Server-Side Perl

CS174

Chris Pollett

Oct 30, 2006.

Outline

- Pattern Matching
- File I/O
- Common Gateway Interface
- Example server side script
- Query String Format
- Cookies

Pattern Matching

- Recall we discussed regular expressions and pattern matching when we talked about Javascript.
- Javascript's version of these concepts were actually based on the notions from Perl.
- Patterns in Perl a delimited by slashes.
- Nonspecial characters match themselves:

  /snow/ # matches the string "snow"
- A period matches any character other than a newline. So /snow./ matches snowy and snows
- [] are used to indicate the OR of its content [abc] matches either a or b or c.
- ^ - matches the start of a line
- $ - matches the end of a line

More Pattern Matching

- \s -- matches any whitespace character
- \d matches any digit
- {} use to indicate number of occurrences of pattern. /xy{4}z/ matches xyyyyz. An SSN could be matched as /\d{3}-\d{2}-\d{4}/
- *, +, ? - are the 0 or more, 1 or more, and 0 or 1 repetition operators as in Javascript.
- Modifiers can be attached to patterns as in Javascript. For instance, x, i.
- The string against which a pattern is matched is by default $_ . For example,

  ```
  if(/rabbit/) {print "there is a rabbit in $_ \n";}
  ```
- To match against some other string one can use the binding operator =~. For example, if($str =~ /rabbit/){…}
- Split can also use patterns: @word = split(/[.,]\s*/, $str);

Remembering Matches and Substitutions

- The part of a string that matched part of a pattern can be saved in implicit variables for future use. For example,

```
"4 July 1776" =~ /(\d+) (\w+) (\d+)/;
print "$2 $1, $3\n"; # prints July 4, 1776
```

- Sometimes is it convenient to be able to reference the parts of the string that preceded the match, the part that matched, or the part that followed the match.
- These are available in the variables: $\`, $&, and $' respectively.
- As in Javascript s is used as the substitution operator and g is used to indicate one wants to do a global substitution:

```
$_ = "Say it ain't so";
s/ain't/ is not/g;
```

File I/O

- As we said a couple of classes ago, files are referenced through program variables called *filehandles*.
- Filehandle names do not begin with special characters (like $’s) and are typically written in upper-case.
- The open function is used to associate an OS file with a particular file handle.
- In opening a file, we also have to say how we are going to access it:
  - < -- means the filehandle should be for input
  - > -- means the filehandle should be for output (creates the file if does not exist. Starts writing at the beginning of the file)
  - >> -- means one should append to the file
  - >+ -- means input from and output to the same file.
- As an example, we could create a file and write to it using:
  - open(OUT, “>myfile”) or die “Error opening the file for writing $!”;
  - print OUT “some stuff”; close OUT;

More File I/O

- Similarly, we can open a file for reading using:

  open(IN, “<myfile”);

  $first_line = <IN>;

  close IN;

- There are also command

  read*(filehandle, buffer, length [, offset]);*

  and

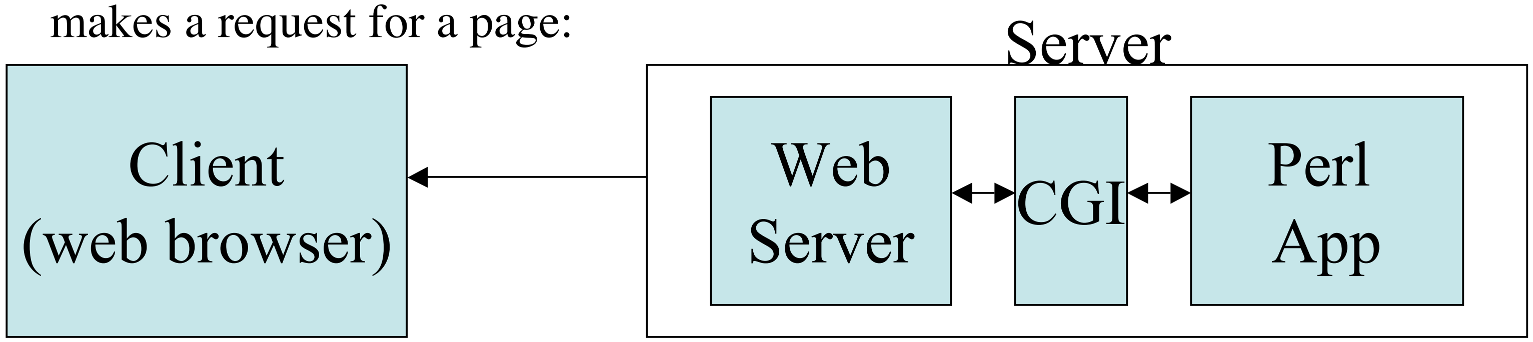
  seek(*filehandle, offset, base*)

  #base can be one of 0 (start of file), 1 (current pos), 2 (end of file)

  #offset can be positive or negative and indicates bytes.

Common Gateway Interface

- We are now going to talk about Perl in the context of web programming.
- We would like our Perl programs to run on the server when a client browser makes a request for a page:



- To accomplish this the web server is typically configured to recognize certain file extensions as being for scripts (Ex: .cgi).
- The server might also expect files from a certain directory to be scripts (Ex: cgi-bin).
- When the server receives a request for such a file, in the traditional approach, it would fork a process and set up the environment variables for this process according to the Common Gateway Interface.
- It would then run the Perl application and echo the results back to the Client.
- As forking can be slow, modern approaches based on mod_perl do a similar idea but within a thread of the web-server.

Example Server Side Script

```
#!/usr/bin/perl
#The line above says what app to use to run
#This script
print " Content-type: text/html  \n\n";
# although we don't send the status line,
# general and response headers, it does need
# to give the entity headers
print <<HTML;
<html><head><title>First CGI</title></head>
<body><h1>My first CGI program!</h1>
<p> The query string was $ENV{'QUERY_STRING'}</p>
</body></html>
HTML
```

Query String Format

- Form data is often sent in the QUERY_STRING environment variable indicated by the last example.
- Therefore, it is useful to be able to parse this variable to get out the name value pairs sent in the form.
- The general format of the query string looks like:

  $\text{name}_1\text{=value}_1\text{\&name}_2\text{=value}_2$ …
- Special characters are replaced with % followed by their ASCII code. Ex %20 is a space, %21 is !, etc.
- Sometimes spaces are replaced with +. This often is done with search engines.

Getting the Data sent from a Form

```
$request_method = $ENV{‘REQUEST_METHOD’};
if($request_method eq “GET”)
{
    $query_string = $ENV{‘QUERY_STRING’};
}
elsif($request_method eq “POST”)
{
    read(STDIN, $query_string,$ENV{‘CONTENT_LENGTH’});
}
else {exit(1);}
```

Cookies

- Sometimes it is useful to remember a client when it comes back.
- To do this one can use the HTTP-Cookie protocol.
- The Server can send as one of its response headers:

Set-Cookie: name=value; expires=some date; path= some path; domain= some_domain;

- When the Client comes back, it will send the cookie as part of its request header as:

Cookie: name=value