What is URL Design?

Johan Sölve

A URL is not a Filename

A URL1 is usually tied to a specific filename on the server, but this is not how things are intended. Usually it's the server technology that leads us into thinking that a URL maps to a specific filename, but URLs (or URIs in general) have conceptually nothing to do with a file system.

http://www.w3.org/TR/chips/

By disconnecting the URL from the server's file system, we get much more freedom when choosing URLs. Each URL doesn't have to correspond to a physical directory or file, and the URLs we use don't even need to have a file extension.

Choosing URLs

So what's a good URL? There is a lot of reading to do on this topic around the web. I've collected a few recommendations from different sources.

http://www.w3.org/Provider/Style/URI

http://www.useit.com/alertbox/990321.html

http://www.w3.org/QA/Tips/uri-choose

A URL should be:

- Short

  Keep URLs short to make them easier to communicate or share.
- Memorable

  Make it easier for users to return or remember a URL they've read somewhere.
- Bookmark-able

  Avoid session specific info in the host or path part of a URL.
- An aid in site navigation

  Let the URL reflect the site structure to help the visitor visualize where he is on a site.
- Guess-able

  A URL that can be guessed is a great help for visitors.
- Hackable

  Make it possible and easy for users to "hack" the URL for example to move upwards in the site hierarchy.
- Persistent ≠ Make it easy for people to link to your site

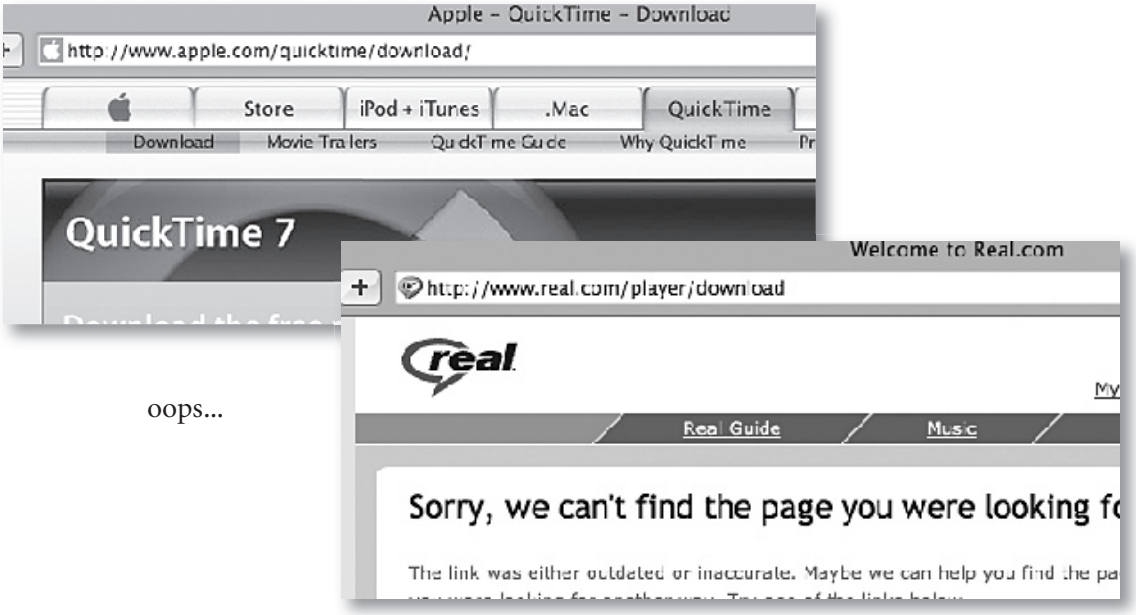
  URLs don't change: people change them.

Let URLs remain forever, do not move pages around.

Avoid causing link-rot.

“Persistent URLs Attract Links, Link-rot equals lost business” (Jacob Nielsen)

If you have to change a URL, either point to the new URL or make it very clear that the page is gone



oops...

- Technology neutral

  Avoid file extensions that is dependent of the server architecture.

  Security - avoids revealing information that is useful for a hacker,

  future proofing - allows changing server architecture without changing any URLs.
- The acid test — can you read a URL to someone over the phone?

Ways to do URL Mapping With Lasso

Apache mod_rewrite

mod_rewrite is a robust and powerful way to parse and manipulate URLs. However it is implemented completely outside of Lasso and is dependent of the web server used. It can also be a bit challenging to master the mod_rewrite rules.

