BINAR ACADEMY

Asynchronous Process

Gold - Chapter 6 - Topic 2

Selamat datang di Chapter 6 Topik 2 online course Fullstack Web dari Binar Academy!

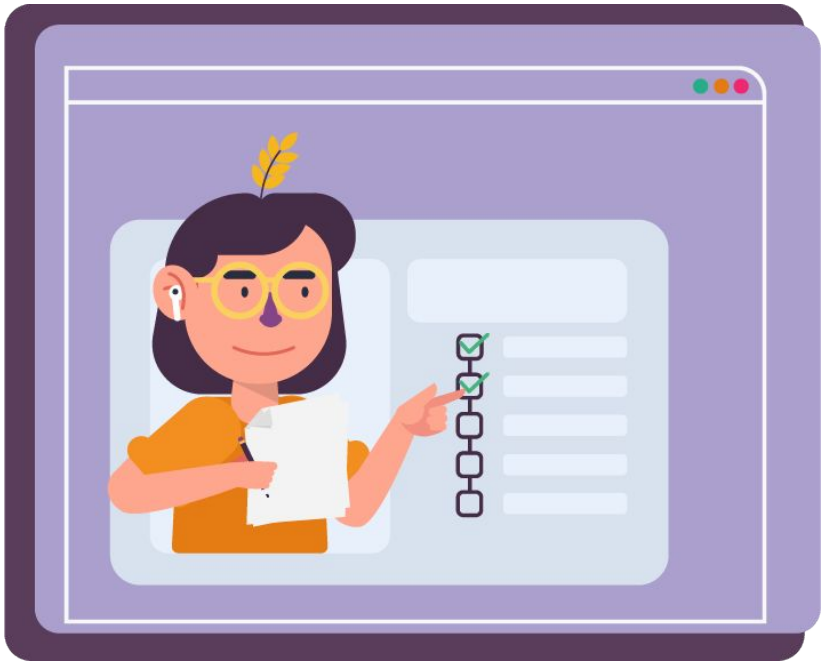
Hai, selamat datang kembali~

Pada topik sebelum ini, kita sudah kupas banyak mengenai design pattern dan macamnya.

Pada topik lanjutan kita akan mempelajari tentang **Asynchronous Process** yang membantu kita supaya bisa berkomunikasi dengan server.

Detailnya, kita bakal bahas hal-hal berikut:

- Memahami asynchronous process
- Mengenal cara penggunaan callback
- Mengenal cara penggunaan promise



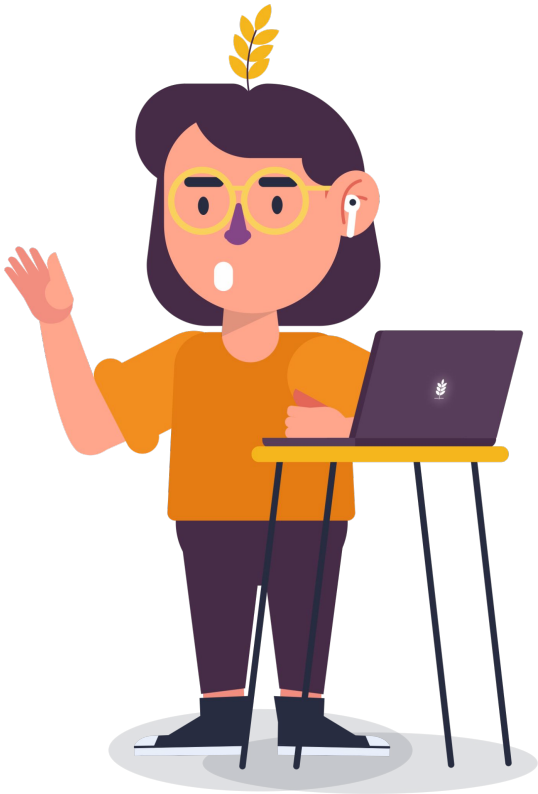
Sekarang kita mau membahas topik yang termasuk fundamental di dunia JavaScript nih, yaitu asynchronous process. Oke langsung aja yuk~

Asynchronous Process

Asynchronous process adalah **salah satu cara mengeksekusi sebuah proses yang mana dalam memprosesnya kita tidak perlu menunggu satu proses selesai untuk melanjutkan proses ke kerjaan lain**.

