CS2003 W09
AJAX & Server Side

Özgür Akgün

Slides adapted from Al Dearle and Saleem Bhatti

AJAX

https://www.w3schools.com/js/js_ajax_intro.asp

Ajax

- Asynchronous JavaScript and XML
- Technique allowing clients to display active content requiring communication with server
  - without user having to reload page
  - not necessarily asynchronous
  - not necessarily XML
- Also referred to as Scripted HTTP

Ajax

- JavaScript code running in browser makes explicit calls to server
  - e.g. check whether selected item is available
  - e.g. download neighbouring map tiles
- In response to some user action
  - e.g. type a character
  - e.g. mouse drag gesture

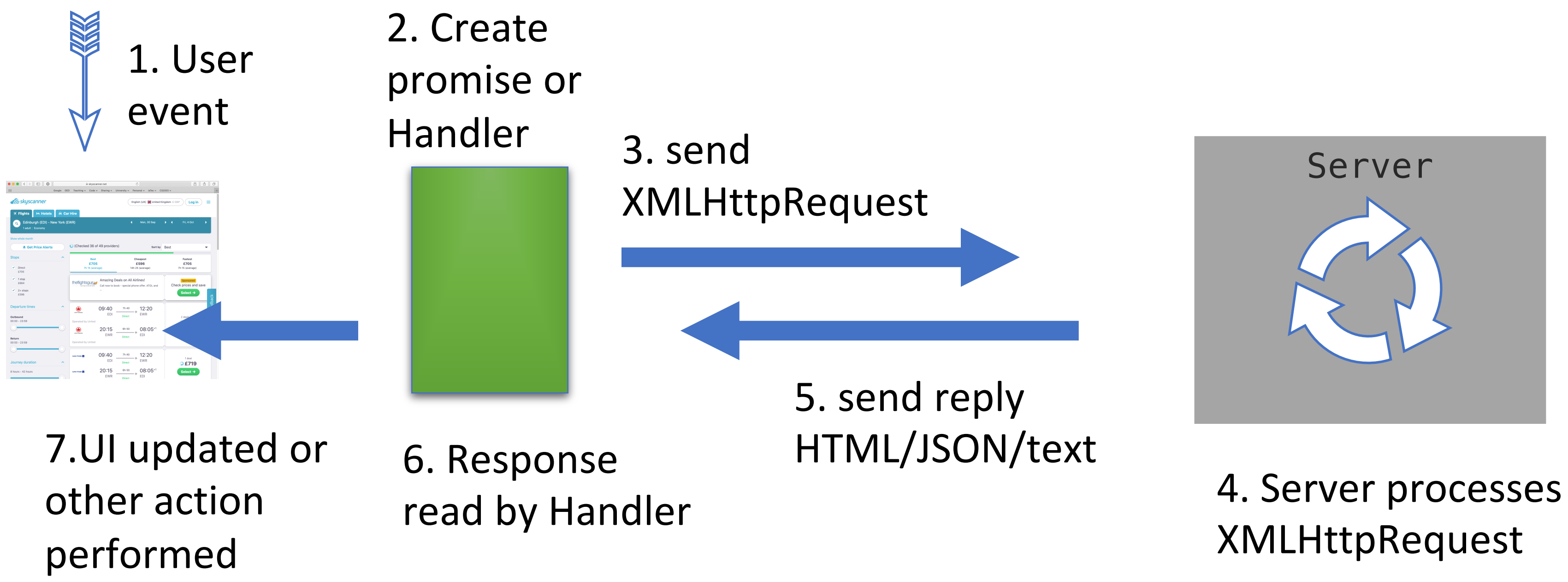
Ajax

- When the result of the call is received from the server, JavaScript code updates contents of page
- This may happen after a significant delay
  - Usually it is better to use **asynchronous** code so that we do not have to synchronously wait for response from the server

Suitability

- Dynamic content
  - page must change without being reloaded
- Client-side computation dependent on data that is:
  - too large to download to client in one go,
  - or subject to frequent change (dynamic)

