“Files” Web Service

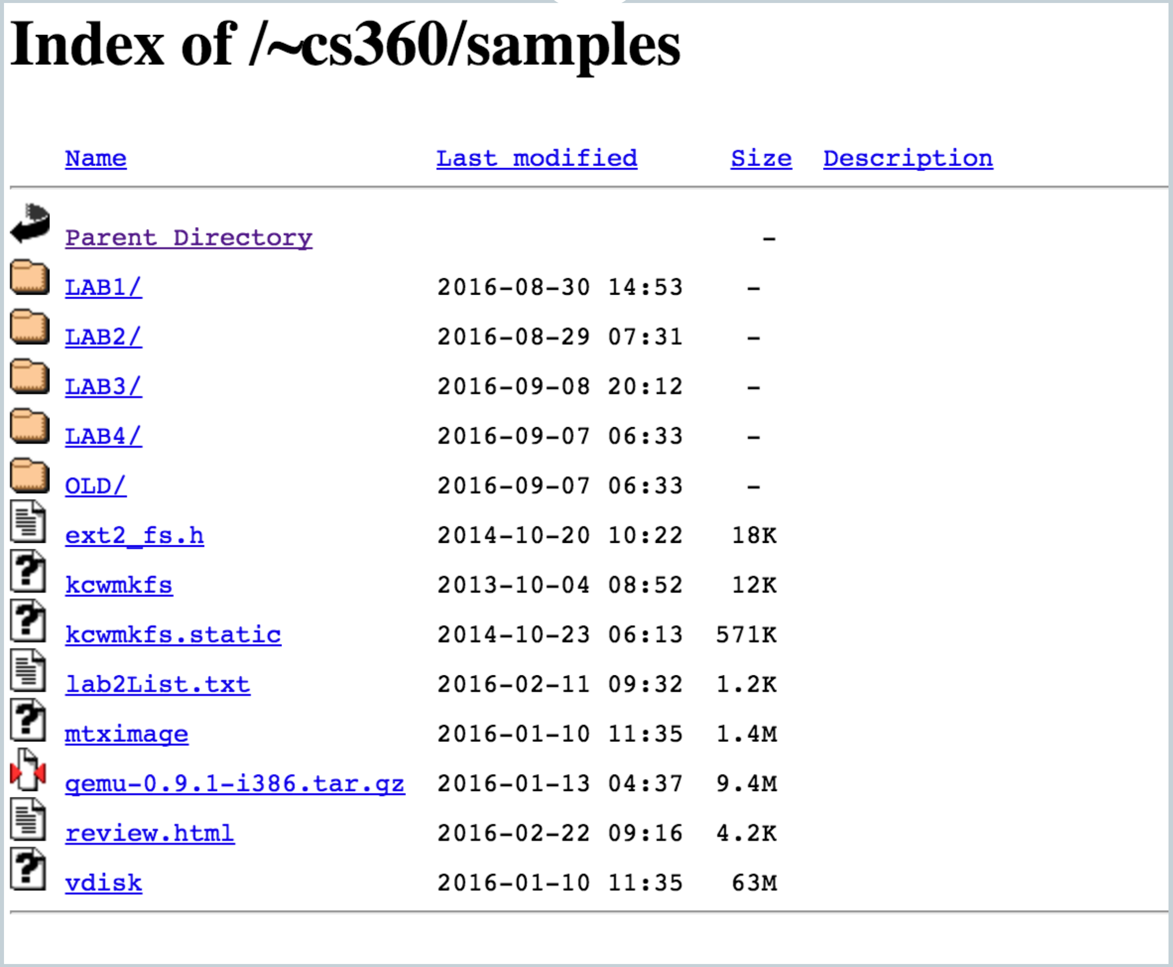
BY EVAN OLDS

CPT S 422

Description

- One of the most basic web services/apps we could make on top of our server core is just one that hosts files from a directory on the machine
- Provides listing of available files as links on the web page
- User can download files through the browser
  - No upload for now, only download

You've probably seen it before…



Index of /~cs360/samples

```
     Name                     Last modified      Size  Description
-----------------------------------------------------------------
     Parent Directory                              -
     LAB1/                    2016-08-30 14:53    -
     LAB2/                    2016-08-29 07:31    -
     LAB3/                    2016-09-08 20:12    -
     LAB4/                    2016-09-07 06:33    -
     OLD/                     2016-09-07 06:33    -
     ext2_fs.h                2014-10-20 10:22   18K
     kcwmkfs                  2013-10-04 08:52   12K
     kcwmkfs.static           2014-10-23 06:13  571K
     lab2List.txt             2016-02-11 09:32  1.2K
     mtximage                 2016-01-10 11:35  1.4M
     qemu-0.9.1-i386.tar.gz   2016-01-13 04:37  9.4M
     review.html              2016-02-22 09:16  4.2K
     vdisk                    2016-01-10 11:35   63M
-----------------------------------------------------------------
```

Why Files Web Service?

