**Promise**

Sebelumnya kita sudah belajar cara menggunakan *asynchronous* pada JavaScript dengan *callback* lalu sekarang kita akan mencoba menggunakan ***promise***. *Promise* sendiri adalah salah satu fitur dari ES6 (ES2015) JavaScript. Konsep *promise* hadir untuk memecahkan masalah yang bertele-tele dengan *callback*, semakin banyak kita menggunakan *callback* untuk proses *asynchronous* semakin kompleks dan sulit kode kita untuk dibaca dan dipelihara. Kita juga akan sering menghadapi *callback* di dalam *callback* dan seterusnya. Masalah seperti ini disebut dengan *Callback Hell*.

Contoh dari *callback hell* :

const verifyUser = (username, password, callback) => {

dataBase.verifyUser(username, password, (error, userInfo) => {

if (error) {

callback(error);

} else {

dataBase.getRoles(username, (error, roles) => {

if (error) {

callback(error);

} else {

dataBase.logAccess(username, (error) => {

if (error) {

callback(error);

} else {

callback(null, userInfo, roles);

}

});

}

});

}

});

};

Contoh lain dari *callback hell*:

doFirst(data, function() {

doSecond(data, function() {

dothird(data, function() {

// Callback Hell

})

})

})

**Konsep *Promise***

***Promise*** sesuai dengan artinya adalah janji. Seperti ketika kita berjanji, jika apa yang kita janjikan bisa kita lakukan maka kita harus melakukannya, jika janjinya ada halangan maka kita tidak bisa melakukannya atau jika janji tersebut belum pada waktunya kita juga harus menunggunya.

Contohnya seperti :

Kita berjanji untuk menonton di bioskop dengan teman pada malam minggu besok

Dari janji tersebut kita mendapatkan beberapa poin

* **Batas waktu dilaksanakan** : Malam minggu besok
* **Rencana** : Menonton di bioskop

Lalu bagaimana jika tiba-tiba turun hujan, apakah kita masih akan berangkat untuk menonton di bioskop? Atau jika ada sesuatu terjadi sebelum janji kita berhasil dipenuhi?

Akhirnya kita mengganti dengan rencana lain jika kita tidak jadi menonton di bioskop dengan *streaming* film. Lalu, kita memiliki rencana lagi yang pasti akan kita lakukan baik kita jadi menonton di bioskop atau pun *streaming* film yaitu tidur.

Maka hasilnya rencana kita:

* **Batas waktu dilaksanakan** : Malam minggu besok.
* **Rencana** : Menonton di bioskop.
* **Rencana jika gagal** : *streaming* film.
* **Rencana selanjutnya apapun yang terjadi** : tidur.

Jika disesuaikan dengan analogi *promise* pada JavaScript menjadi

* ***Pending* / tertunda** = Jika kita belum melewati batas waktu dilaksanakan dan belum mengetahui janji tersebut bisa ditepati atau tidak.
* ***Fulfilled* / terpenuhi** = Jika janji berhasil dipenuhi sebelum batas waktu yang ditentukan.
* ***Rejected* / gagal** = Jika janji gagal ditepati karena suatu hal dan kita melakukan rencana lain.
* ***Settled* / terselesaikan** = Jika semua janji sudah selesai terpenuhi kita sudah bebas melakukan hal lainnya.

**3 Status *Promise* di JavaScript**

Analogi dari sebuah *promise* di JavaScript itu sama seperti kita saat mengambil suatu data baik itu dari *database* maupun *Request API*. Akan ada 3 kondisi yaitu data sedang diproses, data berhasil didapatkan, atau data gagal didapatkan.

Pada *promise* analogi di atas bisa diartikan seperti:

1. ***pending***, jika data sedang diproses.
2. ***fulfilled***, jika data telah berhasil didapatkan.
3. ***rejected***, jika data gagal didapatkan.

|  |  |  |  |
| --- | --- | --- | --- |
| **State** | pending | fulfilled | rejected |
| **Result** | undefined | value | error |

**Contoh menggunakan *promise***

let newPromise = new Promise((resolve, reject) => {

if (true) {

// apa yang dilakukan jika promise fulfilled

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise rejected

reject("Gagal");

}

});

Kita bisa membuat sendiri apa yang akan dilakukan pada sebuah *promise*. Di dalam *promise* ada 2 *keyword* yaitu resolve() dan reject().

* resolve(), jika proses berhasil atau *fullfilled*.
* reject(), jika proses gagal atau *rejected*.

