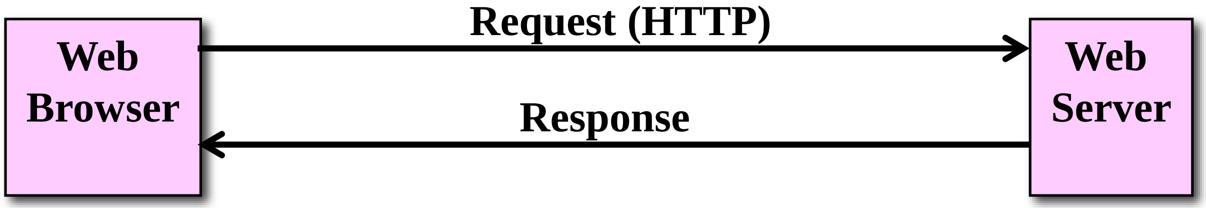
NodeJS HTTP

HTTP Revisited

HTTP is the protocol of the web

- An *application layer* **protocol** typically built on TCP/IP
- **Synchronous** (what does this mean?)
- **Stateless** (what does this mean?)
- **Connection[-oriented]** (… stay tuned)



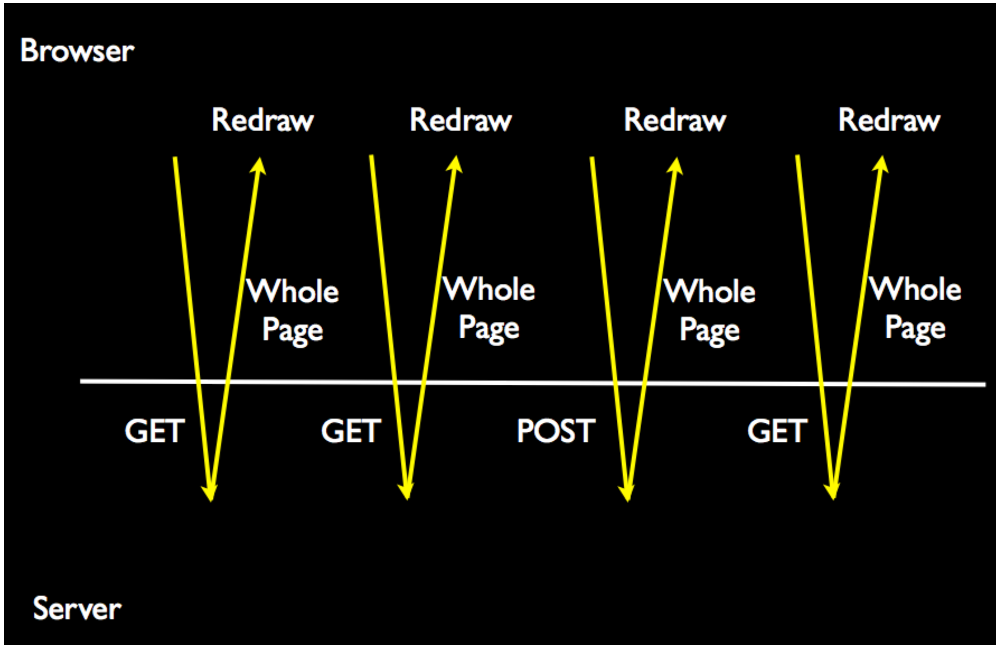
HTTP supports various "methods"

- GET: Makes a request on a resource
- POST: Used to pass input to the server
  - GET encodes on the URL, POST puts it in the body of the message
  - POST handles binary payloads, makes them invisible, and can support input requests of an arbitrary length (w/ Content-Length)
- Others: OPTIONS, PUT, DELETE, CONNECT, TRACE, PATCH
  - We'll cover these later with REST

Server-side Web Applications

The server-side web app pattern:

- A user takes some action like a click on a link or button
- The browser makes a TCP/IP connection to the web server
- The browser sends a POST or GET request
- The server sends back a rendered response to display to the user
- Lather-Rinse-Repeat...



