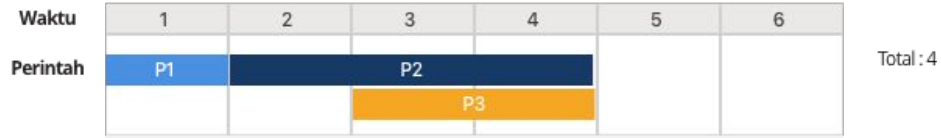
Asynchronous Programming di JavaScript

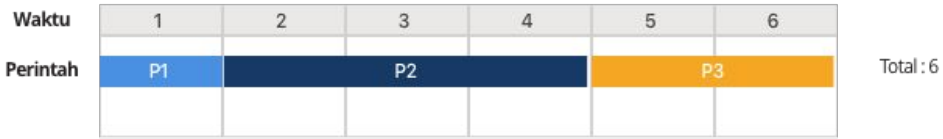
Asynchronous programming / non-blocking I/O ?

- a form of input/output processing that permits other processing to continue before the transmission has finished.
- Synchronous execution usually refers to code executing in sequence.
- Asynchronous execution refers to execution that doesn't run in the sequence it appears in the code.
- the results of execution or output are not always based on code sequence, but based on processing time
- Cases : event, timer, request ajax, listener, interaksi user,... etc.

Asynchronous



Synchronous



Simulasi syncrounous dan Asyncrounous

```
function alertA(){
        console.log(‘alertA’);
        console.log(1);
}
function alertB(){
        console.log(‘alertB’);
        console.log(2);
}
function alertC(){
        console.log(‘alertC’);
        console.log(3);
}
alertA();alertB();alertC()
```

```
function alertA(){
        console.log(‘alertA’);
        console.log(1);
}
function alertB(){
        console.log(‘alertB’);
        setTimeout(() => console.log(2), 0);
}
function alertC(){
        console.log(‘alertC’);
        console.log(3);
}
alertA();alertB();alertC()
```

Blocking dan Non-Blocking di nodejs.org

- Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes.
- Synchronous methods in the Node.js standard library that use libuv are the most commonly used blocking operations.
- All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with Sync.

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); //
blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

Berbagai teknik pemrograman di JavaScript untuk asynchronous programming

- Callback
- Promise
- Async/await

Callback

A callback function is called at the completion of a given task

```
// Callback sebagai Event Listener
document.getElementById("my_button").
addEventListener("click",function(){
   alert('Ouhh aku di klik!')
})
```

```
// Callback sebagai Injeksi sebuah function
function calculate(param1,param2,callback){
  //default operation
  result = param1 + param2

  // callback is function ?
  if (typeof callback == 'function'){
   result= callback(param1,param2)
  }

  return result
}

//execute
a=calculate(2000,4000, function(x,y){return "$ " + (x +
y) })
b=calculate(7000,2000, function(x,y){return "Rp " + (x
* y) })
console.log(a) // $ 6000
console.log(b) // $ 14000
```

Callback

- Callback Pada Asynchronous

```
function p1() {
  console.log('p1 done')
}
function p2() {
  //setTimeout or delay for asynchronous
simulation
  setTimeout(
      function() {
        console.log('p2 done')
      },100
  )
}
function p3() {
  console.log('p3 done')
}
p1()
p2()
p3()
```

```
function p1() {
 console.log('p1 done')
}

function p2(callback) {
 setTimeout(
  function() {
   console.log('p2 done')
    callback()
  },100
  )
}

function p3() {
  console.log('p3 done')
}
p1()
p2(p3)
```

Callback Hell

karena readFileContent( ) adalah proses asynchronous maka di kelola dengan callback seperti berikut :

```
var a = readFileContent("a.md");
var b = readFileContent("b.md");
var c = readFileContent("c.md");
writeFileContent("result.md", a + b + c);
console.log("we are done");
```

```
readFileContent("a.md", function (a){
  readFileContent("b.md", function (b){
    readFileContent("b.md", function (b){
      writeFileContent("result.md", a + b + c,
function(){
          console.log("we are done");
      })
    })
  })
})
```