**Contoh penggunaan *promise fullfilled***

Untuk *fulfilled* hanya bisa tereksekusi jika kita kondisi berhasil pada saat kita melakukan *async*. Kita *set condition* menjadi true untuk simulasi *fulfilled*.

const condition = true;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

}

});

Untuk bisa mengeksekusi *promise* yang sudah dibuat kita bisa memanggil *promise* tersebut menggunakan .then():

const condition = true;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

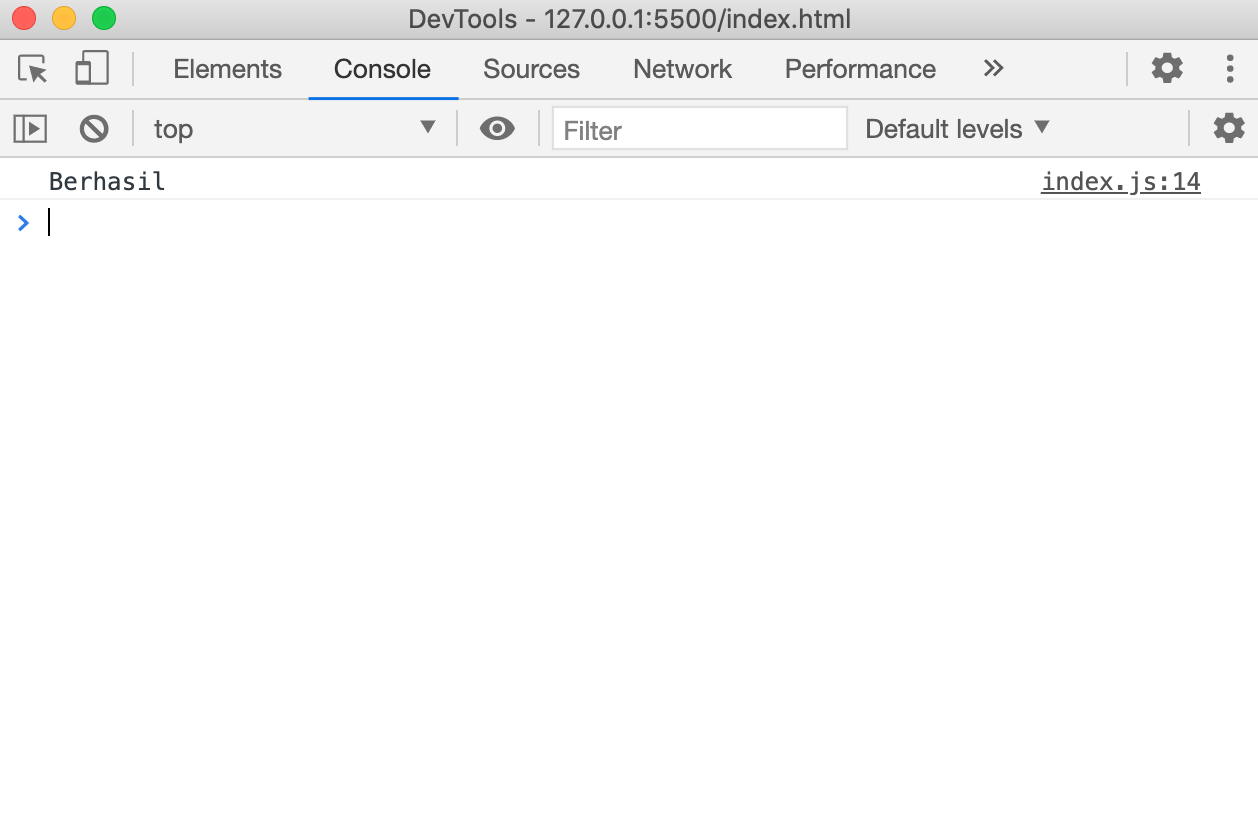
}

});

newPromise.then((result) => {

console.log(result); // Output: "Berhasil"

});



Selain itu kita juga bisa memanggil .then() lebih dari satu kali ketika dibutuhkan seperti contoh di bawah:

const condition = true;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

}

});

