

Backend Development– W1S1

Node.js

Cyrille Jegourel – Singapore University of Technology and Design



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Outline (Day 1, Session 1)

- Installation of Node.js
- Introduction to Node.js
- Recall on Asynchronous execution
- NPM
- Http module and web server

Why Node.js and what is it?

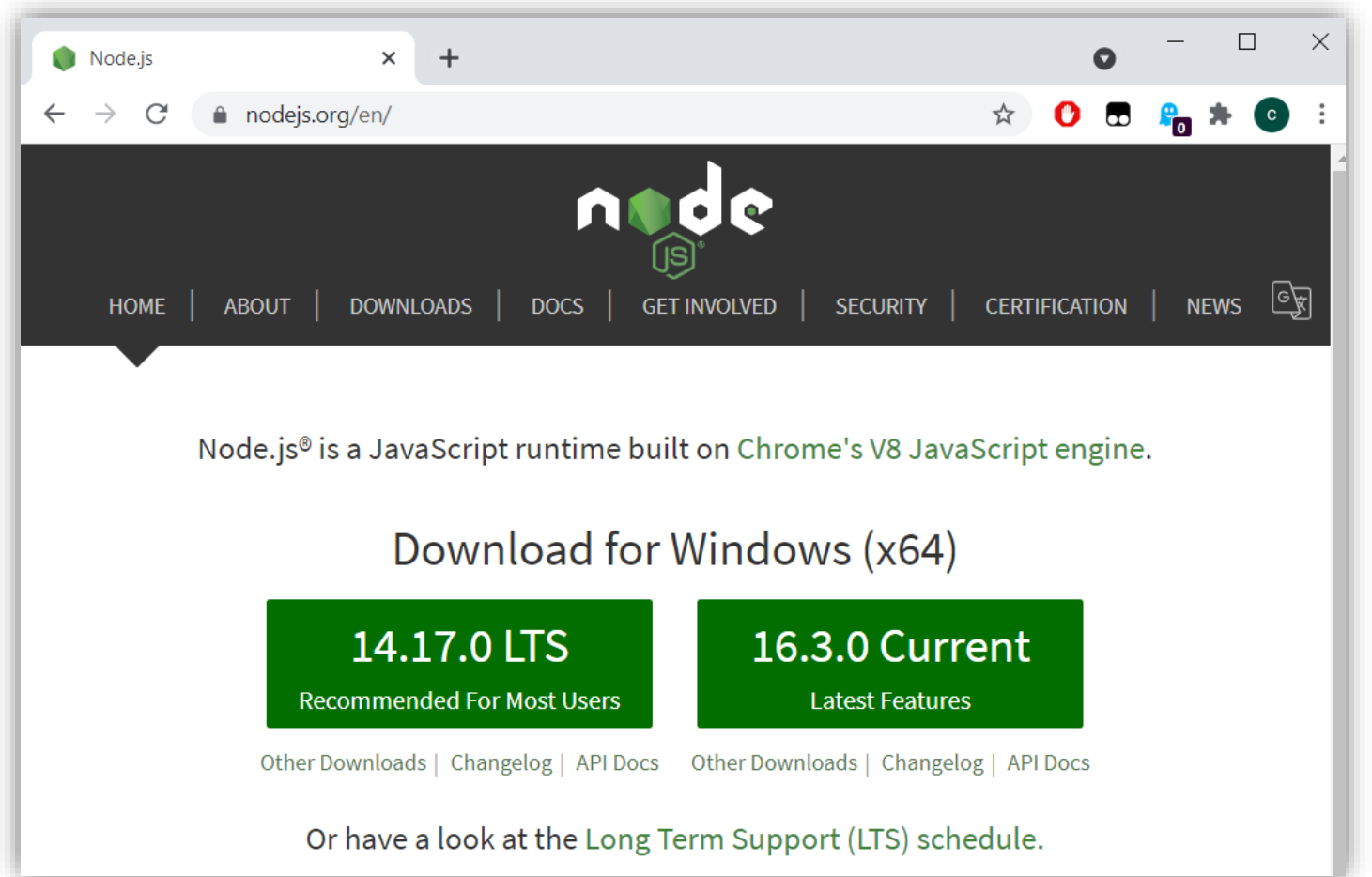
- Open-source, cross-platform, back-end runtime environment that executes JavaScript code outside a web browser.
- Allows to write running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.
- Represents a "JavaScript everywhere" paradigm for web-application development.
- Event-driven architecture capable of asynchronous I/O.
- Allows the creation of Web servers and relies on a collection of "modules" that handle various core functionalities.
- More at <https://en.wikipedia.org/wiki/Node.js>

(Main) references

- “Javascript: A beginner’s guide”, John Pollock, 5th ed, 2020.
 - There is an introduction chapter on Node.js.
- <https://nodejs.org/en/docs/guides/>
 - Very well-written documentation
- https://www.youtube.com/watch?v=Oe421EPjeBE&ab_channel=freeCodeCamp.org
 - A 8-hour video on Node.js and Express.js by John Smilga (see his github here: <https://github.com/john-smilga/node-express-course>).

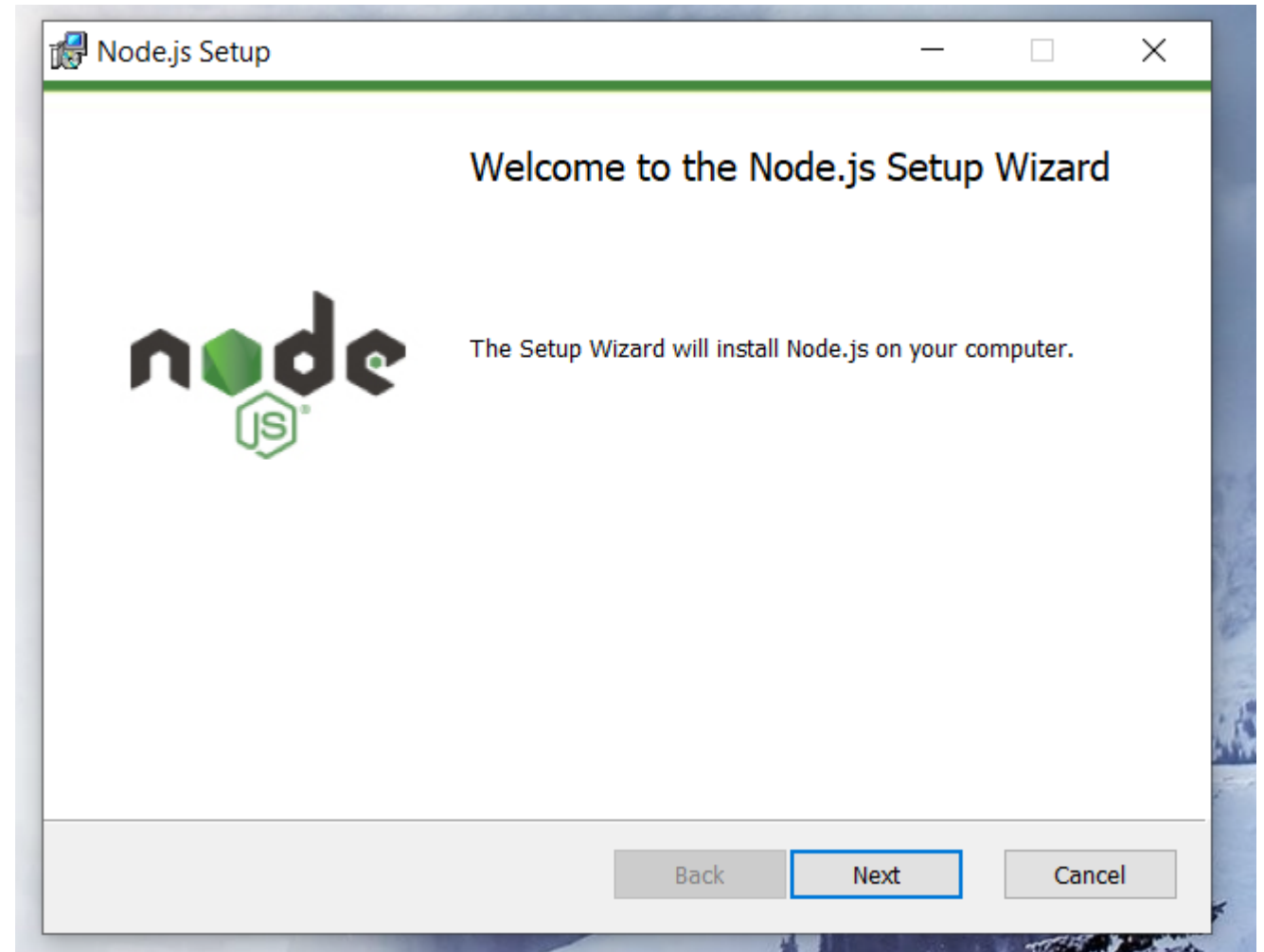
Node.js installation

- Go to nodejs.org and download the Long Term Support version (14.17.0 for me).
- Don't worry if your version is more recent.



