

Bài giảng

Phát triển ứng dụng web

Lê Đình Thanh

VNU-UET FIT

Email: thanhld@vnu.edu.vn

Mobile: 0987.257.504

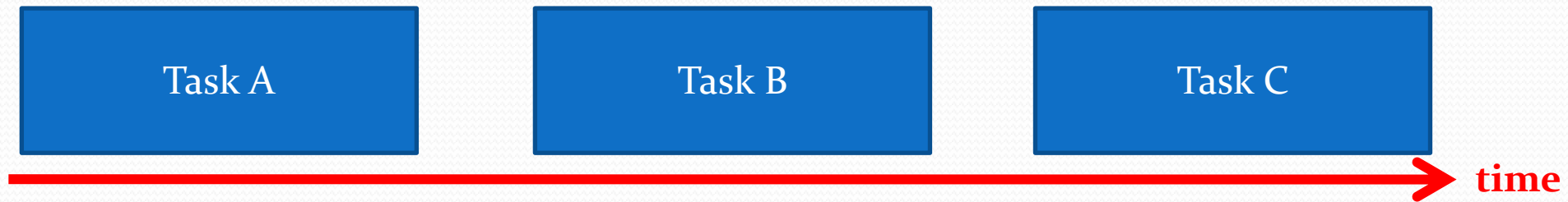
Website: <https://uet.vnu.edu.vn/~thanhld>

JavaScript nâng cao

- Lập trình không đồng bộ
- Vòng lặp sự kiện
- Xử lý song song
- Web API

Lập trình đồng bộ

- Một tác vụ đang xử lý thì phải xử lý xong (kết thúc), tác vụ khác mới có thể chạy được.
 - Mỗi thời điểm chỉ có nhiều nhất một tác vụ được xử lý (đơn nhiệm - singletasking)



- Khi ứng dụng web thực hiện tác vụ mất thời gian và không trả điều khiển cho trình duyệt
 - trình duyệt bị khóa/treo/đóng băng (**blocking/frozen/unresponding**)
 - người dùng không tương tác được

Ví dụ lập trình đồng bộ

```
function sFunction () { return ("Hello S!"); }
```

```
console.log("begin");
```

```
let ret = sFunction();
```

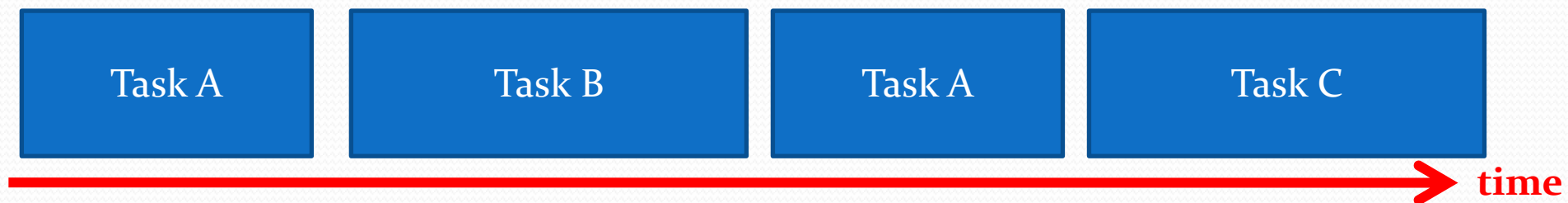
```
console.log(ret);
```

```
console.log("end");
```

```
begin  
Hello S!  
end
```

Lập trình không đồng bộ

- Một tác vụ đang chạy có thể bị hoãn để xử lý một tác vụ khác, sau đó được gọi lại để xử lý tiếp
 - Nhiều tác vụ được xử lý đồng thời (đang được xử lý)
 - Đa nhiệm (multitasking)
 - Lưu ý: Đồng thời (concurrent) khác song song (parallel).



- Phù hợp khi chạy tác vụ mà không muốn chặn (block) tác vụ khác (ví dụ tác vụ chính (trình duyệt))
- Nhiều Web API hiện đại là không đồng bộ, đặc biệt các thao tác I/O, request đến server, giao tiếp mạng.

