Servers

Servers allow web sites to do things, not just display things

choices  test server  URLs  routes
For a dynamic web site, you have to choose a web server, with a *framework* and a scripting or development *language* for writing the pages:

- CGI: Perl or C (obsolete)
- ASP: VBScript or C# (proprietary/1-platform/obsolete)
- JSP: Java (clunky)
- LAMP: PHP
- Django: Python
- Rails: Ruby
- Node: JavaScript
Two choices

For this course, two choices are recommended, and will be covered in this chapter:

**Raw** Node, with standard libraries only

**Express** Node, with express and/or other libraries

Take the raw approach to learn more, take the express approach to get further
If you are using Node (raw or express), the development server `server.js` provided at the start of the course contains some potentially useful details.

But remember that it was designed for development and not production:

It only allows connections from the same computer, and its security is otherwise weak and, in particular it has a serious security vulnerability.
Preliminary security

Security is essential for a potentially public-access service - we will come back to it several times

Step one is to have a dedicated computer for a web site, and put it somewhere physically secure, e.g. a locked machine room

Step two is to restrict login to that computer to a limited number of other computers or people or both

Step three is to establish a firewall, especially for an intranet
In this chapter, we will concentrate on the raw approach, in order to work out what the various issues are, and summarise the other approaches.

The first thing to do is to find out what kinds of request there are, and what information they contain.

So, let's set up a minimal Node server which prints out request details - it may be helpful to keep it around for future use as well.
The documentation is in the Node docs for the http and later the https modules.

The documentation is very difficult to make sense of because (a) it includes very low level things we don't need (b) it mixes client and server info (c) it isn't explicit about types (d) it is poorly explained (e) it assumes the callback style.

Two important pieces of information are that a request has type IncomingMessage and a response has type ServerResponse.
// Minimal server
let HTTP = require('http');
start(8080);
...

Download server1.js to experiment

The 'http' module comes with Node, and so does 'https'

Use 8080, to add firewall security, to avoid root/administrator privileges, or if port 80 is being used by another program
... Provide a service to localhost only.

```javascript
function start(port) {
    let service = HTTP.createServer(handle);
    service.listen(port, 'localhost');
}
```

The server is now listening for browser requests, but only from the computer it is running on (do this while developing, until you are happy with your security)

Every request will cause a call to the `handle` function
Handling requests

```javascript
// Deal with a request.
function handle(request, response) {
    console.log("Method:", request.method);
    console.log("URL:", request.url);
    console.log("Headers:", request.headers);
    reply(response);
}
```

Print details - the most important fields of the request

Look up `http.IncomingMessage` in the Node API docs for the 'http' module to get request details
Responding

... Send a reply.

```javascript
function reply(response) {
  let hdrs = {
    'Content-Type': 'text/plain'
  };
  response.writeHead(200, hdrs);  // 200 = OK
  response.write("OK");
  response.end();
}
```

Send a minimal response (as if delivering a plain text file with one line containing OK)

Look up `http.ServerResponse` in the Node API docs for the 'http' module to get `response` details
When you visit http://localhost:8080/ with your browser, the server will describe the request.

You are quite likely to see two requests, one for URL / which you asked for, and one for URL /favicon.ico which is the browser asking for the site icon.

The test server responds to every request the same way.
let HTTP = require('http');
let FS = require('fs').promises;
let OK = 200, NotFound = 404, BadType = 415;
start(8080);

The next server server2.js delivers HTML pages

Names are defined for success and failure codes

start can't be called until global data has been defined
(some people call it at the bottom of the script)
Add `index.html` if a URL ends with `/`, and give an error if it is not `.html`, return after calling `fail` to stop further processing, convert the url into a file path

Read the file and use the content to reply
function reply(response, content) {
    let hdrs = {
        'Content-Type': 'application/xhtml+xml'
    }
    response.writeHead(OK, hdrs);
    response.write(content);
    response.end();
}

the content comes from readFile

Deliver file content as application/xhtml+xml
(can only handle web pages)
In the case of an error, deliver a one-line plain text response, with a given error code

```javascript
function fail(response, code, message) {
    let hdrs = {
        'Content-Type': 'text/plain'
    };
    response.writeHead(code, hdrs);
    response.write(message);
    response.end();
}
```
Security inside the server is a difficult topic; the first line of defence is to validate the URL in each request.