Here isa configuration example that will pass any request for nonexistent files or directories to index.lasso, passing the requested URL as parameter.

```
RewriteEngine On
#RewriteBase /
# Check to see if the request is a real file or directory
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !.*/Security
# Everything else gets sent to index.lasso
RewriteRule ^(.*)$ /index.lasso?response_filepath=/$1 [QSA,L,NS]
```

IISRewrite and ISAPI_Rewrite for IIS, WebSTAR Rewrite

Similar to mod_rewrite.

Error.lasso

This method uses a custom error.lasso file to parse the requested URL when requesting a non-existent file. The idea is to have Lasso process requests for non-existent files. However Apache won't pass the requested file path to error.lasso, instead the actual path to error.lasso is returned which makes it impossible to parse the URL that was originally requested. In addition, the request must end with .lasso for error.lasso to kick in.

Lasso Built-In using [Define_AtBegin]

Lasso 8 introduced the tag [Define_AtBegin] which lets us hook into the processing of any Lasso page processed by the server. This lets us use Lasso to parse the requested URL before running the actual page.

How to use Define_Atbegin as Pre-Processor for URL Mapping

Introducing [Define_AtBegin]

[Define_AtBegin] is a tag that defines pre-processing code that will be executed before each page on a site is executed or even loaded - for EVERY request to Lasso. The atbegin code is defined globally for each Lasso Site by calling the tag from a lasso file in LassoStartup.

This pre-processing ability gives us a way to analyze the requested URL and map it to a desired action before loading any page for it.

Since [Define_AtBegin] defines pre-processing code that will be executed for every page on a site, it is important to debug the code carefully before using it on a live server. Otherwise every page requested on the site might fail.

Advantages Over Other Methods

Using a Lasso pre-processor to perform URL mapping makes the process mostly self-contained within Lasso which reduces the complexity of the server setup. Less things to setup, less things that can fail. The URL parsing logic, decisions about not to get involved with the request, all of it can be done within Lasso. This also makes it easier to integrate the URL mapping more tightly with the rest of the site's logic, or even databases.

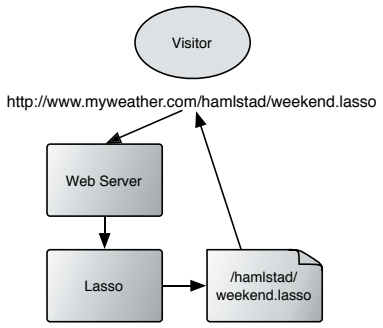
Only a minor configuration needs to be done in the web server. With Apache it can even be configured in a .htaccess file (if allowed on the server) to provide a directory-specific activation.

The pre-processor can also be used for many other useful things.

How does it work?

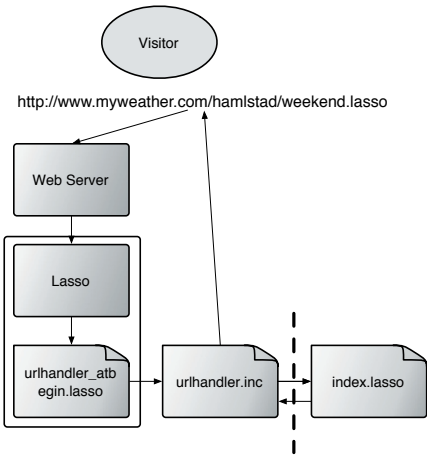
We intend to use a pre-processor to parse the requested URL before executing the page. The pre-processor will be defined by a [Define_AtBegin] call. How will the pre-processor interact with the flow of execution?

Let's begin with looking at what a normal Lasso page request looks like.



- The web server is configured to send URLs with .lasso extension to Lasso.
- The requested URL maps directly to a filename on the web server.
- Lasso executes the requested page.

This is what a page request can look like when we use a pre-processor to do URL mapping.



- The web server is configured to send the request to Lasso even if the requested URL doesn't have a .lasso extension.
- The requested URL has nothing to do with the filesystem on the web server.
- Lasso has a pre-processor defined at startup (urlhandler_atbegin.lasso) which will call a URL handler (urlhandler.inc) to have the requested URL parsed.
- After parsing the URL, urlhandler.inc will execute the site's default page or hub file, passing the parsed URL as argument.

Web Server Configuration

The web server should be configured to have Lasso process all requests for URLs that we want to parse, for example any URL without file extension. We would normally want to exclude files that have file extension to avoid having Lasso process image files, media files, CSS files, javascript files and so on. We could also limit processing of requests to specific directories.