Ajax Control flow


1. User event
2. Create promise or Handler
3. send XMLHttpRequest
Server
4. Server processes XMLHttpRequest
5. send reply HTML/JSON/text
6. Response read by Handler
7.UI updated or other action performed

The XMLHttpRequestObject

- The corner stone of AJAX is the **XMLHttpRequest** which is used to exchange data between the browser and the server
- This permits parts of a web page to be updated without reloading the whole page
- Described here: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

Example:

- root page for demo: https://www.w3schools.com/js/js_ajax_http.asp
- demo: https://www.w3schools.com/js/tryit.asp?filename=tryjs_ajax_first
- The text file used in this example is here:
    - https://www.w3schools.com/js/ajax_info.txt

XMLHTTPRequest.html

```
<!DOCTYPE html>
<html>
<body>

<!-- from: https://www.w3schools.com/js/tryit.asp?filename=tryjs_ajax_xmlhttp -->

<h2>The XMLHttpRequest Object</h2>

<p id="demo">Let AJAX change this text.</p>

<button type="button" onclick="loadDoc()">Change Content</button>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };
  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
</script>

</body>
</html>
```

AJAX and Promises

- There is a new API for AJAX which uses Promises
- It is easier to use than XMLHTTPRequest - more convenient
- It is also more flexible:
  - Can use the Cache API
  - Can deal with servers that don't use CORS (later)
    - CORS: Cross-Origin Resource Sharing
  - Can deal with streams better (doesn't store the responses in memory)

Fetch API

- Standard: https://fetch.spec.whatwg.org/
- Example: https://developers.google.com/web/updates/2015/03/introduction-to-fetch

XMLHttpRequest vs fetch

```
function reqListener() {
  var data = JSON.parse(this.responseText);
  console.log(data);
}

function reqError(err) {
  console.log('Fetch Error :-S', err);
}

var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError;
oReq.open('get', './api/some.json', true);
oReq.send();
```

```
fetch('./api/some.json')
  .then(function(response) { ... })
  .catch(function(err) {
    console.log('Fetch Error :-S', err);
  });
```

Response objects

- many properties, most common/useful:
  - headers - the response headers (often used to check content type)
  - status - the status code
  - statusText - the HTTP status code message
  - ok - a boolean - is the status 200-299
- The response body is an instance of ArrayBuffer, ArrayBufferView, Blob/File, String, FormData or URLSearchParams
- Extracted using arrayBuffer(), blob(), json(), text() and formData()

Loads of details

- May be found here:
  - https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- Importantly:
  - You can supply request options to fetch:
    - fetch(url,options)

Aside - CORS: Cross-Origin Resource Sharing

- For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts
- XMLHttpRequest and the fetch follow the same-origin policy
- Therefore a web application can only request resources from the same origin that the application was loaded from
  - unless the response from other origins includes the appropriate headers.

CORS

- CORS adds HTTP headers which provide browsers with a way to request remote URLs only if they have appropriate permissions
- If a page loaded from one web server wants to access another it must request permission from the server by performing request with an Origin HTTP Header
- This is all you need to know for now
- The School Web servers do not support this feature
- More at:
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
  - https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

Key Points

- Ajax allows client-side code to interact with server
  - request sent as HTTP GET or POST
  - result string extracted from server response
- Data returned from server in various formats
  - plain strings
  - JSON
  - XML

Node.js
Express

NodeJS

- You have seen this a lot already, not really a framework
- Server side
- Scalable solution for writing servers, particularly micro-services
- Lets you share the same language between client and server
- It’s quite fast/scaleable compared to some other server side solutions

NodeJS and Express

- Much as the browser has frameworks, so does Node.js.
- Express.js is a perhaps the most popular one that makes it easier to make RESTful APIs.
- ‘Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications’
- Three major benefits:
  - Defines a **routing table** which is used to perform different actions based on HTTP Method and URL.
  - Permits the dynamic rendering of HTML Pages based on passing arguments to **templates**.
  - Makes the server code simpler by imposing structure