**Warning:** the original server.js provided doesn't do full URL validation.

**Warning:** express doesn't do full URL validation.

**Warning:** other servers/frameworks don't do full URL validation.

Security is your responsibility.
File systems

The main problem is that servers convert the URL path into a file path by prefixing the root folder.

Then they get the file system to find the resource.

That means there are security loopholes, depending on odd features of the file system.
An obvious attack is to use .. in URLs to try to reach files outside the root folder.

Suppose you have a root folder /site/public and a sensitive file /site/secret.txt.

An attack might be to visit /products/../../secret.txt.

This must be prevented and it is strongly recommended that you test it (express and other servers do prevent it, if you use them properly).
Testing can be difficult, e.g. if you type a URL directly into your browser, and then get the server to print out the URL, you may find that the browser has changed the URL before sending the request (especially `/../`).

The same happens with tools like `wget` or `curl`.

For total control, write a program such as `get.js`.
Suppose you have an admin folder inside your site.

And you have password protection of the form "if a URL starts with /admin/ then require a password"

Attacks may use synonyms to bypass the protection
/products/..../admin/adduser
./admin/adduser
//admin/adduser

Express and other servers (and browsers) handle /.../ and /./ but not // so test this!!!
Restricting URLs

The simplest approach to solve these problems is to validate the URL.

Simplest is to reject a URL if it contains / . or //, or contains non-ascii characters, or doesn't start with /, or is too long, see i dotdot, i hidden, i spaces

Safer is to check that the URL matches one of the strings or patterns that you regard as OK, and i reverse your security checks
If you reject URLs containing /., then you are rejecting /./, which is good, but also /../ which is used quite commonly.

But that's probably OK, because /../ is usually resolved by the browser before the request gets to the server.

If you prefer, you can canonicalise the URL, and then reject it if it starts with .. which indicates that it is trying to climb out of the public folder, and then apply security restrictions to the canonical URL.
Hidden files

If you reject URLs containing /., then you are rejecting hidden files / .hidden, which may be a good thing.

On Linux and MacOs, any file or directory whose name starts with a dot is automatically hidden.

On Windows, a flag is used for hidden files, and names starting with a dot have no special meaning, but many tools such as git create files or folders that start with dot and are marked as hidden.

So probably a dot file is hidden, even on Windows.
Spaces

Spaces are officially only allowed in URLs if they are escaped as `%20`

If you are going to allow them, you need to work out (a) how different browsers handle them (b) what gets sent to your server (c) how to handle them safely in your server on your file system (d) what effect, probably confusion, they will have on your users

It is almost always simpler and better to ban them
Need to know

Security can be improved by reversing it (on the 'need to know' principle)

Suppose your site has just two folders, products and admin

Instead of "if a URL starts with /admin/ then require a password" do "if a URL doesn't start with /products/ then require a password"

That would catch more of the problems without having to think too hard
Attackers might try `/ADMIN/adduser` to get past your password protection on `/admin/...` 

Usually Linux servers are case sensitive and Windows servers are case insensitive (the filestore decides!)

I recommend making all requests case sensitive, as in the provided server, and using a [whitelist](#) or [blacklist](#) for exceptions
Another thing that needs to be done is to prevent visitors from accessing files which are not intended to be accessible.

That means e.g. being careful not to put the server program, encryption keys, database files, config files, source code, editor backups, hidden files, and so on inside the public root folder.

It is also a bad idea to have hard links, symbolic links, shortcuts, or anything equivalent, inside a root folder.
Content types

Each successful response needs a content type: the norm is to maintain a table in the server code which maps file extensions to content types (or use the mime module)

```javascript
let types = {
  css : 'text/css',
  js : 'application/javascript',
  ...
}

let type = types[extension];
let hdrs = { 'Content-Type': type }; response.writeHead(200, hdrs);
...
What happens with a URL like `/project/Makefile` which doesn't have an extension?

The server doesn't know, just from the URL, whether it is a file with no extension or a folder with no /.

If it is a file, the server doesn't know what type to use, so the default is often to treat it as a folder.

If it is folder, you **must** redirect the browser, so it is simpler to reject such URLs (maybe with exceptions).
What happens if you treat /x/y as /x/y/ and deliver the file /x/y/index.html?

If there is a relative link on that index page, the browser thinks it is relative to /x/ and not /x/y/ so the link fails

So you must redirect the browser, by sending a response with the corrected URL (using code 301, 302, 303, 307, 308 - good luck working out which is correct)

I think it is better to avoid the inefficiency of multiple requests by banning the original URL
For XHTML, content negotiation is needed to serve an .html page to old and new browsers.