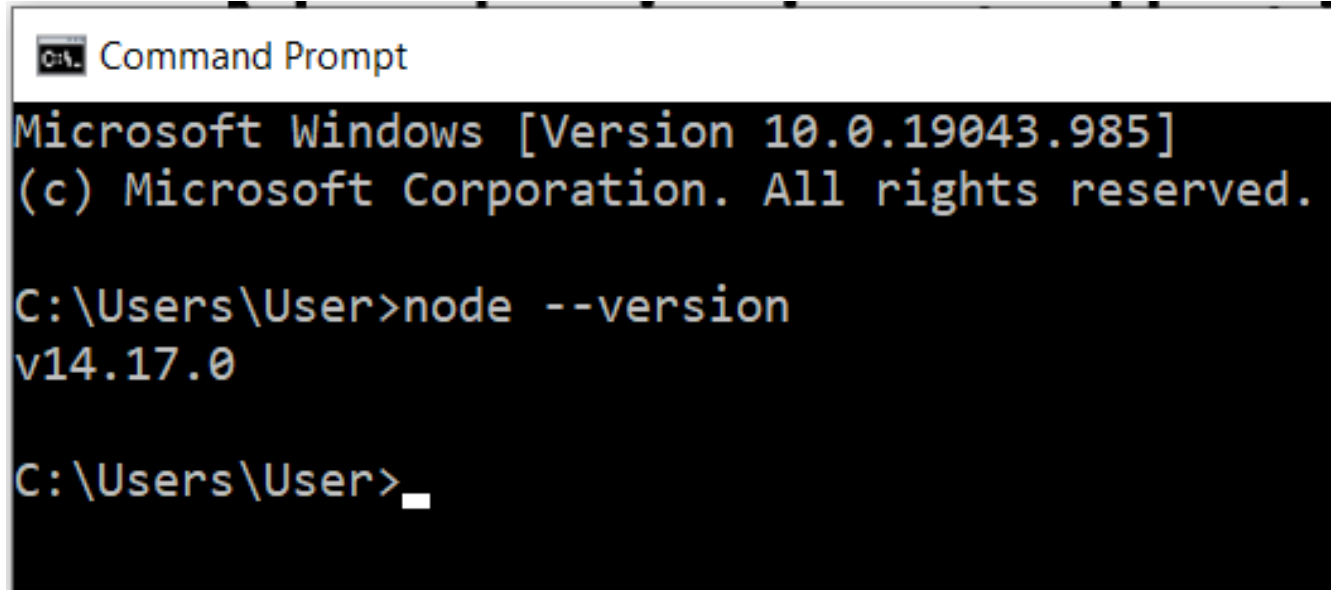
Node.js installation

- Open the setup file.
- Click on Next, accept, Next multiple times, then Install.



Install check

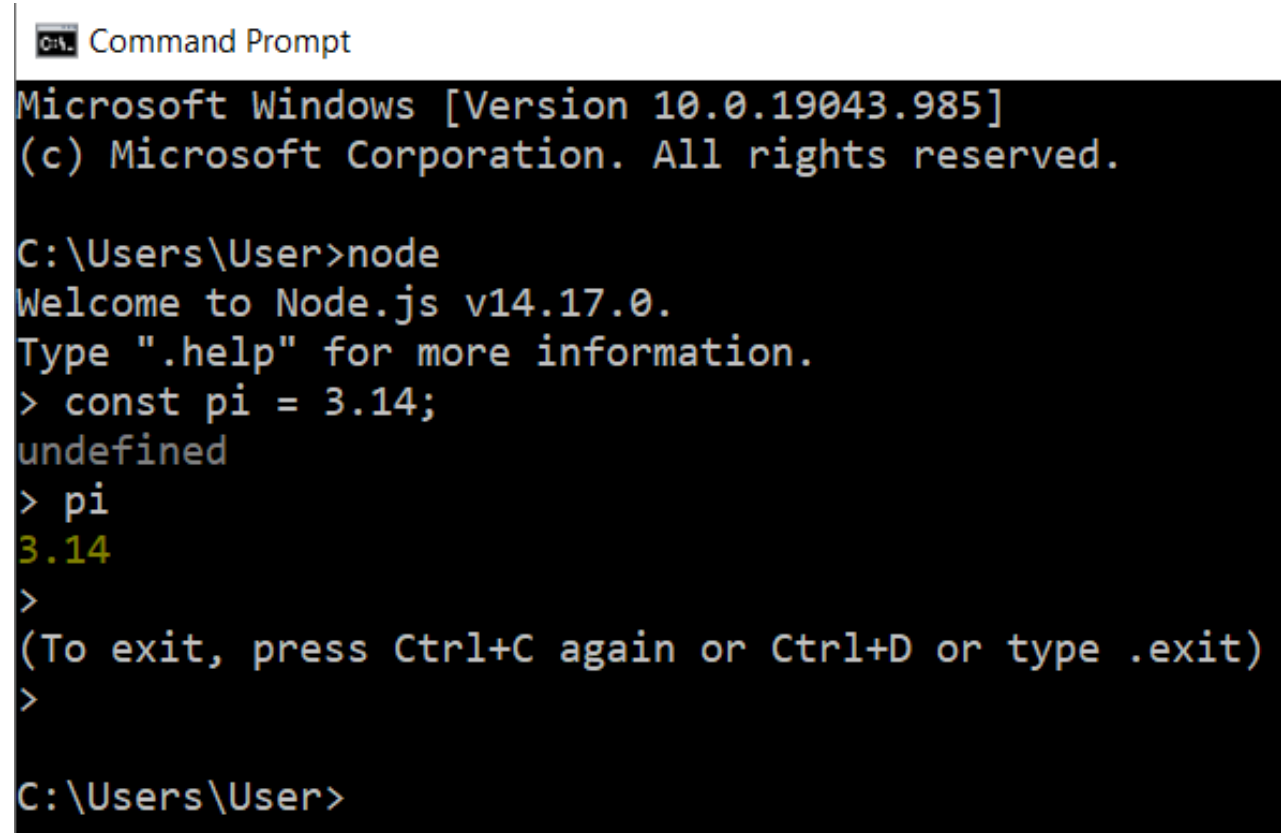
- Open a terminal or command prompt.
- Write **node --version** in the terminal and press **Enter**.
- It should display something like v14.17.0 in the terminal.

A screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The window content shows the following text:

```
Microsoft Windows [Version 10.0.19043.985]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Users\User>node --version  
v14.17.0  
  
C:\Users\User>_
```

Evaluate code: REPL Mode

- Open a terminal or command prompt.
- Type **node** with no argument, press Enter.
- You are in the so-called REPL mode. You can execute short codes here.
- To leave this mode, CTRL + C twice.

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The text inside shows the Windows version and copyright information, followed by the command prompt path "C:\Users\User>". The user has typed "node", and the prompt shows "Welcome to Node.js v14.17.0." and "Type '.help' for more information." The user has entered a command "> const pi = 3.14;" and the prompt shows "undefined". The user has entered another command "> pi" and the prompt shows "3.14" in green. The user has entered a third command ">" and the prompt shows "(To exit, press Ctrl+C again or Ctrl+D or type .exit)". The user has entered a fourth command ">" and the prompt shows "C:\Users\User>".

```
Command Prompt
Microsoft Windows [Version 10.0.19043.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\User>node
Welcome to Node.js v14.17.0.
Type ".help" for more information.
> const pi = 3.14;
undefined
> pi
3.14
>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
>
C:\Users\User>
```


Evaluate code: CLI Mode

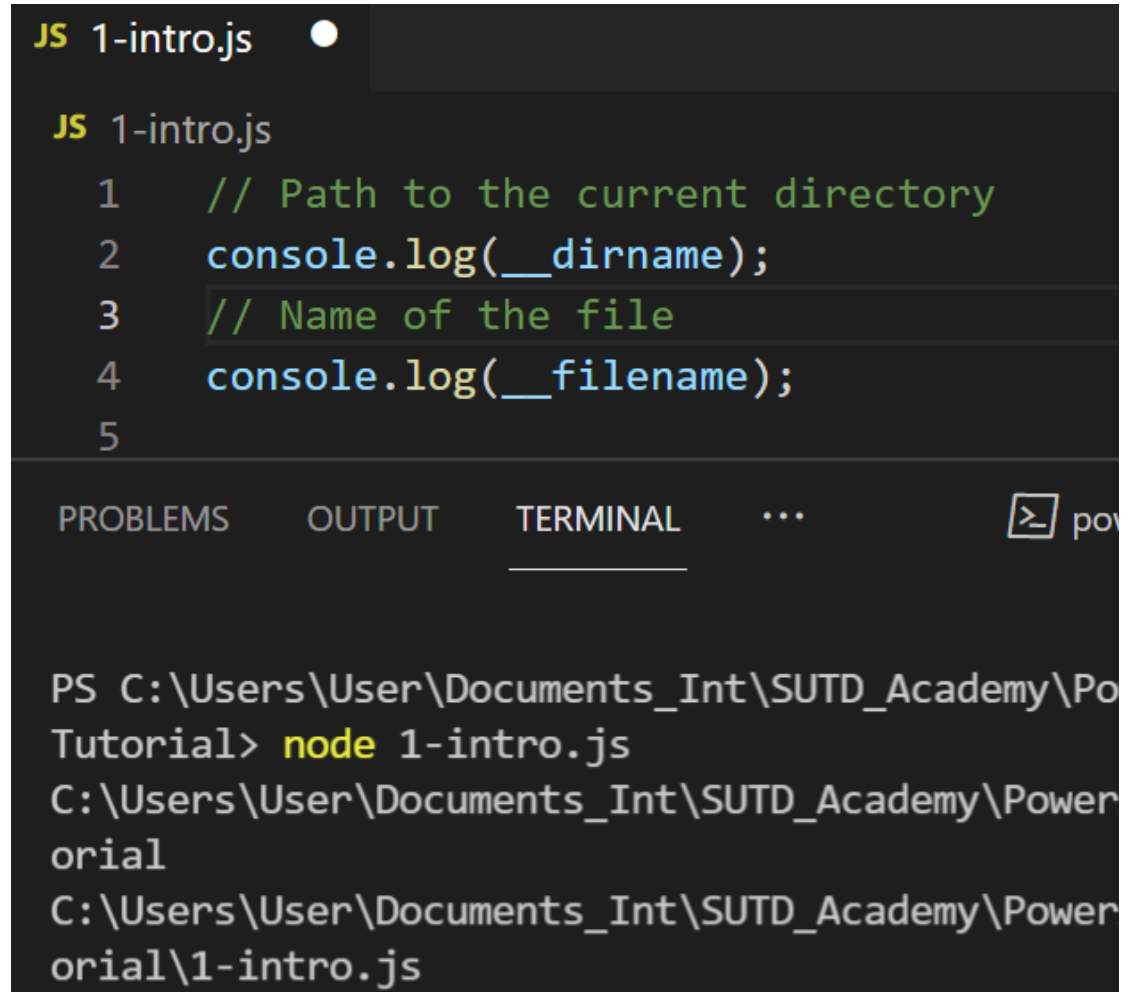
- Write a js file in some folder (e.g., app.js)
- Navigate in a console to this folder (using cd).
- Type **node app.js** (or even **node app**) and press Enter. It will execute the program.
- For the Visual Studio users, with CTRL + `, you can open a terminal below your code. Very convenient! ;)

```
JS app.js > ...  
1   const temperature = 20;  
2  
3   if (temperature < 15) {  
4       console.log("Cold");  
5   }  
6   else {  
7       console.log("Warm");  
8   }  
9   console.log("App test");
```

```
C:\Users\User\Documents_Int\SUT  
ndDev\Tutorial>node app.js  
Warm  
App test
```

