The

language

Uses of the World Wide Web:

- static documents
  (supported by HTML);
- dynamic documents
  (supported by CGI, ASP, various HTML extensions, ...); and
- interactive services
  (supported by `<bigwig>` and MAWL).

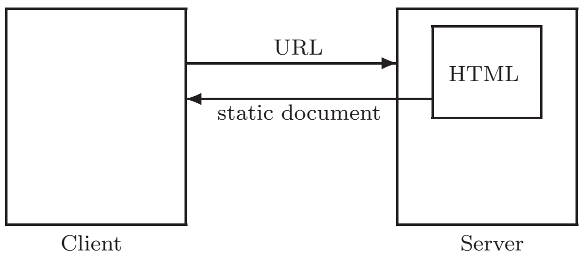
Static documents:

- there are too many documents;
- the documents are rarely updated; and
- the documents are not customized.

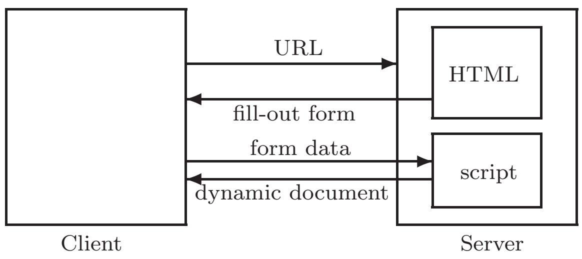
Dynamic documents:

- there are fewer documents;
- the documents are always updated;
- the documents are customized.

Standard interaction:



Common Gateway Interface:



Fill-out forms are an extension of HTML.

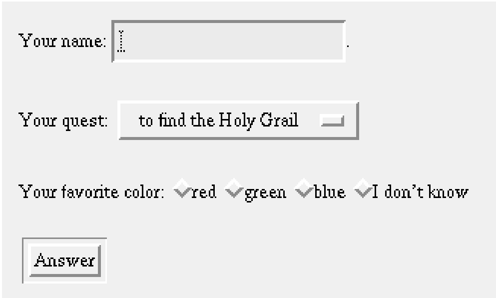
The `<form ...>` tag contains:

- transmission method (`POST` or `GET`);
- URL of the script; and
- a query string.

Extra tags for input fields:

- simple text fields;
- radio buttons;
- menus; and
- submit buttons.

A simple fill-out form:



HTML source for the fill-out form:

```
<form
   method="POST"
   action="http://www.brics.dk/cgi-mis/Python?Questions"
>
Your name:
<input name="name" type="text" size=20>.
<p>
Your quest:
<select name="quest">
<option value="grail">to find the Holy Grail
<option value="wig">to write a WIG compiler
</select>
<p>
Your favorite color:
<input name="color" type="radio" value="red">red
<input name="color" type="radio" value="green">green
<input name="color" type="radio" value="blue">blue
<input name="color" type="radio" value="argh">I don't know
<p>
<input name="submit" type="submit" value="Answer">
</form>
```


After filling out the form and clicking on the submit button, your browser sends following texts to the web server:

```
POST /cgi-mis/Python?Questions HTTP/1.0
Accept: www/source
Accept: text/html
......
User-Agent: ...  ...
From: ...
Content-type: application/x-www-form-urlencoded
Content-length: 47

name=Michael
&quest=wig
&color=blue
&submit=Answer
```

The web server parses the data from the client (e.g., browser), sets environment variables and input, and invokes CGI scripts.