Ví dụ lập trình không đồng bộ

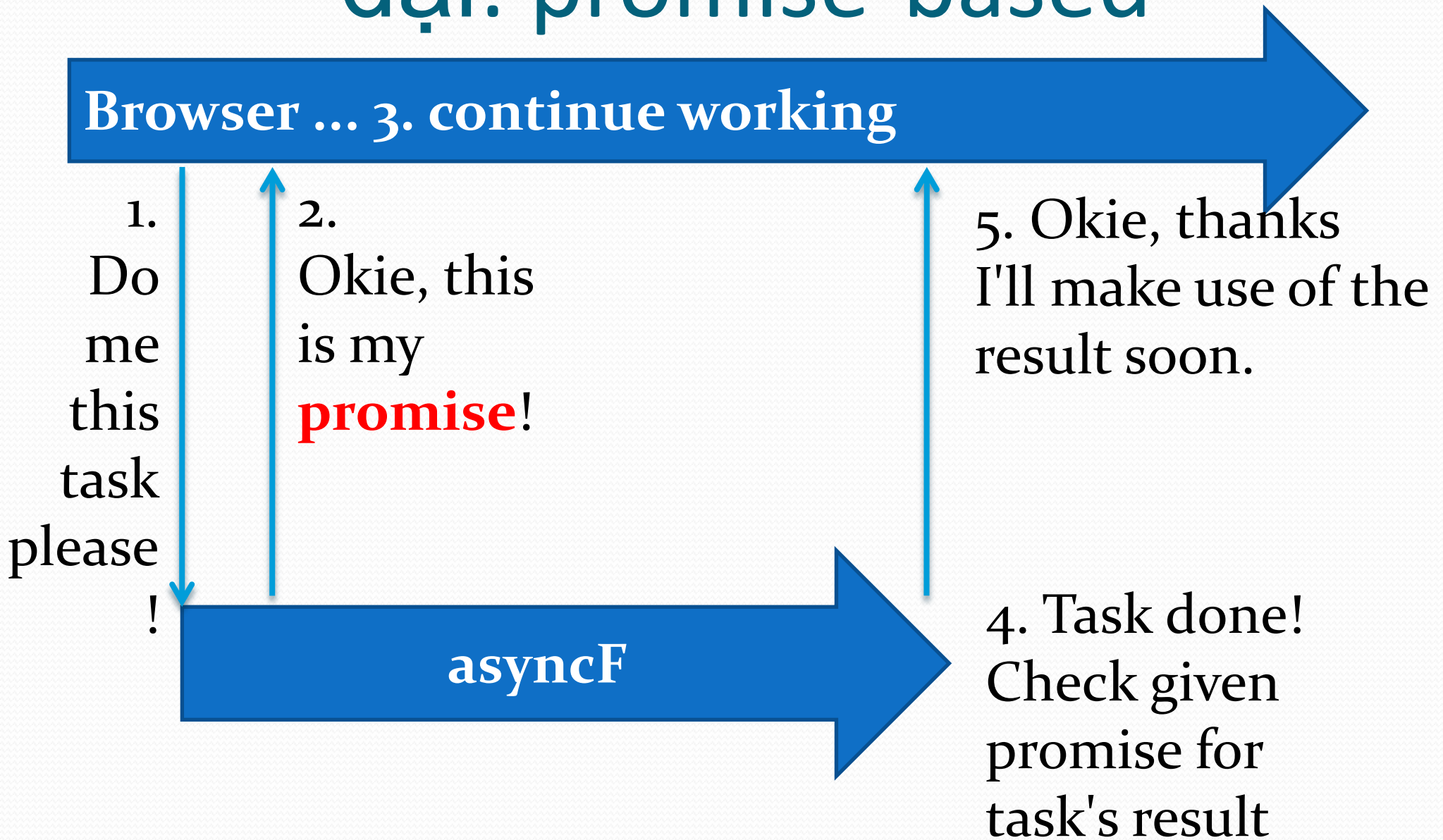
```
function aFunction() {  
  setTimeout( function() {  
    let ret = sFunction();  
    console.log(ret);  
  }, 0);  
}  
console.log("begin");  
aFunction();  
console.log("end");
```

```
begin  
end  
Hello S!
```

Hàm gọi lại

- Là cách thức truyền thống để tạo tác vụ không đồng bộ
- Hàm gọi lại (callback) được truyền vào tham số cho một hàm khác (thường là Web API)
 - `window.setTimeout(callback, time);`
 - `window.setInterval(callback, time);`
 - `window.requestAnimationFrame(callback);`
 - `obj.addEventListener(eventName, callback);`
- Hàm gọi lại không được thực hiện ngay, mà được đưa vào hàng đợi và được "gọi lại" ở một thời điểm trong tương lai.