Two global variables to know

- Before cutting to the chase, you need to know two important global variables:
- `__dirname`: path to the current directory
- `__filename`: name of the file
- There are many other global variables that we will partially see later.



```
JS 1-intro.js
JS 1-intro.js
1  // Path to the current directory
2  console.log(__dirname);
3  // Name of the file
4  console.log(__filename);
5

PROBLEMS  OUTPUT  TERMINAL  ...  [icon] pow

PS C:\Users\User\Documents_Int\SUTD_Academy\Power
Tutorial> node 1-intro.js
C:\Users\User\Documents_Int\SUTD_Academy\Power
orial
C:\Users\User\Documents_Int\SUTD_Academy\Power
orial\1-intro.js
```

Node modules

- Node modules allow you to extend the default functionality available to Node.js scripts and to share variables, functions, objects, etc. with other files.
- Every JS file is a module.
- Some modules are installed by default but some modules need to be installed.
- You can also create your own modules.
- `console.log(module)` gives you access to all the properties of the current module.
- The **exports** property indicates what variables from the module are exported. By default, none!

```
JS 2-firstModule.js > ...
1  const mum = "Sara"
2  const bro = "Hai Dang";
3  const my_name = "Cyrille";
4  const my_secret_lover = "Dewi";
5
6  console.log(module);
7

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node 2-firstModule.js
Module {
  id: '.',
  path: 'C:\\Users\\User\\Desktop\\Tutorial',
  exports: {},
  parent: null,
  filename: 'C:\\Users\\User\\Desktop\\Tutorial\\2-firstModule.js',
  loaded: false,
  children: [],
  paths: [
    'C:\\Users\\User\\Desktop\\Tutorial\\node_modules',
    'C:\\Users\\User\\Desktop\\node_modules',
    'C:\\Users\\User\\node_modules',
    'C:\\Users\\node_modules',
    'C:\\node_modules'
  ]
}
```

Node modules

- You can specify variables and functions allowed for export by setting the `module.exports` object accordingly.
- In the example, note that `my_secret_lover` is not shared, and its value cannot be accessed outside of the file.

```
JS 2-firstModule.js > ...
1  const mum = "Sara"
2  const bro = "Hai Dang";
3  const my_name = "Cyrille";
4  const my_secret_lover = "Dewi";
5
6  module.exports = { mum, bro, my_name }
```

```
JS 3-secondModule.js > ...
1  const sayHi = (name) => {
2    |    console.log(`Hello ${name}!`)
3  }
4
5  module.exports = sayHi
```

Node modules

- If more than one variable is shared by module A, the export variables are stored in an object.
- To make use of module A variables into file B, you can assign to a const variable the shared object of A using the require() function.
- The argument of require() is a path to module A.
- To access the value of a single shared variable, you just need to use the name of the const variable (e.g., sayHi in 4-testModules.js).
- If an object was exported, you need to use the syntax:
object_name_of_B.shared_variable_of_A
- It returns undefined if a variable is not exported or does not exist in module A.

```
JS 4-testModules.js > ...
1  const names = require("./2-firstModule");
2  const sayHi = require("./3-secondModule");
3
4  sayHi("Sergey");
5  sayHi(names.my_name);
6  sayHi(names.bro);
7  sayHi(names.my_secret_lover);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node 4-testModules.js
Hello Sergey!
Hello Cyrille!
Hello Hai Dang!
Hello undefined!
PS C:\Users\User\Desktop\Tutorial> █
```

Other ways to export

- You can directly export a variable (here, e.g. list) using the dot notation after module.exports.
- In 6-testModules2.js file, list is an array/property of the listAndBooks object. The values of the array elements can be accessed using their indices within square brackets.
- The books variable has been exported under the reference toRead. Note that toRead is an object and that the values of its properties can be accessed using dot and/or square brackets notations.
- Also, be careful when you import a file A within a JS file B. In fact, you are invoking A within B. It means that if some calls are made into A, these calls will be performed during the import as well.

```
JS 5-other_exports.js > ...
1  module.exports.list = [3.14, 2.72];
2
3  const books = { "Steinbeck": "Of Mice and Men",
4                  "G. Orwell": "1984"}
5
6  module.exports.toRead = books;
7  console.log(7);
```

```
JS 6-testModules2.js > ...
1  const listAndBooks = require("./5-other_exports");
2
3  console.log(listAndBooks.list[0]);
4  console.log(listAndBooks.toRead.Steinbeck);
5  console.log(listAndBooks.toRead["G. Orwell"]);
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node .\6-testModules2.js
7
3.14
Of Mice and Men
1984
PS C:\Users\User\Desktop\Tutorial> 
```

Built-in modules

- **os**: provides operating system-related utility methods and properties.
- **path**: provides utilities for working with file and directory paths
- **fs**: enables interacting with the file system in a way modelled on standard POSIX functions
- **http**: the HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use.
- To import these modules, just write e.g.:
 - **const os = require('os');**
- Have a look on their documentation if necessary:
 - <https://nodejs.org/dist/latest-v14.x/docs/api/>

```
JS 9-builtin.js > ...
1  const os = require('os');
2  const path = require('path');
3  const fs = require('fs');
4  const http = require('http');
5
6  //example of use
7  console.log(os.release());
8

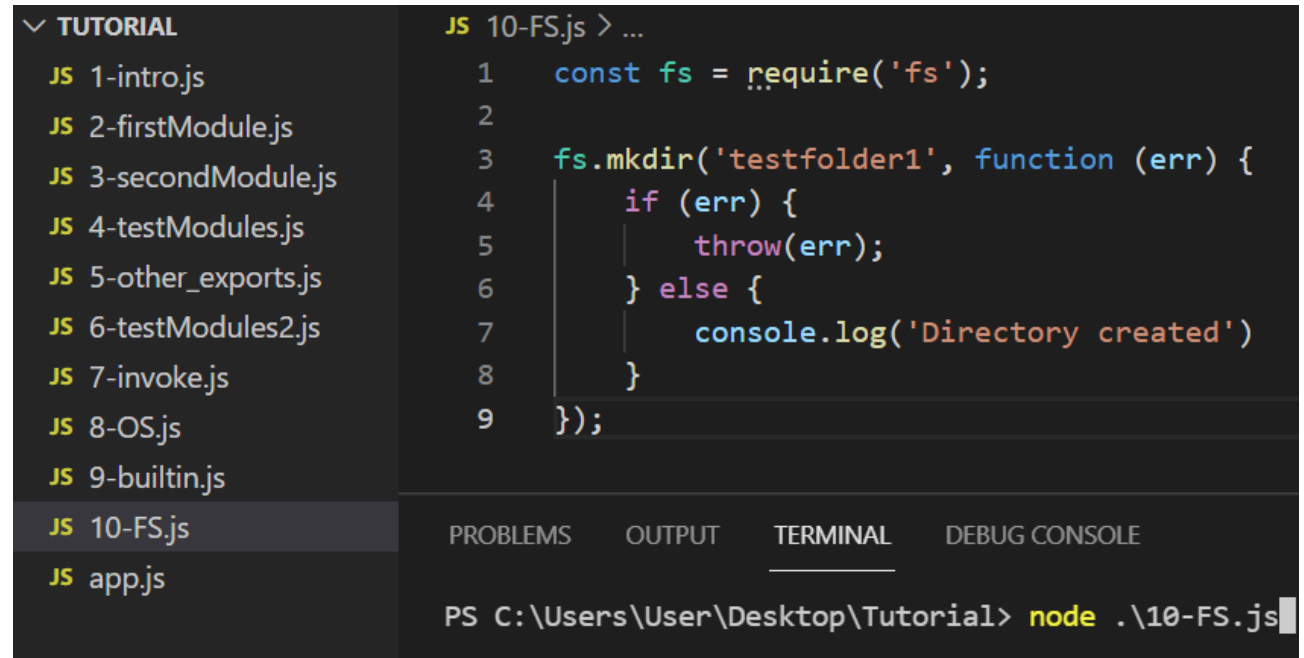
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node .\9-builtin.js
10.0.19043
PS C:\Users\User\Desktop\Tutorial> 
```

About fs module

- fs: enables interacting with the file system in a way modelled on standard POSIX functions
- POSIX (Portable Operating System Interface): a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
- defines the API, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.
- You probably know many fs functions:
 - mkdir, rm, etc.