- Once implemented, we have what's a lot like a "normal" web server
- Can host pages and files; users can get content from our web site hosted by our server
- Ideas for usage
  - Run server code on Raspberry Pi (or some other low powered machine) connected to an external hard drive -> share drive contents on home network
  - Write "apps" in JavaScript. Server can host pages with these apps and they can be run by any device with a web browser that can connect to your server

Needed Elements

- FilesWebService that inherits from the WebService base class
  - Ability to specify which folder to share (most likely an input to the constructor)
  - Ability to produce listings of files from that folder (as an HTML page)
  - Ability to send files in the body of a response when a file link is clicked
  - From the above 2, the ability to distinguish a folder URL from a file URL
  - What else?

Needed Elements

- FilesWebService that inherits from the WebService base class
  - What else? Security
  - If hosted folder is /home/username/Desktop/myShare, don’t want anything outside of that directory to be shared
    - But we do want to share all subfolders within it
- For simplicity we will not implement user accounts and authentication. Anyone that can connect to our server should be able to see and download files (but again, not any files from outside of that folder)

Needed Elements

- Mapping scheme from URI to content
  - Suppose we're hosting the folder /home/username/Desktop/myShare (referred to below as "../myShare")

Browser URL	**Request-Target (URI) in first line of HTTP request**	**Result**
http://localhost:4220/	/	Dynamically generated HTML page with a listing of files+folders in the ../myShare folder
http://localhost:4220/blah.txt	/blah.txt	HTTP response with the contents of ../myShare/blah.txt OR 404 if no such file exists
http://localhost:4220/subdir1	/subdir1	HTML page with listing of files+folders in the ../myShare/subdir1 folder OR 404 if no such folder exists

Mapping Scheme

- At this point most of you could fill in the gaps on the remaining bits that are needed and start implementing the service
- But inevitably you'd come across a few potentially unexpected things
  - Some of you will recognize these, others will not
- Also, we want to make sure we discuss a few security considerations
  - Recall that service should not host content outside of shared folder
- Consider more examples on the next slide

Spaces in URL

- Suppose we do a request to: http://localhost:4220/my file.txt
- There's a space in that file name
- Suppose such a file does exist in the shared folder
- Think critically:
  - Why is this a potential issue?
  - If we tried to put an entry in the table from 2 slides back, what would be the request-target on the first line of the HTTP request?
  - (discuss the above 2 in class, then…)
  - Let's see what the browser does with that URL (demo in class)

Spaces in URL

- It's a problem because the first line of an HTTP request is of the form: METHOD SP REQUEST-TARGET SP VERSION
  - (recall that SP in the specification means single space character)
- URI parsing logic in server expects 2 spaces on that line
- There would be more than 2 if REQUEST-TARGET had spaces
- It turns out spaces aren't allowed in the REQUEST-TARGET
- Demo with browser gives us a hint as to how this is resolved

Percent Encoding

- Wikipedia article
- %20 is space
  - “Percent encoding” is the name of this scheme, so the % character makes sense
  - But why 20?
  - It’s NOT chosen out of nowhere, there’s a simple reason why 20 corresponds to a space character
  - Think critically: What is the reason? (discuss in class)

Percent Encoding

- Wikipedia article
- %20 is space
  - "Percent encoding" is the name of this scheme, so the % character makes sense
  - But why 20?
  - It's NOT chosen out of nowhere, there's a simple reason why 20 corresponds to a space character
  - Think critically: What is the reason?
- Hint: ASCII value for space character is 32
  - You might be thinking: How is that even a hint? 32 != 20
  - (discuss more in class)

Percent Encoding

- Percent encoding uses hexadecimal
  - ASCII value for space is 32 in base 10, that's 20 in base 16
    - 2 * 16 + 0 = 32
- Browser will percent encode certain URL strings if necessary
- Let's see what the browser does for:
  - http://localhost:4220/name with 3 spaces.txt
  - http://localhost:4220/exclamation!
  - http://localhost:4220/C# or C++?