Lập trình không đồng bộ hiện đại: promise-based



Promise

- "Giấy hẹn trả kết quả"
- Chứa kết quả trả về khi hoàn thành, lỗi nếu thất bại
- Được đưa vào hàng đợi và xử lý sau.
 - Khác hàng đợi callback

Hàm tạo Promise()

```
var promiseObj = new Promise(executor);
```

```
var promiseObj = new Promise( function(fulfill, reject) {  
    // typically, some asynchronous operation.  
} );
```

Hai hàm *fulfill*, *reject* tự động được tạo và buộc vào đối tượng promise

- *fulfill* trả về kết quả nếu nhiệm vụ hoàn thành
- *reject* trả về kết quả nếu nhiệm vụ thất bại hay bị từ chối

Ví dụ 1

```
function aFunction1 () {  
    return new Promise(  
        function(fulfill, reject) {  
            fulfill("Hello A1!");  
        }  
    );  
}
```

Ví dụ 2

```
function imgLoad(url) {  
    return new Promise(function(fulfill, reject) {  
        let request = new XMLHttpRequest();  
        request.open('GET', url);  
        request.responseType = 'blob';  
        request.onload = function() {  
            if (request.status === 200) {  
                fulfill(request.response); } else { reject(Error("didn't"));  
            }  
        }  
        request.send();  
    });  
}
```

Tạo promise bằng async

- Thay cho sử dụng constructor để tạo promise
- Từ khóa `async` được thêm vào định nghĩa hàm
 - Câu lệnh `return` của hàm sẽ tự động bao gói kết quả trả về trong promise.
 - Hàm luôn trả về promise
 - Cách viết hàm như hàm đồng bộ

```
async function doTaskA() { return value; };  
let promiseA = doTaskA();
```

Ví dụ 1

```
function aFunction1 () {  
  return new Promise(  
    function(fulfill, reject) {  
      fulfill("Hello A1!");  
    }  
  );  
}
```

```
async function aFunction2 () {  
  return ("Hello A2!");  
}
```

Ví dụ 2

```
async function imgLoad2(url) {  
    let request = new XMLHttpRequest();  
    request.open('GET', url);  
    request.responseType = 'blob';  
    request.onload = function() {  
        if (request.status === 200) {  
            return (request.response);  
        }  
    }  
    request.send();  
}
```

Các phương thức của promise

- Chức năng
 - Trả kết quả
 - tại một thời điểm nào đó trong tương lai
- Các phương thức cho đối tượng/thể hiện
 - then()
 - catch()
 - finally()
- Các phương thức tĩnh
 - all()
 - allSettled()
 - any()
 - race()

promise.then()

- `let promise1 = new Promise(...);`
- `let promise2 = promise1.then(callback);`
- `let promise3 = promise2.then(function(param) {});`

hoặc viết nổi

- `let promise3 = Promise(...).then(callback).then(function(param) {});`
- Nếu kết quả là thành công (fulfill) thì gọi callback và
 - Tự động bóc kết quả chứa trong promise và truyền vào tham số cho callback
 - Chuyển callback thành hàm không đồng bộ
- Tạo thành chuỗi promise

Ví dụ chuỗi promise 1

aFunction1()

.then(/* then () tự động bóc nội dung (kết quả) chứa trong promise và truyền vào tham số cho callback */
function (ret) {
 console.log(ret);
}
);

Ví dụ chuỗi promise 2

```
imgLoad('myLittleVader.jpg')
```

```
.then( /* then () tự động bóc nội dung (kết quả) chứa  
trong promise và truyền vào tham số cho callback */
```

```
function(resp) {
```

```
    let body = document.querySelector('body');
```

```
    let myImage = document.createElement("img");
```

```
    myImage.src = window.URL.createObjectURL(resp);
```

```
    body.appendChild(myImage);
```

```
    return myImage; /* then() tự động tạo promise mới và đưa  
kết quả trả về vào promise mới , trả về promise mới */
```

```
})
```

```
.then (function(newCreatedImage) { ... });
```

promise.catch()

