

ASYNCHRONOUS JAVASCRIPT

By Dmytro Minochkin

softserve

PLAN

Async in JS

Events & Callbacks

Promises

Promise overview

Promise magic

.then(), .catch(),
parallel execution

Async/Await

Write async code as
sync!

ASYNC IN JS

EVENTS

- The most basic form of asynchronous programming in JavaScript.

```
const btn = document.getElementById("btn")  
btn.onclick = (event) ⇒ console.log(event)
```

CALLBACKS: BASIC

- Functions that are called after an asynchronous operation
 - Usually are passed as arguments.

```
import { readFile } from 'fs';

readFile('file.txt', (err, contents) => {
  if (err) {
    throw err
  }
  console.log(contents)
});


console.log("Hi!")
```

CALLBACKS: HARD

```
function getPart1(callback) {  
    $.get('...', callback);  
}  
  
function getPart2(callback) {  
    $.get('...', callback);  
}  
  
function getPart3(callback) {  
    $.get('...', callback);  
}  
  
getPart1(function () {  
    getPart2(function () {  
        getPart3(function () {  
            // Add value  
        });  
    });  
});
```

CALLBACKS: HELL

```
1  function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                 loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SCRIPTS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });
23     });
24   });
25 };
26 }
```



PROMISES

softserve

PROMISES

- Operation that will return its result some time later
- Multiple handlers of one result
- Complex compositions of asynchronous operations
- Easier to handle errors
- You can "recover" from an error

.THEN

```
// function returns a promise
const promise = doSomething(42);

// ...

// handle the result
promise.then(value => {
  console.log(value) // 42
});
```

PROMISES & THENABLES

- **Promises** are objects whose behaviour conforms to the Promise / A + specification
- **Thenables** - objects that have the `.then` method.

```
function doSomething(value) {  
  return {  
    then: function (callback) {  
      callback(value)  
    }  
  }  
}  
  
const promise = doSomething(42);  
  
promise.then(value => {  
  console.log(value) // 42  
});
```

CONSTRUCTOR

- The function passed to `new Promise()` is called the *executor*.
- `.then()` and `resolve()` are called independently.

```
const promise = new Promise((resolve, reject) => {  
  if (true) { resolve(result) }  
  else { reject(error) }  
});
```

PROMISE STATES

- A Promise is in ***one of three states***.
 - **pending**: initial state, neither fulfilled nor rejected.
 - **fulfilled**: meaning that the operation completed successfully.
 - **rejected**: meaning that the operation failed.



WORKING WITH PROMISES

softserve

.THEN() MAGIC

- **.then()** always returns a new promise
 - This new promise is resolved when the previous one was either completed or rejected.
- **.then()** may not have a handler
 - The result is transferred to the next promise
- If you return a value from the handler in **.then()**, it will be the value of the next promise
 - You can transfer data from one promise to the next

```
function doSomething(value) {  
    return new Promise((resolve, reject) => {  
        resolve(value)  
    });  
}  
  
doSomething(0)  
    .then() // optional handler  
    .then(res => [res]) // transform into an array  
    .then(res => res.concat(1)) // add 1  
    .then(res => res.concat(2)) // add 2  
    .then(res => res.concat(3)) // add 3  
    .then(res => console.log(res)) // [ 0, 1, 2, 3 ]  
    .then(res => console.log(res)) // undefined
```

.THEN() & NEW PROMISE

- If you return a promise with .then (), it will be resolved
- The result will be wrapped in a new promise and will be available in the next .then()

```
function doSomething(value) {  
  return new Promise((resolve, reject) => {  
    resolve(value)  
  });  
}  
  
doSomething(1)  
  .then(value => {  
    console.log(value)           // 1  
    return doSomething(2)       // return a promise  
  })  
  .then(value => {  
    console.log(value)           // 2  
    return doSomething(3)       // return a promise  
  })  
  .then(value => {  
    console.log(value)           // 3  
    // ...  
  });
```


ERROR HANDLING

- Promises can be rejected with reason
 - As a second argument, the executor receives a function that rejects the promise
- .then() can receive an error handler as the second argument
 - Both the completion handler and error handler are optional.
- .catch() only accepts an error handler
 - .catch() is similar to .then() without a completion handler.

```
const promise = new Promise((resolve, reject) => {  
  reject('error!')  
});  
  
promise.catch(reason => {  
  // error handler  
  console.error(reason)    // 'error!'  
});  
  
// equal to  
  
promise.then(null, reason => {  
  console.error(reason)    // 'error!'  
});
```