Percent Encoding

- Wikipedia article mentions that reserved characters for percent encoding include: !#$&'()*+,/:;=?@[]
- Let's see what the browser does for:
  - http://localhost:4220/name with 3 spaces.txt
  - http://localhost:4220/exclamation!
  - http://localhost:4220/C# or C++?
  - (will need to look at what's getting sent to the server and not just what's happening in the URL text box in the UI)

Percent Encoding

- Perhaps some unexpected results
- http://localhost:4220/name with 3 spaces.txt
  - This one should be ok: spaces replaced with %20 on Firefox and Chrome
- http://localhost:4220/exclamation!
  - First line of request: "GET /files/exclamation! HTTP/1.1”
  - What happened here? Neither Chrome nor Firefox replaced the ‘!’ character. Not necessarily expected.
- http://localhost:4220/C# or C++?
  - First line of request: "GET /files/C HTTP/1.1"
  - VERY unexpected results. Both Firefox and Chrome seem to be truncating the URI!

Percent Encoding

- Let’s just consider the first two and get part of the explanation
  - http://localhost:4220/name with 3 spaces.txt
  - http://localhost:4220/exclamation!
- Spaces were really the only characters that could mess up the first line of the request, since they’re the reserved delimiter according to the HTTP specification
- So in short, browsers only use percent encoding on spaces

Percent Encoding

- So in short, browsers only use percent encoding on spaces
  - Is this true? What about the ‘#’ character?
- As a test, navigate to `http://localhost:4220/!$&'()*+,/:;=?@[]` in browser
  - That’s a URL with all the percent-encoding reserved characters except for ‘#’
  - First line of request: "GET /!$&'()*+,/:;=?@[] HTTP/1.1”
  - Nothing percent encoded by Chrome or Firefox

Percent Encoding

- If we say “browser only percent encodes spaces in URL and # character is some sort of special case”, then we’ve got maybe 95% of the story. But let’s investigate the other 5% by examining the request data for:
  - http://localhost:4220/C# or C++?
- If we inspect the request data, what might we find?
- Where did the rest of the URL go? (discuss in class, then investigate in debugger)

Percent Encoding

- URL: http://localhost:4220/C# or C++?
- The browser expects the content after a '#' character in the URL to refer to a named anchor on the page
  - So the browser really *does* remove that part ("# or C++?") of the URL
  - It is not sent to the server
- A link on the page that refers to a resource on the server with the '#' character in the name, must encode it as %23
- Implies the files web service must also decode these
  - Many services will need to encode/decode, so consider static utility functions in WebServer class

Files Web Service Handler Logic

1. Percent-decode URI
2. If it refers to file somewhere in the shared folder then send file response
3. Else if it refers to a folder somewhere in the shared folder then send back an HTML listing for the folder
4. Else it's a bad URI, so respond appropriately
    - 400 if bad request
    - 404 if not found
    - (see HTTP status codes here)

Files Web Service Handler Logic

- From previous slide: “Send 400 if bad request”
- What is a bad request?
- Consider a malicious client request:
  - GET /../../sysFile.dat
- If your service isn’t smart, it may unintentionally send that exact string to a file API function which would go up 2 directories outside of your shared folder
  - Need to recognize and reject such requests

File System Abstraction

- Consider an object-oriented design that abstracts the file system
- "422" suffix to avoid naming conflicts
  - FileSys422 class
  - Dir422 class
  - File422 class

File System Abstraction

- FileSys422 class
  - Dir422 GetRoot()
    - Gets the root directory for the file system
    - Parent of this directory is expected to be null

File System Abstraction

- Dir422 class
  - string Name { get; }
  - List<Dir422> GetDirs()
  - List<File422> GetFiles()
  - Dir422 Parent
    - Null for root of file system
  - bool ContainsFile(string fileName, bool recursive)
  - bool ContainsDir(string dirName, bool recursive)
  - Dir422 GetDir(string dirName)
    - Non-recursive, gets directory with the specified name or null if no such directory exists within this one
    - Rejects (i.e. returns null) if dirName contains path separator characters
  - File422 GetFile(string fileName)
    - Analogous to GetDir

