BELAJAR DENGAN JENIUS
AWS & NODE.JS

THEORY + VISUALIZATION + CODE

550+ Pages

GUN GUN FEBRIANZA

“Menulis sebagai Janji Bakti, menuju tanah Air Jaya Sakti”

Gun Gun Febrianza

Bandung, 18 Maret 2020

Open Library Indonesia

Sebuah konsep Perpustakaan Digital Terbuka untuk membantu mempermudah siapapun untuk mengakses ilmu pengetahuan. OpenLibrary.id adalah sebuah gerakan dan konsep pemikiran yang penulis usung sebagai wadah tempat untuk mengabdi kepada masyarakat melalui kontribusi karya tulis. Karya tulis yang diharapkan dapat membantu agar minat baca jutaan pemuda-pemudi di Indonesia terus meningkat. Sebab penulis percaya dengan membaca peluang keberhasilan hidup seseorang kedepanya akan menjadi lebih besar dan membaca dapat membawa kita ketempat yang tidak pernah kita sangka-sangka yaitu tempat yang lebih baik dari sebelumnya.

Penulis sadar gerakan ini memerlukan penulis-penulis lainya agar tujuanya bisa tercapai dan jangkauan manfaatnya bisa lebih luas lagi. Semakin banyak penulis dari berbagai bidang keilmuan akan semakin berwarna manfaat hasil karya tulis yang bisa diberikan untuk masyarakat. Maka dari itu penulis secara terbuka mengundang siapapun yang ingin bergabung menjadi penulis di gerakan *Indonesia Open Library,* agar bisa bertemu dan saling bersilaturahmi.

Orang boleh pandai setinggi langit, tapi selama ia tidak menulis maka ia akan hilang dalam masyarakat dan sejarah

- Pramoedya Ananta Toer –

Untuk teman-teman ku, rekan-rekan sebangsaku, apapun kepercayaan kalian, penulis meminta doa dari rekan-rekan supaya selalu diberi kesehatan, keselamatan dan keberkahan dalam hidup.

Agar tetap bisa menulis dan berkarya bersama sama.

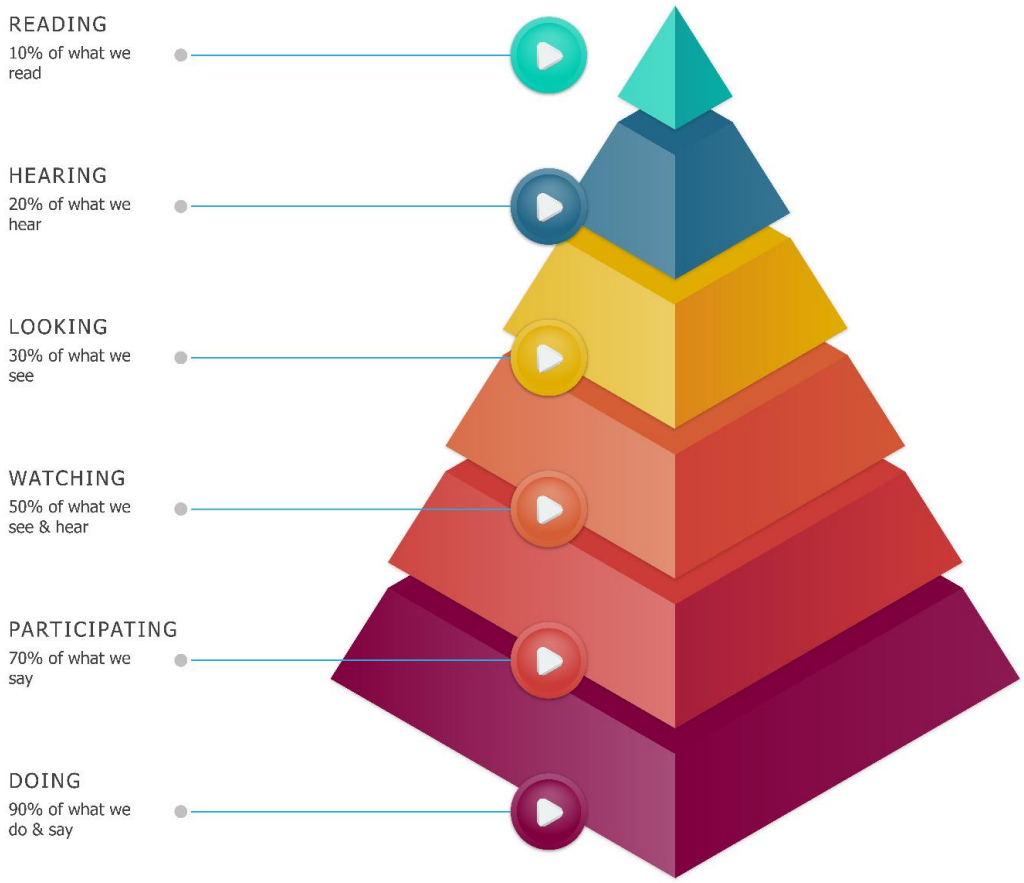
Dari Penulis, yang kelak bercita-cita ingin menjadi seorang guru.

Metode Belajar

Ada satu hal yang harus anda ketahui, jika ingin membaca buku ini anda harus **siap untuk sulit atau menikmati proses belajar** yang akan anda lakukan. Pribadi yang tangguh tidak lahir dari tempat belajar yang serba mudah. Sesungguhnya gagalnya mempelajari ilmu karena kita memusuhinya.

Seperti yang dikatakan Imam Syafii, jika **seumur hidup** kita tidak ingin merasakan hinanya kebodohan maka kita harus merasakan **pahitnya** pendidikan. (Belajar dan Menuntut Ilmu).

Penekanan ini ditegaskan lagi oleh Sayyidina Ali bin abu thalib, "*Knowledge is not attained in comfort*" yang artinya bahwa ilmu pengetahuan tidak akan bisa didapatkan melalui kenyamanan.



The Cone of Learning

Membahas tentang metode belajar ada konsep menarik yang disebut dengan ***The Pyramid of Learning***, diciptakan oleh seorang pendidik di Amerika bernama **Professor Edgar Dale** pada tahun 1946. Saat itu beliau memberi nama metode belajarnya dengan sebutan ***The Cone of Experience*** dimasa kini lebih dikenal dengan sebutan ***The Cone of Learning***.

Ada beberapa tips dari penulis semoga membantu :

1. **Siapkan Buku Catatan Fisik / Digital.**

   Jika dilihat pada *The Cone of Learning* membaca memberi kita ingatan yang sedikit, maka dari itu kita harus mencatatnya.

   "*Ikatlah ilmu dengan tulisan*"

   I don't know what I think until I write it down." — Joan Didion

2. **Cari teman yang tertarik mendengarkan pengetahuan barumu.**

   Disini terdapat fenomena menarik saat kita mengajarkan atau berbagi ilmu yang kita ketahui, salah satunya adalah **The Explanation Effect**. Anda akan memahami kajian dan permasalahan lebih baik.

   "While we teach, we learn." — Seneca

   *"No one learns as much about a subject as one who is **forced to teach it**."*

   — Peter Drucker

3. **Bangun karya-karya kecil yang bisa di bagi agar kamu bersemangat.**

   Setiap karya akan menimbulkan pujian, saran dan kritik, darinya kita dapat melihat kekurangan kita lebih baik lagi dan mengasah dimana keunggulan kita. Dari sini kita akan belajar berpikir kritis setelah dihujani pujian, saran dan kritik. Dari sini kita akan memiliki *driver* yang akan terus mendorong kita setelah dihujani pujian, saran dan kritik.
   Sebuah *driver* yang akan mendorong anda untuk terus maju hingga menjadi sebuah **habit**. Sesuatu yang sudah anda miliki tanpa perlu anda sadari.

   "Set your life on fire. Seek those who fan your flames"

   — Rumi

Learning Problems & Abstraction Control

Ada pepatah menarik dari seorang ilmuwan dan teologian persia :

The Art of Knowing is Knowing what to ignore – Jalāl ad-Dīn Muhammad Rūmī

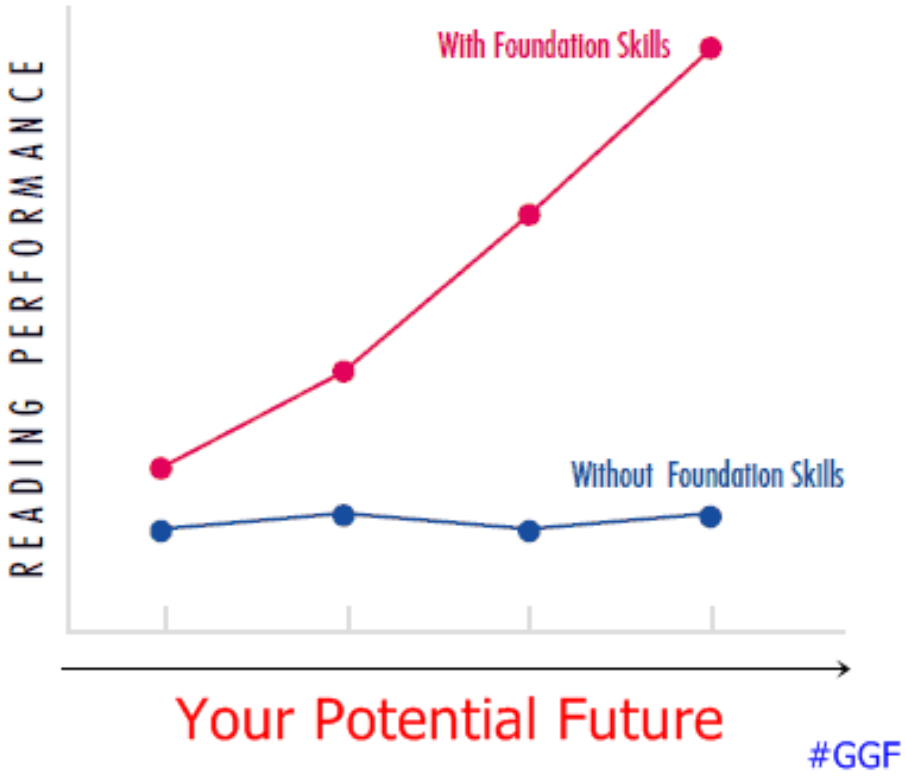
Saat belajar anda harus memahami konsep ***Abstraction Control***. Sebuah daya tahan yang harus anda miliki untuk tetap fokus pada hal-hal yang relevan dan penting untuk anda.

Abaikan atau catat (tandai saja) secara singkat hal hal yang tidak relevan, agar bisa memudahkan proses belajar anda kedepanya ketika waktunya sudah tepat. Fokuslah pada materi yang ingin anda dapatkan pemahamannya, *you only get what you focus on*. Sebuah pemahaman yang anda inginkan, sebuah pemahaman yang bisa anda capai **sedikit demi sedikit.**

"The secret of getting ahead is getting started. The secret of getting started is breaking your complex overwhelming task into ***small management task****, and starting on the first one."*

– Mark Twain

Matthew Effect in Reading



Bagi penulis membaca adalah ladang segala keberuntungan sebab ia membawa kehidupan kita ke arah yang tidak disangka-sangka dan tidak diduga-duga, sebuah kehidupan yang lebih baik dari sebelumnya.

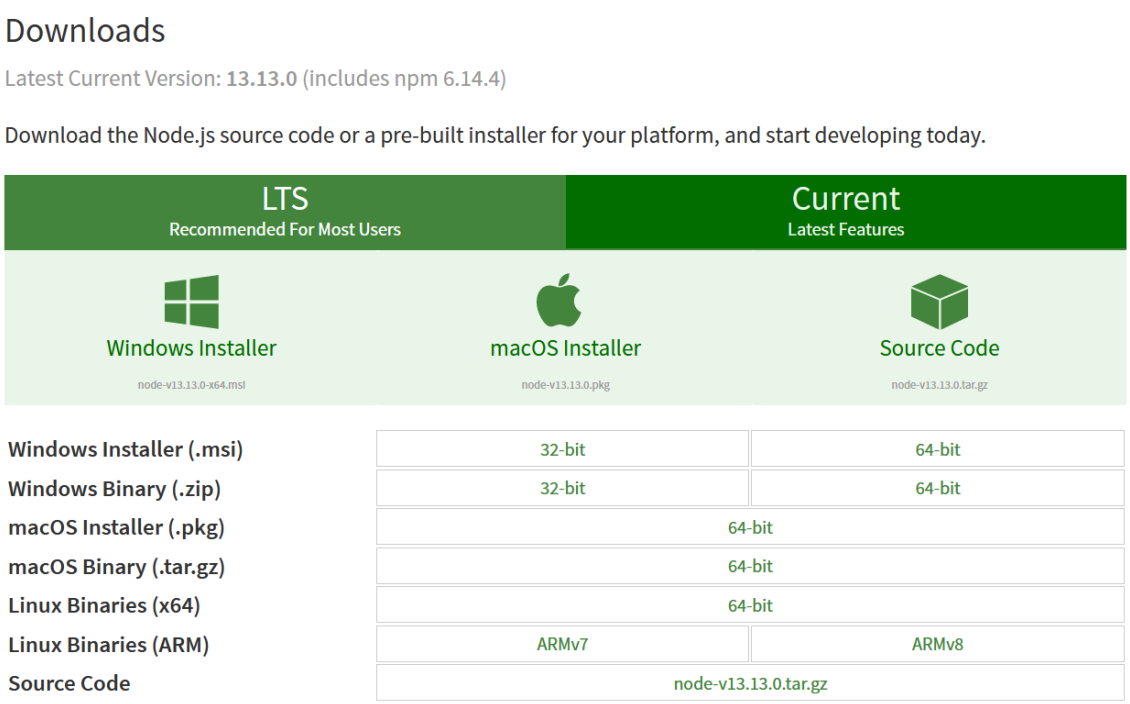
Memperbanyak membaca, memahami dan praktis artinya memperbaiki kehidupan yang hendak kita miliki. Kenapa bisa? Karena kita sedang membantun *Intellectual Capital* yang akan membawa kita untuk memiliki derajat hidup yang lebih baik.

Persiapan

Apa saja yang harus dipersiapkan?

Install Node.js

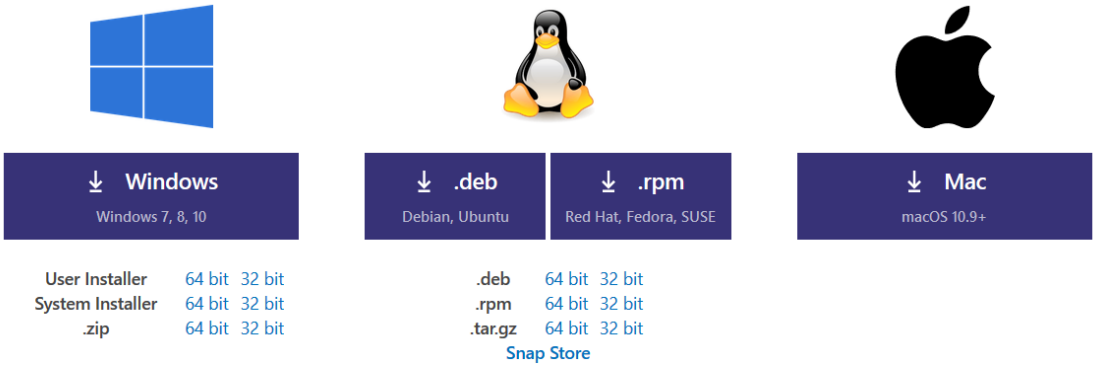
Silahkan *download* di https://nodejs.org/en/download/current/

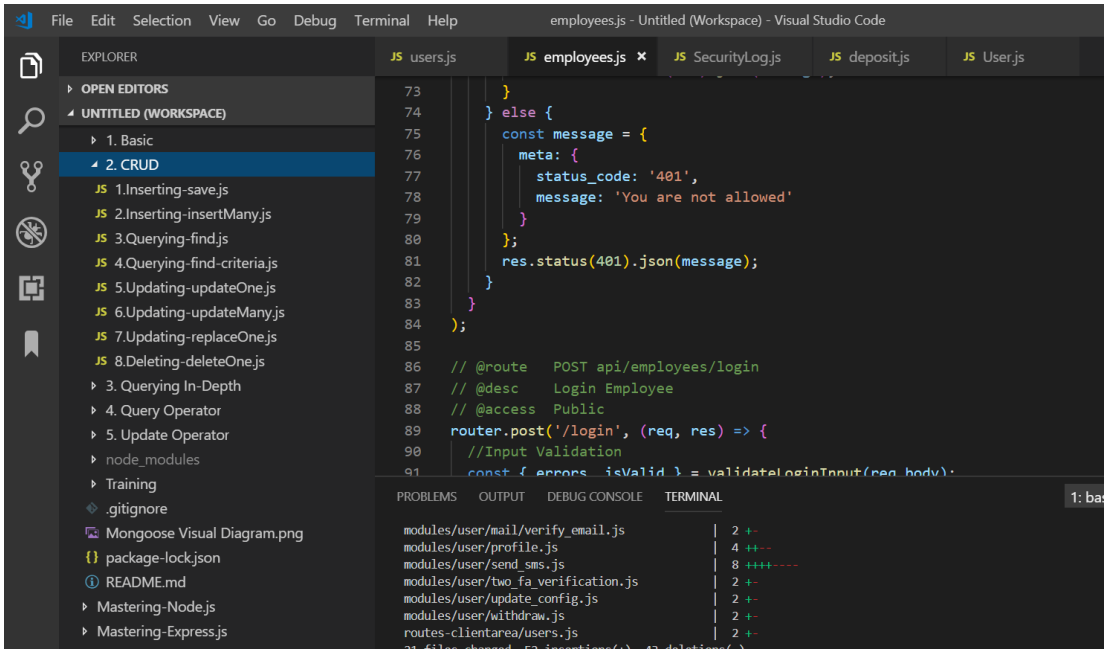


Install Visual Studio Code

Silakan *download* di https://code.visualstudio.com/download

Silahkan *download* sesuai dengan sistem operasi dan arsitektur sistem operasi anda :





Install Python

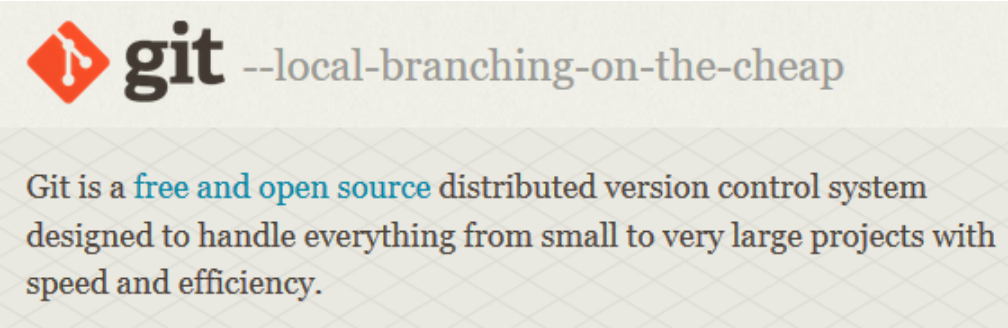
Silahkan download python di : https://www.python.org/downloads/

Install Google Chrome atau Firefox.

Silahkan anda *googling* kunjungi situs resmi *firefox* atau *google*. Penulis menyarankan anda menggunakan *browser firefox*, namun jika anda sudah terbiasa dengan *google chrome developer tool* silahkan menggunakan *google chrome*.

Install git

Silahkan *download* git di : https://git-scm.com/downloads



Install Postman

Silahkan *download postman* di : https://www.postman.com/downloads/

Konvensi Penulisan?

1. Setiap tulisan yang di beri ***bold***, bermakna agar pembaca fokus pada konteks yang sedang dibahas di dalam buku ini.
2. Untuk setiap terminologi **penting** yang muncul untuk pertama kali akan diberi warna biru dan *bold*, contoh ***website***.
3. Jika anda menemukan teks *hyperlink* berwarna biru tanpa *underscore*, maka anda dapat klik tulisan tersebut untuk kembali pada bab tertentu untuk membantu anda belajar. Contoh tulisan yang bisa anda klik sambil menekan CTRL : apa itu *open web platform*?
4. Setiap kode pemrograman akan disimpan di dalam sebuah *box* berwarna *hitam* dengan *font Fira Code* ukuran 12 seperti di bawah ini :

```
var hello = 'Hello World! Gun'
console.log(hello)
```

Output :

```
Hello World! Gun
```

5. Jika sebuah potongan kode pemrograman dan perintah dalam *command line* ditulis diantara suatu paragraf, bentuk *font* yang akan digunakan adalah *font Segoe UI* dengan ukuran 12 dan efek *bold* dengan *background* abu seperti **fungsiTulisNama()** .
6. Setiap informasi penting yang harus dicatat dan diingat oleh pembaca akan disimpan kedalam sebuah tabel seperti pada tabel di bawah ini :

Notes
Jangan lupa simpan *file*, agar data tersimpan jika listrik padam !

7. Untuk setiap interaksi dengan *keyboard* atau *mouse*, seperti *hotkey* atau *mouse click* maka instruksi akan diberi tanda dengan *bold* warna merah, contoh **CTRL+SHIFT+K.**

Feedback?

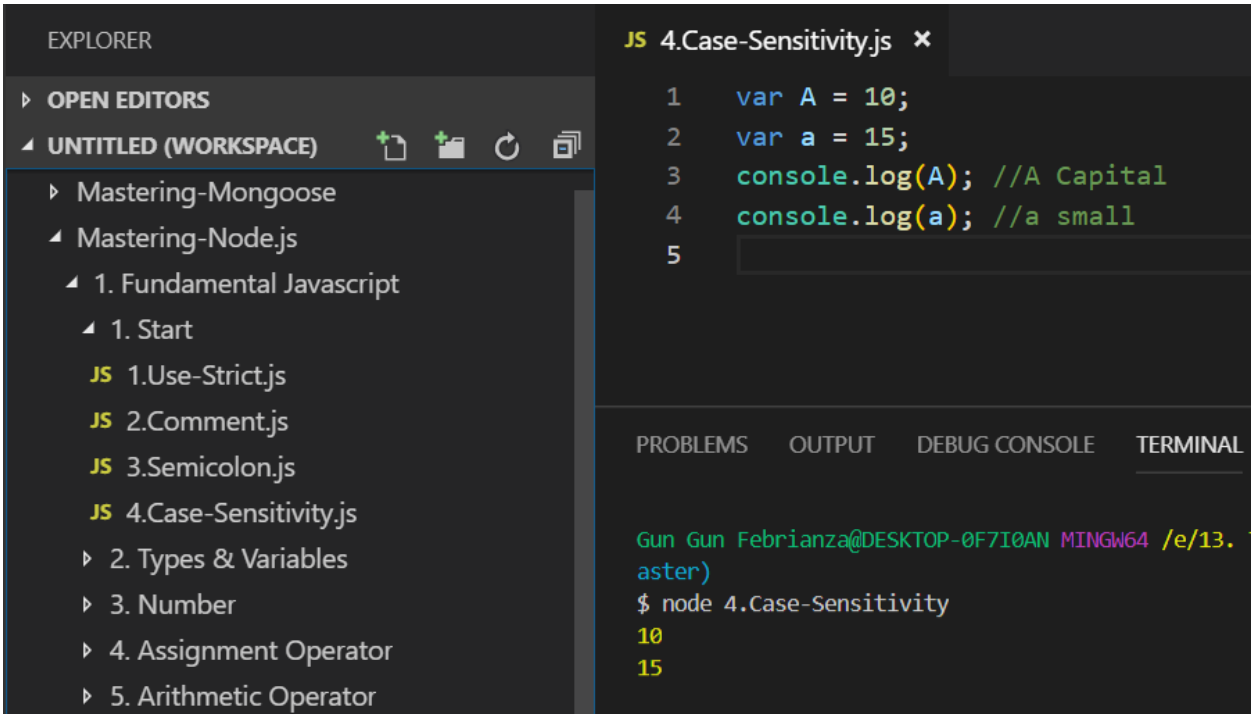
Jika ada *feedback* yang bersifat *private* silahkan diajukan melalui *email* saya gungunfebrianza@gmail.com.

Kode Sumber?

Untuk ***Source Code*** dan bahan ajar silahkan cek di :

https://github.com/gungunfebrianza/Belajar-Dengan-Jenius-AWS-Node.js

Penggunaan Kode ?



Pilih salah satu *file* yang ingin dieksekusi, misal `4.Case-Sensitivity.js` kemudian buka dalam *terminal*. Eksekusi *file* dengan perintah `node [nama file tanpa ekstensi]` contoh :

```
node 4.Case-Sensitivity
```

Terdapat Kesalahan?

Silahkan ajukan isu :

https://github.com/gungunfebrianza/Belajar-Dengan-Jenius-AWS-Node.js

Pertanyaan, Kritik dan Saran?

Seluruh pertanyaan silahkan diajukan melalui grup Pinter Coding :

https://www.facebook.com/groups/pintercoding

Pertanyaan sengaja dialihkan ke grup agar anda bisa mendapat bantuan dari sesama anggota lainya. Bisa berdiskusi untuk berkenalan dengan developer lainya untuk mempermudah proses belajar anda. Kritik dan saran terbaik dari pembaca akan saya tampilkan pada edisi revisi.

Kritik dan saran diperlukan agar *ebook* ini menjadi lebih baik lagi & terus berkembang.

Table of Contents

Contents

Chapter 1

Belajar *Open Web Platform*

Subchapter 1 – Apa itu *Open Web Platform*

The web platform is Write Once, Cry Everywhere.
— Yehuda Katz

Subchapter* 1 – *Objectives

- Mengenal Apa itu ***Open Web Platform*?**
- Mengenal Apa itu ***Technical Specification*?**
- Mengenal Apa itu ***HTML 5.2*?**
- Mengenal Apa itu ***Web Assembly*?**
- Mengenal Apa itu ***EcmaScript*?**
- Mengenal Apa itu ***WebSocket*?**
- Mengenal Apa itu ***WebRTC*?**
- Mengenal Apa itu ***WebGL*?**

Teknologi *web* itu sangat luas dan menarik untuk dipelajari, hampir setiap hari banyak hal baru lahir disana. Pemanfaatan teknologi *web* sudah menjadi kebutuhan sehari-hari, hampir setiap lini bisnis profit dan non profit kini sudah menggunakan teknologi *web* yang telah distandarisasi oleh ***OWP***.

Tapi apa itu *OWP*? Jujur saja dari semua buku tentang *web programming* yang penulis pernah baca belum ada satupun yang menulis tentang *OWP*. Padahal *OWP* adalah dasar informasi yang harus kita ketahui terlebih dahulu sebelum mengenal dunia *web*.

OWP Adalah singkatan dari ***Open Web Platform***, di dalamnya terdapat koleksi teknologi yang dikembangkan dengan konsep ***Open Standard*** oleh *W3C* (***World Wide Web***

Consortium) dan organisasi pemangku standar (*Standards Setting Organization* atau ***SSO***) lainya seperti WHATWG (*The Web Hypertext Application Technology Working Group*), *Unicode Consortium*, IETF (*Internet Engineering Task Force*), dan *Ecma International*.[1]

Terminologi *Open Web Platform* sendiri diperkenalkan oleh *W3C* dan pada tahun 2011 dijelaskan CEO *W3C* yaitu Jeff Jaffe bahwa :

"*OWP adalah sebuah platform untuk membuat inovasi, konsolidasi dan efisiensi harga*."

1. *Technical Specification*

Masing-masing teknologi memiliki ***Specification***, perlu diketahui "*Specification is not user manual*", maksud dari ***specification*** adalah tujuan yang digunakan untuk menjelaskan kepada para *programmer* siapa yang melakukan implementasi teknologi dan fitur apa saja yang harus ada dan bagaimana cara implementasinya.

Koleksi teknologi dalam *Open Web Platform* adalah *computer language* dan *APIs* dalam ruang lingkup teknologi *web* seperti :

2. HTML 5.2

Gambar 1 HTML 5 Technology

HTML (*hypertext markup language*) adalah bahasa *markup* untuk membuat dokumen *web* yang dapat ditampilkan oleh sebuah *web browser*. ***HTML*** dikembangkan oleh *WHATWG* sekumpulan orang-orang yang peduli terhadap teknologi *web*, saat ini pengembangan *HTML* sudah mencapai versi 5.2.

HTML 5.2 memperkenalkan konsep *semantic* dan sekumpulan *APIs* untuk membangun *complex web application*. *HTML 5.2* didesain untuk *adaptable* dengan perangkat *dekstop* dan *mobile* karena seluruh ***browser engine*** dalam *modern browser* sudah mendukung *HTML 5.2*.

Semantic Advantage

Dari sisi ***Semantic*** terdapat *HTML Element* baru untuk berinteraksi dengan *multimedia* dan *graphic content* seperti `<video>`, `<audio>`, `<canvas>` dan dukungan terhadap ***SVG API*** dan ***MathML API*** untuk menampilkan formula matematika dalam dokument *web*.

Connectivity Advantage

Dari sisi ***Connectivity*** terdapat ***WebSocket API*** untuk *full duplex communication* antara *server* dan *client* secara cepat, ***Server Sent Event (SSE) API*** agar ***server*** bisa melakukan *push event* kepada *client* dan ***WebRTC API*** teknologi *real-time communication* untuk melakukan *videoconferencing* di dalam *browser* tanpa perlu menggunakan *plugin* tambahan lagi.

Storage Advantage

Dari sisi ***Storage*** terdapat ***Web Storage API*** untuk menyimpan data pada *browser* dengan format ***key/value***. Terdiri dari *localStorage* dan *sessionStorage*. Lalu terdapat ***IndexedDB API*** untuk minyimpan data dalam jumlah besar, pencarian data menggunakan *IndexedDB API* sangat cepat karena menggunakan *indexing*. Terdapat juga *API* untuk mendeteksi apakah pengguna *browser* dalam keadaan *online* atau *offline* (terhubung ke internet) dan dukungan ***File API*** untuk mengakses *file* dalam sistem operasi kita.

Multimedia Advantage

Dari sisi ***Multimedia*** terdapat dukungan untuk ***Camera API***, dan ***WebVTT*** untuk membuat *subtitle* dan *chapter*. Selain itu terdapat ***Canvas API*** untuk melukis *object*, dukungan untuk ***WebGL*** agar bisa berinteraksi dengan *object* 3 Dimensi dan ***SVG API***.

Performance Advantage

Dari sisi ***Performance*** sudah menggunakan *javascript engine* yang sudah mendukung *JIT compilation*, terdapat ***Web Worker*** jika kita ingin memberdayakan *threads*, terdapat ***XMLHTTPRequest Level 2*** untuk melakukan *fetching* secara *asynchronously* menggunakan ***AJAX***, terdapat ***History API*** yang dapat digunakan untuk memanipulasi *history* dalam *browser*, terdapat ***Drag & Drop API*** untuk memanipulasi *element*, ***Fullscreen API*** untuk *user experience* yang lebih baik dalam menonton tayangan.

Device Access Advantage

Dari sisi ***Device Access*** terdapat dukungan seperti *Geolocation* yang membuat *browser* dapat mengakses lokasi user, terdapat ***Touch Event*** yang dapat digunakan untuk mendeteksi sentuhan layar, terdapat ***Device Orientation Detection*** yang bisa membaca posisi layar secara *potrait* atau *landscape*, dan ***Pointer lock API*** untuk mengunci *pointer* pada suatu *content* dalam *web*.

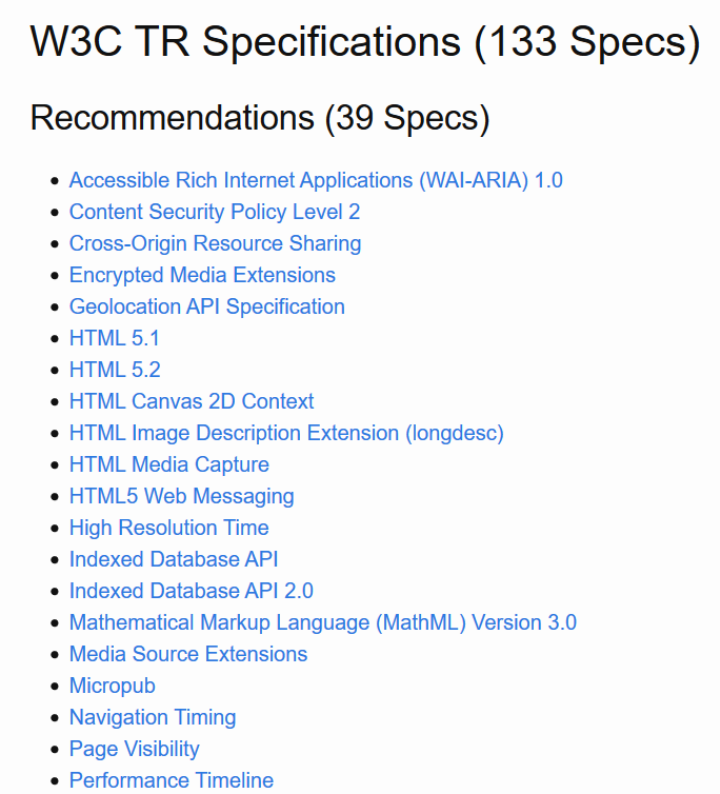
Specification

Spesifikasinya di :

https://html.spec.whatwg.org/multipage/

HTML 5 sudah tumbuh menjadi sekumpulan *browser technology* yang dapat membuat *web developer* untuk mengembangkan *complex web application*. Hampir sebagian besar spesifikasi ***HTML*** *5* dibuat oleh ***W3C Technical Report (TR)*** dokumen, namun ada juga spesifikasi yang dibuat bukan oleh ***W3C*** (***non-W3C Technical Report (TR)***).

Saat buku ini ditulis terdapat ***144 active specification*** sedang dikembangkan dalam teknologi *HTML 5* :



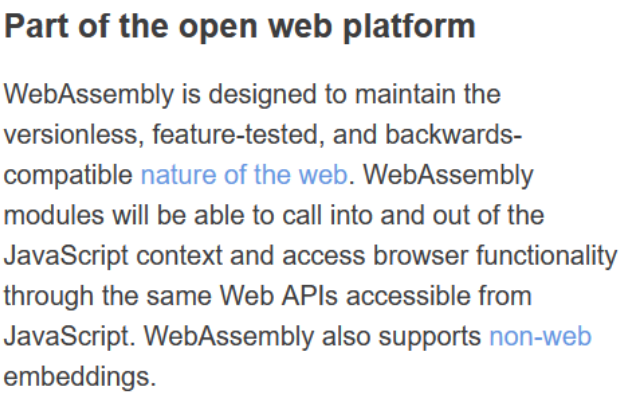
Gambar 2 W3C TR Specification

Jika anda ingin mengetahui lebih lanjut bisa dilihat disini :

http://html5-overview.net/current

3. *Web Assembly*

WebAssembly adalah teknologi terbaru dari ***Open Web Platform*** yang dikembangkan oleh ***W3C***. Pernyataan ini bisa dengan jelas kita lihat disitus resmi *web assembly* :



Gambar 3 Web Assembly As Open Web Platform[2]

Initialy designed agar aplikasi yang ditulis dari bahasa pemrograman seperti *C/C++ & Rust* bisa berjalan di *modern browser*. Bahasa *C/C++ & Rust* akan dikompilasi untuk memproduksi *Web Assembly*, kecepatan *web assembly* memiliki *magnitude* yang lebih cepat dari *javascript*.

Secara teknis *Web Assembly* menggunakan *javascript engine* yang bisa meniru (mimik) *virtual machine* untuk membaca *binary instruction format* dengan gaya *stack-based-machine*.

Impact dari inovasi ***Web Assembly*** adalah pengembangan aplikasi berat di domain seperti ***Computer Vision, 3D Processing, WebVR, & Image Processing*** menjadi bisa dicapai dengan *perfomance* kecepatan mendekati eksekusi *native code*.