?
Tidak ada yg salah

Promise

- Spesifikasi Promise

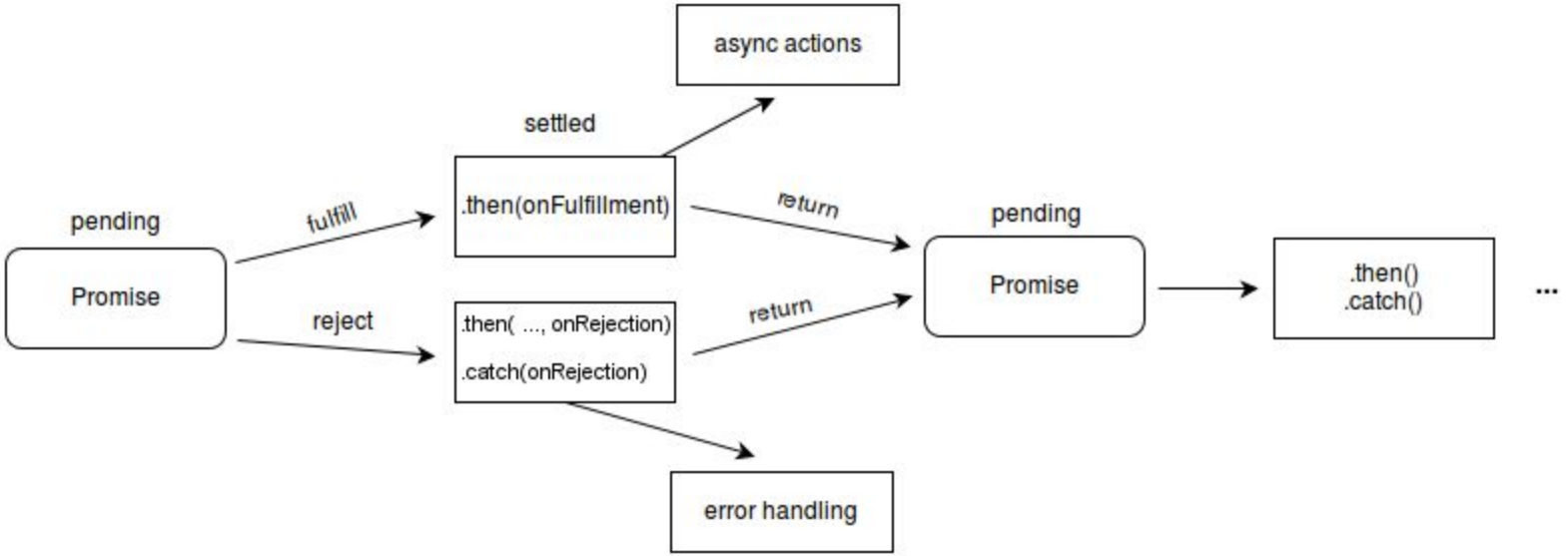
- A promise represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its then method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

- Promise State : pending, fulfilled, rejected
- Consumers: then, catch, finally
- Syntax : New Promise()

```
var janjian = new Promise(
  (resolve,reject) => {
  // salah satu dari 2 callback berikut
  // resolve('Berhasil')
  // reject(new Error('Janji di batalkan'))
  }
)
```

```
janjian
  .then((result) => { console.log (result)})
  .catch((error) => { console.log (error)}
)
```

Promise Flow



The Guarantees of Asyncronous Promise Style

Unlike old-fashioned passed-in callbacks, a promise comes with some guarantees:

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with then(), as above, will be called even after the success or failure of the asynchronous operation.
- Multiple callbacks may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted.

Ref : developer.mozilla.org

Comparing Asyncrounous Callback style vs Promise Style

```
function successCallback(result) {
console.log("Audio file ready at URL: "
+ result); } function
failureCallback(error) {
console.error("Error generating audio
file: " + error); }
```

```
// old Style

createAudioFileAsync(audioSettings,
successCallback, failureCallback);

// Promise Style call

createAudioFileAsync(audioSettings).the
n(successCallback, failureCallback);
```

Comparing Asyncrounous callback style vs Promise Style

- Consumers: then, catch, finally

```
doSomething(function(result) {
  doSomethingElse(result,
function(newResult) {
    doThirdThing(newResult,
function(finalResult) {
      console.log('Got the final result: '
+ finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
doSomething()
.then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: '
+ finalResult);
})
.catch(failureCallback);
```