Additional information is available in several UNIX environment variables (`http://www.cs.mcgill.ca/~hendren/cgi-bin/myenv.cgi?foo`) :

```
QUERY_STRING = foo
SERVER_ADDR = 132.206.51.10
HTTP_ACCEPT_LANGUAGE = en-us,en;q=0.5
SERVER_PROTOCOL = HTTP/1.1
HTTP_CONNECTION = keep-alive
REMOTE_PORT = 35406
HTTP_USER_AGENT = Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.4)
                  Gecko/20030624
HTTP_ACCEPT = text/xml,application/xml,application/xhtml+xml,
              text/html;q=0.9,text/plain;q=0.8,video/x-mng,
              image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
GATEWAY_INTERFACE = CGI/1.1
HTTP_HOST = www.cs.mcgill.ca
SERVER_ADMIN = help@cs.mcgill.ca
SERVER_SOFTWARE = Apache/2.0.43 (Unix) PHP/4.3.0RC2
SCRIPT_URI = http://www.cs.mcgill.ca/~hendren/cgi-bin/myenv.cg
REMOTE_ADDR = 132.206.3.136
SCRIPT_NAME = /~hendren/cgi-bin/myenv.cgi
SCRIPT_URL = /~hendren/cgi-bin/myenv.cgi
HTTP_ACCEPT_ENCODING = gzip,deflate
SERVER_NAME = www.cs.mcgill.ca
DOCUMENT_ROOT = /usr/local/www/data
```



```
REQUEST_URI = /~hendren/cgi-bin/myenv.cgi?Questions
HTTP_ACCEPT_CHARSET = ISO-8859-1,utf-8;q=0.7,*;q=0.7
REQUEST_METHOD = GET
SCRIPT_FILENAME = /u0/prof/hendren/public_html/cgi-bin/myenv.cgi
HTTP_KEEP_ALIVE = 300
PATH = /usr/local/bin:/usr/local/bin:/usr/bin:/bin
SERVER_PORT = 80
```


The script may be written in any programming or scripting language.

The form data appears on standard input as:

```
name=Michael&quest=wig&color=blue&submit=Answer
```

but must first be decoded:

- change `'+'` into a space character; and
- replace `%xy` by the ASCII character with hex value `xy`.

For example, `'='` and `'&'` must be encoded.

The dynamic document is supplied by the script on standard output:

```
Content-type: text/html
                              ⟵ important blank line
Hello Michael,
<p>
Good luck on writing a blue WIG compiler!
```

or may be redirected from a different document:

```
Location: URL
Content-type: text/html
```

How do we know it is really HTML?

We don't...

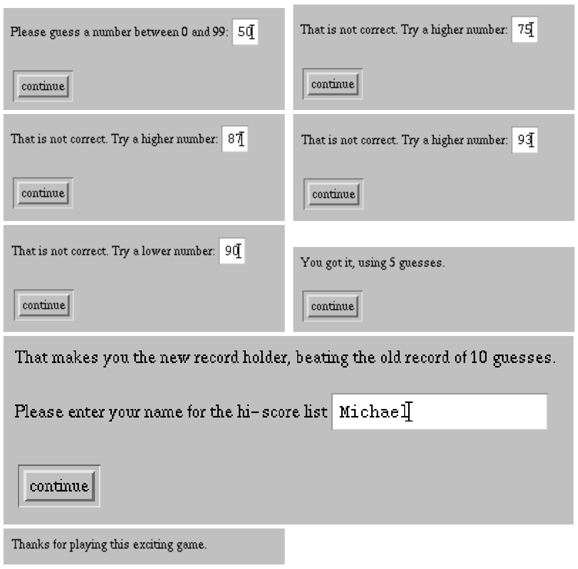
The CGI is a state-less protocol:

- each exchange happens in isolation;
- no information remains on the server; and
- different users cannot communicate.

We would like to have:

- global state;
- sessions;
- concurrent threads; and
- local state.

Interacting with a service:



The WIG language provides:

- global state;
- safe, dynamic documents;
- sequential sessions;
- multiple threads; and
- local state.

A WIG specification is compiled into a self-contained CGI-script.

The ubiquitous counter:

```
service {
  const html Nikolaj = <html> <body>
     <img src="http://www.brics.dk/~mis/babybath.jpg">
     <p>
     <i>You are visitor number <[no]></i>
  </body> </html>;

  int counter;

  session Access() {
    counter = counter + 1;
    exit plug Nikolaj[no = counter];
  }
}
```

Photo would appear here.

