Range-Specific Requests in ASP.NET

This article provides an overview of how range-specific HTTP requests work and then shows how to implement support for such requests in ASP.NET through the use of an HTTP Handler.

Contents [hide]

1 Introduction
2 The Nuts and Bolts of Range-Specific HTTP Requests
3 Range-Specific Requests And Videos On The iPhone and iPod Touch
4 Using the RangeRequestHandlerBase HTTP Handler
5 Creating a Derived Class
6 Registering the HTTP Handler
7 Adding Authorization Checks
8 Debugging a Range-Specific Request
9 Conclusion
10 Works Consulted

Introduction

The HTTP protocol defines the syntax and semantics of how a client, such as a browser, issues a request and how a web server returns the requested content. Typically, a client requests the entire contents of a file, which the web server returns in its response. Alternatively, a client may request a particular range of a file or even a particular set of ranges. Such range requests allow for a client to resume an interrupted download. For example, imagine that while downloading a very large file your browser crashes, or you lose your Internet connection. Once the browser is restarted or Internet connectivity is restored, the browser can request the large file starting just beyond the last-downloaded byte rather than having to download the entire file from the beginning. Range requests are also used by certain mobile devices. Apple's media player on the iPhone and iPod Touch use range requests to download video files.

Web servers such as IIS have long supported range-specific requests. Therefore, if you are serving videos, ZIP files, or other large files using IIS, then your visitors automatically get the benefits of range requests, which include the ability to pause downloads, resume interrupted downloads, and view videos on Apple's devices. But what about ASP.NET? Websites that impose limits as to who can download content, or websites that dynamically generate downloadable content, typically do so through an ASP.NET web page or HTTP Handler. Unfortunately, ASP.NET does not support range-specific requests out of the box.

The good news is that with a fair amount of code it is possible to build in support for range-specific HTTP requests. This article provides an overview of how range-specific HTTP requests work and then shows how to implement support for such requests in ASP.NET through the use of an HTTP Handler. This HTTP Handler, along with a working demo application, is available for download.

The Nuts and Bolts of Range-Specific HTTP Requests

Before we examine how to implement range-specific HTTP requests in ASP.NET, it's worthwhile to take a few moments to dissect how range-specific HTTP requests work at the protocol level. Consider the following use case: a visitor on your site is getting ready to download a (roughly) 50 megabyte ZIP file. When they click the download link the browser sends an HTTP request to the server similar to the following:

Listing 1: An HTTP Request For DancingHampsters.zip

```
1. GET /downloads/DancingHamsters.zip HTTP/1.1
2. Accept-Language: en/us,en
3. User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.1.1)
```

The first line indicates the HTTP method (`GET`), the resource being requested (`/downloads/DancingHamseters.zip`), and the HTTP protocol (HTTP/1.1). The subsequent lines are request headers. In this case there are two headers - `Accept-Language` and `User-Agent` - but there can certainly be more. Each header is a name/value pair and is formatted with the header name followed by a colon (:) followed by the header value, as in `Header: value`.

The server would respond with an HTTP response that includes the content of the requested ZIP file.

Listing 2: The HTTP Response

```
1. HTTP/1.1 200 OK
2. Content-Length: 54216828
3. Content-Type: application/x-zip-compressed
4. Last-Modified: Fri, 24 Jul 2009 21:45:22 GMT
5. ETag: "mTkR3PTZIofQXrAY3ZkNzA=="
6. Accept-Ranges: bytes
7. Binary content of DancingHampsters.zip...
```

The HTTP response starts with the HTTP version number (HTTP/1.1) followed by a status code and message (`200` and `OK`). Next come the response headers, followed by the actual content of the requested resource.

There are a number of important response headers that merit discussion. The first header, `Content-Length`, specifies the total size of the requested file, while `Content-Type` indicates the type of file being downloaded.