newPromise.then((result) => {

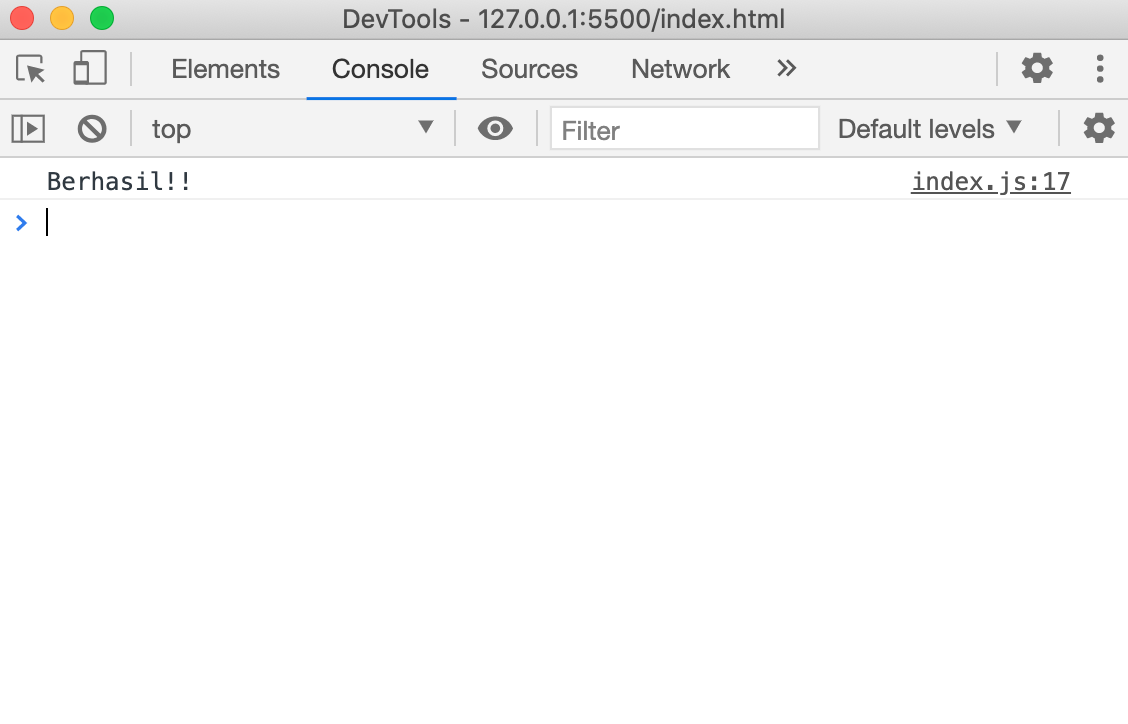
return result;

})

.then((result2) => {

console.log(result2 + "!!"); // Output: Berhasil!!

});



**Contoh penggunaan *promise rejected***

Untuk *rejected* hanya bisa tereksekusi jika kita mengalami *error* pada saat kita melakukan proses *asynchronous*. Kita *set condition* menjadi false untuk simulasi *rejected*.:

const condition = false;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

}

});

Untuk bisa mengantisipasi jika terjadi *error* kita bisa menambahkan .catch() pada *promise*. Sehingga, kita bisa memberi tahu pengguna jika terjadi suatu *error*:

const condition = false;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

}

});

newPromise.then((result) => {

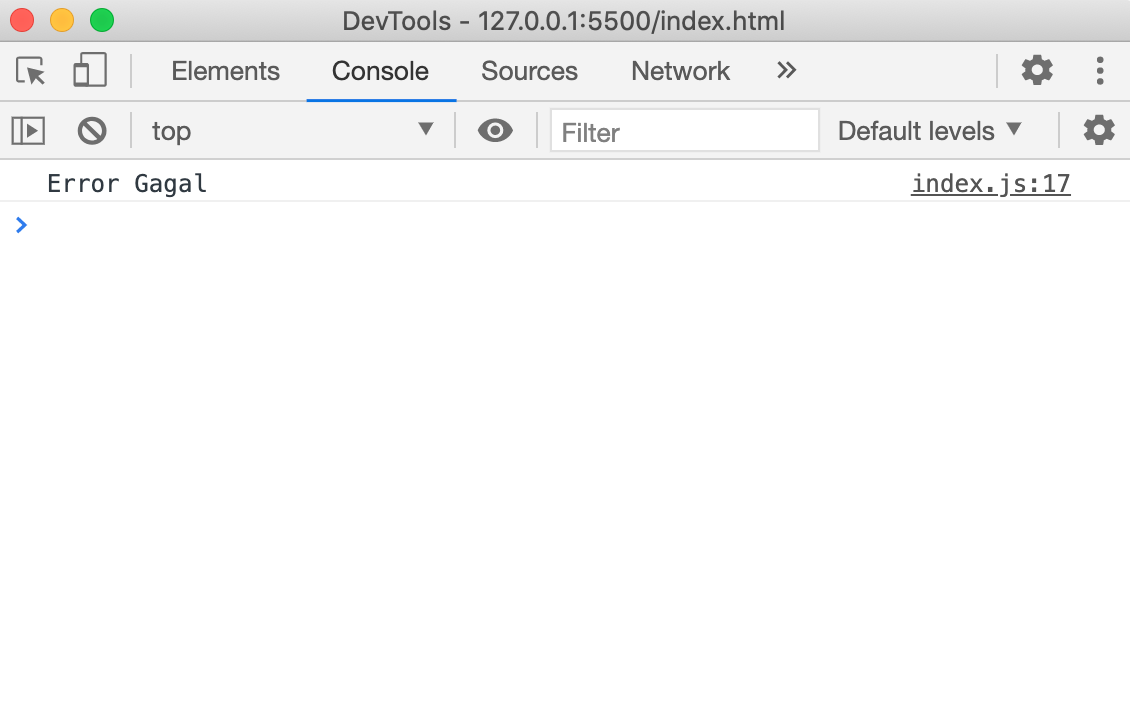
console.log(result);

