WEB SPA APPLICATIONS WITH

React

Michał Jabłoński

About me:

Front – End developer since 2007 | | 2017 JS Full - Stack

Michał Jabłoński

2007	(X)HTML + CSS + JavaScript
2009	Flash + HTML + AS 2.0 + PHP
	…
2011	Flex + AS 3.0 + Java EE + GWT + JavaScript
	…
2017	JS ES6 + Angular + React + Vue + RxJS + TypeScript

https://github.com/michaljabi

[Road].map()

- News in JavaScript
- What is SPA ?
- Node.js environment
  - Transpilation
  - Bundling
  - Polyfills
- Front-Endu problems
- React and it’s place

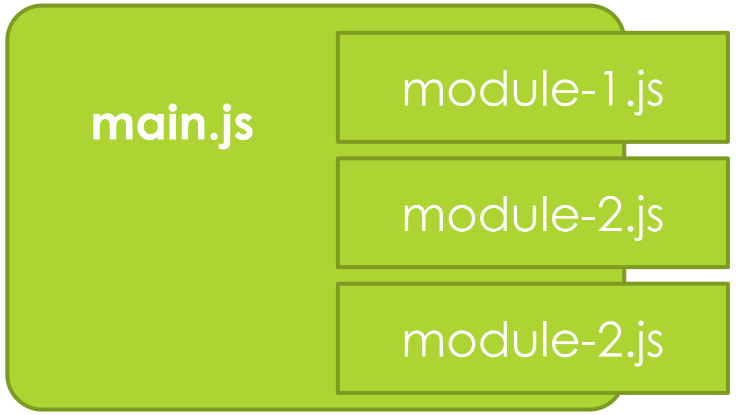
Glossary

- **ESNext -** Reference to current JS version – relatively. JS is developed year by year [https://en.wikipedia.org/wiki/ECMAScript]
- **Transpilation –** A process similar to compilation, except that the code remains at the same level of abstraction (it is still editable). We then change the JS version, e.g. from ES9 to ES5 (backwards compatibility issue).
- **Bundling –** The process of collecting all modules (.js files) and project dependencies (e.g. .css, .jpg, .svg files etc.) into one .js file and several assets files (.js, .css, .jpg, etc.) - preparation of the front-end project end for production with single .html file.

What happened to JavaScript?

Historical context

- In 2009 „Node.js" shows up – it changes the way how you structure your code



- After 2015 new sintactic sugar came up with e.g.:

class | Set() | Map() | ...spread | `interpolation ${x}`

More like...
C#
JAVA

New possibilities: ES6+

ES6, ES7, ES8, ES9, ES10 … ES-next

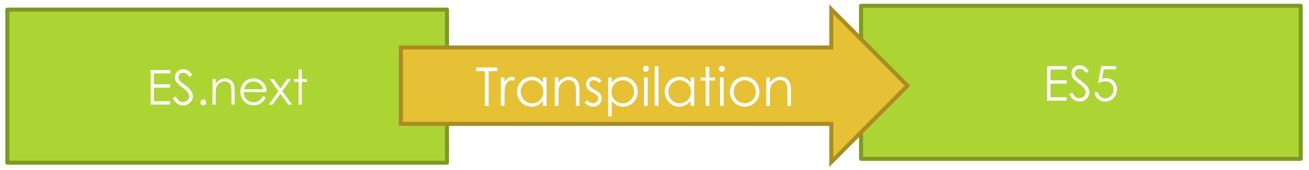
- Modularity
- Lexical declaration of variables: **let**, **const**
- Functions => Arrow
- Default values for function arguments
- Syntactic sugar for the **class**
- Destructuring objects and arrays
- Operators: ...spread and ...rest
- Text interpolation
- Generators
- Operators: **async** / **await**
- And other...

The essential concepts of the language do not change

- Prototypability
- Functions as a First Class Citizen
- Scope of variables
- The context
- Object-oriented nature of JavaScript

