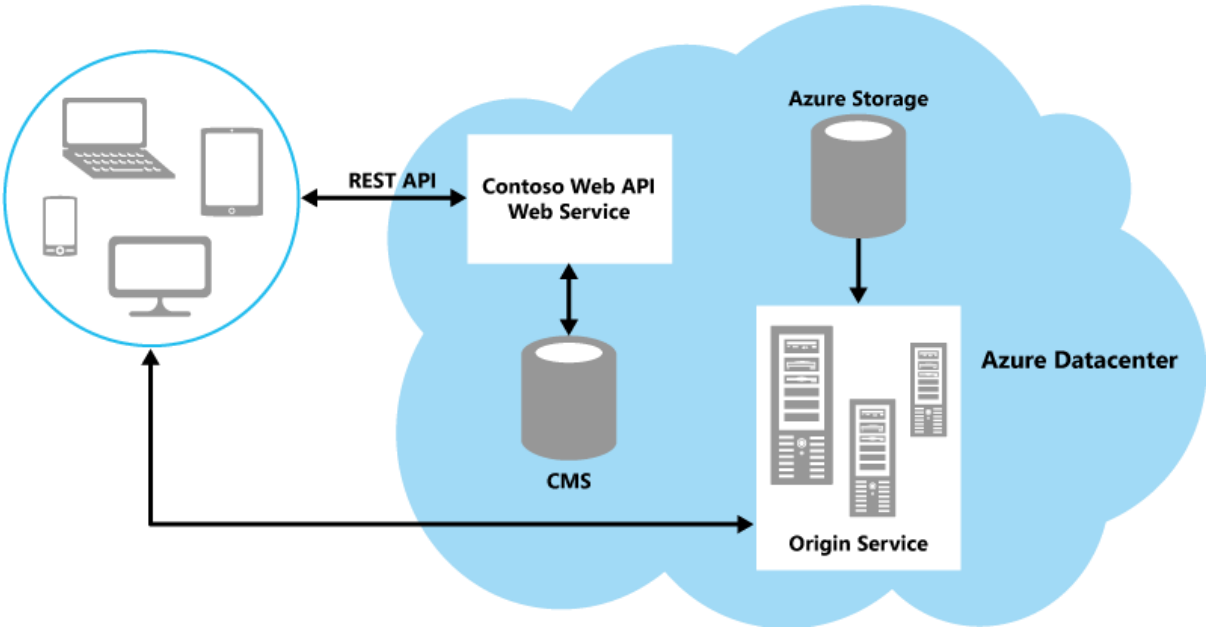
Media Services supports delivering HLS assets protected by an envelope encryption method such as AES-128 transport stream encryption. Transport stream encrypted media must be decrypted prior to media processing. Therefore, media and keys are processed unencrypted inside client apps that do not have to establish trust or guarantee protection of keys and content. Content protected using this approach is less secure than content protected with a DRM technology.

Apple HLS content with PlayReady

Media Services supports delivering HLS assets protected with PlayReady by first creating a Smooth Streaming asset, protecting it with PlayReady, and then using the Azure Media Packager to convert the resulting asset to a HLS asset protected with the Apple HLS key delivery protocol.

Delivery and consumption process in the Contoso Azure Media Services applications

The following figure shows a high-level overview of how the Contoso video client applications consume media assets stored in Azure Storage.



A high-level overview of the Contoso delivery process

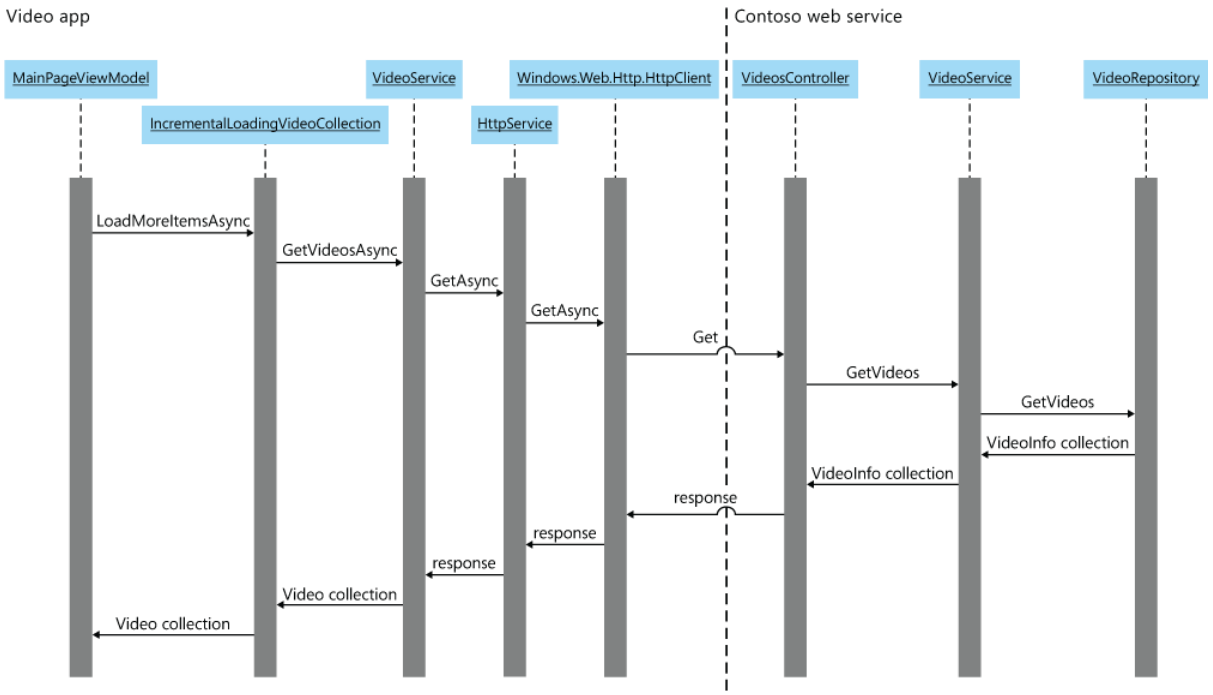
Client apps request a video through a REST web interface. The Contoso web service queries the CMS which returns the URL of the media asset in Azure Storage. The media asset could be a single video file, or a manifest file which references multiple video files. The application then requests the URL content from the Media Services Origin Service, which processes the outbound stream from storage to client app.

The Contoso web service does not provide a mechanism for removing inappropriate content and the users who have uploaded it. In your own application you should implement a user feedback mechanism for notifying the service administrators of inappropriate content. You should also implement functionality that enables service administrators to remove inappropriate content, and suspend or delete any users who have uploaded such content.

The client apps consume content by allowing users to browse videos, play videos, and retrieve recommendations for other videos they may want to watch. Each item will now be discussed in turn.

Browsing videos

The following figure shows the interaction of the classes in the Contoso Windows Store Video application that implement retrieving and displaying thumbnails of the available videos, for the user to browse through.



The interaction of classes that implement browsing videos

> For information on how browsing videos works in the Contoso video web application, see "Appendix C – Understanding the Contoso Video Applications."

The **MainPage** in the application allows the user to browse through thumbnails of the available videos, and select one for viewing. The video thumbnails are displayed in an **AutoRotatingGridView** custom control, with the **ItemTemplate** using an **Image** control to display each thumbnail.

XAML

```
<controls:AutoRotatingGridView
    ...=""
    ItemsSource="{Binding Videos}"
    SelectionMode="None"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    IsItemClickEnabled="True">
  <interactivity:Interaction.Behaviors>
    <core:EventTriggerBehavior EventName="ItemClick">
      <behaviors:NavigateWithEventArgsToPageAction
          TargetPage="Contoso.WindowsStore.Views.VideoDetailPage"
          EventArgsParameterPath="ClickedItem.Id" />
    </core:EventTriggerBehavior>
  </interactivity:Interaction.Behaviors>
  <controls:AutoRotatingGridView.ItemTemplate>
    <DataTemplate>
      ...
      <Image Stretch="Uniform" >
        <Image.Source>
          <BitmapImage UriSource="{Binding ThumbnailUrl}" />
        </Image.Source>
      </Image>
      ...
    </DataTemplate>
```

```
  </controls:AutoRotatingGridView.ItemTemplate>
  ...
</controls:AutoRotatingGridView>
```

> **Note:** The **AutoRotatingGridView** custom control is a view state detecting **GridView** control. When, for example, the view state changes from *DefaultLayout* to *PortraitLayout* the items displayed by the control will be automatically rearranged to use an appropriate layout for the view state. The advantage of this approach is that only one control is required to handle all the view states, rather than having to define multiple controls to handle the different view states.

The **AutoRotatingGridView** custom control binds to the **Videos** collection in the **MainPageViewModel** class, through the **ItemsSource** property. A custom Blend interaction named **NavigateWithEventArgsToPageAction** is used to invoke navigation to the **VideoDetailPage**. The **EventTriggerBehavior** binds the **ItemClick** event of the **AutoRotatingGridView** custom control to the **NavigateWithEventArgsToPageAction**. So when a **GridViewItem** is selected the **NavigateWithEventArgsToPageAction** is executed, which navigates from the **MainPage** to the **VideoDetailPage**, passing in the **Id** of the **ClickedItem** to the **VideoDetailPage**.

When a page is navigated to the **OnNavigatedTo** method in the page's view model is called. The **OnNavigatedTo** method allows the newly displayed page to initialize itself by loading any page state, and by using any navigation parameters passed to it. The following code example shows how the **OnNavigatedTo** method in the **MainPageViewModel** class populates the **Videos** collection with thumbnails to display to the user.

C#

```
public async override void OnNavigatedTo(object navigationParameter,
    string navigationMode, Dictionary<string, object> viewModelState)
{
    ...
    this.Videos = new IncrementalLoadingVideoCollection(this.videoService);
    // We retrieve the first 20 videos from the server.
    // The GridView will automatically request the rest of the items to
    // the collection when needed.
    await this.Videos.LoadMoreItemsAsync(20);
    ...
}
```

The **Videos** property is of type **IncrementalLoadingVideoCollection**, and so this code creates a new instance of that type before calling the **LoadMoreItemsAsync** method to retrieve twenty video thumbnail URLs from the Contoso web service. In turn, the **LoadMoreItemsAsync** method calls the **GetVideosAsync** method of the **VideoService** class.

The Contoso Video application supports incremental loading of thumbnail data on the main page in order to consume the paging functionality offered by the Contoso web service.

C#

```
public async Task<ObservableCollection<Video>> GetVideosAsync(int pageIndex,
    int pageSize)
```

```
{
    var requestUri = new Uri(string.Format(this.paginationQueryString,
        this.videosBaseUrl, pageIndex, pageSize));
    var responseContent = await this.httpService.GetAsync(requestUri);

    var videos =
        JsonConvert.DeserializeObject<ReadOnlyCollection<Video>>(responseContent);

    return new ObservableCollection<Video>(videos);
}
```

This method creates a URI that specifies that the **Get** method will be called on the web service, with the paging options being passed as a parameter. The **HttpService** class, which implements the **IHttpService** interface, is used to make the call to the web service.

When the **GetVideosAsync** method calls **HttpService.GetAsync**, this calls the **Get** method in the **VideosController** class in the Contoso.Api project. For more information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

C#

```
public async Task<HttpResponseMessage> Get(int pageIndex = 1, int pageSize = 10)
{
    ...
    var videoInfos = await videoService.GetEncodedVideos(pageIndex, pageSize);
    var count = await videoService.GetEncodedVideosCount();

    var result = new List<VideoInfoDTO>();

    Mapper.Map(videoInfos, result);

    var response = Request.CreateResponse(HttpStatusCode.OK, result);

    ...
    return response;
}
```

This method calls the **GetEncodedVideos** method in the **VideoService** class. The **VideoService** class is registered as a type mapping against the **IVideoService** interface with the Unity dependency injection container. When the **VideoController** constructor accepts an **IVideoService** type, the Unity container will resolve the type and return an instance of the **VideoService** class.

When a collection of **VideoInfo** objects is returned from the **GetEncodedVideos** method a collection of **VideoInfoDTO** objects is created, with the returned **VideoInfo** objects being mapped onto the **VideoInfoDTO** objects, which are then returned in an **HttpResponseMessage** to the **HttpService.GetAsync** method.