<u>NodeJS: The HTTP module</u>

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

- `http` is a low-level module in NodeJS
- You will normally create your web applications using a convenience/MVC module like Express or HapiJS (stay tuned…)

HTTP Strategy

All server-side web applications follow a general pattern:

1. Process HTTP request headers
   - Figure out who is sending the request & what type of response they seek
2. Process HTTP request parameters
   - From the query string
3. Route the request to the appropriate handler to do the work
   - Remember, your web interface is just an adapter
4. Assemble the response payload
   - You (the web application programmer) are responsible, in whatever language or framework, for processing the request header and payload
5. Set the response headers
   - What is your status code, content-type, custom headers, etc.
6. Write the response
   - Don’t forget to flush/close the output stream!

This pattern is the Template pattern (Go4) combine with the Strategy pattern for step 3. These steps are the basic processing pipeline

HTTP Server

1st example was in NodeIntro notes, this one better

Example (http_server_static.js):

Cyan circles are the steps

1 There is no step 1 here

```
  var fs = require('fs');
  var http = require('http');
  var url = require('url');
  var ROOT_DIR = "html/";
  http.createServer(function (req, res) {
2   var urlObj = url.parse(req.url, true, false);
3   fs.readFile(ROOT_DIR + urlObj.pathname, function (err,data) { 4
      if (err) {
      5 res.writeHead(404);
        res.end(JSON.stringify(err));
        return;
      }
5     res.writeHead(200);
      res.end(data); 6
    });
  }).listen(8080);
```

Create the http server, and on each request execute the callback

If we can’t find the file, return 404

Otherwise all good, and write the data

What port (and optionally hostname) we listen on

HTTP Server Explained

Pretty straightforward actually:

- `createServer` returns an `http.Server` object
- The callback listens for requests
  - When a request is received, an `http.ServerResponse` object is created and bound to the 2nd callback parameter
  - `ServerResponse` inherits from `WritableStream`, so we can write to it
  - You can manipulate response headers, timeout sockets, set the statusCode, and do other expected things on this object
  - The 1st parameter (req) is again an IncomingMessage object (remember, implements ReadableStream) as before.
    - Read the documentation on IncomingMessage carefully; while they recognized that conceptually a socket is reading information, there are some methods and events that are specific to whether it is a client or a server
- You can also listen for other events like when a new connection is established to the server (for logging).
- The listen call accepts a longer form:

```
listen(8080, ‘localhost’, 3, function(){
                    console.log(‘I am ready!’);
  });
```

NodeJS URL Module

Low-level built-in package → `require(‘url’)`

- Allows you to parse and format URLs to marshal data to and from a JS object

```
$ var urlObj = require(‘url’).parse(‘http://www.asu.edu/relpath?action=list’);
$ Console.log(urlObj);
```

```
 Url {
  protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'www.asu.edu',
  port: null,
  hostname: 'www.asu.edu',
  hash: null,
  search: '?action=list',
  query: 'action=list',
  pathname: '/relpath',
  path: '/relpath?action=list',
  href: 'http://www.asu.edu/relpath?action=list' }
```

```
$ urlString = require(‘url’).format(urlObj);  // converts back to a string
```

Dealing with Query Strings

Query strings can be parsed from a URL for property “query” as we saw on the previous page

- However there is another module, “querystring” which provides convenience methods for parsing the query string

```
$ var q = require(‘querystring’);
$ var qs = q.parse(urlObj.query);
$ console.log(qs);
    { action: ‘list’ }
```

You can create the reverse using stringify (in the event you are constructing URLs dynamically to embed in HTML)

```
$ q.stringify(qs);
  ‘action=list’
```