Buat memahami Asynchronous Process, kita perlu tahu dulu tentang apa itu process. Setelah paham kita akan mempelajari implementasi dari asynchronous process di dalam Javascript, yaitu dengan menggunakan sistem **Callback** dan **Promise**.

Oh iya, biar lancar nih pembahasan kita, pastiin kalian udah paham dasar-dasar Javascript yak :)

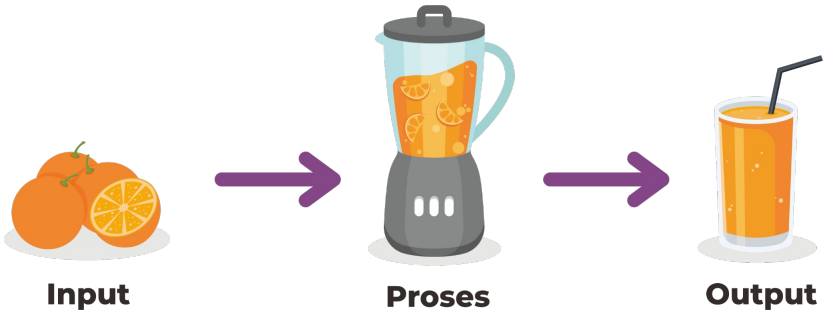


Kita mulai dari Process dulu ya~

Biar lebih gampang dipahami, coba kita bayangin lagi bikin jus buah. Kita akan memasukan buah, air dan gula kemudian diblender dan menjadi sebuah jus buah.

Proses adalah **sebuah aksi yang dilakukan oleh komputer untuk mengolah input menjadi sebuah output**.

Nah, kita korelasikan dengan analogi jus buah. Maka bagian buah, air dan gula adalah inputnya, sedangkan jus buah adalah outputnya! Tapi, jus buah kan ga bisa langsung jadi begitu saja kan ya? Harus ada yang memasukan buah dan mengoperasikan blendernya dong...🤔



Akang jus buah kita panggil : Thread !

Nah, Akang jus buah dalam aktivitas blender ini memiliki peran yang paling vital. Dia berjasa memasukkan buah dan memblender hingga ahkirnya menjadi sebuah jus buah.

Kaitannya dengan proses komputer, orang yang menjalankan proses tersebut biasa kita sebut dengan thread. Jadi, **yang bertanggung jawab terhadap jalannya sebuah proses di dalam komputer adalah thread**.

Mirip dengan proses komputer, aktivitas blender biasanya melibatkan 1 orang saja. Biasanya, 1 proses komputer itu di-*handle* oleh 1 **thread** saja. Karena ga mungkin, 2 orang mencet tombol blender secara bersamaan.

Kemungkinan yang terjadi ketika memblender jus buah :

- Orang tersebut nahan tombol blender, dan nungguin proses blendernya kelar.
- Orang tersebut cabut sehabis mencet tombol blender, dan ngerjain hal lain, dan balik lagi setelah blendernya kelar.

Dari dua kemungkinan tadi kita bisa korelasikan dengan bagaimana Thread menjalankan sebuah proses. Ada yang **synchronous** (Blocking, nungguin selesai baru lanjut), dan ada yang **asynchronous** (Ga blocking, ga nungguin kelar, balik lagi kalo udah kelar).

Yaudah, sekarang kita coba bahas aja apa sih perbedaan mendasar dari dua tipe proses tadi, dan gimana sih cara kerjanya. Lesgooo!

Nah, gimana sekarang pemahaman kalian tentang asynchronous process udah josss kan? Pastinya udah donggg.