The **GetEncodedVideos** method in the **VideoService** class simply calls the **GetEncodedVideos** method in the **VideoRepository** class, and returns the results as a collection of **VideoInfo** objects. The following code example shows the **GetEncodedVideos** method in the **VideoRepository** class.

C#

```
public async Task<ICollection<VideoInfo>> GetEncodedVideos(int pageIndex,
```

```
    int pageSize)
{
    using(VideoContext context = new VideoContext(this.NameOrConnectionString))
    {
        var videos = await context.Videos.Include(v => v.Thumbnails)
            .Where(v => v.EncodingStatus == (int)EncodingState.Complete)
            .OrderByDescending(v => v.LastUpdateDateTime)
            .Skip((pageIndex -1) * pageSize)
            .Take(pageSize)
            .AsNoTracking()
            .ToListAsync()
            .ConfigureAwait(false);

        var result = new List<VideoInfo>();

        foreach(var video in videos)
        {
            var videoInfo = new VideoInfo()
            {
                Id = video.Id,
                Title = video.Title,
                Length = video.Length,
            };

            video.Thumbnails.ToList().ForEach(t => videoInfo.AddThumbnailUrl(
                new VideoThumbnail() { Id = t.Id, Url = t.ThumbnailUrl }));

            result.Add(videoInfo);
        }
        return result;
    }
}
```

Repository objects retrieve data from a context object by performing LINQ queries. Here, a LINQ query retrieves data from the **Video** table and its related tables, for any videos that match the paging parameters passed to the web service. **VideoInfo** objects are created for the data retrieved from the database, with the thumbnail data being added to the **Thumbnails** property of **VideoInfo** objects as **VideoThumbnail** objects. The collection of **VideoInfo** objects is then returned to the calling method.

For more information about using the Repository pattern to access data, see "Appendix A – The Contoso Web Service."

Playing videos

In the Contoso Video application video playback is achieved by using the Microsoft Player Framework, which is an open source video player. The framework allows you to easily add advanced video playback features to Windows Store apps. The player framework supports features such as:

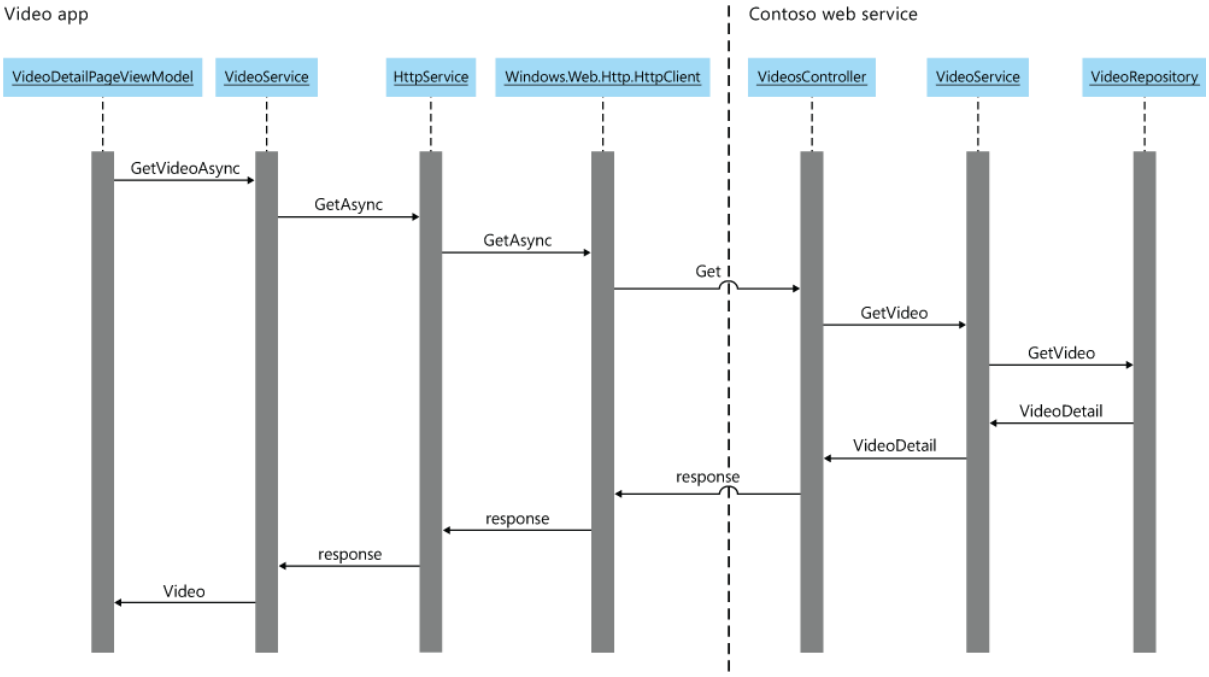
- Adaptive streaming and playback heuristics.
- Closed captioning support.

- DVR-style playback.
- Skinning and styling support.

In turn, the Microsoft Player Framework is built on top of the Smooth Streaming Client SDK, which allows developers to create rich smooth streaming experiences. The SDK is composed of APIs that provide support for simple operations such as play, pause, and stop and also for more complex operations such as selecting and tracking bitrates for smooth streaming playback. For more information, see "Player Framework by Microsoft" and "Smooth Streaming Client SDK."

> Closed captions and subtitles increase the accessibility of your videos. While this information can be embedded inside the video file itself, it is recommended that the data be stored in external files (often referred to as “sidecar” files). The Microsoft Player Framework used in the Contoso video application has support for captions and subtitles in both Timed Text Markup Language (TTML) and the Web Video Text Tracks (WebVTT) formats, while the HTML5 <video> tag currently supports only the WebVTT format. For an example of how to use the Microsoft Player Framework to display captions, see WebVTT Closed caption and subtitling support.

The following figure shows the interaction of the classes in the Contoso Windows Store Video application that implement retrieving and displaying a video for playback by the user.



The interaction of classes that implement retrieving and displaying a video for playback

> For information on how playing videos works in the Contoso video web application, see "Appendix C – Understanding the Contoso Video Applications."

The **VideoDetailPage** in the application allows the user to view a video, while also listing video details and other recommended videos. The video is displayed in a **MediaPlayer** control, from the Player Framework, with the **Source** property of the control binding to the **PlaybackUrl** property in the **VideoDetailPageViewModel** class, which is of type **string**.

XAML

```
<mmppf:MediaPlayer
    ...=""
    Source="{Binding PlaybackUrl}"
    AutoPlay="False"
    IsFullScreenVisible="True"
    IsFullScreenChanged="player_IsFullScreenChanged"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Top">
  <mmppf:MediaPlayer.Plugins>
    <adaptive:AdaptivePlugin />
  </mmppf:MediaPlayer.Plugins>
</mmppf:MediaPlayer>
```

When a page is navigated to the **OnNavigatedTo** method in the page's view model is called. The **OnNavigatedTo** method allows the newly displayed page to initialize itself by loading any page state, and by using any navigation parameters passed to it. The following code example shows how the **OnNavigatedTo** method in the **VideoDetailPageViewModel** class populates the **Video** property with details of the selected video.

C#

```
public async override void OnNavigatedTo(object navigationParameter,
    string navigationModeString, Dictionary<string, object> viewModelState)
{
    ...
    int videoId;

    if (navigationParameter != null &&
        int.TryParse(navigationParameter.ToString(), out videoId))
    {
        Video selectedVideo = await this.videoService.GetVideoAsync(videoId);
        this.Video = selectedVideo;
        ...
    }
    ...
}
```

On the **MainPage** a custom Blend interaction is used to invoke navigation to the **VideoDetailPage**, passing in the ID of the selected video as a parameter. This parameter is parsed and used as a parameter to the **GetVideoAsync** method in the **VideoService** class, in order to retrieve the details for a specific video. When these details are retrieved the **Video** property is set and property change notification updates the **PlaybackUrl** property. The **PlaybackUrl** property is shown in the following code example.

C#

```
public string PlaybackUrl
{
    get
    {
        if (this.Video == null)
        {
            return null;
        }
```

```
        if (this.Video.Videos.Any(x => x.EncodingType ==
            Constants.SmoothStreamingEncodingType))
        {
            return this.Video.Videos.First(x => x.EncodingType ==
                Constants.SmoothStreamingEncodingType).Url;
        }

        return this.video.Videos.Any(x => x.EncodingType == Constants.Mp4) ?
            this.Video.Videos.First(x => x.EncodingType == Constants.Mp4).Url :
            null;
    }
}
```

This property uses a LINQ query to return the URL of the video to be played, with the returned URL being dependent on the encoding type of the retrieved video.

> The Windows Store Contoso Video application prioritizes playing Smooth Streaming assets. The Contoso web service encodes assets to multi-bitrate MP4 files, which are then converted to Smooth Streaming, Apple HLS, or MPEG-DASH on-demand by dynamic packaging. Therefore, there will always be multiple Smooth Streaming assets available through one Smooth Streaming URL, which is the address of the manifest for the multi-bitrate Smooth Streaming assets. However, the **PlaybackUrl** property demonstrates good practice by falling back to using the first available multi-bitrate MP4 URL if a Smooth Streaming URL is unavailable.

The Windows Store and Windows Phone Contoso Video apps both play Smooth Streaming assets. However, the Android and iOS Contoso video apps play HLS assets.

As previously explained, the **OnNavigatedTo** method of the **VideoDetailPageViewModel** class calls the **GetVideoAsync** method in the **VideoService** class, in order to retrieve the details for a specific video. The following code example shows this method.

C#

```
public async Task<Video> GetVideoAsync(int videoId)
{
    var requestUri =
        new Uri(string.Format("{0}?id={1}", this.videosBaseUrl, videoId));
    var responseContent = await this.httpService.GetAsync(requestUri);

    var video = JsonConvert.DeserializeObject<Video>(responseContent);
    return video;
}
```

This method creates a URI that specifies that the **Get** method will be called on the web service, with the video ID being passed as a parameter. The **HttpService** class, which implements the **IHttpService** interface, is used to make the call to the web service.

When the **GetVideoAsync** method calls **HttpService.GetAsync**, this calls the **Get** method in the **VideosController** class in the Contoso.Api project. For more information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

C#

```
public async Task<HttpResponseMessage> Get(int id)
{
    ...
    var videoDetail = await this.videoService.GetVideo(id);
    if(videoDetail.Id != id)
    {
        return Request.CreateResponse(HttpStatusCode.NotFound);
    }

    var result = new VideoDetailDTO();

    Mapper.Map(videoDetail, result);

    return Request.CreateResponse(HttpStatusCode.OK, result);
    ...
}
```

This method calls the **GetVideo** method in the **VideoService** class. The **VideoService** class is registered as a type mapping against the **IVideoService** interface with the Unity dependency injection container. When the **VideoController** constructor accepts an **IVideoService** type, the Unity container will resolve the type and return an instance of the **VideoService** class.

When a **VideoDetail** object is returned from the **GetVideo** method a **VideoDetailDTO** object is created, with the returned **VideoDetail** object being mapped onto the **VideoDetailDTO** object, which is then returned in a **HttpResponseMessage** to the **HttpService.GetAsync** method. If the ID of the retrieved **VideoDetail** object doesn't match the ID that was passed into the **Get** method, a not found message is returned in the **HttpResponseMessage** object.

The **GetVideo** method in the **VideoService** class simply calls the **GetVideo** method in the **VideoRepository** class, and returns the results as a **VideoDetail** object. The following code example shows the **GetVideo** method in the **VideoRepository** class.