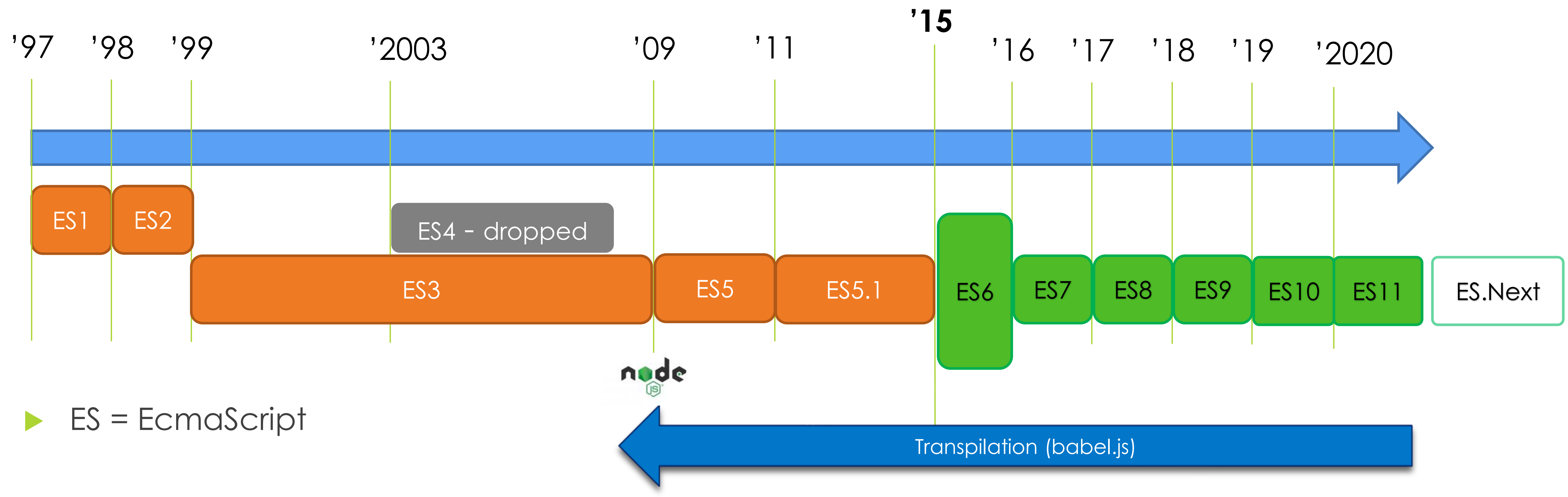
JavaScript development

- Good source: https://**en**.wikipedia.org/wiki/ECMAScript
- Braking changes since: **ES6+** (ES.next)
  - transpilers allowing you to use this syntax today



- Syntactic sugar to many things
- New methods for native code

JavaScript versions

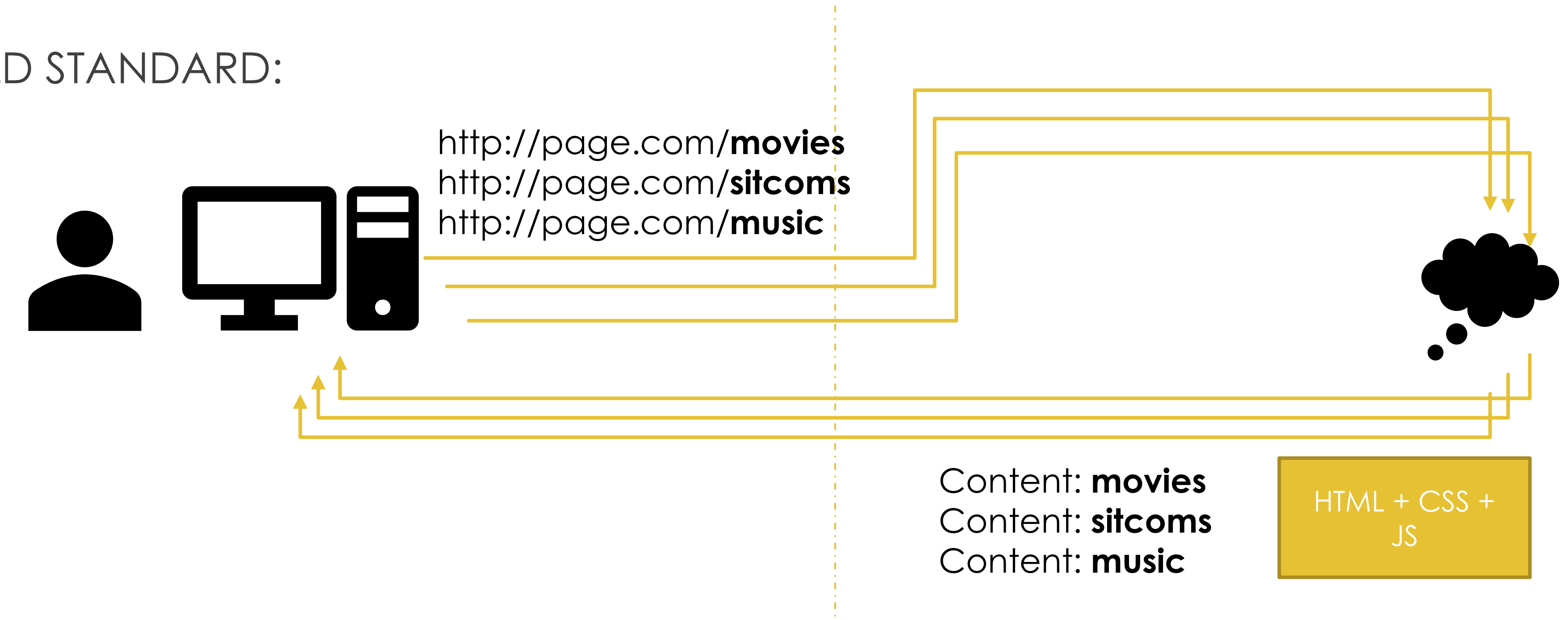


- ES = EcmaScript
- Since 2015, we have had syntactic novelties every year

SPA = Single Page Application

- The browser "does not lose its state" - we do not reload the page and subpages
- Most of the "ACTIONS" happen on the client's browser side

- THE GOOD OLD STANDARD:

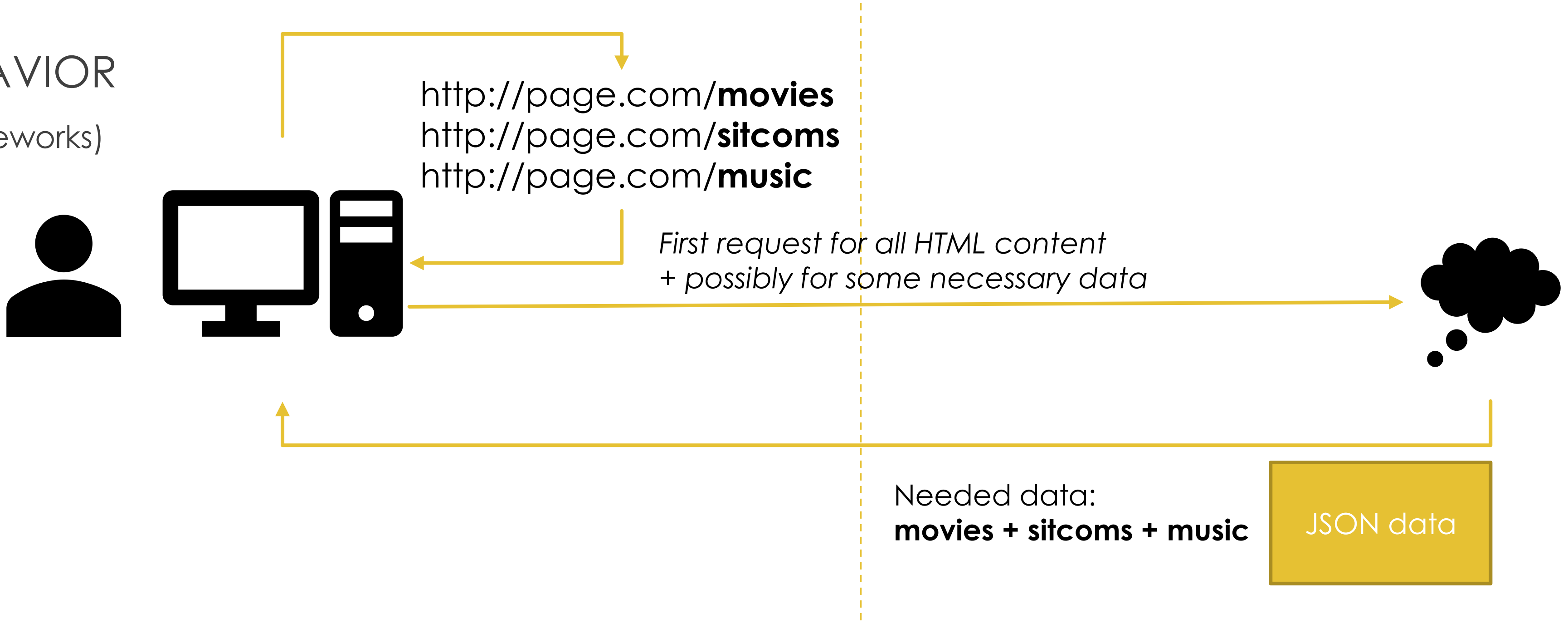

http://page.com/**movies**
http://page.com/**sitcoms**
http://page.com/**music**
Content: **movies**
Content: **sitcoms**
Content: **music**
HTML + CSS + JS

SPA

- Single Page Application
- The internal router monitors "what to show on the page" and changes the address bar depending on the content.

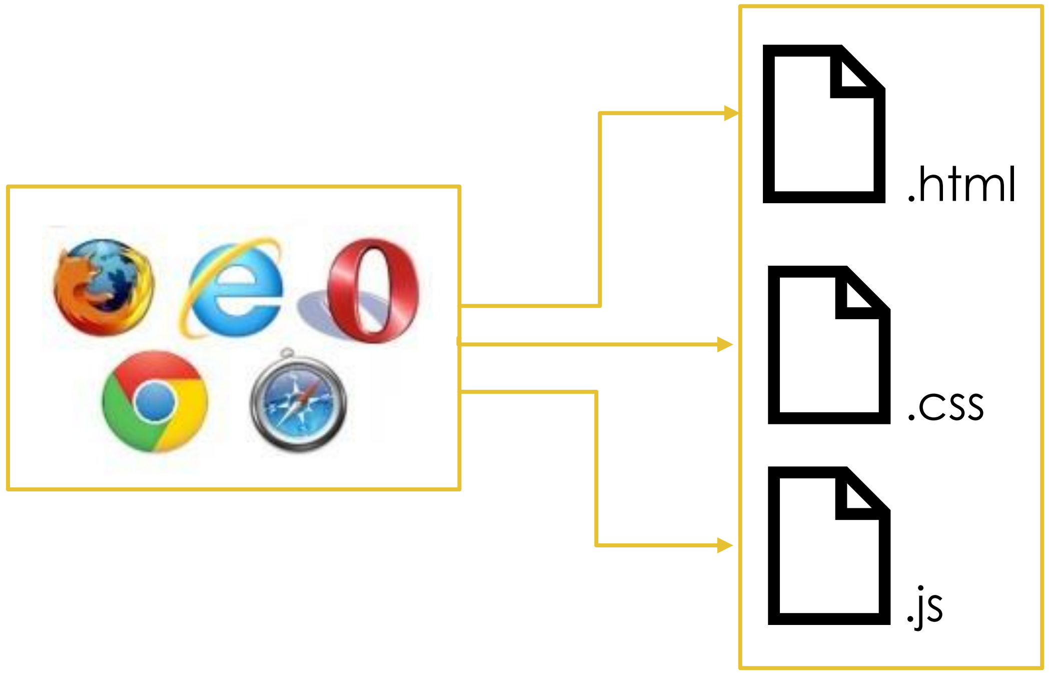
- THIS CHANGES BEHAVIOR

  (in JavaScript-based frameworks)

