Internetværk og Web-programmering

Server-side programming

Forelæsning 7

Michele Albano

Distributed, Embedded, Intelligent Systems

Agenda

- More on fetch()
- Client-side storage
- Non-client-side programming: Node.js
- Node.js as HTTP server

Fetch Streaming API

- Do not get a Promise out of the response
  - If you access the data in any way (.json(), .text(), etc), you cannot re-access it
- The response has method getReader() to get a stream reader object:

```
let reader = response.body.getReader();
while(true) { // Loop until we exit below
    let {done, value} = await reader.read();
    // Verify value is not null. Process the “value”. Parse the data. Check for errors
if (done) { // If this is the last chunk,
        break; // exit the loop
    }
}
```

Streaming Response bodies

- The body property of a Response object is a ReadableStream object.
  - Do not call text() or json() if you want to show a progress bar

```
fetch('big.json')
  .then(response => streamBody(response, updateProgress))
  .then(bodyText => JSON.parse(bodyText))
  .then(handleBigJSONObject);


async function streamBody(response, reportProgress, processChunk) {
  let expectedBytes = parseInt(response.headers.get("Content-Length"));
  let bytesRead = 0;
  let reader = response.body.getReader(); // Read bytes
  let decoder = new TextDecoder("utf-8"); // Bytes to text
  let body = ""; // Text read so far
  while(true) { // Loop until we exit below
    let {done, value} = await reader.read(); // Read a chunk
    if (value) { // If we got a byte array:
      if (processChunk) { // Process the bytes
        let processed = processChunk(value);
        if (processed) {body += processed;}
      } else { // Otherwise, convert bytes
        body += decoder.decode(value, {stream: true});}
      if (reportProgress) { // If a progress callback was
        bytesRead += value.length; // passed, then call it
        reportProgress(bytesRead, bytesRead / expectedBytes);}
    }
    if (done) { // If this is the last chunk,
      break; // exit the loop
    }}
  return body; // Return the body text we accumulated
}
```

POST for fetch()

- Interactions: GET, PUT, DELETE, POST
- To perform a POST, you specify the method, and what you send in a body, for example a string:

```
fetch(url, {
   method: "POST",
body: ”hello world”
}) .then ……
```

- It can be JSON data:

```
fetch(url, {
   method: "POST",
   headers: new Headers({"Content-Type": "application/json"}),
   body: JSON.stringify(requestBody)
}) .then ……
```

FormData for `fetch()`

- Instead of using the “submit” of a HTML form, it is possible to submit a form programmatically
- `formData()` returns an object that can be sent using `fetch()`
- By using this, you lock your message on a *multipart/form-data* encoding
- Very useful for a POST

```
let formData = new FormData().append('username', 'abc123’)
.append('avatar', 'thisismyavatar');

fetch('https://example.com/profile/avatar', {
  method: 'POST',
  body: formData
})
.then(response => response.json())
.catch(error => console.error('Error:', error))
.then(response => console.log('Success:', JSON.stringify(response)))
```

Agenda

- More on fetch()
- Client-side storage
- Non-client-side programming: Node.js
- Node.js as HTTP server

On-browser storage

- HTTP is a stateless protocol.
  - When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.

Three options to store something client-side (on the browser):

- The **Web Storage API** consists of the localStorage and sessionStorage objects
  - Persistent objects that map string keys to string values
  - Can store large (but not huge) amounts of data
  - Difference: sessionStorage is deleted when the browser window/tab is closed
- **IndexedDB** is an asynchronous API to an object database that supports indexing
- **Cookies** were designed to remember (little) information about the user
  - Cookies are saved in name-value pairs like:
    - username = John Doe
  - Data in the cookies is transmitted with every HTTP request, even if the data is only of interest to the client

Cookie manipulation with Javascript

- JavaScript can create, read, and delete cookies with the `document.cookie` property
- Creation:

```
document.cookie = "username=John Doe";
```

- Creation with expiry date (by default, the cookie is deleted when the browser is closed):
  - `document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";`
- Access to cookies:

```
let x = document.cookie;
```

- document.cookie will return all cookies in one string much like: `cookie1=value; cookie2=value; cookie3=value;`

Checking a cookie

- Let us verify if a cookie is set.
  - If so, it will display a greeting.
  - If not, it will display a prompt box, asking for the name of the user, and stores the username cookie for 365 days, by calling the `setCookie` function:
- JavaScript code:

```
function checkCookie() {
  let username = getCookie("username");
  if (username != "") {
    alert("Welcome again " + username);
  } else {
    username = prompt("Please enter your name:", "");
    if (username != "" && username != null) {
      setCookie("username", username, 365);
    }
  }
}
```

Agenda

- More on fetch()
- Client-side storage
- Non-client-side programming: Node.js
- Node.js as HTTP server

Node.js

- Node.js is a program that allows you to apply your JavaScript skills outside of the browser.