C#

```
public async Task<VideoDetail> GetVideo(int id)
{
    using(VideoContext context = new VideoContext(this.NameOrConnectionString))
    {
        var video = await context.Videos
            .AsNoTracking()
            .SingleOrDefaultAsync(v => v.Id == id)
            .ConfigureAwait(false);

        if (video != null)
        {
            // this map is going to force the lazy load of child collections;
            // this is to reduce the payload caused by eager loading the
            // child objects
```

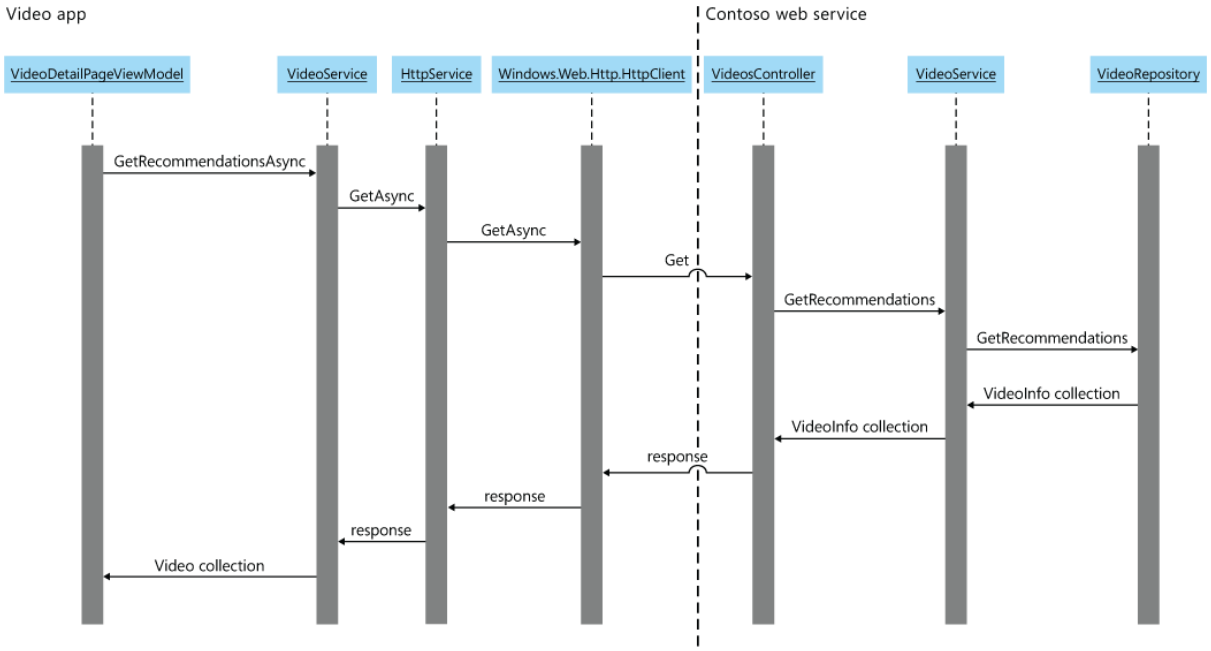
```
            return MapVideoDetail(video);
        }
        else
        {
            return new VideoDetail();
        }
    }
}
```

Repository objects retrieve data from a context object by performing LINQ queries. Here, a LINQ query retrieves data from the **Video** table and its related tables, for the first video that matches the ID passed into the **GetVideo** method. A **VideoDetail** object is then created, populated with the data retrieved from the database, and returned, if a match was found.

For more information about using the Repository pattern to access data, see "Appendix A – The Contoso Web Service."

Retrieving recommendations

As well as allowing the user to view a video, the **VideoDetailPage** also lists video details and other recommended videos. The following figure shows the interaction of the classes in the Contoso Windows Store Video application that implement retrieving and displaying video recommendations for the user.



The interaction of the classes that retrieve recommendations

> For information on how retrieving recommendations works in the Contoso video web application, see "Appendix C – Understanding the Contoso Video Applications."

The **VideoDetailPage** displays video recommendations in a **AutoRotatingGridView** custom control, with the **ItemTemplate** using an **Image** control to display a thumbnail for each recommendation.

XAML

```
<controls:AutoRotatingGridView
    ...=""
    ItemsSource="{Binding RelatedVideos}"
    SelectionMode="None"
    ScrollViewer.IsHorizontalScrollChainingEnabled="False"
    IsItemClickEnabled="True">
  <interactivity:Interaction.Behaviors>
    <core:EventTriggerBehavior EventName="ItemClick">
      <behaviors:NavigateWithEventArgsToPageAction
          TargetPage="Contoso.WindowsStore.Views.VideoDetailPage"
          EventArgsParameterPath="ClickedItem.Id" />
    </core:EventTriggerBehavior>
  </interactivity:Interaction.Behaviors>
  <GridView.ItemTemplate>
    <DataTemplate>
      ...
      <Image Stretch="Uniform" >
        <Image.Source>
          <BitmapImage UriSource="{Binding ThumbnailUrl}" />
        </Image.Source>
      </Image>
      ...
    </DataTemplate>
  </GridView.ItemTemplate>
</controls:AutoRotatingGridView>
```

The **AutoRotatingGridView** custom control binds to the **RelatedVideos** collection in the **VideoDetailPageViewModel** class, through the **ItemsSource** property. A custom Blend interaction named **NavigateWithEventArgsToPageAction** is used to invoke navigation to the **VideoDetailPage**. The **EventTriggerBehavior** binds the **ItemClick** event of the **AutoRotatingGridView** custom control to the **NavigateWithEventArgsToPageAction**. So when a **GridViewItem** is selected the **NavigateWithEventArgsToPageAction** is executed, which navigates from the **VideoDetailPage** to the **VideoDetailPage**, passing in the ID of the **ClickedItem** to the **VideoDetailPage**. The overall effect is to reload the page with the video selected from the recommendations being the video available for playback.

When a page is navigated to the **OnNavigatedTo** method in the page's view model is called. The **OnNavigatedTo** method allows the newly displayed page to initialize itself by loading any page state, and by using any navigation parameters passed to it. The following code example shows how the **OnNavigatedTo** method in the **VideoDetailPageViewModel** class populates the **RelatedVideos** property with details of the recommended videos.

C#

```
public async override void OnNavigatedTo(object navigationParameter,
    string navigationModeString, Dictionary<string, object> viewModelState)
{
    ...
    ReadOnlyCollection<Video> videos = await
                        this.videoService.GetRecommendationsAsync(this.Video.Id);

    this.RelatedVideos = videos;
    ...
}
```

This method calls the **GetRecommendationsAsync** method of the **VideoService** class to retrieve the details of the recommended videos, passing in the ID of the video that can currently be viewed on the **VideoDetailPage**. When the recommended video details are retrieved the **RelatedVideos** property is set and property change notification updates the Boolean **HasRelatedVideos** property. The following code example shows the **GetRecommendationsAsync** method in the **VideoService** class.

C#

```
public async Task<ReadOnlyCollection<Video>> GetRecommendationsAsync(int videoId)
{
    var requestUri = new Uri(string.Format("{0}/{1}/recommendations",
        this.videosBaseUrl, videoId));
    var responseContent = await this.httpService.GetAsync(requestUri);

    var videos =
        JsonConvert.DeserializeObject<ReadOnlyCollection<Video>>(responseContent);
    return videos;
}
```

This method creates a URI that specifies that the **GetRecommendations** method will be called on the web service, with the video ID being passed as a parameter. The **HttpService** class, which implements the **IHttpService** interface, is used to make the call to the web service.

When the **GetRecommendationsAsync** method calls **HttpService.GetAsync**, this calls the **GetRecommendations** method in the **VideosController** class in the Contoso.Api project. For more information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

C#

```
public async Task<HttpResponseMessage> GetRecommendations(int videoId)
{
    ...
    var videoInfos = await this.videoService.GetRecommendations(videoId);
    var result = new List<VideoInfoDTO>();

    Mapper.Map(videoInfos, result);

    return Request.CreateResponse(HttpStatusCode.OK, result);
    ...
}
```

This method calls the **GetRecommendations** method in the **VideoService** class. The **VideoService** class is registered as a type mapping against the **IVideoService** interface with the Unity dependency injection container. When the **VideoController** constructor accepts an **IVideoService** type, the Unity container will resolve the type and return an instance of the **VideoService** class.

When a **VideoInfo** collection is returned from the **GetRecommendations** method, a collection of **VideoInfoDTO** objects is created, with the returned **VideoInfo** collection being mapped onto the **VideoInfoDTO** collection, which is then returned in a **HttpResponseMessage** to the **HttpService.GetAsync** method.

The **GetRecommendations** method in the **VideoService** class simply calls the **GetRecommendations** method in the **VideoRepository** class, and returns the results as a collection of **VideoInfo** objects. The following code example shows the **GetRecommendations** method in the **VideoRepository** class.

C#

```
using (var context = new VideoContext(this.NameOrConnectionString))
{
    // This code currently just pulls 5 randomly encoded videos just for the
    // sake of example. Normally you would build your recommendation
    // engine to your specific needs.
    List<VideoEntity> videos = null;

    var random = new Random();
    var limit = context.Videos.Count(s => s.EncodingStatus == (int)
EncodingState.Complete) - 5;
    var skip = random.Next(0, limit);
    videos = await context.Videos
            .Where(v => v.EncodingStatus == (int)EncodingState.Complete)
            .Include(v => v.Thumbnails)
            .OrderByDescending(v => v.LastUpdateDateTime)
            .Skip(skip).Take(5)
            .AsNoTracking()
            .ToListAsync()
            .ConfigureAwait(false);

    var result = new List<VideoInfo>();
    foreach (var video in videos)
    {
        var videoInfo = new VideoInfo()
        {
            Id = video.Id,
            Title = video.Title,
            Length = video.Length,
        };

        video.Thumbnails.ToList().ForEach(t => videoInfo.AddThumbnailUrl(new
VideoThumbnail() { Id = t.Id, Url = t.ThumbnailUrl }));

        result.Add(videoInfo);
    }

    return result;
}
```

Repository objects retrieve data from a context object by performing queries. Here, a LINQ query retrieves data from the **Video** table and its related tables, for five random videos. **VideoInfo** objects are then created for the data retrieved from the database, with the thumbnail data being added to the **Thumbnails** property of **VideoInfo** objects as **VideoThumbnail** objects. The collection of **VideoInfo** objects is then returned to the calling method.

For more information about using the Repository pattern to access data, see "Appendix A – The Contoso Web Service."

Summary

This chapter has described how the Contoso developers incorporated Media Services' delivery and consumption functionality into their web service and Windows Store client application. It summarized the decisions that they made in order to support their business requirements, and how they designed the code that requests and consumes encoded media.

In this chapter, you saw how dynamic packaging allows a video to be encoded just once, with it being converted in real-time to the format requested by the client, and the role that origin services play in delivering content from storage to the client application. The chapter also discussed how to scale delivery by reserving on-demand streaming reserved units.

More information

- For information about downloading a file from storage, see "Create a SAS Locator to On Demand Content" on MSDN.
- The page, "How to Manage Origins in a Media Services Account" explains how to add multiple streaming origins to your account and how to configure the origins, is available on MSDN.
- You can find the article "How to Scale a Media Service" on MSDN.
- You can find information about content protection at "Microsoft PlayReady."
- For information about using dynamic packaging to deliver MPEG DASH content encrypted with PlayReadY DRM, see "Task Preset for Azure Media Encryptor."
- For information about the Microsoft Player Framework see "Player Framework by Microsoft" on CodePlex.
- You can find the page, "Smooth Streaming Client SDK" lists the functionality supported in the Smooth Streaming Client SDK, available at the Visual Studio Gallery.

Appendix A - The Contoso Microsoft Azure Media Services Web Service

The purpose of the Contoso web service is to take REST requests sent from the client applications, validate these requests by applying the appropriate business logic, and then convert them into the corresponding Media Services operations. This appendix describes how the Contoso web service was designed and implemented to perform these tasks.

Understanding the web service

The developers at Contoso need the web service to provide the functionality to support the core business cases described in Chapter 2, "The Azure Media Services Video-on-Demand Scenario" The CMS uses an SQL database, but the developers wanted to design a system that decoupled the data storage technologies from the business logic of the web service. This approach enables Contoso to switch a particular repository to use a different data storage technology if necessary, without impacting the business logic in the web service. It also helps to ensure that the low level details of the database are not visible to the client applications, which could result in accidental dependencies between the user interface and the database.

> The Contoso web service is implemented in the Contoso.Api project in the solution provided with this guide.

The web service exposes its functionality through a series of controllers that respond to HTTP REST requests. The section "Routing incoming requests to a controller" describes the controllers and the requests that they support.

The business logic inside each controller stores and retrieves information in the database by using the Repository pattern. This pattern enabled the developers to isolate the database-specific aspects of the code inside a repository class, potentially allowing them to build different repository classes for different databases, and easily switching between them.