- Sau tất cả các then()
- Được gọi nếu ít nhất một promise (.then()) thất bại.

```
imgLoad('myLittleVader.jpg')
```

```
.then( function(resp) { ... })
```

```
.then (function(newCreatedImage) { ...})
```

```
.catch (function(err) { console.log(err); });
```

promise.finally()

- Cuối chuỗi promise
- Được thực hiện sau cùng bất luận chuỗi promise hoàn thành tất cả hay có một promise thất bại

```
imgLoad('myLittleVader.jpg')
```

```
.then( function(resp) { ... })
```

```
.then (function(newCreatedImage) { ...})
```

```
.catch (function(err) { console.log(err); })
```

```
.finally(function() { console.log("finished"); });
```

Lập trình không đồng bộ === Lập lịch/kế hoạch

<code>doTaskA()</code>	<code>// promiseA queued</code>
<code>.then (doTaskB)</code>	<code>// promiseA fulfilled, promiseB queued</code>
<code>.then(doTaskC)</code>	<code>// promiseB fulfilled, promiseC queued</code>
<code>...</code>	
<code>.catch(handleError)</code>	<code>// at least one promise was rejected</code>
<code>.finally(finish);</code>	<code>// away do last</code>

Promise.all()

- Khi tất cả các promise hoàn thành hoặc ít nhất một promise thất bại

```
let promiseA = doTaskA() ;
```

```
let promiseB = doTaskB() ;
```

```
let promiseC = doTaskC() ;
```

```
let promiseD = Promise.all([promiseA , promiseB,  
    promiseC]);
```

```
promiseD.then(function(arr) {...});
```

Promise.allSettled()

- Khi tất cả các promise được giải quyết xong (hoặc hoàn thành hoặc thất bại)

```
let promiseA = doTaskA() ;
```

```
let promiseB = doTaskB() ;
```

```
let promiseC = doTaskC() ;
```

```
let promiseD = Promise.allSettled([promiseA , promiseB,  
    promiseC]);
```

```
promiseD.then(function(arr) {...});
```


Promise.any()

- Khi có ít nhất một promise hoàn thành

```
let promiseA = doTaskA() ;
let promiseB = doTaskB() ;
let promiseC = doTaskC() ;
let promiseD = Promise.any([promiseA , promiseB,
    promiseC]);
promiseD.then(function(value) {...});
```
- Có thể không có promise nào hoàn thành (tất cả đều thất bại)

Promise.race()

- Khi có ít nhất một promise hoàn thành hoặc thất bại

```
let promiseA = doTaskA() ;
```

```
let promiseB = doTaskB() ;
```

```
let promiseC = doTaskC() ;
```

```
let promiseD = Promise.race([promiseA , promiseB,  
    promiseC]);
```

```
promiseD.then(function(value) {...});
```

await

- Đợi một promise cho đến khi promise được hoàn thành hoặc thất bại
 - `let ret= await aFunction().catch((err) => { console.error(err); });`
- Chỉ được dùng await trong hàm async

```
1  async function fetchAndDecode(url, type) {
2    let response = await fetch(url);
3
4    let content;
5
6    if (!response.ok) {
7      throw new Error(`HTTP error! status: ${response.status}`);
8    } else {
9      if(type === 'blob') {
10        content = await response.blob();
11      } else if(type === 'text') {
12        content = await response.text();
13      }
14
15      return content;
16    }
17  }
```