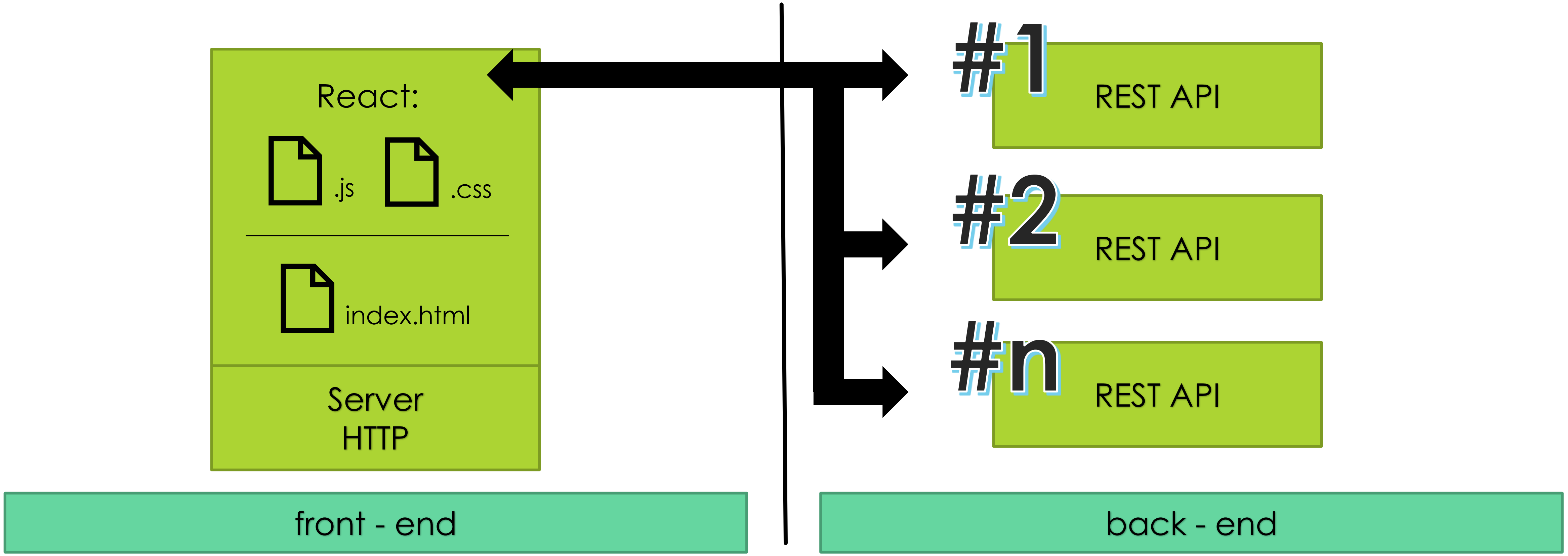

http://page.com/**movies**
http://page.com/**sitcoms**
http://page.com/**music**
First request for all HTML content
+ possibly for some necessary data
Needed data:
movies + sitcoms + music
JSON data

It all comes down to the basics

- HTML + CSS + JS
- Simplify:



Where do we need React?



Remote control via JavaScript

- JS frameworks
- The natural course and development of the AJAX (Asynchronous JavaScript and XML) concept
- JavaScript controls the DOM, dynamically "replacing" the content on the page,
- Controls "Location" references for the browser
- Sends HTTP requests for dynamically loaded data