WebAssembly adalah sebuah *abstraction* untuk *modern hardware*, yang membuatnya menjadi bahasa komputer yang memiliki karakteristik *platform-independent*. *Web assembly* memiliki beberapa keunggulan :

Safe

Tersedianya *managed language* yang melaksanakan *memory safety* dengan cara mencegah program mengakses atau memanipulasi data dan sistem milik pengguna.

Fast

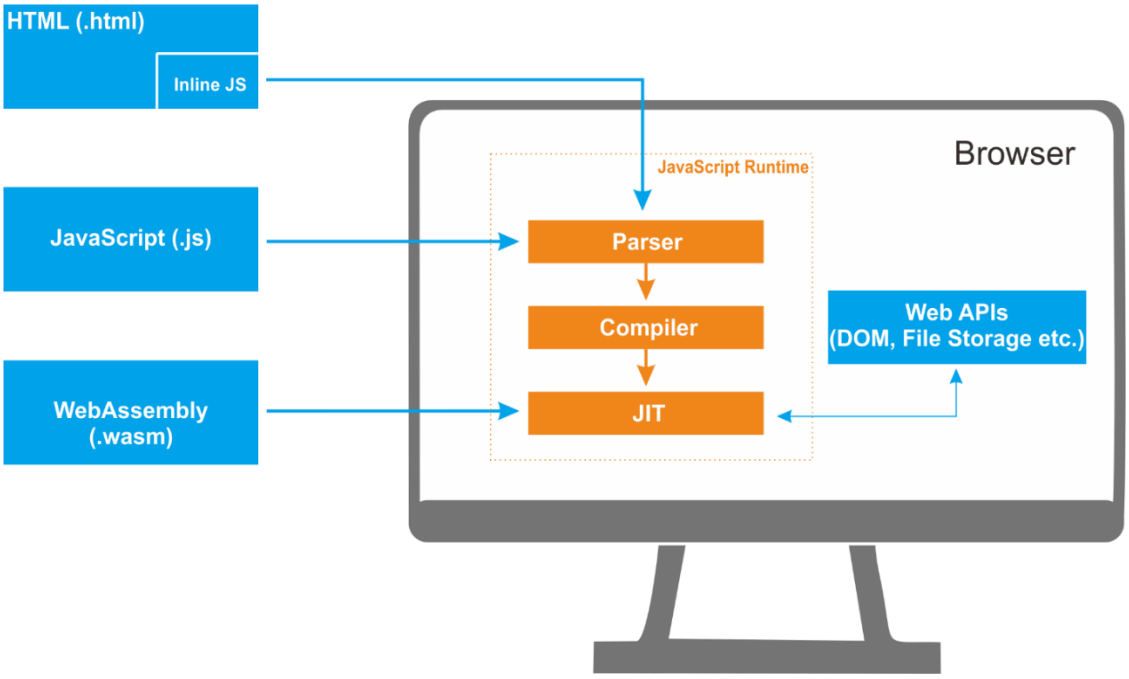
Mampu memproduksi *native code* yang telah dioptimasi sehingga dapat memanfaatkan secara optimal seluruh kemampuan *performance* yang dimiliki mesin.

Portable Code

Bersifat *platform-independent*, berjalan pada semua *modern browser* dan *computer architecture*.

Compact Code

Dikirimkan melalui jaringan ringan untuk mereduksi waktu, hemat *bandwidth* dan responsif.



Gambar 4 Web Assembly on Browser

Specification

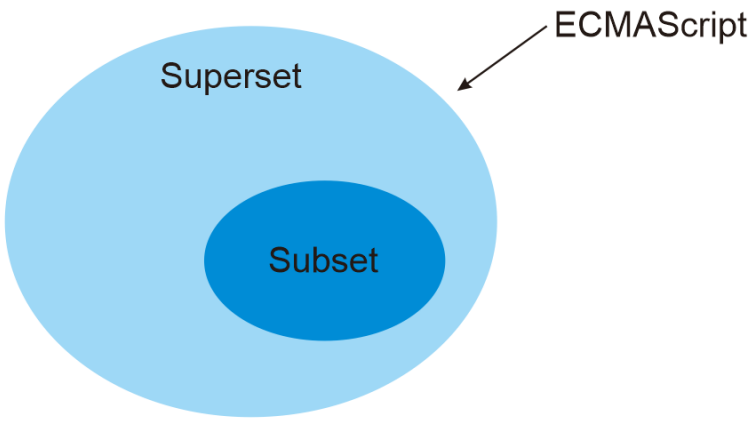
Spesifikasi di :

http://webassembly.github.io/spec/core/

4. *EcmaScript*

Gambar 5 Ecma 262

Javascript adalah bahasa yang telah dijelakan dan distandarisasi di dalam ***ECMA-262*.** Bahasa standar yang ada di dalam ECMA-262 disebut sebagai ***ECMAScript*.** Apa yang kamu ketahui tentang *javascript* didalam ***browser*** dan ***node.js*** adalah ***superset*** dari ***ecmascript*.**[3]



Gambar 6 ECMA Script

ECMAScript sebuah spesifikasi untuk *scripting language* yang distandarisasi oleh *ECMA International*. Diciptakan untuk membuat standarisasi *grammar* dalam *javascript*. Para penyedia layanan *browser* bekerja sama agar bisa membuat *javascript engine* yang dapat mengenali dan mengikuti standar yang dibuat oleh *ECMAScript*.

Saat ini ***EcmaScript 2018*** atau ***ES 9*** adalah versi terbaru dan terakhir yang difinalisasi oleh ***Technical Committee Number 39 (TC39)*** tahun 2018 kemarin pada bulan juni. Setiap kali versi terbaru muncul terdapat fungsionalitas baru yang ditambahkan ke dalam *javascript engine* sebagai *object* baru dan *method* baru.

Specification

Spesifikasi dapat dilihat di :

https://tc39.es/ecma262/

5. *Web Socket*

Web Socket adalah sebuah *protocol*, namun juga terdapat ***Web Socket*** *API* membuat kita mampu menggunakan *WebSocket Protocol*.[4] Dengan *web socket* kita bisa melakukan komunikasi ***Full Duplex*** (komunikasi dua arah seperti telepon) dalam suatu koneksi *TCP* (*Transmission Control Protocol*).

WebSocket API dirancang oleh ***IETF (Internet Engineering Task Force)*** sebuah organisasi yang mengatur standar internet, ***IETF*** juga bersinergi dengan ***W3C*** agar pengembanganya selaras. Saat ini ***WebSocket Protocol*** sudah bisa digunakan hampir diseluruh *browser* seperti *Microsoft Edge, Firefox, Chrome, Internet Explorer, Safari* dan *Opera*.

Specification

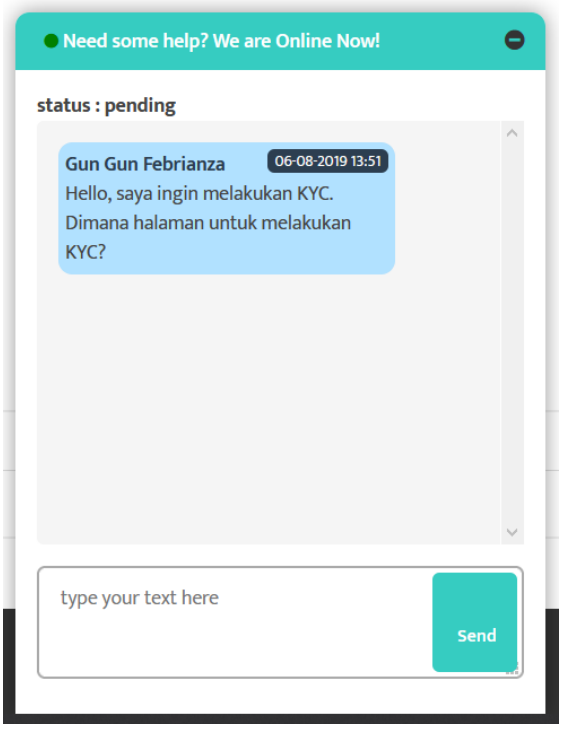
Spesifikasi dapat dilihat di :

https://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17

https://html.spec.whatwg.org/multipage/web-sockets.html#network

Application

Di bawah ini adalah contoh *web application* untuk melakukan *real-time chating* menggunakan *websocket* pada salah satu *platform cryptocurrency exchange* di indonesia (Marketkoin Indonesia) :



Gambar 7 Customer Service on marketkoin.com

6. *WebRTC*

WebRTC adalah sebuah standar baru dan usaha industri untuk memperluas kemampuan *web browser model*.[5] *WebRTC* menyediakan *Communication Protocol* dan API (*Application Programming Interface*) yang bisa membuat *real time communication* dengan koneksi *peer-to-peer*.

Sebuah *Web Browser* tidak hanya meminta *request* kepada *backend server* saja tetapi juga dapat melakukan *request* ke *Web Browser* milik *user* lainya. Implementasinya adalah *Video Conferencing*, *File Transfer*, *Chat* dan *Dekstop Sharing* tanpa memerlukan *plugins* internal atau eksternal.

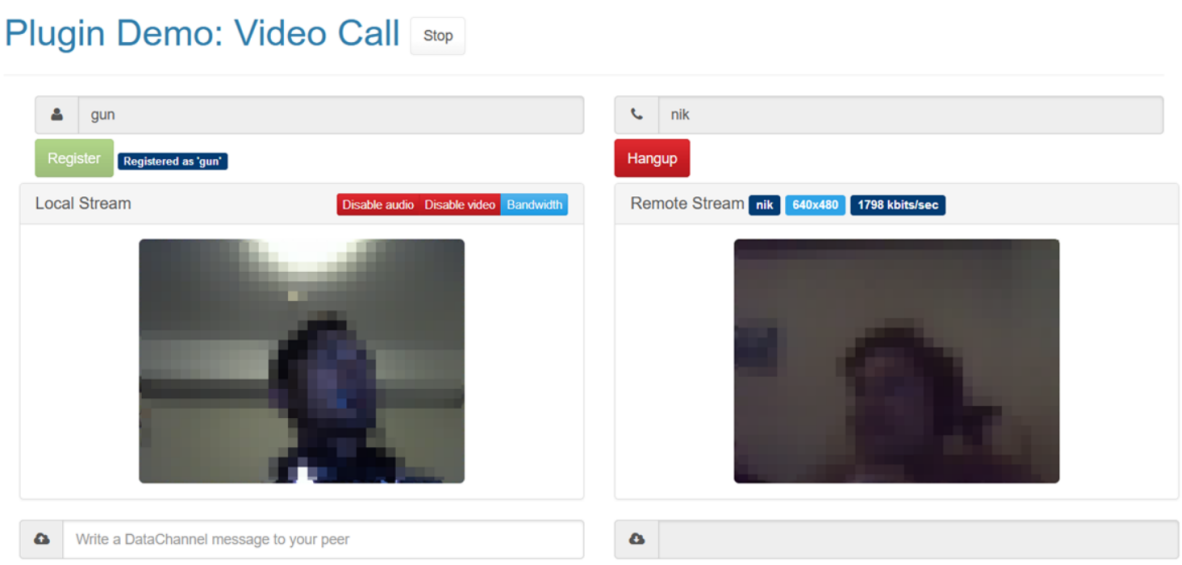
Specification

Spesifikasinya dapat dilihat di :

https://w3c.github.io/webrtc-pc/

Application

Di bawah ini adalah aplikasi pemanfaatan *WebRTC* salah satunya adalah membangun aplikasi ***video call*** di dalam *browser* :



Gambar 8 Video Call dengan WebRTC

Anda dapat mencobanya disini :

https://janus.conf.meetecho.com/videocalltest.html

Jika ingin memulai belajar membangun aplikasi tersebut penulis rekomendasikan mulai dari sini :

https://webrtc.github.io/samples/

7. *WebGL*

WebGL adalah sebuah standar grafik 3 Dimensi didalam web. Dengan *WebGL*, *developer* dapat menikmati secara penuh ***computer graphic rendering hardware*** menggunakan *javascript, web browser* dan *web technology stack*.[6]

WebGL adalah *Javascript* API yang digunakan untuk melakukan *rendering 3D* dan *2D computer graphic* di dalam sebuah *web browser* tanpa menggunakan sebuah *plugin*. Sebuah program yang ditulis untuk *WebGL* terdiri dari bahasa *javascript* dan kode *shader* yang akan dieksekusi oleh *GPU* (*Graphic Processing Unit*).

Pada *HTML* 5, *WebGL* menggunakan *Canvas Element* dan diakses melalui *DOM* (*Document Object Model*). Karena *WebGL* dibangun dengan fondasi yang berasal dari *OpenGL ES 2.0* maka bahasa yang digunakan untuk membuat *shader* adalah *GLSL* (*Open GL Shading Language*).

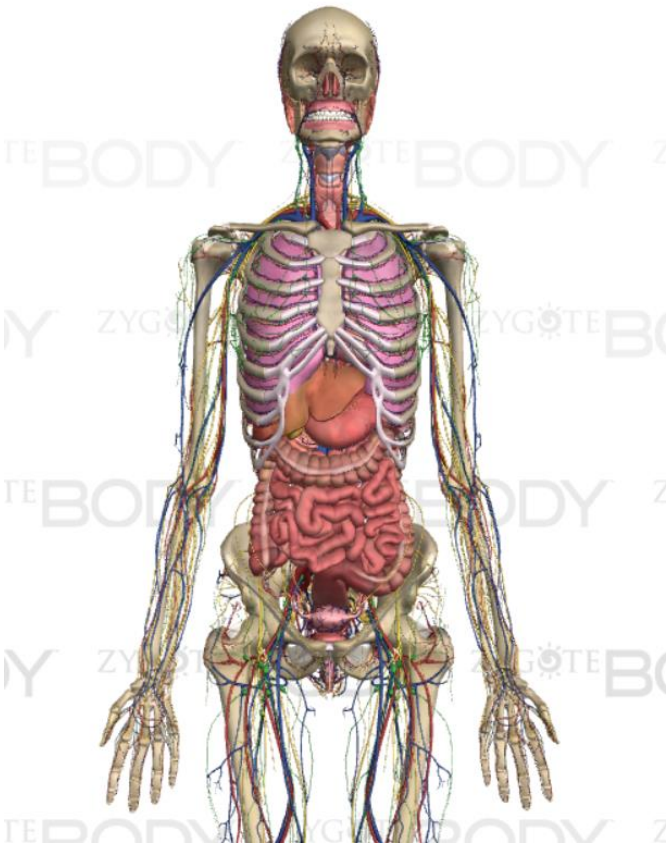
Specification

Spesifikasinya dapat dilihat di :

https://www.khronos.org/registry/webgl/specs/latest/

Application

Di bawah ini adalah aplikasi pemanfaatan *WebGL* untuk dunia kedokteran :



Gambar 9 WebGL Human Anatomy[7]

Salah satu implementasi yang benar-benar bermanfaat untuk dunia kedokteran. Objek 3 Dimensi di atas *pure* dibuat menggunakan *WebGL*. Dan salah satu *library javascript* yang sangat terkenal untuk berinteraksi dengan *WebGL* adalah *Three.js*. Jika anda ingin mengeksplorasi lebih jauh silahkan lihat disini :

https://threejs.org/examples/#webgl_camera

Subchapter 2 – Apa itu Web Application?

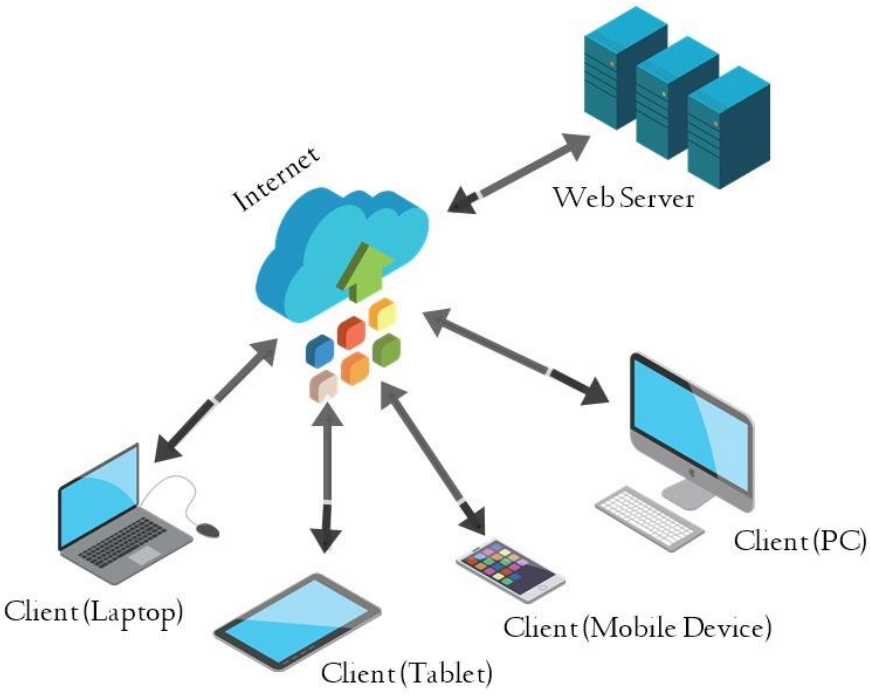
We don't just build websites, we build websites that sells.
— Christopher Dayagdag

Subchapter 2 – Objectives

- Mengenal Apa itu ***Server*?**
- Mengenal Apa itu ***Virtual Private Server*?**
- Mengenal Apa itu ***Web Server*?**
- Mengenal Apa itu ***Web Page*?**
- Mengenal Apa itu ***Network*?**
- Mengenal Apa itu ***Internet*?**
- Mengenal Apa itu ***Internet Exchange Point*?**
- Mengenal Apa itu ***Content Delivery Network (CDN)* ?**
- Mengenal Apa itu ***Cloud Computing*?**
- Mengenal Apa itu ***Serverless Computing*?**

Web Application adalah evolusi dari *web site* atau *web system*. *Web Site* **pertama kali dibuat** oleh ilmuwan bernama ***Tim Berners-Lee*** saat sedang melakukan penelitian di CERN. ***Web Application*** adalah *web system* yang mengijinkan user untuk bisa mengeksekusi ***business logic*** dengan sebuah *browser*.[8]

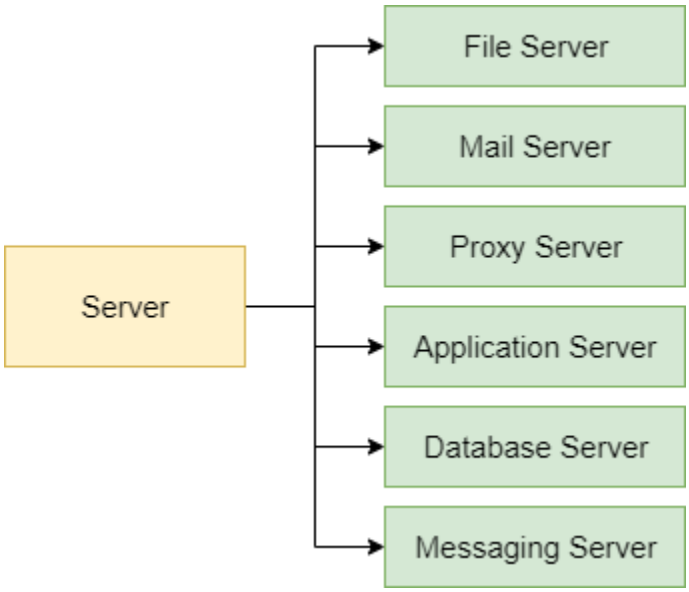
Saat ini pengembangan teknologi *web* telah distandarisasi dalam ***Open Web Platform (OWP).*** Setiap ***Web Application*** terdiri dari dari beberapa elemen yang saling bekerja satu sama lain agar bisa memberikan suatu layanan. Sebuah *web application* terdiri dari ***web server*** dan ***clients*** yang bisa anda lihat pada gambar di bawah ini :



Gambar 10 Web application

Pada gambar di atas terdapat *computer*, *mobile device*, *tablet* dan *laptop*. Pengguna yang menggunakan perangkat tersebut disebut dengan *client*. Seorang *client* biasanya melakukan *request* pada ***Web Server*** menggunakan sebuah ***Web Browser***. Apa itu *web server* dan apa itu *server*? Kita akan mengupasnya sejengkal demi sejengkal.

1. *Server*



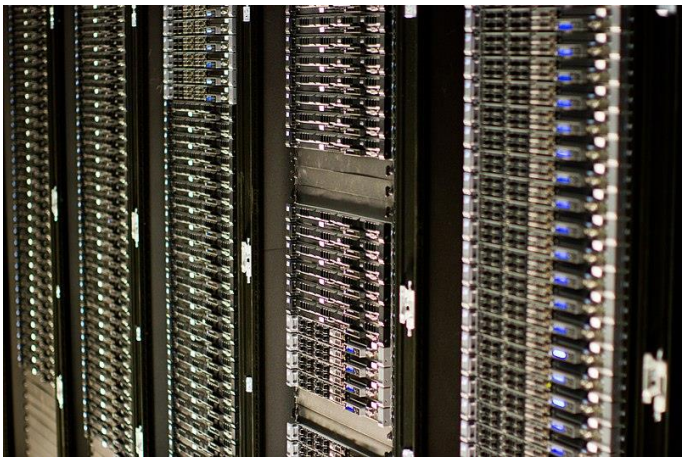
Gambar 11 Server Classification

Terminologi ***Server*** sering kali kita dengar dalam dunia komputer. Seiring berjalanya waktu dan evolusi teknologi komputer terminologi *server* menjadi ambigu. Terminologi *server* sendiri bisa mengacu pada perangkat keras (*hardware*) berupa ***Physical Computer*** dan perangkat lunak (*software*) misal *mail server, database server* atau *print server*. Tujuan *server* adalah untuk memberikan sebuah layanan (*service*) berupa *sharing data*.

Sebuah *single server* bisa memberikan *service* untuk *multiple client*, sebaliknya *single client* dapat mengakses *multiple server*. Untuk menjalankan *server* terdapat beberapa *hardware requirement* (*specification, robustness, cost, noise*) tergantung dari tujuan pembuatan *server* itu sendiri.

Pada umumnya sebuah *server* harus berjalan setiap saat selama (24/7) tanpa mengalami interupsi agar layananya tidak terganggu, biasanya *server* seperti ini memberikan layanan yang memiliki nilai bisnis.

Di bawah ini adalah contoh *server* yang telah dibuat menjadi *server cluster,* terdiri dari sekumpulan komputer *server* agar bisa memberikan kualitas layanan dengan *performance* yang lebih baik.



Gambar 12 Server Cluster

Setiap *server* juga memiliki sistem operasi yang umumnya didesain bukan untuk *client*, sistem operasi yang akan digunakan juga dipengaruhi oleh *platform hardware* (*intel, amd*, *mips*, etc) dan skala pembuatan *server* itu sendiri. Diantaranya adalah *Windows Server, Mac OS X Server* dan *Ubuntu Server*.

Setelah *hardware server* dan sistem operasi *server* disediakan selanjutnya adalah membuat layanan *server* itu sendiri, di bawah ini berapa contoh tipe *server* yang bisa dibuat :

File Server

File Server, sebuah *server* yang memberikan layanan kepada penggunanya untuk dapat berbagi *file* dan *folder* atau *space* penyimpanan data. *File server* dapat berupa :

1. ***File Transfer Protocol (FTP) server***,
2. ***Service Message Block/Common Internet File System (SMB/CIFS) protocol server***,

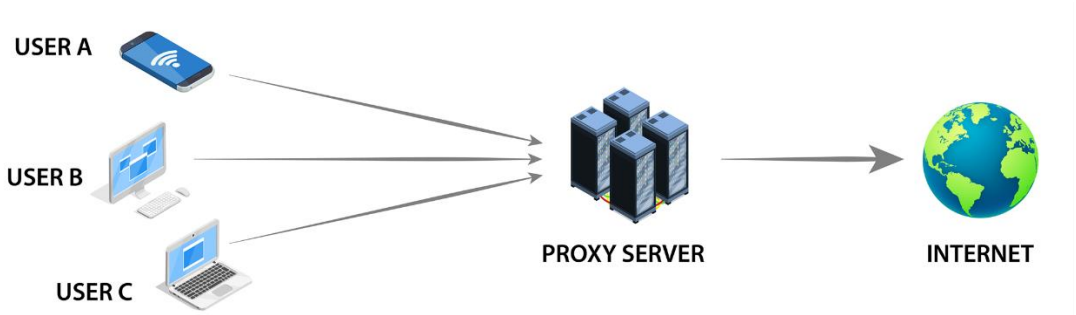
3. ***HTTP server***, atau
4. ***Network File System (NFS) server***.

Mail Server

Mail Server, sebuah *server* yang memberikan layanan kepada penggunanya untuk dapat mengelola surat elektronik. *Mail Server* seringkali disebut dengan ***email server*** atau ***Mail Transport Agent (MTA)***.

Terdapat dua protokol utama yang digunakan dalam *mail server* dan pengiriman *email* diantaranya adalah ***Simple Message Transport Protocol (SMTP)*** dan ***Post Office Protocol 3 (POP3)***. *SMTP* mengangkut pesan antara *mail servers*. *POP3* adalah protokol yang digunakan oleh klien untuk berinteraksi dengan *mail server* agar bisa mengirim dan menerima pesan.

Proxy Server



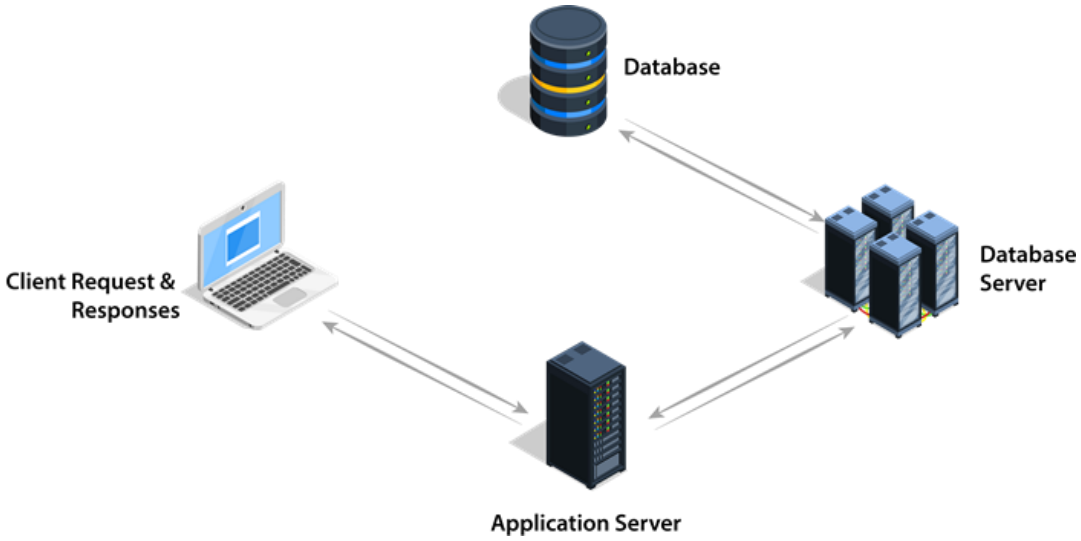
Gambar 13 Proxy Server Illustration

Proxy server adalah *server* yang menjadi perantara dalam jaringan komputer untuk melayani permintaan dari *client* agar bisa melakukan akses *resources* dari suatu *remote server*. Salah satu *proxy server* adalah *Reverse Proxy Server*, *server* akan menerima *request* yang berasal dari internet dalam bentuk ***HTTP Request*** dan mengelolanya di dalam jaringan komputer internal *server*.

Application Server

Di era *cloud computing* sebuah *application server* memiliki fungsi yang hampir sama dengan *Software as a Service (SaaS)*[9]. *Application server* adalah sebuah program yang hanya berjalan dan melakukan komputasi di dalam *server* (*server-side scripting*), hasil komputasi bisa diberikan kepada *client* atau hanya disimpan di dalam *server* saja.

Database Server



Gambar 14 Application & Database Server

Database Server, menyediakan sebuah ***interface*** untuk menanggapi permintaan dari *client*, *interface* dapat diakses melalui *application server* atau secara langsung melalui sebuah *database management system*. Pada umumnya sebuah *application server* akan memberikan data kepada *database server* untuk diproses agar bisa mendapatkan hasil pengolahan data.

Messaging Server

Dalam *Software Architecture* setiap kali kita ingin membuat sebuah interaksi antar mesin komputer kita memerlukan sebuah "*messaging pattern*" untuk mengirimkan pesan. Saat ini terdapat dua pattern diantaranya adalah :

1. *Request-response Pattern* contohnya *HTTP Protocol*
2. *One way pattern* contohnya *UDP Protocol*

Setiap kali *client* berinteraksi dengan *web server* kita harus melakukan *request* terlebih dahulu kepada *server*. Maksud dari request bentuknya adalah sebuah *HTTP Request*, dengan *HTTP Method* yang spesifik seperti GET, POST, PUT dan sebagainya hingga kita mendapatkan kembali *HTTP Response*.

Jadi dalam *HTTP Protocol* kita harus melakukan request terlebih dahulu untuk mendapatkan response, lalu bagaimana untuk mendapatkan *response* tanpa harus melakukan *request* terlebih dahulu? bagaimana caranya melakukan 1 buah *request* namun mampu melakukan *trigger* di dalam server untuk mendapatkan response lebih dari 1?
Kita membutuhkan sebuah *messaging pattern*, ini disebut dengan *Publish-Subscribe (pub-sub) model*. Pada *pub-sub model* terdapat dua komponen :

1. *Publisher*

   Sebuah *service* yang akan menyebarkan pesan (*broadcast the message*) ke setiap *service* yang melakukan *susbcribe*.
2. *Subscriber*

   Sebuah *service* yang akan mendapatkan pesan yang di *broadcast* oleh *publisher*.

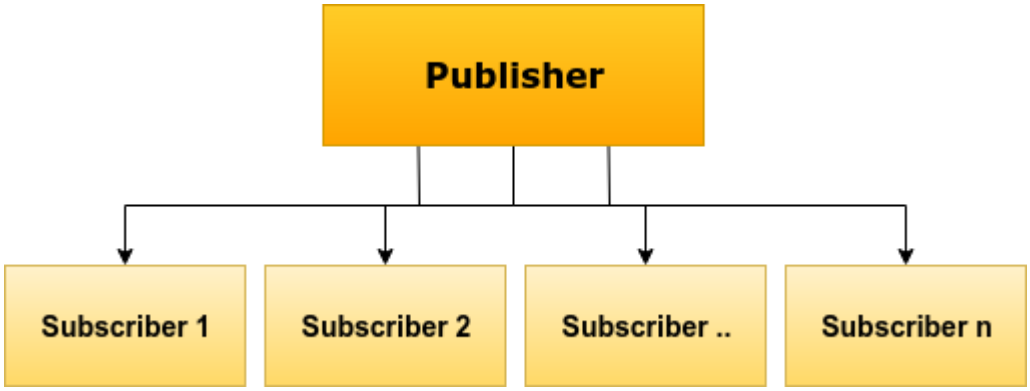
Messaging Server, sebuah *server* yang menjadi layanan perantara yang menerima, meneruskan dan menahan pesan antara *client application* dan *service*.

Publish-subscribe messaging servers adalah salah satu *message server* yang mengkomunikasikan pesan dari *client (publisher)* menuju sebuah *messaging server*. Pesan

di kategorikan dan di dalamnya terdapat sekumpulan *client* yang telah berlangganan (*subscribed client*).

Subscriber dapat menentukan kategori pesan yang ingin di dapatkan.

Salah satu contoh *publish-subscribe messaging service* adalah *Redis, NATS & Faye*.



Gambar 15 Publisher broadcast message to subscriber

2. *Virtual Private Server*

Virtual Private Server (VPS) adalah sebuah komputer virtual atau *server* virtual yang dapat kita gunakan seperti mesin komputer pada umumnya secara *remote*. Kita dapat membangun *Virtual Private Server (VPS)* agar dapat disewakan kepada orang lain atau kita menggunakan *Virtual Private Server (VPS)* dengan cara menyewa layananya pada sebuah *Internet Hosting Service*.

Layanan *VPS* diberikan oleh sebuah ***Internet Hosting Service*** tersimpan dalam sebuah *physical server* yang mereka miliki. Dalam satu *physical server* tunggal mereka dapat membuat lebih dari satu *Virtual Private Server (VPS)*. Jumlah VPS yang dapat diproduksi tergantung dari spesifikasi *physical server* yang mereka miliki. *Virtual Private Server (VPS)* sering juga disebut dengan ***Virtual Dedicated Server (VDS)***.

Di dalam sebuah *Virtual Private Server (VPS)* kita dapat membangun *File Server, Mail Server* atau *Web Server* untuk membuat aplikasi *e-commerce* atau hanya sekedar membuat *website* sederhana. Dengan *VPS* pengguna mendapatkan akses penuh (*super user/root*) untuk mengelola sistem operasi. Sehingga pengguna *VPS* dapat memasang (*install*) seluruh *software* yang mereka inginkan.

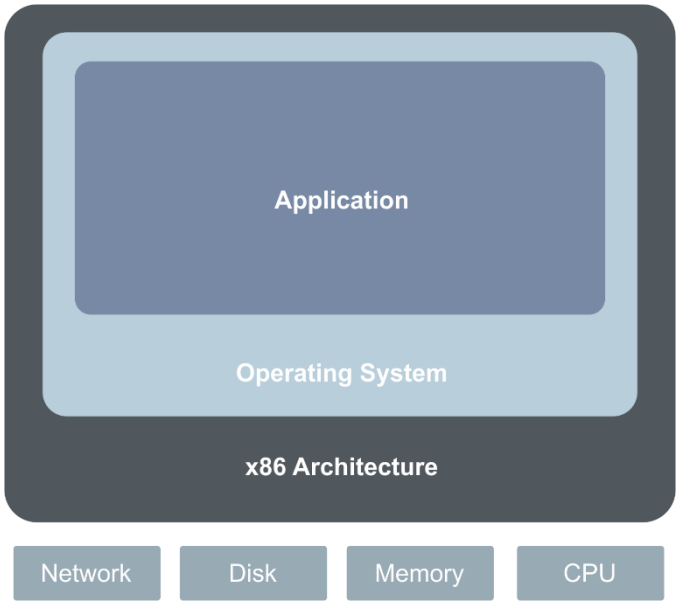
Virtualization

Terminologi ***Virtualization*** memiliki makna membuat sesuatu secara virtual atau artifisial. Dalam buku ***Virtualization Security*** yang diterbitkan *EC-Council*, dikatakan bahwa *Virtualization* adalah kerangka (*framework*) atau metodologi bagaimana membagi sumber daya (*resources*) sebuah komputer agar bisa menjadi sebuah ***multiple execution environment***[10].

Multiple Execution Environment

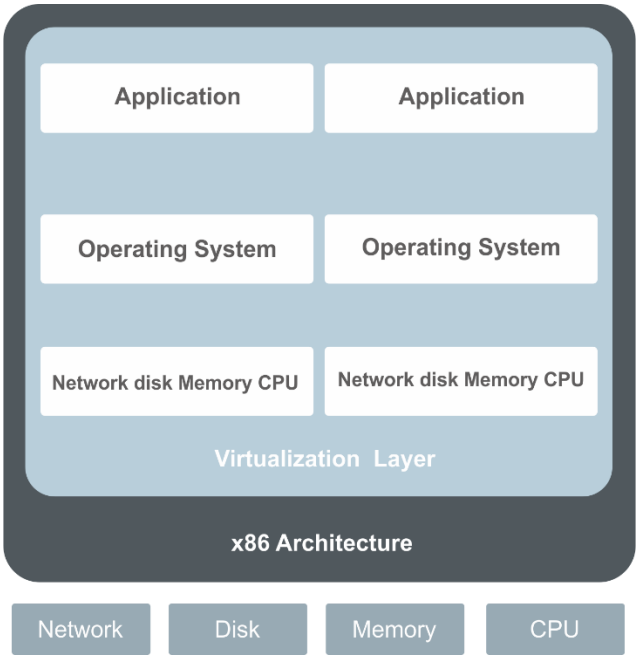
Apa sih yang dimaksud *Multiple Execution Environment*?

