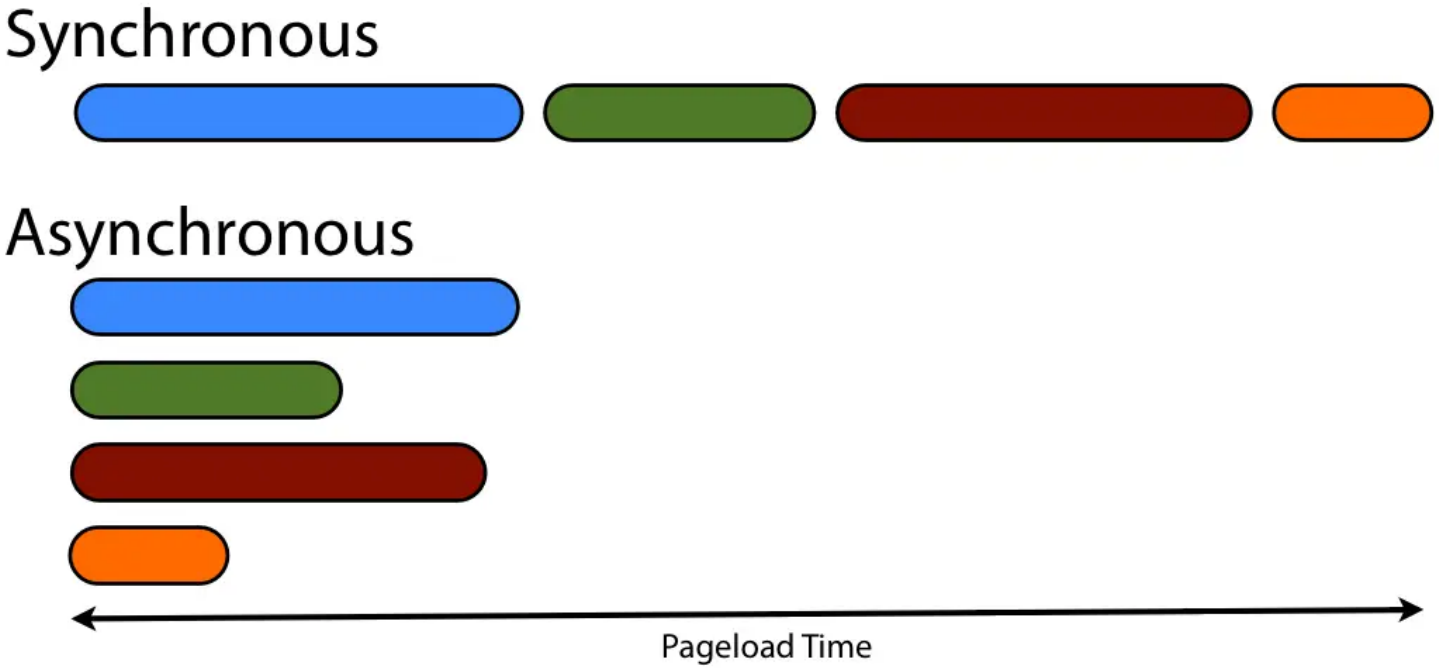
Node.js

Ch 6: Asynchronous Javascript

hogeschool vives

D. HOSTENS & M. DIMA

Synchronous - Asynchronous



— Asynchronous code

- Synchronous = blocking code (wait until ready and then continue)
- Asynchronous = non-blocking code
- We create a new project aan (mkdir + npm init)

```
console.log('Before');
setTimeout(()=> {
    console.log('Simulated DB call...');
}, 2000);
console.log('After');
```

- What do we see in the console? Before - Sim.. -After?
- No! The 'Simulated DB call' will be executed last

```
milan@async-test node index.js
Before
After
Simulated DB call...
```

— asynchronous function setTimeout() analysis

```
setTimeout(
()=> { console.log('Simulated DB call...'); },
 2000);
```

- this function takes two arguments
- argument 1 = function

```
()=> { console.log('Simulated DB call...'); }
```

- argument 2 = a time duration milliseconds - `2000`
- This function can schedule a function to be executed in the future. It will not wait!
- Once scheduled, the remaining code will be executed - `console.log('After')`
- Note that Node default is single threaded = no concurrency
- analogy with restaurant = 1 waiter