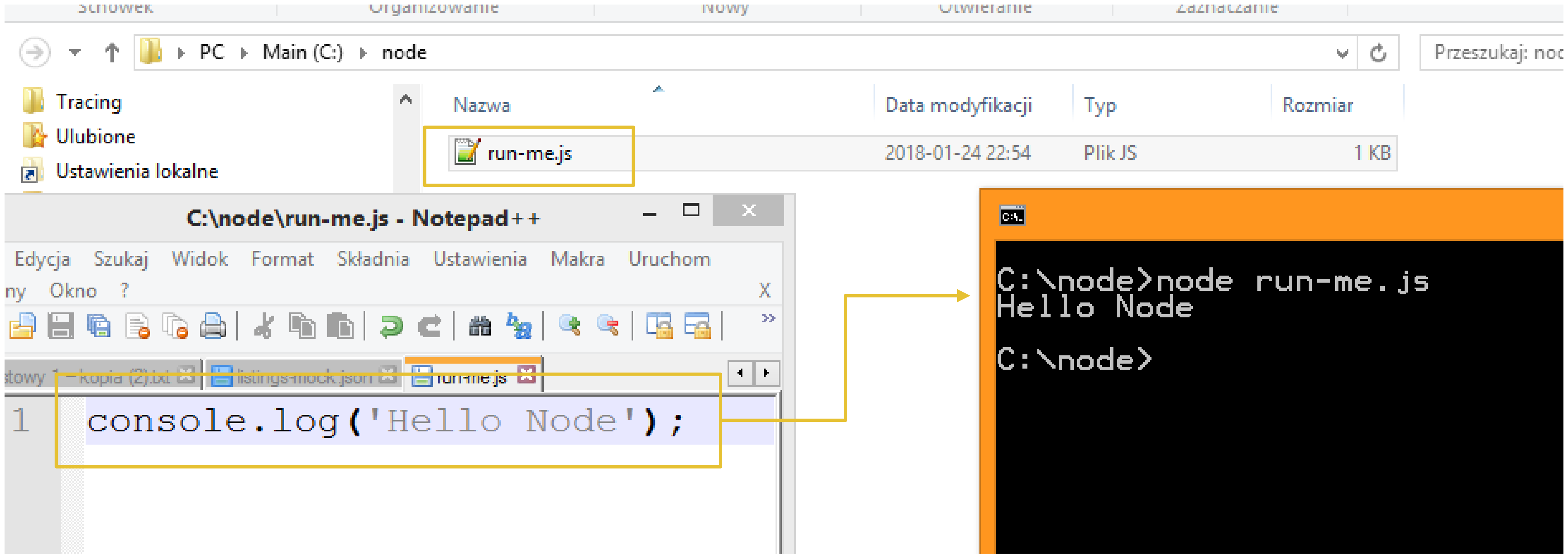
And for this very reason…

- ...we have an "Explosion of JavaScript frameworks"

Environment – Node.js

- **GAME – CHANGER !**
- Idea: commonJS + development of JavaScript modularization
- JavaScript runtime | Server Side JavaScript

https://nodejs.org/en/



Nazwa	Data modyfikacji	Typ	Rozmiar
run-me.js	2018-01-24 22:54	Plik JS	1 KB

```
console.log('Hello Node');
```

```
C:\node>node run-me.js
Hello Node

C:\node>
```

Node Package Manager

- default package manager for Node.js

  https://www.npmjs.com/

- You can install "dependencies" | JavaScript libraries
  - Examples:
  - webpack, reactjs, jquery, bootstrap….
  - example call:

```
> npm install webpack -D
```

What is Transpilation?

- OK, we are using great new products from 2015, the language is "refreshed" and there are new things.
- What about backward compatibility?
- What if I like to use IE11?

- That's right, we need an additional step: Transpilation. Maintaining the level of abstraction of the JS language, but in syntax e.g. from before 2015.

What are Polyfills?

Transpilation alone is sometimes not enough...

- JavaScript is developing not only syntax but also the native API itself

- Thanks to this, for example, we have new functions for the native Array (e.g. findIndex()) or String (.padStart() , .padEnd())

- In order for these elements to be "built-in" in browsers where they do not exist, we need the so-called POLYFILLS

- A popular library that offers them is **core-js**

  https://www.npmjs.com/package/core-js

Polyfill example

```
// https://github.com/uxitten/polyfill/blob/master/string.polyfill.js
// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart
if (!String.prototype.padStart) {
    String.prototype.padStart = function padStart(targetLength, padString) {
        targetLength = targetLength >> 0; //truncate if number, or convert non-number to 0;
        padString = String(typeof padString !== 'undefined' ? padString : ' ,);
        if (this.length >= targetLength) {
            return String(this);
        } else {
            targetLength = targetLength - this.length;
            if (targetLength > padString.length) {
                padString += padString.repeat(targetLength / padString.length);
                //append to original to ensure we are longer than needed
            }
            return padString.slice(0, targetLength) + String(this);
        }
    };
}
```

We check whether there is a .padStart method in the String prototype - if not, we add it.

JavaScript development summary:

2015
ES6

2016 and next years…
ES7, …. ES-next

New keywords, syntactic sugar

Needed:
`Compiler / Transpiler`

e.g.
Babel.io, TypeScript

Development of an existing API (e.g. a string instance gets a .padStart() method)

Needed:
`polyfills` (as code)
(code must be client-side delivered)

e.g.
core-js

What is bunling ?

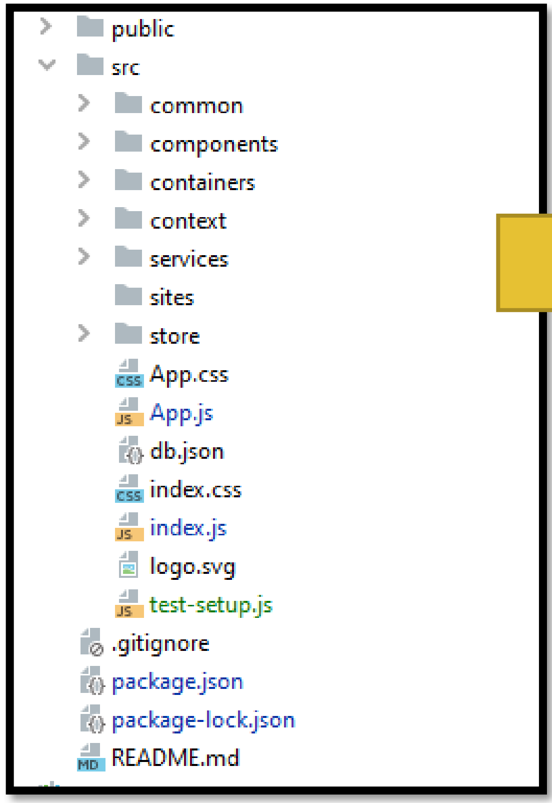
- Ok, using Node.js + ES6+ solves several issues and makes writing code and maintaining it enjoyable

- However, we are still on the "Server Side"! There is no DOM here, there is no BOM, there is no "document", there is no "browser" - what next?