The next two headers - `Last-Modified` and `ETag` - are used to detect a change in content and are instrumental in range-specific HTTP requests. The `Last-Modified` header indicates the date and time the requested resource was last modified. `ETag` defines an entity tag, which is a string that uniquely identifies a version of a file on the server. In a bit, we'll explore how these two headers' values are used by the client when performing a range-specific request.

The final header, `Accept-Ranges`, informs the browser that it can request particular byte ranges of this file, if needed.

Because the `DancingHampsters.zip` file is 50 MB in size, it will take some time for the client to download the file. Imagine that after downloading the first 500,000 bytes the user shuts down their browser and goes home for the day. If the browser supports resumable downloads, the next time the user opens her browser it will send a request for the `DancingHampsters.zip` file starting at the 500,001th byte. That is, rather than trying to download the file from the beginning, the browser picks up where it left off. Such a range-specific HTTP request might look like the following:

Listing 3: A Range-Specific HTTP Request For DancingHampsters.zip

```
1. GET /downloads/DancingHamsters.zip HTTP/1.1
2. Accept-Language: en/us,en
3. User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.1.1)
4. If-Match: "mTkR3PTZIofQXrAY3ZkNzA=="
5. Range: bytes=500001-
```

This request includes two new HTTP headers that weren't present in the initial request: `If-Match` and `Range`. Let's talk about the `Range` header first, as it's more straightforward. The `Range` header indicates the byte range being requested for download. This header may include a starting and ending byte, such as `Range: bytes=500001-500500`. In the example above the `Range` header includes just a starting byte, indicating to the server to return contents from the starting byte to the end. Similarly, this header can be used to request the last *n* bytes of the content; for example, `Range: bytes=-10000` requests the last 10,000 bytes.

The `If-Match` header echoes the `ETag` value sent from the server's earlier response and is used as a check to ensure that the file requested by the browser now is the same file that was requested earlier. If the `DancingHampsters.zip` file was modified between the time the user's download was interrupted and when it was resumed then the ZIP file's current entity tag would differ from the `ETag` value sent in the `If-Match` header. The server is responsible for checking to see if there is such a mismatch; if so, the server should respond with a `412 Precondition Failed` status and the browser, upon receiving this message, would inform the user the download could not be continued.

Assuming that the `DancingHampsters.zip` file has not changed since it was first requested, the server responds with the requested range of content using an HTTP response similar to the following:

Listing 4: The HTTP Response

```
1. HTTP/1.1 206 OK
2. Content-Range: bytes 500001-54216827/54216828
3. Content-Length: 53716827
4. Content-Type: application/x-zip-compressed
5. Last-Modified: Fri, 24 Jul 2009 21:45:22 GMT
6. ETag: "mTkR3PTZIofQXrAY3ZkNzA=="
```

```
7. Accept-Ranges: bytes
8. Binary content of DancingHampsters.zip from byte 500,001 onwards...
```