Sebelum *Virtualization* muncul dahulu kita menggunakan satu sistem operasi untuk setiap mesin. Perhatikan gambar di bawah ini :



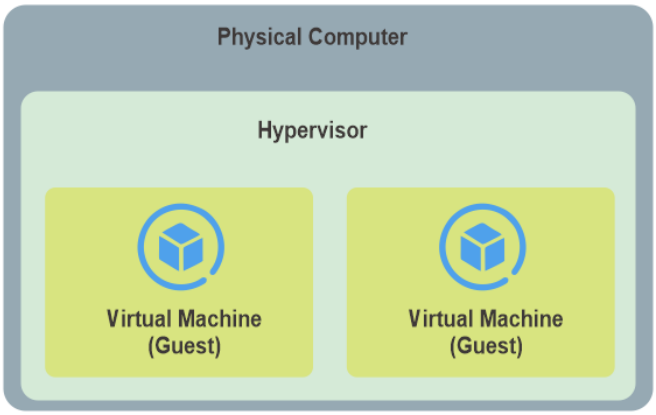
Gambar 16 OS Image pada suatu mesin komputer

Kemudian muncul ***virtualization layer*** yang dapat kita gunakan, misal untuk memasang sistem operasi lebih dari satu dalam satu mesin komputer. Inilah yang dimaksud dengan *Multiple Execution Environment*. Semuanya di isolasi agar masing-masing bisa berjalan dengan baik.



Gambar 17 Virtualization Layer

Pada konteks *Virtual Private Server (VPS)* yang sedang kita bahas, terminologi *virtualization* mengacu pada pembuatan suatu *resource(s)* secara virtual. Sehingga dapat menghemat biaya untuk memaksimalkan pemanfaatan sumber daya suatu komputer. *Virtualization* dapat melakukan ***emulation*** suatu *hardware* menggunakan *software* yang selanjutnya beberapa tehnik dikembangkan agar bisa membangun *server virtualization, dekstop virtualization, network virtualization, storage virtualization* dan masih banyak lagi.



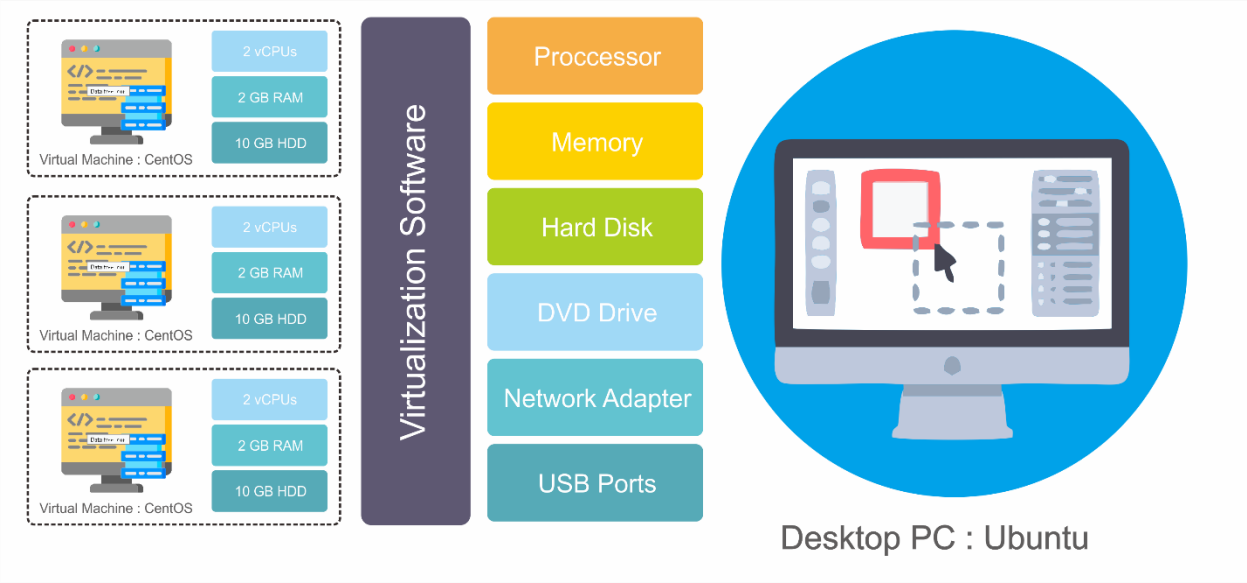
Gambar 18 Virtual Machine

Ketika terdapat sistem operasi virtual misal sistem operasi *linux* di dalam sistem operasi *windows* maka sistem operasi *linux* disebut sebagai ***virtual machine***. Sistem operasi *windows* di sebut sebagai ***host*** dan sistem operasi *linux* yang menjadi *virtual machine* disebut sebagai ***Guest***. *Virtual Machine* memiliki representasi *binary file* yang dapat kita salin atau pindahkan ke dalam *physical computer* sehingga menjadi *portable*.

Virtual Machine

Virtual Private Server (VPS) menggunakan ***Virtual Machine (VM)*** berbasis ***Full Virtualization*** agar dapat meniru sebuah sistem operasi secara keseluruhan. *Virtual Machine* diimplementasikan dengan menambahkan *layer* perangkat lunak pada *physical machine* untuk mendukung sebuah arsitektur mesin secara virtual[11].

Dalam beberapa literatur *Full Virtualization* sering juga disebut sebagai *System Virtual Machine* atau *Hardware Virtual Machine*. Sehingga dalam satu *physical server* atau *physical computer* kita dapat membangun *multiple OS environment* sekaligus. Untuk mewujudkanya kita memerlukan sebuah ***Hypervisor***.



Gambar 19 ilustrasi Virtualization

Hypervisor

Software untuk membuat *virtual machine* di antaranya adalah *Virtualbox* dan *VMware Workstation*. Saat kita membuat *virtual machine* kita dapat membuat *virtual CPU, virtual disk* dan *memory* yang ingin kita alokasikan. Oleh karena itu teknologi *virtualization* sangat membantu untuk membangun *infrastructure* yang *scalable*. *Virtualbox* dan *VMware Workstation* adalah sebuah *hypervisor* yang dapat kita gunakan untuk membangun *guest operating system* di dalam *host operating system*.

Hypervisor adalah *software* yang bertanggung jawab untuk membangun *system virtualization*[12].

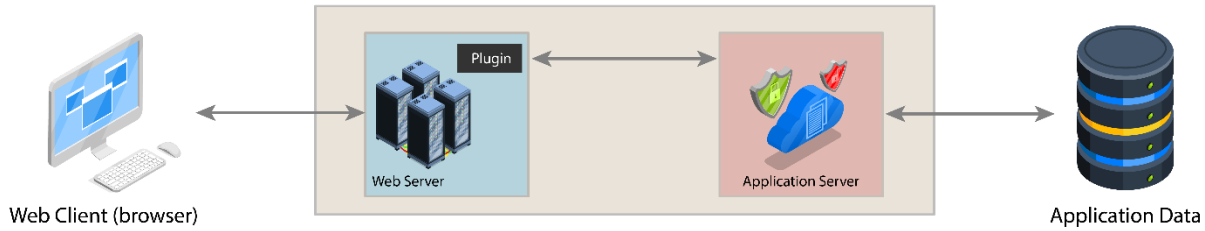
3. *Web Server*

Web server adalah program komputer yang menyimpan, memproses permintaan (*request*) dan mengirimkan sebuah ***Web Page*** melalui protokol yang disebut dengan *HTTP*, sebuah protokol dasar yang saat ini kita gunakan untuk mendistribusikan informasi keseluruh dunia.

Dalam buku ***Web Server Technology*** karya Nancy J. Yeager & Robert E. McGrath yang terbit pada tahun 1996 dikatakan bahwa tugas *web server* adalah menerima permintaan (*request*) dari sebuah *web browser* atas sebuah dokumen di dalam jaringan, membaca permintaan untuk mengetahui *file* apa yang dibutuhkan, mencari *file* yang tersedia, dan mengirim *file* tersebut kepada *web browser*[13].

Web Server awalnya digunakan untuk menyajikan *static content*, namun terus dikembangkan agar mendukung *dynamic content*. Beberapa *web server* memiliki *plugins* yang mendukung *scripting language* seperti *Perl, PHP & ASP*. Penggunaan *scripting language* memberi ruang untuk membangun *Application Server*, yang memiliki akses lebih luas lagi seperti *Connection Pooling* & *Object Pooling*.

Pada beberapa *production environment*, terdapat *web server* yang menjadi sebuah *reverse proxy* untuk *application server* dikarenakan karakteristik *web server* yang cocok untuk menanggapi *static content* dan *application server* untuk *dynamic content*.



Gambar 20 Web Server

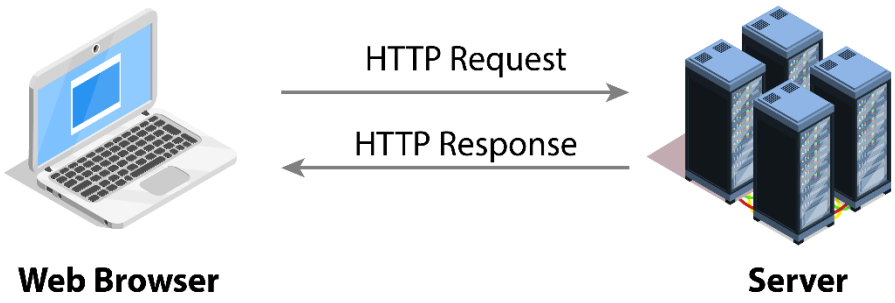
Ada beberapa *Web Server* yang sering digunakan di antaranya adalah :

1. *Apache*
2. *Nginx*
3. *IIS*

4. *Web Page*

Static Web Page

Static Web Page, adalah sebuah halaman yang kontenya tidak akan berubah setiap kali kita melakukan *HTTP Request*. Pada *web browser* kita bisa mengetahui sebuah halaman bersifat *static* dengan melihat ekstensinya pada *address bar web browser*. Jika ekstensinya adalah *htm* atau *html* maka halaman tersebut adalah halaman *static*.



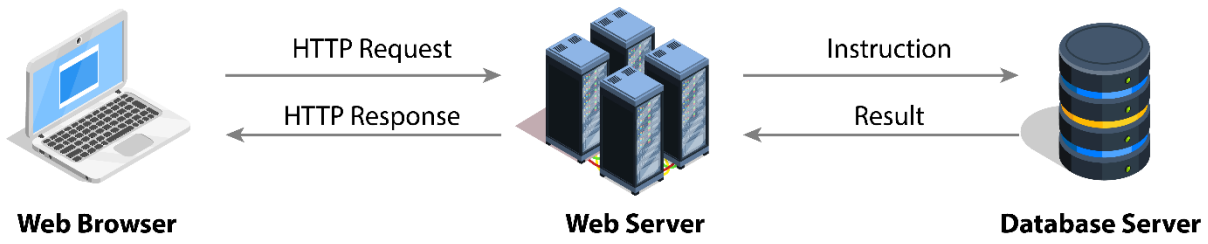
Gambar 21 Static Web Page Processing

Saat *web server* menerima *HTTP Request*, *server* akan mencari *file* dalam *disk drive* yang dimilikinya. Setelah *file* ditemukan *web server* akan kembali mengirimkan *HTTP Response* ke *web browser* milik *client*. *Web browser* akan menerjemahkan kode *HTML* (*Hyper Text Markup Language*) menjadi suatu tampilan visual (*Render*) yang mudah difahami.

Dynamic Web Page

Dynamic Web Page, adalah sebuah halaman yang dihasilkan oleh suatu *program* atau *script* dalam *web server* setiap kali permintaan dilakukan. *Program* atau *script* tersebut akan dieksekusi oleh ***application server*** yang dimiliki oleh *server*.

Sebagai contoh *Client* meminta suatu gambar yang ada di dalam ***database server*** maka *HTTP Request* akan dibaca dan *script* untuk mencari gambar yang diminta oleh *client* akan dieksekusi hasilnya akan diberikan kembali kepada *client*. (ada atau tidak ada gambar tersebut)



Gambar 22 Dynamic Web Page Processing

Permintaan yang dikirimkan ke *web server* termasuk data yang dibutuhkan ***application server*** untuk memproses permintaan. Seperti data yang ada di dalam sebuah *form*, data tersebut sudah termasuk dalam *HTTP Request*. Saat *web server* menerima *HTTP Request* jika terdapat permintaan khusus pada *application server* maka *script* yang sesuai dengan permintaan pada *server* akan dieksekusi.

Jika diperlukan *script* mampu melakukan permintaan pada *database server* sebagai data tambahan untuk menghasilkan sebuah halaman dinamis. Sebuah proses yang diselesaikan oleh *application server* kita bisa menyebutnya dengan ***server-side processing***.

Progressive Web Application (PWA)

Progressive Web Application atau *PWA* adalah *trend* membuat atau mengubah *web application* yang sudah ada (*legacy app*) memiliki *performance* seperti *native application*. *PWA* memiliki karakteristik kecepatan saat memuat ***web application,*** kecepatan saat ***running time*** merespon ***interaction, smooth animation,*** akses ***native device*** dan

menyediakan ***offline functionality*** **(*caching*)** agar aplikasi atau beberapa ***part*** aplikasi tetap berjalan walaupun dalam keadaan tidak terhubung ke internet.

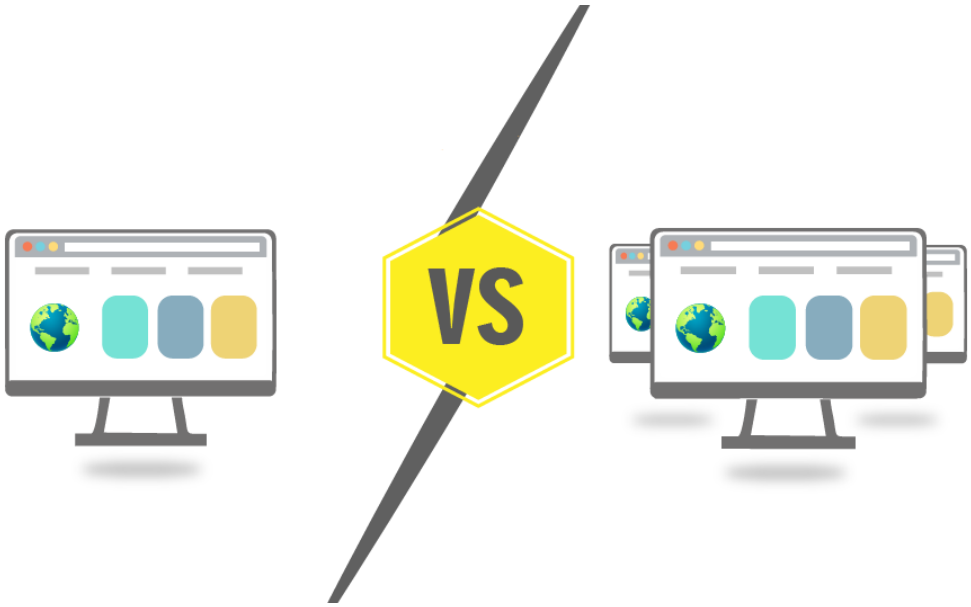
PWA memiliki ***core building block*** diantaranya adalah

1. ***Web Worker***, sebuah *javascript* yang berjalan di dalam *background process* melalui *thread* terpisah sehingga kita bisa melakukan pekerjaan dibelakang layar.
2. ***Background Synchronization*** untuk mengirimkan *request* ke *server* saat terhubung kembali ke internet
3. ***Push Notitification*** untuk menerima informasi yang dikirim dari *server*,
4. ***Application Manifest*** untuk melakukan instalasi dalam ***home screen*** tanpa melalui ***appstore***, dan
5. ***Responsive Web Design (RWD)*** untuk memastikan *layout* dalam *web application* dapat tampil dengan benar disemua layar *device*.
6. ***Native API,*** untuk melakukan akses pada *native API* seperti *Geolocation API* dan *Media API* agar bisa berinteraksi dengan *camera* dan *microphone*.

Dengan *javascript* kita bisa membuat *event* agar bisa memberikan perintah pada *service worker* untuk melakukan suatu pekerjaan dibelakang layar. Salah satunya dalah ***Push-Notification*** ketika *service worker* menerima *web push notification* yang datang dari *server*.

Single Page Application (SPA)

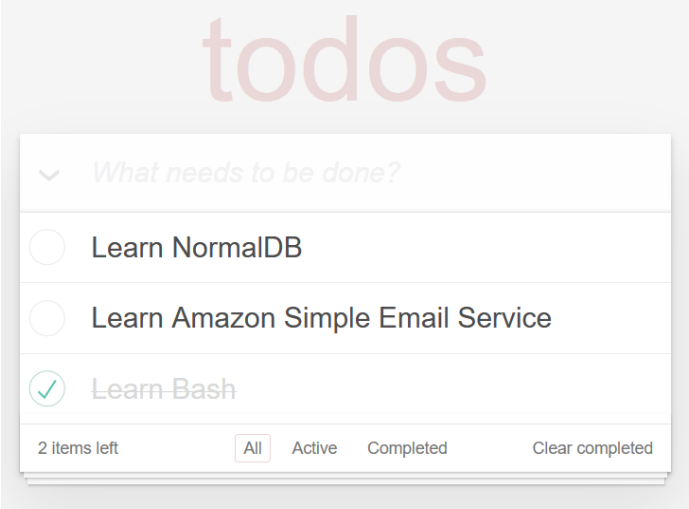
Single-page Application atau *SPA* adalah sebuah paradigma baru dalam pengembangan *web application* yang mendukung ***less server-side code*** dan ***more client-side code***. *Single-page Application* memiliki karakteristik *dynamic web application* dan *real-time update* tanpa harus melakukan ***reload*** halaman. *Server* hanya mengirimkan satu halaman saja kepada *client*, selanjutnya aktivitas *routing* dan *paging* dilakukan di dalam *browser*.



Gambar 23 Single Page Application Illustration

Kita dapat membangun *web application* untuk melakukan ***Create, Read, Update & Delete (CRUD)*** seperti *to-do list application* dan *web application* lainya yang lebih *complex* dengan *single-page application*.

Di bawah ini adalah contoh aplikasi *to-do list* yang dibangun menggunakan *single-page application* yang ditulis menggunakan *pure javaScript* melalui *Backbone.js, AngularJS, Ember.js, KnockoutJS, Dojo, Knockback.js, CanJS, Polymer, React, Mithril, Vue.js dan Marionette.js*:



Gambar 24 Todo List Application

Untuk mempelajarinya anda bisa mengunjungi :

http://todomvc.com/

5. *Network*

Network adalah sebuah sistem jaringan komputer yang membuat *client* dan *server* dapat berkomunikasi.

Sebuah *network* bisa dikategorikan berdasarkan ukuran :

Local Area Network (LAN)

LAN (Local Area Network) sebuah *network* dengan skala yang sangat kecil yang membuat sekumpulan komputer dapat berkomunikasi dengan jarak yang dekat biasanya dalam satu gedung atau ruangan. *Network* seperti ini seringkali disebut dengan ***Intranet***, dapat digunakan untuk menjalankan sebuah *web application* yang hanya bisa diakses oleh pengguna dalam gedung saja.

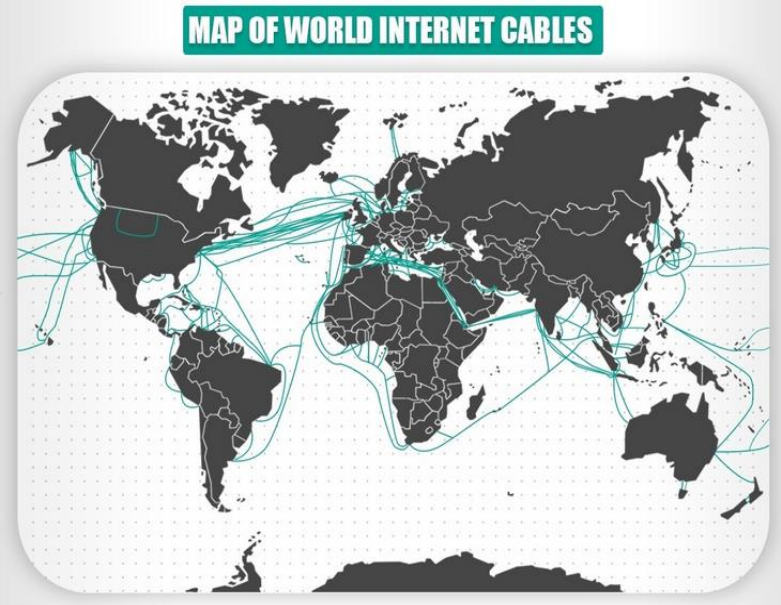
World Area Network (WAN)

WAN (Wide Area Network) terdiri dari sekumpulan *LAN (Local Area Network)* yang saling terhubung. Untuk mengirimkan informasi dari satu *client* ke komputer lainya sebuah *router* akan memastikan *network* mana yang paling dekat ke target komputer dan mengirimnya melalui *network* tersebut. Sebuah *WAN* bisa dimiliki oleh sebuah perusahaan privat atau lebih dari satu perusahaan privat.

Internet Service Provide (ISP)

ISP (***Internet Service Provider***) adalah sebuah perusahaan yang memiliki izin untuk memiliki dan mengatur *WAN* yang terhubung ke *internet* di seluruh dunia melalui ***Internet Exchange Point***. *Internet Service Provider* adalah sebuah organisasi yang menjual akses ke dalam internet[14].

6. *Internet*



Gambar 25 Internet Cables

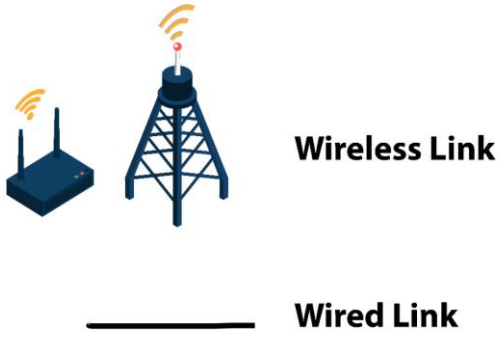
Internet adalah sebuah *network* skala besar yang terdiri dari sekumpulan *networks* kecil yang saling terhubung. Hari Ini *Internet* telah terhubung hampir keseluruh penjuru dunia, kabel *Fiber Optic* telah terpasang mengelilingi bumi melalui lautan, daratan & pegunungan.

Internet Transit

Untuk dapat terhubung ke dalam *internet* sebuah entitas harus menghubungkan dirinya ke dalam suatu entitas yang telah terhubung ke dalam jaringan *internet*. Diwujudkan dengan cara membeli layanan pada *ISP* yang disebut dengan ***Internet Transit*.**

Internet Service Provider juga disebut dengan ***Transit Provider*** sebuah entitas yang menyediakan layanan akses ke dalam *internet*.

Satellite & Fiber Optic



Gambar 26 Communication Link

Tidak hanya *fiber optic*, juga terdapat *satellite*. Keduanya adalah teknologi yang menjadi wujud industri telekomunikasi global. *Medium fiber* sangat disukai oleh korporasi besar untuk transmisi data dan lembaga keuangan untuk sistem transfer dana elektronik karena tingkat keamanan dan redudansi data yang lebih tinggi.

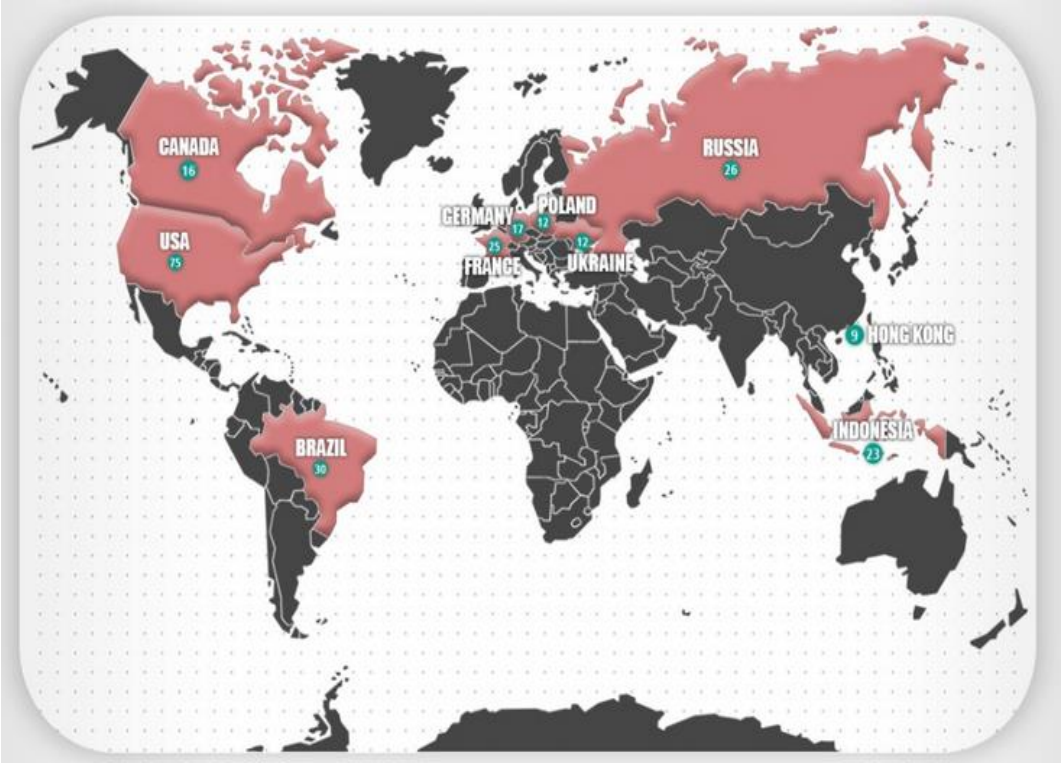
Pada tahun 1950 terdapat lebih dari 5.500 satelit yang didominasi oleh USA dan Rusia. Digunakan untuk aplikasi militer, digunakan oleh perusahaan telekomunikasi, korporasi multinasional, lembaga keuangan, *broadcast* televisi dan radio.

Teknologi *satellite* mulai termarjinalisasi oleh teknologi *fiber optic*. Kecepatan *fiber optic* untuk mengirimkan suara, video dan *traffic* data secepat cahaya (299,782 km/s).[15]

Namun begitu salah satu penyedia *Cloud Service Provider* seperti *Amazon Web Services (AWS)* telah bekerja dengan ***Federal Communication Commision (FCC)*** untuk meluncurkan 3,236 *broadband satellite*. Amazon berambisi ingin menghubungkan lebih dari 10 juta orang agar dapat terhubung dengan internet.

"*The goal here is broadband everywhere,"* Amazon CEO Jeff Bezos.[16]

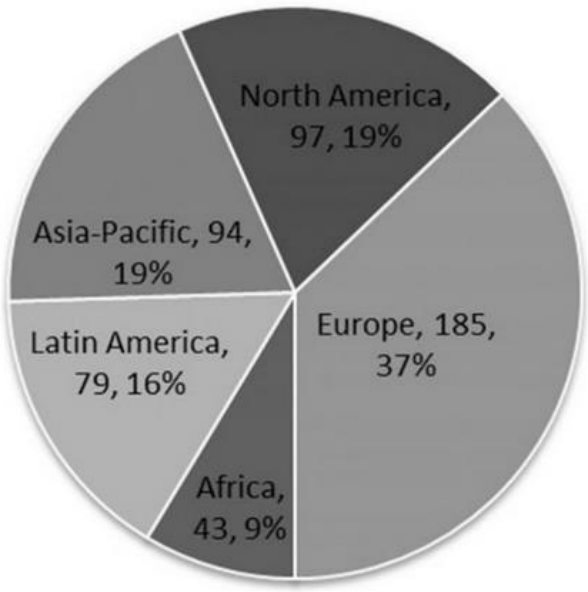
7. *Internet Exchange Point*



Gambar 27 Internet Exchange Point Location

Internet Exchange Point (IXP) adalah sebuah lokasi dimana perusahaan infrastruktur internet seperti *Internet Service Provider (ISP)* dan ***Content Delivery Network (CDN)*** terhubung satu sama lain. Pada dasarnya *Internet* dikendalikan, dimonitor dan dikontrol oleh suatu pemerintah, penguasa, pemilik modal melalui *Internet Service Provider* (Penyedia Layanan Internet). Setiap *ISP* terhubung melalui *Router* besar yang disebut dengan *IXP* (*Internet Exchange Points*) atau sering juga disebut dengan *NAP* (*Network Access Point*), pada gambar di atas lokasi *IXP* ditandai dengan ikon bulat berwarna hijau tua.

Setiap *IXP* di berbagai negara terhubung satu sama lain sehingga kita bisa mengakses *Web Resources* yang tersimpan di dalam setiap *web server* yang ada diseluruh dunia.



Gambar 28 Jumlah Internet Exchange Point Tahun 2017[17]

8. *Content Delivery Network (CDN)*

Content Delivery Network (CDN) atau ***Content Distribution Network (CDN)*** sebuah sekumpulan *server* yang secara geografis saling terhubung di berbagai negara untuk menyediakan layanan penyajian konten internet yang sangat cepat.

Content Delivery Network (CDN) menyediakan layanan untuk menyajikan *file static* seperti *HTML, CSS, Javascript, Image & Video* agar *website* yang dimiliki oleh suatu entitas menjadi lebih cepat. Akses menjadi lebih cepat dengan cara mengirimkan konten dari *server* yang lokasinya terdekat dengan pengunjung situs.

Beberapa perusahaan besar juga sudah menggunakan *Content Delivery Network (CDN)* seperti *Facebook, Netflix* dan *Amazon*. Salah satu penyedia layanan *Content Delivery Network (CDN)* adalah *cloudflare*.

9. *Cloud Computing*

Apa sih ***Cloud Computing*** itu?

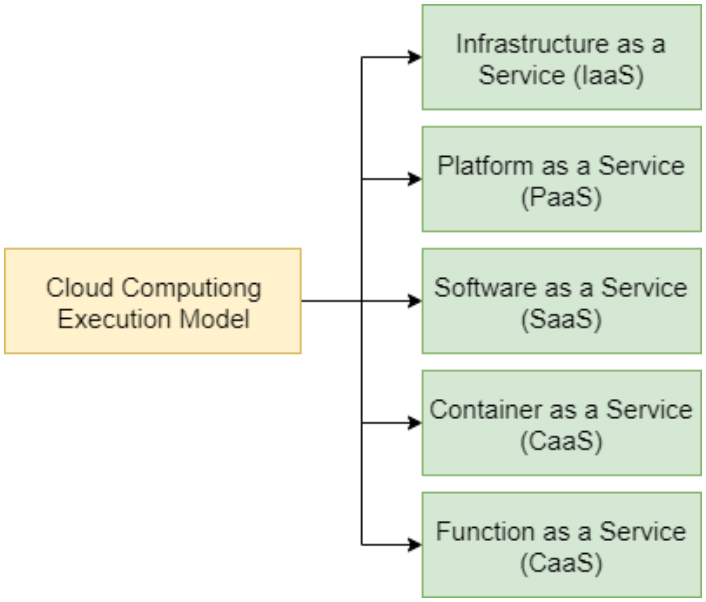
Penjelasan yang ideal untuk menjelaskan cloud computing adalah dengan memberikannya sebuah terminologi yang disebut dengan *Everything as a Service*, disingkat XaaS[18].

Untuk memahaminya secara historis, kita *flashback* lagi pada tahun 1980-2010 terdapat *trend* beberapa perusahaan besar membeli perlengkapan *IT Infrastructure* secara masif dan individual membeli satuan untuk keperluan dirumah.

Namun hari ini adalah era ***ubiquitous computing*** banyak sekali masyarakat yang melakukan akses internet menggunakan *smartphone*. Proses pengolahan dan penyimpanan data tidak dilakukan dalam *mobile device* tetapi pada infrastruktur yang terhubung jarak jauh dan dikendalikan secara *remote*. Inilah awal dimana terminologi ***cloud*** digunakan. Sebuah revolusi besar dalam dunia IT, dari era *mainframes*, ke era *personal computer* hingga akhirnya ke era *cloud*.

Cloud computing memperkenalkan *pay-per-use model* dan membuat abstraksi terhadap *physical server* menggunakan *virtual machine*.

Cloud Computing Execution Model



Gambar 29 Cloud Computing Execution Model

Terdapat 4 macam ***Execution Model*** di dalam *cloud computing* :

Infrastructure as a Service (IaaS)

IaaS menyediakan layanan yang menjadi fondasi dasar sebuah *cloud computing*, terdiri dari *virtual machine*, *storage, network* dan lain-lain.

Platform as a Service (PaaS)

PaaS menyediakan *platform* yang dapat dikembangkan oleh *developer* untuk membuat sebuah aplikasi. *PaaS* menyederhanakan, mempercepat, dan menurunkan biaya yang terkait dengan proses *developing, testing, & deploying application* dengan cara menyembunyikan beberapa *detail* seperti *server management, load balancer* dan *database configuraion*.

PaaS dibangun di atas *IaaS* dengan cara menyembunyikan *infrastructure* dan *operating system* sehingga *developer* dapat lebih fokus untuk menyediakan *business value* dan meringankan beban operasi pengembangan.

Salah satu *PaaS* adalah ***Heroku, Google App Engine*** dan *AWS Elastic Beanstalk*.

Software as a Service (SaaS)

SaaS menyediakan *software* secara lengkap yang dapat digunakan untuk memenuhi kebutuhan kita secara spesifik, contoh layanan *Gmail* yang kita gunakan untuk berkomunikasi melalui *email*.

Container as a Service (CaaS)

Container as a Service (CaaS) menjadi popular saat *docker* pertama kali dirilis ke publik pada tahun 2013. Aktifitas *build* dan *deploy containerized application* pada layanan *cloud computing* menjadi lebih mudah.

Paradigma penggunaan *virtual machine* per *application* diubah dengan cara membangun *multiple container* yang berjalan dalam sebuah *virtual machine* tunggal. Sehingga pemanfaatan *server* menjadi lebih optimal dan mereduksi biaya.

Untuk meraih kemampuan *fault-tolerance, high-availability* dan *scalability* sebuah *orchestration tool* seperti *docker swarm, kubernetes* atau *apache mesos* digunakan untuk memanajemen sekumpulan *container* di dalam sebuah *cluster*.

Hal ini membuat layanan ***CaaS*** diperkenalkan kepada publik agar bisa melakukan *build, ship* dan *run container* secara cepat dan efisien. Juga pekerjaan berat seperti *cluster management, scaling, blue/green deployment, canary update* dan *rollbacks*.

Function as a Service (FaaS)

FaaS menyediakan layanan yang membuat *developer* dapat mengeksekusi sebuah *function (code)* tanpa memikirkan server. Bagaimana mengeksekusi sebuah *function* tanpa harus memanajemen infrastruktur yang sangat kompleks.

Bisnis dapat berkembang tanpa membuat *developer* khawatir permasalahan *scaling* dan pemeliharaan infrastruktur yang sangat kompleks. Paradigma inilah yang membuat istilah ***serverless*** digunakan.

Cloud Service Provider akan melakukan *deployment code* yang dibuat oleh *developer*. Tanggung jawab seperti *provisioning, maintaining* dan *patching* berpindah tangan dari *developer* kepada pihak ***cloud service provider***.

Hal ini membuat *developers* dapat fokus membangun fitur pada aplikasi dan akan membayar waktu komputasi yang telah dikonsumsi.

Cloud Service Provider

Apa itu ***Cloud Service Provider***?

Mereka adalah perusahan besar yang memberikan layanan *computing*, *storage* dan *network resources*. Di antaranya adalah perusahaan *Google, Microsoft* dan *Amazon*. Penggunaaan *cloud* telah mengubah *landscape* dunia Industri IT, menuntut perubahan besar dalam pembangunan infrastruktur, *business planning, software development, computer security* dan tingginya tenaga kerja yang dapat mengoperasikan teknologi *cloud*.

