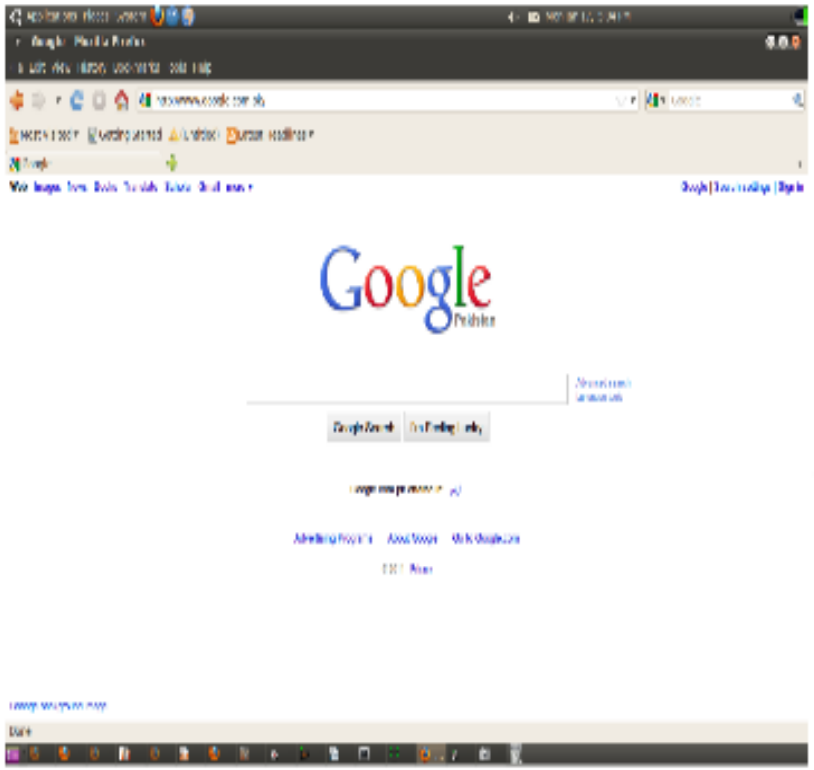
Server-side Programming



1. client requests a resource

2. server sends a response (may attach an entity)

3. client interprets the response

static resources
(html,xml,images,etc)

URI 1

URI 2

URI 3

URI 4

Web Server

dynamic resources
(programs / process)

Static v/s Dynamic Resources

- Static Resources
  - fixed representation: delivered to agents exactly as stored
  - examples: html, xml, images, etc
  - no programming required
  - maintaining large number of static resources can be cumbersome and difficult
- Dynamic Resources
  - reside in form of processes / programs
  - generate client-readable content on the fly
  - representation may change over time
  - example: Google Search

Job of a Web Server

- Organize content / resources
- Serve incoming requests
- Path translation
- Generate response
- Other Issues
  - Security
  - Performance
  - Virtual Hosting
  - URL rewriting
  - Logging
  - Filters

Node.js

- Run-time environment for Javascript-based server side programming using V8
- Written in C, C++ and Javascript
- Event-driven, Single threaded, Asynchronous, Non-blocking I/O
- Useful for I/O intensive applications but not for CPU intensive applications
- Provides several modules:
  - http: a simple in-built web server
  - fs: file system handling
  - url: URL parsing
  - Many others …

Hello world Example

helloworld.js

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});

  response.write('Hello World\n');
  response.end();
}).listen(8888);
```

Command line

```
> node helloworld.js
```

Path Translation

pathtranslation.js

```
var http = require('http');
var fs = require('fs');

http.createServer(function (request, response) {

  basePath = "C:\\nodeprojects\\pathtranslation";  // root folder

  filePath = basePath + request.url;

  fs.readFile(filePath,function(err,contents){
      if (err) {
            response.writeHead(404, {'Content-Type': 'text/html'});
            response.end("404 Not Found");
      }
      else{
            response.writeHead(200, {'Content-Type': 'text/html'});
            response.write(contents);
            response.end();
      }
  });