Simple Dynamic Behavior in an HTTP Server

Suppose we want to take some dynamic behavior based on a parameterized GET query:

```
// http_server_get.js
var http = require('http');
var url = require('url');
var messages = ['Hello World', 'From a Node.js server', 'Take Luck'];
http.createServer(function (req, res) {
    var resBody = ‘’;
    var resMsg = ‘’;
    var urlObj = url.parse(req.url, true, false);     2
    var qstr = urlObj.query;
3   if (!qstr.msg) {
        resMsg = ‘<h2>No msg parameter</h2>\n’;
    } else {
        resMsg = '<h1>'+messages[qstr.msg]+'</h2>’;
    }
4   resBody = resBody + '<html><head><title>Simple HTTP Server</title></head>’;
    resBody = resBody + '<body>’ + resMsg;
    res.setHeader("Content-Type", "text/html");     5
    res.writeHead(200);
    res.end(resBody + '\n</body></html>');     6
}).listen(8080);
```

Again no 1 in this example

HTTP POST

What if we POST to the previous program?

- NodeJS didn’t care! Request could come in via GET or POST
- If we want to take different action or only support a certain method, we have to check for that method using the request object’s method property
  - See http_server_external.js

```
http.createServer(function (req, res) {
  console.log(req.method);
  if (req.method == "POST"){
    var reqData = '';
    req.on('data', function (chunk) {
      reqData += chunk;
    });
    req.on('end', function() {
      var postParams = qstring.parse(reqData);
      getWeather(postParams.city, res);
    });
  } else{
    sendResponse(null, res);
  }
}).listen(8080);
```

OK here
Is a 1

And this chunking is your 4

Cookies: A Client-side State Solution

Idea

- Server sends a simple name and value to client.
- Client returns same name and value when it connects to same site (or same domain, depending on cookie settings).
- Note that the server sets the cookie in its response header, but the client (browser) does not have to accept it.

Typical Uses of Cookies

- Identifying a user during an e-commerce session
- Avoiding username and password (bad!)
- Customizing a site (personalization)
- Focusing advertising (recommenders)

Basically, a very simple mechanism for sending custom per-browser information back to the server – makes it seem like the server knows you have been here before!

Adapted from Core Servlets and JavaServer Pages, © 2008, Marty Hall. Reused with permission