- The bundling process comes to the rescue. That is, using an additional tool to "collect all our CommonJS modules into one .js file and weave it into the HTML page"

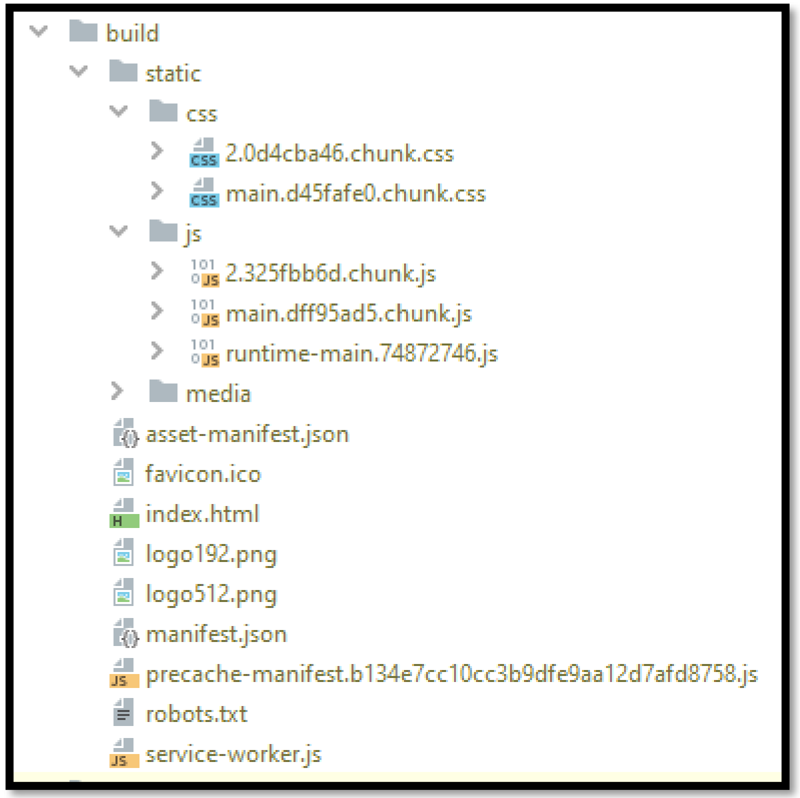
Webpack – the bundler example

DEVELOPMENT



```
> public
v src
    > common
    > components
    > containers
    > context
    > services
      sites
    > store
      App.css
      App.js
      db.json
      index.css
      index.js
      logo.svg
      test-setup.js
  .gitignore
  package.json
  package-lock.json
  README.md
```

PRODUCTION



```
v build
    v static
        v css
            > 2.0d4cba46.chunk.css
            > main.d45fafe0.chunk.css
        v js
            > 2.325fbb6d.chunk.js
            > main.dff95ad5.chunk.js
            > runtime-main.74872746.js
        > media
    asset-manifest.json
    favicon.ico
    index.html
    logo192.png
    logo512.png
    manifest.json
    precache-manifest.b134e7cc10cc3b9dfe9aa12d7afd8758.js
    robots.txt
    service-worker.js
```

https://webpack.js.org

Novadays – faster bundler tools

- **Vite**: https://vitejs.dev/
- **Turbopack**: https://turbo.build/pack/docs

Summary

- What knowledge will be useful when building a SPA application with React?

React - starter

Front-end problems

- **Asynchronous actions**
- **State change**
- **Complex business logic**

Front-end problems

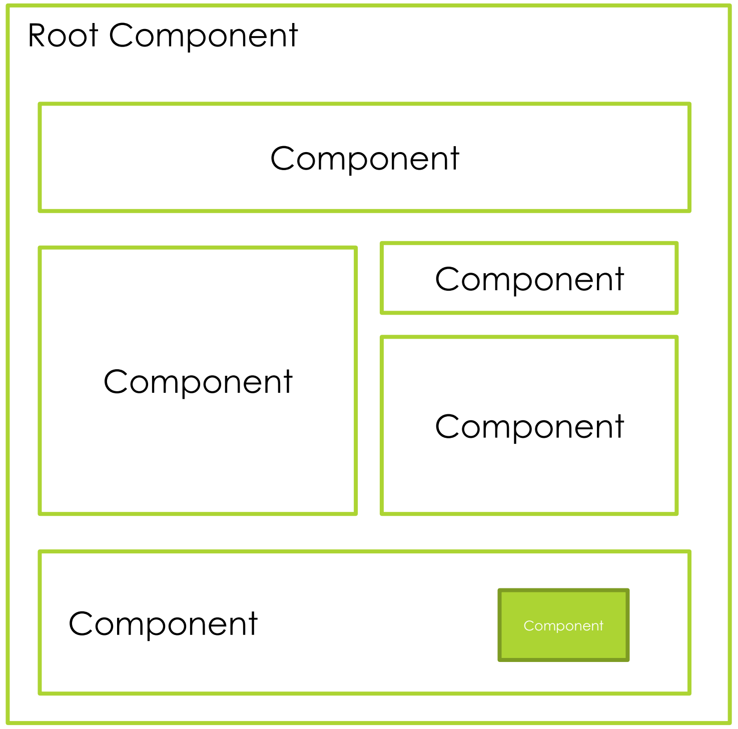
and how to deal with tchem ?