Scalability

Scalability atau skalabilitas adalah kemampuan kinerja suatu sistem untuk tidak terpengaruh ketika ukuran sistem bertambah semakin besar (*increase*) atau berkurang semakin kecil (*decrease*).

Sebagai contoh kita memiliki *web server* yang biasanya menerima *request* rata-rata per menit sebesar 1000 *request* dan merespon dengan waktu akses 30ms, lalu *web server* yang kita miliki menerima *request* rata-rata 2000 *request* per menit dan merespon dengan waktu akses 1000 ms. Hal seperti ini sangat tidak diharapkan.

Terdapat 3 acuan bagaimana suatu sistem berskala :

1. ***Constant Scaling***

   Mengacu pada *performance system* yang tetap atau tidak berubah meskipun beban (*workload*) meningkat. Pada kasus di atas jika ingin mencapai *constant scaling*, sistem harus tetap merespon pada laju 30ms setiap kali ada permintaan (*HTTP Request*).

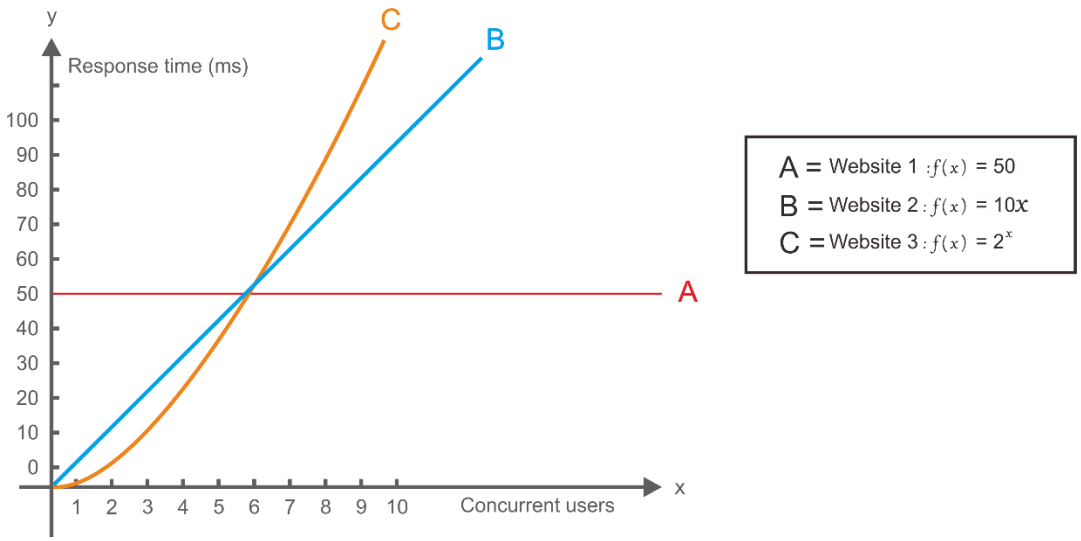
2. ***Linear Scaling***

   Mengacu pada *performance system* yang berubah secara proporsional mengikuti beban (*workload*) yang diterima. Pada kasus di atas jika beban permintaa meningkat dua kali lipat maka kita akan mengetahui laju respon mencapai 60ms.

3. ***Exponential Scaling***

   Mengacu pada *performance system* yang berubah secara tidak proporsional (*disproportionately*) ketika mendapatkan beban (*workload*) yang diterima. Pada kasus di atas kita menghadapi permasalahan *exponential scaling*.

Di bawah ini adalah contoh diagram perbandingan 3 *functions* untuk *modelling* waktu *response* 3 buah *website*. Terdapat 3 *website* yang kita beri label, *website* 1, *website* 2 dan *website* 3.



Gambar 30 Scaling Chart

Pada sumbu X digunakan untuk melihat bagaimana performa *website* menghadapi ***scaling factors*** ketika jumlah pengguna terus meningkat dari 1 hingga 10. Pada *website* 1, waktu respon dijelaskan menggunakan fungsi f(x) = 50. Dikarenakan nilai selalu bernilai 50, maka fungsi merepresentasikan *constant scaling*.

Pada *website* 2, waktu respon dijelaskan menggunakan fungsi f(x) = 10x. Fungsi ini memberikan contoh *linear scaling* waktu respon adalah 10 kali dari jumlah *user*. Satu *user* memerlukan waktu respon sebesar 10 ms, sementara 5 *users* memerlukan waktu respon sebesar 50 ms.

Pada *website* 3, waktu respon dijelaskan menggunakan fungsi f(x) = 2^x. Fungsi ini memberikan contoh *exponential scaling*. Dari kurva (*curve*) yang dihasilkan waktu respon yang diperlukan meningkat secara *double*.

Skalabilitas mengacu pada kemampuan sebuah web untuk beradaptasi dengan meningkatnya permintaan. Skalabilitas bukan tentang bagaimana cara membuat sistem menjadi lebih cepat. Skalabilitas fokus pada bagaimana performa sistem tetap berjalan dengan baik saat kita menambahkan kemampuan komputasi, penyimpanan dan sumber jaringan untuk meningkatkan kapasitas ketika *demand* telah muncul.

Suatu sistem yang performanya terus meningkat secara proporsional mengikuti *demand* di sebut dengan *scalable system*. Ada beberapa cara untuk melakukan *scaling* :

Vertical Scaling

Pada ***vertical scaling*** perangkat keras (*hardware*) yang lebih besar dan tinggi digunakan untuk mengganti perangkat keras lama.

Sebagai contoh sebuah perusahaan pertama kali akan menggunakan *low-end server* misal hanya untuk menampilkan halaman *website*. Namun, ketika pengunjung semakin banyak dan *workload* semakin meningkat maka kapasitas *server* sudah tidak mampu lagi menerima permintaan maka *low-end server* akan diganti dengan *mid-end server*, aktivitas *vertical scaling* ini dilakukan sampai menuju *high-end server*.

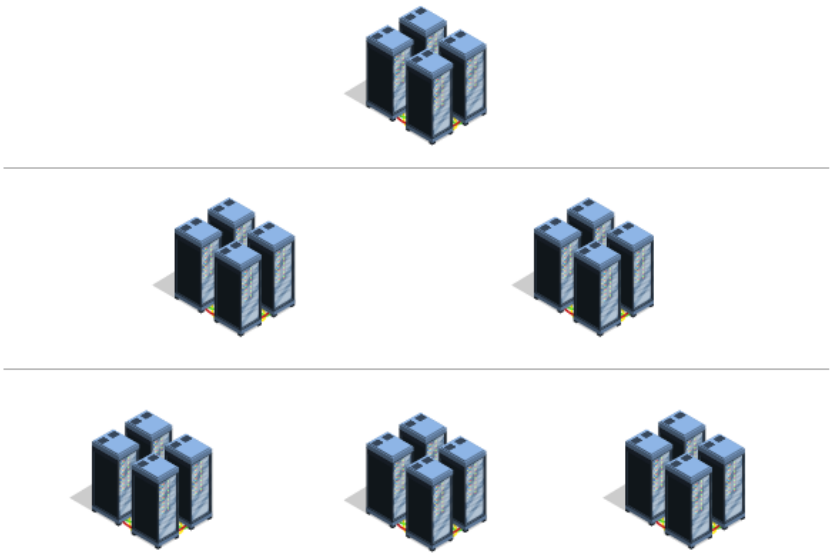
Gambar 31 Vertical Scaling

Horizontal Scaling

Pada ***Horizontal Scaling*** perangkat keras (*hardware*) tambahan digabungkan bersama dengan perangkat keras (*hardware*) lama (*pool of resources*) yang sudah ada.

Sebagai contoh sebuah perusahaan pertama kali akan menggunakan *low-end server* misal hanya untuk menampilkan halaman *website*. Namun, ketika permintaan semakin

banyak dan *performance* mulai mengalami degradasi maka perusahaan akan membeli kembali *low-end server* dengan tipe dan kapasitas yang sama. Beban kerja (*workload*) selanjutnya dilayani oleh dua *low-end server* sekaligus.



Gambar 32 Horizontal Scaling

Autoscaling

Pada *Horizontal* dan *Vertical Scaling* eksekusi *scaling* dilakukan tergantung dari ***administrator*** yang ingin mencabut atau menambahkan *resources* untuk menghadapi perubahan *demand*. Menambahkan *resources* artinya membeli perangkat keras (*hardware*) yang baru, menambah beban baru sebab perlu dikonfigurasi dengan benar dan memiliki keamanan yang baik.

Melalui ***Autoscaling*** sebuah *web server* dapat secara otomatis meningkatkan atau menurunkan *resources* yang digunakan sesuai dengan permintaan (*demand*). Inovasi *autoscaling* memberikan *awareness* untuk menghindari ***over-provisioning*** yaitu penggunaan *resources* yang berlebihan melebihi kebutuhan dan memberikan

awareness untuk menghindari ***under-provisioning*** yaitu penggunaan *resources* yang sangat buruk sehingga *performance* menjadi lamban.

Over-provisioning

Pada *over-provisioning* kita memiliki lebih banyak perangkat keras (*hardware*) yang dibutuhkan sehingga terdapat *resources* yang sia-sia dari waktu ke waktu.

Under-provisioning

Pada *under-provisioning* perangkat keras (*hardware*) tidak memiliki *resources* yang mendukung kebutuhan (*demand*) *client* sehingga beresiko ditinggalkan *client*.

Load Balancer

Load balancer digunakan ketika kita ingin melakukan *routing* sebuah *request* ke salah satu *server* dalam grup *application server* yang kita miliki. Masing-masing *application server* adalah kloningan *image* yang sama agar bisa memperlakukan *request* dan memberikan *response* yang sama. *Load balancer* membantu layanan yang kita miliki agar terhindar dari *request* yang *overload* dengan cara mendistribusikannya ke dalam sekumpulan *server* yang masih *available*.

10. *Serverless Computing*

Serverless computing adalah sebuah *execution model* yang menjadi paradigma baru dalam *cloud computing*. *Serverless* menjadi sebuah model yang membuat masalah *configuration, maintaining,* dan *updating server* bukan lagi bagian dari tugas dan fokus seorang *developer*. *Server* tetap ada namun dikelolah oleh pihak *Cloud Service Provider*. Seperti yang telah di jelaskan sebelumnya dalam kajian tentang *FaaS (Function as a Service)*.

FaaS Provider

Ada beberapa *cloud service provider* yang memberikan layanan *FaaS* diantaranya adalah :

1. *AWS Lambda*
2. *Google Cloud Function*
3. *Microsoft Azure Function*

Karena buku ini didedikasikan untuk salah satu *cloud service provider* yaitu *AWS* maka penulis hanya akan membahas tentang *AWS Lambda*. Mungkin di edisi berikutnya penulis akan mencoba mengeksplorasi *Google Cloud Function* dan *Microsoft Azure Function*.

AWS Lambda

Pada layanan *AWS*, *Lambda* menjadi salah satu layanan yang menyediakan *serverless computing* dimana *developer* dapat memuat kode yang mereka tulis. *Lambda* menggunakan *event-driven architecture*. Kode milik *developer* akan di *trigger* ketika terdapat respon terhadap suatu *event* dan dieksekusi secara *parallel*.

Biaya yang akan kita bayar adalah per eksekusi dengan biaya sebesar 0.20 USD untuk setiap 1 juta *request*, berbeda jika kita menggunakan *EC 2* yang membuat kita akan dikenakan biaya perjam.

Subchapter 3 – Bedah Konsep *HTTP*

Books are slow, books are quiet, the internet is fast and loud.
— Jonathan Safran Foer

Subchapter* 3 – *Objectives

- Mengenal Apa itu ***HTTP & URL*?**
- Mengenal Apa itu ***HTTP & DNS*?**
- Mengenal Apa itu ***HTTP Transaction*?**
- Mengenal Apa itu ***HTTP Request*?**
- Mengenal Apa itu ***HTTP Response*?**
- Mengenal Apa itu ***HTTP Status Message*?**

1. *HTTP & URL*

HTTP

Apa itu ***HTTP*** ? *HTTP* adalah singkatan dari *Hypertext Transfer Protocol*. Sebuah *protocol* dalam *application layer* pada *standarized model* jaringan komputer *TCP/IP* yang digunakan untuk distribusi & kolaborasi sistem informasi ***hypermedia***.

Sebelum membahas apa itu *hypermedia* kita akan membahas terlebih dahulu apa itu *hypertext*?

Hypertext & Hyperlink

Kemudian apa itu ***hypertext*** ? Sebuah *text* yang di dalamnya terdapat *text* yang "terhubung" dengan *text* yang lainya dalam dokumen yang berbeda, kata terhubung ini disebut dengan ***hyperlink*** yang artinya memberikan *link* agar data *text* dalam dokumen yang lainnya bisa diakses. Konsep *hypertext* sendiri telah eksis pada tahun 1941 sebelum **Tim-Berners Lee** menciptakan *protocol HTTP* pada tahun 1989. Pada akhirnya konsep

hypertext diakuisisi sehingga tercipta bahasa *markup* yang kita kenal hari ini yaitu ***HTML*** (*Hypertext Markup Language*).

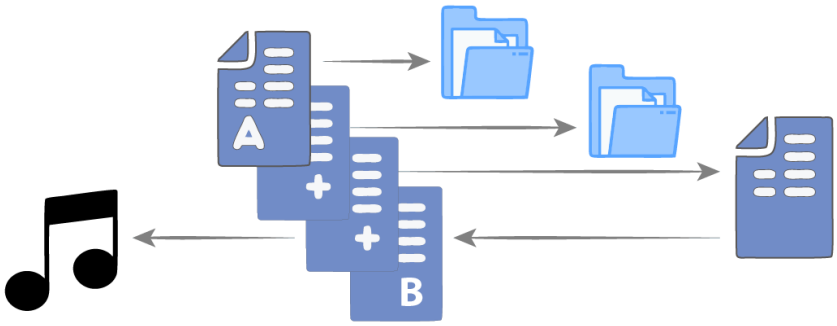
HTML

HTML diciptakan agar kita bisa mengakses halaman *web* yang satu ke halaman *web* yang lainya.

Hypermedia

Sebelumnya pada *hypertext* kita dapat berpindah dari satu teks dokumen ke teks dokumen lainya. ***Hypermedia*** adalah pengembangan lanjut yang lebih mutakhir, data *graphics, audio, video, plaintext* dan *hyperlink* menjadi elemen yang dapat diakses dari dalam dokumen.

Apa itu *Hypermedia***?** *Hypermedia* adalah sebuah ***nonlinear medium*** yang terdiri dari *graphics, audio, video, plaintext* dan *hyperlink*. Kenapa disebut *nonlinear medium*? Karena kita bisa membuka seluruh data dalam *medium* dengan bebas dan acak tanpa harus berurutan (*sequential*).



Gambar 33 Akses Secara Nonlinear

Terminologi ini pertama kali disebutkan oleh Fred Nelson pada tahun 1965.

World Wide Web (www)

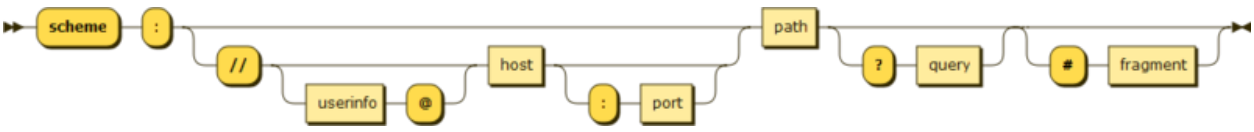
HTTP adalah fondasi untuk komunikasi data dengan ***world wide web*** atau lebih sering dikenal dengan singkatan (*www*). Lalu apa sih *world wide web* itu sendiri?

World wide web adalah sebuah dunia maya (*cyberspace*) dimana sekumpulan dokumen dan sumber *web* lainya bisa dikenali melalui salah satu ***Uniform Resources Identifier (URI)*** yang disebut ***Uniform Resources Locator*** (*URL*), diakses melalui *internet* pada jaringan *Internet Service Provider* yang kita gunakan.

Uniform Resources Identifier (URI)

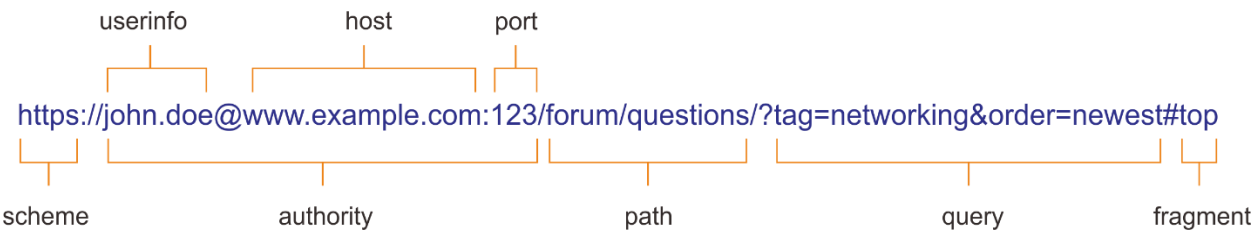
Uniform Resources Identifier (URI) adalah sekumpulan *character* yang membentuk *string* untuk identifikasi suatu ***Web Resources*** di dalam *world wide web (www)*. Agar bisa diterima secara universal terdapat aturan yang harus diikuti. Aturan yang dibuat menentukan ***protocol*** yang akan digunakan untuk mengeksplorasi *web resources* di dalam *world wide web (www)*.

Dibawah ini adalah *syntax diagram* untuk merepresentasikan URI :



Gambar 34 URI Syntax Diagram

Contoh lengkapnya :



Gambar 35 URI Example

Salah satu bentuk dari *Uniform Resources Identifier (URI)* adalah *Uniform Resources Locator (URL)*. *Uniform Resources Locator (URL)* adalah subset dari *Uniform Resources Identifier (URI)*.

URL / Web Resources

Untuk mengakses sebuah *URL* kita memerlukan ***web browser***, di bawah ini adalah struktur sebuah *URL* :

http://www.maudy-ayunda.co/data/profil.html

Terdapat susunan *URL* yang terdiri dari :

Protocol

Pada contoh *URL* di atas yang digunakan yaitu *http* yang ditandai dengan warna biru.

Hostname

Pada contoh URL di atas yang digunakan yaitu www.maudy-ayunda.co yang ditandai dengan warna merah.

Path

Pada contoh URL di atas yang digunakan yaitu /data/ yang ditandai dengan warna hijau.

Filename

Pada contoh URL di atas yang diakses yaitu profil.html yang ditandai dengan warna hitam.

URL seringkali disebut dengan *web address*. *URL* dapat dikenali lokasinya dan menjadi referensi sebuah *web resources* (sumber *web*) pada sebuah jaringan komputer.

Dalam skala *world wide web* akses URL antar negara akan melalui *internet exchange point* yang menghubungkan berbagai jaringan komputer ke seluruh dunia melalui *Internet Service Provider* yang anda gunakan.

Pada tahun 2012, sudah terdapat lebih dari tiga trilliun *web resources* yang terhubung melalui *internet*. Setiap *web resources* yang terhubung dalam *internet* memiliki *URL* sebagai tanda pengenalnya.

Terdapat 4 Istilah yang perlu kita ketahui tentang *URL* yaitu :

Port

Port adalah jalur yang digunakan untuk berinteraksi. Sebagai contoh sebuah halaman bisa diakses dengan menambahkan informasi *port* yang digunakan kedalam *URL* :

http://www.maudy-ayunda.co**:80**/data/profil.html

Secara *default* standarnya *port* yang digunakan adalah *port* 80 namun kita bisa mengabaikan untuk tidak mengisinya. Biasanya *port* digunakan saat kita hendak melakukan *testing, debugging, maintenance* dan *assessment port* dalam *web server*.

Query

Di dalam *URL* juga terdapat istilah *query* yang digunakan setelah tanda **'?'** atau disebut dengan ***question mark***. Dengan format ***name – value pair***, sebagai contoh dalam *URL* terdapat :

http://www.maudy-ayunda.co**/**search**?q=album**

Pada *query string* di atas q adalah *name* dari variabel yang digunakan dan album adalah *value* yang dimilikinya. Di dalam *query string* terdapat informasi yang akan dikirimkan pada *web server* untuk diproses oleh *web server* agar bisa mengetahui apa yang anda minta. Kita bisa mengirimkan lebih dari satu *query string* dengan menambahkan simbol & (***ampersand***) seperti di bawah ini :

http://www.maudy-ayunda.co/search?**q=album**&**z=lagu**

Fragments

Sebuah *fragment* tidak diproses oleh *web server*, sebuah *fragment* akan di proses oleh *web browser* yang kita gunakan. Sebagai contoh di bawah ini terdapat sebuah *fragments* :

http://www.maudy-ayunda.co/search?z=lagu#perahukertas

Pada *URL* di atas kita akan menuju sebuah *HTML element* yang memiliki *attribute id* perahukertas.

Encoding

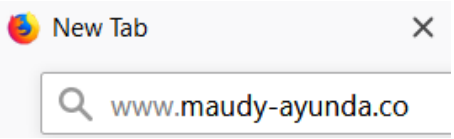
Dalam *URL* juga terdapat istilah ***unsafe character***. Sebuah *character* yang tidak direkomendasikan untuk digunakan untuk *URL*. Sebagai contoh pengunaan simbol # dalam *URL*, sebab telah digunakan untuk membuat *fragments*. Simbol lainya adalah simbol ^ dan *space*. Meskipun begitu kita tetap bisa mentransmisikan *unsafe character* dengan tehnik *percent-encoding* dalam *US-ASCII*. Perhatikan *URL* Di bawah ini :

http://www.maudy-ayunda.co/**%5**Data**%20**saya.txt

Terdapat *string* %20 yang menjadi representasi spasi jika anda ingin membuat *URL* yang menuju ke sebuah *file* dengan nama "^Data saya.txt"

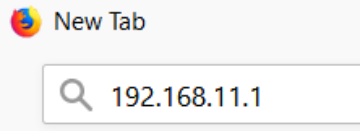
2. *HTTP & DNS*

Peran DNS dapat kita rasakan ketika kita mencoba membuka suatu halaman *website* pada *web browser* melalui *internet*, tentu *hostname* digunakan karena lebih mudah diingat daripada *IP Address*. Misal www.maudy-ayunda.co lebih mudah di ingat daripada 192.168.11.1 :



Gambar 36 Domain

DNS akan mengubah *human readable URL* ke dalam *IP Address*:



Gambar 37 IP Address

IP Address

Setiap *hostname* mempunyai representasi dalam bentuk ***IP Address*** yang regulasinya diatur oleh sebuah badan yang disebut dengan ***Internet Assigned Number Authority*** *(IANA)* dan ***Regional Interest Registries*** *(RIR)*. Setiap penyedia layanan ISP diberbagai negara terhubung dengan kedua lembaga tersebut agar bisa menyediakan layanan internet.

IP Address adalah pengenal untuk setiap perangkat komputer yang terhubung dalam jaringan komputer *TCP/IP*. Baik itu dalam *IPv4* atau *IPv6* (anda bisa mempelajarinya lebih lanjut dalam ilmu jaringan komputer).

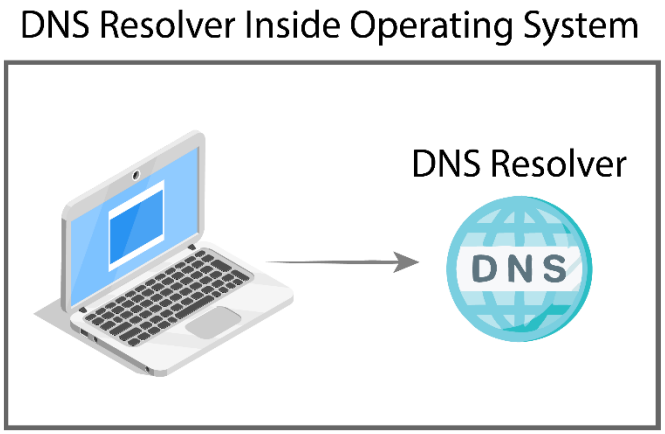
Pernahkah oleh anda terbayang secara ilmiah bagaimana bisa sebuah data yang berada di dalam *web server* bisa muncul dalam *web browser* milik kita?

Ada beberapa proses yang terjadi sebelum berinteraksi dengan *web server*. Tentu sebuah koneksi antara *client* dan *server* harus 'didirikan' terlebih dahulu agar keduanya bisa saling berkomunikasi. Lokasi *web server* sendiri bisa berada dalam sekup intranet atau internet.

DNS Resolver

Web Browser akan terlebih dahulu berinteraksi dengan ***DNS Resolver*** dalam sistem operasi yang kita gunakan baik itu *windows, linux* ataupun *mac*. DNS adalah kependekan dari ***Domain Name System*** yang tugasnya mengubah suatu *hostname* kedalam bentuk *IP Address*. Jika terdapat ***local cache*** untuk *IP Address* dari alamat *website* yang ingin kita buka, akses akan lebih cepat.

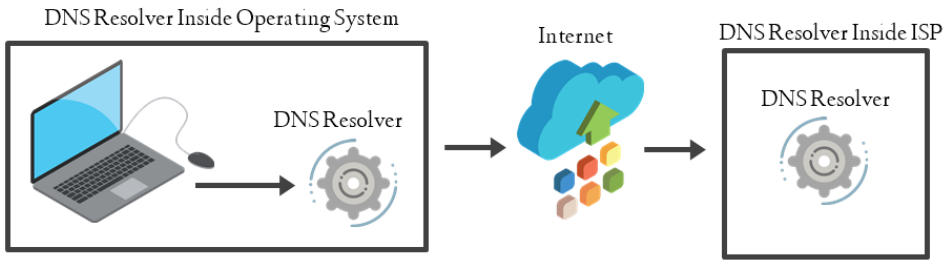
IP Address dari suatu alamat *website* akan tersimpan di dalam *DNS Resolver* pada sistem operasi kita jika sebelumnya kita telah membuka suatu *website*.



Gambar 38 DNS Resolver dalam Sistem operasi

Jika belum tersimpan di dalam *local cache* maka sistem operasi yang kita gunakan akan mengirimkan ***IP Packet,*** yang di dalamnya terdapat alamat IP milik kita dan alamat IP yang akan kita akses beserta *port* yang digunakanya melalui internet. *Secara default port* yang digunakan pada *web server* adalah *port* 80

Kemudian *DNS Resolver* pada *Internet Service Provider* yang kita gunakan akan memeriksa *local cache* milik mereka, apakah ada *IP Address* dari alamat *website* yang hendak kita akses atau tidak. Tentu akses akan lebih cepat jika ada, namun jika tidak maka *DNS Resolver* akan berinteraksi dengan ***Root Server*** yang mengetahui berbagai lokasi ***TLD Server***. TLD adalah singkatan dari ***Top-level Domain***.



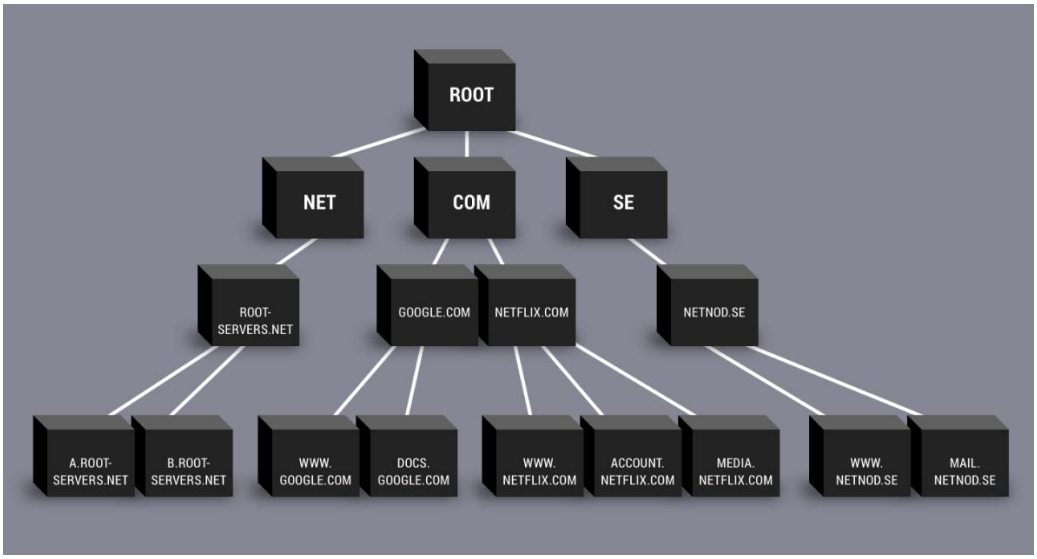
Gambar 39 DNS Resolver dalam ISP

Root Server & TLD Server

Tak banyak materi yang membahas tentang ***root server*** terkecuali jika anda mempelajari sejarah pembangunan internet. *Root Server* adalah tulang belakang yang menjadi penyokokong instrastruktur internet yang kita gunakan bisa berjalan sampai hari ini. *Root server* juga sering kali disebut dengan ***DNS Root Name Server***.

The Internet Corporation for Assigned Names and Numbers (ICANN), adalah entitas yang mengkoordinasikan ***Domain*** dan ***IP addresses*** dalam ***Internet***.

Root Server mengetahui seluruh ***DNS Zone*** yang menyimpan seluruh data ***Top-Level Domain (TLDs)*****,** mulai dari ***Generic TLDs*** seperti (.com, .net, .org, .edu, etc), ***Country Code TLDs*** (.uk, .id, .de, .etc) dan ***Internationalized TLDs*** yang ditulis dalam bahasa china, kanji, tamil dan sebagainya**.** Pada bulan april 2018 sudah terdapat 1534 *top level domain* tercatat.



Gambar 40 Root Server

Lalu apa yang terjadi saat ***Recursive Resolver*** atau ***DNS Resolver*** pada *ISP* yang kita gunakan berinteraksi dengan *Root Server*?

Root Server akan menerima permintaan dari *Recursive Resolver*, respon yang diberikan adalah informasi ***DNS*** dari ***Top-level Domain***. Pada kasus yang diangkat dalam buku ini *top level domain* yang dicari adalah .co (www.maudy-ayunda.co). *Root Server* akan memberikan lokasi TLD Server untuk .co kepada *Recursive Resolver*.

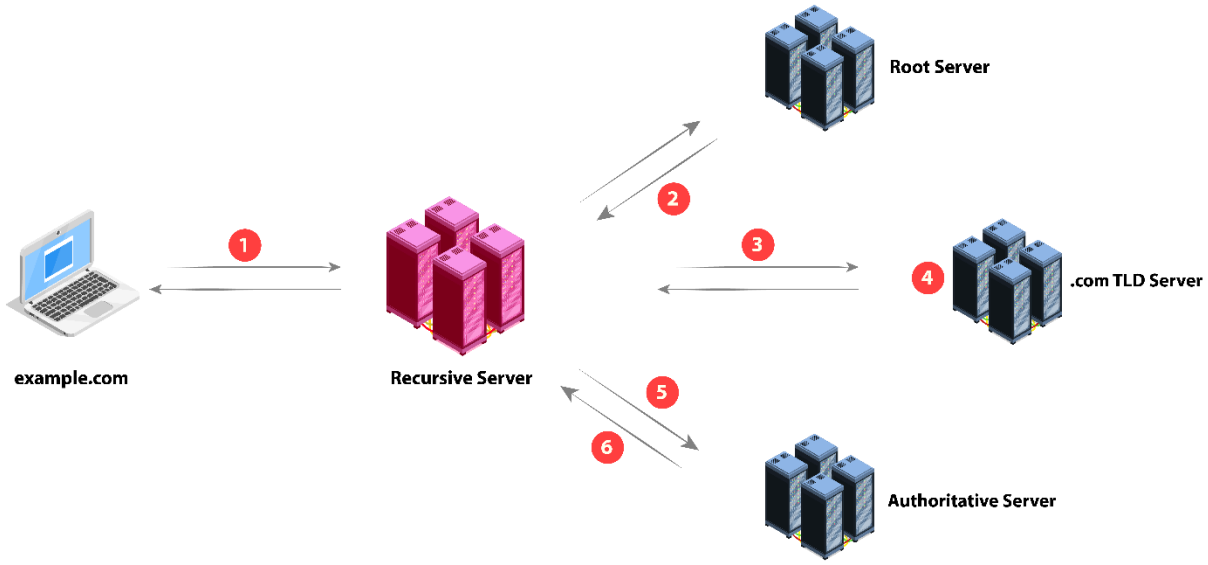
Selanjutnya ***Server TLD .co*** akan memeriksa apakah terdapat *Name Server* dari *hostname* yang hendak kita akses. Sebuah *Name Server* terdiri dari 4 alamat diantaranya adalah :

ns1.maudy-ayunda.co
ns2.maudy-ayunda.co
ns3.maudy-ayunda.co
ns4.maudy-ayunda.co

Informasi ini akan diberikan kepada ***Recursive Resolver*** untuk menemukan ***Authoritative DNS Server*** untuk mancari **maudy-ayunda.co** didalam sebuah *file* bernama ***Zone File***.

Jika di dalam *TLD Server* terdapat *name server* maka dipastikan informasi *IP Address* juga ada, artinya *IP address* dari *hostname* yang hendak kita akses akan tercatat pada ***DNS***

Resolver milik ***Internet Service Provider*** yang kita gunakan. Setelah *IP Address* diketahui maka melalui *ISP* yang kita gunakan, kita bisa menuju alamat *IP* tersebut yang menjadi sebuah *web server*.



Gambar 41 DNS Information

Informasi ***DNS*** dalam ***Zone File*** akan diberikan kembali pada *Recursive Resolver* yang selanjutnya didapatkan oleh pengguna yang mengakses domain maudy-ayunda.co.

Lalu dimana sih lokasi *Root Server*? Mereka ada diberbagai belahan dunia yang manajemenya telah diatur oleh sebuah organisasi. *Root servers* dioperasikan oleh 13 organisasi berbeda[20] :

1. *VeriSign Global Registry Services*
2. *University of Southern California, Information Sciences Institute*
3. *Cogent Communications*
4. *University of Maryland*
5. *NASA Ames Research Center*
6. *Internet Systems Consortium, Inc.*
7. *US DoD Network Information Center*

8. *US Army Research Lab*
9. *Netnod*
10. *VeriSign Global Registry Services*
11. *RIPE NCC*
12. *ICANN*
13. *WIDE Project*

Hampir sebagian besar organisasi yang mengendalikan *root server* telah terlibat semenjak konsep revolusioner *DNS* diciptakan di dalam dunia jaringan komputer. Dari sisi sejarah internet didesain untuk keperluan militer dan yang paling tua adalah ***US Army Research Lab***.

3. *HTTP Transaction*

Setelah mendapatkan *IP Address* dari *hostname* yang ingin kita akses melalui *DNS Server* milik *Internet Service Provider*, selanjutnya kita baru bisa berinteraksi dengan *web server*. Namun sebelum kita melakukan pertukaran data antara *client* and *server* (akses *web resources*), koneksi harus didirikan terlebih dahulu.

TCP Thee-way Handshake

HTTP didukung oleh *protocol TCP* jadi ada 3 fase yang terjadi sebelumnya yaitu *TCP* ***Three-way Handshake*** :

Connection Setup

Dilakukan untuk mendirikan jembatan koneksi.

Gambar 42 TCP Threeway Handshake

Client akan mengirimkan ***SYN Message*** terlebih dahulu kepada *server* sebagai tanda untuk membuka koneksi, kemudian *server* akan memberi respon berupa sinyal *SYN* dan *ACK* (*Acknowledgement*) *Message* sebagai tanda bahwa permintaan untuk mendirikan koneksi diterima dan terakhir *client* akan mengirimkan sinyal ***ACK Message***.

SYN dan *ACK Message* adalah sinyal unik dalam ***Header TCP Segment*** yang bisa dikirimkan oleh *client* & *server* untuk mengetahui status koneksi. Setelah melalui *TCP Threeway Handshake* koneksi akan terbuka agar proses *data exchange* bisa dilakukan.

Data Exhange

Dilakukan antara *client* dan *server*.