Some useful links

- https://expressjs.com
- https://expressjs.com/en/starter/hello-world.html
- https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm - good tutorial
- https://webapplog.com/express-js-fundamentals/ - an ‘essential overview
- https://expressjs.com/en/guide/using-template-engines.html
- https://www.npmjs.com/package/pug
- https://pugjs.org/api/reference.html

Express - example

- In this example, we have defined a **route** that returns ‘Hello World!’ when we send an HTTP request to it.

```
const express = require('express')
const app = express()
const port = 3000
// app.get – application routing for GET!
app.get('/', (req, res) =>     // define route
                res.send('Hello World!’))
app.listen(port, () =>
     console.log(`Listening on port ${port}!`))
```

from: https://expressjs.com/en/starter/hello-world.html

Another simple example

- Serve all the static content from a directory called public:

```
var path = require('path');
var express = require('express');

var app = express();

var staticPath = path.join( dirname, '/public');
app.use(express.static(staticPath));

app.listen(3000, function() {
  console.log('listening');
});
```

Routes

- Routes are part of the ‘extra sauce’ that makes Express a winner, for example…

```
var express = require('express');
var routes = require(‘./routes’);
var user = require(‘./routes/user');

/* more stuff missed out */
app.set(‘port’, SOMEPORT );
var app = express();
app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'),
 function(){
     console.log('listening on port ' +
                 app.get('port'));
});
```

from https://webapplog.com/express-js-fundamentals/

Routes

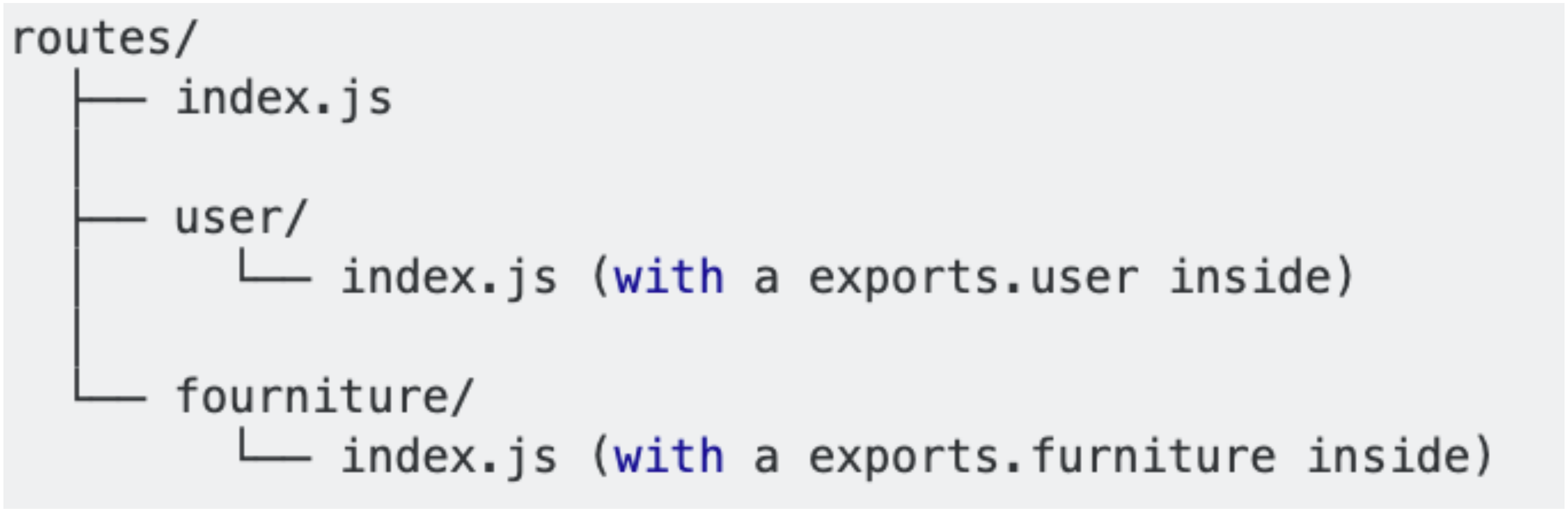
```
app.get('/', routes.index);
app.get('/users', user.list);
```