- **Asynchronous actions**
  - limiting callbacks (callback hell)
  - using Promises and streams
- **State change**
  - no data mutations
  - functional programming
- **Complex business logic**
  - division into smaller parts
  - "dumbing" components
  - dependency extraction

React concept

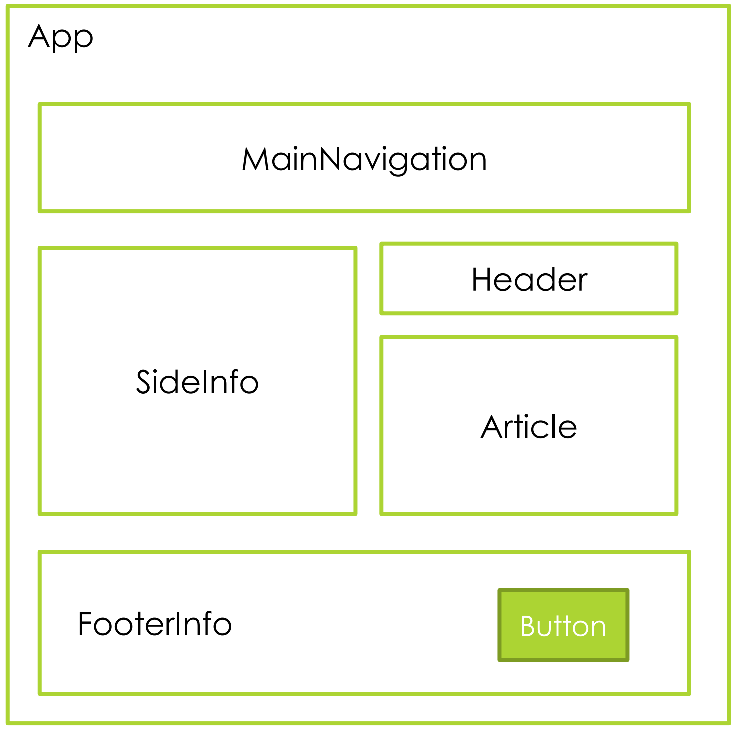
- We build a website from components.
- The component will be the basic unit that can separate a given functionality of the website
- Components will be subject to composition

  (one may be embedded in the other)
- They can communicate by passing each other the so-called "props"
- The structure of the interface looks quite natural, just as if we were using HTML DOM



Where is my Separation Of Concerns ?

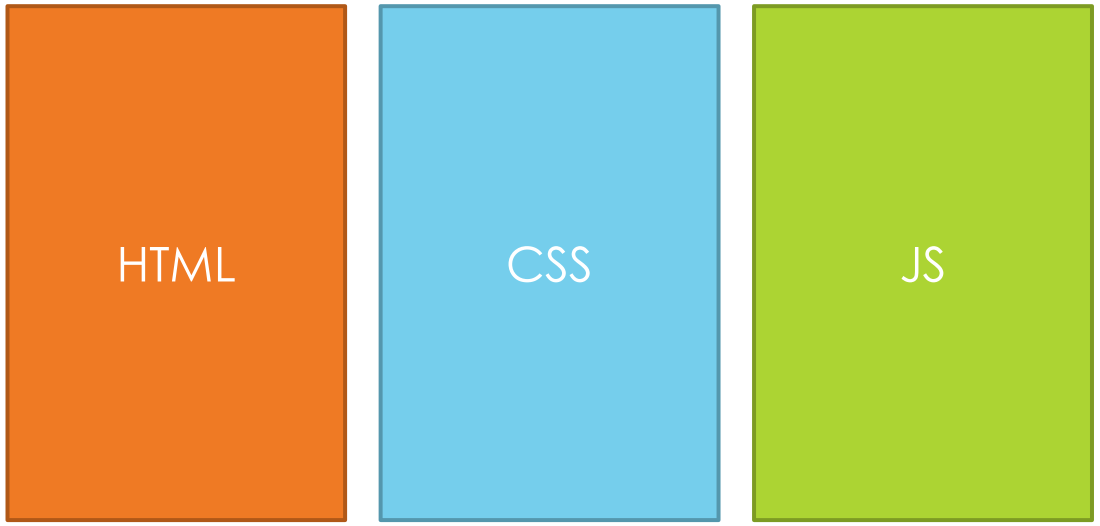
- By separating files by type: JavaScript, CSS and HTML,
  - In fact, you are giving yourself the false impression of SOC
  - This is segregation based on file types, not their functionality or tasks
  - In fact, each element of your UI will have a logic of operation + view + possibly styling
  - In this way, the separation of the "issue" and its "separation" will be the boundaries of the component
- Let's see it another way.... ->