await

- Thay cho then

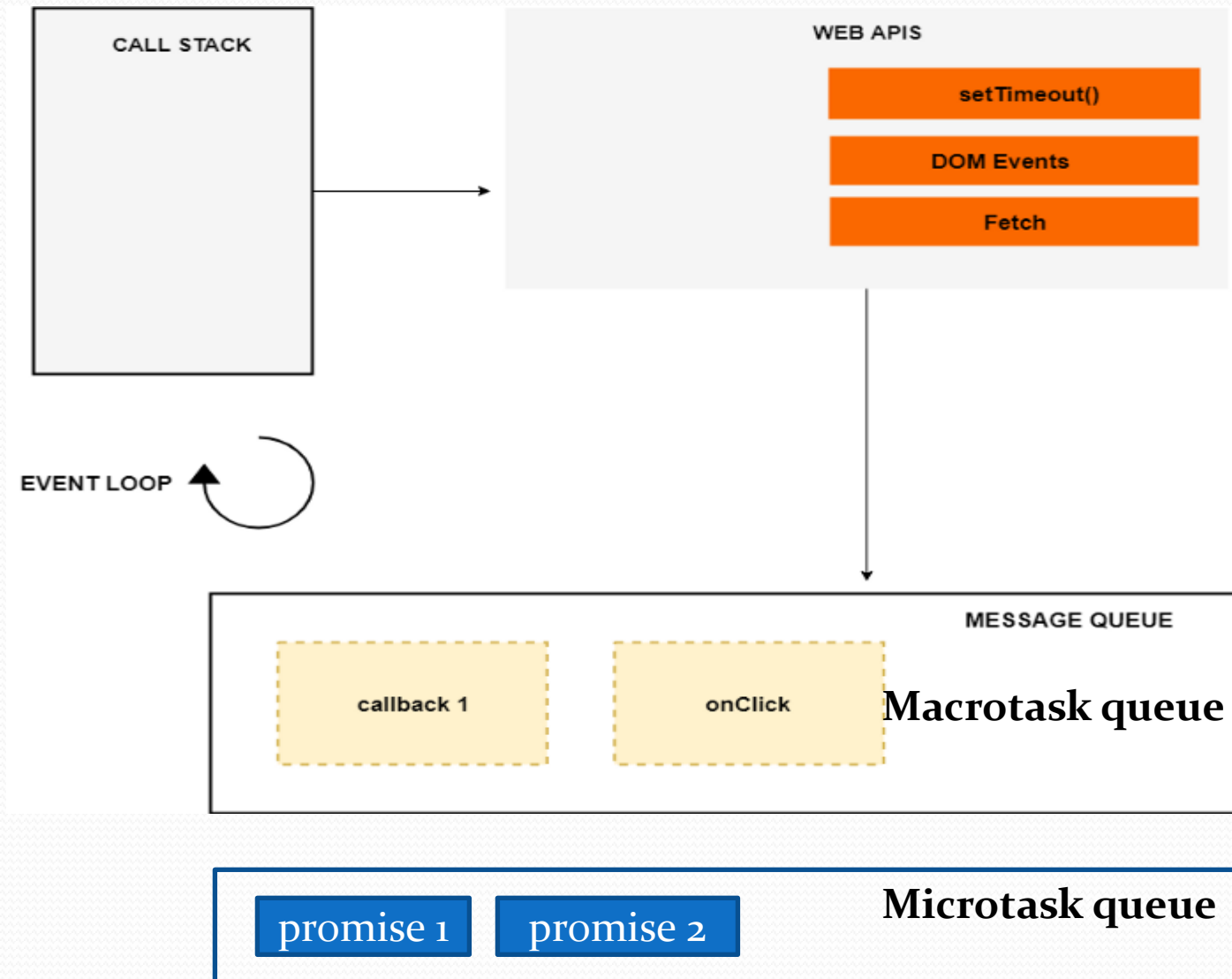
```
async function fetchAndDecode(url, type) {  
  fetch(url).then(function(response) {  
    if (!response.ok) {  
      throw new Error(`HTTP error! status: ${response.status}`);  
    } else {  
      if (type === 'blob') {  
        response.blob().then(function(content) {  
          return content;  
        });  
      } else if (type === 'text') {  
        response.text().then(content => {return content;});  
      }  
    }  
  });  
}
```

```
1  async function fetchAndDecode(url, type) {  
2    let response = await fetch(url);  
3  
4    let content;  
5  
6    if (!response.ok) {  
7      throw new Error(`HTTP error! status: ${response.status}`);  
8    } else {  
9      if (type === 'blob') {  
10         content = await response.blob();  
11      } else if (type === 'text') {  
12         content = await response.text();  
13      }  
14  
15      return content;  
16    }  
}
```

Hàng đợi microtask

- Promise được đưa vào hàng đợi microtask (microtask/job queue).
- Tất cả promise trong hàng đợi, kể cả promise mới phát sinh khi xử lý promise cũ, sẽ được xử lý ngay khi một task kết thúc và trước khi task khác được xử lý.

Browser's JavaScript runtime



Hàng đợi macrotask

- Trình duyệt/JavaScript hoạt động theo sự kiện (event-driven)
 - Thực thi chỉ xảy ra khi có sự kiện (nhằm xử lý sự kiện đó)
 - run script, load, readystatechange, click, mouseover, timeout, ...
- Các hàm xử lý sự kiện (event handler, callback) được đưa vào hàng đợi macrotask (task/message/callback queue).