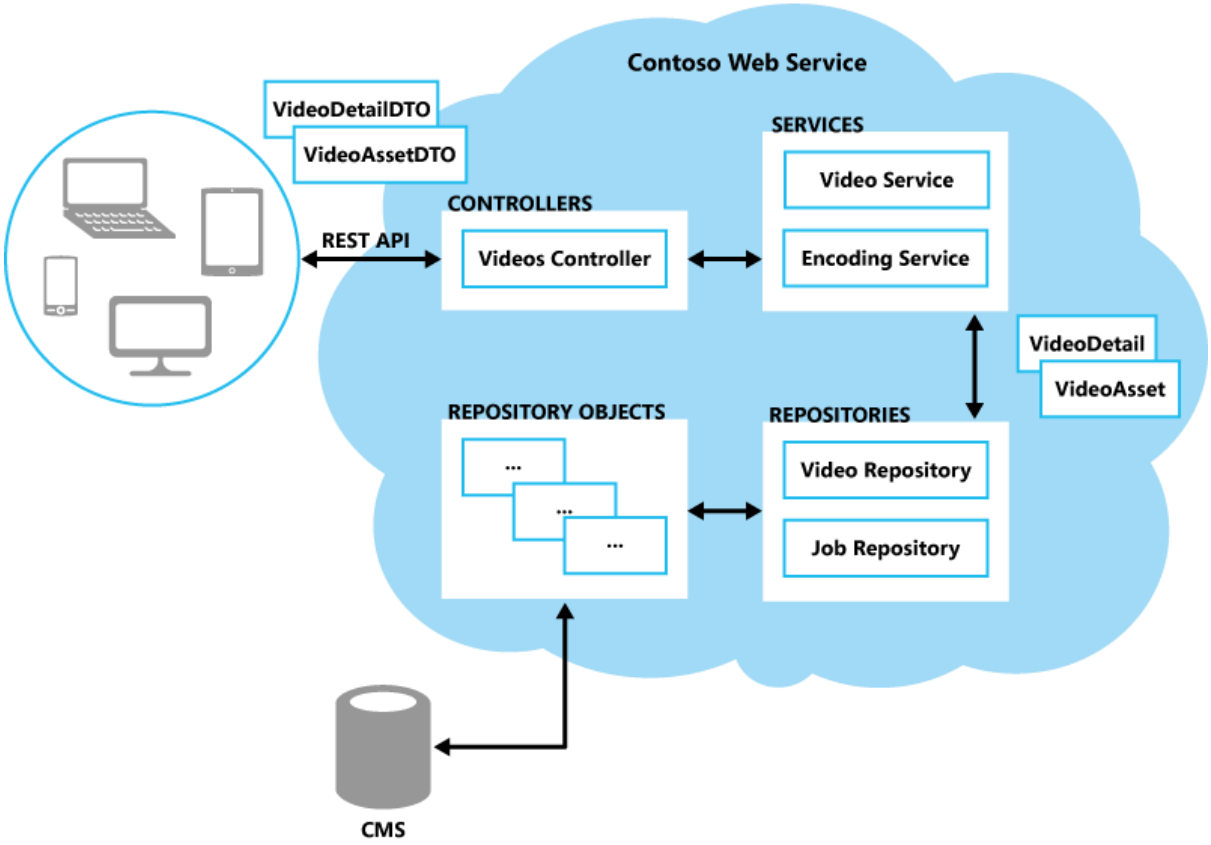
> **Note:** The developers decided to use the Unity Application Block to enable the code to use different repositories without requiring that the web service is rebuilt and deployed. The Unity Application Block allows a developer to specify how an application should resolve references to objects at runtime, rather than at compile time. The section "Instantiating service and repository objects" contains a description of how the developers at Contoso designed the web service to support dynamic configuration by using the Unity Application Block.

The data that passes between the controllers and repository classes are *domain entity objects*. A domain entity object is a database-neutral type that models business-specific data. It provides a logical structure that hides the way that the database stores and retrieves the data. The details of how the database entities are converted into domain entity objects are described in the section "Decoupling entities from the data access technology."

The response messages that the controllers create and send back to the client applications are HTTP REST messages, in many cases wrapping the data that was requested by a client application,

following the Data Transfer Object (DTO) pattern. The section "Transmitting data between a controller and a client" contains more information about the DTO objects created by the Contoso web service.

The following figure illustrates the high level flow of control through the Contoso web service. This figure depicts a client application sending a request that is handled by a controller in the web service. Controllers respond to requests, accessing data through services which use repositories. The repositories use database-specific repository objects to implement the data access logic used to retrieve and store data. The repositories also convert information between the database-specific format and neutral entity objects. The controller then uses the entity objects to construct DTOs that it passes back in the REST responses to the client apps.



The high-level flow of control through the Contoso web service

> **Note:** The Contoso web service implements only minimal security measures. Any application that knows the URL of the web service can connect to it and send requests. A real world implementation of this system would include more robust authentication, and prevent unauthorized use.

Routing incoming requests to a controller

All incoming REST requests are routed to a controller based on the URI that the client application specifies. The controllers are implemented by using the controller classes defined in the Controllers folder of the Contoso.Api project. The **VideosController** contains the following methods:

- **Get**: Returns a paged collection of videos.

- **Get**: Returns a video with a specified ID or a list of videos suitable for paging.
- **GetRecommendations**: Returns a list of videos that users who viewed the specified video also viewed.
- **GetResolutions**: Returns a list of resolutions from which a resolution can be selected to encode a video to.
- **GenerateAsset**: Creates an empty asset in Media Services that corresponds to a video to be uploaded to blob storage.
- **Publish**: Publishes a video uploaded to blob storage and submits it to the encoding pipeline for processing.
- **Update**: Updates the video details for a video with a specified ID in the CMS.

Note: The Contoso web service also defines the **HomeContoller** class. This class is the controller that is associated with the root URI of the web service, and it generates the default **Home** view for the website. This view displays a welcome page describing the ASP.NET Web API.

The Contoso web service uses attribute-based routing, where the routes exposed by the web service are defined as attributes on controller classes and controller methods. Attribute-based routing is enabled by calling **MapHttpAttributeRoutes** during configuration.

The **RoutePrefix** attribute is used to annotate a controller class with a route prefix that applies to all methods within the controller. The route prefix for the **VideosController** class is api/videos. This route responds with a JSON object or a JSON array containing the data returned by the **Get** method of the specified controller, passing the ID as the parameter to this method. An example of a typical route is api/videos/71 which returns the video details for the video with an ID of 71.

Additional routes can be specified by using the **Route** attribute to annotate a controller method so that the method is exposed directly via a route. The following table summarizes the methods in the **VideoController** class are annotated with the **Route** attribute.

Method	Route	Example
GetRecommendations	{videoId:int}/recommendations	api/videos/77/recommendations
Resolutions	resolutions	api/videos/resolutions
GenerateAsset	generateasset	api/videos/generateasset
Publish	publish	api/videos/publish

These routes respond with a JSON object or a JSON array containing the data returned by the specified method of the controller, passing in parameters to the methods.

For more information about defining routes for Web API web services see "Attribute Routing in Web API 2."

Note: The web service is configured to accept and handle cross-origin resource sharing (CORS) requests. This feature enables the web client application to run in a separate web domain from the Contoso web service.

Transmitting data between a controller and a client

The controllers receive REST requests and send REST responses. The data received in these requests and sent in these responses can contain structured information, such as the details of a video. The Contoso web service defines a series of serializable data types that define the shape of this structured information. These types act as data transfer objects (DTO), and the information that they contain is transmitted as JSON objects and JSON arrays. These DTOs are implemented in the Models folder of the Contoso.Api project. The following table briefly describes these classes.

Model class	Description
VideoAssetDTO	Specifies the details of a video asset. The properties are the shared access signature locator, and an asset ID.
VideoDetailDTO	Holds the details of a video, including its ID, title, description, length, thumbnails, subtitles, captions, tags, and metadata.
VideoInfoDTO	Contains the ID, title, thumbnails, and length of a video.
VideoMetadataDTO	Holds the ID, name, and metadata for a video.
VideoPlayDTO	Specifies the URL and encoding type for video.
VideoPublishDTO	Inherits from the **VideoSaveDTO**, adding details required when publishing a video, such as the asset ID, clip times, whether thumbnails are included, and the resolution.
VideoSaveDTO	Contains the ID, title, description, and length of a video to be saved.
VideoThumbnailDTO	Specifies the ID and URL for a video's thumbnail.

The benefits of using DTOs to pass data to and receive data from a web service are that:

- By transmitting more data in a single remote call, the app can reduce the number of remote calls. In most scenarios, a remote call carrying a large amount of data takes virtually the same time as a call that carries only a small amount of data.
- Passing more data in a single remote call more effectively hides the internals of the web service behind a coarse-grained interface.
- Defining a DTO can help in the discovery of meaningful business objects. When creating DTOs, you often notice groupings of elements that are presented to a user as a cohesive set of information. Often these groups serve as useful prototypes for objects that describe the business domain that the app deals with.
- Encapsulating data into a serializable object can improve testability.

The Contoso web service uses AutoMapper to convert the domain entity objects that it receives from the various repository classes into DTOs. AutoMapper simplifies the process of translating

objects from one format into another by defining a set of mapping rules that can be used throughout an application. In the Contoso web service, these rules are defined in the AutoMapperConfig.cs file in the App_Start folder in the Contoso.Api project.

As an example, the following code example shows the **VideoInfo** domain entity type that contains the data returned by the **GetRecommendations** method in the **VideoRepository** class.

C#

```
public class VideoInfo
{
    private readonly IList<VideoThumbnail> thumbnails =
        new List<VideoThumbnail>();

    public int Id { get; set; }

    public string Title { get; set; }

    public string Length { get; set; }

    public IReadOnlyCollection<VideoThumbnail> Thumbnails
    {
        get { return new ReadOnlyCollection<VideoThumbnail>(this.thumbnails); }
    }
        ...
}
```

The **VideoInfoDTO** class is shown in the following code example, and is returned by the **GetRecommendations** method of the **VideosController** class.

C#

```
public class VideoInfoDTO
{
    public int Id { get; set; }
    public string Title { get; set; }
    public ICollection<VideoThumbnailDTO> Thumbnails { get; set; }
    public string Length { get; set; }
}
```

The AutoMapperConfig.cs file contains mappings that specify how to populate a **VideoInfoDTO** from a **VideoInfo** entity object, as shown in the following code example.

C#

```
public static void SetAutoMapperConfiguration()
{
    Mapper.CreateMap<VideoInfo, VideoInfoDTO>();
    ...
}
```

This mapping specifies that the **VideoInfoDTO** properties should be read from the corresponding **VideoInfo** properties.

The **GetRecommendations** method in the **VideosController** class uses the static **Map** method of the **Mapper** class to translate a collection of **VideoInfo** domain entity objects returned by the

VideoRepository class into a collection of **VideoInfoDTO** objects that it sends back in the REST response to the client application.

C#

```
public async Task<HttpResponseMessage> GetRecommendations(int videoId)
{
    ...
    var videoInfos = await this.videoService.GetRecommendations(videoId);
    var result = new List<VideoInfoDTO>();

    Mapper.Map(videoInfos, result);

    return Request.CreateResponse(HttpStatusCode.OK, result);
    ...
}
```

For more information about AutoMapper, see "AutoMapper."

Using the Repository pattern to access data

The Repository pattern enables you to separate the code that accesses a data store from the business logic that uses the data in the store, minimizing any dependencies that the controller might have on the data store. The **VideosController** creates one or more service classes that use repository objects to connect to the database and retrieve, create, or update data. For example, the following code shows the **Get** method in the **VideosController** class that uses the **GetVideo** method of the **VideoService** class to fetch metadata for a video.

C#

```
public async Task<HttpResponseMessage> Get(int id)
{
    ...
    var videoDetail = await this.videoService.GetVideo(id);
    ...
}
```

In turn, the **GetVideo** method of the **VideoService** class uses the **GetVideo** method of the **VideoRepository** class to fetch metadata for a video.