Configuration example for Apache, to put in a virtual host configuration or in a .htaccess file:

```
<LocationMatch "^[^\.]+$">
  # anywhere without file extensions
  SetHandler lasso8-handler
</LocationMatch>
```

See "Additional Apache configuration examples" below for other setups.

For IIS 6.0 on Windows 2003 Server, you can set up a wildcard mapping for which will direct any request to Lasso on a folder by folder basis.

See " Installing Wildcard Application Mappings (IIS 6.0)"

http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/5c5ae5e0-f4f9-44b0-a743-f4c3a5ff68ec.mspx

Also see "Setting Application Mappings in IIS 6.0 (IIS 6.0)"

http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/4c840252-fab7-427e-a197-7facb6649106.mspx

Lasso Configuration

We begin with putting a Define_AtBegin script in a .lasso file in the LassoStartup folder for the current Lasso Site and then restart LassoService for the site.

We want to make the atbegin handler very generic and have as little logic in the actual atbegin-handler as possible to avoid restarting Lasso when changing the logic for URL mapping. We also want it to be completely transparent to not interfere with normal operation for virtual hosts on the same Lasso Site that don't use URL mapping.

urlhandler_atbegin.lasso

```
<?LassoScript
define_atbegin: {
  if: file_exists: '/urlhandler.inc';
    include: '/urlhandler.inc';
  /if;
};
```

This atbegin handler looks for the file urlhandler.inc in the virtual host's root and includes this file to do the actual processing, to either map the URL to an action, pass it on as a normal request or return an error. By keeping the atbegin handler simple as this, it becomes transparent so that virtual hosts that do not have urlhandler.inc in the web root are not affected at all by the atbegin handler.

[Define_AtBegin] use a compound expression to define the pre-processing code. A compound expression is a LassoScript that is placed between { }. It's a way to specify a set of Lasso tags that can be stored and executed later. When [Define_AtBegin] is called at startup, the compound expression is stored internally by Lasso, and will be retrieved and executed before every page request.

Handling the URL

Now that we have all Lasso requests for a host passing through one spot, we get the opportunity to parse the requested URL by looking at [response_filepath]. Now we can do whatever we want to map the URL to a desired action on the web site.

Here is an example URL handler that will parse URLs for the news and products sections of a web site:

urlhandler.inc

```
<?LassoScript
  // this file is called by the atbegin handler, so it is executed
  // before any page is being processed.
  if:response_filepath->(endswith:'.lasso') ||
      response_filepath->(endswith:'.lassoapp');
    // don't do anything for normal .lasso and .lassoapp requests
  else;
    if:response_filepath->(beginswith:'/news/') ||
        response_filepath->(beginswith:'/products/');
      var:'url_path'=response_filepath, 'section'='';
      $url_path->(removeleading:'/');
      if:!$url_path->(endswith:'/');
        $url_path += '/';
      /if;
      if:$url_path->(beginswith:'news/');
```

```
        // check for pattern /news/2004/12/31/keyword/
        //using regular expression
        var:'pathcheck'=(string_findregexp:$url_path, -find=
          '^news/(20\\d{2})/(0[1-9]|1[0-2])/' + '([^/]+)/([^/]*)');
        if:$pathcheck->size >= 5;
          $section = 'news';
          var:'newsdate'=$pathcheck->(get:2) + '/' +
                         $pathcheck->(get:3) + '/' +
                         $pathcheck->(get:4);
          $newsdate = (date:$newsdate, -format='%Y/%m/%d');
          var:'newskeyword'=$pathcheck->(get:5);
          var:'newsextra'='';

          if:$pathcheck->size >= 6;
            $newsextra = $pathcheck->(get:6);
          /if;
        /if;
      else:$url_path->(beginswith:'products/');
        $section = 'products';
        // split up the path in components
        $url_path = $url_path->(split:'/');
      /if;
      // run site
      // use absolute path!
      $__HTML_REPLY__ = include:'/index.lasso';
      abort;
    /if;
  /if;
?>
```

Let's walk through this code bit by bit:

urlhandler.inc

```
<?LassoScript
  // this file is called by the atbegin handler, so it is executed
  // before any page is being processed.
  if:response_filepath->(endswith:'.lasso') ||
      response_filepath->(endswith:'.lassoapp');
    // don't do anything for normal .lasso and .lassoapp requests
```

