



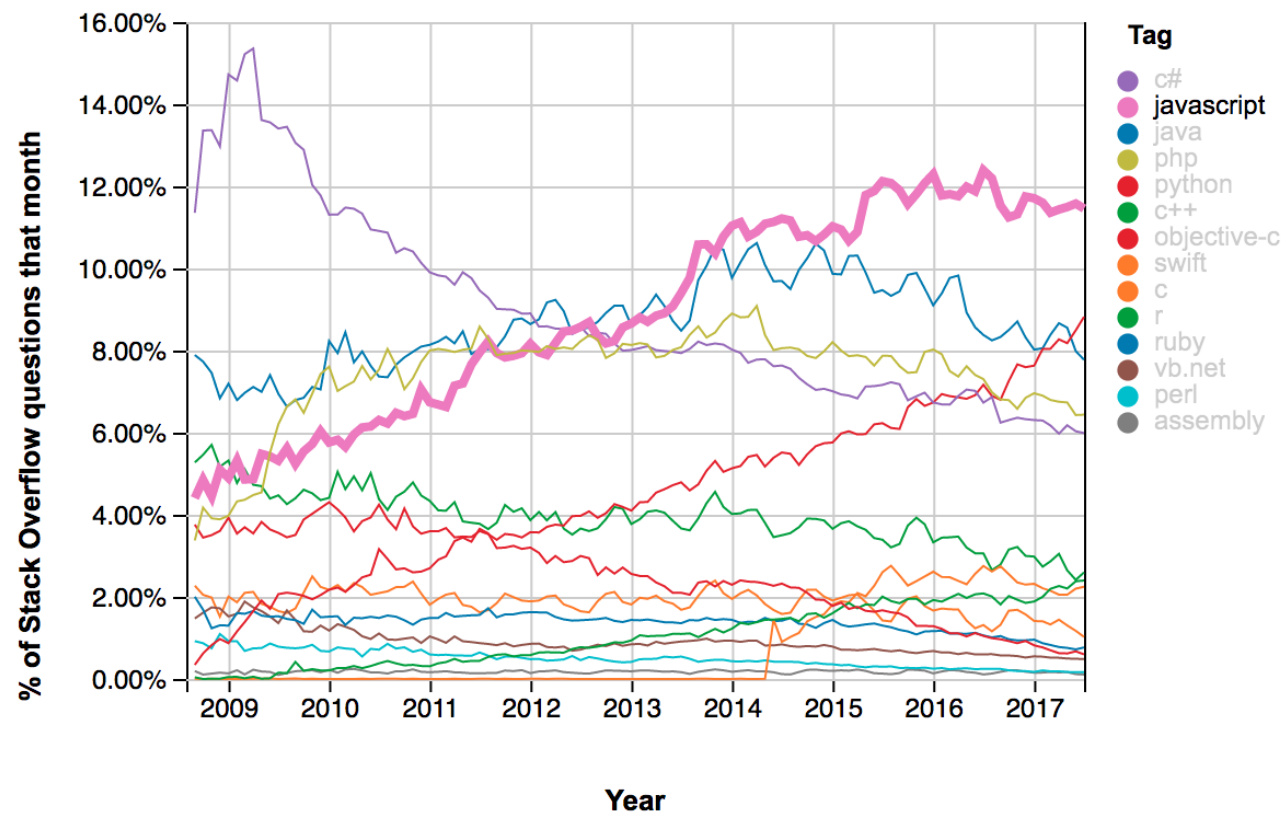
# Code.Hub

The first Hub for Developers  
Ztoupis Konstantinos

React pilot

Code.Learn Program:  
**React**

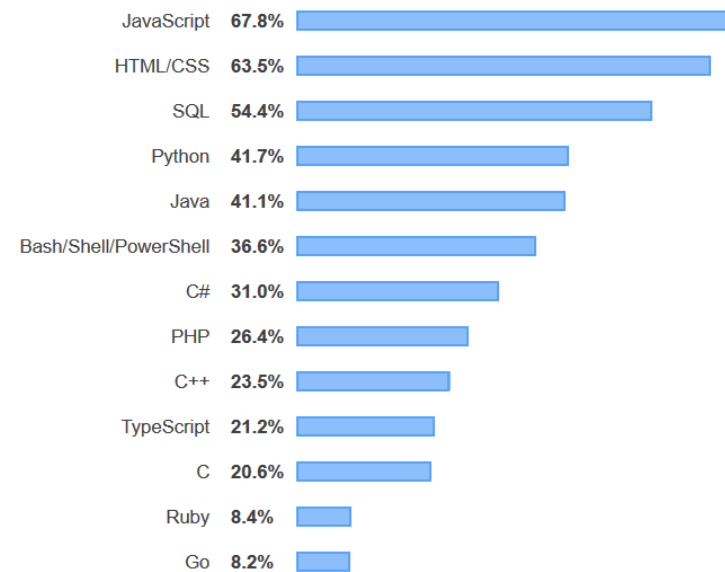
# JavaScript



## Programming, Scripting, and Markup Languages

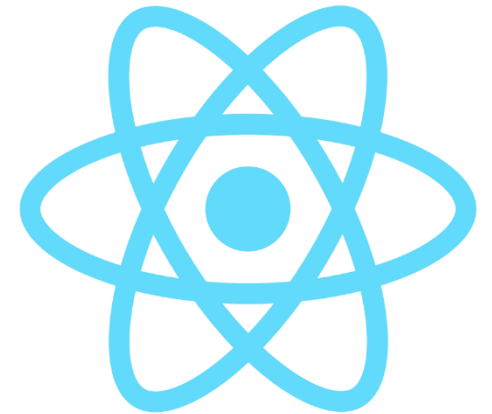
All Respondents

Professional Developers



# React intro

- what is React?
- why is it so popular?
- is it simple to learn?
- is it simple to setup?



# What is React?

- react is a JavaScript library
- developed at Facebook
- released to the world in 2013 (Facebook, Instagram, other applications)
- dividing the UI into a collection of components

# Why is React so popular?

- Less complex than the other alternatives
- Perfect timing
- Backed by Facebook

# Is React simple to learn?

- react is simpler than alternative frameworks
- be integrated Redux, GraphQL and other technologies
- react has a very small API
- 4 concepts
  - Components
  - JSX
  - State
  - Props

# Is React simple to setup?

- directly in the web page
- create-react-app
- CodeSandbox
- custom

# Directly in the web page

```
<body>
...
<script
  src="....libs/react/16.7.0/react.js">
</script>
<script
  src="...libs/react-dom/16.7.0/react-dom.js">