- We extract our simulated DB Call to a separate function `getUser()`
- To make it a bit more realistic we pass an user id along as parameter. `function getUser(id){...}`
- This functions returns an object from the DB `return{id: id, gitHubUserName: 'MVives'};`
- When we invoke the getUser(id) function the value will not be written to a variable, at least not immediately
- This would return an `Undefined`. The moment the console.log is executed this value hasn't returned yet!

```
const user = getUser(1); console.log('User',
user);
milan@async-test〽node index.js
Before
User undefined
After
Reading a user from database...
```

— Complete file index.js

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

const user = getUser(1);
console.log('User', user);

console.log('After');

function getUser(id){
setTimeout(()=> {
    console.log('Reading a user from database...');
    return ({id: id, gitHubUsername:
'MilanVives'});
}, 2000);
}
```

— 3 Patterns

- The function we pass along with `setTimeout()` as the argument will only be executed two seconds later!
- The object we are returning will not yet be available the moment we execute `getUser(1)`.
- How can we access our user object?
- **3 Patterns**
  - Callbacks
  - Promises
  - Async-Await (Promises with *syntactic sugar*)

— Pattern 1 callback

- We change the signature of the function getUser
- We also pass a second parameter along, a callback

```
function getUser(id, callback){...}
```

- This callback function will be invoked once the answer is back

```
callback({id: id, giHubUsername: 'MilanVives'});
```

- No we change the call of getUser(id) with a function that will be invoked with the user object as an argument

```
getUser(1, function(user){
    console.log('User', user);
} );
```

```
milan@async-test~node index.js
Before
After
Reading a user from database...
User { id: 1, giHubUsername: 'MilanVives' }
```

— index.js

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

getUser(1, function(user){ // arrow function also ok
    console.log('User', user);
} );

console.log('After');

function getUser(id, callback){
setTimeout(()=> {
    console.log('Reading a user from database...');
    callback({id: id, giHubUsername: 'MilanVives'});
}, 2000);
}
```

Next step

- We create a second function getRepositories that will pick up the users repositories

```
function getRepositories(username){
    return ['repo1', 'repo2', 'repo3'];
}
```

- We simulate an async DB call with `setTimeout()`

```
function getRepositories(username, callback){
    setTimeout(()=> {
        console.log('Calling Github API...');
        callback(['repo1', 'repo2', 'repo3']);
    }, 2000);
}
```

```
milan@async-test〽node index3_callback2.js
Before
After
Reading a user from database...
Calling Github API...
Repos [ 'repo1', 'repo2', 'repo3' ]
milan@async-test〽
```

— index.js

```
console.log('Before');
getUser(1, function(user){
    getRepositories(user.gitHubUsername, (repos) => {
        console.log('Repos', repos);
    });
});
console.log('After');

function getUser(id, callback){
    setTimeout(()=> {
        console.log('Reading a user from database...');
        callback({id: id, giHubUsername: 'MilanVives'});
    }, 2000);
}

function getRepositories(username, callback){
    setTimeout(()=> {
        console.log('Calling Github API...');
        callback(['repo1', 'repo2', 'repo3']);
    }, 2000);
}
```

— Callback hell

- This will result in a nested structure
- Suppose after that we want to get the repo's commits

```
getUser(1, function(user){
    getRepositories(user.gitHubUsername, (repos) => {
        getCommits(repos[0], (commits) => {...})
```

- This is what we call Callback hell (Christmas tree problem)
- This code would look a lot cleaner synchronous

```
const user = getUser(1);
const repos = getRepositories(user.gitHubUsername);
const commits = getCommits(repos[0]);
```

```
console.log('1. Before');
const user = getuser(1, function(user){
    //console.log('User', user);
    getRepositories(user.gitHubUsername, (repos)=>{
        //console.log('Repos: ',repos);
        getCommits(repos[0], (commits)=> {
            console.log('Commits', commits);
        })
    });
});
console.log('3. After');
function getuser(id, callback) {
setTimeout(()=> {
    console.log('2. Reading user from DB...');
    callback({id: id, gitHubUsername: 'MilanVives'});
},2000);
}
function getRepositories(username, callback){
    setTimeout(()=> {
        console.log('4. Calling Github API...');
        callback(['repo1','repo2','repo3']);
    },2000);
}
function getCommits(repo, callback){
            setTimeout(()=> {
            console.log('5. Calling Github API...');
            callback(['commit1','commit2','commit3']);
        },2000);
}
```