File System Abstraction

- File422 class
  - string Name { get; }
  - Stream OpenRead()
  - Stream OpenReadWrite()
    - Will probably not need until we implement uploading
  - Dir422 Parent
    - Expected to always be non-null for files

File System Abstraction

- Why do this?
- Answer specific to our context:
  - Files web service gets a FileSys422 object passed to constructor
  - Hosts content from this file system
  - File system object designed to restrict access outside the root folder

File System Abstraction

- Why do this?
- More generic answer:
  - Almost all modern apps need to read and write configuration/data files
  - Some want to deal with the app folder for their data, others maybe a remote database through SQL blobs, others some sort of cloud storage service, others to encrypted files, and others maybe all of the above
  - App doesn’t need to care about where/how file data is stored if dealing with abstract file system object

File System Abstraction

- Why do this?
  - Testing
- Easy to simulate
  - Have simulated file systems, perhaps entirely in memory
    - class MemFileSys : FileSys422
  - Scenarios where files load/save slowly
  - Scenarios where files don't load at all, despite existing on the disk (or whatever storage location)
    - Why/how can this happen? (discuss)
  - Scenarios where file/directory contents are changing rapidly
    - Files/directories getting deleted and/or new ones added

File System Abstraction

- Implement a set of classes to do interaction with the “normal” file system
  - class StandardFileSystem : FileSys422
- Constructor takes a path on the actual file system
  - /home/uername/Desktop/myShare
- Can be implemented to treat that path as the root and not allow access to things above it
  - Dir422 object returned by StandardFileSystem.GetRoot() will have a null parent

File System Abstraction

- Implement a set of classes to do interaction with the “normal” file system
- Implementation is relatively easy
  - System.IO.Directory
  - System.IO.File
  - and of course the FileStream class
- Read-only implementation for now
  - Will add writing if and when we add upload to our files web service

Files Web Service

- Has member variable of type FileSys422
  - Let's say it's: FileSys422 m_fs
- When a GET comes into the handler, map URI to a File422 object in the file system
  - "/" is m_fs.GetRoot()
  - All others require mapping to either a directory or file
  - How to implement?

```
void SomeFuncInOurService(string URI) {
    if URI maps to directory -> send HTML file listing
    else if URI maps to file -> send file contents
    else -> send 404
}
```

- Discuss implementation in class

Producing an HTML File Listing

- Since learning HTML isn’t really a goal in this class, here’s an idea of how to build a listing of files

```
string GetHTMLListing(Dir422 dir)
{
 StringBuilder sb = new StringBuilder(“<html>”);
 foreach (File422 file in dir.GetFiles()) {
  sb.AppendFormat(
   “<a href=‘{0}’>{1}</a><br>”,
   GetHREFFromFile422(file),
   file.Name);
 }
 // For you TODO: produce another listing of directories
 sb.Append(“</html>”);
 return sb.ToString();
}
```

Producing an HTML File Listing

- Discuss in class if time permits:

```
string GetHREFFromFile422(File422 file) {
  // ?
}
```

- If the file were named "hello.txt" and in the root (hosted folder), the HREF (hyperlink reference) would be "/hello.txt"
- If the file were named "movie.mp4" and in a folder called "Movies" that resides in the root (hosted folder) then the HREF would be "/Movies/movie.mp4"

Sending Back File Contents

- A request for file contents should send in the response
  - 200 OK line
  - Content-Length with appropriate value header
  - Content-Type header
  - All contents from file as body of the response
    - Stream it out through the response socket
- Add function to WebRequest class:
  - `public bool WriteGenericFileResponse(Stream fileData)`

Content-Type Header

- Can look at specification here for info on how these types are defined, or here for a list of most possibilities
- But in short, it's likely to just be one of these:
  - application/octet-stream
    - (Recall: octet means 8 bits, or a byte, so this is sort of for generic byte streams)
  - text/html
  - text/xml
  - image/png
  - image/gif
  - image/jp2
  - video/mp4

FilesWebService Conclusion

- Share a directory of files through your web server
- First service that actually does some useful work
- Requires small extensions in server core to assist with decoding URIs and sending responses, but most of the work is in the FilesWebService class
- Testing
  - Create FileSys422 object, populate some files, send it as the share to the server
  - Request files from server and verify against content from FileSys422 object