A one-player guessing game:

```
service {
  const html GetSeed = <html> <body> ... </body> </html>;
  const html GameSeeded = <html> <body> ... </body> </html>;
  const html Init = <html> <body> ... </body> </html>;
  const html Retry = <html> <body> ... </body> </html>;
  const html Again = <html> <body> ... </body> </html>;
  const html Done = <html> <body> ... </body> </html>;
  const html Record = <html> <body> ... </body> </html>;
  const html Finish = <html> <body> ... </body> </html>;
  const html List = <html> <body> ... </body> </html>;

  int plays, record;
  int seed;
  string holder;
...
}
```



```
int nextRandom()
  { int current;

    seed = (25173 * seed + 13849) % 65536;
    return(seed);
  }

  session Seed() {
    show GetSeed receive[seed = seed];
    exit GameSeeded;
  }
```



```
session Play() {
    int number, guesses, guess;
    string localholder;

    number = nextRandom() % 100;
    plays = plays+1;
    guesses = 1;
    show Init receive[guess = guess];
    while (guess>99) show Retry receive[guess = guess];
    while (guess!=number) {
       guesses = guesses+1;
       if (guess>number)
          show plug Again[correction = "lower"]
              receive[guess = guess];
       else
          show plug Again[correction = "higher"]
              receive[guess = guess];
       while (guess>99) show Retry receive[guess = guess];
    }
    show plug Done[trys = guesses];
    if (record==0 || record>guesses) {
       show plug Record[old = record]
         receive [localholder = name];
       holder = localholder;
       record = guesses;
    }
    exit Finish;
  }

  session HiScore() {
    exit plug List[plays = plays,
             holder = holder,record = record]; }
```



```
const html GetSeed = <html> <body>
    Please enter an integer seed for the random
                                  number generator:
    <input name="seed" type="text" size=5>
  </body> </html>;

  const html GameSeeded = <html> <body>
    Ok, now the game can proceed, the generator is seeded.
  </body> </html>;

  const html Init = <html> <body>
    Please guess a number between 0 and 99:
    <input name="guess" type="text" size=2>
  </body> </html>;

  const html Retry = <html> <body>
    That number is too large!
    <p>
    Please keep your guess between 0 and 99:
    <input name="guess" type="text" size=2>
  </body> </html>;

  const html Again = <html> <body>
    That is not correct. Try a <[correction]> number:
    <input name="guess" type="text" size=2>
  </body> </html>;
```

```
const html Again = <html> <body>
   That is not correct. Try a <[correction]> number:
   <input name="guess" type="text" size=2>
 </body> </html>;

 const html Done = <html> <body>
   You got it, using <[trys]> guesses.
 </body> </html>;

 const html Record = <html> <body>
   That makes you the new record holder,
   beating the old record of <[old]> guesses.
   <p>
   Please enter your name for the hi-score list
   <input name="name" type="text" size=20>
 </body> </html>;

 const html Finish = <html> <body>
   Thanks for playing this exciting game.
 </body> </html>;

 const html List = <html> <body>
   In <[plays]> plays of this game, the record
   holder is <[holder]> with <[record]> guesses.
 </body> </html>;
```


Syntax for WIG html:

```
htmls : html | htmls html ;
html : "const" "html" identifier "="
       "<html>" htmlbodies "</html>" ;

htmlbodies : /* empty */ | nehtmlbodies;
nehtmlbodies : htmlbody | nehtmlbodies htmlbody;
htmlbody : "<" identifier attributes ">"
         | "</" identifier ">"
         | "<[" identifier "]>"
         | whatever
         | meta
         | "<" "input" inputattrs ">"
         | "<" "select" inputattrs ">" htmlbodies
           "</" "select" ">";

inputattrs : inputattr | inputattrs inputattr;
inputattr : "name" "=" attr
          | "type" "=" inputtype
          | attribute;
inputtype : "text" | "radio";

attributes : /* empty */ | neattributes;
neattributes : attribute | neattributes attribute;
attribute : attr | attr "=" attr;
attr : identifier | stringconst;
```


Comments on WIG html:

- documents are implicitly forms;
- the `<[foo]>` tag defines gaps to be filled in dynamically;
- `<input...>` and `<select...>` tags are explicitly recognized; and
- all other tags and plain text are permitted but ignored.