Separation of issues – in practice

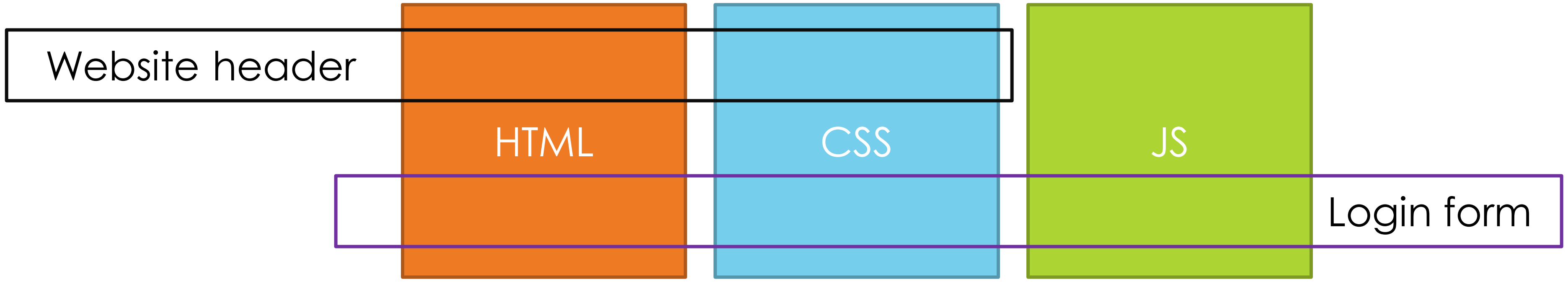
- So far, when writing any web code, we divided our issues into data types: CSS / HTML / JS

- It was unthinkable to mix HTML with JS or JS with CSS etc.
- The division was as follows:



Are you sure this is a good division?

- It quickly turned out that writing complex applications in this way was difficult to maintain
- HTML, JS and CSS files were used to create multi-thousand-word code with lots of dependencies
  - This was terribly hard to debug
- Well, how do I know whether the current error on the website in a given part is a problem in JS, or maybe CSS or HTML?
- The general division is good, but it should be noted that we can distinguish smaller pieces: "components" based on all several layers



Component structure breaks the rules...!?

- React notices these - that's why it doesn't bother it that HTML + JS + CSS live in one file...
- … because the goal of the tasks they will carry out will concern this particular component (common issue)

- The concept is easy to get used to because:
- HTML (actually JSX) will only be rendered by one function

- CSS can still be included globally in the project and used by all components. However, we can separate style sheets for a specific component in the same file (a form of encapsulation - css modules)

React takes care of V in MV*

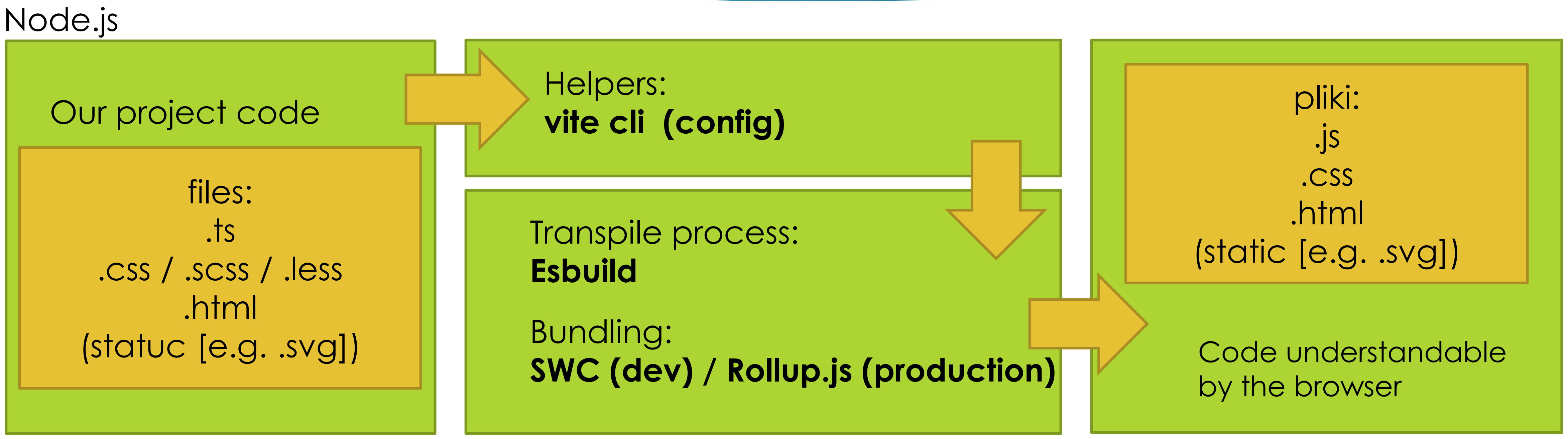
- React can take care of the View and the interactions associated with it.
- However, it cannot execute, for example, an Ajax query
- It can outline what our application "state" looks like - but it does not store a data model "as such" (although it can help us provide correct data to the component using PropTypes)
- It does not have a built-in routing mechanism

- We will solve all these useful features using other libraries whose use in building React applications is quite common

How we will build our application – summary.

- Our application is a SPA - that's why we ultimately want to build the following files from it:
  - **html**, **js**, **css** + possibly static assets (font, **svg**, **jpg**, etc...)
- The development environment will be Node.js
- We will scaffold app with Vite: https://vitejs.dev/
- This way, preparing the application will only involve writing React code (components) and installing the dependencies needed for REST and application state management.

Technical:


Node.js
Our project code
files:
.ts
.css / .scss / .less
.html
(statuc [e.g. .svg])
Helpers:
vite cli (config)
Transpile process:
Esbuild
Bundling:
SWC (dev) / Rollup.js (production)
pliki:
.js
.css
.html
(static [e.g. .svg])
Code understandable
by the browser