```javascript
// For the .html extension:
let otype = "text/html";
let ntype = "application/xhtml+xml";
let header = request.headers.accept;
let accepts = header.split(",");
if (accepts.indexOf(ntype) >= 0) type = ntype;
else type = otype;
```

**Warning:** express and other servers deliver all HTML pages as "text/html" by default.
Dynamic URLs

For dynamic resources, e.g. pages generated from database data, it is up to you to invent URL conventions.

As well as dynamically generated HTML pages, dynamic URLs may present requests to the server to update the database, or deal with an upload - you could call these action URLs.

There are lots of possibilities, including using express.
In an express server, I personally find it extremely difficult to understand what happens to a particular request, or to debug what is happening to a particular request, or work out how to do 'unusual' things such as content negotiation.

On the other hand, express has a very good 'mix-and-match' style, with a wide range of off-the-shelf tools.

Maybe the best approach is to use those tools directly, without using express's obscure routing.
URL rewriting allows you to make logical and intuitive URLs available to users.

Suppose you publish a URL /pets/dogs/odie.html.

When it arrives in a server request, you can change it to /pets/dogs.html?name=odie so you can use a template for all dogs, filled in with data about a particular dog.

Or rewrite to /pets.html?type=dog&name=odie so you can use a single template for all pets.
A **route** is a rule for sending a dynamic URL to the right **handler** function for dealing with the request.

The handler function may use a template and a database to construct a web page to be sent back to the browser.

**OR** the handler may carry out some update to the database, or deal with an upload, and send a confirmation message back to the browser.

Routes are most often based on URL prefixes.
In the prefix case, you recognise any URL which starts with a fixed set of handler prefixes, e.g. `/set/x/42` or `/set?x=42` need to be handled by a function `set`.

The server contains code to recognize the action URLs and call the right handler function, passing it the request, response, etc.

The advantage is simplicity when there are few handlers, and the danger is a poor, non-modular, monolithic design, i.e. all in a one-piece program.
Modularity

A good way to make routing more modular in a DIY server is for dynamic or action URLs to be of the form `/do/set?x=42`

The `/do/` prefix is used to recognize the URL as an action URL.

The next part `set` is used as the name of a module `set.js` in your server module folder (not inside `public`!)

You pass it the parameters and it handles the request, see 📑 table of handlers
Table of handlers

You can maintain a table of handlers

```javascript
let handlers = ...;
let f = handlers[name];
f(request, response);
```

If you are using modules, you can load a module if the handler hasn't been used before (using node's old non-standard module system)

```javascript
let m = require("./do/" + name + ".js");
let f = m.handle;
handlers[name] = f;
```
The method **GET** represents a normal request for a page or file - it has no body

The method **POST** sends data from a form - it has a body with the parameters and their values

The method **PUT** is very useful for simple uploads - you can arrange for the filename to be in the URL, and the body to be the file contents (the alternative, **POST** with type `multipart/form-data`, is complex and inefficient)
To avoid nasty callback issues and allow the `async/await` keywords to be used, a promise version of the `http` library is needed.

At present, there isn't one built into node, so try this:

```
npm install q-io -save
```

It provides `require("q-io/fs")` equivalent to `require("fs").promises`, and `require("q-io/http")` which is like `http` with promises.
async function handle(request, response) {
    console.log("Method:", request.method);
    console.log("URL:", request.url);
    console.log("Headers:", request.headers);
    let body = await request.body.read();
    console.log("Body:", body.toString());
    response.status = 200;
    response.headers = {
        'Content-Type': 'text/plain'
    }
    response.body = ["OK"];
    return response;
}

Like server 1, this responds OK to every request

The body is not instantly available - it has to be read in
To test, create a minimal form page

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Test</title>
</head>
<body>
<form method="post" action="http://localhost:8080/">
  <input name="pet" value="dog"/>
  <input name="car" value="bmw"/>
  <input type="submit" value="Send"/>
</form>
</body>
</html>
```

Double click to view with browser (with a file: URL, not via the server) then press Send
The body of the request posted from a form contains:

```
pet=dog&car=bmw
```

This can be unpacked with the `querystring` module

```javascript
let QS = require("querystring");
...
let params = QS.parse(body);
console.log(params.pet, params.car);
```
In the express world, *middleware* refers to a piece of software which sits between the main *handle* function and the individual handlers which do the real work - they adapt the request in some way, e.g. check passwords and permissions.