Before execution

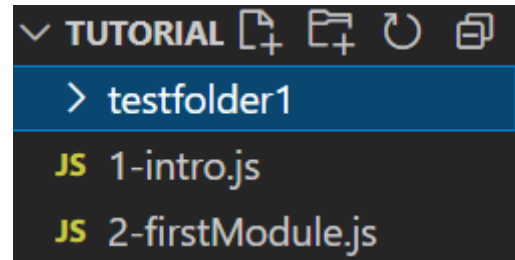


```
JS 10-FS.js > ...
1  const fs = require('fs');
2
3  fs.mkdir('testfolder1', function (err) {
4    if (err) {
5      throw(err);
6    } else {
7      console.log('Directory created')
8    }
9  });
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node .\10-FS.js

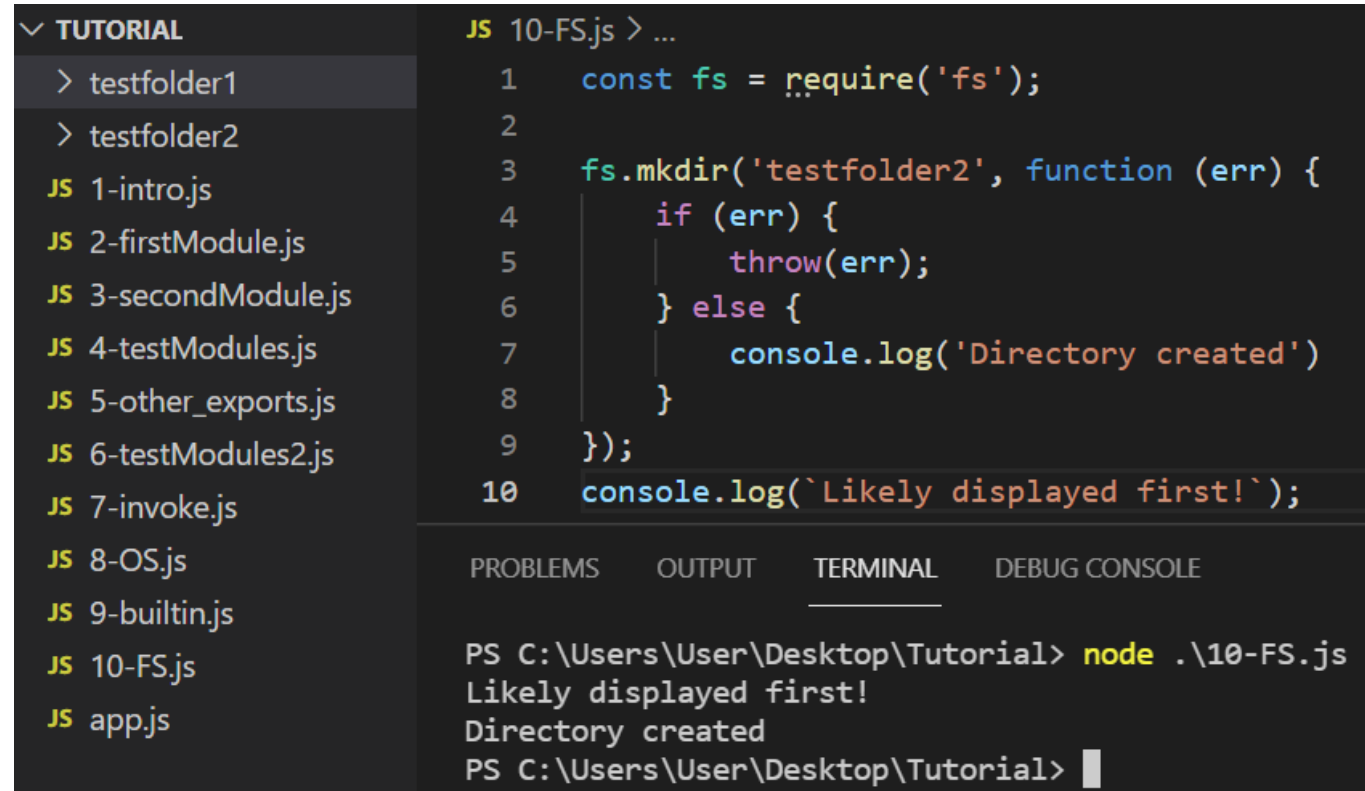
After execution



```
▼ TUTORIAL
> testfolder1
JS 1-intro.js
JS 2-firstModule.js
```


Asynchronous execution

- In the example, the 2nd argument is a function executed once the folder creation has completed.
- By default, node.js runs asynchronously, meaning that the code can continue on doing other things while waiting for an action to complete.
- Although the console.log at line 10 comes after the fs.mkdir method, it starts executing immediately after the method call and will not wait for the folder creation to be completed.



```

TUTORIAL
> testfolder1
> testfolder2
JS 1-intro.js
JS 2-firstModule.js
JS 3-secondModule.js
JS 4-testModules.js
JS 5-other_exports.js
JS 6-testModules2.js
JS 7-invoke.js
JS 8-OS.js
JS 9-builtin.js
JS 10-FS.js
JS app.js

JS 10-FS.js > ...
1  const fs = require('fs');
2
3  fs.mkdir('testfolder2', function (err) {
4      if (err) {
5          throw(err);
6      } else {
7          console.log('Directory created')
8      }
9  });
10 console.log(`Likely displayed first!`);

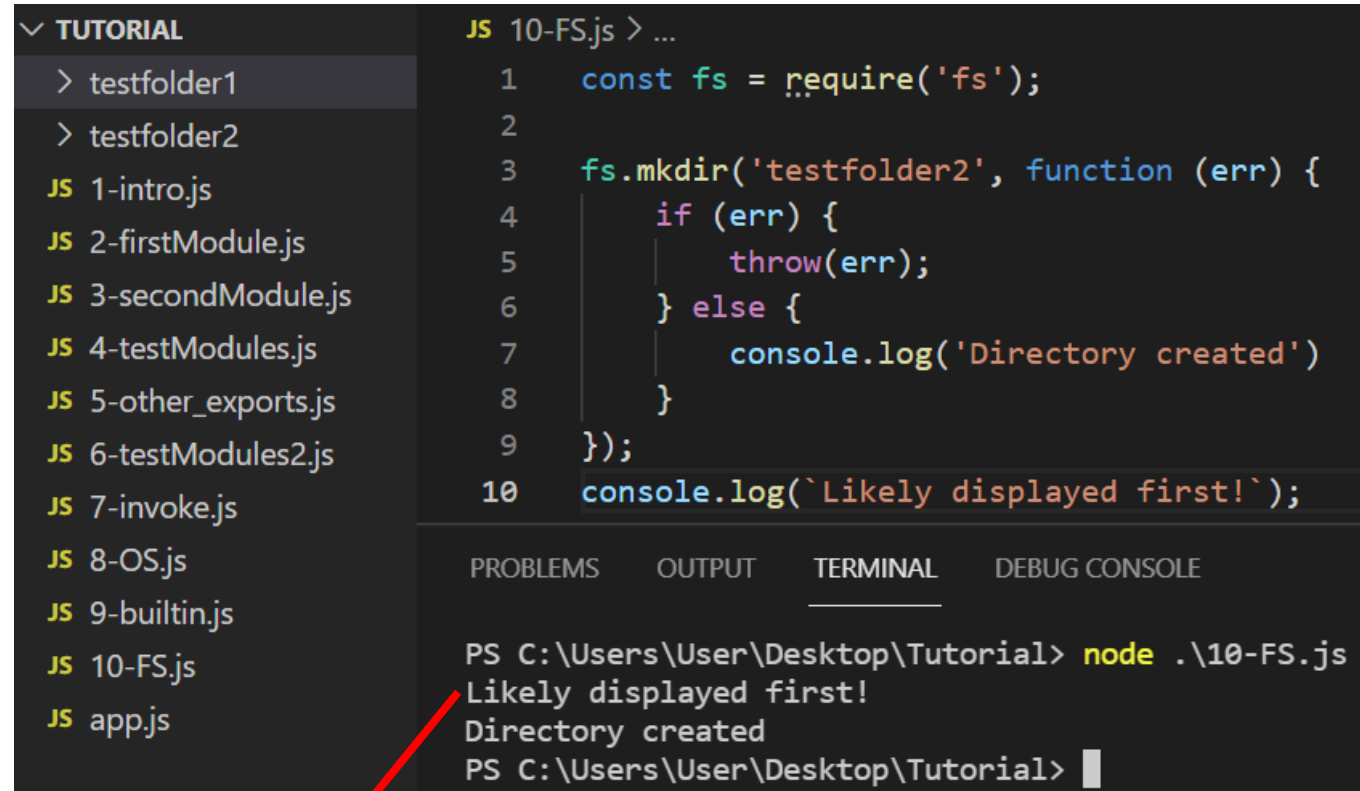
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node .\10-FS.js
Likely displayed first!
Directory created
PS C:\Users\User\Desktop\Tutorial>

```

Asynchronous execution

- In the example, the 2nd argument is a function executed once the folder creation has completed.
- By default, node.js runs asynchronously, meaning that the code can continue on doing other things while waiting for an action to complete.
- Although the console.log at line 10 comes after the fs.mkdir method, it starts executing immediately after the method call and will not wait for the folder creation to be completed.



```
JS 10-FS.js > ...
1  const fs = require('fs');
2
3  fs.mkdir('testfolder2', function (err) {
4      if (err) {
5          throw(err);
6      } else {
7          console.log('Directory created')
8      }
9  });
10 console.log(`Likely displayed first!`);
```

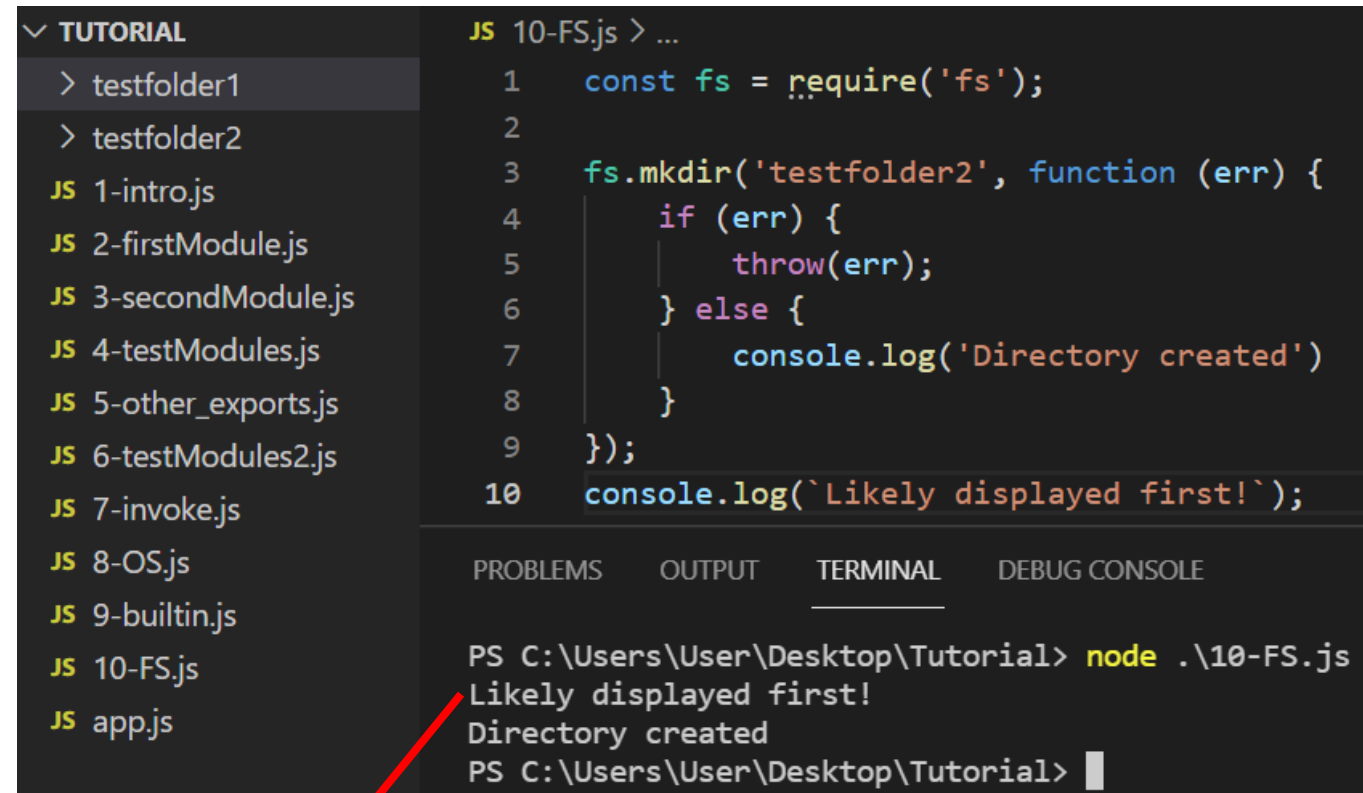
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\10-FS.js
Likely displayed first!
Directory created
PS C:\Users\User\Desktop\Tutorial>
```

In general, this message will be indeed displayed first!

