香港中文大學
The Chinese University of Hong Kong

ASYNCHRONOUS JS AND FETCH API

CSCI2720 2022–23 Term 1

Building Web Applications

Dr. Chuck-jee Chau
chuckjee@cse.cuhk.edu.hk

OUTLINE

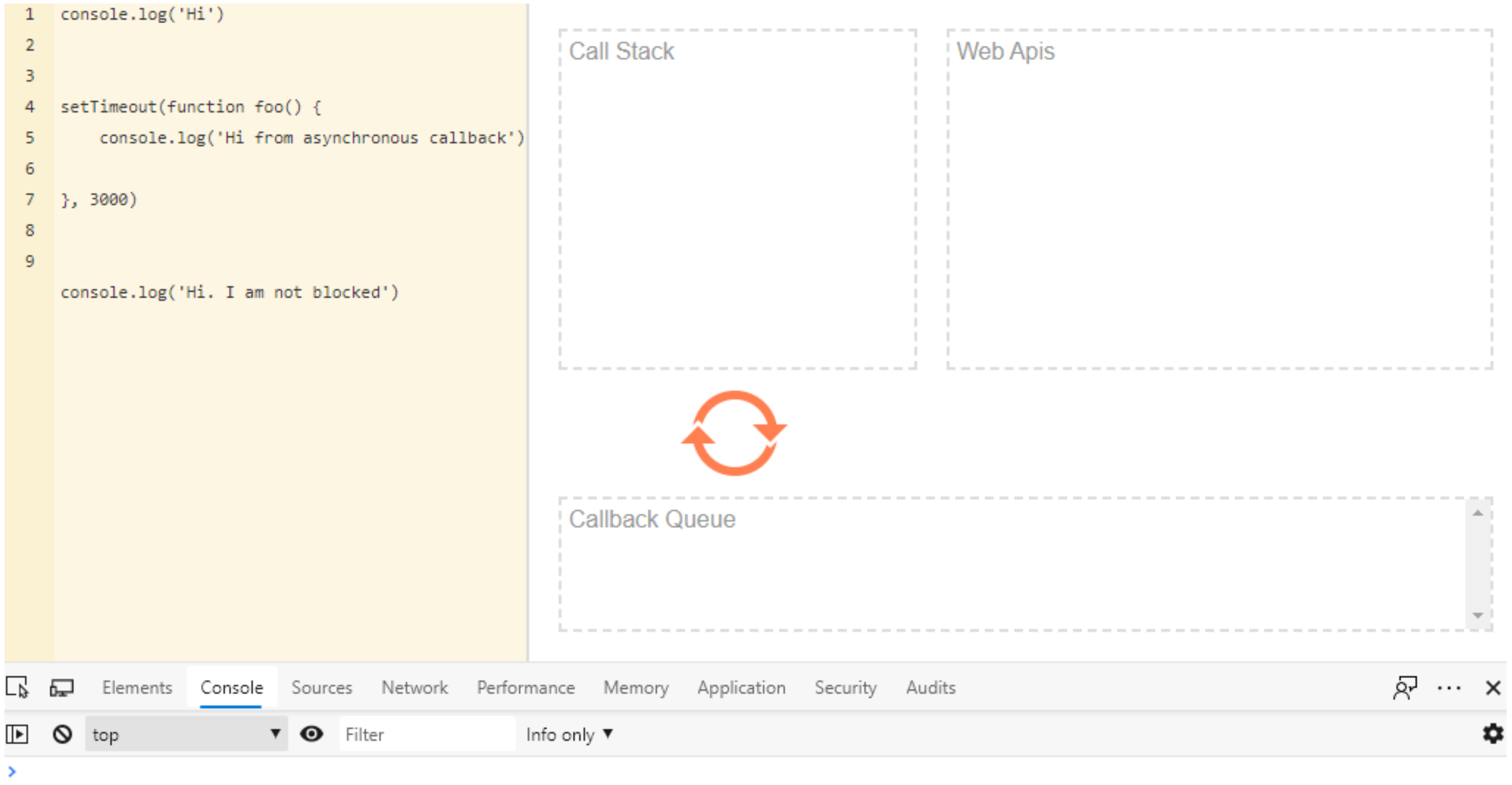
- Asynchrony in JavaScript
- Callback functions
- Promise