Gambar 43 Data Exchange Process

Disini terdapat istilah ***request-response*** dimana kita melakukan permintaan dan *web server* memberikan respon yang selanjutnya diterima oleh *web browser* kita. Mekanisme *request-response* ini selanjutnya disebut dengan ***HTTP Request*** dan ***HTTP Response***, jika proses *request-response* sudah selesai kegiatan ini disebut dengan ***HTTP Transaction***. Respon akan di *parse* oleh *web browser* kita sehingga bisa tampil yaitu halaman *website* www.maudy-ayunda.co.

Selanjutnya *DNS Resolver* dalam sistem operasi kita akan menyimpan *IP address* dari *hostname* yang telah kita akses dalam sebuah *local cache* agar akses pada suatu halaman *website* bisa menjadi lebih cepat.

Connection Termination

Untuk menutup kembali koneksi.

Setiap kali kita telah selesai melakukan *transaction* yaitu mendapatkan halaman yang kita inginkan maka koneksi *TCP* akan ditutup. *Client* akan mengirimkan sinyal *FIN* yang dibalas oleh *server* dengan sinyal *ACK*, dilanjutkan dengan pengiriman kembali *ACK* dan *FIN*. Terakhir *client* membalas dengan mengirimkan sinyal *ACK*.

Furthermore jika anda ingin lebih tahu secara detail lagi kajian *computer network* yang mengalami *intersect* dengan buku ini, saya rekomendasikan untuk membaca buku *Cables & Wireless Network – Theory and Practice* (2016) Karya Mario Marquez Da Silva Halaman 237.

4. *HTTP Request*

Apa sih yang ada di dalam *HTTP Request* yang kita kirimkan?

Dan seperti apa pula *HTTP Response* yang kita terima? Bagi seorang *web developer* memahami *message* yang ada di dalamnya adalah sesuatu yang sangat fundamental, agar bisa membuat *web application* yang baik, memahami masalah yang terjadi dan *debug* suatu *issue* ketika *web application* yang dibuat tidak berjalan.

Di dalam sebuah *HTTP Request* terdapat *message* yang telah diformat agar dapat dimengerti oleh *web server*. Sebaliknya *server* juga akan mengirimkan *HTTP Response* yang di dalamnya terdapat *message* yang dapat dimengerti oleh *web browser* milik *client*. Keduanya memiliki pesan berbeda yang diproses dalam ***single transaction***.

Isi di dalam ***HTTP Message*** adalah sebuah *plain ASCII text* yang formatnya mengacu kepada ***HTTP Specification*** yang telah diberi standar.

HTTP Method

Dalam *HTTP Request* terdapat beberapa *method* yang seringkali digunakan diantaranya adalah :

Table 1 HTTP Method

No	***HTTP Method***	***Description***
1	*GET*	Untuk mendapatkan *web resource*
2	*POST*	Untuk update *web resources*
3	*DELETE*	Untuk menghapus *web resources*
4	*PUT*	Untuk menyimpan *web resources*
5	*HEAD*	Untuk mendapatkan *header web resources*

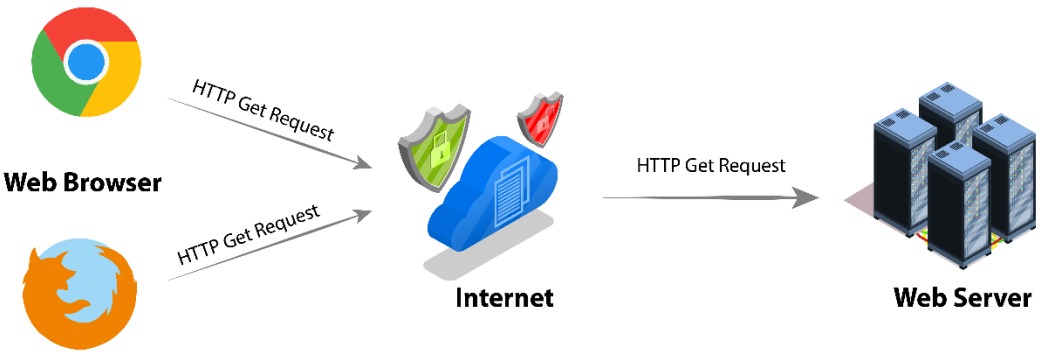
Setiap kali sebuah *HTTP Request* dilakukan di dalamnya harus terdapat salah satu *HTTP Method*.

Dengan ***GET Request*** kita bisa memperoleh *web resources* yang kita inginkan, baik itu berupa halaman, gambar, suara, video dan dokumen. Pada *HTTP data hypermedia* akan di *encoded* kedalam bentuk *text*.

Di bawah ini adalah contoh skema *GET Request* :

```
<a href="http://maudy-ayunda.co">Kunjungi Web Maudy</a>
```

Pada *HTML Element* di atas ketika di *render* oleh sebuah *web browser* akan membentuk sebuah *hyperlink*, jika kita melakukan klik pada *link* di atas *web browser* akan mengirimkan *HTTP GET Request*.



Gambar 44 HTTP Get Request

Lalu seperti apakah *message* di dalam *HTTP Request*?

Anda bisa melihatnya di bawah ini :

```
GET http://maudy-ayunda.co/ HTTP/1.1
Host: maudy-ayunda.co
```

Jika kita ingin mengirim suatu *data* melalui *web browser* kita bisa menggunakan ***POST Request***. Sebagai contoh ketika kita ingin melakuan *login* kedalam sebuah *website* tentu kita harus mengirimkan data akun berupa *username* dan *password*.

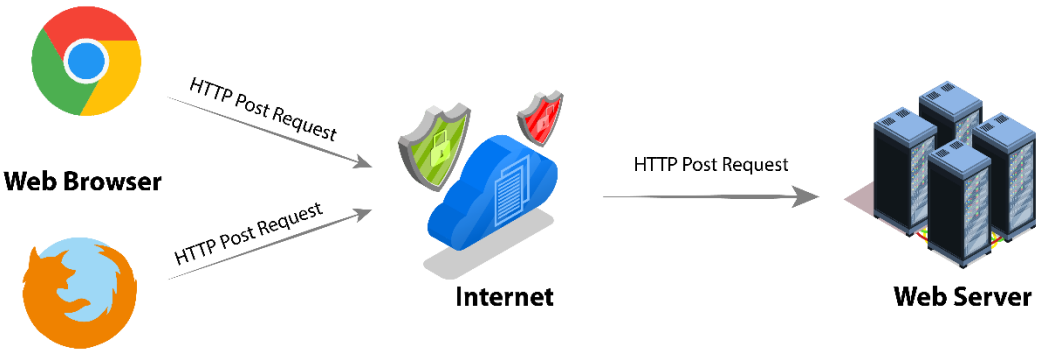
Di bawah ini adalah contoh kode untuk *form* halaman *login* menggunakan PHP :

```
<form action="login.php" method="POST">
<label for="username">Username</label>
<input id="username" name="username" type="text" />
<label for="password">Password</label>
<input id="password" name="password" type="password" />
<input type="submit" value="Login"/>
</form>
```

Ketika seorang pengguna menekan tombol *login*, maka *data* di dalam kedua *input text* tersebut akan dikirim kepada *web server* menggunakan *HTTP Post Request*.

Data akan diproses oleh *script* di dalam *web application* yaitu **login.php**

```
POST http://maudy-ayunda.co/login.php HTTP/1.1
Host: maudy-ayunda.co
username=gun&password=maudy
```



Gambar 45 HTTP Post Request

Message

Adapun *format message* untuk *HTTP Request* yang kita kirimkan sebagai permintaan untuk *web server* adalah sebagai berikut :

```
[http method] [url] [http version]
[header]
[body]
```

Saat pengiriman *message* bentuknya berupa *text* dengan *character encoding* ASCII (*American Standard Code for Information Interchange*). Semenjak tahun 1999 kita sudah menggunakan versi ***http 1.1***, namun semenjak tahun 2015 kita sudah bisa menggunakan ***http versi 2.0*** yang akan dijelaskan dihalaman selanjutnya.

Pada *format message* di atas **`[http method]`** adalah *http method* yang digunakan**,** **`[url]`** adalah target *URL* dan adalah **`[http version]`** versi *http* yang digunakan. Untuk **[header]** sebelumnya kita telah menggunakanya yaitu *host* (salah satu *http header* yang paling dibutuhkan) pada contoh sebelumnya yaitu Host: maudy-ayunda.co dan untuk **`[body]`** bisa berupa data akun untuk *login* seperti yang sudah kita pelajari dihalaman sebelumnya.

HTTP Header

Selain itu di dalam *http header* juga terdapat ***Header Attribute*** yang bisa *client* gunakan untuk meminta *web resources* dalam bahasa asing lainya. Misalnya jika *client* ingin meminta *resources* dalam bahasa jerman maka di dalam *http header* akan disertakan *attribute* (*accept-language*) seperti pada *note* di bawah ini :

```
GET http://maudy-ayunda.co/ HTTP/1.1
Host: maudy-ayunda.co
Accept-Language: de-DE
```

Ada banyak sekali *HTTP Header attribute* yang dijelaskan dalam *HTTP Specification*, beberapa *http header attribute* termasuk kedalam *general header* yang bisa disisipkan di dalam *http message* baik untuk *response* atau *request*. *Client* atau *server* bisa menyisipkan *header attribute date* yang digunakan untuk mengindikasikan kapan *message* tersebut dilakukan.

```
GET http://maudy-ayunda.co/ HTTP/1.1
Host: maudy-ayunda.co
Accept-Language: de-DE
```

```
Date: Sat, 08 April 2017 21:12:00 GMT
```

Header Attribute

Ada beberapa *http header attribute* yang populer dan sering kali muncul yaitu :

Table 2 HTTP Header Attribute

No	***Header Attribute***	***Description***
1	*Referer*	Ketika seorang pengguna melakukan klik pada suatu *hyperlink* dalam suatu halaman, maka URL halaman tersebut akan dicatat dalam *http header*.
2	*User Agent*	Informasi tentang *software* yang melakukan *request*. Di gunakan untuk mengetahui *browser* apa yang digunakan oleh *client* untuk melakukan *request*.
3	*Accept*	Menjelaskan media *type* yang dimana *user agent* akan menerima. Selain itu juga terdapat *accept-language, accept-encoding, accept-charset* dan sebagainya.
4	*Cookie*	Informasi *cookies* yang bisa digunakan *server* untuk mengenali *user*.
5	*If-Modified-Since*	Tanggal ketika *user agent* menerima *web resources*.

Di bawah ini adalah contoh pesan *HTTP Request* secara penuh :

```
GET http://maudy-ayunda.co/ HTTP/1.1
Host: maudy-ayunda.co
Date: Sat, 08 April 2017 21:12:00 GMT
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) Chrome/16.0.912.75
Safari/535.7
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://www.google.com/url?&q=odetocode
Accept-Encoding: gzip,deflate,sdch Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Jika kita lihat beberapa *attribute header* memiliki lebih dari satu nilai, contohnya adalah ***accept header***. Setiap *MIME* yang dapat diterima terdaftar dalam *accept header*. Jika dilihat dalam *HTTP Request* di atas kita dapat menerima *file HTML, XHTML, XML* dan (*/*) segala jenis *hypermedia*.

MIME

MIME adalah singkatan dari (*Multipurpose Internet Mail Extension*), jika dilihat dari namanya seperti sebuah komponen dalam teknologi email. Memang betul, *MIME* secara original awalnya memang didesain untuk *email communication*. *HTTP* masih bergantung pada *MIME* untuk tujuan yang sama sehingga tidak perlu melakukan *reinventing the wheel* menciptakan standard baru untuk mengganti *MIME*.

Penerapannya ketika *client* melakukan *request* sebuah *HTML webpage*, maka *server* akan merespon *HTTP Request* dengan memberikan *HTML Webpage* yang memiliki *attribute header* **"text/html".** Nilai ***text*** artinya berupa *primary media type* yaitu data berupa teks dan ***html*** artinya *media subtype* atau *extension* dari *file*.

Jika *client* melakukan *request* sebuah gambar maka *server* akan merespon *HTTP Request* dengan memberikan gambar yang memiliki *attribute header* **"image/jpeg"** untuk gambar dengan ***format jpg***, **"image/gif"** untuk gambar dengan *format gif*, dan **"image/png"** untuk gambar dengan ***format png***.

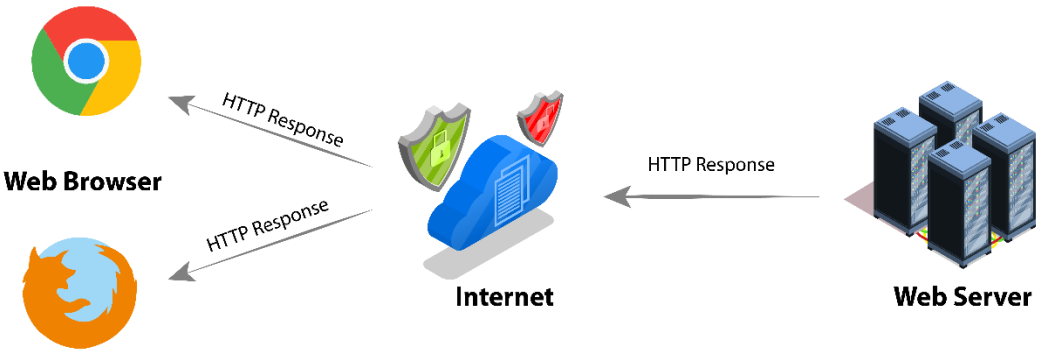
Jika kita perhatikan pada *HTTP Request* di atas terdapat "**q"** dibeberapa *attribute header*. Nilai q akan selalu berada di antara 0 sampai dengan 1 yang merepresentasikan *quality value*. Nilai *quality value* ini didapatkan dari seberapa besar *user* atau *user agent* seringkali melakukan permintaan pada media tersebut di *server*. Semakin sering nilai q akan meningkat dengan maksimum nilai 1.

5. *HTTP Response*

Sebuah *HTTP Response* memiliki struktur yang hampir sama dengan *HTTP Request* :

```
[version] [status] [reason]
[headers]
[body]
```

Setelah ***client*** mengirimkan *HTTP Request* selanjutnya *Server* akan memberikan *HTTP Response* kepada *client*, anda bisa melihat illustrasinya pada gambar di bawah ini :



Gambar 46 HTTP Post Request

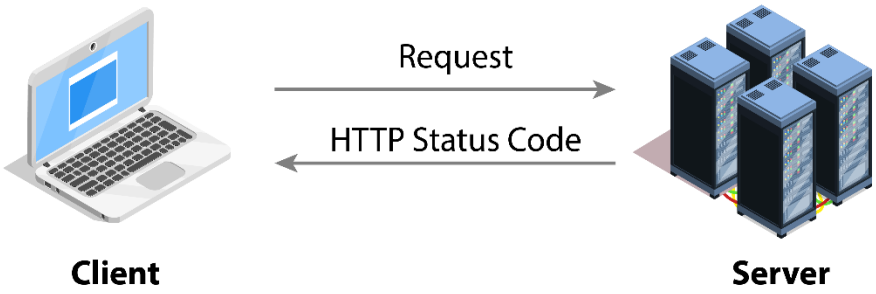
Di bawah ini adalah contoh pesan secara penuh pada *HTTP Response* :

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/7.0
X-AspNet-Version: 2.0.50727
X-Powered-By: ASP.NET
Date: Sat, 08 April 2017 21:12:00 GMT
Connection: close
Content-Length: 17151
<html>
<head>
<title>Halaman Resmi Maudy Ayunda</title>
</head>
<body>
... content ...
</body>
```

```
</html>
```

Pada *HTTP Response* di atas kita mendapatkan *status code* 200 Ok? Apakah itu anda akan mengetahuinya di halaman selanjutnya. Pesan yang diberi warna merah adalah body dari *HTTP Response* berupa *content* yang kita minta dan akan dimuat di dalam *web browser*.

6. *HTTP Status Message*



Gambar 47 Status Code

Saat sebuah *web browser* yang digunakan *client* mengirimkan permintaan (*request*) kepada *web server*, sebuah kesalahan bisa saja terjadi. Informasi adanya kesalahan atau tidak dapat diketahui melalui *HTTP Status Message* yang akan kita terima melalui *HTTP Response*. Di bawah ini adalah *list* beberapa ***HTTP Status Message*** yang secara umum akan di dapatkan *client*.

Table 3 General HTTP Status Message

Status Code	***Reason***	***Description***
200	*Ok*	Sebuah status yang ingin dilihat siapapun, *status code* dengan nilai 200 artinya semua berjalan dengan lancar.
301	*Moved Permanently*	*Web Resources* yang anda akses sudah tidak ada karena berpindah alamat secara permanen dari *URL* lama yang anda akses ke *URL* baru yang lain.

302	*Moved Temporarily*	*Web Resources* yang anda akses telah berpindah *URL* untuk sementara waktu, sehingga *URL* original harus tetap diketahui oleh *client*.
304	*Not Modified*	*Server* akan memberi tahu *client* bahwa *resources* tidak berubah semenjak terakhir kali *client* menerima *resources* sehingga bisa menggunakan data yang telah di *cached* secara *local*.
400	*Bad Request*	*Server* tidak mampu memahami permintaan yang diberikan *client* karena terdapat kesalahan dalam *HTTP Message* yang diberikan.
403	*Forbidden*	*Server* menolak permintaan anda untuk mengakses *web resources*
404	*Not Found*	*Web Resources* yang anda minta tidak ada
500	*Internal Service Error*	*Server* mengalami masalah saat memproses data yang diminta, biasanya karena kesalahan *programming* dalam *application server*.
503	*Service Unvailable*	*Server* tidak mampu memberikan layanan pada permintaan yang diberikan. Ini terjadi ketika *server* mengalami ***throttling*** akibat permintaan yang sangat banyak.

Jika dikategorikan terdapat 5 kategori *HTTP Status Message* :

Table 4 General HTTP Status Message

Status Code	***Category***
100-199	*Informational*
200-299	*Successful*
300-399	*Redirection*
400-499	*Client Error*
500-599	*Server Error*

Anda bisa melihatnya secara detail dalam halaman ***Appendix***, tentang ***HTTP Status Message***.

Subchapter 4 – Web Security

I Trust everyone. It's the devil inside them I don't trust.
— John Bridger

Subchapter 4 – Objectives

- Mengenal **Apa itu *Data in The Low Level*?**
- Mengenal **Apa itu *Cryptography*?**
- Mengenal **Apa itu *Man In The Middle Attack*?**
- Mengenal **Apa itu *HTTPS*?**
- Mengenal **Apa itu *Secure Socket Layer*?**

1. *Data in The Low Level*

Host

Gambar 48 End Point

Host sering kali disebut dengan *end point* adalah sekumpulan *device* yang dapat melakukan komputasi seperti *PC, Server, Laptop* & *Smartphone*.

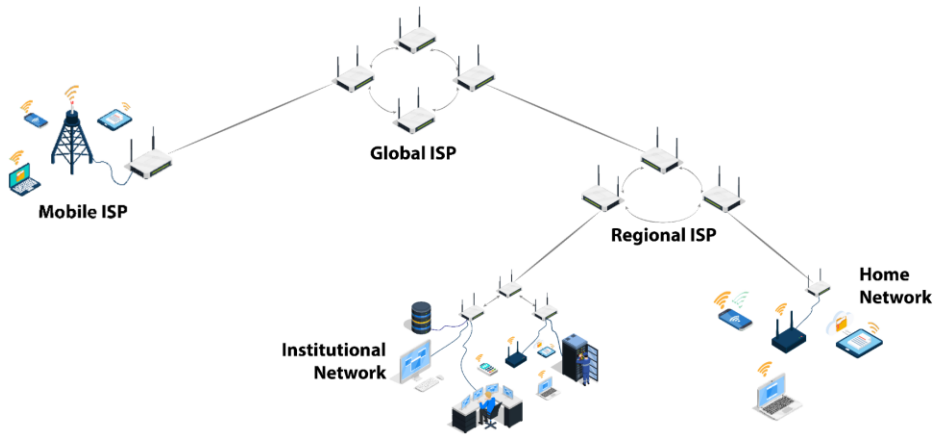
Socket

Internet dibangun menggunakan protokol *TCP/IP*. Internet adalah sebuah jaringan *packet-switching*, maksudnya ketika terdapat dua *host* berbeda ingin berkomunikasi satu sama lain maka data harus di *package* terlebih dahulu menjadi sebuah *packet*.

Setiap *packet* lengkap dengan alamat penerimanya tersimpan dalam *packet header, packet* dapat mengalir hingga ribuan mil keseluruh dunia melewati berbagai sistem komputer *(hops)* di berbagai negara.

Packet akan menemui *router*, selanjutnya *router* akan melakukan analisa pada alamat yang dituju. Proses *routing* dilakukan agar *packet* sampai pada *target host,* dapat melalui sebuah *router* tunggal atau serangkaian *router* terlebih dahulu yang dikalkulasikan lebih dekat menuju lokasi *target host*.

Packet akan melalui berbagai *router* antara jalur pengirim dan penerima. Jika di dalam *packet* terdapat data yang sensitif maka pengirim ingin memastikan bahwa *packet* yang dikirim haruslah aman dan hanya penerima yang berhak membacanya. Sebuah *router* juga memiliki kelemahan yang bisa dieksploitasi agar *traffic data* di arahkan ke penerima yang lain.



Gambar 49 Internet Illustration

Protokol *TCP/IP* sendiri tidak memberikan keamanan yang didesain secara otomatis. Siapapun yang dapat atau memiliki akses ke dalam *communication links* mampu mendapatkan data secara penuh dan mengubah *traffic* tanpa terdeteksi.

Sebelumnya kita telah mempelajari konsep *TCP Three-way Handshake*, istilah *socket* mengacu kepada sebuah koneksi *TCP* yang berhasil dibangun. Kedua belah pihak yaitu

client dan *server* akan memiliki *socket*. Masing-masing pihak akan berkomunikasi melalui jalur ini. Jika *encryption* telah digunakan dan seorang *attacker* mampu mendapatkan data yang telah di *encrypt* namun tidak bisa membacanya dan memodifikasinya.

Untuk mengamankan komunikasi ini *Netscape* mengembangkan *socket* yang memiliki mekanisme keamanan. Mereka membuat *Project SSL* (*Secure Socket Layer*).

Pengembangan ini membawa arah baru dalam dunia pengembangan web, inovasi-inovasi terkait perdagangan dan keuangan tumbuh dengan subur dibangun oleh para *economic agent*.

Sebelum membahas lebih lanjut alangkah lebih baik jika kita membahas beberapa dasar kajian dalam *computer architecture*.

Bit

Bit adalah kependekan dari ***Binary Digit***, unit terkecil sebuah informasi dalam mesin komputer. Satu buah *bit* dapat menampung dua nilai diantaranya adalah **0** atau **1**.

Jika terdapat 8 *bit* maka kita dapat menyebutnya sebagai **1 *byte***.

Byte

Byte adalah kependekan dari ***Binary Term***, sebuah unit penyimpanan yang sudah memiliki kapabilitas paling sederhana untuk menyimpan sebuah karakter tunggal.

1 *byte* = sekumpulan *bit* (terdapat delapan bit). Contoh : 0 1 0 1 1 0 1 0

1 *byte* dapat menyimpan karakter contoh : 'A' atau 'x' atau '$'

Bytes

Byte adalah unit yang dipat digunakan untuk menyimpan informasi, seluruh penyimpanan diukur menggunakan *bytes*.

Table Unit of Data

Number of Bytes	Unit	Representation
1	*Byte*	*One Character*
1024	*KiloByte (Kb)*	*Small Text In notepad*
1,048,576	*MegaByte (Mb)*	*Ebook*
1,073,741,824	*GigaByte (Gb)*	*Movie*
1,099,511,627,776	*TeraByte (Tb)*	*Big Data*

Character

Character adalah unit terkecil dalam sistem teks dan memiliki makna. Sekumpulan *character* dapat membentuk *string* yang selanjutnya dapat digunakan untuk memvisualisasikan suatu bahasa verbal secara digital. Contoh *character* adalah abjad, angka dan simbol lainya.

Gambar 50 Grapheme

ASCII

Komputer merepresentasikan sebuah data dengan *number*, di awal pengembangan komputer tepatnya sekitar tahun 1940. Penggunaan teks dalam komputer untuk disimpan dan dimanipulasi dapat dilakukan, dengan cara merepresentasikan abjad dalam alfabet menggunakan *number*. Sebagai contoh angka 65 merepresentasikan huruf A dan angka 66 merepresentasikan huruf B hingga seterusnya.

Pada tahun 1950 saat komputer sudah semakin banyak digunakan untuk berkomunikasi, standar untuk merepresentasikan *text* agar dapat difahami oleh berbagai model dan *brand* komputer diusung.

ASCII (*American Standard Code for Information Interchange*) adalah karya yang diusung, pertama dipublikasikan pada tahun 1963. Saat pertama kali dipublikasikan *ASCII* masih digunakan untuk *teleprinter technology*. *ASCII* terus direvisi hingga akhirnya *7-bit ASCII Table* diadopsi oleh *American National Standards Institute* (*ANSI*).

Dengan *7-bit* maka terdapat 128 *unique binary pattern* yang dapat digunakan untuk merepresentasikan suatu karakter. Kita dapat merepresentasikan ***alphanumeric*** (abjad a-z, A-Z, angka 0-9, dan *special character* seperti "!@#$%^&*"). Pada gambar di bawah ini huruf kapital G memiliki representasi dalam bentuk biner 100 0111 dan huruf kapital F memiliki representasi dalam bentuk biner 100 0110 :

100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H

Gambar 51 Sample ASCII Code

Pada huruf kapital G angka 107 adalah representasi dalam *octal numeral system*, angka 71 adalah representasi dalam *decimal numeral system* dan angka 46 adalah representasi dalam *hexadecimal*. Representasi tidak hanya dalam bentuk *binary*. Untuk *table ASCII* lebih lengkapnya anda dapat melihat di wikipedia.

Data Transmission

Zaman dahulu saat teknologi *modem* masih tahap pengembangan terdapat masalah jika kita mentransmisikan data biner. Interpretasi yang salah dapat terjadi ketika mentransmisikan data gambar atau *executable file*.

Komputer berkomunikasi menggunakan bahasa biner 1 & 0, namun manusia ingin berkomunikasi dalam bentuk yang lebih luas (*rich form*) seperti teks atau gambar. Untuk dapat mentransmisikan data teks dan gambar tersebut tentu kita harus mengubahnya

(*encoding*) kedalam biner terlebih dahulu, kemudian menejermahkanya kembali (*decoding*) ke dalam teks dan gambar.

Sebelumnya ada banyak sekali teknik *encoding* yang telah dikembangkan. *ASCII* menerapkan standar *7-bit* per karakter, namun hampir sebagian besar komputer saat ini menyimpan data biner dalam memori – dalam bentuk *bytes* yang terdiri dari 8 bit sehingga *ASCII* sangat tidak cocok jika digunakan untuk melakukan transmisi data.

Untuk mengatasi permasalahan ini *Base64 Encoding* diperkenalkan dengan cara melakukan *encoding* serangkaian *bytes* kedalam *bytes* yang lebih aman untuk ditransmisikan tanpa mengalami *corruption*.

Base64 Encoding

Base64 adalah salah satu *encoding* yang sangat ***reliable*** untuk transmisi data. *Base64* seringkali disebut sebagai *binary to text encoding*. Sebelum memahami apa itu *base64*, tentu kita harus memahami apa itu *encoding* pada konteks ini?

Encoding

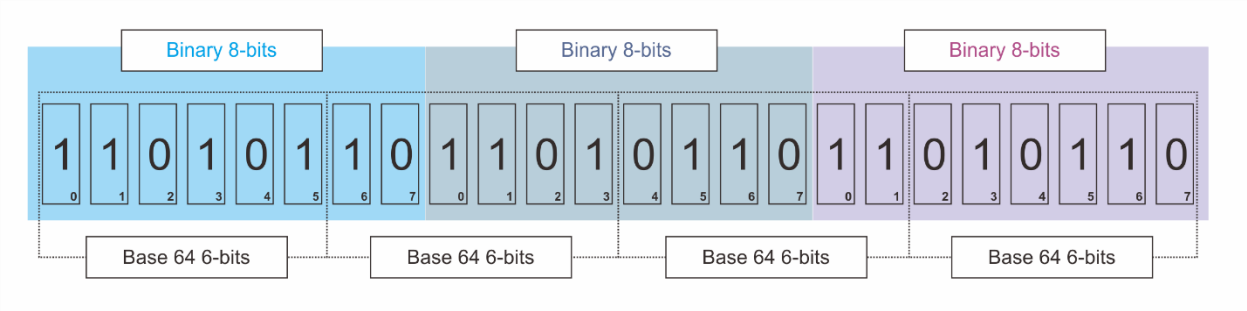
Encoding adalah proses untuk mengubah suatu data menjadi sebuah format yang lebih efisien untuk disimpan atau ditransmisikan.

Decoding

Decoding adalah format yang telah di *encoding* akan dikembalikan lagi kedalam bentuk data original.

Base64 di desain untuk membawa data dalam format biner pada seluruh *channel* yang mengandalkan konten berbasis teks seperti dalam *World Wide Web* (*WWW*). Kita dapat menggunakanya untuk menyisipkan gambar dan data biner lainya di dalam *textual assets* seperti *HTML* dan *CSS*.

Base64 membagi sebuah *input* ke dalam bentuk ***6-bit chunks***. Diberi nama *base64* karena hasil dari $2^6 = 64$, dan setiap *6-bit input* akan diubah kedalam *ASCII characters*.

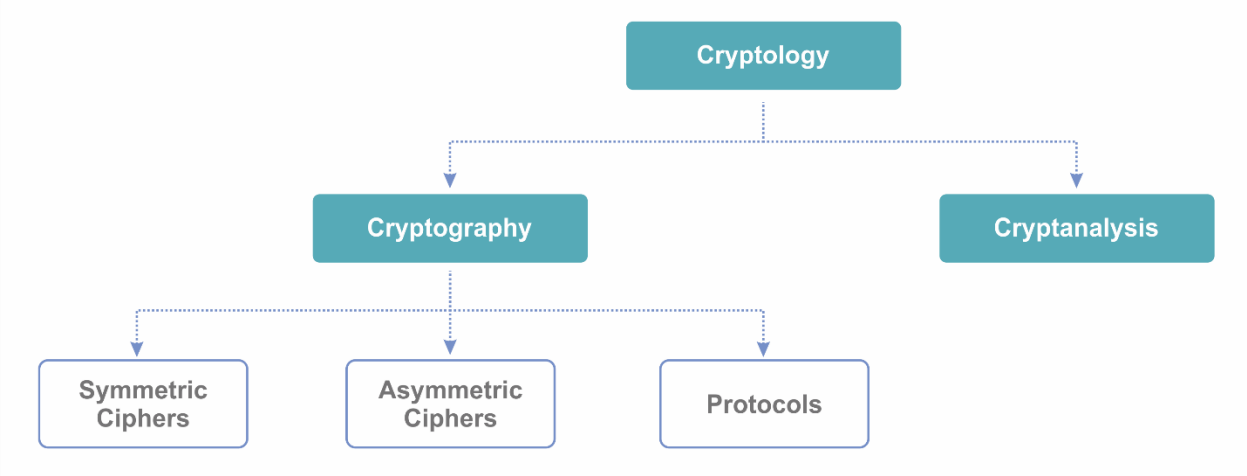


Gambar 52 Base64 Representation

2. *Cryptography*

Sebelumnya kita telah belajar apa itu *TCP Threeway Handshake*, Apa itu *Socket & Packet*?

Sekarang kita akan membahas kajian ilmu *Cryptography* yang digunakan untuk melindungi komunikasi data dalam *SSL*. Kita akan membahas dunia *cryptography* sebagai suplemen untuk memahami dunia keamanan dalam komputer.



Gambar 53 Cryptology Abstraction

Dalam *Concise Oxford English Dictionary* terminologi ***Cryptography*** dijelaskan sebagai **seni untuk menulis dan memecahkan sandi (*codes*)**. Jika dilihat secara historis penjelasan ini sangat akurat untuk ***Classical Cryptography***, namun hari ini *cryptography* sudah berkembang menjadi sesuatu yang sangat luas. Penjelasan tersebut lebih memfokuskan pada istilah *codes* yang telah digunakan selama beberapa abad untuk membuat komunikasi rahasia.

Hari ini *cryptography* lebih dari sekedar seni, ***Modern Cryptography*** sudah menggunakan pendekatan ilmu matematika untuk melindungi informasi digital, sistem dan komputasi terdistribusi dari serangan pihak ketiga.

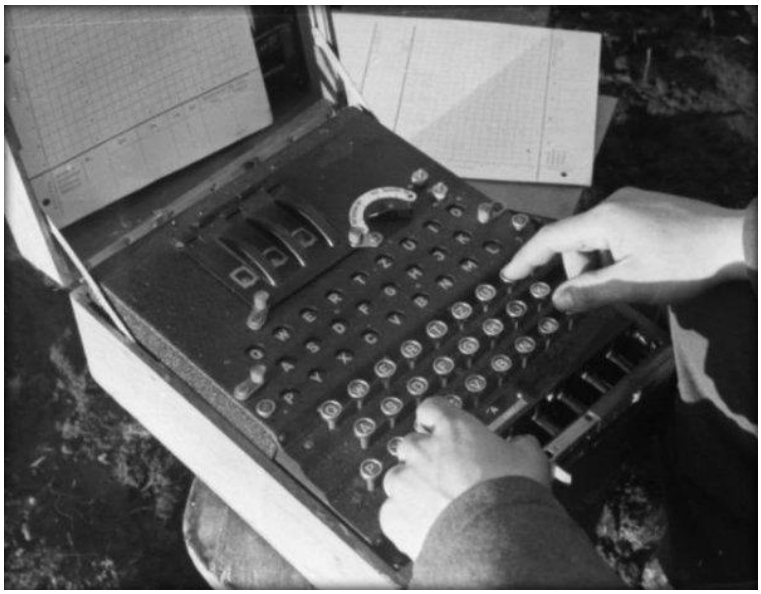
Perbedaan penting yang paling menonjol antara classical *cryptography* dan modern *cryptography* **adalah pada adopsinya**. Sebelum tahun 1980 classical *cryptography*

digunakan oleh militer dan pemerintah, pada modern *cryptography* hampir disemua lini tentang keamanan sistem kita pasti menggunakanya.

Cryptanalysis

Menurut Professor Eli Biham, ***Cryptography*** adalah ilmu untuk membuat pesan rahasia. *Cryptography* adalah salah satu cabang ilmu *cryptology* dalam sains komputer. ***Cryptology*** adalah study ilmiah untuk melindungi informasi dan membongkar informasi rahasia. Di dalam *cryptology* terdapat cabang ilmu ***Cryptoanalysis***, sebuah ilmu atau analisis untuk membongkar pesan rahasia.

Alan Turing adalah seorang *mathematician* & ***Cryptoanalyst*** yang sukses memecahkan mesin *enigma* yang dibuat oleh pasukan jerman untuk melindungi informasi (***Information Security***) saat melakukan komunikasi dalam perang dunia kedua (*World War II*).



Gambar 54 Enigma Machine[21]

Information Security

Cryptography digunakan untuk melakukan *infosec* atau *information security*, sebuah praktek untuk melindungi informasi dengan cara mengurangi resiko. Pencegahan terdapat akses & penggunaan yang tidak diizinkan, mencegah *disclosure, disruption, deletion/destruction, corruption, modification, inspection, recording* dan *devaluation*.

Ketika suatu *cryptography* digunakan dengan benar maka syarat-syarat di bawah ini mampu dipenuhi :

Confidentiality

Memastikan informasi rahasia hanya sampai pada orang yang berhak dan tidak sampai pada orang yang tidak berhak.

Availability

Memastikan informasi dapat diakses ketika dibutuhkan.

Integrity

Memastikan bahwa informasi tetap konsisten, akurat dan terpercaya. Saat data dikirim, data tetap sama tanpa bisa dimodifikasi oleh orang-orang yang tidak berhak.

