



| The European Synchrotron

Introduction to web development

Part 1 - Introduction to JS

Part 2 - Introduction to React





Javascript is standardised by the ECMAScript standard <https://tc39.es/ecma262/>

First created by Brendan Eich at Netscape in 1995

Is dynamically and weakly typed language with prototype based object orientation

The runtime is single threaded

Two simple examples

```
class Rectangle {
  constructor (height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area () {
    return this.calcArea ();
  }
  // Method
  calcArea () {
    return this.height * this.width;
  }
}

const square = new Rectangle (10, 10);

console.log (square.area); // 100
```

```
function factorial (n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial (n - 1);
  }
}
```

How does this code run ?

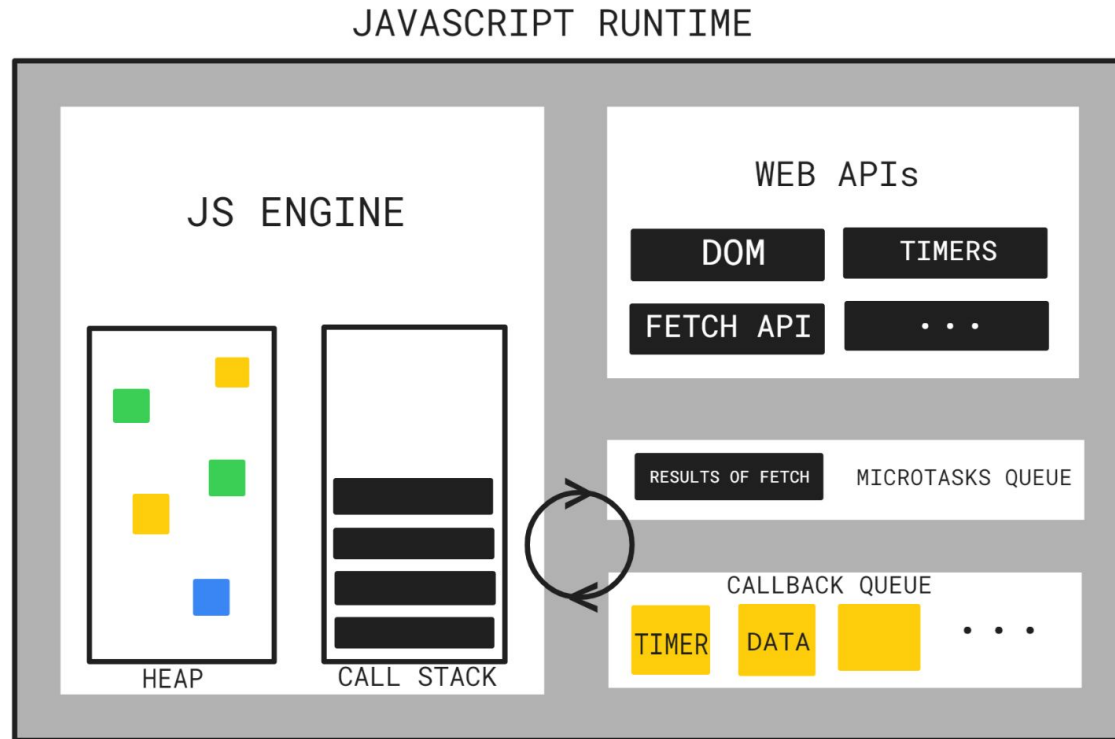
Let's have a look at the runtime environment !

The screenshot shows the MxCuBE-Web interface in a browser window. The top navigation bar includes 'Samples', 'Data collection', 'Equipment', and 'System log'. The main control panel displays various parameters: Energy (12.4000 keV), Resolution (0.740 Å), Transmission (10.0%), Cryo (200.00 k), Wavelength (1.00 Å), Detector (10.1 mm), and Flux (2.30e+12 ph/s). Below these are status indicators for Detector (READY), Sample Changer (READY), Fast Shutter (CLOSED), Safety shutter (CLOSED), and Ring Current (193.5 mA). The central part of the interface features a live video feed of a sample, with a 50 µm scale bar in the bottom left. To the left of the video are controls for Phase Control (Centring), Beam size (10), Omega (311.10), Kappa (11.0), and Kappa Phi (22.0). A toolbar above the video includes icons for Snapshot, Draw grid, 3-click centring, Focus, Zoom (2x/VEL.1), Backlight, Frontlight, and Video size. On the right side, there is a 'JAVASCRIPT RUNTIME' diagram. This diagram illustrates the interaction between the JS ENGINE (containing HEAP and CALL STACK) and WEB APIs (including DOM, TIMERS, and FETCH API). It also shows the flow of data through a RESULTS OF FETCH and MICROTASKS QUEUE, and a CALLBACK QUEUE containing a TIMER and DATA. A 'Log messages' section is visible at the bottom of the runtime diagram.