Asynchronous execution

- Node.js is **nonblocking**, ie, allows other code to run while a specific op that could take time is executing.
- Main advantage: this “long” operation does not block the rest of the code.
- When this operation is complete, in general, a callback function (e.g., the 2nd argument of `fs.mkdir`) is executed.



```

TUTORIAL
> testfolder1
> testfolder2
JS 1-intro.js
JS 2-firstModule.js
JS 3-secondModule.js
JS 4-testModules.js
JS 5-other_exports.js
JS 6-testModules2.js
JS 7-invoke.js
JS 8-OS.js
JS 9-builtin.js
JS 10-FS.js
JS app.js

JS 10-FS.js > ...
1  const fs = require('fs');
2
3  fs.mkdir('testfolder2', function (err) {
4      if (err) {
5          throw(err);
6      } else {
7          console.log('Directory created')
8      }
9  });
10 console.log(`Likely displayed first!`);

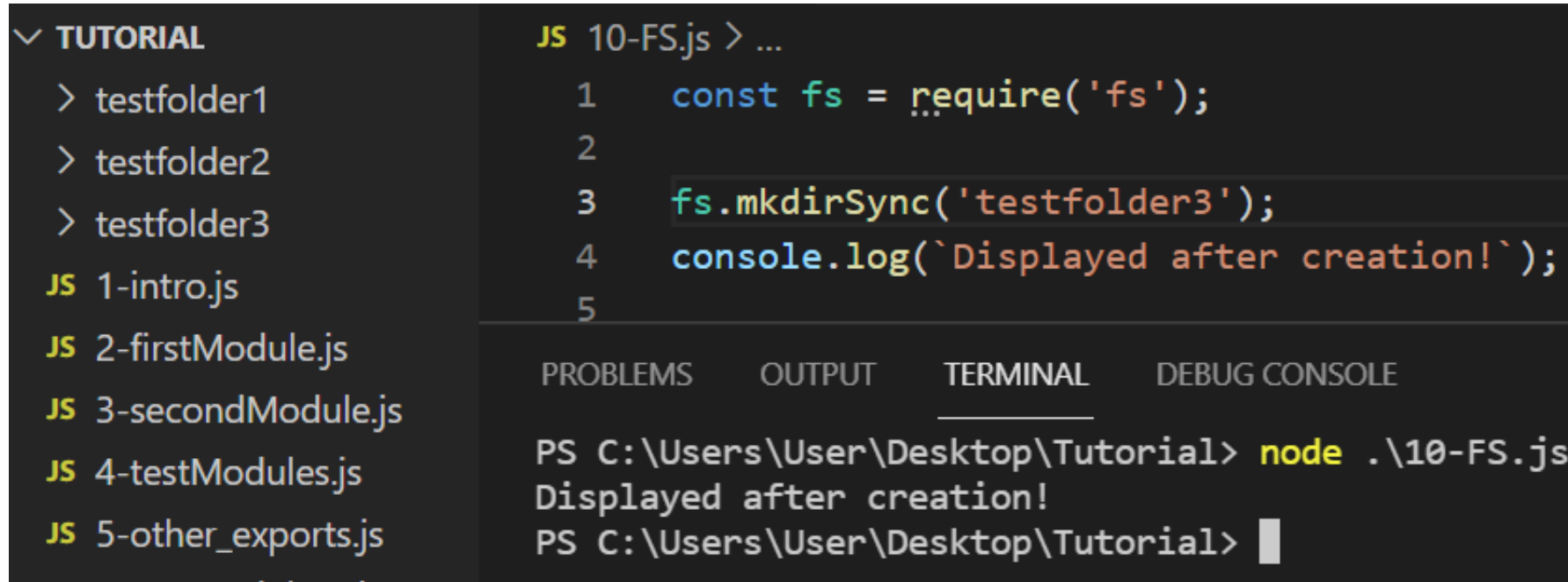
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\User\Desktop\Tutorial> node .\10-FS.js
Likely displayed first!
Directory created
PS C:\Users\User\Desktop\Tutorial>

```

In general, this message will be indeed displayed first!

Synchronous execution



The screenshot shows the Visual Studio Code interface. On the left, a file explorer shows a project named 'TUTORIAL' with several files: 'testfolder1', 'testfolder2', 'testfolder3', '1-intro.js', '2-firstModule.js', '3-secondModule.js', '4-testModules.js', and '5-other_exports.js'. The file '10-FS.js' is open in the editor. The code in the editor is as follows:

```
JS 10-FS.js > ...  
1  const fs = require('fs');  
2  
3  fs.mkdirSync('testfolder3');  
4  console.log(`Displayed after creation!`);  
5
```

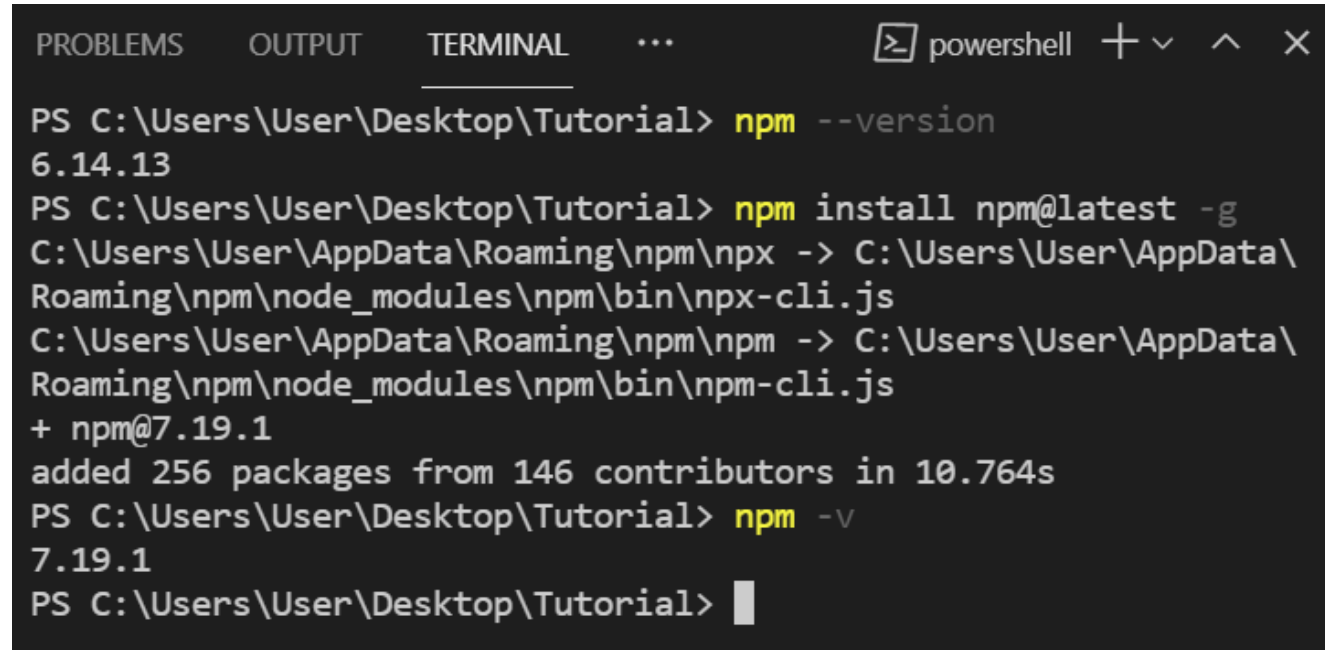
Below the editor, the 'TERMINAL' tab is active, showing the command prompt output:

```
PS C:\Users\User\Desktop\Tutorial> node .\10-FS.js  
Displayed after creation!  
PS C:\Users\User\Desktop\Tutorial>
```

- Node.js offers synchronised versions of most functions though it is often not considered the best practice to use them as it slows down the program execution among other issues.
- The name of these functions often end by Sync: e.g. fs.mkdirSync.
- The code below the fs.mkdirSync call runs only after completion of the folder creation.