</script>
</body>
```



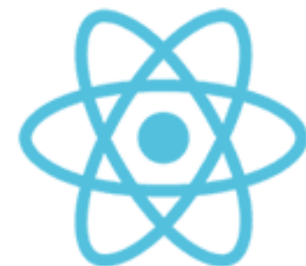
# Directly in the web page

for JSX you need  
Babel

```
<script src="https://unpkg.com/babel-
```

# Create-react-app - Philosophy

- **One Dependency:** There is just one build dependency.
- **No Configuration Required:** You don't need to configure anything.
- **No Lock-In:** You can “eject” to a custom setup at any time.



Create-React-App

# Create-react-app

- React, JSX, ES6, and Flow syntax support
- language extras beyond ES6 like the object spread operator
- autoprefixed CSS, so you don't need -webkit- or other prefixes
- a fast interactive unit test runner with built-in support for coverage reporting

# Create-react-app

- a live development server that warns about common mistakes
- a build script to bundle JS, CSS, and images for production, with hashes and sourcemaps
- an offline-first service worker and a web app manifest, meeting all the Progressive Web App criteria
- hassle-free updates for the above tools with a single dependency

# Create-react-app-What's Included?

- TypeScript: `create-react-app-typescript`.
- parcel instead of Webpack: `create-react-app-parcel`
- React Native: `create-react-native-app`