Ciphertext

Terminologi ***Encryption*** digunakan ketika kita mengimplementasikan ilmu *cryptography* untuk mengubah suatu informasi menjadi sesuatu yang tidak bisa dibaca (***chipertext***), sehingga nilai ***Confidentiality*** dari informasi tersebut terjaga. Informasi yang telah di enkripsi disebut dengan ***Encrypted Information***.

Plaintext

Dalam ilmu *cryptography*, terminologi ***plaintext*** mengacu pada pesan yang belum dilindungi (*unencrypted message*).

Cipher

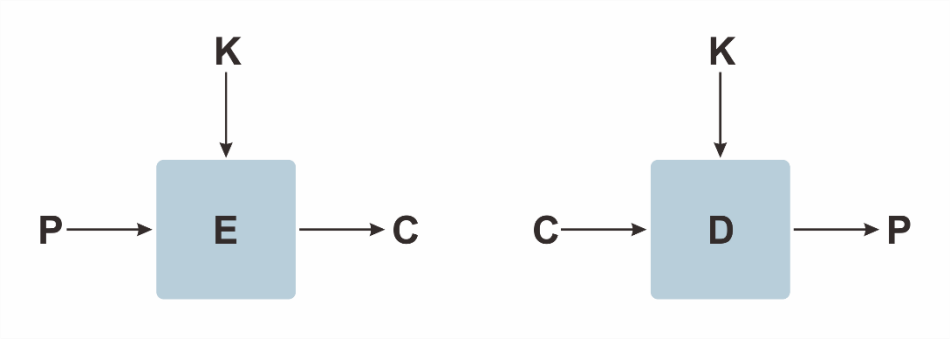
Setiap *encryption* menggunakan suatu algoritma yang disebut dengan ***Cipher*** dan memiliki nilai rahasia yang disebut dengan ***secret key***. Jika kita tidak memiliki *secret key* maka kita tidak akan bisa melakukan proses ***Decryption*** untuk membaca sebuah *encrypted information*.

Encryption & Decryption

Sebuah *cipher* memiliki dua fungsi yaitu *encryption* untuk mengubah *plaintext* kedalam *ciphertext* dan *decryption* untuk mengubah *ciphertext* kedalam *plaintext*.

Secret Key

Pada gambar di bawah ini E merepresentasikan sebuah *box* yang membutuhkan parameter *plaintext* (P) dan *secret key* (K) untuk melakukan proses *encryption* agar bisa memproduksi *ciphertext* (C). Notasinya dapat disederhanakan menjadi C = E(K, P). Untuk *decryption* notasinya dapat disederhanakan menjadi P= D(K, C).



Gambar 55 Encryption & Decrytion

Keyspace

Algoritma *encryption* yang baik mampu memproduksi *ciphertext* yang sangat sulit dianalisa oleh seorang *cryptanalyst*. Jika kita memiliki *cipher* yang bagus, maka opsi yang dimiliki oleh *cryptanalyst* atau *attacker* adalah mencoba seluruh kemungkinan *decryption key*. Cara ini dikenal dengan sebutan *exhaustive key search*. Cenderung membutuhkan waktu yang lama atau tidak diketahui sama sekali dan membutuhkan *resources* untuk melakukan komputasi yang sangat besar.

Secara *scientific* keamanan sebuah *ciphertext* tergantung dari *secret-key* atau *private-key* itu sendiri. Jika *secret-key* dipilih menggunakan ***keyspace*** yang sangat besar maka untuk memecahkan *encryption* diperlukan iterasi untuk memecahkan seluruh kemungkinan kunci yang sangat besar jumlahnya. Sehingga kita dapat menyebut *cipher* tersebut *computationally secure*.

Keyspace adalah ***set*** **seluruh kemungkinan** kunci (*key*) yang bisa digunakan oleh *cipher*. Set adalah *branch* ilmu matematika dalam statistika, *set* dalam bahasa indonesia disebut dengan himpunan. Saya mencoba menyederhanakan bahasa dengan harapan tanpa kehilangan substansinya, kata seluruh kemungkinan kunci secara teknis bisa disebut permutasi dari kunci.

Sebagai contoh jika terdapat *key* yang memiliki panjang 8 *bit (key length)*, maka *keyspace* yang dihasilkan sebesar 2^8 yaitu 256 kemungkinan kunci. *Key Length* adalah salah satu alat ukur untuk menentukan kekuatan suatu *encryption*.

Secara teknis *cipher* adalah sebuah *function* yang memerlukan *key* sebagai *parameter* dan *input plaintext* sebagai *parameter*, *function* tersebut akan memproduksi *ciphertext*.

Contoh lainya, pada 56 *bit key* terdapat *keyspace* yang memiliki kemungkinan 2^{56} *keys* dan jika anda memiliki komputer yang dapat membuat dan menguji *keys* dengan

kecepatan 1 milyar perdetik. Maka untuk memecahkan 56 *bit keyspace* dengan kecepatan 1 milyar perdetik kita memerlukan waktu *approximately* sekitar 2 tahun.

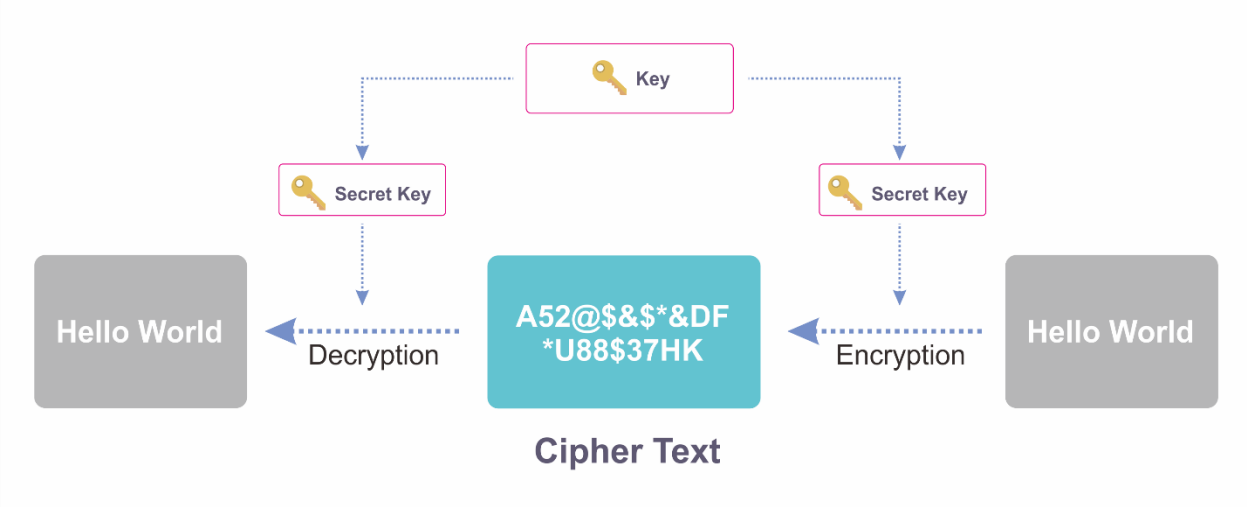
Bagaimana cara menghitungnya?

1 milyar sendiri adalah 2^{30} sehingga kita memerlukan 2^{26} detik untuk mencoba seluruh kemungkinan kunci dalam 56 *bit*. 2^{26} detik jika diubah ke dalam tahun maka kurang lebih sekitar 2 tahun.

Symmetric Cryptography

Pada *symmetric cryptographic scheme* hanya terdapat satu kunci yang sering disebut dengan *private key* atau *secret key* atau dalam beberapa literasi disebut dengan *shared secret-key*. ***Symmetric Encryption*** membutuhkan algoritma yang sama dan ***shared secret-key*** yang digunakan untuk enkripsi *plaintext* ke dalam *ciphertext* dan *decrypt ciphertext* ke dalam *plaintext*[22].

Contoh terdapat dua orang bernama Maudy Ayunda dan Gun Gun Febrianza yang berkomunikasi di dalam *insecure channel*, sebagai pengirim Maudy Ayunda harus memiliki *secret key* untuk melakukan *encryption* pada *plaintext* sebelum dikirimkan kepada Gun Gun Febrianza. Penerima pesanya Gun Gun Febrianza juga harus memiliki *secret-key* tersebut agar dapat membaca isi pesanya dengan cara melakukan *decryption*.



Gambar 56 Symmetric Encryption

Channel dan *insecure channel* yang dimaksud disini adalah *communication link*. *Symmetric cryptography* terdiri dari dua klasifikasi yaitu *block cipher* dan *stream cipher*.

Block Cipher

Block Cipher atau ***Block Encryption Algorithm*** akan membagi sebuah *plaintext* menjadi *block(s)* kecil dengan frequensi ukuran 64 *bits* atau 128 *bits* (16 *bytes*), kemudian masing-masing *block* akan di enkripsi menjadi *ciphertext block(s)*. Saat proses *decryption* dilakukan masing-masing *ciphertext block* akan didekripsi ke dalam *plaintext block(s)* untuk di susun ulang.

Kekurangan dari *block cipher* adalah setiap kali kita melakukan *encryption* pada suatu data maka *data length* tersebut harus sesuai dengan ukuran *encryption block*.

Agar *block cipher* bisa menerima data dengan *arbitrary length* dan data yang ukuranya tidak melebihi *block size* diperlukan mekanisme khusus agar bisa digunakanya untuk praktis.

Block Cipher juga memiliki sifat yang ***deterministic*** yaitu akan terus memproduksi *output* yang sama jika *input* adalah data yang sama.

Beberapa algoritma dalam *block cipher* di antaranya adalah *Advanced Encryption Standard (AES), Data Encryption Standard (DES), & Tripple Data Encryption Standard (3DES)*.

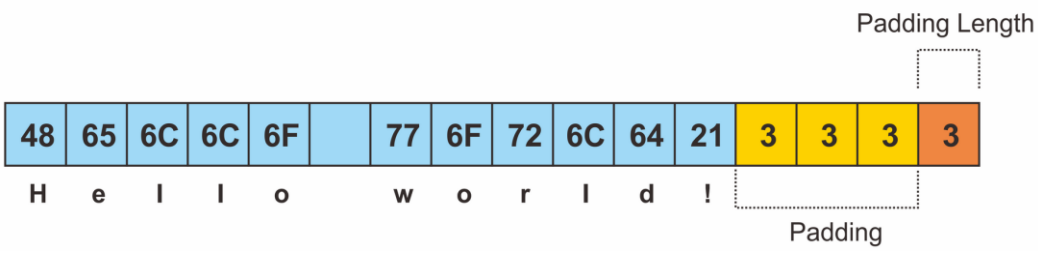
Padding

Jika terdapat data yang akan kita *encryption* namun ukuran *data length* tersebut lebih kecil dari ukuran *encryption block*. Sebagai contoh jika kita memiliki *symmetric encryption AES* dengan *key length* 128 *bit* (16 *bytes*) maka kita memerlukan *input data* dengan ukuran 16 *bytes* dan pada *output* akan memproduksi data dengan ukuran yang sama.

Tapi bagaimana jika *input data* yang diberikan lebih kecil dari 16 *bytes*?

Untuk mengatasi permasalahan ini terdapat solusi yaitu dengan menambahkan data tambahan yang disebut dengan ***padding***. Penerima data dapat mengetahui *padding* melalui format tertentu dan mengetahui seberapa banyak *bytes padding* yang harus dihapus.

Pada *SSL/TLS byte* terakhir dari sebuah *encryption block* adalah informasi panjang *padding*. Seluruh *padding bytes* dibuat dengan ukuran yang sama sesuai dengan *padding length byte*.

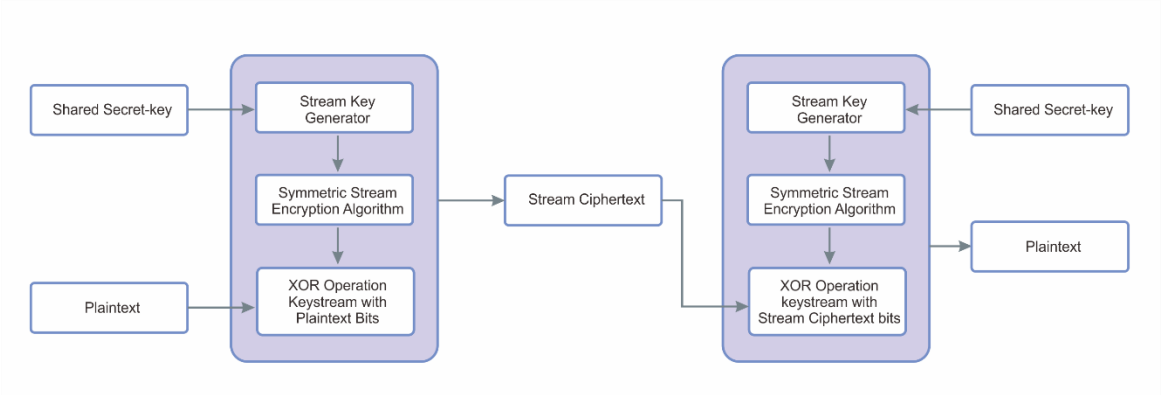


Gambar 57 Padding dalam SSL/TLS

Penerima data akan membuang *padding* dengan cara membaca *bytes* terakhir.

Stream Cipher

Pada ***Stream Cipher*** atau ***Stream Encryption Algorithm*****,** *input plaintext* tidak akan dibagi menjadi kumpulan *block(s)*. *Input plaintext* akan menerima *XOR operation* dengan *stream of bit* yang diproduksi dari *shared secret-key* untuk mengkonversi *plaintext* ke dalam *ciphertext*. Saat *decryption* pada *stream of ciphertext bits* dilakukan, *XOR operation* dilakukan menggunakan *stream of bit* yang diproduksi dari *shared secret-key*.



Gambar 58 Stream Cipher

RC4 adalah salah satu *stream encryption algorithm* yang paling banyak digunakan untuk *Secure Socket Layer (SSL), Transport Security Layer (TSL), Datagram TLS (DTLS)* dan *Wired Equivalent Privacy (WEP)*.

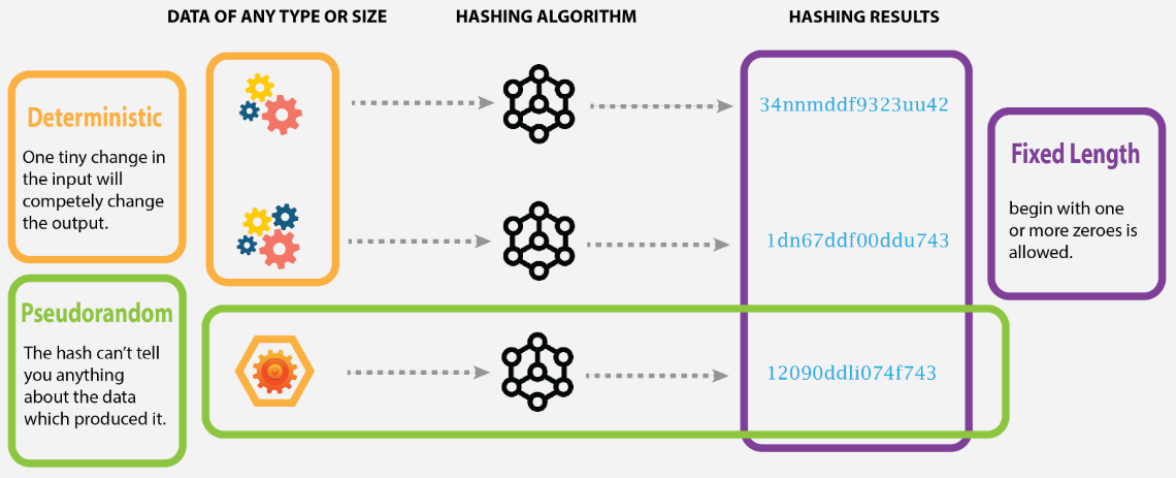
Stream Cipher memiliki keunggulan dalam hal kecepatan sehingga sangat cocok untuk melakukan komputasi pada *resources* yang terbatas, seperti dalam komunikasi *cellphones*.

Beberapa *symmetric algorithm* seperti *AES* dan *3DES* sangat aman, cepat dan sudah digunakan oleh banyak orang untuk berbagai aplikasi. *Symmetric algorithm* memiliki kelemahan yaitu :

Key Distribution Problem

Secret key harus di transmisikan melalui *secure channel*. Jika di transmisikan melalui *insecure channel* maka *secret key* dapat diketahui oleh sorang *sniffer* dalam jaringan.

Hash Function



Gambar 59 Illustrasi Hash Function

Hash Function adalah sebuah algoritma yang dapat mengubah *input* dengan *arbitrary length* tertentu ke dalam *output* yang memiliki ukuran *fixed* (misal 128 *bit*). ***Hashing*** adalah proses yang terjadi saat *hash function* mengolah *input* untuk memproduksi *message digest* atau *hash value*.

Hash Function memiliki beberapa *properties* di antaranya adalah ***Determinism*** yaitu *input* yang sama akan selalu menghasilkan *output* yang sama. *Output* dari sebuah *hash* memiliki karakteristik ***Pseudorandom***, pesan aslinya hampir mustahil untuk diketahui.

Hash Value

Output dari *hash function* disebut dengan *hash value* sering juga disebut dengan *message digest*. *Hash value* dapat digunakan untuk memverifikasi *integrity* suatu data

sehingga ***Message Digest*** juga adalah salah satu bentuk dari *Message Authentication Code (MAC)*[23].

Collision Resistance

Secara komputasi *hash function* tidak boleh menghasilkan *output hash value* yang sama dari dua *input* yang berbeda.

Message Authentication Codes (MAC)

Salah satu ancaman atau ***threat*** terbesar setelah penemuan *encryption* dalam komunikasi data adalah tidak adanya *message authentication*. Dalam hal ini seorang penerima pesan tidak dapat memastikan dari mana dan siapa pemilik pesan dibalik pesan yang diterimanya. Untuk mengatasi permasalahan ini *Message Authentication Code (MAC)* di ciptakan.

Message Authentication Code (MAC) dibuat untuk mencegah seorang *attacker* mencoba mengubah pesan yang dikirimkan oleh seorang entitas untuk entitas lainya tanpa terdeteksi.

Jadi apa itu *MAC*? *Message Authentication Code (MAC)* adalah sebuah *property* atau sandi singkat pada suatu data yang dapat digunakan untuk memeriksa keaslian sebuah pesan. Penerima dapat mengetahui jika pesan telah diubah oleh seseorang.

Message Authentication Code (*MAC*) dapat dibuat menggunakan *cryptographic hash* atau *Symmetric Encryption Algorithm*.

Cryptographic Hash

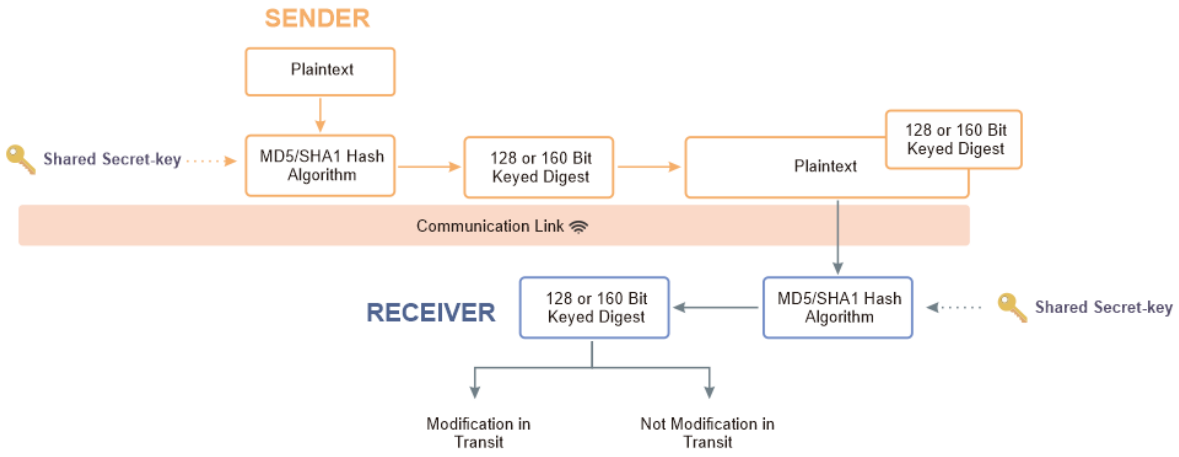
Jika kita ingin menggunakan *MAC-based communication* menggunakan *cryptographic hash*, pengirim (*sender*) akan membuat *plaintext* dan mengeksekusi sebuah *hash*

function dengan *parameter shared secret-key* untuk memproduksi *message digest* atau *keyed digest*.

Terdapat beberapa *hash function* yang dapat digunakan untuk memproduksi *message digest* atau *keyed digest*, diantaranya adalah *MD5* atau *SHA1 algorithm*.

Selanjutnya *Keyed Digest* disisipkan ke dalam *plaintext*, dikirimkan melalui *communication link* kepada seorang *receiver*.

Seorang penerima (*receiver*) akan memeriksa kembali *keyed digest* yang diterimanya, dengan cara membandingkan kembali *keyed-digest shared secret-key* dari pengirim dan *keyed-digest* dari *secret-key* yang dimilikinya.



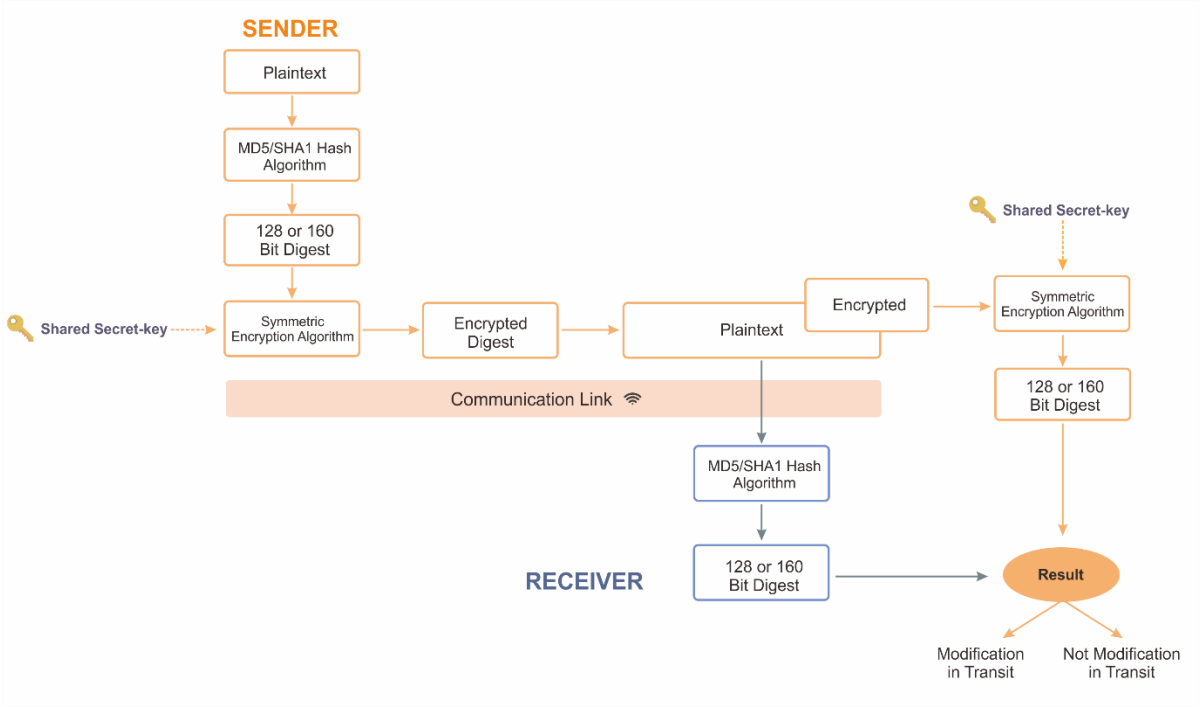
Gambar 60 MAC Menggunakan Cryptographic Hash

Symmetric Encryption Algorithm

Jika kita ingin menggunakan *MAC-based communication* menggunakan *symmetric encryption algorithm*, *sender* menerapkan *cryptographic hash algorithm* pada *plaintext* untuk memproduksi *message digest* sederhana tanpa kunci (*key*) dan melakukan *encryption* pada *message digest* menggunakan *symmetric encryption algorithm* dan *shared secret-key*. Selanjutnya pengirim (*sender*) melakukan transimisi *plaintext message* dan *encrypted message digest*.

Saat penerima (*receiver*) mendapatkan *plaintext message*, komputasi *hash function* dilakukan untuk mendapatkan *message digest*. Selanjutnya penerima (*receiver*) menggunakan *shared secret-key* untuk melakukan *decryption* pada *encrypted message digest*.

Hasilnya akan dibandingkan dengan *digest* yang telah di *decrypt*, jika *digest* yang telah di *decrypt* memiliki nilai yang sama dengan *digest* yang telah dikalkulasikan maka penerima (*receiver*) mengetahui bahwa pengirim memiliki salinan *shared secret-key* yang sama.



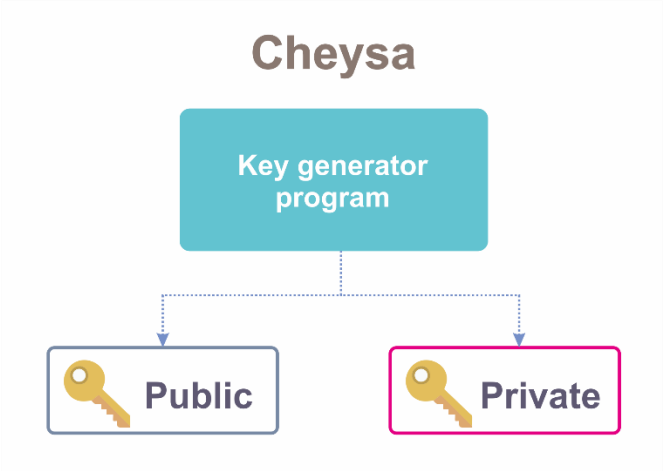
Gambar 61 MAC menggunakan Symmetric Encryption

Kedua pendekatan baik dari *Cryptographic Hash* atau *Symmetric Encryption Algorithm* juga menyediakan ***data-origin authentication*** untuk mengetahui siapa pengirimnya dan *data integrity*. Keduanya dapat digunakan untuk melakukan message authentication dengan syarat *shared secret-key* hanya boleh dimiliki oleh mereka yang berwenang. Jika *shared secret-key* kuncinya telah diketahui oleh seseorang, maka

sudah dapat dikatakan bahwa *message authentication* telah berhasil di *compromise* dan tidak lagi valid.

Assymetric Cryptography

Assymetric Cryptography atau sering disebut ***Public Key Cryptography*** menggunakan formula matematika agar bisa melakukan enkripsi (*encrypt*) dan dekripsi (*decrypt*) pada suatu data dengan menggunakan kunci pasangan yang sama. *Program* yang menggunakan *Assymetric Encryption* untuk memproduksi *Public Key & Private Key* disebut dengan ***Key Generation Program***.



Gambar 62 Key Generation Program

Setiap pasang kunci terdiri dari *Public Key & Private Key*.

Public Key

Public key adalah kunci yang bisa diberikan kepada seluruh pihak yang tertarik untuk berkomunikasi.

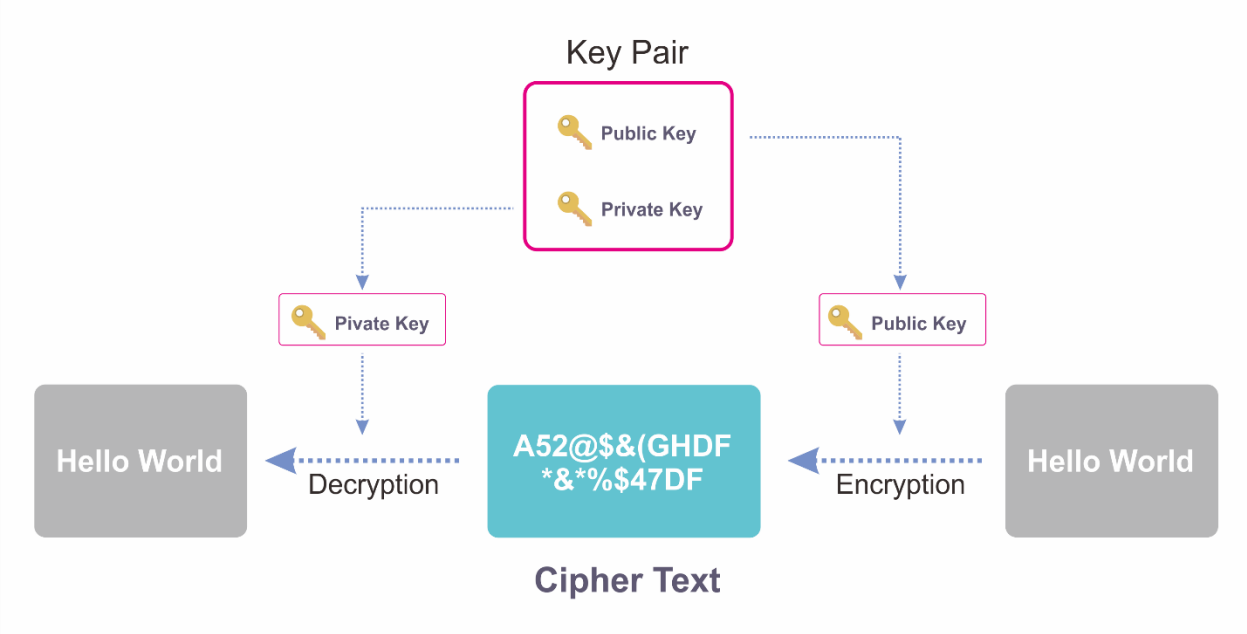
Private Key

Private Key adalah kunci rahasia yang harus dijaga rahasia oleh pembuatnya.

Secara teknis atau ***under the hood***, *Public Key & Private Key* adalah sebuah nilai matematis yang telah dibuat menggunakan algoritma matematika tertentu dan digunakan untuk melakukan *encrypt* dan *decrypt* pada data.

Pada *asymmetric cryptography*, data akan di *encrypt* menggunakan *public key* dan data hanya dapat di baca oleh entitas yang memiliki *private key*. *Encrypt* digunakan untuk melindungi data dengan cara mengubah pesan ke dalam bentuk yang tidak bisa dibaca oleh pihak yang tidak berwenang.

Perhatikan ilustrasi gambar di bawah ini :



Gambar 63 Assymetric Cryptography

Public-key cryptography pertama kali dipublikasikan oleh Whitfield Diffie, Martin Hellman dan Ralph Merkle pada tahun 1976.

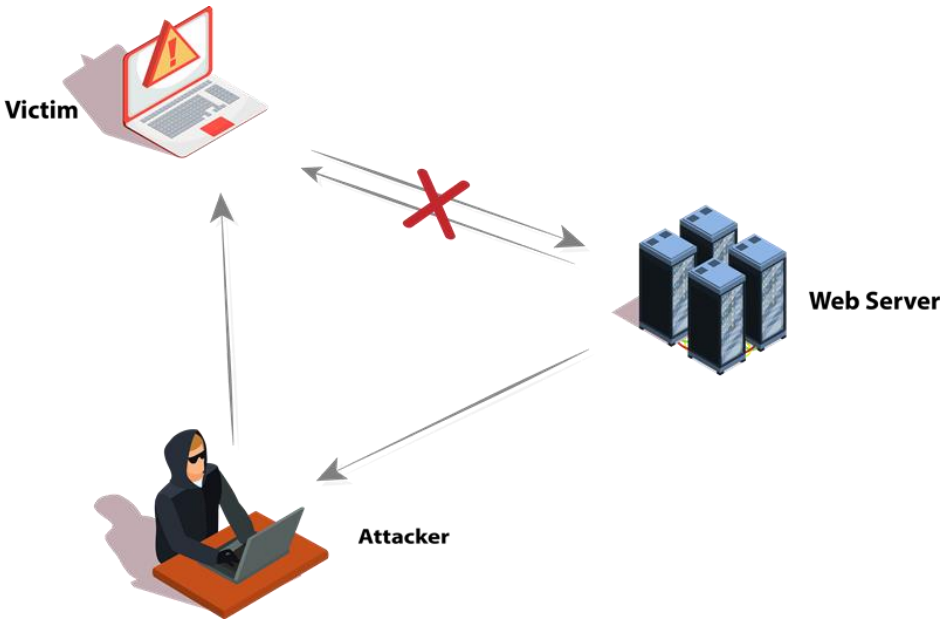
Cryptography Protocol

Cryptography Protocol adalah sebuah karya hasil dari implementasi *cryptographic algorithm*. Salah satu contoh *cryptographic protocol* adalah *SSL (Secure Socket Layer) / TLS*

(Transport Layer Security) yang dibangun menggunakan *symmetric* dan *asymmetric algorithm*.

3. *Man In The Middle (MITM) Attack*

Saat kita terhubung di dalam sebuah jaringan komputer yang tidak aman baik itu secara *wire* atau *wireless* melalui *wi-fi acess point*. Seseorang di dalam satu jaringan dapat melakukan *packet-sniffing* untuk mendapatkan informasi sensitif, juga terdapat serangan lainya seperti ***packet injection*** untuk menampilkan iklan yang bisa memiliki *malware (malicious software)* untuk mencuri data sensitif ke dalam halaman yang akan di dapatkan oleh pengguna.



Gambar 64 Ilustrasi MITM Attack

Salah satu varian serangan *MITM Attack* adalah ***Eavesdropping***.

Eavesdropping

Tindakan untuk mengetahui sebuah komunikasi privat tanpa diketahui disebut dengan *eavesdropping*. Dalam dunia keamanan komputer hal seperti ini dapat terjadi, misal dua entitas sedang berkomunikasi dalam satu jaringan komputer atau jaringan yang lebih besar seperti internet namun disadap oleh seorang *hacker*. Kedua entitas tersebut adalah

Gun Gun Febrianza dan Maudy Ayunda. Di bawah ini adalah sebuah skenario *MITM Attack* :

Jika kita menggunakan ***Assymetric Encryption*** untuk melindungi informasi, tentu Gun Gun Febrianza harus memproduksi terlebih dahulu ***Private Key*** & ***Public Key***. Setelah diproduksi, *public key* dikirimkan kepada Maudy Ayunda namun sang *hacker* melakukan ***interception*** untuk mencegah *public key* sampai kepada Maudy Ayunda.

Sebaliknya sang *hacker* mencuri *public key* dari Gun Gun Febrianza, sang *hacker* juga memproduksi *private key & public key* dan *public key* diberikan kepada Maudy Ayunda seolah-olah dikirimkan oleh Gun Gun Febrianza.

Selanjutnya dengan *public key* palsu dari sang *hacker*, Maudy mencoba melakukan proses enkripsi pesan yang akan dikirimkanya menggunakan *public key* milik sang *hacker*. Saat Maudy Ayunda mencoba mengirimkan pesan kepada Gun Gun Febrianza, maka pesan tersebut akan di *intercept* oleh sang *hacker*. Menggunakan *private key* yang dimilikinya sang *hacker* dapat dengan mudah membuka pesan yang dikirim oleh *Maudy*.

Sang *hacker* dapat membuat pesan palsu menggunakan *public key* milik Gun Gun Febrianza dan mengirimkan pesan palsu tersebut kepada Gun Gun Febrianza seolah-olah berasal dari Maudy Ayunda. Begitu juga ketika Maudy Ayunda mencoba memproduksi *Private Key & Public Key*, maka sang *hacker* akan mencurinya dan memanipulasi komunikasi secara keseluruhan.

Secara sederhana terdapat 4 macam serangan *MITM* :

Sniffing

Sang *hacker* mengetahui seluruh percakapan antara Gun Gun Febrianza & Maudy Ayunda.

Intercepting

Sang *hacker* menahan pesan milik Gun Gun Febrianza atau Maudy Ayunda.