NPM

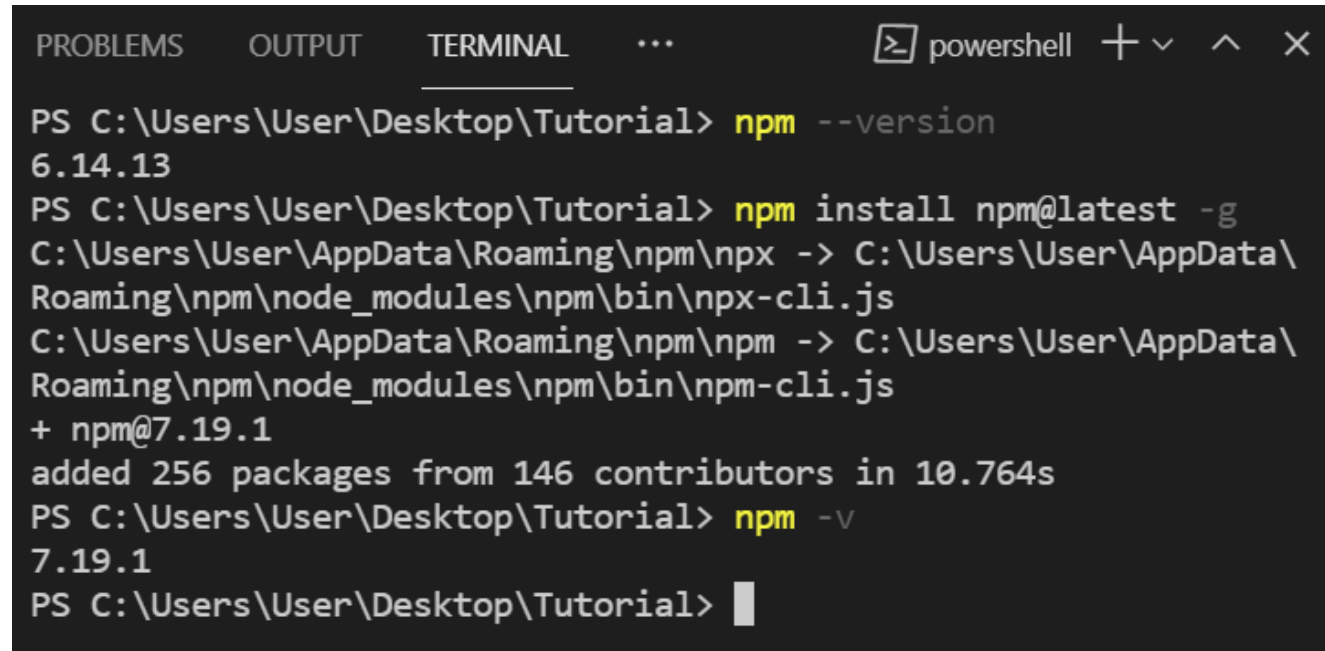
- Node automatically installed the Node Package Manager (NPM).
- NPM enables to:
 - Reuse your own code in other projects.
 - Use code written by other developers
 - Share our solutions with other developers.
- NPM project is hosted at www.npmjs.com
- Well-known and useful frameworks are hosted there: React, Express, etc.
- To check your npm version, just execute **npm --version** in the console and update it with **npm install npm@latest -g**



```
PROBLEMS OUTPUT TERMINAL ... powershell + v ^ x
PS C:\Users\User\Desktop\Tutorial> npm --version
6.14.13
PS C:\Users\User\Desktop\Tutorial> npm install npm@latest -g
C:\Users\User\AppData\Roaming\npm\npx -> C:\Users\User\AppData\Roaming\npm\node_modules\npm\bin\npx-cli.js
C:\Users\User\AppData\Roaming\npm\npm -> C:\Users\User\AppData\Roaming\npm\node_modules\npm\bin\npm-cli.js
+ npm@7.19.1
added 256 packages from 146 contributors in 10.764s
PS C:\Users\User\Desktop\Tutorial> npm -v
7.19.1
PS C:\Users\User\Desktop\Tutorial> 
```

NPM: local and global dependencies

- To install a package locally (for use only in a particular project):
 - `npm i package_name`
- To install a package globally (for use in any project):
 - `npm install -g package_name`
- To install a package in the local `node_modules` folder. (it will be created automatically at your first npm call)
 - `npm install package_name`
- To uninstall a package:
 - `npm uninstall package_name`



```
PROBLEMS OUTPUT TERMINAL ... powershell + v ^ x
PS C:\Users\User\Desktop\Tutorial> npm --version
6.14.13
PS C:\Users\User\Desktop\Tutorial> npm install npm@latest -g
C:\Users\User\AppData\Roaming\npm\npx -> C:\Users\User\AppData\Roaming\npm\node_modules\npm\bin\npx-cli.js
C:\Users\User\AppData\Roaming\npm\npm -> C:\Users\User\AppData\Roaming\npm\node_modules\npm\bin\npm-cli.js
+ npm@7.19.1
added 256 packages from 146 contributors in 10.764s
PS C:\Users\User\Desktop\Tutorial> npm -v
7.19.1
PS C:\Users\User\Desktop\Tutorial> 
```

NPM: example with the Express package

- We will use later Express to create a web server for an application.
- Now that Express is installed within the project folder, we can make use of it as previously with `require()` function.

```
PS C:\Users\User\Desktop\Tutorial> npm install express

added 50 packages, and audited 51 packages in 5s

found 0 vulnerabilities
PS C:\Users\User\Desktop\Tutorial> 
```

```
JS 11-exampress.js > ...
```

```
1  const express = require("express");
2  let http = express();
3  console.log("We can now use Express.")
4  console.log(http);
```

PROBLEMS

OUTPUT

TERMINAL

DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\11-exampress.js
```

```
We can now use Express.
```

```
<ref *1> [Function: app] {
  _events: [Object: null prototype] { mount: [Function: onmount] },
  _eventsCount: 1,
  _maxListeners: undefined,
```

Event loop

- One of the most important aspects of node.js.
- Allows node.js to perform non-blocking I/O, in spite of JS being single-threaded, by offloading operations to the system kernel whenever possible.
- Note: Most of modern kernels are not single-threaded though.
- Very complex topic, so let's start first by a simple example next slide

Sources:

<https://nodejs.dev/learn/the-nodejs-event-loop>

<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>

Many videos about Event loops from simple to complex on Youtube.

Call stack

- LIFO stack: last in, first out
- FIFO stack: first in, first out
- The call stack is a LIFO stack data structure that stores information about the active subroutines of program.
- Why having a call stack?
 - to keep track of the point at which the program should return after subroutines finish executing.

```
JS 12-simple_el.js > [🔍] foo
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    bar();
8    baz();
9  }
10
11  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\12-simple_el.js
foo
bar
baz
PS C:\Users\User\Desktop\Tutorial> █
```

A simple call stack

```
JS 12-simple_el.js > [e] foo
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    bar();
8    baz();
9  }
10
11 foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\12-simple_el.js
foo
bar
baz
PS C:\Users\User\Desktop\Tutorial>
```

The event loop on every iteration looks if there's something in the call stack, and executes it until the call stack is empty.



Call stack with callback

- So far, nothing surprising, everything is executed in order.
- Now, let's execute a program with a callback.
- `setTimeout` makes a call to a function (1st argument) and executes it after a timer in milliseconds (2nd argument) has expired... *but only after the other functions in the call stack has executed.*

```
JS 13-simple_cb.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    baz();
9  }
10
11  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\13-simple_cb.js
foo
baz
bar
PS C:\Users\User\Desktop\Tutorial> █
```

Call stack with callback

```
JS 13-simple_cb.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    baz();
9  }
10
11  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\13-simple_cb.js
foo
baz
bar
PS C:\Users\User\Desktop\Tutorial> █
```

Callback functions are put in a FIFO **message queue**, executed once the call stack is empty.



Call stack with promises

- ES6 introduced the concept of job queues used notably by promises
- Now, let's execute a program with a callback and a promise.
- Job queue offers a way to execute the result of an async function as soon as possible, rather than being put at the end of the call stack.
- Promises that resolve before the current function ends will be executed right after the current function and before the callbacks.

```
JS 14-simple_ps.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    new Promise((resolve, reject) =>
9      resolve('should be right after baz, before bar')
10    ).then(resolve => console.log(resolve))
11    baz();
12  }
13
14  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\14-simple_ps.js
foo
baz
should be right after baz, before bar
bar
PS C:\Users\User\Desktop\Tutorial> █
```

Call stack with promises (I)

```
JS 14-simple_ps.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    new Promise((resolve, reject) =>
9      resolve('should be right after baz, before bar')
10    ).then(resolve => console.log(resolve))
11    baz();
12  }
13
14  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\14-simple_ps.js
foo
baz
should be right after baz, before bar
bar
PS C:\Users\User\Desktop\Tutorial> |
```



Call stack with promises (II)

```
JS 14-simple_ps.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    new Promise((resolve, reject) =>
9      resolve('should be right after baz, before bar'))
10   ).then(resolve => console.log(resolve))
11   baz();
12 }
13
14 foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\14-simple_ps.js
foo
baz
should be right after baz, before bar
bar
PS C:\Users\User\Desktop\Tutorial> 
```



The message queue puts you at the back of the queue, while the job queue is a fast pass ticket.

Call stack with promises (III)

```
JS 14-simple_ps.js > ...
1  const bar = () => console.log('bar');
2
3  const baz = () => console.log('baz');
4
5  const foo = () => {
6    console.log('foo');
7    setTimeout(bar, 0);
8    new Promise((resolve, reject) =>
9      resolve('should be right after baz, before bar')
10    ).then(resolve => console.log(resolve))
11    baz();
12  }
13
14  foo();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\14-simple_ps.js
foo
baz
should be right after baz, before bar
bar
PS C:\Users\User\Desktop\Tutorial> |
```



The message queue puts you at the back of the queue, while the job queue is a fast pass ticket.

Call stack with promises (summary)



Event-driven programming

- In JS, browser-side, the actions of the users are handled through **events**: reaction to mouse movements, clicks, buttons, etc.
- Is this event-driven programming style applicable server-side? Yes.
- Listen for specific events and execute callback functions in response of these events

Event emitter

- Backend-side, we need the **events** module and its **EventEmitter** class.
- An instance of this class exposes, among many others, the **on** and **emit** methods.
- **on** is used to add a callback function that's going to be executed when the event is triggered.
- **emit** is used to trigger an event.
- More on the **events** module at <https://nodejs.org/api/events.html>

```
JS 15-event.js > ...
1  const EventEmitter = require('events');
2  const instEvEmitter = new EventEmitter();
3
4  instEvEmitter.on('start', ()=>{
5    console.log('Data received');
6  })
7
8  instEvEmitter.emit('start')
9  // Fyi, we could also pass arguments
10 // in emit and the event handler.
11
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\15-event.js
Data received
PS C:\Users\User\Desktop\Tutorial> 
```

Event emitter

- You can pass arguments in emit and the event handlers.
- You can also trigger several actions at once after listening for a specific event.
- Important: The emit must be placed after the on methods. You can't emit an action if you haven't listened for the event.