Setelah ini kita akan lihat bedanya asynchronous dan synchronous. Skuyyy~

Synchronous vs Asynchronous

Oke, seperti yang dijelasin tadi, ada dua cara menjalankan sebuah proses, yaitu **synchronous** dan **asynchronous**. Kedua cara ini sangat krusial di dalam sebuah aplikasi, jadi kita ga bisa pake synchronous process aja untuk mecahin semua problem, kita juga perlu menggunakan asynchronous process at some point.

Synchronous

Asynchronous

Synchronous Process

Proses ini dijalankan di dalam main thread, yaitu thread yang digunakan untuk menjalankan aplikasi kita. Misal kita menjalankan aplikasi kita menggunakan Thread A, maka si proses synchronous ini akan dijalankan di dalam Thread A juga.

Nah, synchronous process ini juga bisa disebut sebagai **blocking process**. Ketika kita menjalankan proses blocking, thread yang digunakan akan menunggu prosesnya selesai baru mengeksekusi proses lainnya.

Artinya, **thread tersebut ga bisa melakukan hal lain selagi proses yang dia jalankan belum selesai**.

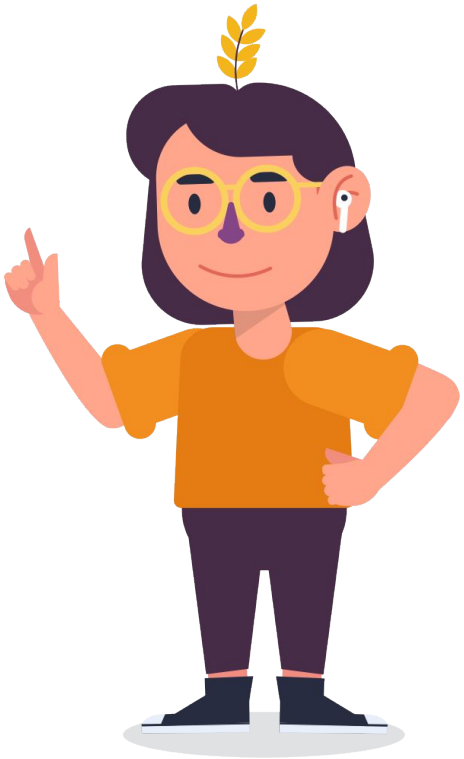
Contoh kode Synchronous di dalam Javascript bisa dilihat di dalam kotak di samping ini:

Bukti kode diatas synchronous adalah `console.log("sequence", sequence);` tidak akan dijalankan kalau `const sequence = generateSequence(100000);` belum selesai dieksekusi, kalian bisa membuktikan dengan menaikkan parameter dari `generateSequence` dengan angka yang sangat besar, output log akan muncul lebih lama sejalan dengan besarnya input fungsi tersebut.

```
function generateSequence(n) {
  const results = [];
  for (let i = 1; i <= n; i++) {
    results.push(i);
  }
  return results;
}

const sequence = generateSequence(100000);
console.log("sequence", sequence);
```