C#

```
public async Task<VideoDetail> GetVideo(int id)
{
    var result = await this.videoRepository.GetVideo(id).ConfigureAwait(false);
    return result;
}
```

Each repository object is an instance of a database-specific repository class. The repository classes implement a set of database-agnostic interfaces. The methods defined by the repository interfaces return or manipulate collections and instances of domain entity objects. The following table summarizes the repository interfaces.

Repository interface	Methods	Description
IJobRepository	**GetJob** **SaveJob**	This repository interface provides access to encoding job information. The **GetJob** method returns an **EncodingJob** that matches the specified job ID. The **SaveJob** method writes an **EncodingJob** domain entity object to the database.
IVideoRepository	**GetEncodedVideos** **GetVideo** **GetRecommendations** **SaveVideo** **GetEncodedVideosCount**	This repository interface manages videos in the database and provides access to recommendation information for a video. The **GetEncodedVideos** method returns a collection of **VideoInfo** objects suitable for paging. The **GetVideo** method returns a **VideoDetail** object that matches the specified ID. The **GetRecommendations** method returns a collection of **VideoInfo** objects that match the specified ID. The **SaveVideo** method saves the **VideoDetail** object passed into the method as a parameter to the database. The **GetEncodedVideosCount** methods returns an integer that represents the number of completely encoded videos listed in the Content Management System database.

The repository interfaces are defined in the Contoso.Repositories project.

The repository classes and the data types that the repository classes use to retrieve and modify data in the SQL Server database, are defined in the Contoso.Repositories.Impl.Sql project. This project contains two repository classes named **JobRepository** and **VideoRepository**, that use the Entity Framework v6 to connect to the database. The classes provide the business methods that the service classes use to store and retrieve data. Internally, these repository classes retrieve and save data by using a *context* object (**JobContext** or **VideoContext** as appropriate). Each context object is responsible for connecting to the database and managing or retrieving data from the appropriate tables.

Retrieving and storing data in the database

The context classes are defined in the Contoso.Repositories.Impl.Sql project. They are all custom Entity Framework context objects derived from the **DbContext** class that exposes the data from the SQL Server database through a group of public **IDbSet** collection properties. The following code example shows how some of the properties for handling encoding jobs are defined in the **JobContext** class.

C#

```
public class JobContext : DbContext
{
    ...
    public IDbSet<JobEntity> Jobs { get; set; }
    public IDbSet<TaskEntity> Tasks { get; set; }
    public IDbSet<AssetEntity> Assets { get; set; }
    public IDbSet<TaskAssetEntity> TaskAssets { get; set; }
    ...
}
```

The type parameters referenced by the **IDbSet** collection properties are entity classes that are specific to the Entity Framework repositories. They correspond to individual tables in the SQL Server database. These classes are defined in the DataEntities folder in the Contoso.Repositories.Impl.Sql project. The following table summarizes the entity classes.

Entity class	Description
AssetEntity	This class contains the asset ID of an asset being processed by a task in an encoding job.
BaseEntity	This is the base class from which all other entity classes derive, and defines the Id property.
JobEntity	This class contains information about an encoding job for a **VideoEntity** object, including the collection of **TaskEntity** objects that make up the encoding job.
MetadataEntity	This class contains the metadata name and value, and a collection of **VideoEntity** objects that the metadata value is applied to.
TaskAssetEntity	This class contains information about an **AssetEntity** object being processed by a **TaskEntity** object.
TaskEntity	This class contains information about a processing Task for a **JobEntity** object.
VideoEntity	This class contains the data that describes a video. Many of the properties correspond to fields in the **Video** table in the SQL Server database, and relationships between the **Video** table and other tables such as **Metadata** and **VideoThumbnail** are implemented as collections of the appropriate entity object.
VideoThumbnailEntity	This class contains the thumbnail information for a

	VideoEntity object.
VideoUrlEntity	This class describes the URL and encoding type for a **VideoEntity** object.

The Contoso developers followed a code first approach to implementing the entity model for the repositories and they defined each of the entity classes manually. The following code example shows the **JobEntity** class.

C#

```
public class JobEntity : BaseEntity
{
    public string JobUuId { get; set; }
    public DateTime CreateDateTime { get; set; }
    public ICollection<TaskEntity> Tasks { get; set; }
    public int VideoId { get; set; }
    public virtual VideoEntity Video { get; set; }
}
```

The **JobContext** class uses the Entity Framework Fluent API to map the entity classes to tables in the SQL Server database and specify the relationships between tables. The following code example shows the **OnModelCreating** event handler method for the **JobContext** class that populates the **Jobs**, **Tasks**, **Assets**, and **TaskAssets IDbSet** collections. The event handler runs when the **JobContext** object is created, and it passes a reference to the in-memory data model builder that is constructing the model for the Entity Framework as the parameter.

C#

```
public class JobContext : DbContext
{
    ...
    public IDbSet<JobEntity> Jobs { get; set; }
    public IDbSet<TaskEntity> Tasks { get; set; }
    public IDbSet<AssetEntity> Assets { get; set; }
    public IDbSet<TaskAssetEntity> TaskAssets { get; set; }
    ...

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        ...
        modelBuilder.Entity<JobEntity>().Property(j => j.Id)
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        modelBuilder.Entity<JobEntity>()
            .ToTable("Job")
            .HasKey(j => j.Id);

        modelBuilder.Entity<JobEntity>()
            .HasMany(j => j.Tasks)
            .WithOptional()
            .HasForeignKey(t => t.JobId)
            .WillCascadeOnDelete(true);

        ...
```

```
    }
}
```

The code in this method describes how to associate each of the entity classes with the corresponding tables in the database. Specifically, the code stipulates that:

- Instances of the **JobEntity** type should be mapped to rows in the **Job** table in the SQL Server database.
- The **Id** field contains the primary key used to establish relationships with other objects.
- The **Job** table in the database has a one-to-many relationship with the **Task** table, with the relationship also being a many to optional relationship.
- The **JobId** property of the **Task** table is used as a foreign key.
- The tasks for a job should be removed if the job is deleted.

For more information about constructing an entity model at runtime by using the Fluent API, see "Configuring/Mapping Properties and Types with the Fluent API" and "Configuring Relationships with the Fluent API."

Reading and writing the data for objects

In the context classes, each **IDbSet** property is an enumerable collection. This means that a repository object can retrieve data from a context object by performing LINQ queries. The following code example shows the **GetJob** method in the **JobRepository** class that uses a **JobContext** object to retrieve job data from the database through the **Jobs IDbSet** collection.

C#

```
public class JobRepository : IJobRepository
{
    ...
    public async Task<EncodingJob> GetJob(string jobId)
    {
        using (JobContext context = new JobContext(this.NameOrConnectionString))
        {
            var job = await context.Jobs.Include(j => j.Tasks.Select(t =>
                t.TaskAssets.Select(a => a.Asset))).SingleOrDefaultAsync(j =>
                j.JobUuId == jobId).ConfigureAwait(false);
            ...
        }
    }
}
```

The data for a **JobEntity** object also includes task information that's retrieved from a separate table in the SQL Server database. The **Include** method ensures that the tasks are populated when the data for the **JobEntity** object is retrieved from the database (the Entity Framework performs lazy evaluation by default).

All the changes made to the collections attached to a context object are packaged up and converted into the corresponding SQL commands by the **SaveChangesAsync** method inherited from the

DbContext class. For example, the **SaveJob** method in the **JobRepository** class creates and populates:

- **TaskEntity** objects that are added to the **Tasks** collection in the **JobEntity** object.
- **TaskAssetEntity** objects that are added to the **TaskAssets** collection in the **TaskEntity** object.
- **AssetEntity** objects that are added to the **Asset** collection in the **TaskAssetEntity** object.

The **JobEntity** object is then added to the Jobs collection in the **DbContext** object. The call to the **SaveChangesAsync** method ensures that the new job, tasks, task assets, and asset details are saved to the tables in the SQL Server database.

C#

```
public async Task SaveJob(EncodingJob job)
{
    ...
    var jobEntity = new JobEntity()
    {
        CreateDateTime = DateTime.UtcNow,
        Id = job.Id,
        JobUuId = job.EncodingJobUuId,
        VideoId = job.VideoId,
        Tasks = new List<TaskEntity>()
    };

    foreach(var encodingTask in job.EncodingTasks)
    {
        var taskEntity = new TaskEntity()
        {
            Id = encodingTask.Id,
            TaskUuId = encodingTask.EncodingTaskUuId,
            TaskAssets = new List<TaskAssetEntity>(),
            JobId = job.Id
        };

        foreach(var encodingAsset in encodingTask.EncodingAssets)
        {
            taskEntity.TaskAssets.Add(new TaskAssetEntity()
            {
                Asset = new AssetEntity()
                {
                    AssetUuId = encodingAsset.EncodingAssetUuId,
                    Id = encodingAsset.Id
                },
                IsInputAsset = encodingAsset.IsInputAsset,
                Task = taskEntity,
                AssetId = encodingAsset.Id,
                TaskId = taskEntity.Id
            });
        }
```

```
        jobEntity.Tasks.Add(taskEntity);
    }

    using (JobContext context = new JobContext())
    {
        ...
        if(jobEntity.Id != 0)
        {
            context.Jobs.Attach(jobEntity);
            context.Entry(jobEntity).State = EntityState.Modified;
        }
        else
        {
            context.Jobs.Add(jobEntity);
        }

        await context.SaveChangesAsync().ConfigureAwait(false);
    }
    ...
}
```

Decoupling entities from the data access technology

The data that passes between the controllers and service and repository classes are instances of *domain entity objects*. These are database-neutral types that help to remove any dependencies that the controller classes might otherwise have on the way that the data is physically stored, and can be found in the Contoso.Domain project.

The repository classes in the Contoso web service pass domain entity objects to the controller classes through the service classes. Therefore, an important task of the repository classes is to take the data that they retrieve from a database and return it as one or more of the domain entity objects.

For example, the **VideoRepository** class uses the **VideoContext** class to retrieve data from the database, which in turn uses a database-specific class. Therefore, the **VideoRepository** class must also convert the collection of data in the database-specific class to a collection of domain entity objects that can be passed to the relevant controller class. The following code example shows how this is accomplished in the **GetRecommendations** method.

C#

```
public async Task<ICollection<VideoInfo>> GetRecommendations(int id)
{
    using (var context = new VideoContext(this.NameOrConnectionString))
    {
        // This code currently just pulls 5 randomly encoded videos just for
        // the sake of example. Normally you would build you recommendation
        // engine to your specific needs
        List<VideoEntity> videos = null;

        var random = new Random();
```

```
        var limit = context.Videos.Count(s => s.EncodingStatus == (int)
EncodingState.Complete) - 5;
        var skip = random.Next(0, limit);
        videos = await context.Videos
            .Where(v => v.EncodingStatus == (int)EncodingState.Complete)
            .Include(v => v.Thumbnails)
            .OrderByDescending(v => v.LastUpdateDateTime)
            .Skip(skip).Take(5)
            .AsNoTracking()
            .ToListAsync()
            .ConfigureAwait(false);

        var result = new List<VideoInfo>();
        foreach (var video in videos)
        {
            var videoInfo = new VideoInfo()
            {
                Id = video.Id,
                Title = video.Title,
                Length = video.Length,
            };

            video.Thumbnails.ToList().ForEach(t => videoInfo.AddThumbnailUrl(new
VideoThumbnail() { Id = t.Id, Url = t.ThumbnailUrl }));

                result.Add(videoInfo);
        }

        return result;
    }
}
```

The **GetMetadata** method retrieves data from the database by querying the **Videos** property in the **VideoContext** class, which is an enumerable collection of **VideoEntity** objects. The collection of **VideoEntity** objects is then converted into a collection of **VideoInfo** domain entity objects that can be passed to the **VideosController** class.

Instantiating service and repository objects

The service and repository classes implement a common set of interfaces that enabled the Contoso developers to decouple the business logic in the **VideosController** class from the database-specific code. However, the controller still has to instantiate the appropriate service classes, and each service class still has to instantiate the appropriate repository class. This process can introduce dependencies back into the code for the controller classes if it is not performed carefully. To eliminate these dependencies the Contoso developers chose to use the Unity Application Block to inject references to the service and repository classes.