# Create-react-app - STRUCTURE

package.json

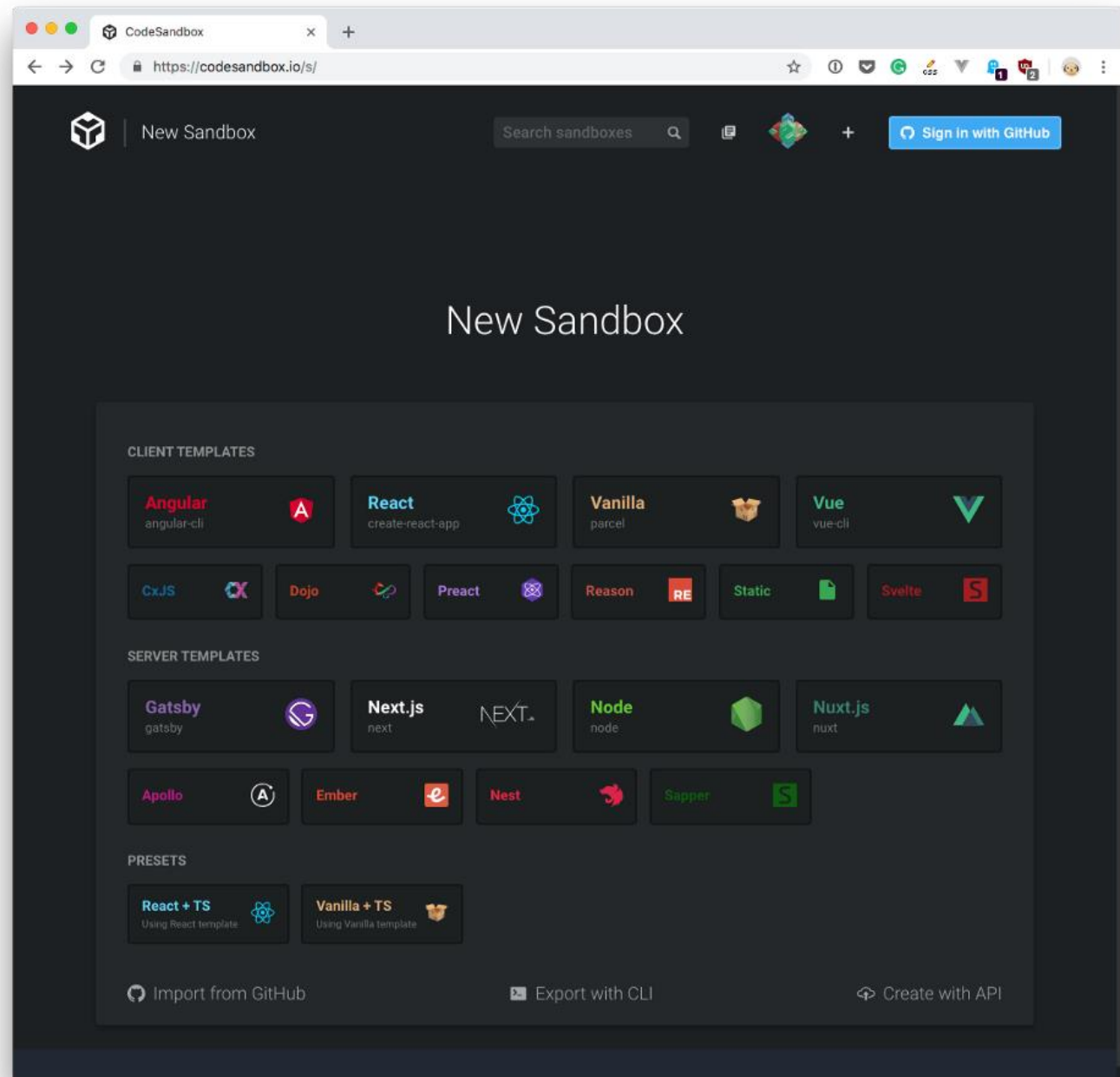
```
package.json
1 {
2   "name": "myfirstreactapp",
3   "version": "0.1.0",
4   "private": true,
5   "dependencies": {
6     "react": "^15.5.4",
7     "react-dom": "^15.5.4"
8   },
9   "devDependencies": {
10    "react-scripts": "1.0.7"
11  },
12  "scripts": {
13    "start": "react-scripts start",
14    "build": "react-scripts build",
15    "test": "react-scripts test --env=jsdom",
16    "eject": "react-scripts eject"
17  }
18 }
```

app structure

```
.gitignore
README.md
node_modules
package-lock.json
package.json
public
src
```