Vòng lặp sự kiện (Event loop)

```
while (true) {  
  if (the call stack is empty) {  
    while (the microtask queue is not empty) {  
      dequeue the oldest (micro)task and run it (push it to the call  
        stack)  
    }  
    if (the macrotask queue is not empty) {  
      dequeue the oldest (macro)task and run it (push it to the call  
        stack)  
    }  
    repaint (DOMchanged)  
  }  
}
```


Ví dụ

```
console.log('Start');
```

```
setTimeout(() => { console.log('setTimeout 1'); }, 10);
```

```
setTimeout(() => { console.log('setTimeout 2'); }, 0);
```

```
new Promise(function(fulfill, reject) { fulfill('Promise 1'); })  
.then (function(res) {console.log(res); return "Sub Promise 1";})  
.then (function(res) {console.log(res);});
```

```
new Promise((fulfill, reject) => { fulfill('Promise 2'); })  
.then (res => {console.log(res); return "Sub Promise 2";})  
.then (res => {console.log(res);});
```

```
console.log('End');
```

Start

End

Promise 1

Promise 2

Sub Promise 1

Sub Promise 2

setTimeout 2

setTimeout 1



Thêm về microtask

- promise là một dạng microtask
 - Mỗi khi được tạo, promise được đưa vào hàng đợi microtask
- Có thể tạo và thêm một microtask vào hàng đợi microtask bằng hàm `queueMicrotask(function())`;

Lời khuyên

- Sử dụng promise/hàm không đồng bộ để tránh block luồng chính với các thao tác
 - Vào/ra
 - Gửi request/nhận response
 - Giao tiếp mạng
- Không sử dụng promise/hàm không đồng bộ với các thao tác
 - Chỉ xử lý trong CPU
 - Vì chỉ có tác dụng thay đổi thứ tự thực hiện các job và task
 - Vẫn block tác vụ tiếp theo

Xử lý song song

JavaScript đơn luồng

- Đơn luồng (single-threaded)
 - Dù có nhiều CPU hoặc CPU có nhiều core, các tác vụ JavaScript cũng chỉ được xử lý trên một luồng (main thread)
 - Các tác vụ được thực thi một cách tuần tự (sequentially)
- Giải pháp cho xử lý đa luồng trong môi trường thực thi của trình duyệt
 - Web worker
 - chạy ở luồng khác
 - tận dụng được CPU và CPU core khác
 - chạy các tác vụ lớn
 - luồng chính/trình duyệt) không bị block

Web worker

- Cho phép chạy tệp JavaScript ở luồng (thread) riêng ở chế độ nền bên ngoài luồng chính (ngữ cảnh window)
 - Không làm cho luồng chính bị treo (non-blocking)
 - Giao tiếp với luồng chính bằng gửi/nhận thông báo (message)
 - Gửi: **postMessage(m)**;
 - Nhận: **onmessage = function() {}**
- Có thể mã bất kỳ, ngoại trừ
 - không truy cập được DOM
 - không sử dụng được một số phương thức và thuộc tính của window
 - **Functions and classes available to Web Workers**

Phân loại web worker

- Chuyên biệt (dedicated web worker)
 - chỉ làm việc được với một luồng chính (window)
- Chia sẻ (shared web worker)
 - có thể làm việc (chia sẻ) được với nhiều luồng chính (window, tab, iframe)
 - cùng domain

Dedicated web workers

```
1  onmessage = function(e) {  
2      console.log('Worker: Message received from main script');  
3      let result = e.data[0] * e.data[1];  
4      if (isNaN(result)) {  
5          postMessage('Please write two numbers');  
6      } else {  
7          let workerResult = 'Result: ' + result;  
8          console.log('Worker: Posting message back to main script');  
9          postMessage(workerResult);  
10     }  
11 }
```


Trang chính

```
1  const first = document.querySelector('#number1');

6  if (window.Worker) {
7      const myWorker = new Worker("worker.js");
8
9      first.onchange = function() {
10         myWorker.postMessage([first.value, second.value]);
11         console.log('Message posted to worker');
12     }
19     myWorker.onmessage = function(e) {
20         result.textContent = e.data;
21         console.log('Message received from worker');
22     }
```