The **VideosController** class references the services that it uses through the service interface. The following code example shows how the **VideosController** class references the **VideoService** and **EncodingService** class through the **IVideoService** and **IEncodingService** interfaces.

C#

```
public VideosController(IVideoService videoService,
    IEncodingService encodingService)
{
    ...
    this.videoService = videoService;
    this.encodingService = encodingService;
}
```

The constructor is passed a reference to the **IVideoService** and **IEncodingService** objects that are used by the methods in the **VideosController** class. At runtime, the Contoso web service uses the Unity Application Block to resolve the **IVideoService** and **IEncodingService** references that were passed to the **VideosController** constructor, and creates new **VideoService** and **EncodingService** objects. The **UnityConfig** class in the App_Start folder of the Contoso.Api project is responsible for registering the type mappings with the Unity container. The following code example shows the **RegisterTypes** method from the **UnityConfig** class.

C#

```
public static void RegisterTypes(IUnityContainer container)
{
    ...
    container.RegisterType<IVideoService, VideoService>(
        new PerResolveLifetimeManager());
    container.RegisterType<IEncodingService, EncodingService>(
        new PerResolveLifetimeManager());
}
```

This registration enables the Unity Application Block to resolve references to the **IVideoService** and **IEncodingService** interfaces as instances of the **VideoService** and **EncodingService** class in the Contoso.Domain.Services.Impl assembly.

The same mechanism is used in service classes to instantiate repository classes. For example, the **VideoService** constructor is passed an **IVideoRepository** object that is used by methods in the **VideoService** class. At runtime, the Contoso web service uses the Unity Application Block to resolve the **IVideoRepository** reference passed to the **VideoService** constructor and creates a new **VideoRepository** object.

For more information about using the Unity Application Block to resolve dependencies, see "Unity Container."

More information

- The page "Attribute Routing in Web API 2" is available on the ASP.NET website.
- You can find details about AutoMapper on GitHub.
- The Repository pattern is described in "Patterns of Enterprise Application Architecture" on Martin Fowler's website.
- Information describing how to use the Fluent API in the Entity Framework 6.0 to map types to database tables is available on MSDN at "Configuring/Mapping Properties and Types with the Fluent API."

- Information describing how to configure relationships between entities by using the Fluent API is available at "Configuring Relationships with the Fluent API."
- You can download the Unity Application Block from MSDN.

Appendix B - Microsoft Azure Media Services Encoder Presets

This appendix contains tables which list task preset encoding strings for Microsoft Azure Media Encoder. These strings can be used when creating encoding tasks for media content in a Media Services application. The presets enable the submission of encoding jobs to the Media Encoder using various media content formats.

Note: If the source video is not of the correct resolution it will be scaled horizontally to the width of the profile target and its height will be scaled to match the aspect ratio of the source.

Note: The material in this Appendix is provided for your convenience. The most current information can be found at Media Services Encoder System Presets on MSDN.

H.264 coding presets

Preset string	Description	Scenario	Resolution	File extension
H264 Broadband 1080p	Produces an MP4 file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with 1080p CBR video encoded at 6750 kbps using H.264 High Profile.	Use this preset to produce a downloadable file for 1080p content (16:9 aspect ratio) for delivery over broadband connections.	1920x1080	.mp4
H264 Broadband 720p	Produces an MP4 file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with 720p CBR video encoded at 4500 kbps using H.264 Main Profile.	Use this preset to produce a downloadable file for 720p content (16:9 aspect ratio) for delivery over broadband connections.	1280x720	.mp4
H264 Broadband SD 16x9	Produces an MP4 file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with SD VBR video encoded at 2200 kbps using H.264 Main Profile.	Use this preset to produce a downloadable file for SD content (16:9 aspect ratio) for delivery over broadband connections.	852x480	.mp4
H264 Broadband SD 4x3	Produces an MP4 file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with SD VBR video encoded at 1800 kbps using H.264 Main Profile.	Use this preset to produce a downloadable file for SD content (4:3 aspect ratio) for delivery over broadband connections.	640x480	.mp4
H264 Smooth Streaming 1080p	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with 1080p CBR video encoded at 8 bitrates from 400 kbps to 6000 kbps using H.264	Use this preset to produce an asset from 1080p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	1920x1080	.ismv, .isma

	High Profile, and two second GOPs.			
H264 Smooth Streaming 720p	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with 720p CBR video encoded at 6 bitrates ranging from 400 kbps to 3400 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	1280x720	.ismv, .isma
H264 Smooth Streaming 720p for 3G or 4G	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 56 kbps using AAC, with 720p CBR video encoded at 8 bitrates ranging from 150 kbps to 3400 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery over 3G or 4G connections via IIS Smooth Streaming.	1280x720	.ismv, .isma
H264 Smooth Streaming SD 16x9	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with SD CBR video encoded at 5 bitrates ranging from 400 kbps to 1900 kbps using H.264 Main Profile, and two second GOPs	Use this preset to produce an asset from SD content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	852x480	.ismv, .isma
H264 Smooth Streaming SD 4x3	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with SD CBR video encoded at 5 bitrates ranging from 400 kbps to 1600 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (4:3 aspect ratio) for delivery via IIS Smooth Streaming.	640x480	.ismv, .isma
H264 Adaptive Bitrate MP4 Set 1080p	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using AAC, with 1080p CBR video encoded at 8 bitrates ranging from 400 kbps to 6000 kbps using H.264 High Profile, and two second GOPs.	Use this preset to produce an asset from 1080p content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	1920x1080	
H264 Adaptive Bitrate MP4 Set 720p	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with 720p CBR video encoded at 6 bitrates ranging from 400 kbps to 3400 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	1280x720	

H264 Adaptive Bitrate MP4 Set SD 16x9	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with SD CBR video encoded at 5 bitrates ranging from 400 kbps to 1900 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	852x480	
H264 Adaptive Bitrate MP4 Set SD 4x3	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with SD CBR video encoded at 5 bitrates ranging from 400 kbps to 1600 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (4:3 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	640x480	
H264 Adaptive Bitrate MP4 Set 1080p for iOS Cellular Only	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 56 kbps using AAC, with 1080p CBR video encoded at 8 bitrates ranging from 400 kbps to 6000 kbps using H.264 High Profile, and two second GOPs.	Use this preset to produce an asset from 1080p content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	1920x1080	
H264 Adaptive Bitrate MP4 Set 720p for iOS Cellular Only	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 56 kbps using AAC, with 720p CBR video encoded at 6 bitrates ranging from 400 kbps to 3400 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	1280x720	
H264 Adaptive Bitrate MP4 Set SD 16x9 for iOS Cellular Only	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 56 kbps using AAC, with SD CBR video encoded at 5 bitrates ranging from 400 kbps to 1900 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (16:9 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	852x480	
H264 Adaptive Bitrate MP4 Set SD 4x3 for iOS Cellular Only	Produces an asset with multiple GOP-aligned MP4 files at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 56 kbps using AAC, with SD CBR video encoded at 4 bitrates ranging from 400 kbps to 1600 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (4:3 aspect ratio) for delivery via one of many adaptive streaming technologies after suitable packaging.	640x480.	
H264	Produces a Smooth Streaming	Use this preset to produce at	1280x720	.ismv,

Smooth Streaming 720p Xbox Live ADK	asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 96 kbps using AAC, with 720p CBR video encoded at 8 bitrates ranging from 350 kbps to 4500 kbps using H.264 High Profile, and two second GOPs.	an asset from 720p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming to Xbox Live Applications.		.isma
H264 Smooth Streaming Windows Phone 7 Series	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 64 kbps using HE-AAC Level 1, with SD CBR video encoded at 5 birrates ranging from 200 kbps to 1000 kbps using H.264 Main Profile, and two second GOPs.	Use this preset to produce an asset from SD content (16:9 aspect ratio) for delivery via IIS Smooth Streaming to Windows Phone 7 series devices.	640x360	

VC-1 coding presets

Preset string	Description	Scenario	Resolution	File extension
VC1 Broadband 1080p	Produces a Windows Media file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 1080p VBR video encoded at 6750 kbps using VC-1 Advanced Profile.	Use this preset to produce a downloadable file for 1080p content (16:9 aspect ratio) for delivery over broadband connections.	1920x1080	.wmv
VC1 Broadband 720p	Produces a Windows Media file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 720p VBR video encoded at 4500 kbps using VC-1 Advanced Profile.	Use this preset to produce a downloadable file for 720p content (16:9 aspect ratio) for delivery over broadband connections.	1280x720	.wmv
VC1 Broadband SD 16x9	Produces a Windows Media file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with SD VBR video encoded at 2200 kbps using VC-1 Advanced Profile.	Use this preset to produce a downloadable file for SD content (16:9 aspect ratio) for delivery over broadband connections.	852x480	.wmv
VC1 Broadband SD 4x3	Produces a Windows Media file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with SD VBR video encoded at 1800 kbps using VC-1 Advanced Profile.	Use this preset to produce a downloadable file for SD content (4:3 aspect ratio) for delivery over broadband connections	640x480	.wmv
VC1 Smooth Streaming 1080p	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 1080p VBR video encoded at 8 bitrates	Use this preset to produce an asset from 1080p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	1920x1080	.ismv, .isma

	ranging from 400 kbps to 6000 kbps using VC-1 Advanced Profile, and two second GOPs.			
VC1 Smooth Streaming 720p	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 720p VBR video encoded at 6 bitrates from 400 kbps to 3400 kbps using VC-1 Advanced Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	1280x720	.ismv, .isma
VC1 Smooth Streaming SD 16x9	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 64 kbps using WMA Pro, with SD VBR video encoded at 5 bitrates ranging from 400 kbps to 1900 kbps using VC-1 Advanced Profile, and two second GOPs.	Use this preset to produce an asset from SD content (16:9 aspect ratio) for delivery via IIS Smooth Streaming.	852x480	.ismv, .isma
VC1 Smooth Streaming SD 4x3	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 64 kbps using WMA Pro, with SD VBR video encoded at 5 bitrates ranging from 400 kbps to 1600 kbps using VC-1 Advanced Profile, and two second GOPs.	Use this preset to produce an asset from SD Content (4:3 aspect ratio) for delivery via IIS Smooth Streaming	640x480	.ismv, .isma
VC1 Smooth Streaming 1080p Xbox Live ADK	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 1080p VBR video encoded at 10 bitrates ranging from 350 kbps to 9000 kbps using VC-1 Advanced Profile, and two second GOPs.	Use this preset to produce an asset from 1080p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming to Xbox Live Applications.	1920x1080	.ismv, .isma
VC1 Smooth Streaming 720p Xbox Live ADK	Produces a Smooth Streaming asset at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 128 kbps using WMA Pro, with 720p VBR video encoded at 8 bitrates ranging from 350 kbps to 4500 kbps using VC-1 Advanced Profile, and two second GOPs.	Use this preset to produce an asset from 720p content (16:9 aspect ratio) for delivery via IIS Smooth Streaming to Xbox Live Applications.	1280x720	.ismv, .isma

Audio coding presets

Preset string	Description	Scenario	File extension
AAC Good Quality Audio	Produces an MP4 file at 44.1 kHz 16 bits/sample stereo CBR audio encoded at 192 kbps using AAC.	Use this preset to produce an audio-only file for music services.	.mp4
WMA High Quality Audio	Produces a Windows Media file at 44.1 kHz 16 bit/sample stereo audio encoded using WMA.	Use this preset to produce an audio-only file for music services.	.wma

More information

- Media Services Encoder System Presets
- Process Assets with the Media Services SDK for .NET
- Encoding your media with Dolby Digital Plus
- Task Preset for Azure Media Encryptor
- Task Preset for Azure Media Packager
- Task Preset for Thumbnail Generation

Appendix C - Understanding the Contoso Microsoft Azure Media Services Video Applications

Microsoft provides SDKs and Player Frameworks that enable you to create client applications that can consume streaming media from Media Services. Player Frameworks are built on top of SDKs and contain user controls for common application scenarios.

This appendix describes the Visual Studio project structure for the different Contoso video applications, and focuses on the implementation of the Contoso video web application.

Understanding the Contoso video web application

The Contoso web video application is implemented in the Contoso.WebClient project in the solution provided with this guide. This project uses Knockout.js to provide MVVM and data binding support.