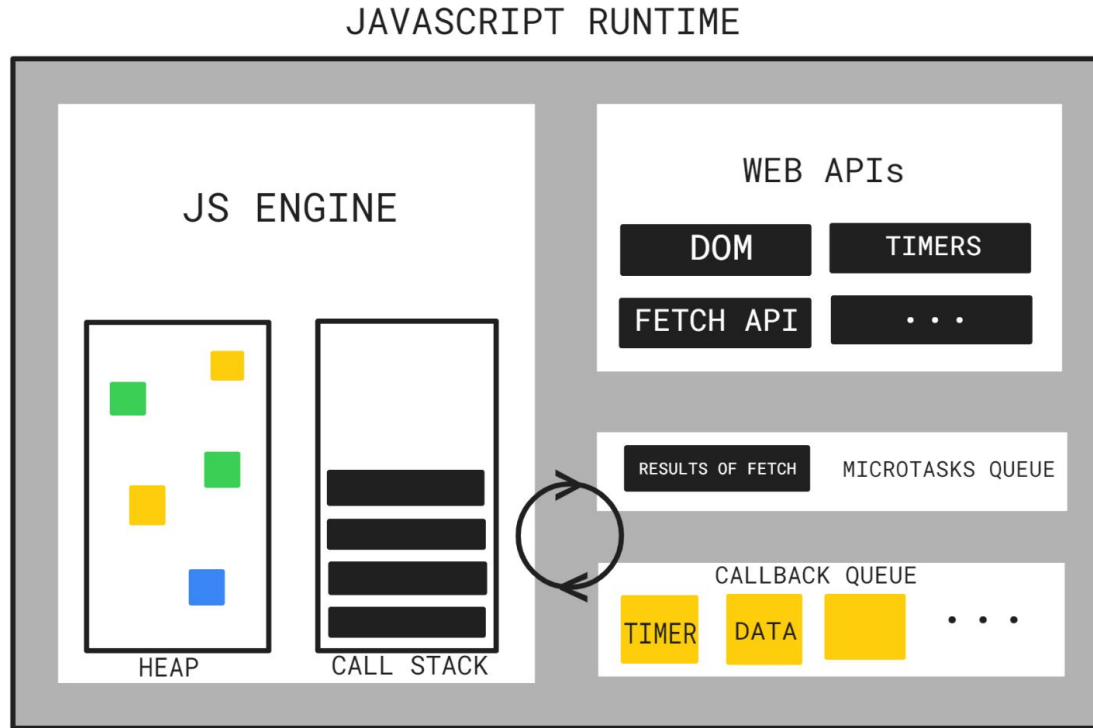
Web API's, provided by Browser, standardised by W3C



Well documented at:
<https://developer.mozilla.org/en-US/docs/Web/API>



- Fairly simple architecture, heap, call stack and two queues
- For a real deep dive: <https://github.com/v8/v8>



- Each engine runs in a single thread and has one eventloop (one per origin)
- The browser decides internally which source to pick events from, user input, requests and so on
- Callback queue is processed when call stack is empty, new tasks executes at next iteration
- Microtask queue is processed between tasks (but new tasks are executed immediately)

Javascript explained (simply)