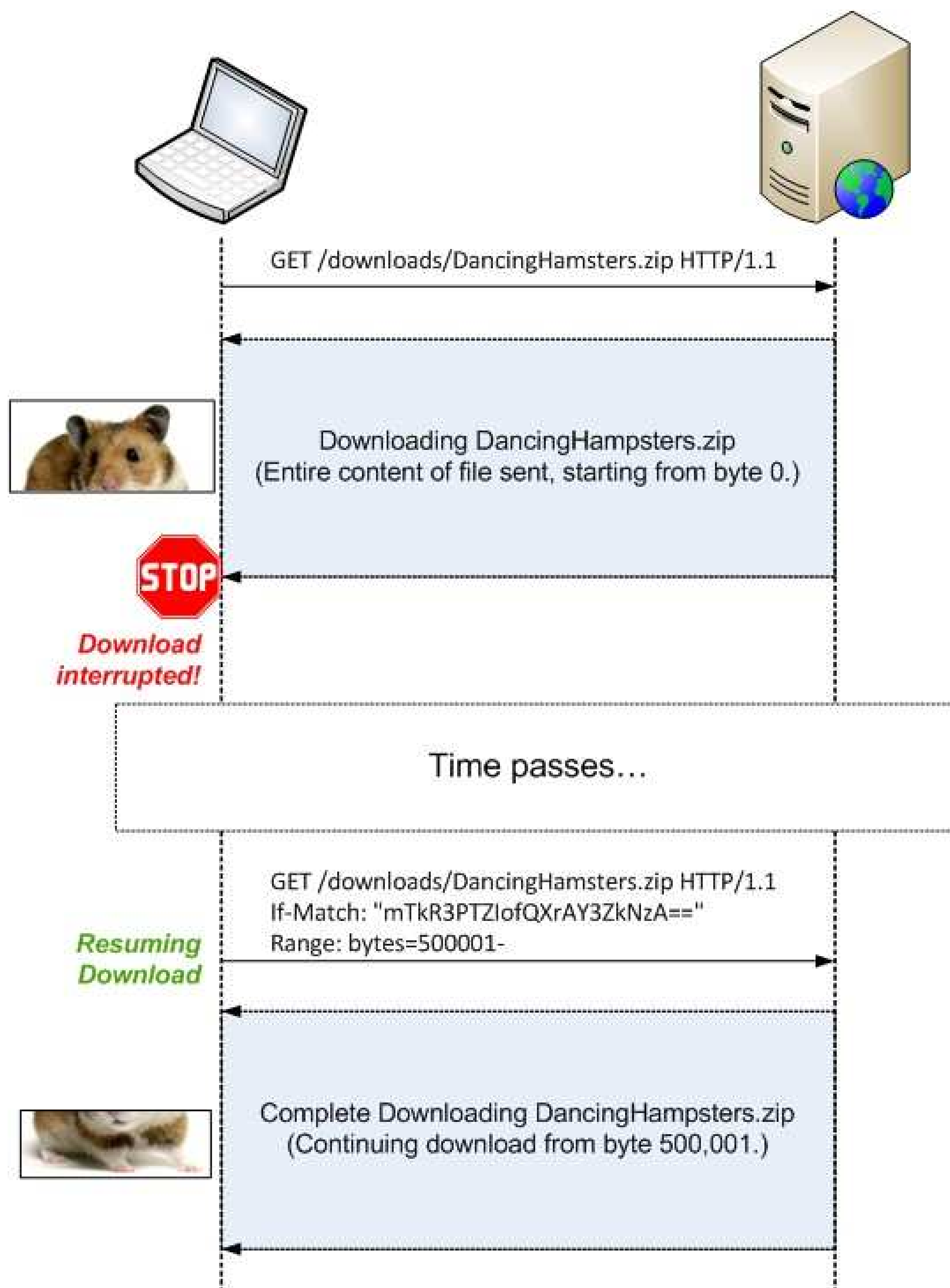
A range-specific response uses a status code 206 instead of 200, which indicates that only a portion of the content is being returned. The `Content-Range` header notes the range of bytes returned along with the total length, while the `Content-Length` header specifies the amount of data returned in this response.

Note:

The description of how range-specific HTTP requests work and the request/response messages we examined are a valid example, but only offer a glimpse into the full functionality defined in the HTTP/1.1 protocol. For example, in addition to (or instead of) sending an `If-Match` header, browsers may send other headers, including `If-None-Match`, `If-Modified-Since`, `If-Unmodified-Since`, and `If-Range`. Furthermore, a browser may request a set of range offsets in the Range header, such as `Range: bytes=10000-49999,500000-999999,-250000`, which would request: the 40,000 bytes between bytes 10,000 and 49,999; the 500,000 bytes between bytes 500,000 and 999,999; and the last 250,000 bytes of the file. While the code available for download handles these more intricate scenarios, we won't explore them in this article.

Figure 1 illustrates the workflow we just discussed.

Figure 1: The Browser Resumes Downloading DancingHampsters.zip



Range-Specific Requests And Videos On The iPhone and iPod Touch

My introduction to range-specific requests came when working on a website that served video content to its members. Initially, all videos were converted into a Flash video format and displayed in a web page much like the videos at YouTube. However, this prevented iPhone and iPod Touch users from accessing the video content from their devices because Apple's devices do not support Flash. To show videos on an Apple device you need to convert them to MPEG-4 videos (`.mp4`).

