What Every Developer Should Know About URLs

03/05/2010 · 2757 words · 13 min read

I have recently written about the value of fundamentals in software development. I am still firmly of the opinion that you need to have your fundamentals down solid, if you want to be a decent developer. However, several people made a valid point in response to that post, in that it is often difficult to know what the fundamentals actually are (*be they macro or micro level*). So, I thought it would be a good idea to do an ongoing series of posts on some of the things that I consider to be fundamental – this post is the first instalment.

Being a developer this day and age, it would be almost impossible for you to avoid doing some kind of web-related work at some point in your career. That means you will inevitably have to deal with URLs at one time or another. We all know what URLs are about, but there is a difference between knowing URLs like a user and knowing them like a developer should know them.

As a web developer you really have no excuse for not knowing everything there is to know about URLs, there is just not that much to them. But, I have found that even **experienced developers often have some glaring holes in their knowledge of URLs**. So, I thought I would do a quick tour of everything that every developer should know about URLs. Strap yourself in – this won't take long :).

The Structure Of A URL

Structure

This is easy, starts with HTTP and ends with .com right :)? Most URLs have the same general syntax, made up of the following nine parts:

```
<scheme>://<username>:<password>@<host>:<port>/<path>;<parameters>?<query>
```

Most URLs won't contain all of the parts. The most common components, as you undoubtedly know, are the scheme, host and path. Let's have a look at each of these in turn:

- **scheme** – this basically specifies the protocol to use to access the resource addressed by the URL (*e.g. http, ftp*). There are a multitude of different schemes. A

scheme is official if it has been registered with the IANA (*like http and ftp*), but there are many unofficial (*not registered*) schemes which are also in common use (*such as sftp, or svn*). The scheme must start with a letter and is separated from the rest of the URL by the first : (*colon*) character. That's right, the ***//*** is not part of the separator but is infact the beginning of the next part of the URL.

- **username** – this along with the password, the host and the port form what's known as the authority part of the URL. Some schemes require authentication information to access a resource this is the username part of that authentication information. The username and password are very common in ftp URLs, they are less common in http URLs, but you do come across them fairly regularly.

- **password** – the other part of the authentication information for a URL, it is separated from the username by another : (*colon*) character. The username and password will be separated from the host by an @ (*at*) character. You may supply just the username or both the username and password e.g.:

  ```
  ftp://some_user@blah.com/
  ftp://some_user:some_path@blah.com/
  ```

  If you don't supply the username and password and the URL you're trying to access requires one, the application you're using (*e.g. browser*) will supply some defaults.

- **host** – as I mentioned, it is one of the components that makes up the authority part of the URL. The host can be either a domain name or an IP address, as we all should know the domain name will resolve to an IP address (*via a DNS lookup*) to identify the machine we're trying to access.

- **port** – the last part of the authority. It basically tells us what network port a particular application on the machine we're connecting to is listening on. As we all know, for HTTP the default port is 80, if the port is omitted from an http URL, this is assumed.

- **path** – is separated from the URL components preceding it by a / (*slash*) character. A path is a sequence of segments separated by / characters. The path basically tells us where on the server machine a resource lives. Each of the path segments can contain parameters which are separated from the segment by a ; (*semi-colon*) character e.g.:

  ```
  http://www.blah.com/some;param1=foo/crazy;param2=bar/path.html
  ```

  The URL above is perfectly valid, although this ability of path segments to hold parameters is almost never used (*I've never seen it personally*).

- **parameters** – talking about parameters, these can also appear after the path but before the query string, also separated from the rest of the URL and from each other by ; characters e.g.:

  ```
  http://www.blah.com/some/crazy/path.html;param1=foo;param2=bar
  ```

  As I said, they are not very common

  - **query** – these on the other hand are very common as every web developer would know. This is the preferred way to send some parameters to a resource on the server. These are ***key=value*** pairs and are separated from the rest of the URL by a ? (*question mark*) character and are normally separated from each

other by & (*ampersand*) characters. What you may not know is the fact that it is legal to separate them from each other by the ; (*semi-colon*) character as well. The following URLs are equivalent:

```
http://www.blah.com/some/crazy/path.html?param1=foo&param2=bar
http://www.blah.com/some/crazy/path.html?param1=foo;param2=bar
```

- **fragment** – this is an optional part of the URL and is used to address a particular part of a resource. We usually see these used to link to a particular section of an html document. A fragment is separated from the rest of the URL with a # (*hash*) character. When requesting a resource addressed by a URL from a server, the client (*i.e. browser*) will usually not send the fragment to the server (*at least not where HTTP is concerned*). Once the client has fetched the resource, it will then use the fragment to address the relevant part.