Syntax for WIG statements:

```
stms : /* empty */ | nestms;
;
nestns : stm | nestms stm
;
stm : ";"
    | "show" document receive ";"
    | "exit" document ";"
    | "return" ";"
    | "return" exp ";"
    | "if" "(" exp ")" stm
    | "if" "(" exp ")" stm "else" stm
    | "while" "(" exp ")" stm
    | compoundstm
    | exp ";"
;
document : identifier
         | "plug" identifier "[" plugs "]";

receive : /* empty */
        | "receive" "[" inputs "]";

compoundstm : "{" nevariables stms "}"
            | "{" stms "}" ;

plugs : plug | plugs "," plug;

plug : identifier = exp;

inputs : /* empty */ | neinputs;
neinputs : input | neinputs "," input;
input : lvalue = identifier;
```

Syntax for WIG expressions:

```
exp : lvalue
    | lvalue "=" exp
    | exp "==" exp
    | exp "!=" exp
    | exp "<" exp
    | exp ">" exp
    | exp "<=" exp
    | exp ">=" exp
    | "!" exp
    | "-" exp
    | exp "+" exp
    | exp "-" exp
    | exp "*" exp
    | exp "/" exp
    | exp "%" exp
    | exp "&&" exp
    | exp "||" exp
    | exp "<<" exp
    | exp "\+" identifier
    | exp "\+" "(" identifiers ")"
    | exp "\-" identifier
    | exp "\-" "(" identifiers ")"
    | identifier "(" exps ")"
    | intconst
    | "true"
    | "false"
    | stringconst
    | "tuple" "{" fieldvalues "}"
    | "(" exp ")"
;
```



```
exps : /* empty */ | neexps;
neexps : exp | neexps "," exp;

lvalue : identifier | identifier "." identifier;

fieldvalues : /* empty */ | fieldvalues "," fieldvalue;
fieldvalue : identifier "=" exp;
```


Syntax for WIG types and functions:

```
schemas: /* empty */ | neschemas;
neschemas: schema | neschemas schema;
schema : "schema" identifier "{" fields "}";

fields : /* empty */ | nefields;
nefields : field | nefields field;
field : simpletype identifier ";";

simpletype : "int" | "bool" | "string" | "void";
type : simpletype | "tuple" identifier;

functions : /* empty */ | nefunctions;
nefunctions : function | nefunctions function;
function : type identifier "(" arguments ")" compoundstm;

arguments : /* empty */ | nearguments;
nearguments : argument | nearguments "," argument;
argument : type identifier;
```


Syntax for WIG sessions and services:

```
sessions : session | sessions session;
session : "session" identifier "(" ")" compoundstm;

nevariables : variable | nevariables variable ;
variable : type identifiers ";" ;
identifiers : identifier | identifiers "," identifier ;

service : "service" "{" htmls schemas
               nevariables functions sessions "}"
        | "service" "{" htmls schemas
               functions sessions "}"
;
```

Some open questions on WIG semantics:

- what happens if not all gaps are plugged?
- what happens if a gap is plugged twice?
- must all form inputs be received?
- what are the allowed operations on tuples?
- what are the type rules?
- are global variables safe for concurrent threads?

There are many such questions to ponder.

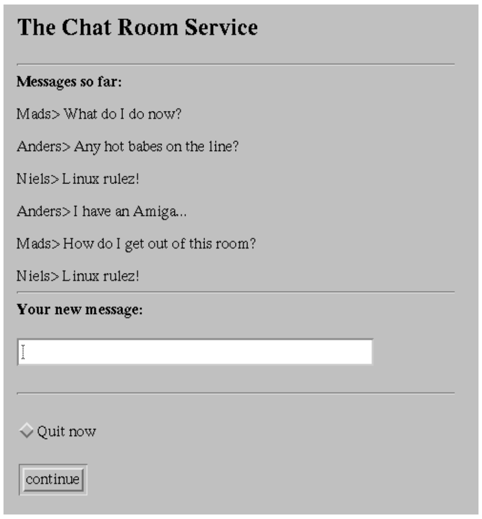
A simple chat room:

```
service {
  const html Logon = <html> <body>
    <h1>Welcome to The Chat Room</h1>
    Please enter your on-line name:
    <input name="name" type="text" size=25>
  </body> </html>;

  const html Update = <html> <body>
    <h1>The Chat Room Service</h1>    <hr>
    <b>Messages so far:</b>    <p>
    <[msg0]><p><[msg1]><p><[msg2]><p><[msg3]><p>
    <[msg4]><p><[msg5]><p>
    <hr>
    <b>Your new message:</b>
    <p>
    <input name="msg" type="text" size=40>
    <p>
    <hr>
    <p>
    <input name="quit" type="radio" value="yes"> Quit now
  </body> </html>;

  const html ByeBye = <html> <body>
    <h1>Thanks for using The Chat Room</h1>
    You made <[conns]> connections
    and wrote <[msgs]> messages.
  </body> </html>;

  string msg0,msg1,msg2,msg3,msg4,msg5;
...
```