Developer Tools — MXCuBE — http://localhost:3000/

Sources Outline Search

MotorInput.jsx X index.js queue.js serverIO.js PassControlDialog.jsx general.js SampleViewContainer.js jsmpeg.min.js

```
53     this.props.decimalPoints,
54   });
55 }
56 }
57 /* eslint-enable react/no-set-state */
58
59 stepChange(name, operator) {
60   const { value, step } = this.props;
61   const newValue = value + step * operator;
62
63   this.motorValue.value = this.props.value.toFixed(this.props.decimalPoints);
64   this.motorValue.defaultValue = newValue;
65   this.props.save(name, newValue);
66 }
67
68 stopMotor(name) {
69   this.props.stop(name);
70 }
71
72 render() {
73   debugger;
74   const { value, motorName, step, suffix, decimalPoints } = this.props;
75   const valueCropped = value.toFixed(decimalPoints);
76   const inputCSS = cx('form-control rw-input', {
77     'input-bg-edited': this.state.edited,
78     'input-bg-moving':
79       this.props.state === MOTOR_STATE.BUSY ||
80       this.props.state === MOTOR_STATE.MOVING,
81     'input-bg-ready': this.props.state === MOTOR_STATE.READY,
82     'input-bg-fault':
83       this.props.state === MOTOR_STATE.FAULT ||
84       this.props.state === MOTOR_STATE.OFF ||
85       this.props.state === MOTOR_STATE.ALARM ||
86       this.props.state === MOTOR_STATE.OFFLINE ||
87       this.props.state === MOTOR_STATE.UNKNOWN ||
88       this.props.state === MOTOR_STATE.INVALID,
89     'input-bq-onlimit':

```

Paused on debugger statement

Watch expressions +

Breakpoints -

Pause on exceptions

Call stack

render MotorInput.jsx:73

- React 9
- unstable_runWithPriority scheduler.development.js:401
- React 4
- Redux 7
- getInitialState general.js:333
- promise callback
- getInitialState general.js:332

Expand rows

Scopes Map

render

- <this>: {}
- arguments: Arguments
- decimalPoints: (uninitialized)
- inputCSS: (uninitialized)
- motorName: (uninitialized)
- step: (uninitialized)
- suffix: (uninitialized)
- value: (uninitialized)
- valueCropped: (uninitialized)
- BBlock
- jsx

(From bundle.js) (63, 80)

Filter Output

Errors Warnings Logs Info Debug CSS XHR Requests

next state > Object { login: (-), queue: (-), uiproperties: (-), sampleGrid: (-), sampleChanger: (-), sampleChangerMaintenance: (-), taskForm: (-), sampleview: (-), logger: (-), general: (-), - } [redux-logger.js:335](#)

action UPDATE_SHAPES @ 12:18:14.664 [redux-logger.js:313](#)

prev state > Object { login: (-), queue: (-), uiproperties: (-), sampleGrid: (-), sampleChanger: (-), sampleChangerMaintenance: (-), taskForm: (-), sampleview: (-), logger: (-), general: (-), - } [redux-logger.js:323](#)

action > Object { type: "UPDATE_SHAPES", shapes: [] } [redux-logger.js:327](#)

next state > Object { login: (-), queue: (-), uiproperties: (-), sampleGrid: (-), sampleChanger: (-), sampleChangerMaintenance: (-), taskForm: (-), sampleview: (-), logger: (-), general: (-), - } [redux-logger.js:335](#)

Firefox can't establish a connection to the server at ws://localhost:3000/socket.io/7EIO=4&transport=websocket&sid=Zpzq664VmRZCS0cKAAAs. [websocket.js:35](#)

Browser provide good tools for debugging and seeing what's going on



I'm personally using MDN as reference:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>



If you would like to try example while we are speaking, you can try on <https://playcode.io/javascript>

- Weakly typed
- *this* keyword
- Promises
- Prototypal inheritance

Weakly typed

Weakly typed, the interpreter makes sometimes difficult (and perhaps unexpected) decisions regarding types.

This process is referred to as type coercion (implicit type conversion)

For instance:

```
"mxcube" + 48;           // -> "mxcube48"  
"b" + "a" + +"a" + "a"; // -> 'baNaNaN'  
true + true;            // -> 2
```

Luckily the linter will warn us about likely unwanted coercion

Javascript == operator performs coercion while the === (strict equality does not), always use ===

There is very nice project that goes through details like this:

<https://github.com/denysdovhan/wtfjs#-examples>

This keyword

The value of `this` is bound at runtime and depends on how a function is called (not to which object it belongs)

We can override the somewhat odd behaviour of `this` so that it always refers to a class instance using the `bind` method.