Since .lasso and .lassoapp URLs are always set to be processed by Lasso, those requests will also get into the pre-processor. But we don't want to interfere with those request so we will just let them pass through. We will only deal with extension-less URLs.

```
else;
    if:response_filepath->(beginswith:'/news/') ||
        response_filepath->(beginswith:'/products/');
```

The URL path begins with either /news/ or /products/, so we know we should look closer at the path

```
var: 'url_path'=response_filepath,
'section'='';
```

We use the variable "section" to keep track of what main section of the site we are visiting. This is used later on.

```
$url_path -> (removeleading: '/');
if: !($url_path -> endswith: '/');
$url_path += '/';
/if;
```

Trim the leading / from the path, and add a trailing / if there is none. We don't deal with paths that have file extensions here, so we don't expect to end up with a path like news/current.lasso/.

```
if: $url_path -> (beginswith: 'news/');
// check for pattern /news/2004/12/31/keyword/
//using regular expression
var: 'pathcheck'=(string_findregexp: $url_path,
      -find='^news/(20\\d{2})/(0[1-9]|1[0-2])/'
            + '([0-2][1-9]|3[0-1])/' + '([^/]+)/([^/]*)');
```

We use a fairly strictly defined pattern for the URL path for news. This breaks the recommendation for "hackable URLs" since the pattern check doesn't allow the visitor to hack off the day or month from the URL, for example if he wants to get to a news archive for a year or month. We should also add rules to allow a URL like `/news/current' to always return the most current news article.

[string_findregexp] returns a nicely split array if the path matches our defined pattern.

```
if: $pathcheck -> size >= 5;
  $section = 'news';
  var: 'newsdate'=($pathcheck -> (get: 2))
      + '/' + ($pathcheck -> (get: 3))
      + '/' + ($pathcheck -> (get: 4));
  $newsdate = (date: $newsdate, -format='%Y/%m/%d');
  var: 'newskeyword'=($pathcheck -> (get: 5)),
  var: 'newsextra'='';
  if: $pathcheck -> size >= 6;
    $newsextra=($pathcheck -> (get: 6));
  /if;
/if;
```

The result of the parsing for a news URL is a few variables:

$section tells the site what main section we are in.

$newsdate contains the publication date that we will use when looking up the news article in our database.

$newskeyword contains an identification string, usually the news headline in lowercase and with spaces removed.

$newsextra contains an optional extra item, for example `comments' that will lead to a page related to the news article.

```
else: $url_path -> beginswith: 'products/';
$section = 'products';
// split up the path in components
$url_path = $url_path -> (split: '/');
```

For the `products' section we have much more liberal rules. We will simply pass on the $url_path variable as array to use later on when looking up the current product category in our product database.

In a real world case we would continue with more rules for different site sections.

```
/if;
// run site
// use absolute path!
$__HTML_REPLY__ = include:  '/index.lasso';
abort;
```

This is where all the action happens. Index.lasso is the central hub page for the actual site, and the variables we have set above will be handled in the site code to show the desired page.

Remember, this file is run by the atbegin handler, and the code run by define_atbegin will never return any output to the web page since Lasso will clear the page buffer once the real page begins processing. Therefore we must explicitly put the page result into the page buffer, which is stored in the variable $__html_reply__.

Once we have stored the page output into the page buffer we must also stop any further processing of the request using the abort tag, otherwise Lasso would try to run the file that was actually requested, which is probably nonexistent since we use virtual paths in the URL.

```
    /if;
  /if;
?>
```

Error Handling

The entire mechanism for URL mapping relies on the fact that a URL is not a filename. That means that we can technically never end up in a "File not found" situation, so there's no natural mechanism to provide file not found errors.

Nevertheless it's quite possible to get a request for a URL that is invalid and that doesn't match anything relevant in our site. For these situations we need to provide a proper error page and most importantly a proper HTTP status code, so search engine crawlers, bookmark checkers and other automated visitors know about the error.

For URLs that are invalid we should serve a "404 Not Found" status code.

For URLs that have been valid but are no longer available we should serve a "410 Gone" status code.

http://www.w3.org/Protocols/HTTP/1.1/spec.html#Status-Codes

Here is a custom tag to easily set the HTTP status code without affecting the rest of the HTTP header:

```
define_tag: 'setHTTPstatus', -required='statuscode';
// replace status code but keep leading HTTP with version
  $__http_header__ =
    (string_replaceregexp: $__http_header__,
          -find='(^HTTP\\S+)\\s+.*?\r\n',
          -replace='\\1 ' + #statuscode + '\r\n');
/define_tag;
```

Debugging AtBegin Scripts

One common method to debug tricky code problems is to insert the [abort] tag on different places and look at what has been output on the page before getting to the [abort] tag.