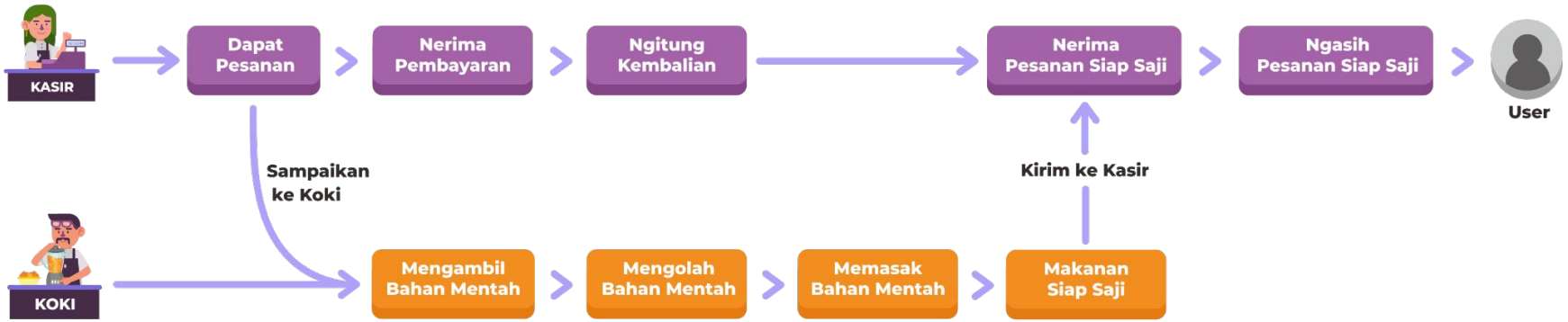
Asynchronous Process

Berbeda dengan synchronous process yang blocking, asynchronous process ini adalah **proses yang tidak blocking**, artinya ketika kita menjalankan proses tersebut, kita tidak menunggu proses itu selesai dieksekusi baru menjalankan proses lain, melainkan kita akan mendelegasikan proses tersebut ke Thread lain, dan ketika sudah selesai, Thread lain akan memberi tahu main Thread kalau proses tersebut sudah selesai.



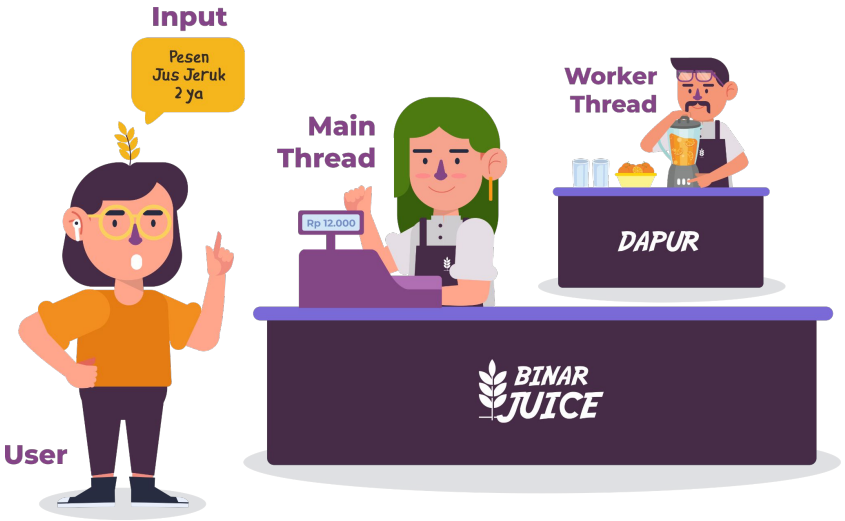
Dari diagram dibawah, dapat kita lihat bahwa ada 2 thread, yaitu Kasir, dan Koki. Kedua thread tersebut bekerja bersamaan dan saling kerja sama. Kasir menerima pesanan dan mendelegasikan pesanan tersebut ke Koki, setelah itu kasir tidak menunggu Koki selesai memasak, melainkan dia langsung mengerjakan proses lain, yaitu menerima pembayaran dari pembeli dan memberikan kembalian.

Setelah pesanan dari pembeli selesai dimasak oleh sang Koki, koki akan memberikan pesanan tersebut ke Kasir, dan kasir meneruskan proses yang dapat dilakukan. Apabila pesanan sudah siap, saatnya memberikannya kepada pembeli. **Koki** dan **Kasir** adalah **Thread yang menjalankan aplikasi bernama Restoran**.



Nah dari ilustrasi barusan, **Kasir adalah Main Thread**. Kasir menerima input dari user dan memberikan feedback ke user. **Koki adalah Worker Thread** yang bekerja apabila ada delegasi dari Main Thread. **Proses yang dijalankan oleh Worker Thread, itu yang kita sebut sebagai Asynchronous Process**.

Proses pendelegasian dari Main Thread ke Worker Thread disebut dengan istilah detaching. Sedangkan proses pendelegasian proses dari Worker Thread ke Main Thread biasa kita sebut sebagai callback.