This Contoso.WebClient project contains the following folders:

- **App_Start**: Contains the configuration logic for the application.
- **Content**: Contains the CSS files for the application.
- **Controllers**: Contains the controller class used by the application.
- **fonts**: Contains the font data that is loaded when the application is bootstrapped.
- **Images**: Contains the image that is animated when data is being retrieved from the Contoso web service.
- **Scripts**: Contains the JavaScript files that implement the view models that contain the application logic.
- **Views**: Contains the views that display data to the user.

The following sections explain how the Contoso web client application implements the primary business use cases outlined in Chapter 2, "The Azure Media Services Video-on-Demand Scenario."

> The Contoso video web application does not support capturing videos. It only supports uploading videos to Media Services.

Browsing videos

The following figure shows how the Contoso web video application allows the user to browse thumbnails of available videos that can be viewed.

Thumbnails for each video that can be viewed

The **VideoList** view allows the user to browse thumbnails of the available videos, and select one for viewing. The following code example shows the HTML that displays the list of videos.

HTML

```
<div class="videolist" id="videolist">
    ...
    <div data-bind="ifnot: Error">
        ...
        <div data-bind="foreach: VideoList">
            <div class='videoItem'>
                <div>
                    <img class='videoItemThumbnail' data-bind="attr: { src:
ThumbnailUrl, tabindex: TabIndex, videoId: Id }" alt="No Image Available">
                </div>
                <div class='videoItemInfo'>
                    <p data-bind="text: Title"> </p>
                    <span>Length: </span><span data-bind="text: Length"> </span>
                </div>
            </div>
        </div>
        ...
    </div>
    ...
</div>
```

This code binds a **div** element to the **VideoList** property in the **VideoListViewModel** class. A thumbnail representing each video is displayed for each item in the **VideoList** by using an **img** element, which binds to the **ThumbnailUrl** of the **VideoList** item. The **class** attributes of the **DIV** elements are set to use styles from the **videoList.css** file.

The instance of the **VideoListViewModel** is bound to the **videolist** DOM element. When the **VideoListViewModel** instance is created the **ajaxGetVideoList** function is executed, which retrieves the list of videos for display. The following code example shows the **ajaxGetVideoList** function.

JavaScript

```
function ajaxGetVideoList() {
    reset();
    videoDetailVM.reset();
    self.ShowProgress(true);
    var request = {
        type: "GET",
        url: Contoso.apiRoot + "/videos/",
        data: { pageIndex: pageIndex, pageSize: pageSize },
        timeout: Contoso.ajaxTimeOut,
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
    ...
```

This function creates a URL that specifies the address that will be requested from the web service, with the paging options being passed as a parameter. When AJAX makes the request the **Get** method in the **VideosController** class in the Contoso.Api project is invoked. This method returns a collection of **VideoInfoDTO** objects in the response message, from which the **VideoList** collection is populated. For more information about how the **VideosController** class returns a list of videos to be displayed to the user, see "Browsing videos."

For information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

> The Contoso web video application consumes the paging functionality offered by the Contoso web service.

Playing videos

The following figure shows how the video player control in the Contoso web video application, along with video information beneath the player. A **Show Clip** button is displayed directly beneath the video player if a clip of the video also exists.

Show Clip

Title:

The Big Buck Bunny (2008)

Length:

00:09:56

Description:

Project Peach was the Blender Foundation's second open movie. The movie started production in October 2007 and premiered April 2008. The targets this time were to create good hair/fur editing and rendering, more advanced support for cartoon characters, and improve performance with complex outdoors environments with grass, trees and leaves. Big Buck Bunny was the first project created in the Blender Institute's studio in Amsterdam. A giant, gentle rabbit finds his happy morning walk being disturbed by rodents who kill his two favorite butterflies. In rage, he sets up a masterful plan to avenge the deaths of the butterflies. You can watch the full movie on YouTube. http://www.bigbuckbunny.org/

Id:

1031

Video playback

> Most web browsers only support progressive download with the HTML5 **video** element. Additional frameworks are required to enable support for browser-based streaming. For more information about these frameworks see "Developing Azure Media Services Client Applications."

The **VideoList** view allows the user to control the playback of a selected video, and allows the user to view a clip from the video if one is available. The following code example shows the HTML that displays the video player.

HTML

```
<div class="videodetail" id="videodetail">
        ...
        <div data-bind="with: VideoDetail">
            ...
            <video id="selectedvideo" data-bind="attr: { poster: ThumbnailUrl },
foreach: Videos" preload controls>
                <source data-bind="attr: { src: Url, type: EncodingType }" />
            </video>
            ...
        </div>
</div>
```

This code binds a **div** element to the **VideoDetail** property of the **VideoDetailViewModel** class. The **video** element specifies the video content to be played, and binds to the **ThumbnailUrl** of the **VideoDetail** object in order to display the video thumbnail. The **source** element provides a mechanism to specify an alternative media resource for the **video** element, and binds to the **Url** of the **VideoDetail** object, from which the video stream can be retrieved. The **class** attributes of the **DIV** elements are set to use styles from the **videoList.css** file.

> A similar mechanism is used to display video clips, if the **Show Clip** button is toggled.

The instance of the **VideoDetailVideoModel** is bound to the **videodetail** DOM element. When the **VideoListViewModel** instance is created the **ajaxGetVideoDetail** function is executed, which retrieves the video details for the video to be played. The following code example shows the **ajaxGetVideoDetail** function.

JavaScript

```
function ajaxGetVideoDetail(id) {
    reset();
    self.ShowProgress(true);
    var request = {
        type: "GET",
        dataType: "json",
        timeout: Contoso.ajaxTimeOut,
        url: Contoso.apiRoot + "/videos/" + id,
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
    ...
```

This function creates a URL that specifies the address that will be requested from the web service, which includes the ID of the video whose details will be retrieved. When AJAX makes the request the **Get** method in the **VideosController** class in the Contoso.Api project is invoked. This method returns a **VideoDetailDTO** object in the response message, from which the **VideoDetail** object is populated. For more information about how the **VideosController** class returns the video details of the video to be played, see "Playing videos."

For information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

Retrieving recommendations

The following figure shows how the Contoso web video application displays the list of recommended videos to the user.

The list of recommended videos to view

The **VideoList** view allows the user to browse a list of recommended videos, and select one for viewing. The following code example shows the HTML that displays the list of recommendations.

HTML

```
<div class="recommendationlist" id="recommendationlist">
    ...
       <label id="labelRecommendation" hidden>Recommended Videos</label>
       <div data-bind="foreach: RecommendationList">
           <div class='recommendationItem'>
               <div>
                   <img class='recommendationItemThumbnail' data-bind="attr: { src:
       ThumbnailUrl, tabindex: TabIndex, videoId: Id }" alt="No Image Available">
               </div>
                   <div class='recommendationItemInfo'>
                       <p data-bind="text: Title"> </p>
                       <span>Length: </span><span data-bind="text: Length"> </span>
                   </div>
           </div>
       </div>
    ...
</div>
```

This code binds a **div** element to the **RecommendationList** property of the **RecommendationListViewModel** class. A thumbnail representing each video is displayed for each item in the **RecommendationList** by using an **img** element, which binds to the **ThumbnailUrl** of the **RecommendationList** item. A **p** and a **span** element are used to display the title and length of the video through appropriate data binding. The **class** attributes of the **DIV** elements are set to use styles from the **videoList.css** file.

The instance of the **RecommendationListViewModel** is bound to the **recommendationlist** DOM element. When the **RecommendationListViewModel** is created the **ajaxGetRecommendationList** function is executed, which retrieves the recommendations. The following code example shows the **ajaxGetRecommendationList** function.

JavaScript

```
function ajaxGetRecommendationList(id) {
    self.ShowProgress(true);
    $("#labelRecommendation").hide();
    var request = {
        type: "GET",
        dataType: "json",
        timeout: Contoso.ajaxTimeOut,
        url: Contoso.apiRoot + "/videos/" + id + "/recommendations",
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
    ...
```

This function creates a URL that specifies the address that will be requested from the web service, which includes the ID of the video to retrieve recommendations for. When AJAX makes the request the **GetRecommendations** method in the **VideosController** class in the Contoso.Api project is invoked. This method returns a collection of **VideoInfoDTO** objects in the response message, from which the **RecommendationList** is populated. For more information about how the **VideosController** class returns a list of recommendations to be displayed to the user, see "Retrieving recommendations."

For information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

Uploading a video

The following figure shows how the Contoso web video application allows the user to upload a video to Media Services for processing. When a new video is uploaded a new asset is created by Media Services, and the asset is uploaded to Azure Storage before the assets details are published to the Content Management System (CMS).

The video details that must be provided prior to upload

The **Upload** view allows the user to upload a video to Media Services, with the following code example showing the relevant HTML.

HTML

```
<div class="video-upload">
    <label class="metadata-label" for=" video-inputfile">Select a video file</label>
    <div id="file-input-parent">
        <input class="file-input" id="file-input" type="file" title="pick a file"
accept="video/*" />
    </div>
    ...
    <div data-bind="visible: showUploadButton">
        <label class="metadata-label" for="submit-button">Upload</label>
        <input id="submit-button" data-bind="click: onSubmit" type="button"
value="Submit" title="upload selected video file"/>
```

```
    </div>
    ...
</div>
```

This code uses an **input** element to allow the user to select a file for upload from the file system, with an **input** element also being used to submit the file for upload. When the **Submit** button is selected the **onSubmit** function is executed which performs validation of user input, before creating a **SubmitObject** that contains the video details entered by the user. Then, the **ajaxGenerateAsset** function is called, which is shown in the following code example.

JavaScript

```
function ajaxGenerateAsset(submitObject) {
    if (cancelUpload) return cancelCompleted(submitObject);
        submitObject.progressStatus = "Calling server to generate asset";
        showProgress(submitObject);
        var request = {
            url: Contoso.apiRoot + "/videos/generateasset",
            type: "GET",
            dataType: "json",
            data: { filename: submitObject.file.name },
            timeout: Contoso.ajaxTimeOut,
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
    ...
```

This function creates a URL that specifies the address that will be requested from the web service. When AJAX makes the request the **GenerateAsset** method in the **VideosController** class in the Contoso.Api project is invoked. This method returns a **VideoAssetDTO** object in the response message, which is handled by the **onAjaxSuccess** function. This function updates the **SubmitObject** with an asset ID, and a shared access signature locator URL for the asset, before calling the **ajaxUploadFile** function. For more information about how the **VideosController** class generates a new Media Services asset from the uploaded file, see "Upload process in the Contoso applications."

The **ajaxUploadFile** function uses the **FileChunker** object to slice the file for upload into a series of 256KB blocks. The **onloadend** event of the **FileReader** object is triggered when a block has been created by the **readNextBlock** function. This event is handled by the **readerOnloadend** function in order to upload the block to Azure Storage.

JavaScript

```
function readerOnloadend(evt) {
    var requestData;
    if (cancelUpload) return cancelCompleted(submitObject);
    if (evt.target.readyState == FileReader.DONE) {
        var blockId = blockIdPrefix + pad(blockIdArray.length, 6);
        blockIdArray.push(btoa(blockId));
        requestData = new Uint8Array(evt.target.result);
        var request = {
            url: submitObject.sasLocator + '&comp=block&blockid=' +
blockIdArray[blockIdArray.length - 1],
            type: "PUT",
            data: requestData,
```

```
            processData: false,
            beforeSend: onAjaxBeforeSend,
            };
            $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
}
```

This function creates a URL that specifies the address that will be requested from the web service. When AJAX makes the request the block is uploaded to the URL in Azure Storage that is specified by the shared access signature locator URL that was earlier returned by the **GenerateAsset** method in the **VideosController** class. When the operation completes the **onAjaxSuccess** function is executed which in turn either calls **readNextBlock** to create the next block to be uploaded, or if chunking has completed, calls the **ajaxCommitBlockList** function which is shown in the following code example.