# CodeSandbox

- start a React project
- create-react-app structure
- no installation locally



# ECMAScript

- Defined by **European Computer Manufacturers Association** (ECMA)
- Specification is called **ECMAScript** or ECMA-262
  - JavaScript 5.1 (**ES5**) - <https://www.ecma-international.org/ecma-262/5.1/>
  - JavaScript 6 (**ES6**) - <https://www.ecma-international.org/ecma-262/6.0/>
- **ECMAScript Technical Committee** is called **TC39**
- TC39 has bi-monthly face-to-face meetings
- Besides defining the standard,
  - “TC39 members create and test implementations of the candidate specification to verify its correctness and the feasibility of creating interoperable implementations.”
- **Current members** include
  - Brendan Eich (Mozilla, JavaScript inventor), Allen Wirfs-Brock (Mozilla), Dave Herman (Mozilla), Brandon Benvie (Mozilla), Mark Miller (Google), Alex Russell (Google, Dojo Toolkit), Erik Arvidsson (Google, Traceur), Domenic Denicola (Google), Luke Hoban (Microsoft), Yehuda Katz (Tilde Inc., Ember.js), Rick Waldron (Boucoup, jQuery), and many more



# ES5 vs. ES6

- ECMAScript 5 did not add any new syntax
- ECMAScript 6 does!
- ES6 is backward compatible with ES5, which is backward compatible with ES3
- Many ES6 features provide
- **syntactic sugar** for more concise code
- Spec sizes
  - **ES5 - 258 pages**
  - **ES6 - 652 pages**
- One goal of ES6 and beyond is to make JavaScript a **better target for compiling to from other languages**

# One JavaScript

- Approach named by David Herman
- Allows JavaScript to evolve without versioning
  - avoids migration issues like Python 2 to Python 3
- “Don’t break the web!”
  - removing features would cause existing web apps to stop working
  - can add new, better features
  - ES5 strict mode was a bit of a mistake since it broke some existing code
    - this is why ES6 supports “sloppy mode” code outside modules and class definitions
- Use linting tools to detect use of “deprecated” features
  - ex. switching from **var** to **let** and **const** and using rest parameters in place of **arguments** object

# Transpilers

- Compilers translate code one language to another
  - ex. Java to bytecode
- Transpilers translate code to the same language
- There are several transpilers that translate ES6 code to ES5

# Use ES6 Today?

checking the current feature-wise support for all engines

		Compilers/polyfills							Desktop browsers																	Servers/runtimes										Mobile																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
		56%	71%	71%	72%	50%	69%	17%	5%	11%	96%	96%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%

<https://kangax.github.io/compat-table/es6/>

# ES6 Features

- The following slides describe most of the features in ES6 Also see Luke Hoban's (TC39 member) summary
- <https://github.com/lukehoban/es6features>

# Javascript core concepts

- Variables
- Arrow functions
- Rest and spread
- Object and array destructuring
- Template literals
- Classes
- Callbacks
- Promises
- ES Modules

# Variables

## let

- a new feature introduced in ES2015
- is essentially a block scoped version of var
- its scope is limited to the block, statement or expression where it's defined, and all the contained inner blocks
- completely discard the use of var
- defining let outside of any function - contrary to var - does not create a global variable

# Variables

`const`

- its value can't be changed
- does not provide immutability, just makes sure that the reference can't be changed
- has block scope, same as `let`
- use `const` for variables that don't need to be reassigned later in the app



# Arrow functions

```
const car = function() {...}
```



```
const car = () => {...}
```

omit the brackets and write on a single line

```
const car = () => start()
```

pass parameters

```
const car = (type, year) => start(type, year)  
const car = type => start (type)
```

# Default parameters - Parameters not specified

## Old Way

Unspecified parameters are set to undefined.