After converting the raw video to MPEG-4 and uploading them to the web server, I powered up an iPhone, launched Safari, and punched in the URL to the test video. Within a few seconds the video started playing. Mission accomplished!

Almost. There was still the important issue of protecting the videos from unauthorized access. Only members in good standing were allowed to view videos, and certain videos were only accessible to paying members. To complicate things further, the site restricted how many videos a member could view in a 24 hour period, and that number could vary on a user-by-user basis.

To protect these videos from being viewed by anyone who knew the URL to them, I created an HTTP Handler and configured it to handle all incoming requests to `.mp4` files. When an MP4 file is requested the HTTP Handler calls down to the business logic layer to determine if the user requesting the video is authorized to view it. If so, the HTTP Handler returns the contents of the video file to the requestor using the `Response.TransmitFile` method. If the user is not authorized then an alternative video that explains why the requested video could not be viewed is displayed instead.

After implementing this HTTP Handler I returned to my iPhone to repeat the test. When I entered the URL to the video, I was not shown the video but was instead greeted with an error message. After a bit of research I came across an article titled Content Delivery for Mobile Devices that noted that Apple's iPhone (and iPod Touch) use range-specific requests for audio and video files. First, the Safari web browser makes a normal `GET` request to the file. On response, the media player is opened and starts by requesting the first two bytes of the audio or video file to ensure that the web server supports range-specific requests.

IIS, as well as most other modern web servers, support range-specific requests, which is why the videos played on the iPhone prior to the addition of the HTTP Handler. However, once I added the HTTP Handler and configured the web server to route requests for MP4 files to it, range-specific requests were no long supported out of the box. When the output stream is generated from ASP.NET then it is up to the developer to implement the logic to process range-specific requests.

After a bit more searching I found a helpful article by Alexander Schaaf titled Tracking and Resuming Large File Downloads in ASP.NET that explained how range-specific requests worked and offered a Visual Basic HTTP Handler for implementing them in ASP.NET 1.x. I took the code shared by Alexander, refactored it, added some new functionality, utilized some of the framework and language features added since the .NET 1.x days, and ported the code to C#. This HTTP Handler, which I've named `RangeRequestHandlerBase`, is available for download along with a demo application that shows how to use it.

After creating the `RangeRequestHandlerBase` class, I went back to my HTTP Handler for serving MP4 files and had it derive from `RangeRequestHandlerBase`. And voila! With that one little change I was now able to serve MP4 videos from ASP.NET and correctly view them in an iPhone or iPod Touch.

The remainder of this article shows how to create an HTTP Handler that extends the `RangeRequestHandlerBase` class. This article does not explore the depths of the `RangeRequestHandlerBase` class source code. If you are interested in examining this class's functionality in depth I suggest that you start by reading Alexander's article, Tracking and Resuming Large File Downloads in ASP.NET, and then read through the `RangeRequestHandlerBase` class source code, which is generously commented.

Using the RangeRequestHandlerBase HTTP Handler

The `RangeRequestHandlerBase` HTTP Handler may be used in any project where you want or need to support range-specific requests for content that is served from ASP.NET. The content being served may be generated at runtime - such as packing requested content into a ZIP file and serving it on the fly - or it may be content that exists on the web server's file system but is only accessible to authorized users. Supporting range-specific requests allows for interrupted downloads to be seamlessly resumed and enables visitors to pause and continue downloads. And range-specific requests must be supported for audio and video content served to the iPhone or iPod Touch.

Note:

If you use ASP.NET to prevent unauthorized access to static files and are using IIS 7, you may be able to serve the content through IIS using IIS 7's URL authorization rules feature. IIS 7's URL authorization rules allow you to allow or deny access to content on the web server through rules defined in an XML file, similar to how ASP.NET's URL authorization works. And by serving the content from IIS you won't need to worry about range-specific requests, as IIS supports them natively.