You will encounter the `bind` function in the `mxcube` code base.

Let's have a look at an example

This keyword

```
import React from "react";

class SimpleExample extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this); // we bind this to SimpleExample
  }

  handleClick(e) {
    console.log(this); // SimpleExample {props: Object, ...}
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Click here</button>
      </div>
    );
  }
}
```

Use bind

Avoid using this outside of classes

Promises

Promises

Provides an abstraction on low level asynchronous code, based on callbacks

A promise is an object returned by an asynchronous function, like a future or greenlet in Python, it contains the current state of the execution (Pending, Fulfilled, Rejected)

A promise takes two callbacks, a success (resolve) and a failure (rejected) and can be chained with then

```
const fetchPromise = fetch("https://mxcube.esrf.fr/");

fetchPromise
  .then((response) => { // Promise call backs are put in the micro task queue
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json(); // Returns a promise
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error(`Could not get mxcube ${error}`);
  });
```

Promises

Today however we can use `async` and `await` instead (like in Python), when writing new code prefer `async` and `await`

```
const fetchPromise = fetch("https://mxcube.esrf.fr/");

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.error(`Could not get mxcube ${error}`);
  });

async function aFetch() {
  try {
    const result = await fetch("https://mxcube.esrf.fr/"); // The statements after await are put on the microtask queue
  } catch(error) {
    result = "";
    console.log(error);
  }

  console.log(result);
}
```

Prototype based inheritance

You might already have heard about prototype based inheritance, and you will probably encounter the `[[prototype]]` or `__proto__` attributes when debugging Javascript code.

It is indeed a bit awkward if one have never seen it before,

Classes define a predefined structure/taxonomy while prototypes define from which objects to inherit behaviour from via its prototype chain.

An objects prototype can be changed runtime and there is not necessarily well defined taxonomy.

Javascript has Class based inheritance built on top of the prototype based one.

We are using classes in mxcube web code base so you will encounter them.

React is phasing out the class based components favoring what's called functional components (more about that later). **Axel will make a tutorial about this**

Classes are still very useful and we will keep using them for other things

```
export default class InOutSwitch extends React.Component {
  constructor (props) {
    super (props);
    this.setOff = this.setOff.bind(this);
    this.setOn = this.setOn.bind(this);
    this.onLinkRightClick = this.onLinkRightClick.bind(this);
    this.onOptionsRightClick = this.onOptionsRightClick.bind(this);
  }
  setOff () {
  }
  setOn () {
  }
  render () {
  }
}
```

Other often used features

Import/export statement:

```
import { objectOne } from "module-name";
```

It's possible like in Python to perform * imports and that it should be avoided.

Spread operator (...) - yes three dots :

Allows for collections to be expanded like python's * and ** operators

```
const numbers = [1, 2, 3];  
console.log(sum(...numbers)); // -> 6
```

“standard/expected” language constructs for iteration, conditionals and so on.

Oh yes, and the linter does a good job of telling you when you did something you probably didn't mean to ;)



Javascript library for building user interfaces

Created at Meta (Facebook)

Part II - Introduction to React

```
<!DOCTYPE html>
<html itemscope="" itemtype="http://schema.org/WebPage" lang="en-FR">
  <head>
  </head>
  <body jsmode="hspDDf" jsaction="xjhTif:.CLIENT;O2vyse:.CLIENT;IVKtfe:.CLIENT;Ez7VMc:.CLIENT;...ENT;YcfJ:.CLIENT;szjOR:.CLIENT;JL9QDc:.CLIENT;kWlxhc:.CLIENT">
    <style data-impl="1695977902624">
      .L3eUgb{display:flex;flex-direction:column;height:100%;o3j99{flex-shrink:0;box-sizing:border-box}.n1xJcf{height:60px}.LLD4me{min-height:150px;max-height:290px;height:calc(100% - 560px)}.yr19Zb{min-height:92px}.ikrT4e{max-height:160px}.mwhT9d{position:absolute;left:-1000px}.ADHj4e{padding-top:0px;padding-bottom:85px}.oWyZre{width:100%;height:500px;border-width:0}.qarstb{flex-grow:1}
    </style>
    <div class="L3eUgb" data-hveid="1">
      <div class="o3j99 n1xJcf Ne6nSd">
        <div class="o3j99 LLD4me yr19Zb LS80J">
          <style data-impl="1695977902636">
            <div class="k1zIA r5k4se">
              <style data-impl="1695977902636">
                
              </div>
            </div>
          <div class="o3j99 ikrT4e om7nfv">
```