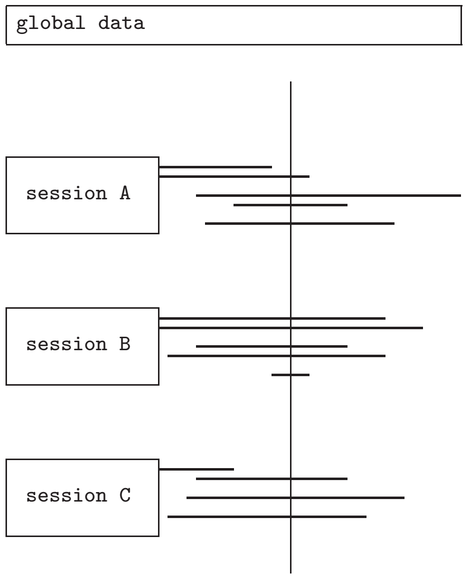


```
session Chat() {
    string name,msg,quit;
    int connections, written;

    show Logon receive [name = name];
    while (quit!="yes") {
      show plug Update[msg0 = msg0,
                       msg1 = msg1,
                       msg2 = msg2,
                       msg3 = msg3,
                       msg4 = msg4,
                       msg5 = msg5]
      receive[msg = msg, quit = quit];
      connections = connections+1;
      if (msg!="") {
        written = written+1;
        msg0 = msg1;
        msg1 = msg2;
        msg2 = msg3;
        msg3 = msg4;
        msg4 = msg5;
        msg5 = name + "> " + msg;
      }
    }
    exit plug ByeBye[conns = connections,
                     msgs = written];
  }
}
```


A sample chat:



Concurrent threads in a service:



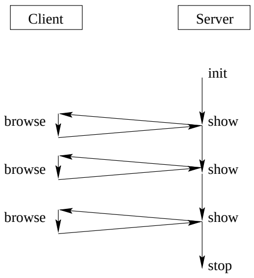
Maintaining global and local state:

- global variables reside in shared files;
- local variables reside in program variables inside each thread.

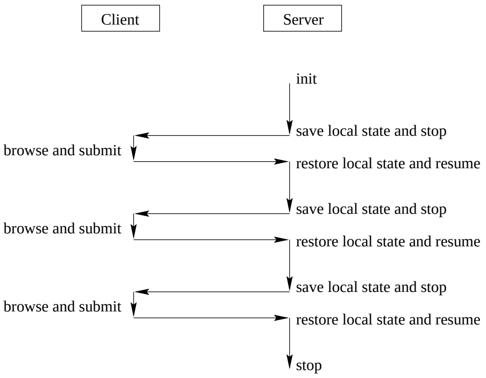
Emulating a sequential thread:

- each `show` causes the CGI-thread to save the local state and stop;
- each form submission causes the CGI-thread to resume and restore the local state.

A WIG session thread:



Corresponding CGI-threads:



Some synchronization issues and solutions:

- exclusive updates of global data:
  global file locking;
- critical regions:
  mutex semaphores.

Some security issues and solutions:

- tampering with the state:
  keep all state on the server;
- hijacking a session:
  use random keys in session id;
- backing up a thread:
  the server has the program counter.