Tampering

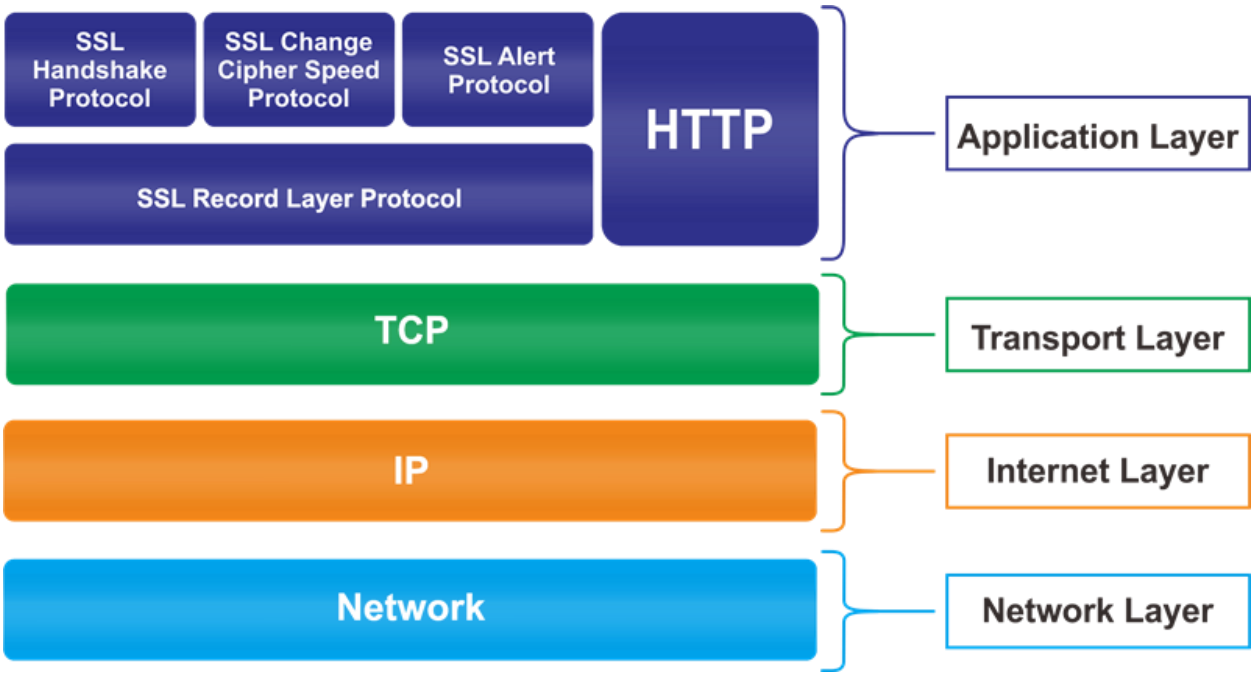
Sang *hacker* mengubah pesan yang dikirimkkan oleh Gun Gun Febrianza atau Maudy Ayunda.

Fabricating

Sang *hacker* menyamar menjadi Gun Gun Febrianza atau Maudy Ayunda untuk mengambil komunikasi dan identitas secara penuh.

4. *HTTPS*

HTTPS (Hyper Text Transfer Protocol Secure adalah salah satu *protocol* dalam *application layer* pada model jaringan komputer *TCP/IP*. *HTTPS* adalah pengembangan lebih lanjut dari *HTTP* (*Hyper Text Tranfer Protocol*), digunakan untuk **membangun komunikasi yang aman** antar jaringan komputer dan digunakan dalam internet.



Gambar 65 SSL Under The Hood

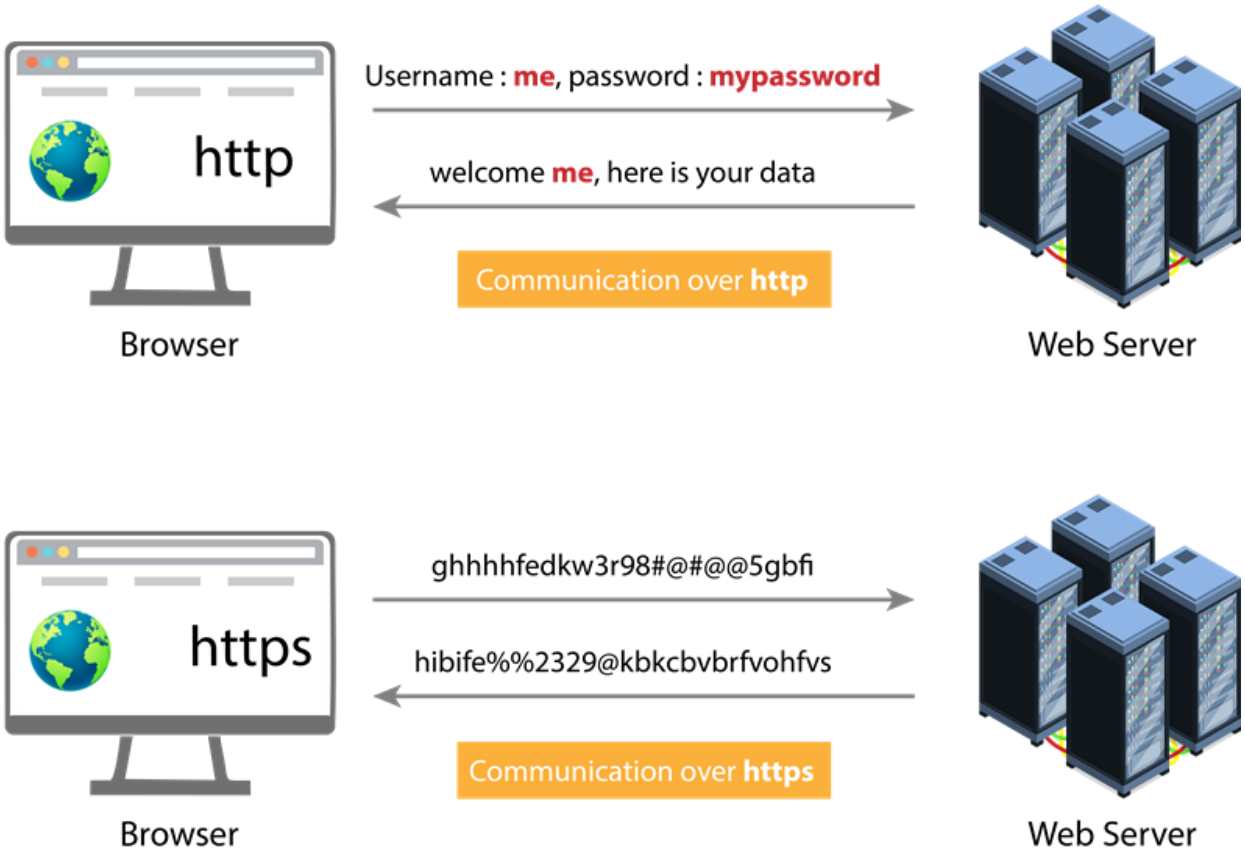
Data akan dilindungi dalam protokol *Transport Layer Security (TLS)* atau *Secure Socket Layer (SSL)*. *HTTP* seringkali juga disebut sebagai *HTTP* melalui jalur *TLS/SSL*. Pada gambar di bawah ini adalah perbedaan penggunaan protokol pada *HTTP* dan *HTTPS* :

Gambar 66 HTTP

Gambar 67 HTTPS

Perbedaan *HTTP & HTTPS*

Saat kita bertukar data melalui *HTTP*, data format teks yang dikirimkan dari *browser* menuju *server* dapat di baca dengan mudah. Pada *HTTPS* data telah dienkripsi untuk dilindungi sehingga tidak dapat dibaca dan dimodifikasi oleh seorang *hacker*.



Gambar 68 Perbedaan Komunikasi HTTP & HTTPS

Tabel Perbedaan *HTTP & HTTPS*

HTTP	HTTPS
Data berupa *Hypertext format*	Data berupa *Encrypted format*
Secara *default* menggunakan *port* 443	Secara *default* menggunakan *port* 443
Data tidak diamankan	Diamankan dengan *TLS/SSL*
Protocol yang digunkan http://	*Protocol* yang digunkan https://

Manfaat HTTPS

Terdapat manfaat besar jika kita menggunakan *HTTPS* (*Hyper Text Transfer Protocol*) di antaranya adalah :

Secure Communication

HTTPS melakukan *tunneling* dari Protokol *HTTP* melalui Protokol *TLS/SSL* yang akan melakukan enkripsi pada ***HTTP Payload*** (data). Setiap *HTTP Request* dan *HTTP Response* akan dikirimkan dengan aman sehingga para pelaku yang melakukan *sniffing* termasuk *Internet Service Provider (ISP)* tidak akan mengetahui apa yang kita lakukan. Komunikasi antara *browser* dengan *server* menjadi aman.

Data Integrity

HTTPS menyediakan integritas data dengan cara melakukan enkripsi pada data, sehingga jika terjadi aktivitas *Eavesdropping* maka data tidak akan bisa dibaca dan dimodifikasi.

Privacy & Security

HTTPS juga membantu memberikan keamanan privasi pada data yang digunakan untuk bertransaksi.

Faster Performance

HTTPS meningkatkan kecepatan data transfer jika dibandingkan dengan *HTTP* dengan cara mereduksi data yang dikirimkan.

SEO (Search Engine Optimization)

Berdasarkan algoritma mesin pencari yang terbaru termasuk *google* jika kita menggunakan *HTTPS* maka ini dapat meningkatkan *SEO ranking*.

New Standard

Teknologi *HTTP/2* Akan mendominasi dan menjadi standard, untuk menggunakanya diwajibkan *server* dan *browser* harus sudah mendukung *HTTPS*.

5. *Secure Socket Layer (SSL)*

Setelah membahas *cryptography* & *HTTPS* memahami kajian *SSL* bisa menjadi lebih mudah. Jadi apa itu *SSL*?

Secure Sockets Layer (SSL) adalah *protocol* yang digunakan sebagai standar untuk membuat koneksi internet aman. Seluruh data sensitif yang dikirimkan antar sistem dapat dilindungi, mencegah tindakan kriminal seperti membaca dan memodifikasi informasi yang hendak dikirimkan.

Semenjak pertama kali *Netscape* mengajukan proposal *SSL* pada tahun 1990, protokol *SSL* telah berevolusi menjadi tiga versi yaitu versi 1.0, 2.0, 3.0 hingga akhirnya digantikan oleh *TLS* (*Transport Layer Security*) Protokol[24].

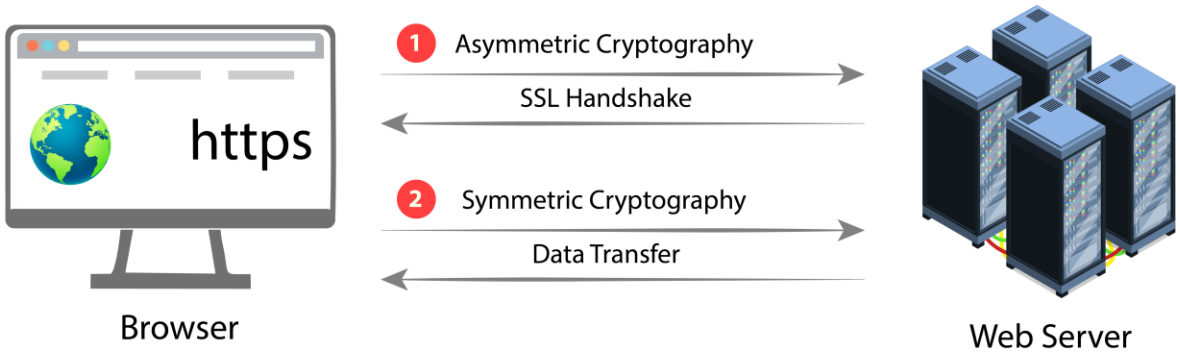
SSL/TLS protokol biasanya digunakan untuk *server* dan *client*, namun dapat juga digunakan untuk *server* ke *server* dan *client* ke *client* jika dibutuhkan. Data pribadi yang sensitif dan data mengenai keuangan dapat dilindungi dengan baik. Maka dari itu migrasi dari

Transport Socket Layer (TLS)

Transport Layer Security (TLS) adalah versi terbaru dan teraman dari SSL. Namun kenapa masih saja menggunakan terminologi SSL? Karena SSL memiliki sejarah pengembangan yang panjang dan masih menjadi terminologi yang paling banyak dikenali oleh orang-orang pada umumnya.

SSL Handshake

Untuk memberikan keamanan yang maksimal saat melakukan transfer data, protokol *SSL* menggunakan *asymmetric* dan *symmetric cryptography*. Perhatikan ilustrasi gambar di bawah ini dimana *asymmetric* dan *symmetric cryptography* digunakan :



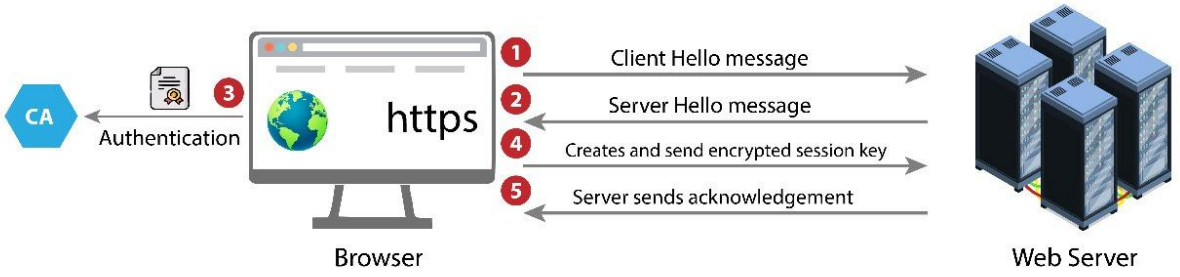
Gambar 69 SSL Communication

SSL Communication antara *browser* dan *web server* terdiri dari dua langkah yaitu :

1. *SSL Handshake*
2. *Data transfer*.

Transaksi data untuk berkomunikasi lewat protokol *SSL* diperlukan *SSL handshake* terlebih dahulu. Pada saat melakukan *SSL Handshake*, *asymmetric cryptography* digunakan agar *browser* dapat memverifikasi *Web Server*, mendapatkan *public key* dan membangun jembatan komunikasi yang aman dengan cara melakukan *encryption* setiap kali data dikirimkan.

Di bawah ini adalah beberapa langkah saat terjadi *SSL Handshake* :



Gambar 70 SSL Handshake in-depth

1. *Client* mengirimkan pesan "*client hello*", *SSL version number, cipher settings, session-specific data* dan informasi penting lainya yang dibutuhkan oleh *server* agar bisa berkomunikasi dengan *client*.
2. *Server* akan merespon pesan "*server hello*", *SSL version number, cipher settings, session-specific data,* sebuah *SSL certificate* dengan *public key* dan informasi lainya yang dibutuhkan oleh *client* agar bisa berkomunikasi dengan *server* melalui jalur *SSL*.
3. *Client* memverifikasi *server's SSL certificate* dari *CA* (***Certificate Authority***) dan melakukan *authentication* pada *server*. Jika *authentication* gagal maka *client* akan menolak untuk terhubung pada *SSL connection* dan mengeksekusi sebuah *exception* agar *browser* memperingatkan *client*. Jika *authentication* berhasil maka kita akan menuju ke langkah 4.
4. *Client* membuat *session key* yang akan di *encrypt* menggunakan *public key* milik *server* dan mengirimkanya kepada *server*. Jika *server* telah meminta *client authentication*, maka *client* akan mengirimkan *certificate* yang dimilikinya kepada *server*.
5. *Server* akan melakukan *decryption* pada *session key* menggunakan *private key* yang dimilikinya dan mengirimkan sebuah pengakuan (*acknowledgement*) kepada *client* berikut dengan *session key*.

Setelah *SSL Shakehand* dilakukan baik *client* dan *server* memiliki *session key* yang valid untuk melakukan *encryption* dan *decryption* data. Pada fase ini *Public key* dan *Private key* tidak akan lagi digunakan.

Chapter 2

Setup Learning Environment

Sebelum memulai belajar kita harus mengenal lingkungan belajar yang akan digunakan, ada beberapa *software* yang harus kita fahami agar proses belajar kita maksimal dan pengembangan *application* yang ingin kita buat bisa optimal.

Subchapter 1 – *Visual Studio Code*

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

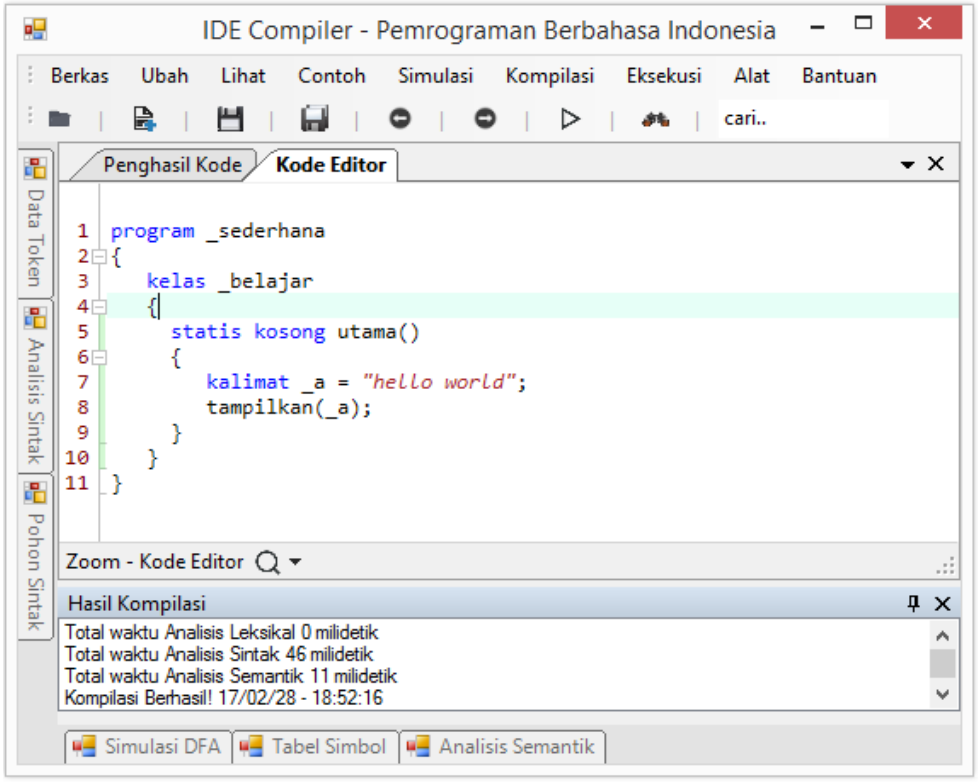
— Martin Fowler

Subchapter* 1 – *Objectives

- Mempelajari ***Install Programming Language***
- Mempelajari ***Install Keybinding***
- Mempelajari ***Install & Change Theme Editor***
- Mempelajari ***Install Extension***
- Mempelajari ***Terminal Visual Studio Code***
- Mempelajari ***Font Ligature***

Saya sebagai penulis buku ini pernah membuat IDE (*Integrated Development Environment*) yang di dalamnya terdapat *code editor*, skripsi penulis adalah pembangunan *compiler* dan pembuatan bahasa pemrograman berbahasa Indonesia.

Penulis masih ingat pembimbing memaksa untuk membangun *code editor* yang bisa mempermudah pengguna dalam menulis kode pemrograman berbahasa indonesia. Sungguh permintaan yang berat 😊 di bawah ini penampakanya :



Gambar 71 IDE untuk Pemograman berbahasa Indonesia

Jadi diri sini, dari pengalaman ini penulis begitu percaya diri membahas kajian seputar *code editor*.

Jika teman anda bertanya mengapa anda memilih suatu *code editor* tentu anda harus memiliki jawaban yang jelas dan menginspirasi. Di bawah ini adalah beberapa alasan mengapa kita memilih *code editor* :

1. Memiliki mekanisme untuk membuat performa dari *code editor* ringan.
2. Tersedia fitur ***Intellisense*** yang terdiri dari ***Syntax Highlighting*** & ***Autocomplete***.
3. Tersedia fitur untuk melakukan ***Debugging***.
4. Tersedia fitur untuk berinteraksi dengan ***Git***.
5. Tersedia fitur untuk melakukan ***liveshare***.

6. Tersedia fitur ***Add-ons*** untuk *code editor* yang dikembangkan oleh komunitas aktif dan pengembang *expert*.

Sebelumnya penulis menggunakan *atom*, kemudian bermigrasi ke *visual studio code*. Meskipun begitu *atom* terkadang digunakan untuk menguji proyek-proyek sederhana atau menguji kode *snippet*. *Visual studio code* menjadi *code editor* paling popular dalam **Stack Overflow Developer Survey pada** tahun 2018-2019.

Visual studio code adalah *code editor* yang dibangun menggunakan **node.js** di atas base **electron.js** agar bisa berjalan di dalam *dekstop environment*. Sederhananya, *Electron.js* adalah *framework* yang dapat membuat aplikasi *web* menjadi aplikasi *dekstop* agar menjadi *cross-platform application* yang berjalan di semua sistem operasi.

Jadi *visual studio code* sendiri dibangun menggunakan *javascript*. *Visual Studio Code* jika bersifat *open source*, anda bisa ikut mengembangkanya atau memodifikasinya untuk keperluan anda sendiri. Anda perlu memahami ***Typescript*** dan *CSS* jika ingin memodifikasi *source code* dari *visual studio code*.

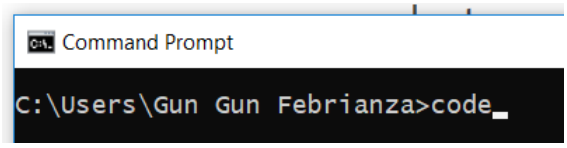
Link Project Visual Studio Code :

https://github.com/microsoft/vscode

Untuk mengetahui update fitur-fitur yang telah dikembangkan silahkan cek di :

https://code.visualstudio.com/updates/

Untuk membuka *viscode* anda dapat menggunakan *command prompt*, cukup beri instruksi code maka *editor* akan tampil :



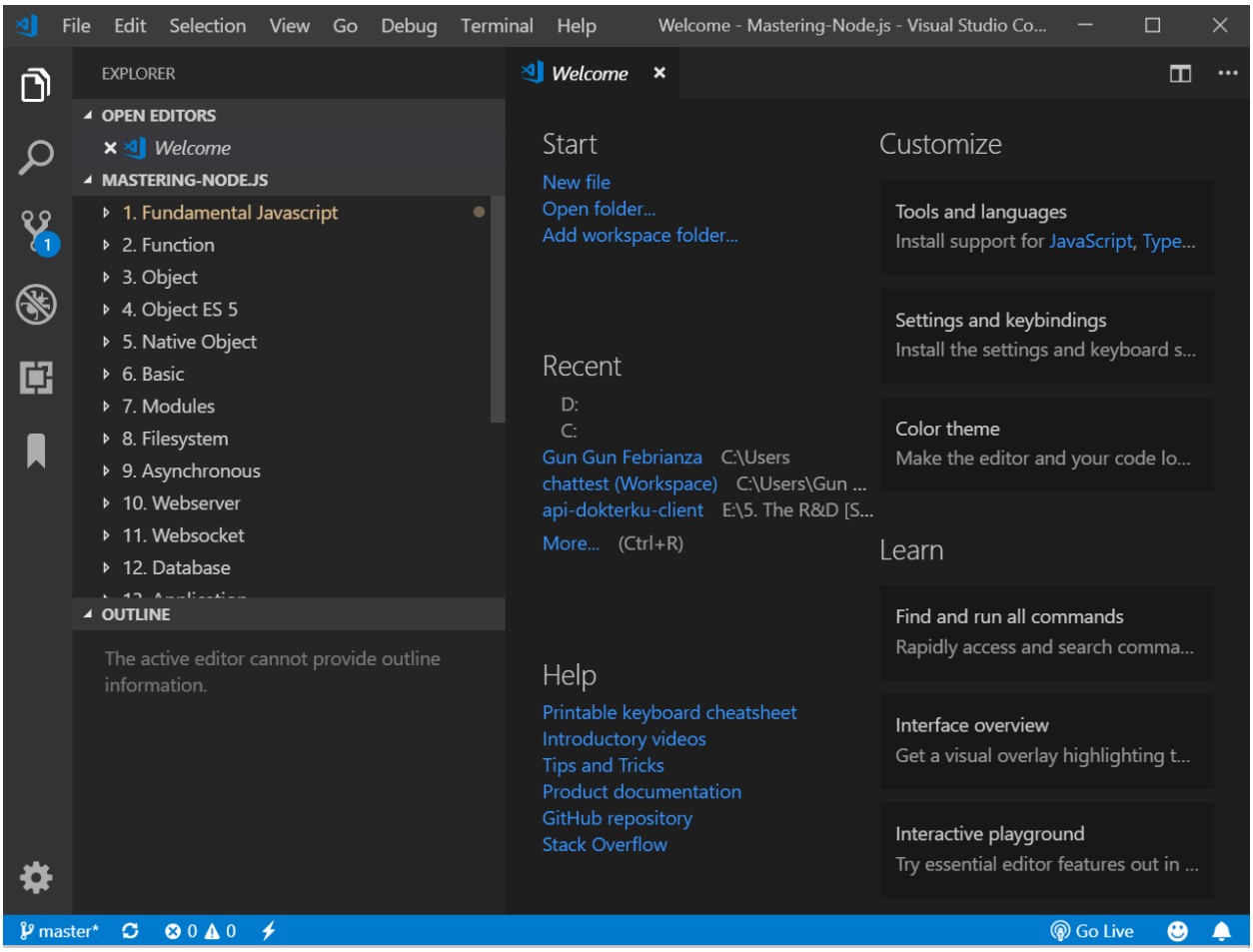
Gambar 72 Command Prompt

Anda juga bisa membuat *viscode* langsung membuka suatu *directory* :

```
C:\Users\Gun Gun Febrianza>code "E:\13. The Kaizer Arsenal - Back-end\Node.js - GITHUB\Mastering-Node.js"
```

Gambar 73 Command Prompt Part II

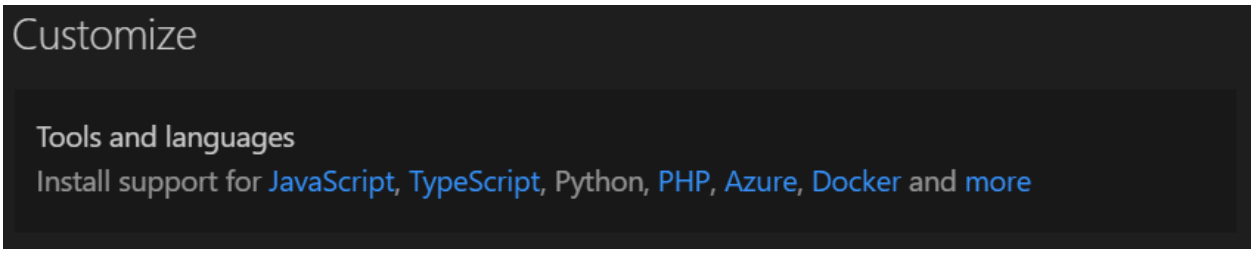
Di bawah ini adalah tampilan *user-interface* dari *viscode* :



Gambar 74 Visual Studio Code Interface

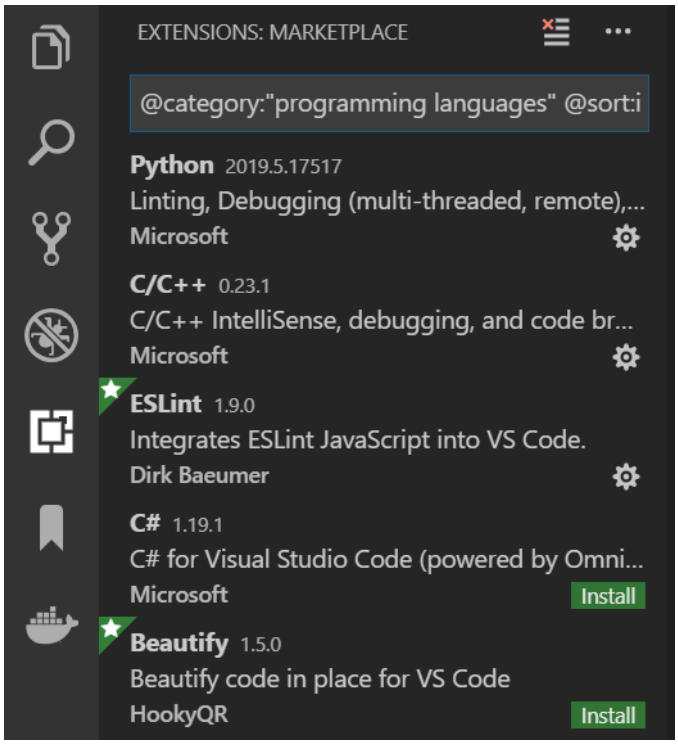
1. *Install Programming Language Support*

Pada menu *customize* anda bisa melihat kita bisa membuat *viscode* yang kita miliki mendukung bahasa pemrograman lainya seperti, *python, php* atau *script* untuk *azure* dan *docker*. Install *javascript* dan *typescript* dengan cara melakukan klik pada tulisan *javascript* dan *typescript*. Untuk mengetahui lebih banyak klik **Tools and languages**.



Gambar 75 Install Tools & language

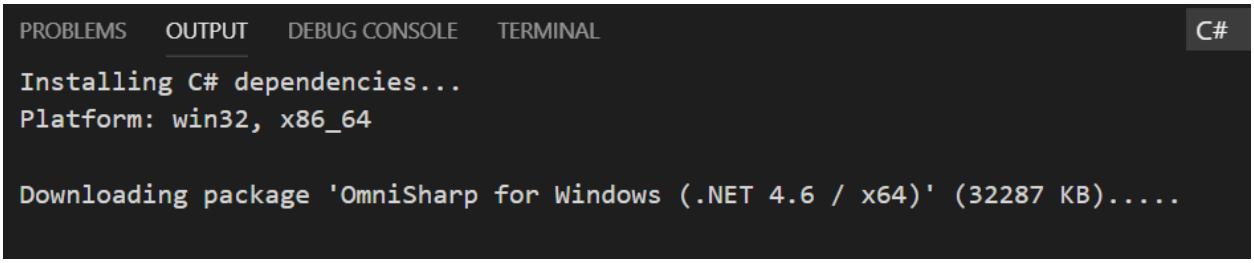
Pada menu sebelah kiri akan muncul kolom pencarian seperti gambar di bawah ini :



Gambar 76 Search Tools & Language

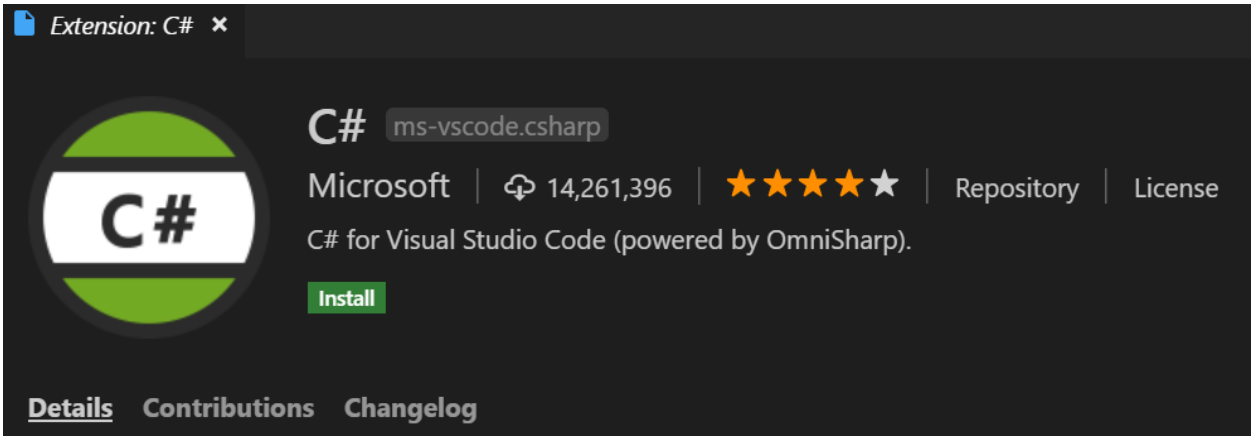
Lakukan *scrolling* ke bawah untuk mengetahui lebih banyak lagi, pada gambar di atas penulis telah melakukan instalasi bahasa *python, C++* dan *linter* untuk *EcmaScript*.

Kita akan mencoba melakukan instalasi bahasa C#, klik tombol berwarna hijau dengan label *install* pada bahasa C# :



Gambar 77 Instalasi C# Language

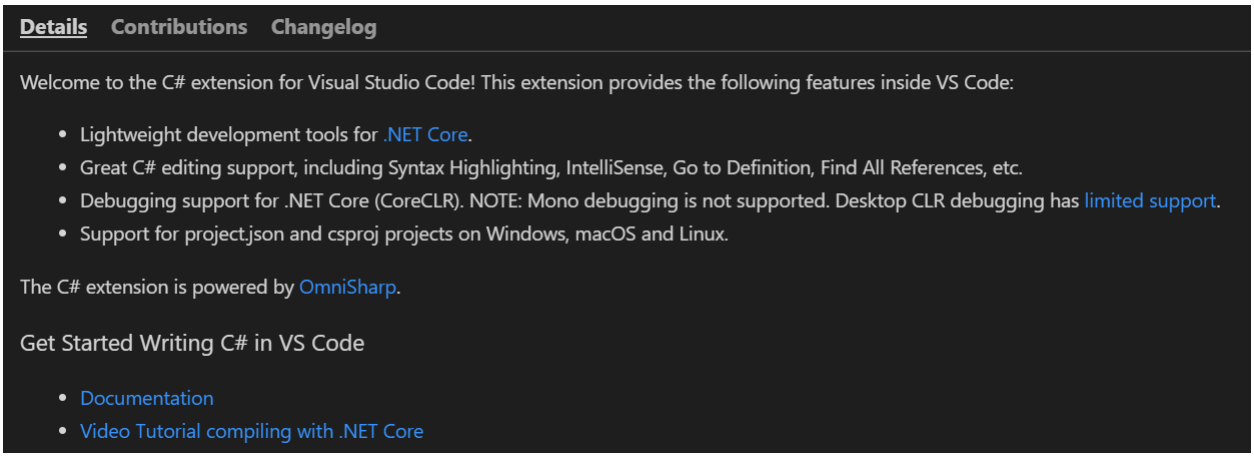
Pada kolom *output* kita akan melihat informasi *download package* yang kita butuhkan untuk dapat menggunakan bahasa pemrograman C# dalam *viscode*.



Gambar 78 C# Information

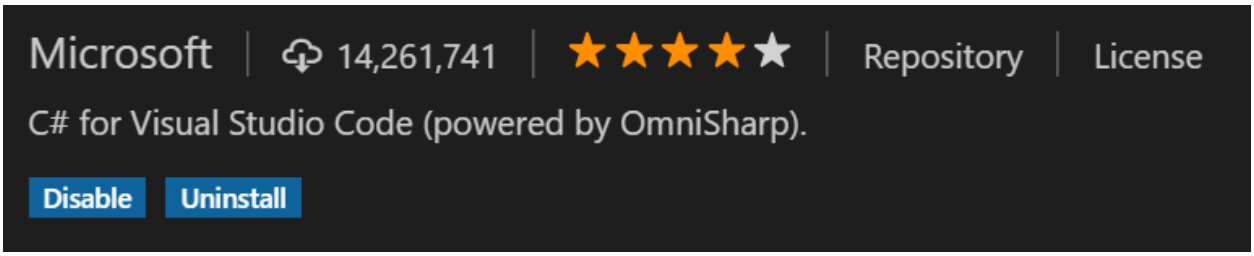
Maka akan muncul informasi mengenai bahasa pemograman yang akan kita *install*, seperti jumlah *download, rating, repository* dan *license*. Semakin besar jumlah *rating* dan pengguna yang melakukan instalasi maka dipastikan kualitasnya bagus.