- It provides the program `node` to execute javascript files:
  - `node hello.js`

- Or it can be used in REPL (read-eval-print-loop) mode:
  - `node`
- Then, an interpreter gets available to evaluate javascript interactively
- For example, it is possible to type
  - `global.` [TAB][TAB]
- And it will provide everything that can be typed after "`global.`"

Event loop

- Javascript code runs on a single thread, which performs “event loops”:
- Pay attention to how you write your code and avoid anything that could block the thread, like synchronous network calls or infinite loops
- Every time the event loop takes a full trip, we call it a tick
- Example: a function can be passed to `process.nextTick()`. It will be executed on the current iteration of the event loop, after the current operation ends

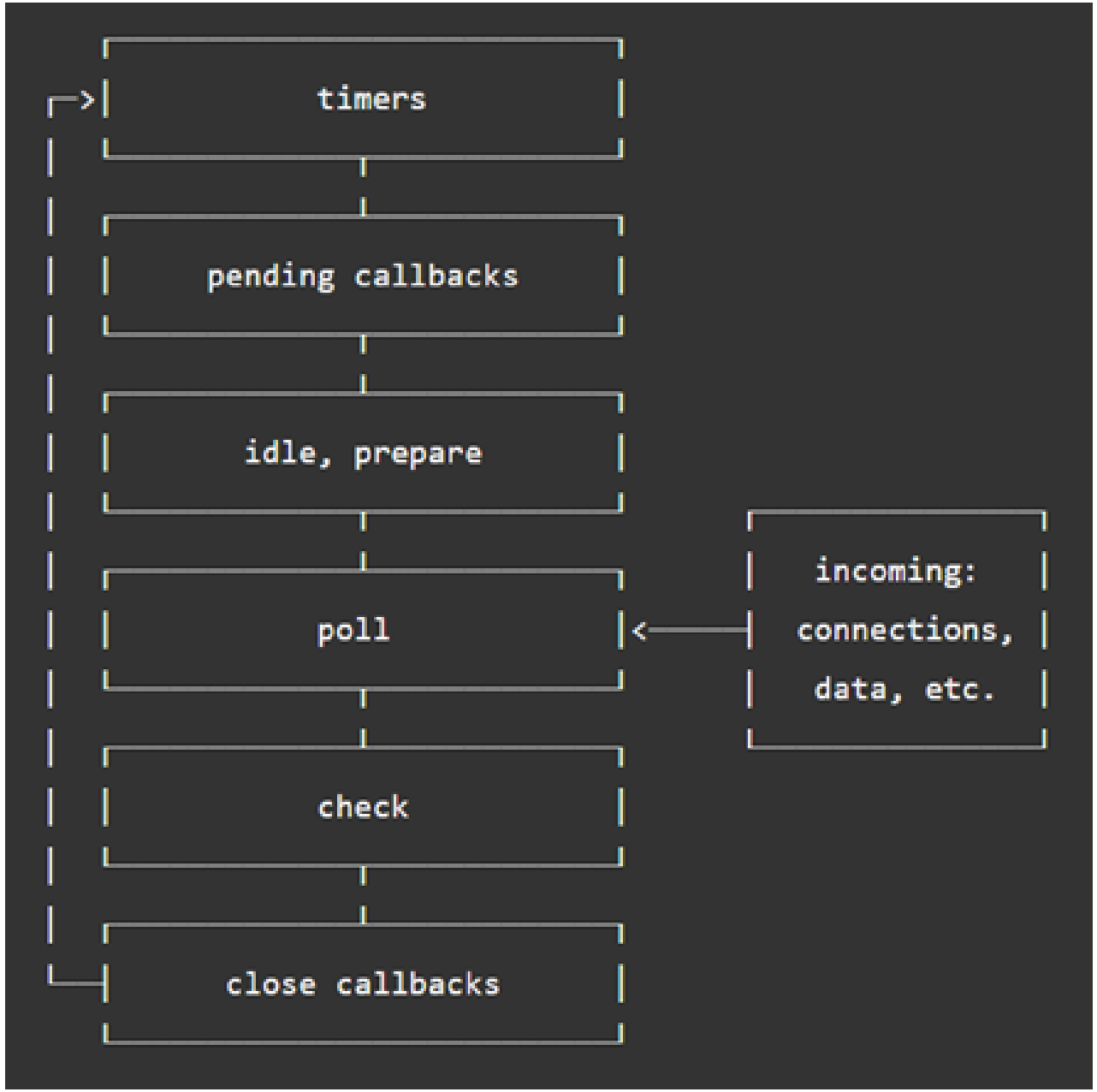
```
process.nextTick(() => {
  //do something
})
```

- Other options (both for next tick, or another tick in the future):
  - `setImmediate`: in a late phase of this tick
  - `setTimeout`: beginning of the proper tick

Event loop of Node.js

JS: all code still runs on a single event loop.

- timers: this phase executes callbacks scheduled by setTimeout() and setInterval().
- pending callbacks: executes I/O callbacks deferred to the next loop iteration.
- idle, prepare: only used internally.
- poll: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and setImmediate()); node will block here when appropriate.
- check: setImmediate() callbacks are invoked here.
- close callbacks: some close callbacks, e.g. socket.on('close', ...).