You need to explicitly set them if you want a

```
function myFunc(a,b) {  
  a = a || 1;  
  b = b || "Hello";  
}
```

## New Way

Can explicitly define default values if parameter is not defined.

```
function myFunc (a = 1, b =  
"Hello") {
```

# Rest and spread

expand an array, an object or a string using the spread operator ...

```
const array = [1, 2, 3];  
const newArray = [...array];  
const oldObject = {a:1, b:2}  
const newObject = { ...oldObject };  
const string = 'string';  
const arrayString = [...string] // ['s', 't', 'r', 'i', 'n', 'g'];
```

# Rest and spread

useful for  
functions

```
const func = (a1, a2, a3) => {}  
const a = [1, 2, 3];  
func(...a)
```

# Rest and spread

rest  
element

```
const array = [1, 2, 3, 4];  
const [a, b, ...c] = array;  
  
const { a, b, ...c } = {a: 1, b: 2, c: 3, d: 4};
```

# Rest parameters ...

## Old Way

Parameters not listed but passed can be accessed using the arguments array.

```
function myFunc() {  
  var a = arguments[0];  
  var b = arguments[1];  
  var c = arguments[2];  
  arguments[N]  
  //  
}
```

## New Way

Additional parameters can be placed into a named array.

```
function myFunc  
(a,b,...theArgsArray) {  
  var c = theArgsArray[0];  
}
```

# Spread operator ...

## Old Way

Expand an array to pass its values to a function or insert it into an array.

```
var anArray = [1,2,3];  
myFunc.apply(null, anArray);  
var o = [5].concat(anArray).concat([6]);
```

## New Way

Works on iterable types: strings & arrays

```
var anArray = [1,2,3];  
myFunc(...anArray);  
var o = [5, ...anArray, 6];
```

# Object and array destructuring

```
const object = {a: 1, b: 2, c: 3, d: 4};  
const { a: t, b } = object;
```

```
const array = [1, 2, 3, 4];  
const [ a, b ] = array;
```



# Destructuring assignment

## Old Way

Expand an array to pass its values to a function or insert it into an array.

```
var a = arr[0];  
var b = arr[1];  
var c = arr[2];  
  
var name = obj.name;  
var age = obj.age;  
var salary = obj.salary;  
  
function render(props) {  
  var name = props.name;  
  var age = props.age;  
}
```

## New Way

Works on iterable types: strings & arrays


```
const [a,b,c] = arr;  
const {name, age, salary} = obj;  
function render({name, age}) {  
}
```

# Template literals

use backticks instead of single or double quotes

```
const aString = `a string`;
```

```
const aString = 'first line\n' + 'second line';
```



```
const aString = `first line  
second line`;
```

# Interpolation

interpolate variables and expressions into strings

```
${...}
```

```
const a = 'a';  
const string = `get ${a}`  
  
const string = `get ${2*3}`  
const string2 = `get ${x===1 ? 'a' : 'b'}`
```

# Template string literals

## Old Way

Use string concatenation to build up string from variables.

```
function formatGreetings(name, age) {  
  var str = "Hi " + name +  
    " your age is " + age;  
  ...  
}
```

## New Way

Very useful in frontend code.  
Strings can be delimited by `" "`, `' '`,  
or `` ``

```
function formatGreetings(name,  
age) {  
  var str = `Hi ${name} your age is  
    ${age}`;  
}
```

Also allows multi-line strings:

```
`This string has
```

# For of

## Old Way

Iterator over an array

```
var a = [5,6,7];  
var sum = 0;  
for (var i = 0; i < a.length; i++) {  
    sum += a[i];  
}
```

## New Way

Iterate over arrays, strings, Map, Set,  
without using indexes.

```
let sum = 0;  
for (ent of a) {  
    sum += ent;  
}
```

# Classes

```
class Car {  
  constructor(type) {  
    this.type = type;  
  }  
  
  hi() {  
    return `My type is ${this.type}`;  
  }  
}
```

# Classes

- a class has an identifier, create new objects using new ClassIdentifier()
- when the object is initialized, the constructor method is called
- a class also has as many methods as it needs

# Class inheritance

- can extend another class
- inherit all the methods

```
class Jeep extends Car {  
  hi() {  
    return `${super.hi()} and I am a Jeep`;  
  }  
}
```

```
const a = new Car('Big car');  
a.hi();
```



# Static methods

- methods are defined on the instance
- static methods are executed on the class

```
class Car {  
    static start () {  
        return 'start engine';  
    }  
}
```

```
Car.start();
```