```
JS 15-event.js > ...
1  const EventEmitter = require('events');
2  const instEvEmitter = new EventEmitter();
3
4  instEvEmitter.on('start', ()=>{
5    |   console.log('Data received');
6  })
7
8  instEvEmitter.on('start', (a, b)=>{
9    |   console.log(`a + b is equal to ${a+b}.`);
10 })
11
12
13 instEvEmitter.emit('start', 3, 4)
14
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\15-event.js
Data received
a + b is equal to 7.
PS C:\Users\User\Desktop\Tutorial> █
```

About http module

- For recall, http is a client-server communication protocol used by browsers to view webpages.
- An http server is a software that understands URL addresses.
- The http module is used to create an http server.
- To create a server, use the `http.createServer()` method.

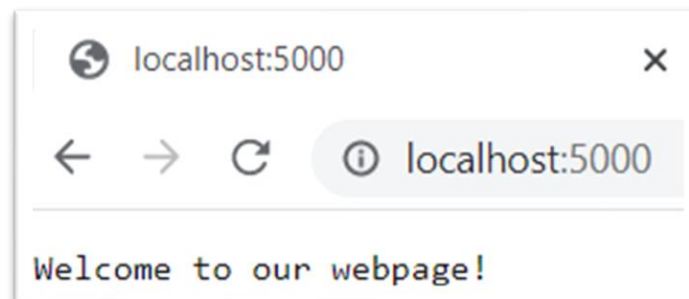
About http module

- The `createServer()` method looks for a callback with two object parameters, in practice, `req` and `res`.
- `req` represents the incoming request (e.g. a client is requesting from the web browser your webpage).
- `res` is what you are sending back.
- You also need to specify what port your server is listening to (e.g., here, 5000 corresponds to the number of the port).
- At the execution of the program, no exiting because the web server needs to keep on waiting for future incoming requests.
- After exiting with `CTRL+C`, refresh your browser. Note that localhost now refuses to connect.

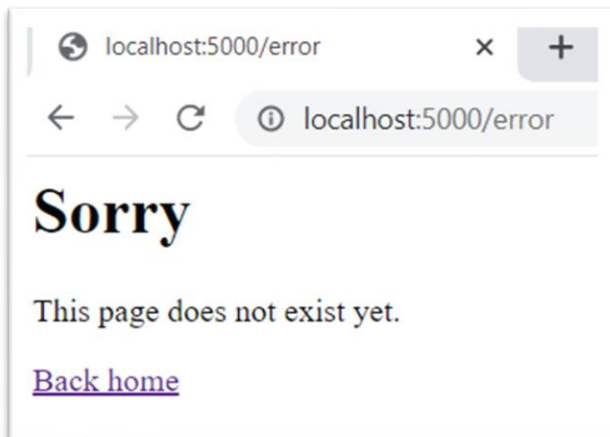
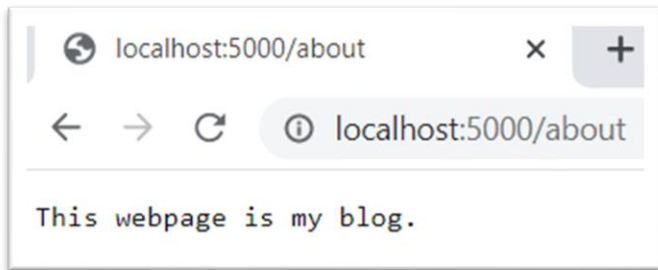
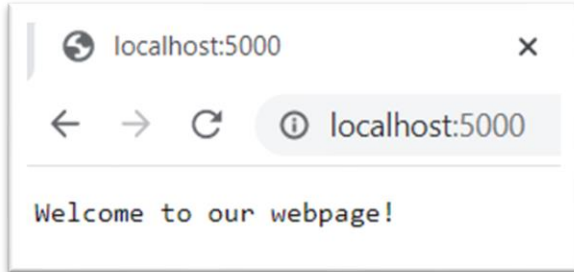
```
JS 16-http_module.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4      res.write("Welcome to our webpage!")
5      res.end()
6  })
7
8  const port = server.listen(5000);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\16-http_module.js
Server running at port undefined
PS C:\Users\User\Desktop\Tutorial> node .\16-http_module.js
```



More features about the http module



```
JS 17-http_req.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4      if (req.url === '/') {
5          res.end('Welcome to our webpage!')
6      }
7      if (req.url === '/about') {
8          res.end('This webpage is my blog.')
9      }
10     res.end(`
11         <h1>Sorry<\h1>
12         <p>This page does not exist yet.<\p>
13         <a href = "/">Back home<\a>
14     `)
15 });
16
17 const port = server.listen(5000);
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\17-http_req.js
```

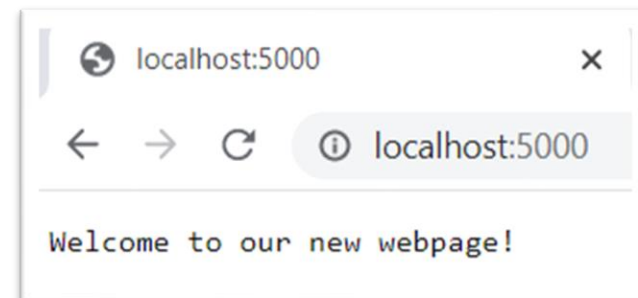
Create a server using Event Emitter API

- A server also has a **on** method and can listen for a **'request'** event.
- Once the request event occurs, the callback of the **on** method is triggered as previously.
- Servers can always listen a **'request'** event (see Node.js doc).
- In fact, even if it seems that we haven't set up an event, the class `http.Server` extends the class `EventEmitter`.
- Using events in the backend is very common in node.js!

```
JS 18-event-server.js > ...
1  const http = require('http');
2
3  //Alternative: use Event Emitter API
4  const server = http.createServer()
5
6  server.on('request', (req, res) => {
7    res.end("Welcome to our new webpage!");
8  })
9
10 const port = server.listen(5000);
```

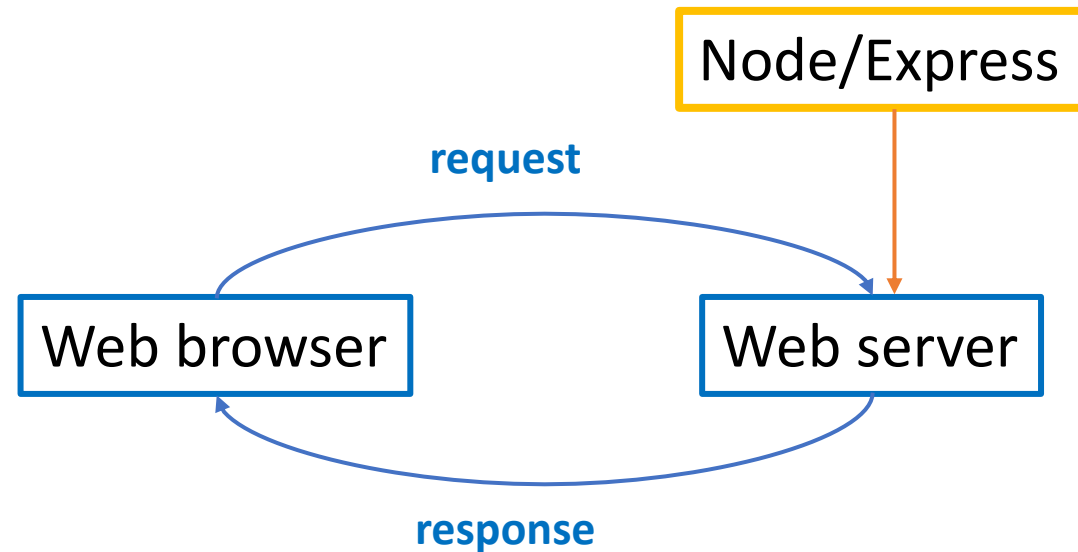
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\User\Desktop\Tutorial> node .\18-event-server.js
█
```



HTTP messages

- Every time we type a URL in a web browser, we perform a request to the server responsible for serving these resources.
- The server then sends a response to the request.
- These requests and responses are nothing else than data exchanged on the web called HTTP messages.
- Node.js is used to build web servers.
- Later, we will use Express.js, a Node framework, very useful for the creation of web servers.



HTTP: About the port numbers

- Port are communication endpoints in networking.
- Used to identify specific process and type of network service.
- Numbered from 0 to 65535.
- Port numbers are always associated with an IP address of a host and the type of transport protocol used for communication.
- Ports from 0 to 1023 are reserved for common TCP/IP applications and are called well-known ports.
- By default, HTTP uses port 80 and HTTPS uses port 443.
- For local deployment, you can use a value between 1024 and 49151. I use 5000.

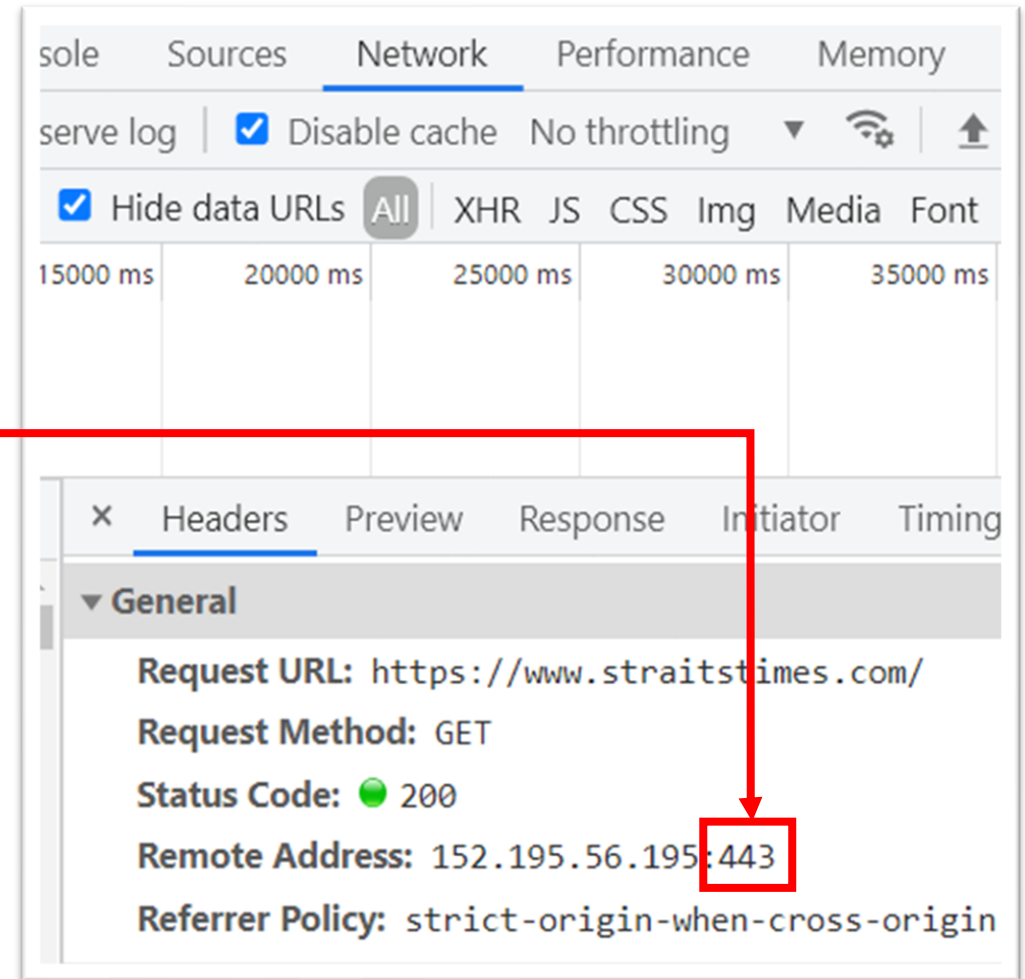
Notable well-known port numbers

Number	Assignment
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH) Secure Login
23	Telnet remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) email delivery
53	Domain Name System (DNS) service
67, 68	Dynamic Host Configuration Protocol (DHCP)
80	Hypertext Transfer Protocol (HTTP) used in the World Wide Web
110	Post Office Protocol (POP3)
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Message Access Protocol (IMAP) Management of digital mail
161	Simple Network Management Protocol (SNMP)
194	Internet Relay Chat (IRC)
443	HTTP Secure (HTTPS) HTTP over TLS/SSL

