# Client Server Explained

What happens in detail in client server situations

- Web server
- AJAX

pdf

# Simplest Web Server Situation

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser | | Web Server |
| test.html | | Folder “www” with files |

# Simplest Web Server Situation 2

You type a URL in the address bar of the browser, or click on a hyperlink in the HTML you are reading. The URL is
http ://server.com/test2.html

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser | –> | Web Server |
| test.html | | Folder “www” with files |

# Simplest Web Server Situation 3

The server receives the URL, tests if it corresponds to a file, reads the file, send the data

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser<br>test.html | <– | Web Server<br>Folder “www” with files |

# Simplest Web Server Situation 4

The browser receives the new data and shows it

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser | | Web Server |
| test2.html | | Folder “www” with files |

# Focus on the server only

The simplest web server receives a request and send a response. So it is a function like :

```
function webserver( request, response ) {
    ...
}
```

To be able to write the code, we need :

- the request object to contain all info about the received request : URL, headers, origin, etc
- the response object to have all the methods to configure then send the response to the client : setHeader, send, etc

There is extra stuff you do not need to know in detail (TCP, HTTP text exchanged, encryption, compression, etc)

# Simplest Web Server 2

Logic of a web server serving only files :

- does the URL correspond to an existing file ?
  - if yes, read the file and send content
  - if no, send error message

```
function webserver( request, response ) {
 const filepath = request.url.substring(1); // remove initial /
  if (fs.existsSync(filepath)) {
   response.setHeader("Content-Type", mime.getType(filepath));
    response.end(fs.readFileSync(filepath));
  } else {
    response.statusCode = 404;
   response.end("the file " + filepath + " does not exist on the s
  }
}
```

# Simplest Web Server 3

Logic of a web server serving only generated data :

- does the URL belong to the list of URLs I know ?
  - if yes, generate corresponding content and send
  - if no, send error message

Logic of a web server serving both :

- is is a URL for a file ?
  - if yes, does the URL correspond to an existing file ?
    - if yes, read the file and send content
    - if no, send error message
  - if no, does the URL belong to the list of URLs I know ?
    - if yes, generate corresponding content and send
    - if no, send error message

So simple web server code is a list of ifs with conditions on the URL.

# Simplest Web Server 4

```
function webserver( request, response ) {
   if (request.url === '/addmodule') {
       ... code for add module ...
   } else if (request.url === '/changegrade') {
       ... code to change grade ...
   } else ...
}
```

# Simplest AJAX Situation

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser | | Web Server |
| test.html | | Folder “www” with file “test.txt” |
| Ajax code | | |

# Simplest AJAX Situation 2

You press on a button that triggers the Ajax code

| Your computer | < Network > | Another compute |
|---|---|---|
| Browser | | Web Server |
| test.html | | Folder “www” witl |
| `var xhr = new XMLHttpRequest();` | | |
| `xhr.open("GET", "test.txt");` | | |
| `xhr.onload = function() {...}` | | |
| `xhr.send();` | –> | |

# Simplest AJAX Situation 3

Server gets request for “test.txt”, sends file content

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser | | Web Server |
| test.html | <– | Folder “www” with file “test.txt” |
| Ajax code | | |

# Simplest AJAX Situation 4

The code in the `xhr.onload` function is called with `this.responseText` equal to the content of the test.txt file.

| Your computer | < Network > | Another computer |
|---|---|---|
| Browser<br>test.html<br>`alert(this.responseText);` | | Web Server<br>Folder “www” with file “te |

# Timing

There are five delays :

- internal delay in your machine between the button and the Ajax code (microseconds)
- network delay for the browser to send request to server (10s of milliseconds)
- server delay computing the response, maybe delay reading file from disk (microseconds to milliseconds)
- network delay for the server to send the test.txt data to the browser
- internal delay for the browser to execute the final code

Because of the delays, this is called asynchronous programming. Synchronous programming is when one program executes from beginning to end. Asynchronous programming has many bits of code which get activated when they called or when their data is ready.

# More complex situations

- more complex URL to get
- request with other method (PUT, POST, DELETE)
  - for PUT and POST, there is more code in the Ajax part to send more stuff to the server
- more complex code in the onload function, e.g. create a whole piece of HTML from the received data
- the web server serves a generated file, not an existing file
  - the web server accesses a database and generates HTML from the database response
  - the web server can be PHP, python (django), Java (TomCat), JS (node.js)

Choice : the code can be placed in the onload function on the client, or on the web server.

# Programming the server

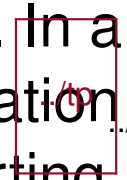
When programming a web application server, there are many steps.