- The Fetch API
- Something about AJAX

ASYNCHRONY IN JAVASCRIPT

- Still remember that? JavaScript is single-threaded
  - One execution at a time: the ***Call Stack*** determines what to run next, sequentially
  - If there is an action which takes a while, everything else is blocked…
    → *synchronous* programming
- ***Event loop*** in the JS engine keeps checking if the call stack is empty, and brings events from the callback queue to the call stack

ASYNCHRONY IN JAVASCRIPT



```
console.log('Hi')


setTimeout(function foo() {
    console.log('Hi from asynchronous callback')

}, 3000)


console.log('Hi. I am not blocked')
```

See: https://thecodest.co/blog/asynchronous-and-single-threaded-javascript-meet-the-event-loop/
http://latentflip.com/loupe/ *(check the console too!)*

ASYNCHRONY IN JAVASCRIPT

- Asynchronous programming in JS
  - A task is started, but without waiting for it to finish
  - When the task is done, *something* would happen…
    - Events, callbacks, promises, …
  - This is important for I/O which requires waiting
    - Web data submission or retrieval
    - Database execution
  - A precise control of the *execution order* of steps is necessary

SCHEDULING CALLS IN JS

- Two methods for ***executing a function*** later
  - **`setTimeout()`** – run the function after certain time
  - **`setInterval()`** – run the function repeatedly at interval
  - **`clearTimeout()`** and **`clearInterval()`** are the stopping mechanisms
- Possible to nest them for special timing settings
- *See: https://javascript.info/settimeout-setinterval*

```
setTimeout(() => console.log("hello"), 2000);
// hello appears after 2 seconds
```

CALLBACK FUNCTIONS

- When a function is passed as an ***argument*** of another function to be called later, that is a ***callback function*** (or just a *callback*)
  - It will be called when the calling function is done

```
function waitnprint(str, cb) {
  setTimeout( function() { // ...wait for a while...
    console.log(str);
    cb();
  }, 1000);
}
// the callback needs to be called manually
```

CALLBACK FUNCTIONS

- The *Callback Hell*: multiple waits are possible by chaining up callbacks, but the code looks ugly

```
waitnprint("Hello", function() {
  waitnprint("World", function() {
    waitnprint("!", function() {
      waitnprint("END", function(){} );
    })
  })
});
```

Simplified with arrow functions

```
waitnprint("Hello", ()=>
  waitnprint("World", ()=>
    waitnprint("!", ()=>
      waitnprint("END", ()=>{} )
    )
  )
)
```

https://codepen.io/chuckjee/pen/LYVKMZv

PROMISE

- The **`Promise`** object represents the results of an asynchronous execution, with 3 states:
  - `pending`: initial state
  - `fulfilled`: task was done! → a result value can be found
  - `rejected`: task failed → an error object can be found
- Involves a ***success*** callback and a ***failure*** callback
  - Both are *optional*!
- The **`Promise`** object is now widely used by async operations
  - Used via the **`then()`** method of the promise, which takes two callbacks
  - `async/await` is available as ***syntactic sugar*** which gets popular as well

PROMISE

- The syntax of a Promise

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)
  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

- *See: https://www.w3schools.com/js/js_promise.asp*

PROMISE CHAIN

https://codepen.io/chuckjee/pen/eYNwbBK

```
function waitnprint(str) {
  return new Promise((resolve, reject) => {
    setTimeout( function() { // ...wait for a while...
      console.log(str);
      resolve();
    }, 1000);
  })
}
```

```
waitnprint("Hello")
  .then(()=>waitnprint("World"))
  .then(()=>waitnprint("!"))
  .then(()=>waitnprint("END"))
  .catch((err)=>{...});
```