})

.catch((error) => {

console.log(error.message); // Output: "Error Gagal"

});



Selain .then() dan .catch() dalam *promise* kita juga memiliki .finally() dalam *promise* JavaScript.

.finally() adalah fungsi *callback* yang pasti tereksekusi dalam kondisi apapun (*fullfield* ataupun *rejected*).

const condition = true;

let newPromise = new Promise((resolve, reject) => {

if (condition) {

// apa yang dilakukan jika promise 'fulfilled'

resolve("Berhasil");

} else {

// apa yang dilakukan jika promise 'rejected'

reject(new Error("Error Gagal"));

}

});

newPromise

.then((result) => {

console.log(result); // Output: Berhasil

})

.catch((error) => {

console.log(error);

})

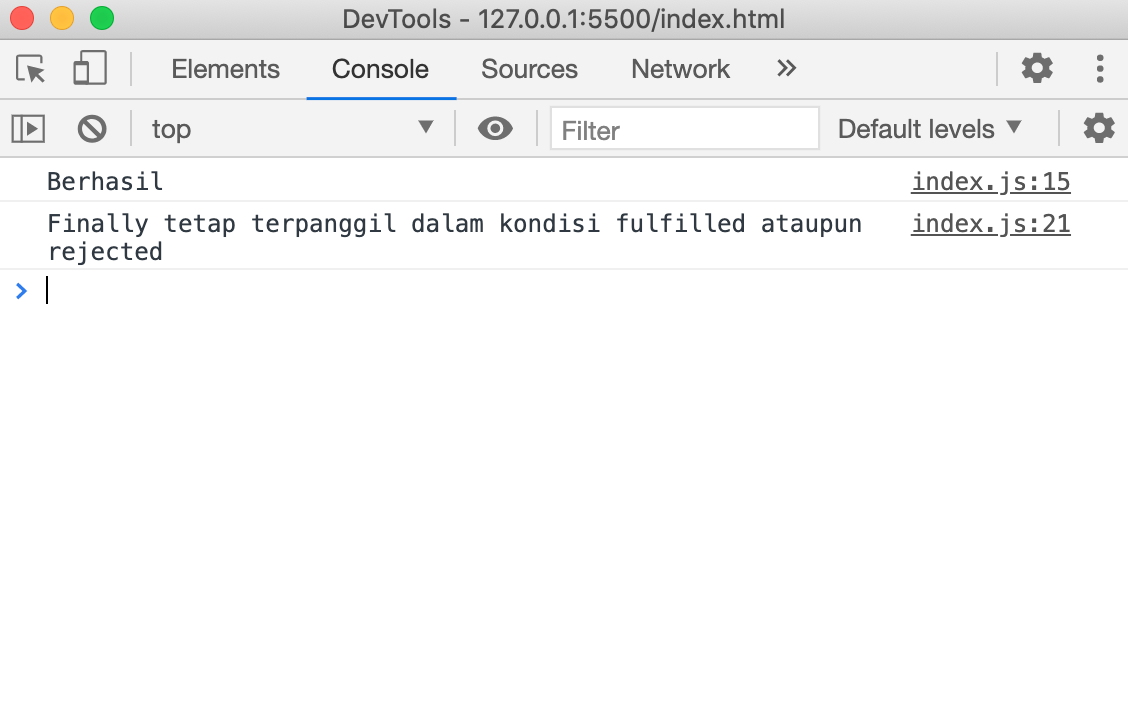
.finally(() => {

console.log(

"Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected"

); // Output: Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected

});



# Promise Instance

Pada topik sebelumnya, kita telah menyinggung .then(), .catch(), dan .finally(). Pada banyak kasus biasanya kita dapat menggunakan promise instance seperti .then(), .catch(), dan .finally() untuk mengantisipasi response saat kita melakukan request ke server.