Provides a framework that make it possible to write interfaces in a declarative way (without directly interfacing with the DOM)

Done by writing components (widgets) that express what will be rendered for a certain state

React manages state for components, decides when to render

Part II - Introduction to React

```
JS index.js x
1 import { StrictMode } from "react";
2 import { createRoot } from "react-dom/client";
3
4 import App from "./App";
5
6 const rootElement = document.getElementById("root");
7 const root = createRoot(rootElement);
8
9 root.render(
10   <StrictMode>
11     <App />
12   </StrictMode>
13 );
```

```
1 import "./styles.css";
2
3 export default function App() {
4   return (
5     <div className="App">
6       <h1>Hello from MXCuBE</h1>
7     </div>
8   );
9 }
```

```
<> index.html x
18
19   Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/
20   work correctly both with client-side routing and a non-
21   Learn how to configure a non-root public URL by running
22   -->
23   <title>React App</title>
24 </head>
25
26 <body>
27   <noscript>
28     You need to enable JavaScript to run this app.
29   </noscript>
30   <div id="root"></div>
31 </body>
```


These three parts, all the libraries and resources are built into a bundle via a build chain.

React creates single page interfaces - the DOM is updated instead of changing page

You can try this out by typing `react.new` in your browser

You can use the `create-react-app` to get the tool chain installed locally (<https://create-react-app.dev/docs/getting-started>)

You need `node.js` (Javascript runtime) to run the tool chain (<https://nodejs.org/en>)

```
1 import "./styles.css";
2
3 export default function App() {
4   return (
5     <div className="App">
6      <h1>Hello from MXCuBE</h1>
7     </div>
8   );
9 }
```

This is called JSX code
which the browser does
not understand

The browser only understands Javascript and HTML so we need to build/compile/transpile the code into a Javascript.

In addition to this, Javascript language features are supported to a varying degree across browsers

Additional libraries and the build chain takes care of this for us and creates a bundle that is usable by recent browsers.

```
1 import "./styles.css";
2
3 export default function App() {
4   return (
5     <div className="App">
6     <h1>Hello from MXCuBE</h1>
7     </div>
8   );
9 }
```

A component can be expressed as a function or a Class

We are so far using Classes in mxcube but they are getting phased out

A JSX component gets translated into what's called a React.Element instance and added to a virtual DOM

A software called Babel (part of the tool chain) takes care of this

You can try it out and test how code is “transpiled” at: <https://babeljs.io/repl>

```
export default function App() {  
  return (  
    <div className="App">  
      <h1>Hello from MXCuBE</h1>  
    </div>  
  );  
}
```



```
import { jsx as _jsx } from "react/jsx-runtime";  
export default function App() {  
  return /*#__PURE__*/_jsx("div", {  
    className: "App",  
    children: /*#__PURE__*/_jsx("h1", {  
      children: "Hello from MXCuBE"  
    })  
  });  
}
```

The return of the `_jsx` function is a `React.Element`

All the elements are inserted in what’s called the virtual DOM, a big tree structure

The synchronisation of the actual DOM (seen by the browser) and the virtual DOM is called reconciliation

It’s an expensive operation to update the entire DOM, something called DOM diffing is used to optimise the rendering.

```
1 import './styles.css';
2
3 export default function App() {
4   return (
5     <div className="App">
6       <h1>Hello from MXCuBE</h1>
7     </div>
8   );
9 }
```

Each component has a life cycle from when it gets “mounted” in the virtual DOM until it gets “unmounted” and a state

The component gets re-rendered when the state changes

We can catch life cycle events in what’s called life cycle methods or hooks (for functional components),

Axel and Mikel will mention more about all this in the practical part

Part II - Introduction to React

When we develop we can run the tool chain so that it updates (re builds) on change, with “pnpm start”

This means that we can quickly see the changes we make and we can add debugger statements in the code

The browser “developer tools” are very complete and useful to find out what’s going on.