}).listen(8888);
```

Search Engine Example

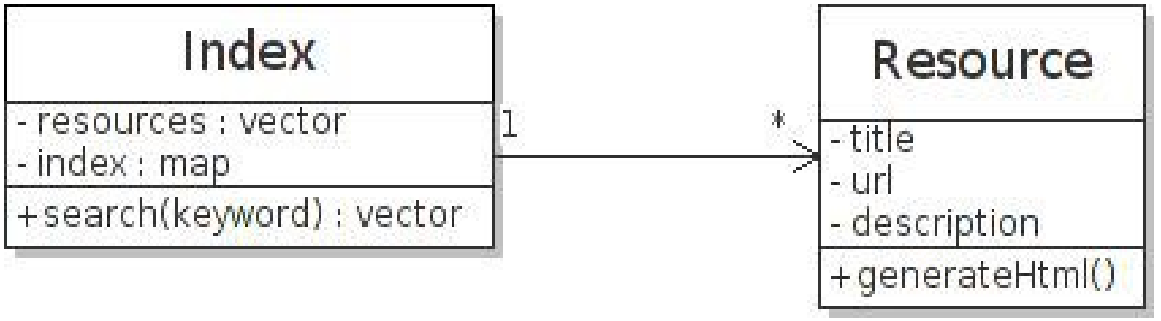
- Crawls internet and find resources
  - Stores resources information in an index
  - Each resource may be mapped to several keywords
- To serve a user search request
  - Search the index against supplied keyword
  - Find all matching resources
  - Rank and sort the results
  - Return corresponding html to user agent

```
<html>
	<body>
		<form method="GET" action="./search">
			<input type="text" name="q" />
			<input type="submit" value="Search" />
		</form>
	</body>
</html>
```

Google
Google Search Engine

Yahoo
Yahoo Search Engine

Node JS implementation




```
class Index {
        constructor(){
                this.resources = [];
                this.index = [];

                this.resources[0] = new
                                Resource('Yahoo',
                                'https://www.yahoo.com',
                                 'Yahoo Search Engine');


                // …

                this.index['search engine'] =
                        [this.resources[0],this.resources[1]];
        }

        search(keyword){
                return this.index[keyword];
        }

}
```

```
class Resource{
        constructor(t,u,d){
                this.title = t;
                this.url = u;
                this.description = d;
        }

        generateHtml(){
                var html = "<p>";
                html += "<a href='" + this.url + "'>" + this.title +
                                "</a>";
                html += "<br/>";
                html += "<p>" + this.description + "</p>";

                return html;
        }
}
```

```
var http = require('http');
var fs = require('fs');
var url = require('url');

Engine = require('./engine.js');


http.createServer(function (request, response) {

  basePath = "C:\\nodeprojects\\searchengine";

  filePath = basePath + request.url;

  if(request.url == "/" || request.url == "/index"){

        filePath = basePath + "\\index.html";

        fs.readFile(filePath,function(err,contents){
            response.writeHead(200, {'Content-Type':
                                     'text/html'});
            response.write(contents);
            response.end();
        });
  }
```

```
  else{
        var u = url.parse(request.url,true);
        var qs = u.query;
        var index = new Engine.Index();
        var resources = index.search(qs.q);
        var html = "";

        for(i=0; i < resources.length; i++)
              html += resources[i].generateHtml();

        response.writeHead(200, {'Content-Type':
                                 'text/html'});
        response.write(html);
        response.end();
  }


}).listen(8888);

console.log("server listening at port 8888");
```

request handling

response handling

Server-side Programming
Key issues

- Request / Response handling
- Form Validation
- State Management
- Application Architectural issues
- Language and Framework selection
- Engineering issues
  - Managing security, performance, usabilty and accessibility, etc.

Form Validation

- Possible approaches
  - Client-side
    - Better response time
    - Insecure – can be easily bypassed
  - Server-side
    - Not as responsive as client-side
    - Secure – not easy to bypass by end-users
    - State management required
  - Client-side + Server-side
    - Combines benefits of both: performance + security

Client-side validation