One thing to note about this is that when executing a page by calling it from an AtBegin script as we do in `urlhandler.inc`, we can't use the [abort] for debugging any more. If we put [abort] anywhere in the Lasso code that is executed in index.lasso, it will result in a completely blank page. This is because it will stop any further processing

whatsoever which means that the code in urlhandlerinc that assigns `$_ _html_reply_ _` to the output of index.lasso will not be executed either. The end result is that putting an [abort] tag anywhere in index.lasso will result in a completely blank page.

Lasso SiteAdmin settings

No settings in Lasso SiteAdmin should need to be changed, but apparently it seems to be needed to add wildcard (.*) as allowed extension in the File Tags Settings under File Extensions. It is not entirely clear if and why this appears to be needed sometimes.

Additional Apache configuration examples

This configuration will have Lasso process any non-file request (no periods in the URL) only in a specific folder.

```
<Location ~ "^/site/[^\.]+$">
  # files without file extensions in specific folder
  SetHandler lasso8-handler
</Location>
```

This configuration will have Lasso process ANY request in a specific folder. This can be useful to offer protected file downloads, using the [file_stream] tag.

```
<Location ~ "^/downloads/">
  # any file in specific folder
  SetHandler lasso8-handler
</Location>
```

Other uses for AtBegin

Methods That Can Be Implemented in Urlhandler.inc

- Protection of .inc files

Protecting .inc files from being accessed directly is important to not reveal unprocessed source code, or run include files out of context. This protection requires that .inc files are set to be processed by Lasso.

urlhandler.inc

```
<?LassoScript
  if: response_filepath -> (endswith: '.inc') ;
    // .inc files are not allowed to be called directly
    $__HTML_REPLY__ = '<h1>Not authorized</h1>';
    abort;
  /if;
?>
```

- Locking down siteadmin.lassoapp

  This code makes it impossible to access siteadmin.lassoapp except from the local computer.

urlhandler.inc

```
<?LassoScript
  // make sure we're ok to use siteadmin
  if: response_filepath -> split: '/' -> last -> (beginswith: 'siteadmin.')
   && response_filepath -> (endswith: '.lassoapp')
   && client_ip != '127.0.0.1';
      $__HTML_REPLY__ = '<h1>Not authorized</h1>';
      abort;
  /if;
?>
```

Methods That Are Implemented as Separate atBegin Handlers

- Logging and timing (when used in pair with [Define_AtEnd])

```
<?lassoscript
  define_atbegin: {
    var: '_pagetimerstart'=_date_msec;
    define_atend: {
      log_detail: 'http://' + server_name + response_filepath + ' '
        + (_date_msec - $_pagetimerstart) + ' ms';
    };
  };
?>
```

- Debugging troublesome pages

  [Bil Corry] If it were me, I'd create a define_atbegin script that logs every page request to a database. Then have a define_atend that erases that db entry. The only entries left will be those that never finished executing.

- Branding

  Use an AtBegin handler install an AtEnd-handler on every page that adds an ISP logo, adds site information, copyright info or disclaimer as html comment, adds banner ads or other things by manipulating the `$ _ _ html _ reply _ _` variable before it is served to the visitor.

- Optimizing HTML code and collapsing whitespace

  Use an AtBegin handler install an AtEnd-handler on every page that removes excess whitespace and line breaks from the html source.

- Gzip compression of served HTML pages

  Tip of the week for September 23, 2005 includes a Gzip compression module for Lasso 8.1. The module allows Lasso to automatically compress all pages using Gzip for compatible browsers. The full source code for the modules is included and is a good example of how to create at-begin and at-end processes.

http://www.omnipilot.com/Tip%20of%20the%20Week.1768.8959.lasso

- Protected downloads with real filenames

  Serve a file for download using [file_stream] by reading the original file from outside of the web root, after checking that the user is authorized to download the file. By passing requests for ANY file to Lasso, the user can request the file by it's real name.

- Dynamically generated media files with real filenames

  Generate pdf or image files on the fly and serve them directly, even if the file is requested by its real name.

---

1 URL or URI? In this document we will refer to web addresses with the term "URL", but some of the referred links use the more general term "URI". In short, a URL is one kind of URI. Here we talk about web addresses, so let's stick to URL for now.