Keep in mind that URL authorization rules will only work for scenarios where there are simple authorization rules dictating access to the protected files. For example, if you only permit authenticated users to view content or base viewing rights on the visitor's role, then URL authorization rules may fit the bill. If the rules are more complex then you may need to have ASP.NET handle the request.

Creating a Derived Class

The `RangeRequestHandlerBase` class is `abstract`, meaning that you cannot use it directly. Instead, you must create a class that extends it. The `RangeRequestHandlerBase` class contains the logic for supporting range-specific requests for a given file, but in order to do its job it needs, at minimum, information about the file that is to be served. This information is what you must specify from the derived class. The derived class may optionally specify the file's entity tag and MIME type.

The download available with this article includes two such derived classes: MP4DownloadHandler, for serving MP4 files; and ZIPDownloadHandler, for serving ZIP files. The code for the MP4DownloadHandler class follows.

Listing 5: The MP4DownloadHandler Class

```
01. public class MP4DownloadHandler : RangeRequestHandlerBase
02. {
03.     public override FileInfo GetRequestedFileInfo(HttpContext context)
04.     {
05.         if (File.Exists(context.Request.PhysicalPath))
06.             return new FileInfo(context.Request.PhysicalPath);
07.         else
08.             return null;
09.     }
10.     public override string GetRequestedFileMimeType(HttpContext context)
11.     {
12.         return "video/mp4";
13.     }
14. }
```

As you can see, the MP4DownloadHandler class extends RangeRequestHandlerBase. Any class that extends RangeRequestHandlerBase must provide an implementation for the GetRequestedFileInfo method, which returns a FileInfo object for the requested file. In this example the MP4 file exists on the web server's file system so the GetRequestedFileInfo method simply returns a new FileInfo object for the requested file. If the file is being generated dynamically then you would need to first generate it (if it did not already exist), then save it to the file system in some temporary location, and finally return a FileInfo object for that new file.

The GetRequestedFileMimeType should be overridden and return the requested file's appropriate MIME type. If you do not override this method then the RangeRequestHandlerBase class uses a MIME type of application/octet-stream. See the Internet Assigned Numbers Authority (IANA)'s MIME Media Types page for a list of registered MIME types.

The MP4DownloadHandler class uses the entity tag generated by the RangeRequestHandlerBase class, which is the MD5 hash of the requested file's full name appended with the requested file's last write time. You can provide a custom entity tag value if you'd like by overriding the GetRequestedFileEntityTag method.

Registering the HTTP Handler

In order to have the MP4DownloadHandler class handle requests for MP4 files we need to register the HTTP Module in Web.config.

If you are using IIS 7 with the Classic pipeline or an earlier version of IIS, locate the <httpHandlers> section in the <system.web> element and add the following configuration:

```
1. <system.web>
2.     ...
3.     <httpHandlers>
4.         ...
5.         <add verb="*" path="*.mp4" type="MP4DownloadHandler" />
6.     </httpHandlers>
7. </system.web>
```

You will also need to configure IIS to route all requests for .mp4 files to the ASP.NET ISAPI Extension, otherwise IIS will handle requests for MP4 files and your HTTP Handler will never be executed. See How To: Configure an HTTP Handler Extension in IIS for more information.

If you are using IIS 7's Integrated pipeline then you need to register the HTTP Handler in the <system.webServer> element's <handlers> section like so:

```
1. <system.webServer>
2.     ...
3.     <handlers>
4.         ...
5.         <add name="MP4DownloadHandler" verb="*" path="*.mp4" type="MP4DownloadHandler"
6.             preCondition="integratedMode"/>
7.     </handlers>
```

```
8. </system.webServer>
```

And that's all you have to do to support range-specific requests for content served by ASP.NET!

To summarize:

1. Create a class that extends `RangeRequestHandlerBase` and override, at minimum, the `GetRequestedFileInfo` method.
2. Register the HTTP Handler in Web.config and configure IIS to map the appropriate file extension to the ASP.NET ISAPI Extension (if needed).