Shared web worker

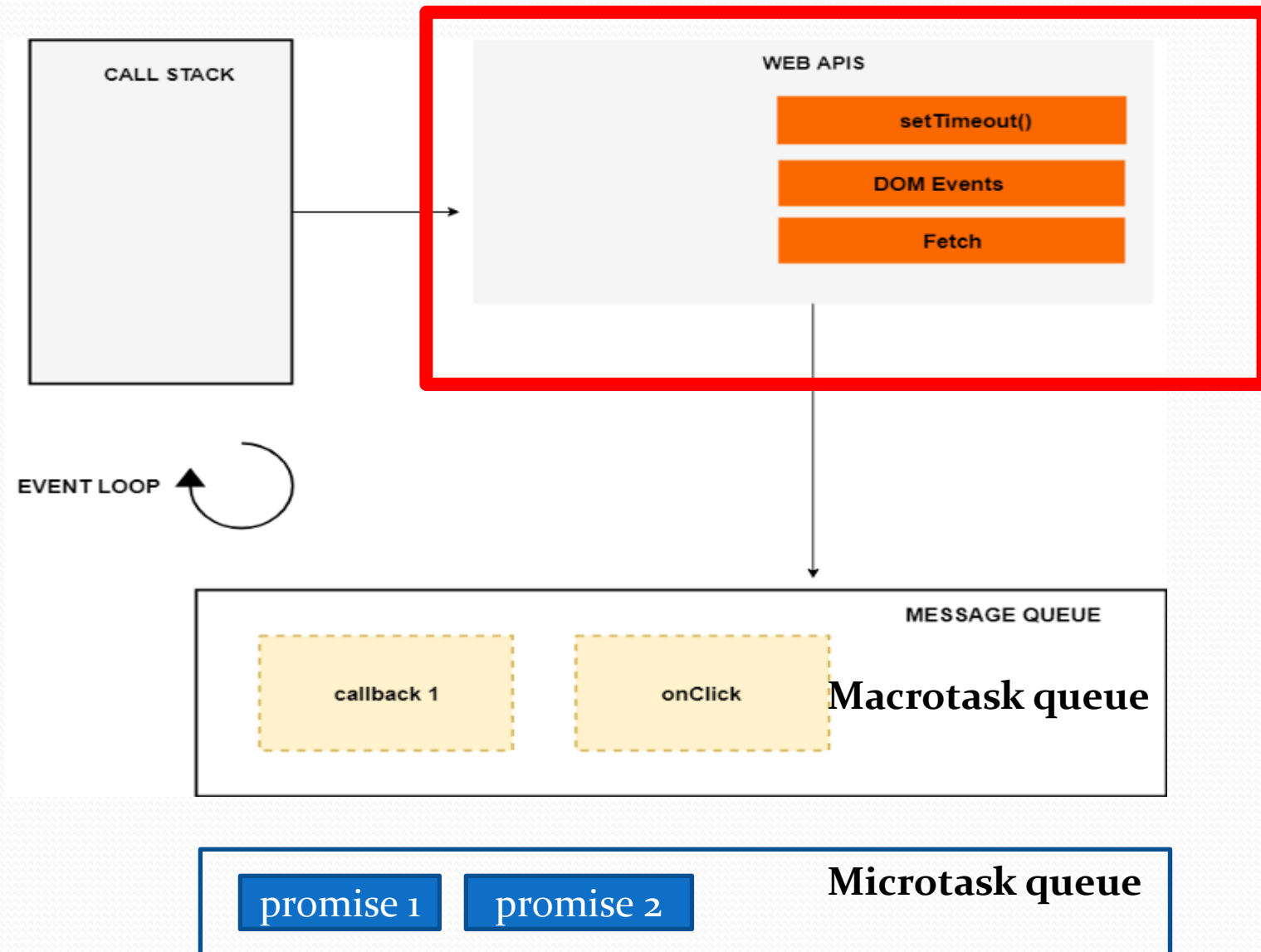
```
1 let c = 0;
2 onconnect = function(e) {
3     let port = e.ports[0];
4     port.onmessage = function(e) {
5         c++;
6         port.postMessage(c);
7     };
8 }
```

Trang chính

```
1 <!DOCTYPE html><html><head>
2   <meta charset="utf-8">
3   <title>Shared worker 1</title>
4 </head><body>
5   <button class="inc">Increase</button>
6   <span class="result"></span>
7   <script>
8     let worker = new SharedWorker("sw.js");
9
10    document.querySelector(".inc").onclick = function() {
11      worker.port.postMessage("inc");
12    };
13
14    worker.port.onmessage = function(e) {
15      document.querySelector(".result").innerHTML = e.data;
16    };
17  </script>
18 </body>
19 </html>
```

Web API

Browser's JavaScript runtime



Specifications

This is a list of all the APIs that are available.