- The first of these tells all GET requests for / to be dealt with by the code in index.js
- The second says that requests for http://...../users will be handled by the code found in the users directory
- In each folder Express expects an index.js function:

```
exports.xxxx =  function(req, res){
    res.render(…); } // render coming up
```

Directory structure

- Hopefully you can see that with this framework you can start to ‘tidy up’ the complexity of your server code for a real industrial scale web server

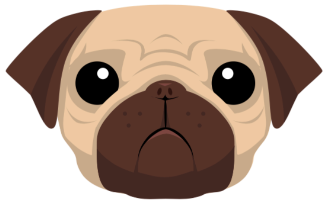


```
routes/
  ├── index.js
  │
  ├── user/
  │     └── index.js (with a exports.user inside)
  │
  └── fourniture/
        └── index.js (with a exports.furniture inside)
```

https://stackoverflow.com/questions/16548586/adding-a-new-route-to-node-express

Templates

- Templates are static template files
- At runtime, a template engine replaces variables in a template file with actual values, and transforms the template into HTML sent to the client.
- Many different template engines exists and the Express Middleware lets these be plugged into the server infrastructure
- By default Express used an engine that you might see being called **Jade** but has been renamed to **Pug** because Jade was a registered trademark
- We will only look at this one



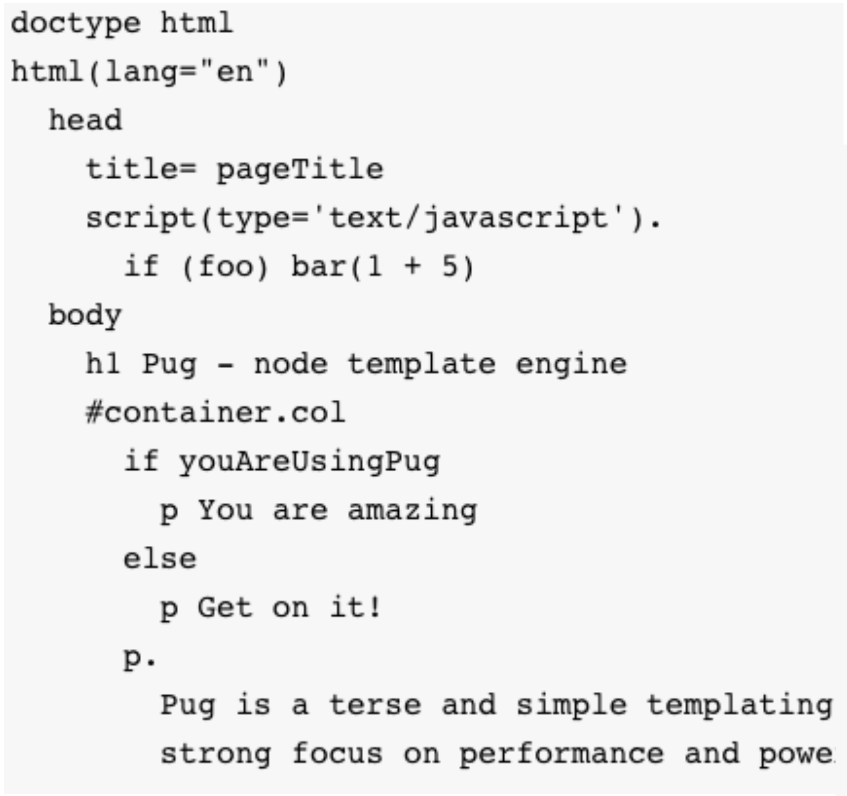
A simple pug template

```
html
  head
    title= title
  body
    h1= message
```