Promise Chain

```
const getPost = () => fetch('https://jsonplaceholder.typicode.com/posts/1')
const getAuthor = (id) => fetch('https://jsonplaceholder.typicode.com/users/' + id)
const getComment = (id) => fetch('https://jsonplaceholder.typicode.com/users/' + id)

getPost() // #1.fetch post
.then(postResponse => postResponse.json()) // #2. get & return post json
.then(postResponse => getAuthor(postResponse.id) // #3. fetch author
  .then(authorResponse => authorResponse.json()  // #4 get & return author json
    .then(authorResponse => getComment(postResponse.id) // #5 fetch comment
      .then(commentResponse => commentResponse.json()) // #6 get & return comment json
      .then(commentResponse => {  // #7 time to combine all results
          return ({postResponse,authorResponse,commentResponse}) // #8 combine & return all reponses
        })
    )
  )
  .then(results => { // #9 read all responses
    console.log(results.postResponse)
    console.log(results.authorResponse)
    console.log(results.commentResponse)

  })
)
.catch(error => console.log(error)) //# 10 error handling
```

Promise.all

- waits for all promise executions to finish and returns an array of output

```
const getPost = () => fetch('https://jsonplaceholder.typicode.com/posts/1')
const getAuthor = (id) => fetch('https://jsonplaceholder.typicode.com/users/' + id)
const getComment = (id) => fetch('https://jsonplaceholder.typicode.com/users/' + id)

var a = getPost().then(res => res.json())
var b = a.then(res => getAuthor(res.id)).then(res => res.json())
var c = a.then(res => getComment(res.id)).then(res => res.json())
Promise.all([a,b,c]).then(res => console.log(res))
```

async/await di JavaScript (ES2017)

- async → change the function to asynchronous
- await → suspends execution until the asynchronous process completes

PROMISED

```
function fetchWithPromise (id)  {
  fetch(endpoint + id)
  .then(response => {
    return response.json();
  })
  .then(response => {
    console.log(response);
  })
}
```

ASYNC/AWAIT

```
async function fetchWithAsyncAwait (id)  {
  let response = await fetch(endpoint + id)
  response = await response.json()
  console.log(response)
}
```

Async/await Error Handing

```
async function fetchWithAsyncAwait (id)  {
  try {
    let response = await fetch(endpoint + id)
    response = await response.json()
    console.log(response)
  } catch (error) {
    console.log('opps' + error)
  }
}
```

Async/await Serial or Parallel ?

```
const firstPromise= () => (new Promise((resolve,reject) => {
  setTimeout(() =>{ resolve('first Promise')},1000)
}))

const secondPromise = () => ( new Promise((resolve,reject) =>{
  setTimeout(() =>{ resolve('second Promise')},1000)
}))

const thirdPromise = () => ( new Promise((resolve,reject) =>{
  setTimeout(() =>{ resolve('third Promise')},1000)
}))

async function asyncParalel() {
  let a =firstPromise()
  let b= secondPromise()
  let c= thirdPromise()
  console.log('done')
}

async function asyncSerial() {
   let a= await firstPromise()
   let b= await secondPromise()
   let c= await thirdPromise()
   console.log('done')
}
```

paralel (1000 seconds)

serial (3000 seconds)

Generator Function

- Regular functions return only one, single value (or nothing).
- Generators can return (“yield”) multiple values, one after another, on-demand. They work great with iterables, allowing to create data streams with ease.
- Syntax : function* f(...) or function *f(...)?

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
// "generator function" creates "generator object"
let generator = generateSequence();
alert(generator); // [object Generator]

let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}
```

```
let two = generator.next();
alert(JSON.stringify(two)); // {value: 2, done: false}

let three = generator.next();
alert(JSON.stringify(three)); // {value: 3, done:
true}
```

Generators are iterable

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2
}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2, then 3
}
```

Using generators for iterables

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // a shorthand for
[Symbol.iterator]: function*()
    for(let value = this.from; value <= this.to;
value++) {
      yield value;
    }
  }
};

alert( [...range] ); // 1,2,3,4,5
```

Generator composition

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

“ Generators were similar to iterable objects, with a special syntax to generate values. But in fact they are much more powerful and flexible.

That’s because yield is a two-way street: it not only returns the result to the outside, but also can pass the value inside the generator. “

generator.throw

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)
    alert("The execution does not reach here, because the exception is
thrown above");
  } catch(e) {
    alert(e); // shows the error
  }
}

let generator = gen();
let question = generator.next().value;
generator.throw(new Error("The answer is not found in my database")); //
(2)
```

“Before software can be reusable it first has to be usable.” ~

Ralph Johnson ~

Don't waste your life without praying …….

Let get started