— A possible solution: named functions

- We replace the *anonymous functions* by *named functions*
- The second argument of our functions is now an anonymous function
- We start at the deepest level

```
getCommits(repos[0], (commits) => {
    console.log(commits);
})
```

- We create a named function `displayCommits()`

```
function displayCommits(commits){
    console.log(commits);
}
```

- Then we replace the second argument in `getCommits`

```
getCommits(repos[0], displayCommits)
```

- ! Please note there is no `()` at the end of `displayCommits`

Better solution: promises

- Promises = good fit for asynchronous code
- A promise holds the eventual result of an asynchronous operation
- It can either be a value or an error
- Has 3 possible states:
  - Pending state = at creation
  - Fulfilled state = success (value present)
  - Rejected state = error
- We create a new file in our project: `promises.js`

— Promises

- A promise object would typically look like this:

```
const p = new Promise((resolve, reject) => {
 // async code
});
```

- This will result in either a value or an error
- The two parameters resolve and reject are functions
- By invoking those two functions we can either send a result or error to the *consumer* of this promise

```
const p = new Promise((resolve, reject) => {
 // async code
 resolve(1);
});
```

- For an error we send a reject instead of resolve
- Good practice: return error object instead of string

```
reject(new Error('message'));
```

Consuming a promise

- Our promise resolves in 1 hardcoded a.t.m. - `resolve(1)`
- Hoe do we consume this promise?
- promise-object `p` has two methods: `then` and `catch`

```
p.then(result => console.log('Result', result));
```

- We make our code a bit more 'asynchronous'

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(1);
    }, 2000);
});
p.then(result => console.log('Result', result));
```

```
milan@async-test node promises.js
Result 1
```

— Catching Errors: reject() + catch()

```
const p = new Promise((resolve, reject) => {
 // async code
    setTimeout(() => {
    //resolve(1);
    reject(new Error('message'));
    }, 2000);
});
p
.then(result => console.log('Result', result))
.catch(err => console.log('Error',
err.message));
milan@async-test〽node promises.js
Error message
```

- Each error object has a message *property*

— Lifecycle of a promise

- First: Pending- state
- If succes: resolved/fulfilled
- If error: to rejected
- `resolve()` returns an object at succes
- `reject()` returns an error
- Please note: we have to write our own logic to return a success or error
- Good practice: it's better to use promises instead of callbacks

— Replace callbacks by promises

- This callback function

```
function getUser(id, callback){
    setTimeout(()=> {
        console.log('Reading a user from database...');
        callback({id: id, gitHubUsername: 'MilanVives'});
    }, 2000);
}
```

- We change it to a promise

```
function getUser(id){
    return new Promise((resolve, reject) => {
        setTimeout(()=> {
            console.log('Reading a user from database...');
            resolve({id: id, gitHubUsername: 'MilanVives'});
        }, 2000);//async work
    });
}
```

— Consuming a promise

```
getUser(1, function(user){
    getRepositories(user.gitHubUsername, (repos) => {
        getCommits(repos[0], (commits) => {
         console.log('Repos', repos);
        })
    })
});
```

- will be replaced by

```
getUser(1)
    .then(user =>
getRepositories(user.gitHubUsername))
    .then(repos => getCommits(repos[0]))
    .then(commits => console.log('Commits', commits));
```

```
milan@async-test〽node promises.js
Before
After
Reading a user from database...
Calling Github API...
Getting commits...
Commits [ 'commit1', 'commit2', 'commit3' ]
```

— promises.js