# Private methods

no private or protected  
methods

# Getters and setters

- accessing the variable
- modifying the value

```
class Car {  
  constructor(type) {  
    this.type = type;  
  }  
  set type(value) {  
    this.type = value;  
  }  
  get type () {  
    return this.type;  
  }  
}
```

# Promises

- starts in pending state
- waits for it to either return the promise in a resolved state, or in a rejected state
- are used by standard modern Web APIs like Fetch or Service Workers

```
new Promise( /* executor */ function(resolve, reject) { ... } )
```

# ES Modules

a module is a JavaScript file that exports one or more values (objects, functions or variables), using the export keyword

```
import stringUpperCase from 'module';
```

```
export const stringUpperCase = (string) => string.toUpperCase();
```

# ES Modules

valid import syntax

```
import { time } from './time.js'  
import { random } from '../../randomNumber.js'
```

invalid import syntax

```
import { time } from 'time.js'  
import { random } from 'randomNumber.js'
```

# React concepts

- Single Page Applications
- Declarative
- Immutability
- Purity
- Composition
- The Virtual DOM
- Unidirectional Data Flow

# Single Page Applications

before...

- less capable browsers
- poor javascript performance
- pages from a server
- at event, a new request to server and the browser subsequently loaded the new page



# Single Page Applications

now...

- many modern javascript frameworks
- more capable browsers
- high javascript performance
- load the application code once
- at event, a request on the server and a part of the app is updated

# Single Page Applications

- Facebook
- Gmail
- Airbnb
- Asana
- Atlassian
- Cloudflare
- Dropbox
- NY Times
- BBC
- Instagram
- Netflix
- Podio
- Uber
- WhatsApp

# Single Page Applications

## pros

- much faster to the user
- less resources for the server
- build a mobile app with existing server-side code
- easy transformation into Progressive Web Apps
- better focus in working (backend - frontend)

# Single Page Applications

## cons

- scroll position
- search engine ranking
- analytics
- memory leaks

# JavaScript built in methods

- map
- reduce
- filter
- find

# Declarative

- build Web interfaces without even touching the DOM directly
- event system without interact with the actual DOM Events
- The opposite of declarative is **imperative**
  - looking up elements in the DOM using jQuery or DOM events
  - you tell the browser exactly what to do, instead of telling it what you need

# Declarative

```
$("#btn").click(function(){
  $(this).toggleClass("active");
  if( $(this).text() === "Active" ) {
    $(this).text("Inactive")
  } else {
    $(this).text("Active")
  }
});
```

Iterative  
way

Declarative way

```
this.setState({
  isActive: !this.state.isActive
});
```

```
<Button onclick="this.handleClick"
```

# Immutability

immutable: value cannot change after it's created, to update its value, you create a new value  
state → setState()

- mutations can be centralized
- cleaner and simpler code
- code is optimized by library



# Purity

- does not mutate objects
- returns a new object
- no side effects
- same output when called with the same input

# Composition

small and lean components and use them to *compose* more functionality on top of them

- specialized version of component
- pass methods as props
- using children
- higher order components

# Composition

composition allows you to build more complex functionality by combining small and focused functions.

Like `map()` to create a new array from an initial set, and then filtering the result using `filter`

```
const list = ['Apple', 'Orange', 'Egg']  
list.map(item => item[0]).filter(item => item === 'A') //'A'
```

# Composition

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

```
<Widget>  
  <SearchForm />  
  <Results>  
    <Header />  
    <SportsTable />  
    <ElectronicsTable />  
  </Results>  
</Widget>
```