A Ambient Light Events	F Fetch API File System API ⚠ Frame Timing API Fullscreen API	Media Source Extensions ⚠ MediaStream Recording	S Streams ⚠
B Background Tasks Battery API 🗑 Beacon Bluetooth API Broadcast Channel API	G Gamepad API ⚠ Geolocation API	N Navigation Timing Network Information API ⚠	T Touch Events
C CSS Counter Styles CSS Font Loading API ⚠ CSSOM	H HTML Drag and Drop API High Resolution Time History API	P Page Visibility API Payment Request API Performance API Performance Timeline API Permissions API Pointer Events	U URL API V Vibration API Visual Viewports W Web Animations

Một số API quan trọng

- DOM
- Fetch
- URL
- ...

Fetch API

- Cung cấp giao diện cho phép
 - tạo và gửi request
 - nhận response
 - tạo response
- Các đối tượng
 - Request
 - Response
 - Headers
 - Body (mixin)
- Phương thức toàn cục
 - fetch()
 - bản thay thế hiện đại hơn XMLHttpRequest (AJAX)

fetch()

- Đơn giản với GET method

```
1  fetch('http://example.com/movies.json')  
2    .then(response => response.json())  
3    .then(data => console.log(data));
```

fetch()

- Cấu hình header và body của HTTP request

```
const response = await fetch(url, {  
  method: 'POST', // *GET, POST, PUT, DELETE, etc.  
  mode: 'cors', // no-cors, *cors, same-origin  
  cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached  
  credentials: 'same-origin', // include, *same-origin, omit  
  headers: {  
    'Content-Type': 'application/json'  
    // 'Content-Type': 'application/x-www-form-urlencoded',  
  },  
  redirect: 'follow', // manual, *follow, error  
  referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, c  
  body: JSON.stringify(data) // body data type must match "Content-Type" header  
});  
  
return response.json(); // parses JSON response into native JavaScript objects
```

fetch()

- POST dữ liệu JSON

```
1  const data = { username: 'example' };
2
3  fetch('https://example.com/profile', {
4    method: 'POST', // or 'PUT'
5    headers: {
6      'Content-Type': 'application/json',
7    },
8    body: JSON.stringify(data),
9  })
10 .then(response => response.json())
11 .then(data => {
12   console.log('Success:', data);
13 })
14 .catch((error) => {
15   console.error('Error:', error);
16 });
```

fetch()

- Upload file (form)

```
1  const formData = new FormData();
2  const fileField = document.querySelector('input[type="file"]');
3
4  formData.append('username', 'abc123');
5  formData.append('avatar', fileField.files[0]);
6
7  fetch('https://example.com/profile/avatar', {
8    method: 'PUT',
9    body: formData
10 })
11 .then(response => response.json())
12 .then(result => {
13   console.log('Success:', result);
14 })
15 .catch(error => {
16   console.error('Error:', error);
17 });
```

Headers và Request

- Tạo đối tượng Request rồi gửi
 - Ít dùng

```
1  const myHeaders = new Headers({
2    'Content-Type': 'text/plain',
3    'Content-Length': content.length.toString(),
4    'X-Custom-Header': 'ProcessThisImmediately'
5  });
3  const myRequest = new Request('flowers.jpg', {
4    method: 'GET',
5    headers: myHeaders,
6    mode: 'cors',
7    cache: 'default',
8  });
9
10 fetch(myRequest)
11   .then(response => response.blob())
12   .then(myBlob => {
13     myImage.src = URL.createObjectURL(myBlob);
14   });
```

Response

- Thuộc tính
 - `Response.status`
 - `Response.statusText`
 - `Response.ok`
 - ...
- Phương thức
 - `Response.arrayBuffer()`
 - `Response.blob()`
 - `Response.json();`
 - `Response.text();`
 - ...



Tiếp theo

PWA