Source: [https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))

HTTP: About the port numbers

- Port are communication endpoints in networking.
- Used to identify specific process and type of network service.
- Numbered from 0 to 65535.
- Port numbers are always associated with an IP address of a host and the type of transport protocol used for communication.
- Ports from 0 to 1023 are reserved for common TCP/IP applications and are called well-known ports.
- By default, HTTP uses port 80 and HTTPS uses port 443.
- For local deployment, you can use a value between 1024 and 49151. I use 5000.



HTTP: More details

- Recall: no exiting in these programs. The server stays on, waiting for requests.
- Everytime you refresh the webpage (here, localhost:5000), a message is displayed in the console.
- Browser side, nothing happens because the response end() method is missing.
- The response end() method signals to the server that all of the response headers and body have been sent; that server should consider this message complete.
- The response end() method MUST be called on each response.
- More details at <https://nodejs.org/api/http.html>

The image shows a development environment with VS Code. The editor displays a file named `19-another_server.js` with the following JavaScript code:

```
JS 19-another_server.js > ...
1  const http = require('http')
2
3  const server = http.createServer((req, res) => {
4      console.log('User sent a request')
5      res.write('Home page')
6      //res.end()
7  })
8
9  server.listen(5000)
```

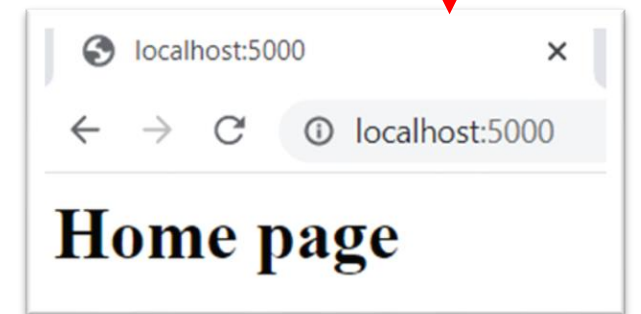
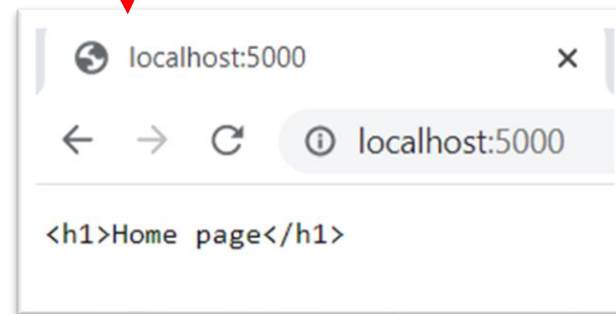
Below the editor is the integrated terminal. The top bar shows tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE. The terminal shows the command `node .\19-another_server.js` being executed, followed by three lines of output: `User sent a request`.

A red arrow originates from the `//res.end()` comment in the code and points to a new browser tab. The browser tab is titled "New Tab" and shows the address bar with `localhost:5000`.

HTTP headers

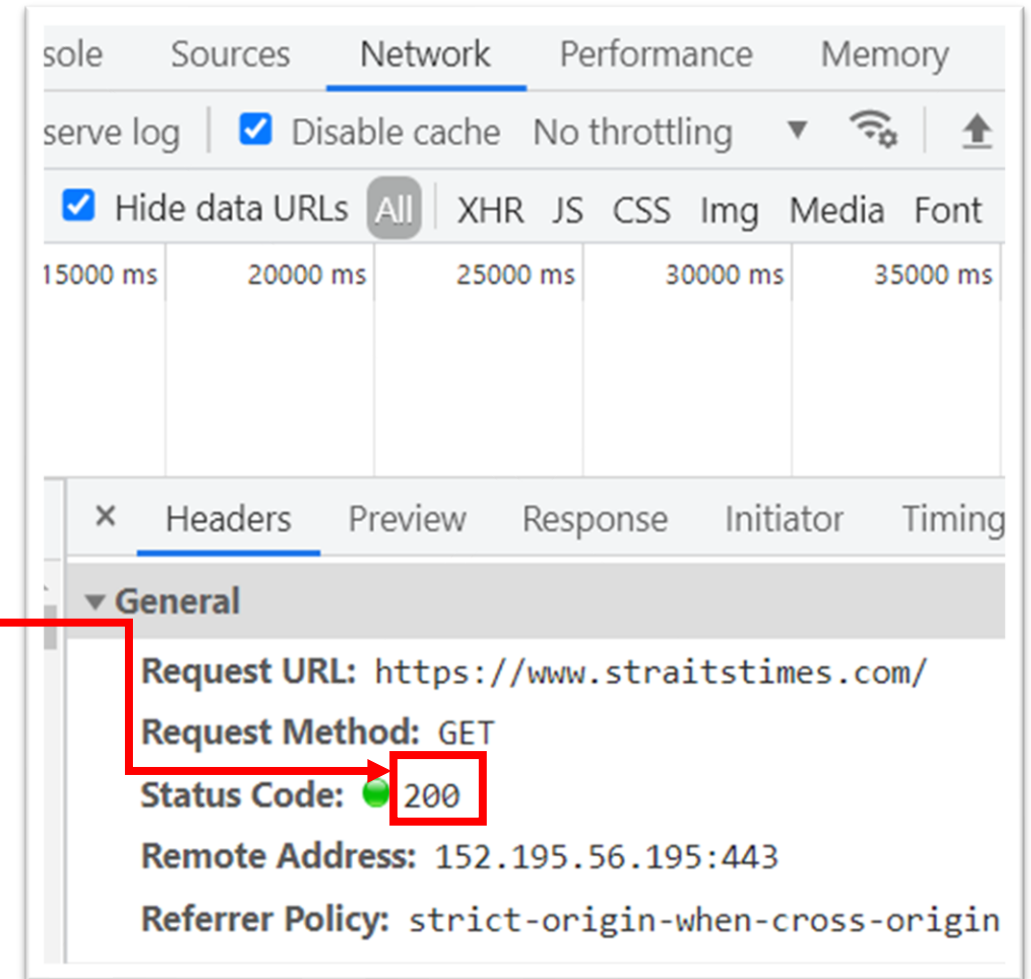
- You can send back metadata using the `res.writeHead()` method.
- In particular, you need to fetch `writeHead()` with two arguments: a status code (e.g., 200 which means “success” here) and a header.
- Headers are sent as object of key-value pairs. Common pairs are ‘content-type’: ‘text/html’ or ‘content-type’: ‘text/plain’.
- Note that these metadata matter as they impact the response.

```
JS 19-another_server.js > ...
1  const http = require('http')
2
3  const server = http.createServer((req, res) => {
4    res.writeHead(200, { 'content-type': 'text/plain' })
5    //res.writeHead(200, { 'content-type': 'text/html' })
6    res.write('<h1>Home page</h1>')
7    res.end()
8  })
9
10 server.listen(5000)
```



HTTP: About the status codes

- HTTP response status code indicate whether a specific HTTP request has been successfully completed.
- In the previous program, the status code informed the browser whether the request was successful or not.
- More on status codes at:
<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>
- A few standard status codes: 200 for ok (request is successful), 400 for Bad Request (the server could not understand the request due to invalid syntax), 404 for Not Found (the server can not find the requested resource). In the browser, this means the URL is not recognized.
- Cautious: it is your responsibility to indicate the correct status code in the header.



HTTP: html file

- Into the write method, you can directly pass the content of an html file.
- For that, it must have been previously read and all the necessary information must have been stored into a readFile variable. This operation requires fs.
- Why readFile Sync?
- Because we need to read first the potentially large content of the files which might be sent in the responses. => We need to guarantee that this reading operation has completed before setting up the server.

```
JS 20-http-html.js > ...
1  const { readFileSync } = require('fs')
2  const http = require('http')
3
4  const homepage = readFileSync('./index.html')
5  const server = http.createServer((req, res) => {
6      res.writeHead(200, { 'content-type': 'text/html' })
7      //res.writeHead(200, { 'content-type': 'text/plain' })
8      res.write(homepage)
9      res.end()
10 })
11
12 server.listen(5000)
```

```
<> index.html > ...
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5          <title>My blog</title>
6      </head>
7      <body>
8          <h1>Homepage</h1>
9          <p>This is my blog homepage</p>
10     </body>
11 </html>
```


Activity 1: Set up a simple web server

- Given the previous slides, set up with Node.js a simple web server using the the http module on a localhost which sends back the content of an html file.
- By default, the server should send back a valid status code and a response interpreted in html.
- If the requested URL is /plain, the server should send back a valid status code and a response interpreted in plain text.
- Otherwise, for any other URL requests, the server should send back an invalid status code, an error message with a back button bringing the user to the home page.
- On the server side, whenever the user hit the server, a message containing the type of the request (GET, POST, DELETE, etc...) should be displayed on the screen. Look at any Node.js documentations online if necessary.

Summary

- Installation of Node.js
- Introduction to Node.js
- Recall on Asynchronous execution
- NPM
- Http module and web server