RECOVERY

- Promises are able to "recover"
 - If you return a value from the error handler, it will go to the completion handler of the next promise.

```
function doSomething(value) {  
  return new Promise((resolve, reject) => {  
    reject(value)  
  })  
}  
  
// ---  
  
doSomething(42)  
  .then(value => {  
    console.log(`Result: ${value}`)           // will never execute  
  }, reason => {  
    console.error(`Error: ${reason}`)         // 42  
    return 'recovered!'                       // recover the promise  
  })  
  .then(value => {  
    console.log(`Result: ${value}`)           // 'recovered!'  
  }, reason => {  
    console.error(`Error: ${error}`)         // there is no error  
  });
```

FAIL SILENTLY

- If an error occurs in a promise that does not have an error handler, the promise will "keep quiet" about the error.

CREATING COMPLETED PROMISES

- `Promise.resolve()` and `Promise.reject()` allow you to create a completed promise
 - They can take primitives as an argument and wrap them in a promise.

```
const resolved = Promise.resolve(42);

resolved.then(value => console.log(value)) // 42

// ...

const rejected = Promise.reject('Boom!');

rejected.catch(reason => console.error(reason)) // 'Boom!'
```

BACK TO THENABLES

- Promise.resolve () and Promise.reject () can take thenable arguments
 - They can turn thenables into real promises.

```
const thenable = {  
  then: (resolve, reject) => {  
    resolve(42)  
  }  
}  
  
// convert to promise  
const promise = Promise.resolve(thenable)  
  
promise.then(value => console.log(value)) // 42
```

PARALLEL EXECUTION: ALL

- Promise.all() is waits for all promises to complete
 - Returns a new promise, which resolves when all promises are completed.

```
const p1 = Promise.resolve('i');
const p2 = Promise.resolve('like');
const p3 = Promise.resolve('promises!');

const p4 = Promise.all([p1, p2, p3]);

p4.then(values => {
  console.log(Array.isArray(values)) // true
  console.log(values[0])             // 'i'
  console.log(values[1])             // 'like'
  console.log(values[2])             // 'promises!'
});
```

PARALLEL EXECUTION: RACE

- Promise.race () creates a race among promises
 - Returns a new promise, which completes when the fastest promise ends.

```
const p1 = Promise.resolve('i');
const p2 = new Promise((resolve, reject) => resolve('like'));
const p3 = new Promise((resolve, reject) => resolve('promises!'));

const p4 = Promise.race([p1, p2, p3]);

p4.then(value => {
  console.log(value);    // 'i'
})
```

ASYNCR/AWAIT T

ASYNC/AWAIT

- ES7 introduced a new way to add async behaviour in JavaScript
 - Working with promises became easier
- New **async** and **await** keywords
 - **async** functions will implicitly return a **promise**.
 - Don't have to create new promises yourself.

```
// promise way
Promise.resolve("Hello");

// is equal to

async function greet() {
  return "Hello";
}
```

SUSPEND EXECUTION

- **await** keyword *suspends* the asynchronous function and waits for the awaited value return a resolved promise
- To get the value of this resolved promise just assign variables to the awaited promise value!
 - Like we previously did with the then() callback
- await doesn't work in global scope in NodeJS

```
const one = () => Promise.resolve('One!');

async function asyncTest() {
  const res = await one();
  console.log(res);
  if (res === 'One!') {
    return "Two";
  }
}

// await in global scope works only in browsers,
// in NodeJS await in global scope is experimental
const result = await asyncTest(); // One!
console.log(result); // Two
```

ADD SUGAR

- Async/Await is just syntactic sugar for promises, so you can treat async function like promises

```
const one = () ⇒ Promise.resolve('One!');

async function asyncTest() {
  const res = await one();
  console.log(res);
  if (res === 'One!') {
    return "Two";
  }
}

asyncTest().then(result ⇒ { // One!
  console.log(result); // Two
});
```

ERROR HANDLING

- Async/await allow us to handle errors the same way we do with synchronous code with **try...catch**

```
async function thisThrows() {  
  throw new Error("Thrown from thisThrows()");  
}  
  
async function run() {  
  try {  
    await thisThrows();  
  } catch (e) {  
    console.error(e);  
  } finally {  
    console.log('We do cleanup here');  
  }  
}  
  
run();  
  
// Output:  
// Error: Thrown from thisThrows()  
//   ... stacktrace  
// We do cleanup here
```



softserve

FUTURE