```
<html>
	<head>
		<script type="text/javascript">
			function validate(){
				var result = true;
				 result = result && emptyCheck('uid','User');
				 result = result && emptyCheck('pw','Password');
				return result;
			}
			function emptyCheck(id,fname){
				if (document.getElementById(id).value == ""){
					alert(fname + ' cannot be empty');
					return false;
				}
				return true;
			}

		</script>
	</head>
	<body>
		<form method="POST" action="loginhandler">
		    User <input type="text" name="uid" id=”uid” />
		    Password <input type="password" name="pw" id=”pw” />
		    <input type="submit" value="login" onclick=”validate()”>
		</form>
	</body>
</html>
```

Server-side validation

loginform.html

```
<html>
<body>
	<form method="POST" action="loginhandler">
		User <input type="text" name="uid" id=”uid” />
		Password <input type="password" name="pw" id=”pw” />
		<input type="submit" value="login">
	</form>
</body>
</html>
```

main.js

```
var http = require('http');
var url = require('url');

http.createServer(function (request, response) {

	var u = url.parse(request.url,true);
	 If (u.pathname == “loginhandler”){

		if( u.query.uid == “” || u.query.pw == “”){
			// redirect to loginform with error message
		}
		else{
			// process login
		}
	}

}).listen(8888);
```

State Management

- HTTP is stateless i.e.
  - Every request from client to server is unique
  - Server doesn't remember the client and the last operation the client performed
  - Suitable for web sites where all you need is to access the statically available information
  - Helpful because server doesn't need to know you in order to serve you – so all content is accessible
- Need for state
  - When we talk about web applications that generate dynamic content based upon the users/clients
  - Server need to know the user
  - May be the last action performed e.g. login, logout, search, etc.
  - Example: gmail, piazza, etc.

State Management

- Client-side
  - URLs
  - Form hidden fields
  - Cookies
- Server-side
  - Sessions

Managing state on the client-side

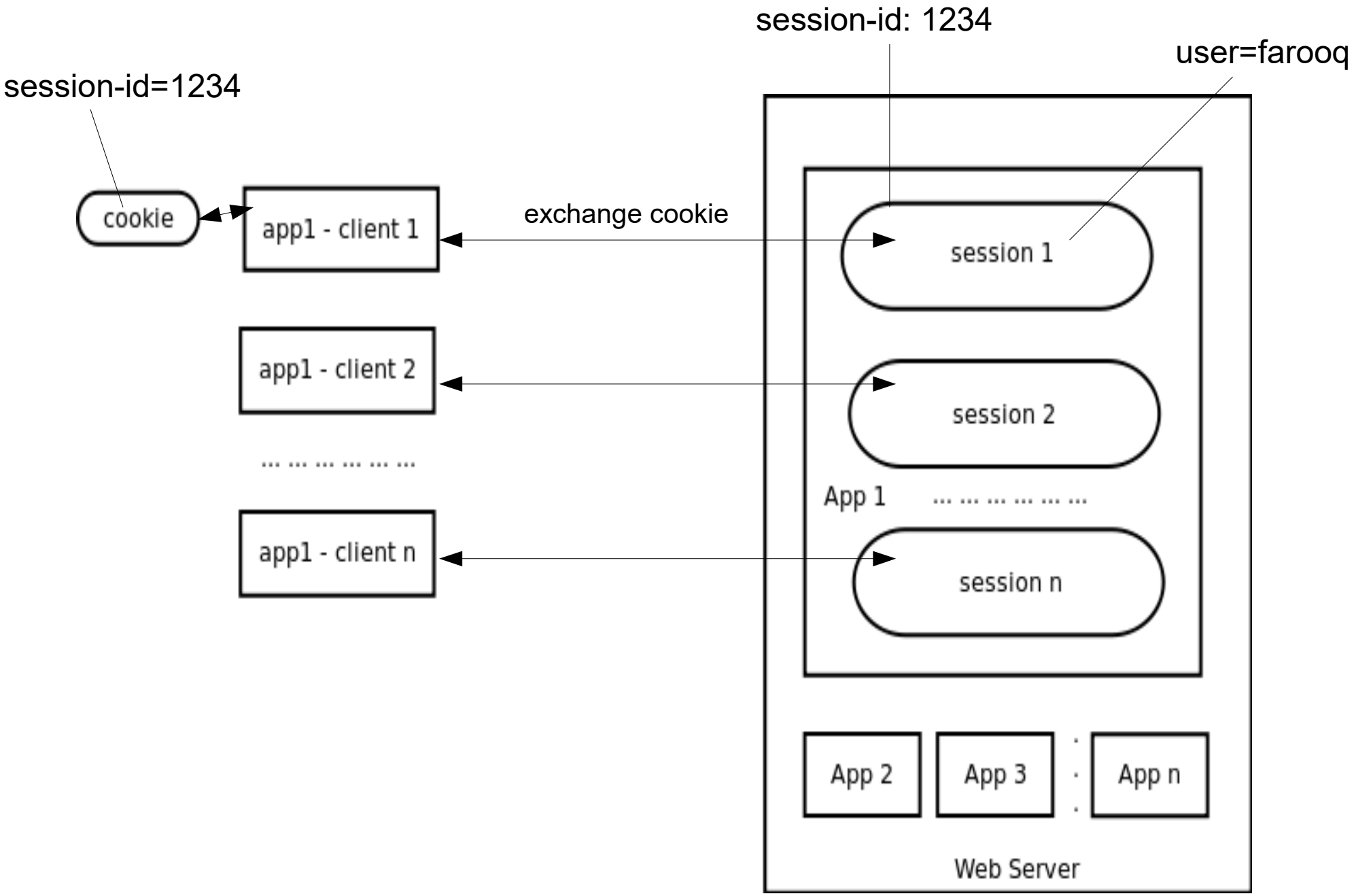
