DISKUSIKAN MATERI
LAPORKAN MATERI

Daftar Modul

Masukkan kata kunci

Persetujuan Hak Cipta
Modul 1: Introduction Course (Gratis)
Prasyarat Kelas (Gratis)
Apa yang Akan Kita Pelajari? (Gratis)
Tools Requirement
Modul 2: ECMAScript 6 (Gratis)
Running Snippet Code (Gratis)
Persiapan Project Latihan (Gratis)

Callback Function

Hal yang membuat bingung ketika kita bekerja dengan synchronous dan asynchronous program adalah bagaimana menangani suatu nilai yang didapatkan secara asynchronous pada program yang berjalan secara synchronous. Contohnya seperti kode berikut:

```
const getCake = () => {
 let cake = null;
 console.log("Sedang membuat kue, silakan tunggu ....")
 setTimeout(() => {
   cake = "Kue Selesai!"
 }, 3000)
 return cake;
}

const cake = getCake();
console.log(cake);

/*output:
Sedang membuat kue, silakan tunggu ....
null
*/
```

Jika kita melakukan hal seperti ini untuk mencetak nilai `cake` yang sesungguhnya, maka hal tersebut tidak akan pernah terjadi. Karena seperti yang sudah kita ketahui fungsi `setTimeout()` tidak akan menghentikan JavaScript untuk mengeksekusi kode yang ada selanjutnya. Jadi fungsi `getCake()` akan selalu mengembalikan nilai `null`, karena kode `return cake` akan dieksekusi terlebih dahulu dibandingkan dengan `cake = "Kue Selesai!"`. Kode asynchronous perlu disusun dengan cara yang berbeda dari synchronous code. Cara paling dasar adalah dengan menggunakan *callback function.*

Apa itu *callback function*? Mari kita bayangkan kembali melalui pandangan dunia nyata. Kita menunggu pesanan kopi datang di meja kita tapi , pada saat itu juga kita tidak bisa berada terus di tempat itu karena ada urusan mendadak. Pada kasus ini mungkin terdapat dua aksi yang bisa kita lakukan:

- (Synchronous) Kita tetap menunggu di meja hingga kopi itu datang dan kemudian meninggalkan kedai kopi.
- (Asynchronous) Kita meminta tolong kepada teman kita untuk menerima kopi itu, dan bertemu nanti untuk memberikan kopinya. Sehingga kita tidak perlu menunggu untuk meninggalkan kedai kopi.

Nah pada JavaScript, teman kita berperan layaknya *callback function.* Ia diperintahkan pada sebuah fungsi asynchronous kemudian akan dipanggil/digunakan ketika tugas itu selesai.

Bagaimana cara melakukannya? Yang pertama kita tambahkan parameter dengan nama callback pada fungsi asynchronous.

```
const getCake = callback => {
 let cake = null;
 console.log("Sedang membuat kue, silakan tunggu ....")
 setTimeout(() => {
   cake = "Kue Selesai!";
 }, 3000)
 return cake;
}
```

Kemudian kita panggil/gunakan `callback` yang diisikan dengan data yang akan dibawa (cake) ketika task selesai dilakukan.

```
setTimeout(function() {
   cake = "Kue Selesai!";
   callback(cake);
 }, 3000)
```

Setelah menggunakan callback, fungsi tidak perlu lagi mengembalikan nilai. Sehingga kita bisa hapus kode `return cake;`. Sehingga keseluruhan fungsi akan tampak seperti ini:

```
const getCake = callback => {
 let cake = null;
 console.log("Sedang membuat kue, silakan tunggu ....")
 setTimeout(() => {
   cake = "Kue Selesai!";
   callback(cake);
 }, 3000)
}
```

Kemudian untuk menggunakan fungsi `getCake`, kita ubah kode dari:

```
const cake = getCake();
console.log(cake);
```

Menjadi:

```
getCake(cake => {
 console.log(cake);
})
```

Sehingga ketika dijalankan akan sesuai dengan harapan kita.

```
const getCake = callback => {
 let cake = null;
 console.log("Sedang membuat kue, silakan tunggu ....")
 setTimeout(() => {
   cake = "Kue Selesai!";
   callback(cake);
 }, 3000)
}

getCake(cake => {
 console.log(cake);
})

/* output:
Sedang membuat kue, silakan tunggu ....
---- setelah 3 detik ----
Kue Selesai!
*/
```

Callback Hell

Kita sudah mengetahui bahwa callback dibutuhkan untuk mendapatkan nilai yang berasal dari *asynchronous function*. Lantas bagaimana jika terdapat proses satu sama lain yang saling bergantungan? Contohnya, untuk membuat kue tahapan yang perlu kita lakukan adalah:

1. Mempersiapkan bahan
2. Membuat adonan
3. Menyiapkan adonan ke cetakan
4. Memanggang adonan

Tahapan tersebut sangat tergantung satu sama lain. Kita tidak bisa memanggang adonan sebelum membuat adonannya, dan kita tidak bisa membuat adonan tanpa mempersiapkan bahannya terlebih dahulu. Jika seluruh tahapan tersebut berjalan secara *synchronous*, mungkin kita bisa melakukanya seperti ini:

```
function makeACake(...rawIngredients) {
 const ingredients = gatheringIngredients(rawIngredients),
 dough = makeTheDough(ingredients),
 pouredDough = pourDough(dough),
 cake = bakeACake(pourDough),
 console.log(cake);
}
```

Namun jika fungsi-fungsi tersebut berjalan secara *asynchronous*, maka kita akan membuat yang namanya *callback hell*. Callback hell terjadi karena banyak sekali callback function yang bersarang karena saling membutuhkan satu sama lain, sehingga kode akan tampak seperti ini:

```
function makeACake(...rawIngredients) {
 gatheringIngredients(rawIngredients, function(ingridients) {
   makeTheDough(ingridients, function(dough) {
     pourDough(dough, function(pouredDough) {
       bakeACake(pouredDough, function(cake) {
         console.log(cake);
       });
     });
   });
 });
}
```

Melihat kode seperti ini saja, kepala jadi pusing. Terbayang sulitnya memelihara kode ini di masa yang akan datang.

Lantas apa solusi agar kita dapat menghindari *callback hell*? Salah satunya adalah dengan menggunakan Promise.

```
function makeACake(...rawIngredients) {
 gatheringIngredients(rawIngredients)
 .then(makeTheDough)
 .then(pourDough)
 .then(bakeACake)
 .then(console.log);
}
```

Dengan Promise, kita dapat meminimalisir callback hell dan mengubahnya menjadi kode yang

sangat mudah dibaca. Bahkan dengan kode seperti itu, non-developer pun dapat mengerti apa maksud dari kode tersebut.

⬅ KEMBALI KE MATERI SEBELUMNYA

LANJUTKAN KE MATERI BERIKUTNYA ➡