A middleware function is given the request and response by *handle*, chooses a handler, and passes it the possibly modified request and response.

Middleware functions are also called *filters*. 
Roughly speaking, your choices are:

- write your own server, using callbacks
- write your own server, using `async/await` (maybe using `flows`)
- use express, with callbacks (as tutorials tell you to do)
- use express, with `async/await` (much better, but harder to find out how)
A template is a web page, reused with different data.

Some convention is used to indicate points in the template where data is inserted.

If you are taking the raw approach to building a server, you could invent your own conventions, but you may want to use some ready-made template system.

Express allows you to choose between template engines (ejs, pug, haml, handlebars, combyne, swig, blade, ...).
Password protection

Passwords must not be sent over the network without encryption.

Using http, passwords are sent using base64 encoding. This is mostly to turn special characters into ascii, and to prevent accidental prying, but it is not encryption.

To encrypt properly, you must switch to https.

- encryption protocols
- saving passwords
- HTTPS sites
- HTTPS forwarding
Encryption protocols

For serious https work, you need to configure your server to avoid encryption algorithms or protocols which are now thought to be *weak*

Look for a web site (e.g. ssllabs) which will carry out an "SSL server test" on the encryption facilities offered by your site.
How are passwords saved in databases?

Usually, they are 'encrypted' first with MD5

Note
(a) MD5 is really a digest (hash) mechanism and not an encryption
(b) it is now regarded as weak and something like SHA256 or bcrypt is recommended instead
(c) it is only secondary security - the main security is not letting anyone access the database in the first place
It used to be common for a site to provide both HTTP and HTTPS (technically two servers, but running in the same program)

It is becoming common to provide only HTTPS

HTTP/2 based on SPDY, and Let's Encrypt, are likely to increase this trend
Suppose your server provides both an http service and an https service for the same site.

And suppose that some areas of the site are protected and some are not.

If an http request is made for a protected area, it is important that the request should be redirected to https.

That means sending a response to the browser, asking it to try again on the https URL.
To use https, you need to provide a private key (from a public/private pair) and a certificate

For this unit, a self-signed certificate is fine (though a browser will ask the user to confirm trust), or if you have a domain you can use Let's Encrypt

These can be loaded from files before the server starts (or even embedded in the code) but must be kept secret
Form submission

Suppose your site has a form page for a privileged user to fill in.

The form data will be sent to an action (specified in the action attribute), normally a dynamic URL.

The form page should be protected, so the user can see a comforting protection icon.

But the action URL really must be protected, because it is the one that carries sensitive data and/or asks the server to do updates.
Input Validation

Data validation can optionally be done on a form page (by HTML5 or script) for quick response.

But validation **must** also be done on the server side, on the action URL that receives the data, for security and data consistency.

For good security, your attitude should not just be "what damaging data could a hacker send" but "what range of data is safe".

It is better to start very restrictive and expand than the other way round.
Output Validation

Suppose a user can enter text which ends up displayed on a web page (e.g. in a blog post or even in an error message)

Care needs to be taken that it can't do any harm by having any special HTML/CSS/JS meaning

The main danger is **XSS** (cross-site scripting) attacks

The issues are subtle, so it is best to use a security-aware library feature
A session is needed to track multiple requests coming from the same user, and keep some state data.

This is particularly necessary for protected pages, just to remember that the user has already logged in and needn't login again.

It is possible to use OAuth to delegate logins.

But otherwise, how do sessions work?
Shared secrets

It is not enough to use IP addresses and port numbers, partly because they can be forged, and partly because they are not always specific to users and their browsers.

It is normally done by having a shared secret, i.e. an unguessable random number held by both the browser and server.

Since nobody must see it, HTTPS must be used, even if password login isn't used.
Generating a shared secret

It is the server which must generate the shared secret on a user's first visit.

A username or fixed id number or pseudo-random number is *not* good enough.

A time-of-day random number is very often used, which is OK but not excellent.

An id signed with an encryption key is good, and so is a crypto-random number (usually entropy-based).
A cookie is a name/value pair which the server sets in a response header

After that, the browser will send the same cookie in a request header every time it visits the same server

This is ideal for a shared secret and is then called a session cookie and it lasts as long as you choose

There is no need to store any data in the browser other than a session id – everything else can be in the server

(The main exception is using browser localStorage when there is no configurable server)