Always send the state information as part of every request. For example, consider using gmail:

- In order to login, type your user-id and password
- Server authenticates you and show you your **inbox**
- Server asks the client to store user-id information
  - Either using URL, Hidden fields, Cookies
- Server terminates the connection due to stateless nature
- Now, in order to view first unread email you select the **email-id** and resend your **user-id**
  - This information can be sent either using URL, hidden fields or cookies
- Server:
  - gets your **user-id,**
  - locate your inbox based upon user-id
  - lookup the email in your inbox based upon the **email-id**

Managing state on the server-side

Server maintains the state – you don't have to send it every time you make the request. For example, again consider using gmail:

- In order to login, type your user-id and password
- Server authenticates you and show you your **inbox**
- Server stores your user-id in the session for a specific time-period
- Server terminates the connection due to stateless nature
- Now, in order to view first unread email you select the **email-id** ~~and resend your **user-id**~~
  - ~~This information can be sent either using URL, hidden fields or cookies~~
- Server:
  - gets your **user-id** from session**,**
  - locate your inbox based upon user-id
  - lookup the email in your inbox based upon the **email-id**
  - Clears your session once the time-period expires (session timeout)


session-id: 1234
user=farooq
session-id=1234
cookie
app1 - client 1
exchange cookie
session 1
app1 - client 2
session 2
... ...
App 1
... ...
app1 - client n
session n
App 2
App 3
App n
Web Server

Cookies vs Session

- Stored on client
- More prone to attacks
- Lower Performance
- Preferred for long-term storage
- Easy to scale

- Stored on server
- Safe
- Better performance
- Preferred for short-term storage
- Difficult to scale

HTTP Persistent Connection

- Uses same TCP connection for multiple request/response cycles
- Can be used by setting header as:

  Connection: Keep-alive

- An optimization to save time lost in creating/dropping connection
- **Does not affect stateless nature**