That's it, all you need to know about the structure of a URL. From now on you no longer have any excuse for calling the fragment – "***that hash link thingy to go to a particular part of the html file***".

Special Characters In URLs

 Special Character

There is a lot of confusion regarding which characters are safe to use in a URL and which are not, as well as how a URL should be properly encoded. Developers often try to infer this stuff from general knowledge *(i.e. the / and : characters should obviously be encoded since they have special meaning in a URL*). This is not necessary, you should know this stuff solid – it's simple. Here is the low down.

There are several sets of characters you need to be aware of when it comes to URLs. Firstly, the characters that have special meaning within a URL are known as ***reserved characters***, these are:

```
";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" | "$" | ","
```

What this means is that these characters are normally used in a URL as-is and are meaningful within a URL context (*i.e. separate components from each other etc*.). If a part of a URL (*such as a query parameter*), is likely to contain one of these characters, it should be escaped before being included in the URL. I have spoken about URL encoding before, check it out, we will revisit it shortly.

The second set of characters to be aware of is the ***unreserved set***. It is made up of the following characters

```
"-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"
```

The characters can be included as-is in any part of the URL (*note that they may not be allowed as part of a particular component of a URL*). This basically means you don't need to encode/escape these characters when including them as part of a URL. **You CAN escape them without changing the semantics of a URL, but it is not recommended**.

The third set to be aware of is the_ **'unwise' set**_, i.e. it is unwise to use these characters as part of a URL. It is made up of the following characters

```
"{" | "}" | "|" | "\" | "^" | "[" | "]" | "`"
```

These characters are considered unwise to use in a URL because gateways are known to sometimes modify such characters, or they are used as delimiters. That doesn't mean that these characters will always be modified by a gateway, but it can happen. So, if you include these as part of a URL without escaping them, you do this at your own risk. What it really means is **you should always escape these characters if a part of your URL (*i.e. like a query param*) is likely to contain them**.

The last set of characters is the excluded set. It is made up of all ASCII control characters, the space character as well the following characters (*known as delimiters*)

```
"<" | ">" | "#" | "%" | '"'
```

The control characters are non-printable US-ASCII characters (*i.e. hexadecimal 00-1F as well as 7F*). These characters must always be escaped if they are included in a component of a URL. Some, such as # (*hash*) and % (*percent*) have special meaning within the context of a URL (*they can really be considered equivalent to the reserved characters*). Other characters in this set have no printable representation and therefore escaping them is the only way to represent them. **The <, > and " characters should be escaped since these characters are often used to delimit URLs in text**.

To URL encode/escape a character we simply append its 2 character ASCII hexadecimal value to the % character. So, *the URL encoding of a space character is %20* – we have all seen that one. The % character itself is encoded as %25.

That's all you need to know about various special characters in URLs. Of course aside from those characters, alpha-numerics are allowed and don't need to be encoded :).

A few things you have to remember. A URL should always be in its encoded form. The only time you should decode parts of the URL is when you're pulling the URL apart (*for whatever reason*). Each part of the URL must be encoded separately, this should be pretty obvious, you don't want to try encoding an already constructed URL, since there is no way to distinguish when reserved characters are used for their reserved purpose (*they shouldn't be encoded*) and when they are part of a URL component (*which means they should be encoded*). Lastly you should never try to double encode/decode a URL. Consider that if you encode a URL once but try to decode it twice and one of the URL components contains the % character you can destroy your URL e.g.:

```
http://blah.com/yadda.html?param1=abc%613
```

When encoded it will look like this:

```
http://blah.com/yadda.html?param1=abc%25613
```

If you try to decode it twice you will get:

1. `http://blah.com/yadda.html?param1=abc%613`

   Correct

   - `http://blah.com/yadda.html?param1=abca3`

     Stuffed

By the way I am not just pulling this stuff out of thin air. It is all defined in RFC 2396, you can go and check it out if you like, although it is by no means the most entertaining thing you can read, I'd like to hope my post is somewhat less dry :).