Different kinds of cookies

Chocolate chip, oatmeal, sugar, molasses… wait

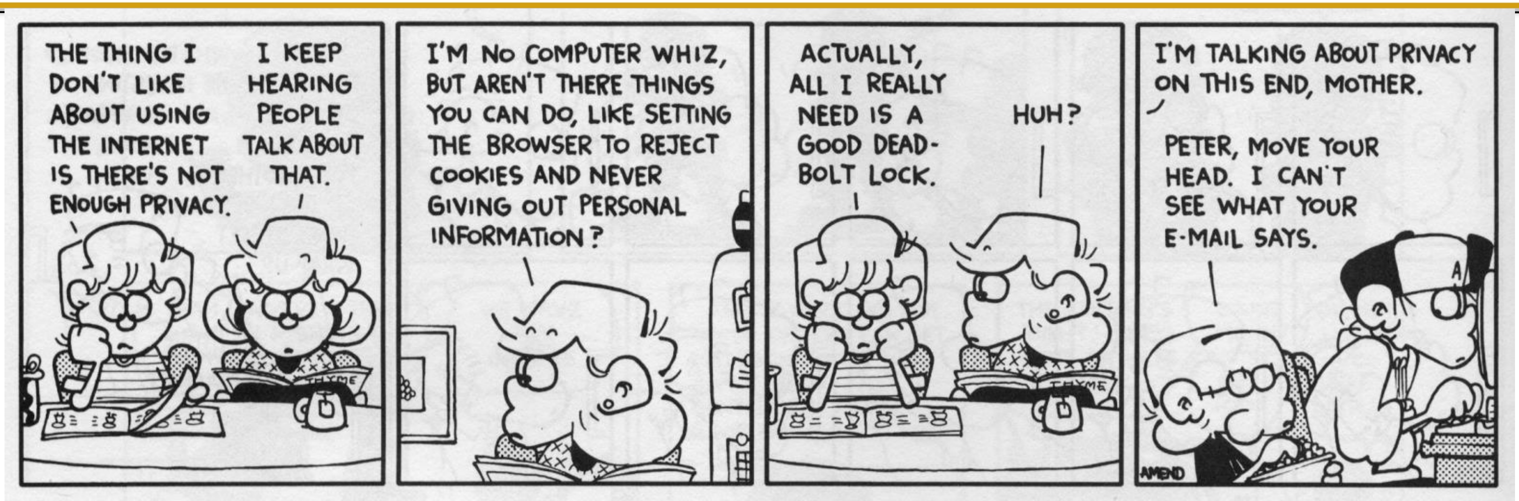
Session cookie

- Only used in memory in that session of the browser instance
- You close the browser, the cookie is gone

- Persistent cookie
  - Lasts for a specific time period
- Secure cookie
  - Only transmitted over HTTPS by indicating a Secure flag
  - If HTTP attempted it is not sent
  - Protects against snopping or “cookie stealing”
- HttpOnly
  - Prevents reading cookie values in Javascript by giving HttpOnly flag
  - Avoids having malicious JS compromise cookie info
- 3rd party vs. Samesite
  - Whether a cookie can originate from a domain other than where served from

Latest spec: https://tools.ietf.org/html/rfc6265

Cookies and Privacy


THE THING I DON'T LIKE ABOUT USING THE INTERNET IS THERE'S NOT ENOUGH PRIVACY.
I KEEP HEARING PEOPLE TALK ABOUT THAT.
I'M NO COMPUTER WHIZ, BUT AREN'T THERE THINGS YOU CAN DO, LIKE SETTING THE BROWSER TO REJECT COOKIES AND NEVER GIVING OUT PERSONAL INFORMATION?
ACTUALLY, ALL I REALLY NEED IS A GOOD DEAD-BOLT LOCK.
HUH?
I'M TALKING ABOUT PRIVACY ON THIS END, MOTHER.
PETER, MOVE YOUR HEAD. I CAN'T SEE WHAT YOUR E-MAIL SAYS.
THYME
AMEND

FoxTrot © 1998 Bill Amend. Reprinted with permission of Universal Press Syndicate. All rights reserved.

The problem is privacy, not security.

- If you give out personal info, servers can link it to previous actions
- Servers can share cookie info through 3rd parties like doubleclick.net
- Poorly designed sites store sensitive info (e.g. credit card #) in cookie

Moral for web application authors

- If cookies are not critical to your task, avoid developing apps that totally fail when cookies are disabled (or check for cookie disabling)
- Don't put sensitive info in cookies

HTTP API summary

It’s all here: https://nodejs.org/api/http.html

- Focus on
  - Class http.server to encapsulate an http server
  - Class http.IncomingMessage
  - Class http.ServerResponse
  - Function http.createServer
  - Functions http.request, http.get
- Cookies are set in the response header explicitly
  - This is the “long way”
  - In the future we will see some convenience mechanisms
- Cookies and State Management
  - Cookies are not a great solution for storing conversational state
    - Cookie may be poisoned or stolen
    - Limited in what they can store anyway
    - Especially bad if it is a *persistent cookie*
  - Cookies are a more reasonable solution for user preferences
    - Provided that doesn’t mean keeping sensitive info on the client

Summary

HTTP module

- The http module in node is very low-level
- But http by itself is not that complex! What gets complex is how we want to handle incoming requests
  - Route requests to the write service (Controller)
  - Abstract out presentation (View)
  - Decouple our world state (Model)
  - Manage interactions (Conversational State)

Cookies

- Simple name-value pairs set (or strictly, “request to be set”) by the server
- Stored in the browser
- Sent on each subsequent request to that domain (URL)
- Typical use case is *personalization*, not *authentication* or *conversational state*