A file called index.pug for example

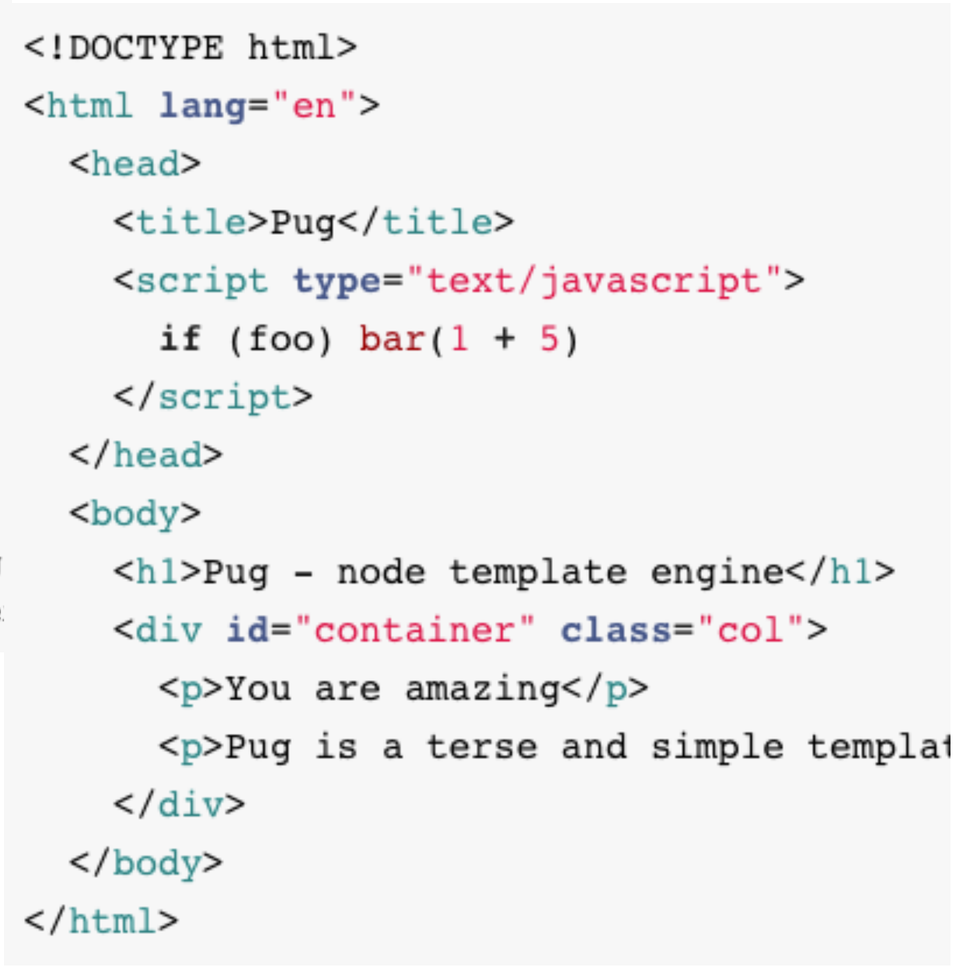
```
app.set('view engine', 'pug')
app.get('/', function (req, res) {
  res.render('index',
             { title: 'Hey',
               message: 'Hello there!' } )
} )
```

A full Pug Template



```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) bar(1 + 5)
  body
    h1 Pug - node template engine
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
      p.
        Pug is a terse and simple templating
        strong focus on performance and powe
```

Yields:



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Pug</title>
    <script type="text/javascript">
      if (foo) bar(1 + 5)
    </script>
  </head>
  <body>
    <h1>Pug - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>Pug is a terse and simple templat
    </div>
  </body>
</html>
```

from:
https://www.npmjs.com/package/pug
reman:
https://pugjs.org/api/reference.html

Good and bad things about Pug

- Good:
  - Using templating allows you to write boilerplate pages that may be specialised (shorter code/no code)
  - Good readability - you write paragraphs
  - Use indentation instead of tabs for structure
- Bad:
  - The white space indentation is error prone
  - You cannot use HTML - only pug
- Finally, a full example:
  - https://github.com/bmorelli25/simple-nodejs-weather-app/tree/pug