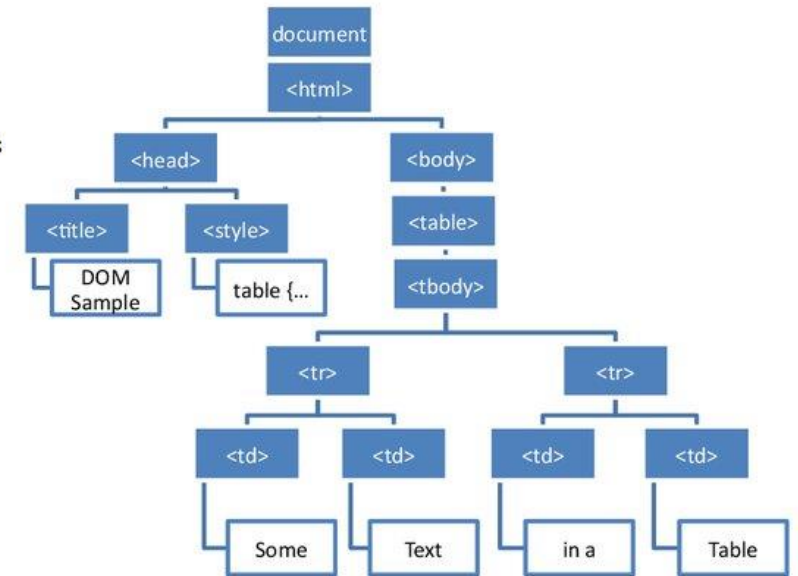
# The DOM

The browser builds the DOM by parsing the code you write, it does this before it renders the page