And voila, you can serve your ASP.NET-generated content to Apple devices and support resumable downloads.

Adding Authorization Checks

The `MP4DownloadHandler` HTTP Handler does not include any sort of authorization check and therefore allows anyone to access the video. The good news is that adding authorization checks is a walk in the park - simply override the `RangeRequestHandlerBase` class's `CheckAuthorizationRules` method and return a Boolean value indicating whether the user is authorized to view the requested content.

The following code snippet shows a `CheckAuthorizationRules` method that restricts access of MP4 files to users the "Admin" role. Of course, more intricate authorization rules could be implemented here. For example, there might be a database table that indicates what users can view what videos that the `CheckAuthorizationRules` method would check to determine if the current request was authorized.

```
01. public class MP4DownloadHandler : RangeRequestHandlerBase
02. {
03.     protected override bool CheckAuthorizationRules(HttpContext context)
04.     {
05.         // Only allow users in the "Admin" role to view videos
06.         if (context.User.IsInRole("Admin") == false)
07.         {
08.             context.Response.StatusCode = 200;
09.             base.AddHeader(context.Response, "Content-Type", "text/html");
10.             context.Response.Write("<h1>Only Admins are allowed to view videos...</h1>");
11.             return false;
12.         }
13.         return true;
14.     }
15.     ...
16. }
```

Debugging a Range-Specific Request

If you are interested in learning more about how range-specific requests work, or have bumped into a problem where you think the range-specific request logic is not working or is not returning the expected data, then you should take advantage of the logging and debugging functionality baked into the `RangeRequestHandlerBase` class.

If the application is running in Debug mode then the `RangeRequestHandlerBase` class automatically logs all incoming request headers and outgoing response headers to a file named `ResumableFileDownloadHandler.log`. This log file shows the exact headers exchanged when making a range-specific request; this information is priceless when it comes to understanding how these requests work.

When a non-range-specific request arrives the `RangeRequestHandlerBase` class uses `Response.TransmitFile` to send the entire contents of the file using the most efficient method. When a range-specific request arrives the `RangeRequestHandlerBase` class opens the requested file and starts streaming the requested bytes to the client in 10 kilobyte chunks (although this block size can be customized via the `BufferSize` property).

One difficulty of testing range-specific requests is that most tests are done in a Local Area Network (LAN) environment where files even hundreds of megabytes in size can be downloaded nearly instantly. To help testing I made it so that when in Debug mode non-range-specific requests are sent 10 KB at a time (rather than via `Response.TransmitFile`). Moreover, I added a configurable delay between each 10 KB chunk, which applies to both non-range-specific and range-specific requests. This delay is measured in milliseconds and is specified via the `DEBUGGING_SLEEP_TIME` constant. This value is set to 0 by default and should be returned to 0 when done testing. But while testing set it to, say, 500 to add a half second delay between

chunks. This allows ample time to test interrupting, pausing, and resuming a download of files in a LAN environment.

Conclusion

The HTTP/1.1 protocol added support for range-specific requests, which allows a client to optionally request one or more sections of a file. Range-specific requests are commonly used by browsers and download managers to resume interrupted downloads. They are also used to play audio and video files on Apple's iPhone and iPod Touch devices.

Microsoft's IIS web server supports range-specific requests out of the box. However, content generated from ASP.NET does not; it is up to the developer to add such support, if needed. This article examined how range-specific requests work and introduced an HTTP Handler base class named `RangeRequestHandlerBase` that will get you started down the path of implementing range-specific requests.

Happy Programming!

Works Consulted

- Content Delivery for Mobile Devices
- Hypertext Transfer Protocol - HTTP/1.1 (RFC 2616)
- PRB: Response.WriteFile Cannot Download a Large File
- Tracking and Resuming Large File Downloads in ASP.NET

Original Url: http://dotnetslackers.com/articles/aspnet/Range-Specific-Requests-in-ASP-NET.aspx