Absolute vs Relative URLs


Absolut

The last thing that every developer should know is the difference between an absolute and relative URL as well as how to turn a relative URL into its absolute form.

The first part of that is pretty easy, if a URL contains a scheme (*such as http*), then it can be considered an absolute URL. Relative URLs are a little bit more complicated.

A relative URL is always interpreted relative to another URL (*hence the name :)*), **this other URL is known as the base URL**. To convert a relative URL into its absolute form we firstly need to figure out the base URL, and then, depending on the syntax of our relative URL we combine it with the base to form its absolute form.

We normally see a relative URL inside an html document. In this case there are two ways to find out what the base is.

1. The base URL may have been explicitly specified in the document using the HTML tag.
2. If no base tag is specified, then the URL of the html document in which the relative URL is found should be treated as the base.

Once we have a base URL, we can try and turn our relative URL into an absolute one. First, we need to try and break our relative URL into components (*i.e. scheme, authority (host, port), path, query string, fragment*). Once this is done, there are several special cases to be aware of, all of which mean that our relative URL wasn't really relative.

- if there is no scheme, authority or path, then the relative URL is a reference to the base URL

- if there is a scheme then the relative URL is actually an absolute URL and should be treated as such
- if there is no scheme, but there is an authority (*host, port*), then our relative URL is likely a network path, we take the scheme from our base URL and append our "*relative*" URL to it separating the two by ://

If none of those special cases occurred then we have a real relative URL on our hands. Now we need to proceed as follows.

- we inherit the scheme, and authority (host, port) from the base URL
- if our relative URL begins with /, then it is an absolute path, we append it to the scheme and authority we inherited from the base using appropriate separators to get our absolute URL
- if relative URL does not begin with / then we take the path of from base URL, discarding everything after the last / character
- we then take our relative URL and append it to the resulting path, we now need to do a little further processing which depends on the first several characters of our relative URL
- if there is a ./ (*dot slash*) anywhere in a resulting path we remove it (*this means our relative URL started with ./ i.e. ./blah.html*)
- if there is a ../ (*dot dot slash*) anywhere in the path then we remove it as well as the preceding segment of the path i.e. all occurrences of **"/../"** are removed, keep doing this step until no more ../ can be found anywhere in the path (*this means our relative path started with one or more ../ i.e. ../blah.html or ../../blah.html etc.*)
- if the path ends with .. then we remove it and the preceding segment of the path, i.e. **"/.."** is removed (*this means our relative path was .. (dot dot)*)
- if the path ends with a . (dot) then we remove it (*this most likely means our relative path was . (dot)*)

At this point we simply append any query string or fragment that our relative URL may have contained to our URL using appropriate separators and we have finished turning our relative URL into an absolute one.

Here are some examples of applying the above algorithm:

```
1)
base: http://www.blah.com/yadda1/yadda2/yadda3?param1=foo#bar
relative: rel1




final absolute: http://www.blah.com/yadda1/yadda2/rel1



2)
base: http://www.blah.com/yadda1/yadda2/yadda3?param1=foo#bar
relative: /rel1
```

```
final absolute: http://www.blah.com/rel1
```

```
3)
base: http://www.blah.com/yadda1/yadda2/yadda3?param1=foo#bar
relative: ../rel1
```

```
final absolute: http://www.blah.com/yadda1/rel1
```

```
4)
base: http://www.blah.com/yadda1/yadda2/yadda3?param1=foo#bar
relative: ./rel1?param2=baz#bar2
```

```
final absolute: http://www.blah.com/yadda1/yadda2/rel1?param2=baz#bar2
```

```
5)
base: http://www.blah.com/yadda1/yadda2/yadda3?param1=foo#bar
relative: ..
```

```
final absolute: http://www.blah.com/yadda1/
```

```
1         Now you should be able to confidently turn any relative URL int
2
3         There you go that's really all there is to know about URLs, it'
4
5         <span style="font-size: 10px; font-family: trebuchet ms;">Image
```

#coding #fundamentals #urls

‹ 8 Types Of Software Consulting Firms – Which One Do You Work For?

Executing Multiple Commands – A Bash Productivity Tip ›