Contoh berikut penggunaan promise instance untuk proses mengambil data API dari [https://jsonplaceholder.typicode.com/](https://skilvul.com/courses/javascript-intermediate/lessons/javascript-asynchronous/topics/url):

fetch("https://jsonplaceholder.typicode.com/users")

.then((response) => response.json()) // Jika data berhasil didapatkan

.then((json) => console.log(json))

.catch((error) => console.log(error)) // Jika data tidak berhasil didapatkan

.finally(() => {

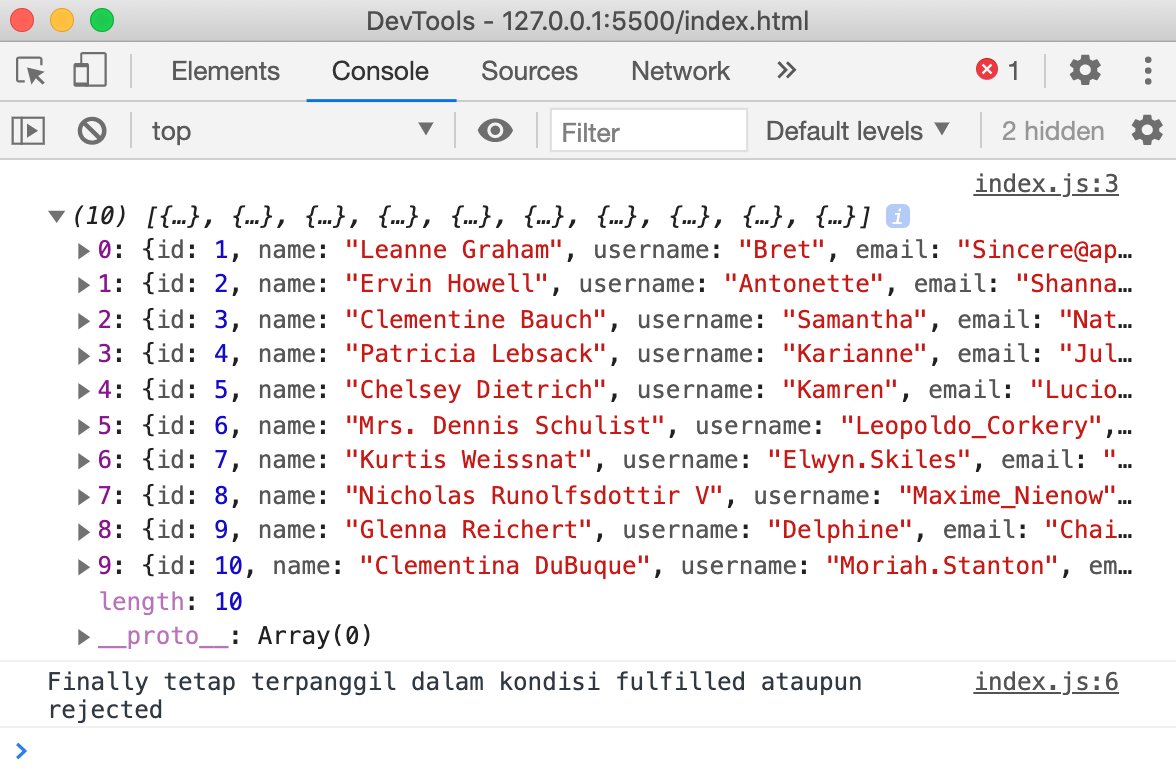
console.log(

"Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected"

); // Finally tetap terpanggil dalam kondisi fulfilled ataupun rejected

});

Jika kode di atas dijalankan, maka akan terlihat tampilan berikut pada console:



Dari kode di atas, kita dapat lihat bahwa **.then() digunakan untuk mengantisipasi keadaan fulfilled**, setelah response didapatkan dan menampilkan data array of object dari server.

Kemudian **.catch() digunakan untuk mengantisipasi keadaan rejected**, jika response gagal didapatkan atau terdapat error lain dari server, lantas apa yang ingin kita lakukan pada aplikasi? Menampilkan page error? Atau menampilkan alert?

Terakhir, **.finally() digunakan untuk mengantisipasi apa yang akan aplikasi lakukan saat response berhasil ataupun gagal didapatkan**. Namun, penggunaan .finally() biasanya jarang dilakukan.