Node.js: difference with browser's JS

- First of all, there are no *document*, *window* and all the other objects that are provided by the browser.
- There are a number of bindings that are provided by default, such as `process` (command line arguments) and `console`

```
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`)
})
process.exit(0)
```

- Environment variables:

```
console.log(process.env.PATH)
```

Program life cycle

- Node programs do not exit until they are done running the initial file and until all callbacks have been called and there are no more pending events
- A Node-based server program that listens for incoming network connections will theoretically run forever because it will always be waiting for more events
- Forcing program to quit: `process.exit(0)`
- CTRL+C?
  - `process.on("SIGINT", ()=>{})`
- Exceptions in callbacks or event handlers must be handled locally or not handled at all
- Not to crash the program, register a global handler function:

```
process.setUncaughtExceptionCaptureCallback(e => {
    console.error("Uncaught exception:", e);
});
```

Packages and modules

- Code is shipped as packages and modules
- A package is a file or directory that is described by a package.json file, it can be installed with package managers (e.g.: npm), and it comprises one or more modules (libraries) grouped (or packaged) together.
- Old style: CommonJS modules
- A module is any file or directory that can be loaded by Node.js' `require()`
  - It can be from a specified path:
  - `const config = require('/path/to/file');`
  - It can be looked up (`module.paths`):
  - `Let coolModule = require(‘find-me');`
- The module’s variables and functions are made available using `modules.export(…)`
- Package.json: type is “commonjs”

New approach to packages and modules

- New style: Standard ES6 modules do not use `require`
- `export` to make code available
- `import` to access it
- Explicit file extension: `.mjs` ← for readability
- Package.json: type is “module” ← required

```
{  "main": "eventLoop.js",  "type": "module"}
```

- ES6 modules can load CommonJS modules using the `import` keyword

Npm package manager

- Example of a `require`:

```
const library = require('./library')
```

- Example of a package:

Shapes <- Package name
- Circle.js <-
- Rectangle.js <- Modules that belong to the Shapes package
- Square.js <-

- You install the package (`npm install Shapes`), `require` it, and have access to the Circle, Rectangle, and Square modules.

Promisifying: make promises easily

```
const promisifiedReadFile =
    util.promisify(fs.readFile);
promisifiedReadFile('promisify1.js', 'utf8')
    .then((data) => {console.log("received ", data)})
    .catch((err) => {
        console.log('Error', err);
    }
);
```

Streams

- Buffer class: a lot like a string
  - It is a sequence of bytes instead of a sequence of characters
  - Very common when reading data from files or from the network
  - Very common when manipulating binary data

```
let b = Buffer.from([0x41, 0x42, 0x43]); // <Buffer 41 42 43>
b.toString() // => "ABC"; default "utf8"
b.toString("hex") // => "414243“
let computer = Buffer.from("IBM3111", "ascii"); // Convert string to Buffer
for(let i = 0; i < computer.length; i++) { // Use Buffer as byte array
    computer[i]--; // Buffers are mutable
}
computer.toString("ascii") // => "HAL2000"
```

Interaction with the file system

- Module “fs” provides methods to read and write files. Asynchronous mode:

```
import { readFile, writeFile } from 'node:fs';

readFile('fsExample.js', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

writeFile('message.txt', "hej world", 'utf8', (err) => {
if (err) throw err;
console.log('The file has been saved!');
});
```

Synchronous read

- The “fs” module allows also for synchronous interactions

```
import { readFileSync } from 'node:fs';
const data = readFileSync('./message.txt',
            {encoding:'utf8', flag:'r'});

// Display the file data
console.log(data);
```

- Not very JS-like: it could stop the JS thread for a long time!

Promises and async/await file system interaction

```
// Promise-based asynchronous read
fs.promises
    .readFile("data.csv", "utf8")
    .then(processFileText)
    .catch(handleReadError);

// Or use the Promise API with await inside an async function
async function processText(filename, encoding="utf8") {
    let text = await fs.promises.readFile(filename, encoding);
    // ... process the text here...
}
```

Agenda

- More on fetch()
- Client-side storage
- Non-client-side programming: Node.js
- Node.js as HTTP server

Node.js as a webserver

- One of the most common usage of node.js is to implement a webserver
- First step:

```
import http from 'http';
```

- or:

```
import https from 'https’;
```

HTTP client

- `http.get() / https.get()`
  - It is a native API, there is no need to install third party modules
  - Quite low level
  - The response is a stream (“buffer” of bytes)

- However, please install the `node-fetch` module
  - support for Promises
  - same API as window.fetch
  - few dependencies

The HTTP server

- Create a new Server object
- Call its listen() method to listen for requests on a specified port
- Register an event handler for “request” events
  - Read the client’s request (particularly the request.url property)
  - Write a response

A touch of reality

- Node’s built-in modules are all you need to write simple HTTP and HTTPS servers
- Production servers tend to use external libraries—such as the Express framework—that provide “middleware” and other higher-level utilities for backend web developers

A simple webserver

- Let’s now take a look at the three files in the BMI webserver