JavaScript

```
function ajaxCommitBlockList(submitObject, blockIds) {
    ...
    var requestData = '<?xml version="1.0" encoding="utf-8"?><BlockList>';
    for (var i = 0; i < blockIds.length; i++) {
        requestData += '<Latest>' + blockIds[i] + '</Latest>';
    }
    requestData += '</BlockList>';

    var request = {
        url: submitObject.sasLocator + '&comp=blocklist',
        type: "PUT",
        data: requestData,
        contentType: "text/plain; charset=UTF-8",
        crossDomain: true,
        cache: false,
        beforeSend: onAjaxBeforeSend,
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
    ...
```

This function commits the block list to Azure, so that the blocks can be reassembled into the uploaded file. When AJAX makes the request the block list for file is uploaded to the URL in Azure Storage that is specified by the shared access signature locator URL that was earlier returned by the **GenerateAsset** method in the **VideosController** class. When the operation completes the **onAjaxSuccess** function is executed which signals that the blocks have been reassembled and hence that the file has been successfully uploaded. In turn the **onAjaxSuccess** function calls the **ajaxPublish** function, which is shown in the following code example.

JavaScript

```
function ajaxPublish(submitObject) {
    ...
    var videoPublishDTO = {};
    videoPublishDTO.AssetId = submitObject.assetId;
    videoPublishDTO.Title = submitObject.videoTitle;
    videoPublishDTO.Description = submitObject.videoDescription;
    videoPublishDTO.Resolution = submitObject.videoResolution;
    videoPublishDTO.ClipStartTime = submitObject.clipStartTime;
```

```
    videoPublishDTO.ClipEndTime = submitObject.clipEndTime;
    videoPublishDTO.IncludeThumbnails = submitObject.includeThumbnails;
    var request = {
        url: Contoso.apiRoot + "/videos/publish",
        type: "POST",
        dataType: "json",
        data: JSON.stringify(videoPublishDTO),
        contentType: "application/json;charset=utf-8",
        crossDomain: true,
        timeout: Contoso.ajaxTimeOut,
    };
    $.ajax(request).done(onAjaxSuccess).fail(onAjaxError);
```

This function creates a new **videoPublishDTO** object, which contains the details of the video uploaded to Azure Storage. A URL is then created that specifies the address that will be requested from the web service. When AJAX makes the request the **Publish** method in the **VideosController** class in the Contoso.Api project is invoked, with the **videoPublishDTO** object being passed as a parameter to this method. This method saves the uploaded video details to the CMS database and starts the encoding process on the uploaded asset. For more information about how the **Publish** method in the **VideosController** class, see "Upload process in the Contoso applications."

For information about how different methods are invoked in the Contoso web service, see "Appendix A – The Contoso Web Service."

Understanding the Contoso video Windows Store application

The Contoso Windows Store video application is primarily implemented in the Contoso.WindowsStore project in the solution provided with this guide. However, some of the business logic is implemented in the Contoso.UILogic project.

The Contoso.WindowsStore project contains the following folders.

Folder	Description
Assets	Contains images for the splash screen, tile, and other Windows Store application required images.
Behaviors	Contains the **NavigateWithEventArgsToPageAction** behavior that is exposed to view classes.
Controls	Contains the **AutoRotatingGridView** control.
Converters	Contains data converters such as the **SecondsToTimeFormatConverter**.
Events	Contains the **SharePersonalInformationChangedEvent** class.
Helpers	Contains the **VisualTreeUtilities** class that can be used to walk the XAML visual tree.
Models	Contains the **IncrementalLoadingVideoCollection** class that is used to provide incremental loading support to the application.

Services	Contains interfaces and classes that implement services that are provided to the application, such as the **FilePickerService** and **CaptureService** classes.
Strings	Contains resource strings used by the project, with subfolders for each supported locale.
ViewModels	Contains the Windows Runtime specific application logic that is exposed to XAML controls.
Views	Contains the pages and Flyout for the application.

The Contoso.UILogic project contains classes and other resources that are shared with the Windows Store and Windows Phone projects. Placing these classes into a separate assembly enables reuse by the two applications, and provides a simple mechanism for ensuring that view models are independent from their corresponding views. The project contains the following folders.

Folder	Description
Common	Contains the **CaptureSource** enumeration and constants that are used by the Windows Store and Windows Phone projects, such as the address of the web service.
Models	Contains the entities that are used by view model classes.
Services	Contains interfaces and classes that implement services that are provided to the application, such as the **VideoService** class.
ViewModels	Contains the shared application logic that is exposed to XAML controls in the Windows Store and Windows Phone projects.

> The main chapters in this guide explain how the Contoso video Windows Store application works, and how it interacts with the Contoso web service to upload, encode, and consume videos.

For information about the frameworks with which you can delivery adaptive streaming content to Windows applications, see "Developing Azure Media Services Client Applications."

Understanding the Contoso video Windows Phone application

The Contoso Windows Phone video application is primarily implemented in the Contoso.WindowsPhone project in the solution provided with this guide. However, some of the business logic is implemented in the Contoso.UILogic project.

The Contoso.WindowsPhone project contains the following folders.

Folder	Description
Assets	Contains images for the tiles, and other Windows Phone application required images.

Behaviors	Contains behaviors that are exposed to view classes in order to respond to events from view model classes.
Controls	Contains the **CaptureSourceViewport** control.
Converters	Contains data converters such as the **BooleanToVisibilityConverter**.
DesignViewModels	Contains design-time view model classes that are used to display sample data in the visual designer.
Resources	Contains resource strings used by the project.
Services	Contains interfaces and classes that implement services that are provided to the application, such as the **FilePickerService** and **CaptureService** classes.
Styles	Contains a resource dictionary that defines the application styles used by the app.
ViewModels	Contains the Windows Phone specific application logic that is exposed to XAML controls.
Views	Contains the pages for the application.

> The Contoso video Windows Phone application shares much of its application logic with the Contoso video Windows Store application. Therefore, an understanding of how the Windows Phone application works can be gained by reading the main chapters in this guide, which explain how the Contoso video Windows Store application works, and how it interacts with the Contoso web service to upload, encode, and consume videos.

For information about the structure of the Contoso.UILogic project see the second table in "Understanding the Contoso video Windows Store application."

For information about the frameworks with which you can deliver adaptive streaming content to Windows Phone applications, see "Developing Azure Media Services Client Applications."

Understanding the other Contoso video applications

> The Contoso video applications for iOS and Android can be downloaded from the CodePlex site for this guide (https://wamsg.codeplex.com/).

For information about the frameworks with which you can deliver adaptive streaming content to iOS and Android devices, see "Developing Azure Media Services Client Applications."

More information

- For information about the iOS and Android Contoso video applications, see the Azure Media Services Guidance Community site on CodePlex.
- For information about the Microsoft SDKs and Player Frameworks that allow you to create client applications that can consume streaming media from Media Services, see "Developing Azure Media Services Client Applications" on MSDN.

Bibliography

You can download the sample code described in this book from the Microsoft Download Center at http://aka.ms/amsg-code.

Chapter 1 – Introduction to Windows Azure Media Services

- The article "Managing assets across multiple storage accounts in Windows Azure Media Services and defining load balancing strategy" is available on a blog site.
- You can find information about content protection at "Protecting Assets with Microsoft PlayReady".
- You can find more information about the SDKs and Player Frameworks that allow you to create client applications that can consume streaming media from Media Services at "Developing Windows Azure Media Services Client Applications".

Chapter 2 – The Contoso Scenario

- You can find the *"General Guidelines and Limitations (Windows Azure SQL Database)"* page on MSDN.
- The patterns & practices guide "Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence" is available from MSDN.
- You can find information about the Entity Framework in the Data Developer Center, available on MSDN.
- The Repository pattern is described on MSDN.
- The patterns & practices guide "Developing a Windows Store business app using C#, XAML, and Prism for the Windows Runtime" is available on MSDN.
- The patterns & practices guide *"Developing Multi-tenant Applications for the Cloud, 3rd Edition"* is available on MSDN.
- For information about Prism for the Windows Runtime, see "Developing a Windows Store business app using C#, XAML, and Prism for the Windows Runtime".
- For information about using Unity, see "Unity Container".
- For information on how to bootstrap a Windows Store application that uses Prism for the Windows Runtime, see "Bootstrapping an MVVM Windows Store app Quickstart using C#, XAML, and Prism".

Chapter 3 – Uploading Media

- The page, "Ingesting Assets with the Media Services REST API", describing how to ingest assets into Media Services using the REST API, is available on MSDN.
- You can find the page, "Ingesting Assets in Bulk with the REST API", describing how to use the REST API to ingest assets into Media Services in bulk, on MSDN.
- For information about high speed ingest technology, see "Uploading Media" on MSDN.
- You can find the page, "Managing Media Services Assets across Multiple Storage Accounts", on MSDN.
- For information about ingesting assets in bulk, see the page "Ingesting Assets in Bulk with the Media Services SDK for .NET" on MSDN.
- For information about creating a Media Services account and associate it with a storage account, see "How to Create a Media Services Account" on MSDN.
- The page, "Setup for Development on the Media Services SDK for .NET", describes how to set up a Visual Studio project for Media Services development, is available on MSDN.
- For more information about lazy initialization, see "Lazy Initialization" on MSDN.
- For a detailed description of how to encrypt configuration information see "Encrypting Configuration Information Using Protected Configuration" on MSDN.
- For information about how to connect to a web service from a Windows Store application, see "Connecting to an HTTP server using Windows.Web.Http.HttpClient" on MSDN.

Chapter 4 – Encoding and Processing Media

- For information about additional support codecs and filters in Media Services, see "Codec Objects" and "DirectShow Filters" on MSDN.
- For information about how to configure the Windows Azure Media Packager, see "Task Preset for Windows Azure Media Packager" on MSDN.
- You can download the "IIS Smooth Streaming Technical Overview" paper from the Microsoft Download Center.
- The page, "Windows Azure Management Portal", explains tasks that can be accomplished by using the Windows Azure Management Portal, and is available on MSDN.
- You can find the article "How to Scale a Media Service" on MSDN.
- The article "Windows Azure Queues and Windows Azure Service Bus Queues – Compared and Contrasted" is available on MSDN.
- For information about customizing the settings of a thumbnail file see "Task Preset for Thumbnail Generation" on MSDN.

- The page, "Percent-encoding" explains the mechanism for encoding URI information, and is available on Wikipedia.

Chapter 5 – Delivering and Consuming Media

- For information about downloading a file from storage, see "Create a SAS Locator to On Demand Content" on MSDN.
- The page, "How to Manage Origins in a Media Services Account", which explains how to add multiple streaming origins to your account and how to configure the origins, is available on MSDN.
- You can find the article "How to Scale a Media Service" on MSDN.
- You can find information about content protection at "Microsoft PlayReady".
- For information about using dynamic packaging to deliver MPEG DASH content encrypted with PlayReadY DRM, see "Task Preset for Windows Azure Media Encryptor".
- For information about the Microsoft Player Framework see "Player Framework by Microsoft" on CodePlex.
- You can find the page, "Smooth Streaming Client SDK", which lists the functionality supported in the Smooth Streaming Client SDK, available at the Visual Studio Gallery.

Appendix A – The Contoso Web Service

- The page "Attribute Routing in Web API 2" is available on the ASP.NET website.
- You can find details about AutoMapper on GitHub, at https://github.com/AutoMapper.
- The Repository pattern is described in "Patterns of Enterprise Application Architecture" on Martin Fowler's website.
- Information describing how to use the Fluent API in the Entity Framework 6.0 to map types to database tables is available on MSDN at "Configuring/Mapping Properties and Types with the Fluent API".
- Information describing how to configure relationships between entities by using the Fluent API is available at "Configuring Relationships with the Fluent API".
- You can download the Unity Application Block from MSDN at http://msdn.com/unity.

Appendix B – Windows Azure Media Encoder Presets

- Media Services Encoder System Presets
- Process Assets with the Media Services SDK for .NET
- Encoding your media with Dolby Digital Plus
- Task Preset for Windows Azure Media Encryptor

- Task Preset for Windows Azure Media Packager
- Task Preset for Thumbnail Generation

Appendix C – Understanding the Contoso Video Applications

- For information about the iOS and Android Contoso video applications, see "patterns & practices: Windows Azure Media Services Guidance"
- For information about the Microsoft SDKs and Player Frameworks that allow you to create client applications that can consume streaming media from Media Services, see "Developing Windows Azure Media Services Client Applications" on MSDN.