## DOM Tree



```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Sample</title>
    <style type="text/css">
      table {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <table>
      <tbody>
        <tr>
          <td>Some</td>
          <td>Text</td>
        </tr>
        <tr>
          <td>in a</td>
          <td>Table</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```



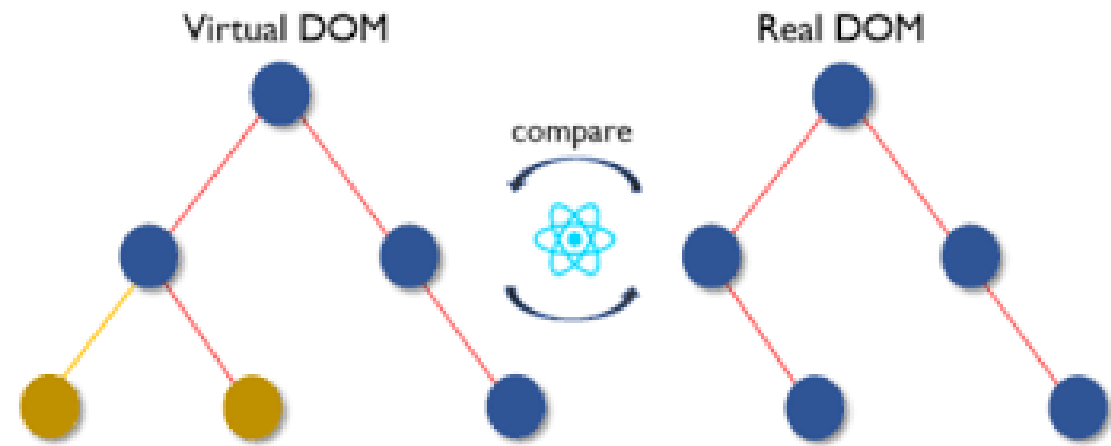
# The Issue

Most modern web pages have huge DOM structures and a simple change would cost too much, resulting in slower loading pages



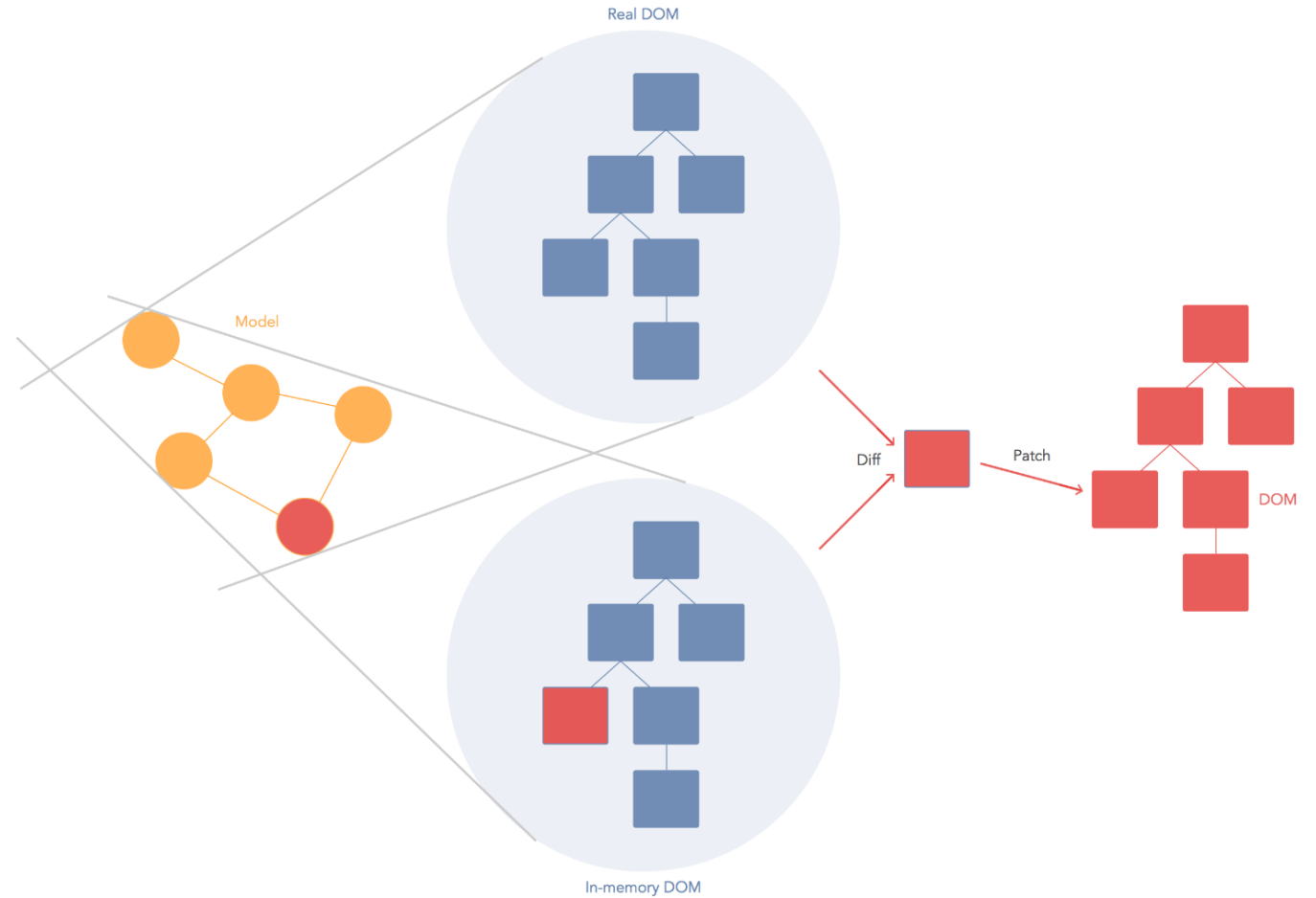
# The Virtual DOM

- a copy of the HTML
- an abstraction of the HTML DOM



# Reconciliation

The process through which  
React updates the DOM





# Unidirectional Data Flow

data has only one way to be transferred to other parts of the application

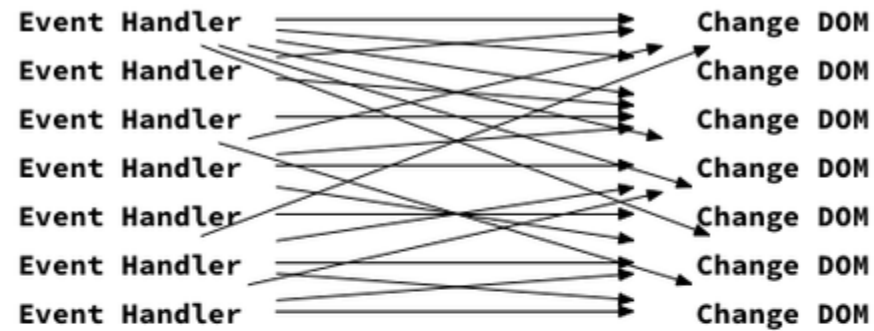
- state is passed to the view and to child components
- actions are triggered by the view
- actions can update the state
- the state change is passed to the view and to child components

# Unidirectional Data Flow

- less error prone, as you have more control over your data
- easier to debug, as you know *what* is coming from *where*
- more efficient, as the library already knows what the boundaries are of each part of the system

# Unidirectional Data Flow

## jQuery Style



## React.js Style