Note: Here only one **`catch()`** *at the end of the promise chain to handle the errors*

PROMISE FINALLY

- Similar to a chain of *try-catch-finally*, now the syntax is applicable to promises as well
  - *We will talk about fetch() in a moment*

```
fetch('https://www.google.com')
  .then((response) => {
    console.log(response.status);
  })
  .catch((error) => {
    console.log(error);
  })
  .finally(() => {
    document.querySelector('#spinner').style.display='none';
  });
```

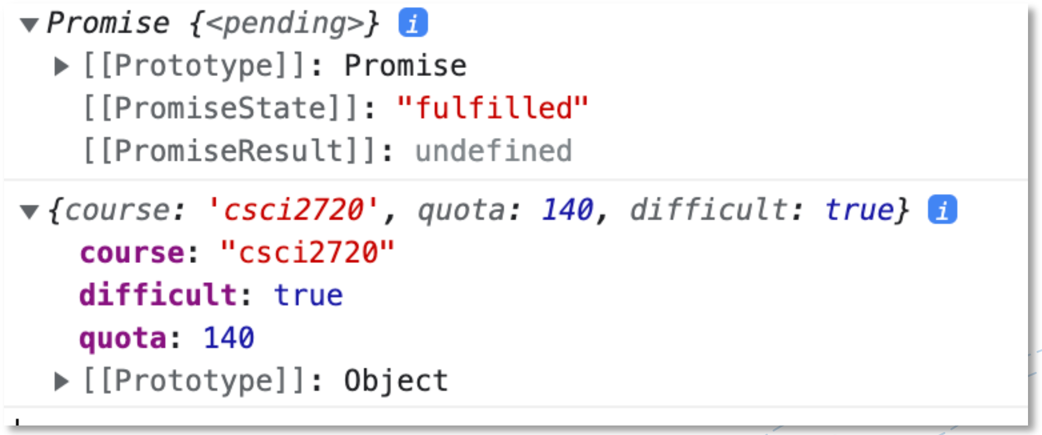
THE FETCH API

- Without reloading a web page, how can new data be retrieved from the server?
  - Asynchronous data retrieval: **`fetch()`**
  - **`fetch()`** returns a Promise object for easy handling
- For security reasons, such async JS data loading by default requires “same origin”, i.e., on the same server/port
- **`fetch()`** can also be used for submitting data to server
- *See: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch*

THE FETCH API

- When the main page is on https://www.cse.cuhk.edu.hk/~chuckjee , run this in console: *(otherwise there would be a CORS error)*

```
fetch('https://www.cse.cuhk.edu.hk/~chuckjee/csci2720.json')
// parsing retrieved data as JSON
.then(res=>res.json())
// displaying data
.then(data=>console.log(data))
```



```
▼Promise {<pending>} i
  ▶[[Prototype]]: Promise
   [[PromiseState]]: "fulfilled"
   [[PromiseResult]]: undefined
▼{course: 'csci2720', quota: 140, difficult: true} i
   course: "csci2720"
   difficult: true
   quota: 140
  ▶[[Prototype]]: Object
```

- The response can be handled as:
  - `arrayBuffer()`, `blob()`, `json()`, `text()`, `formData()`

SOMETHING ABOUT AJAX

- Some years ago, most of async data retrieval was done with ***AJAX*** (asynchronous JavaScript and XML), using an **`XMLHttpRequest`** (XHR) object, perhaps with the help of jQuery
- Nowadays **`fetch()`** becomes a more prominent way thanks to the simplicity with syntax
- There are *subtle differences* between the two options

READ FURTHER…

MDN Using Promises

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

A guide to writing asynchronous JavaScript programs

http://callbackhell.com

Fetch on javascript.info

https://javascript.info/fetch