Contoh kode Asynchronous di dalam Javascript bisa dilihat di dalam kotak di samping ini:

Output `Synchronous: Aku jalan duluan yak!` akan muncul di log paling atas yang menandakan dia jalankan secara synchronous. Karena `generateSequence(10000)` adalah asynchronous process, maka dari itu fungsi tersebut ketika dijalankan, main thread tidak perlu menunggu fungsi tersebut selesai, melainkan main thread akan mendelegasikan fungsi tersebut untuk dieksekusi oleh **Worker Thread**, yang mana ketika fungsi sudah selesai dijalankan, main thread akan menerima callback melalui method `.then` dan akan mengeksekusi kode tersebut setelah tidak ada urusan lagi di main thread yang perlu dijalankan.

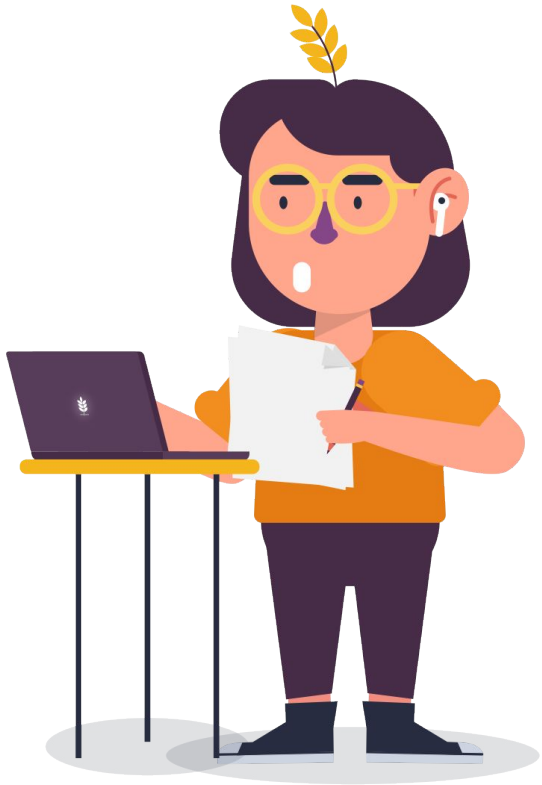
```
async function generateSequence(n) {
  const results = [];

  for (let i = 1; i <= n; i++) results.push(i);

  return results;
}

generateSequence(10000).then((sequence) => {
  console.log("Asynchronous:",
    "Aku jalannya asynchronous" +
    " jadi aku dieksekusi belakangan" +
    " kalo udah gaada urusan di Main Thread."
  );

  console.log("sequence:", sequence);
});

console.log("Synchronous:", "Aku jalan duluan yak!")
```

BINAR

Kalo kamu udah tahu Asynchronous Process dan Synchronous Process sekarang saatnya kita masuk ke pembahasan callback dan promise, hmmm kayak gimana ya kira–kira?

Callback

Callback adalah **delegasi proses dari worker thread ke main thread**. Callback ini sangat umum digunakan di dalam Javascript, yang mana hampir semua standard library di dalam **Node.js** menggunakan callback pattern. Ketika kita ingin menjalankan sebuah fungsi, pasti diminta untuk nambahin callback function untuk menghandle ketika fungsi tersebut selesai dijalankan.



Sebagai contoh, fungsi untuk membaca sebuah file `fs.readFile`

Parameter ketiga dari `fs.readFile` adalah callback function. Fungsi ini akan dipanggil ketika proses membaca file selesai. Proses membaca isi file ini dijalankan secara asynchronous. Artinya, **main thread akan memasang callback yang akan dijalankan ketika proses membaca file selesai**.

Namun, penggunaan callback pattern ini mulai berkurang dan mulai obsolete karena sudah ada sintaks baru di dalam Javascript, yaitu **Promise**.