A tiny WIG service:

```
service {
  const html Welcome = <html> <body>
    Welcome!
  </body> </html>;

  const html Pledge = <html> <body>
    How much do you want to contribute?
    <input name="contribution" type="text" size=4>
  </body> </html>;

  const html Total = <html> <body>
    The total is now <[total]>.
  </body> </html>;

  int amount;

  session Contribute() {
     int i;
     i= 87;
     show Welcome;
     show Pledge receive[i = contribution];
     amount = amount + i;
     exit plug Total[total = amount];
  }
}
```



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "runwig.h"

char *url;
char *sessionid;
int pc;
FILE *f;

void output_Welcome()
{ printf("Welcome!\n");
}

void output_Pledge()
{ printf("How much do you want to contribute?\n");
  printf("<input name=\"contribution\"
               type=\"text\" size=4>\n");
}

void output_Total(char *total)
{ printf("The total is now %s.\n",total);
}

int local_Contribute_i;

void main() {...}
```



```
/* initialize pseudorandom generator */
srand48(time((time_t *)0));
/* get form fields from CGI input */
parseFields();
/* assign the url of this service */
url = "http://dovs-www.daimi.aau.dk/cgi-mis/tiny";
/* find current sessionid from environment */
sessionid = getenv("QUERY_STRING");

/* do we start a new thread? */
if (strcmp(sessionid,"Contribute")==0)
   goto start_Contribute;
/* do we resume an old thread? */
if (strncmp(sessionid,"Contribute$",11)==0)
   goto restart_Contribute;
/* otherwise report an error */
printf("Content-type: text/html\n\n");
printf("<title>Illegal Request</title>\n");
printf("<h1>Illegal request: %s</h1>\n",sessionid);
exit(1);
```



```
/* start up a new thread */
start_Contribute:
/* initialize local variables */
local_Contribute_i = 87;
/* assign a random sessionid */
sessionid = randomString("Contribute",20);

/* show Welcome; */
printf("Content-type: text/html\n\n");
printf("<form method=\"POST\" action=\"%s?%s\">\n",
       url,sessionid);
output_Welcome();
printf("<p><input type=\"submit\" value=\"continue\">\n");
printf("</form>\n");
/* save local state */
f = fopen(sessionid,"w");
fprintf(f,"1\n");
fprintf(f,"%i\n",local_Contribute_i);
fclose(f);
/* terminate thread */
exit(0);
/* and resume from here */
Contribute_1:
```

```
/* show Pledge... */
printf("Content-type: text/html\n\n");
printf("<form method=\"POST\" action=\"%s?%s\">\n",
       url,sessionid);
output_Pledge();
printf("<p><input type=\"submit\" value=\"continue\">");
printf("</form>\n");
/* save local state */
f = fopen(sessionid,"w");
fprintf(f,"2\n");
fprintf(f,"%i\n",local_Contribute_i);
fclose(f);
/* terminate thread */
exit(0);
/* and resume from here */
Contribute_2:

/* ...receive[i = contribution]; */
local_Contribute_i = atoi(getField("contribution"));
/* amount = amount + i; */
putGlobalInt("global_tiny_amount",
             getGlobalInt("global_tiny_amount")
             +local_Contribute_i);
/* exit plug Total[total = amount]; */
printf("Content-type: text/html\n\n");
output_Total(itoa(getGlobalInt("global_tiny_amount")));
exit(0);
```



```
/* restart a thread */
restart_Contribute:
/* restore local state */
f = fopen(sessionid,"r");
fscanf(f,"%i\n",&pc);
fscanf(f,"%i\n",&local_Contribute_i);
/* jump to current pc */
if (pc==1) goto Contribute_1;
if (pc==2) goto Contribute_2;
```


The library `runwig.h` implements:

```
void parseFields();
char *getField(char *name);

char *randomString(char *name,int size);

int getGlobalInt(char *name);
void putGlobalInt(char *name,int value);

char *itoa(int i);
```


The service is installed as:

```
#!/bin/csh
gcc tiny.c /u0/prof/hendren/WIG/04/runwig.c -o tiny4.cgi
cp tiny4.cgi ~/public_html/cgi-bin
chmod 755 ~/public_html/cgi-bin/tiny4.cgi
```

and is invoked by:

```
http://www.cs.mcgill.ca/~hendren/cgi-bin/tiny.cgi?Contribute
```