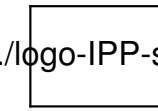
1. The user loads a page from the server
2. The user interacts with the page
3. The page generates a request to the server to do something
4. The server computes the request to the server

It is important to distinguish 1. and 3. even though these are two requests to the server. The first request is for an interface. The second request carries the action.

The important part of the first request is to create the interface capable of sending actions. This means HTML and CSS for the semantics and visual part of the interface, plus client-side javascript for the potential to send actions.

The important part of the second request is the action itself. In a travel reservation system, the action is to create the reservation and buy the ticket. There is also an interface part: the reporting





# Skeleton of a server

I assume that the server is programmed in JavaScript with Express.

1. Create the express application(s) : this is the object which carries all the functional power
2. Create the HTTP(S) server(s) from the application
   2.1 Possibly, create credentials (if HTTPS)
3. Configure the server and position the middleware
4. Configure all the routes, i.e. URL-processing function pair
   4.1 Routes for interface creation
   4.2 Routes for action implementation
5. Start the server(s)

An HTTP server listens to possibly many HTTP requests. To process a request, the server prepares a request object, with all the received information, and a response object that can be used to create the response. For each URL there will be one main function computing a response.

# Application, server and middleware

The application object is the one which implements all the configuration, server creation, server starting actions.

The server object(s) is(are) the one(s) which implements HTTP, so you tell them the options you want : security, encryption, ports, subprotocols like WebSockets. . .

The middleware needs a definition. There are actions to do on one/some/all requests, before or after the main processing.

- creating a log of all requests
- authentification of the user requesting an action
- detection of requests which should be modified, e.g. translate old URLs to a new format
- detection of requests which should be redirected elsewhere, i.e. sending the URL to another server
- detection of hacking, e.g. blocking connection from “bad” IPs
- detection of complex requests, e.g. processing of file

# Middleware

A middleware is a function which receives a request and a response object as well as a `next` function. A middleware does its processing with request and response, then calls `next()` to allow the other middleware to be executed, then the main processing to happen, then possibly does more of its processing after.

Express has a list of middlewares to call, the list you gave it upon configuration.

A logging middleware gets the URL from the request, writes it in a log and passes the buck.

A hacking detection middleware checks the origin IP, passes the buck if the IP is OK, and stops everything if the IP is not OK.

A performance checking middleware would check the time, pass the buck, and when all the processing is finished, check the time again and log the execution time somewhere.

# Routes

A route is a pairing between a URL and the processing function for that URL. Sometimes the URL is constant, sometimes the URL has parameters.

From the previous slides, what Express does upon receiving a request is :

1. prepare the request and response object
2. execute the middlewares
3. if not interrupted, look for the matching route
4. execute the processing function of the matching route

If the matching route creates an interface, then the job is to prepare the required objects then render the pug template.

If the matching route is one of action, then the job is to implement the action and possibly report the results.

# Interface

Pug is a template language to easily create HTML pages. It is a mixture of simplified HTML and JavaScript code. Pug is like python : indentation determine the inclusion in the above tag. Pug templates are simpler if you adopt the following structure :

- in your JS code, compute all the objects that will be necessary for the template
- call `render` on your `pug` view, passing all the objects as input

Separating the object computation makes for cleaner code and simpler templates.

# Pug

What you can do in pug :

- insert a variable in any HTML content
- on a condition, add an HTML subtree (attribute, text content, tags. . .)
- loop on an array to add an HTML subtree (to create a table)
- and as usual, add styling and scripts
  - it is possible to simply import existing style and scripts
  - it is also possible to create CSS with a style tag inside pug, using template functionality
  - it is also possible to create JavaScript with a script tag inside pug, using template functionality

# Summary

So you have many possible places where to put your JS code :

1. express configuration : before the start of the server
2. express middleware : when express is checking the URL of the request
3. express processing to create an interface, e.g. with a pug template
4. client-side javascript which creates a part of the interface
5. client-side javascript responding to user interaction, e.g. to send an action
6. express middleware checking the action URL
7. express processing to implement the action, e.g. change the database
8. express processing to create action reporting, which is like 3.

Step 5. is typically Ajax code. Step 4. is optional. Steps 2. and 6. are similar but apply to different data

# Difficulty

It is difficult to pass information from the server/express context to the client-side javascript context. This is information you have access to in step 3. and you need it in step 5. One way to do it is :

- you have one object `obj` to transmit
- pass `obj` to pug
- in pug, add a script tag with content `let info = JSON.parse("#{JSON.stringify(obj)}");`

Then you will be able to use the variable `info` in the client-side JavaScript.