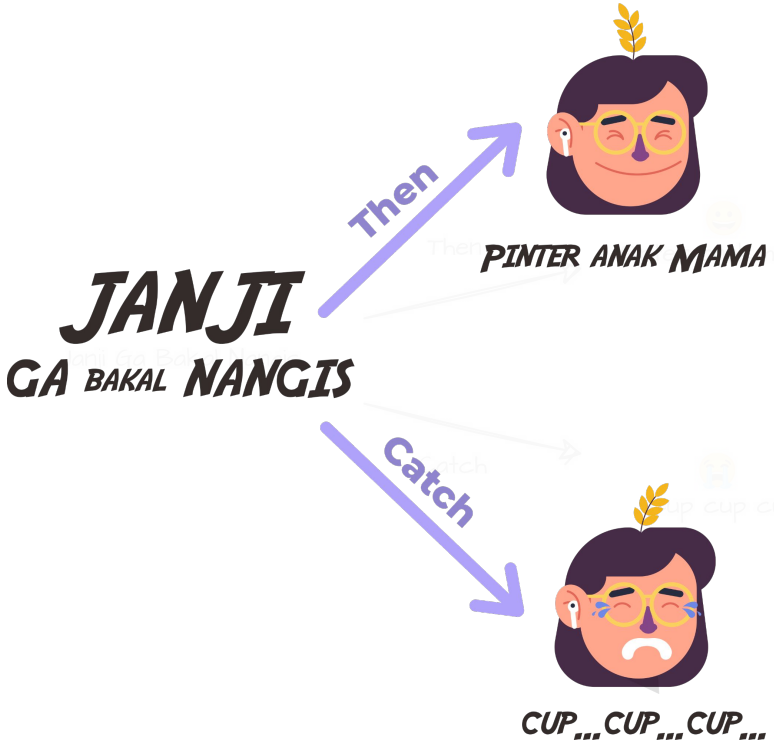
```
const fs = require("fs")

fs.readFile("asynchronous.js", "utf-8", (err, content) => {
  if (!!err) throw err;

  console.log(content);
})
```

BINAR

Promise

Promise adalah sebuah object di dalam Javascript yang mempunyai 2 possible value, yaitu promise tersebut ditepati **(resolved)**, atau promise tersebut diingkari **(rejected)**. Promise ini digunakan untuk mengabstraksi suatu proses yang akan dijalankan secara asynchronous. Dalam proses itu pasti akan ada dua possible value, yaitu berhasil (resolved) atau tidak (rejcected).

Untuk menghandle dua jenis value itu, promise memiliki 2 method, yaitu `.then` dan `.catch` dimana **then** digunakan untuk menghandle apabila promise tersebut terpenuhi, dan sebaliknya, **catch** digunakan untuk menghandle apabila promise tersebut gagal.



Kalian sebenernya udah pernah implementasi Promise sebelumnya, yaitu ketika kalian menggunakan sequelize. Model yang digenerate oleh sequelize **memiliki method create, findAll, findByPk, destroy, update**. Setiap method tersebut mengembalikan sebuah Promise. Maka dari itu setiap kali kalian memanggil method tersebut, pasti kalian akan menambahkan dua method untuk menghandle hasil dari promise tersebut.

Promise ini tetep pake yang namanya callback, bedanya callbacknya ini dipasang pada method yang specific. Jadi, ada 2 fungsi yang dijalankan untuk 2 kasus berbeda, yaitu rejected dan resolved.

Nah, karena Promise masih ngeutilize callback, maka dari itu, kalo kita ingin menjalankan Promise lagi ketika kita selesai menjalankan suatu Promise, akan muncul yang namanya callback Hell 🔥.

```
Post.create({ title: "Hello World", body: "Lorem ipsum dolor sit amet" })
  .then((post) => {
    console.log("Horay, post berhasil dibuat!");
    console.log(post);
  })
  .catch((err) => {
    console.log("Yah, gagal membuat post :(");
    console.error(err);
  })
```

Tuh, bisa kalian lihat sendiri, dengan memanggil promise setelah promise, menyebabkan kode jadi ga readable~