```
console.log('Before');
getUser(1)
    .then(user => getRepositories(user.gitHubUsername))
    .then(repos => getCommits(repos[0]))
    .then(commits => console.log('Commits', commits))
    .catch(err => console.log('Error', err.message));
console.log('After');
function getUser(id){
    return new Promise((resolve, reject) => {
        setTimeout(()=> {
            console.log('Reading a user from database...');
            resolve({id: id, gitHubUsername: 'MilanVives'});
        }, 2000);//async work
    });
}
function getRepositories(username){
    return new Promise((resolve, reject) => {
        setTimeout(()=> {
            console.log('Calling Github API...');
            resolve(['repo1', 'repo2', 'repo3']);
        }, 2000);
    })
}
function getCommits(repos){
    return new Promise((resolve, reject)=> {
        setTimeout(()=> {
            console.log('Getting commits...');
            resolve(['commit1', 'commit2', 'commit3']);
        }, 2000);
    })
}
```

— Execute promises concurrently (parallel)

```
const p1 = new Promise((resolve)=> {
    setTimeout(() => {
        console.log('Async operation 1...');
        resolve(1);
    }, 2000);
});
const p2 = new Promise((resolve)=> {
    setTimeout(() => {
        console.log('Async operation 2...');
        resolve(2);
    }, 2000);
});
Promise.all([p1, p2])
    .then(result => console.log(result));
milan@async-test〽node promise-api.js
Async operation 1...
Async operation 2...
[ 1, 2 ]
```

— Promise API

- If one of the promises fails they will all fail
- To only select the first fulfilled promise we change the:

```
Promise.all([p1, p2])
```

- by

```
Promise.race([p1, p2])
```

- Return a fulfilled promise

```
const p = Promise.resolve({id: 1});
p.then(result => console.log(result));
```

- Return a rejected promise

```
const p = Promise.reject(new Error('reason'));
p.catch(error => console.log(error));
```

— Promises syntax

```
const done = true //toggle here false/true

const isItDoneYet = new Promise((resolve, reject) => {
  if (done) {
    const workDone = 'Here is the thing I built'
    resolve(workDone)
  } else {
    const why = 'Still working on something else'
    reject(why)
  }
});
//consume the promise
  isItDoneYet
    .then(ok =>  console.log(ok))
    .catch(err => console.error(err));
```

— Async - Await

- Promises with *syntactic sugar*
- Similar to C#
- We rewrite this code with *Async-Await*

```
getUser(1)
    .then(user =>
getRepositories(user.gitHubUsername))
    .then(repos => getCommits(repos[0]))
    .then(commits => console.log('Commits', commits))
    .catch(err => console.log('Error', err.message));
```

- *becomes*

```
const user = await getUser(1);
const repos = await
getRepositories(user.gitHubUsername);
const commits = await getCommits(repos[0]);
console.log(commits);
```

— Async - Await

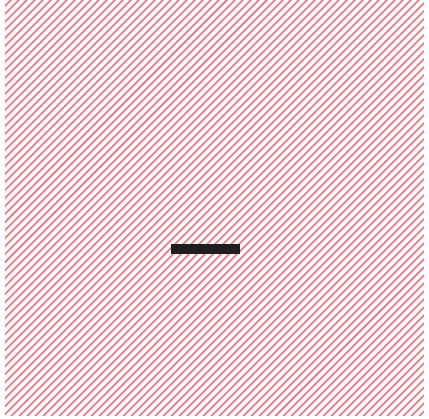
- When using **await** it should also be inside a function decorated with **async**
- We rewrite our code:

```
async function displayCommits() {
    const user = await getUser(1);
    const repos = await getRepositories(user.gitHubUsername);
    const commits = await getCommits(repos[0]);
    console.log(commits);
}
displayCommits();
milan@async-test〽node promise_from_cbh.js
Before
After
Reading a user from database...
Calling Github API...
Getting commits...
[ 'commit1', 'commit2', 'commit3' ]
```

— Catching errors

- Please note we now do not have a catch anymore for errors
- Solution: wrap everything in a try - catch block

```
async function displayCommits() {
    try{
        const user = await getUser(1);
        const repos = await
getRepositories(user.gitHubUsername);
        const commits = await
getCommits(repos[0]);
        console.log(commits);
    }
   catch(err){
       console.log('Error', err);
   }
}
displayCommits();
```



- The complete code from this chapter: https://github.com/MilanVives/NodeLes6.git

- Lab

- Rewrite the callbacks with promises + Async-Await
- Link see Toledo