Nah, maka dari itu, ada solusi lain, kita tetep bisa pake `.then` dan `.catch` bedanya di setiap callback, kita return Promise selanjutnya yang ingin kita eksekusi. Jadinya kurang lebih akan seperti ini:

```
User.create({ username: "Sabrina", "password": "xxxxxx" })
  .then((user) => {
    Profile.create({ userId: user.id, fullname: "Sabrina Kaczynski" })
      .then((profile) => {
        console.log("OK", profile);
      })
      .catch((err) => {
        console.error("FAIL", err.message);
      })
  })
  .catch((err) => {
    console.error("FAIL", err.message);
  })
```

Kode di samping sebenarnya dari segi implementasi sama aja, bedanya ia lebih concise dan ga terlalu menjorok ke kanan. Tapi tetep aja, kode yang bagus tuh kode yang ga terlalu banyak menjorok ke kanan.

Nah, untungnya, di Javascript sekarang udah ada sintaks baru, yaitu **async / await**, yang mana sintaks ini akan mempermudah kita dalam menjalankan Promise.

```
User.create({ username: "Sabrina", "password": "xxxxxx" })
  .then((user) => {
    return Profile.create({
      userId: user.id,
      fullname: "Sabrina Kaczynski"
    })
  })
  .then((profile) => {
    console.log("OK", profile);
  })
  .catch((err) => {
    console.error("FAIL", err.message);
  })
```

Async / Await

Async / await adalah ***syntactic sugar*** **yang mempermudah kita dalam mengeksekusi dan memudahkan dalam membuat sebuah Promise**. Sintaks ini hanyalah berupa keyword async dan await, dimana async kita taruh sebelum definisi fungsi, dan await ditaruh di depan eksekusi Promise.

Jauh lebih ringkas bukan? Tapi dari kode di samping ada yang ganjal, yaitu cara handle rejected promise.

```
async function handleCreateTask(req, res) {
  const post = await Task.create({ name: "Belajar Javascript" });
  res.status(200).json(post);
}
```

Promise : Asyn / Await

Untuk menghandle rejected promise, kita bisa menggunakan **try catch** statement.

```
async function handleCreateTask(req, res) {
  try {
    const post = await Task.create({ name: "Belajar Javascript" });
    res.status(200).json(post);
  }

  catch(err) {
    res.status(422).json(err)
  }
}
```

Nah, sekarang kita coba refactor code kita yang ada 2 promise tadi. Kalo kita refactor kurang lebih akan jauh lebih ringkas seperti ini:

```
async function handleCreateUser(req, res) {
  try {
    const user = await User.create({
      username: "Sabrina",
      password: "xxxxxx",
    })
    const profile = await Profile.create({
      userId: user.id,
      fullname: "Sabrina Kaczynski"
    })

    res.status(201).json(profile)
  }

  catch(err) {
    res.status(422).json(err)
  }
}
```

Oh iya, setiap fungsi yang kita definisikan sebagai **async function akan otomatis menjadi sebuah Promise**. Jadi, kita bisa menggunakan async / await untuk membuat Promise juga. Dimana return dari function tersebut akan menjadi resolved value dari Promise dan error yang terjadi di dalam fungsi tersebut akan menjadi rejected value.

Ada batasan juga untuk sintaks async / await ini, kita tidak bisa menggunakan keyword await secara global, kita hanya bisa menggunakan keyword tersebut di dalam async function saja.

```
async function generateSequence(n) {
  if (typeof n != "number") throw new Error("n must be a number");

  const results = [];

  for (let i = 1; i <= n; i++) results.push(i);

  return results;
}

generateSequence(10000)
  .then(console.log) // [1,2,3...1000]
  .catch(console.error);

generateSequence("Aku bukan angka")
  .then(console.log)
  .catch(console.error); // n must be a number
```

BINAR

Mantap! Sekarang kalian udah tahu gimana caranya bikin asynchronous process di dalam kode kalian, ini bisa diterapin untuk mengimprove performa dari aplikasi kalian.

Referensi buat tambahan kamu belajar ~

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Terima Kasih!

Next Topic

loading...

BINAR