Pada kolom *details* juga kita bisa melihat informasi lebih detail dari *package* yang hendak kita install. Seperti *documentation* cara penggunaanya dan *update* pengembangan terbaru.



Gambar 79 C# Detail Information

Jika instalasi berhasil maka akan muncul label dengan informasi *disable* dan *uninstall*.

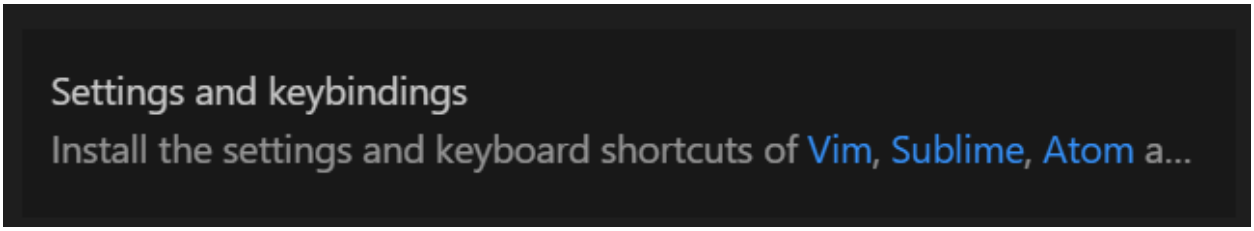


Gambar 80 Success Installed

Karena ini hanya sebagai pembelajaran semata agar anda mengetahui cara untuk melakukan instalasi bahasa pemrograman silahkan *uninstall* kembali.

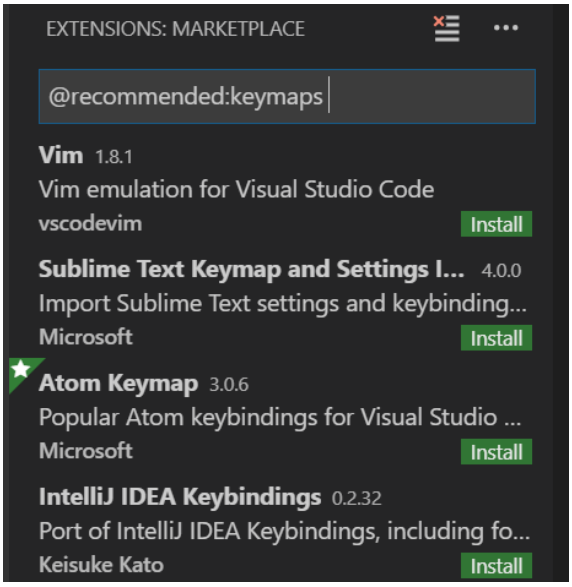
2. *Install Keybinding*

Keyboard Shortcut memiliki peran vital untuk penulisan kode, perubahan *keyboard shortcut* pada *code editor* baru tentu akan menyulitkan. Untuk mengatasi permasalahan ini pada menu *keybindings* kita bisa melakukan instalasi *keymap extension*,



Gambar 81 Key Bindings

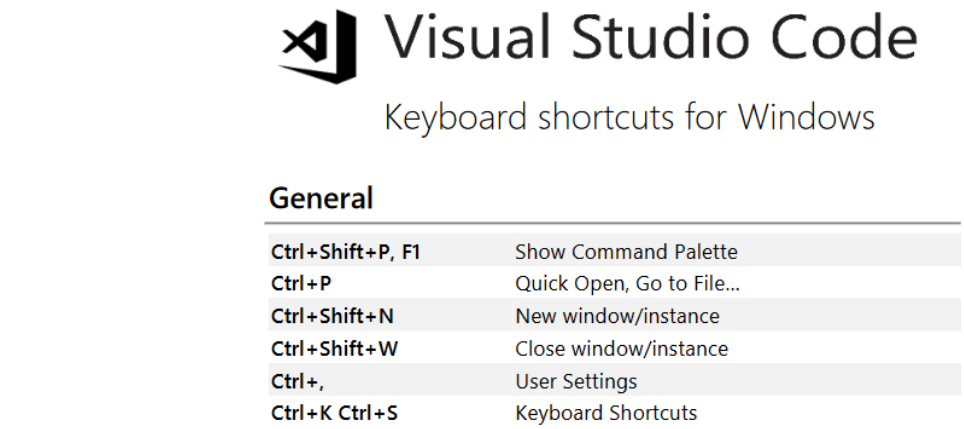
Terdapat beberapa *keymaps* yang bisa kita gunakan termasuk *Atom Keymap*.



Gambar 82 Keymap Extension

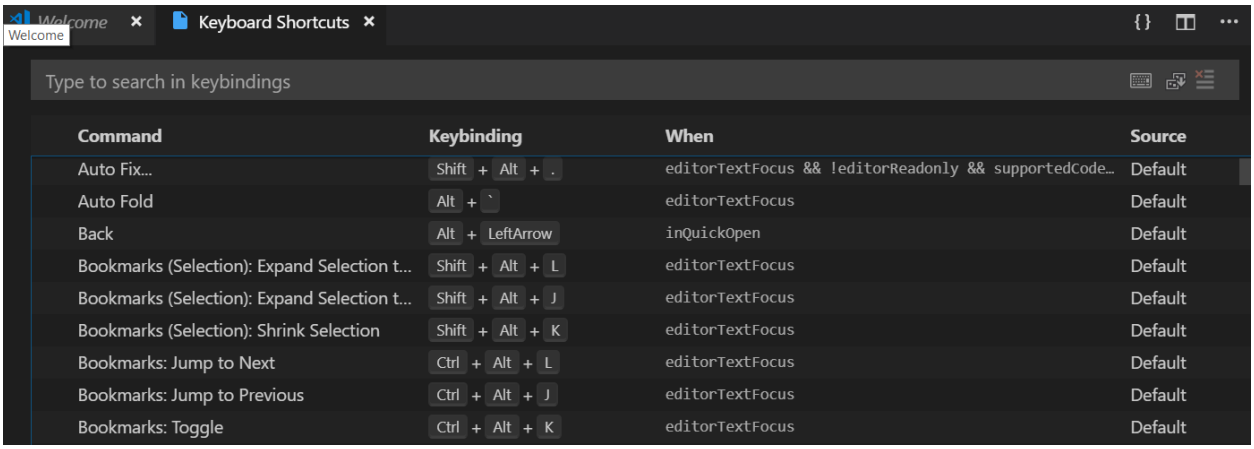
Saat ini penulis masih menggunakan ***Keyboard Shortcut Default*** bawaan dari *viscode*. Untuk ***Cheatsheet*** lengkapnya bisa di *print* dan tempel di tembok. Silahkan lihat disini :

https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf



Gambar 83 Visual Studio Code Keyboard Shortcut

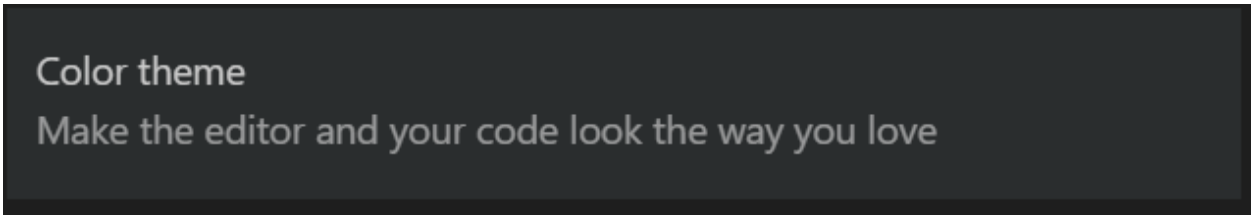
Jika anda ingin mengubah *keyboard shortcut* silahkan memilih menu ***File → Preferences → Keyboard Shortcuts*** atau tekan tombol ***CTRL+K*** kemudian ***CTRL+S***.



Gambar 84 Default Keyboard Shortcut

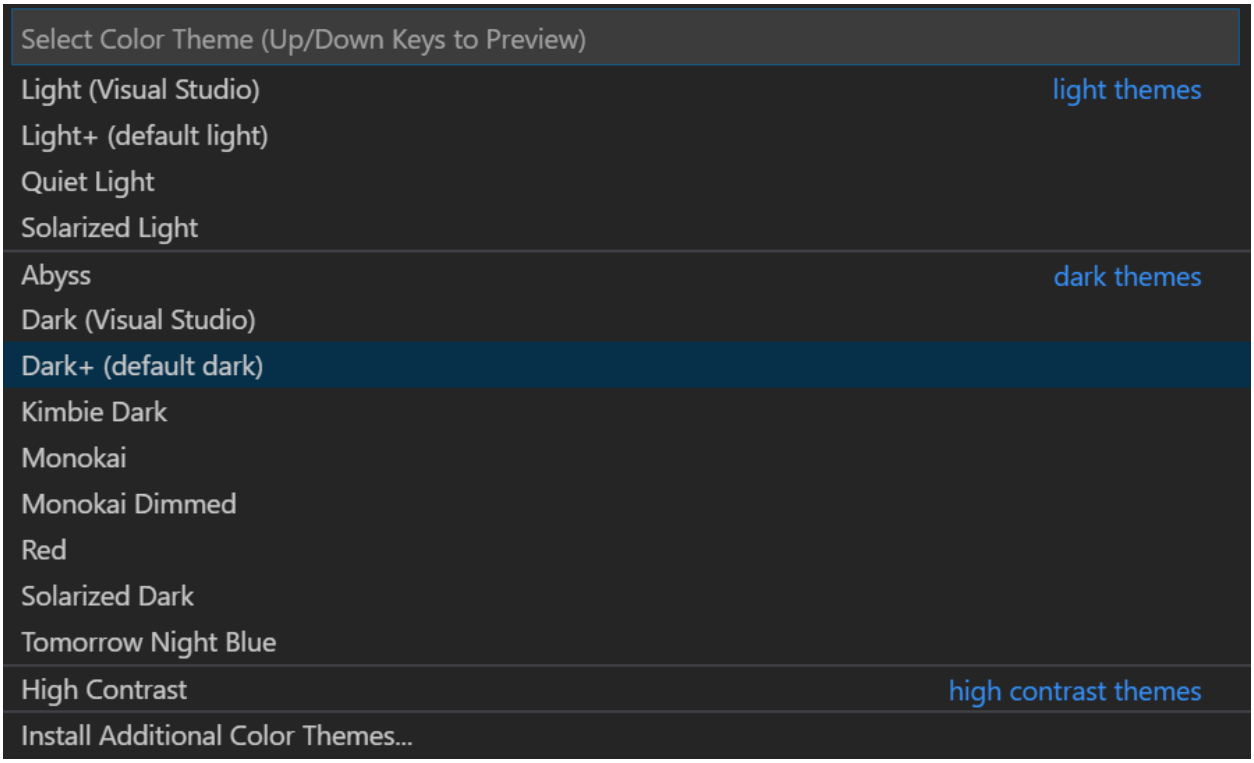
3. *Install & Change Theme Editor*

Pada ***Color Theme*** kita bisa memilih *theme editor* yang tersedia. Ada banyak pilihan dan kita juga bisa melakukan instalasi *Theme* yang telah disediakan komunitas, dibuat oleh *expert*.



Gambar 85 Color Theme Menu

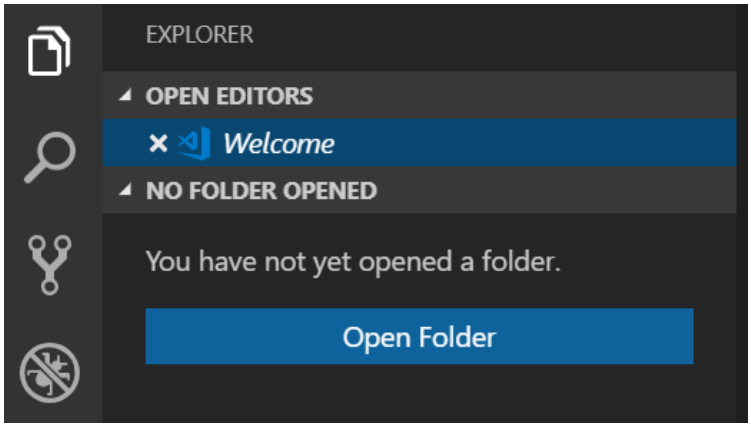
Jika anda kurang puas dengan *theme* yang tersedia anda bisa mencari *theme* lainya dengan memilih menu paling bawah **Install Additional Color Themes**.



Gambar 86 Install New Themes

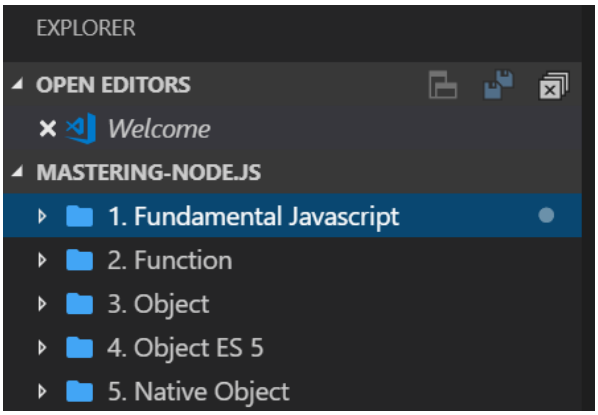
4. *The File Explorer*

Sekarang kita akan mempelajari ***file explorer*** yang disediakan oleh *viscode*. Klik **Open Folder** untuk membuka *folder* proyek yang pernah anda buat.



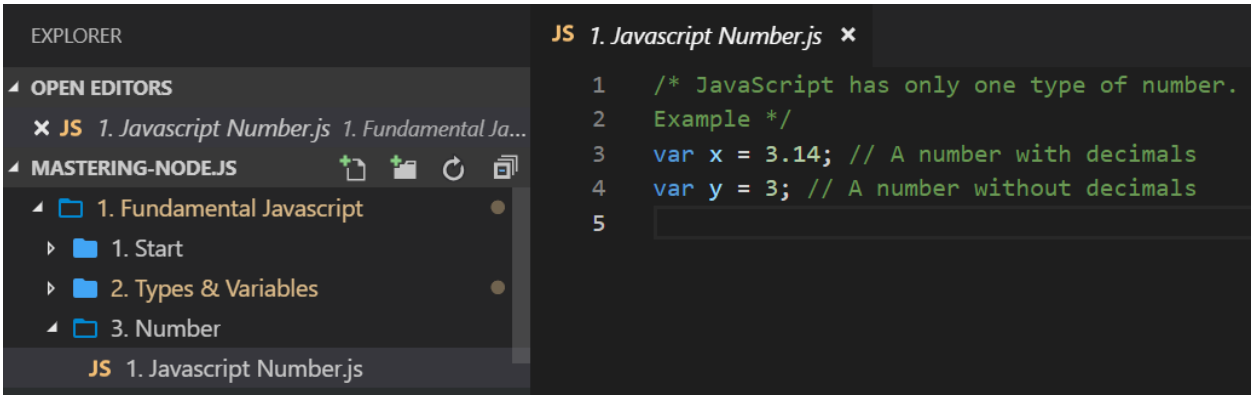
Gambar 87 File Explorer

Di bawah ini adalah daftar *folder* dan *file* yang telah penulis muat ke dalam *viscode explorer* :



Gambar 88 Display Folder in File Explorer

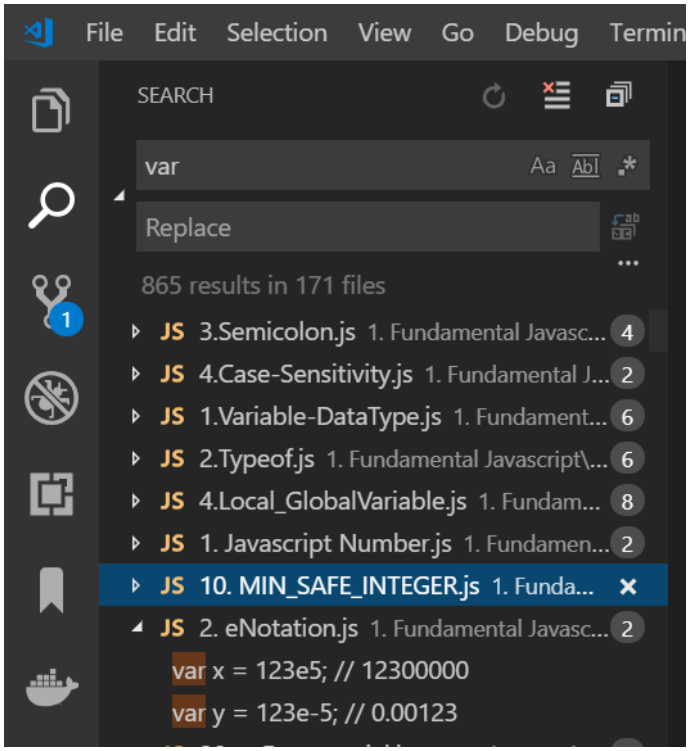
Jika kita ingin membuka salah satu *file* ke dalam *code editor*, klik *file* tersebut :



Gambar 89 Display File in The Code Editor

5. *Search Feature*

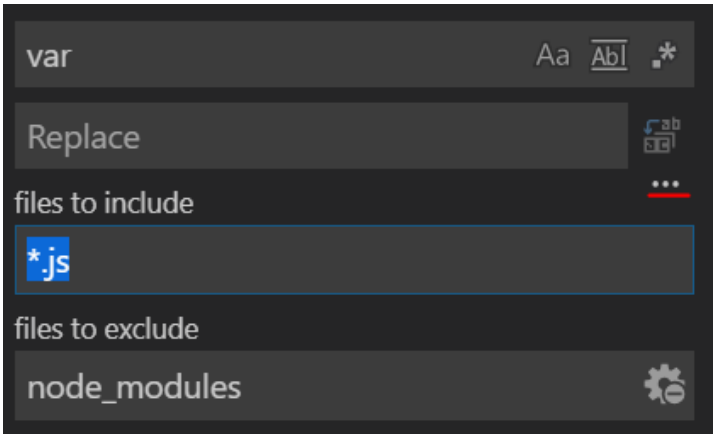
Viscode memiliki fitur yang bisa kita gunakan untuk mencari suatu *string* dalam *files* proyek yang kita muat. Klik gambar kaca pembesar, kemudian masukan *string* yang ingin kita cari. Pada kasus ini penulis memasukan *keyword* **var** :



Gambar 90 Search String

Pada gambar di atas kita bisa melihat ada banyak *file* yang memiliki ***string var***, kita bisa melakukan operasi ***replace*** pada seluruh *file* atau hanya pada satu *file* saja. Untuk membuka *file* yang memiliki ***string var*** anda tinggal klik daftar *file* yang muncul dalam kolom pencarian.

Kita juga bisa mengatur *file* dengan ekstensi apa saja yang akan kita cari dan membatasi *file* dan *folder* mana saja yang tidak ingin kita cari.

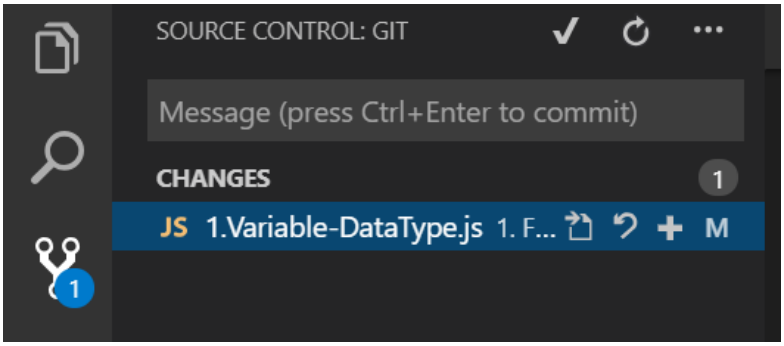


Gambar 91 Include & Exclude on Search String

Untuk melakukan pencarian dengan ***filter*** klik *icon* yang diberi garis merah, pada gambar di atas kita membatasi pencarian *string* hanya untuk *file* .js saja dan pencarian tidak boleh dilakukan didalam *folder* yang memiliki nama ***node_modules***.

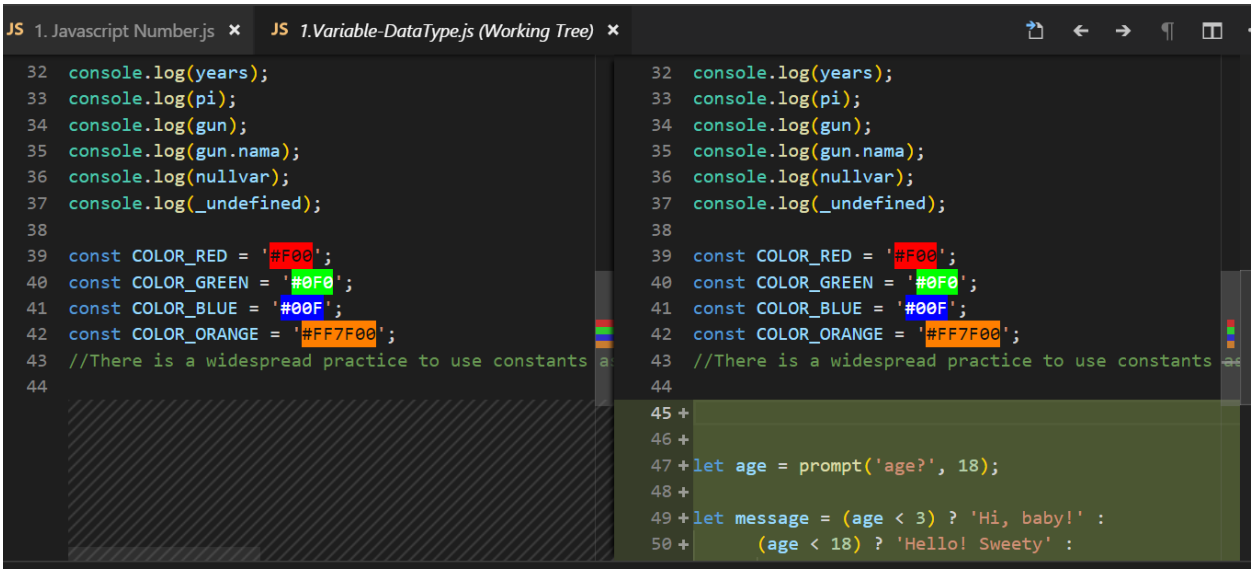
6. *Source Control*

Viscode mendukung **git** sehingga kita bisa melacak setiap perubahan yang terjadi. Untuk melihatnya klik ikon *working tree*, pada kasus ini penulis telah mengubah salah satu *file* di dalam proyek :



Gambar 92 Source Control

Jika kita klik *file* tersebut maka kita bisa melihat perubahan yang telah kita lakukan sebelum dan sesudahnya :



Gambar 93 Diff Mode

7. *Debugger*

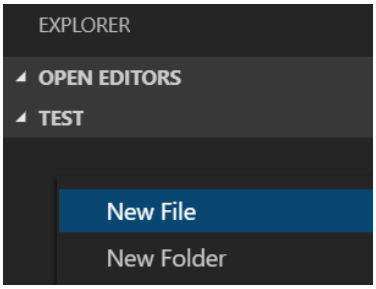
Viscode menyediakan *debugger* untuk berbagai bahasa pemrograman, kita akan mempelajari cara melakukan *debugging* dalam *node.js* pada *chapter Debugging Node.js*.

8. *Extension*

Sebelum mengeksplorasi *extension* buatlah sebuah *folder* dengan nama ***test*** sebagai tempat uji coba:

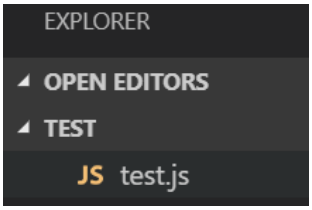
Gambar 94 Create New Folder

Kemudian muat *folder* tersebut sebagai *workspace* kedalam *viscode*, setelah itu buatlah *file* dengan nama **test.js** dengan cara melakukan klik kanan pada kolom *explorer* :



Gambar 95 Create New File

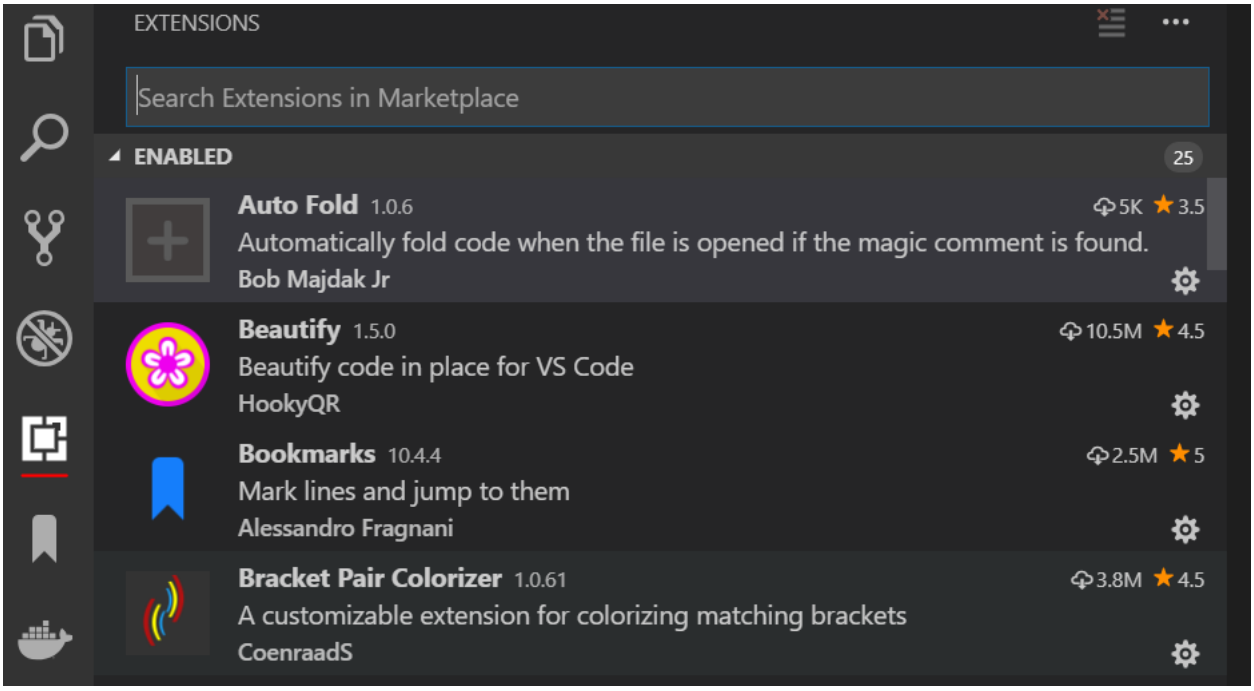
Biarkan *file* tersebut berisi kode kosong :



Gambar 96 Create New File

Setup selesai, sekarang kita bisa melanjutkan kajian *extension*.

Extension adalah tempat untuk menambahkan berbagai fitur yang dapat mempermudah dan mempercepat kita menulis kode. Produktivitas kita menjadi lebih masimal dengan *tools* yang telah disediakan dan dikembangkan oleh *expert* di komunitas *viscode*.



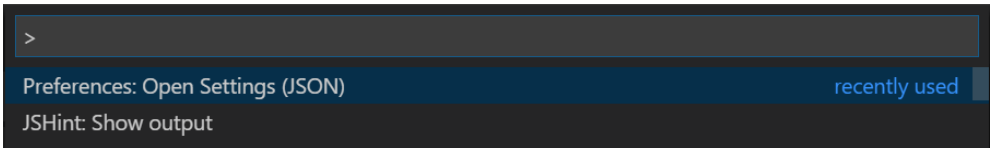
Gambar 97 Extension

Pada gambar di atas penulis sudah melakukan instalasi beberapa *extension* di antaranya adalah :

Auto Fold

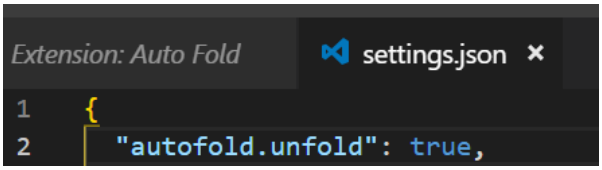
Ekstensi ini berguna agar *viscode* langsung membuka seluruh baris kode yang tertutup oleh *bracket* {...}, sehingga anda tidak perlu membukanya secara manual.

Install ekstensi ini kemudian tekan tombol **F1** pada *viscode* hingga muncul kolom perintah kemudian Ketik **Open Settings** seperti pada gambar di bawah ini :



Gambar 98 Viscode Settings

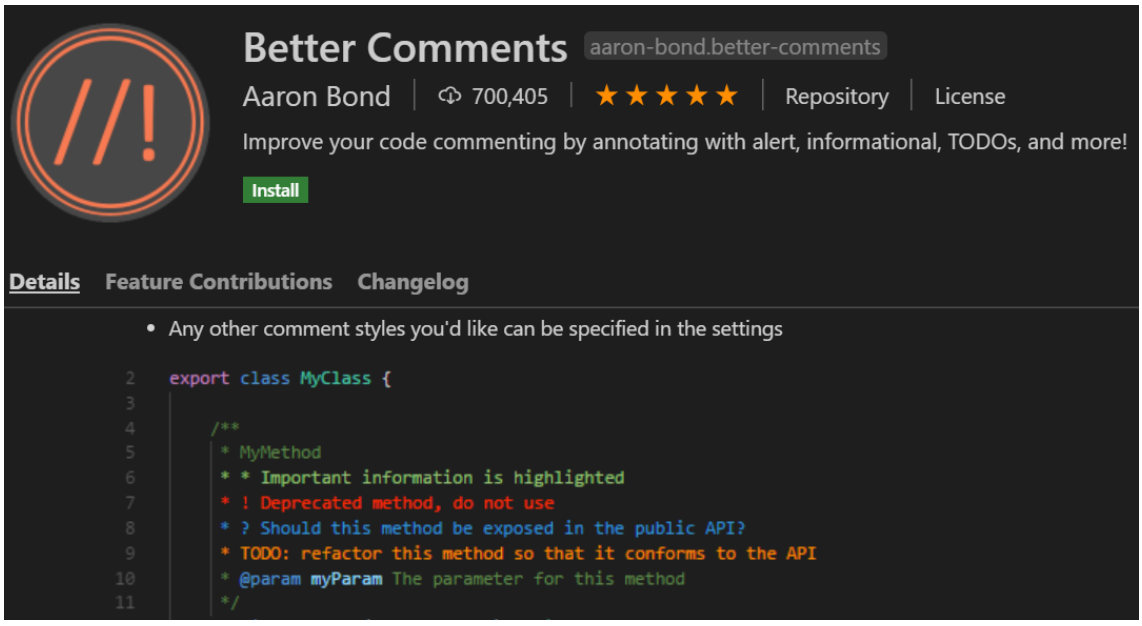
Pada **setting.json** masukan kode **"autofold.unfold": true**, agar ekstensi *auto fold* yang kita gunakan bekerja. Perhatikan gambar di bawah ini :



Gambar 99 Autofold Configuration

Better Comment

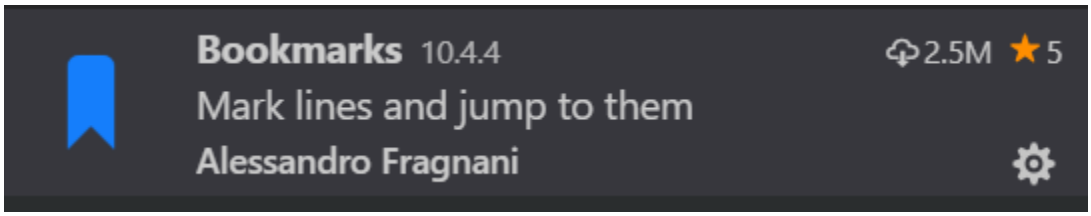
Ekstensi ***better comment*** membantu kita membedakan jenis komentar dengan warna menarik yang mudah diingat.



Gambar 100 Better Comment Extension

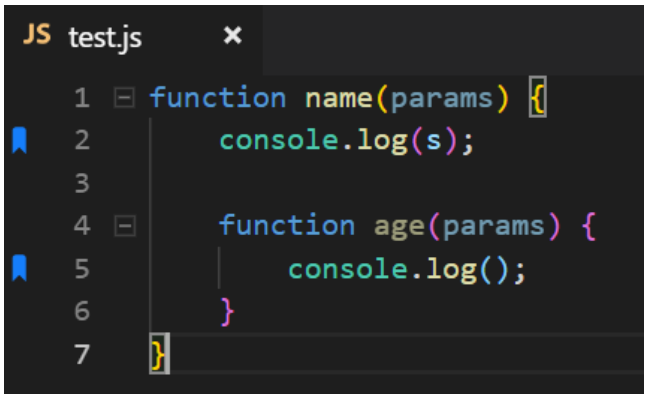
Bookmarks

Ekstensi ini sangat membantu jika jumlah kode yang telah kita buat sudah sangat panjang, baik itu ratusan ataupun ribuan baris. ***Scrolling code*** menjadi salah satu hal yang memakan waktu, untuk itu ***bookmarks code*** sangat membantu agar kita bisa meloncat ke alamat kode yang kita inginkan.



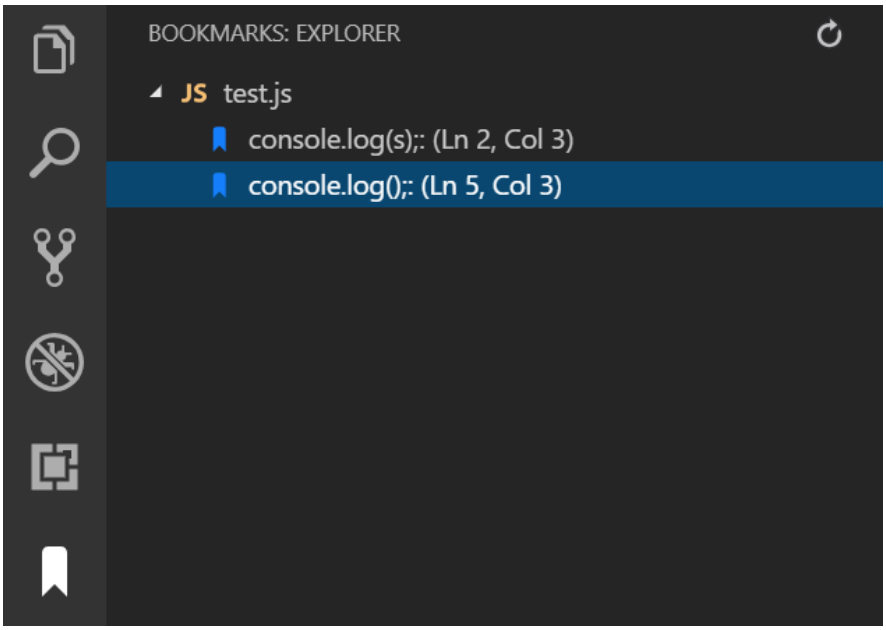
Gambar 101 Bookmarks Extension

Kita bisa melakukan *bookmark code* dengan cara menekan tombol **CTRL+ALT+K**, perhatikan gambar di bawah ini :



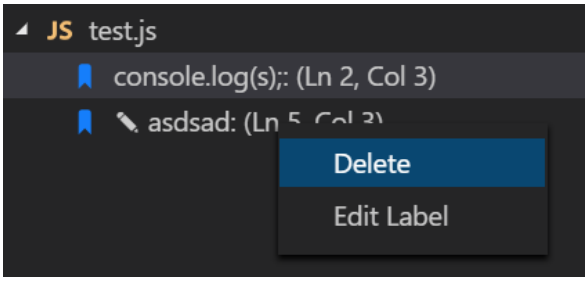
Gambar 102 Bookmarked Codes

Pada gambar di atas kita bisa melihat terdapat dua baris kode yang telah kita *bookmark*. Baris kode kedua dan baris kode kelima, jika kita ingin loncat kesetiap *bookmark* maka tekan tombol **CTRL+ALT+L**. Silahkan fahami terlebih dahulu.



Gambar 103 Bookmark Explorer

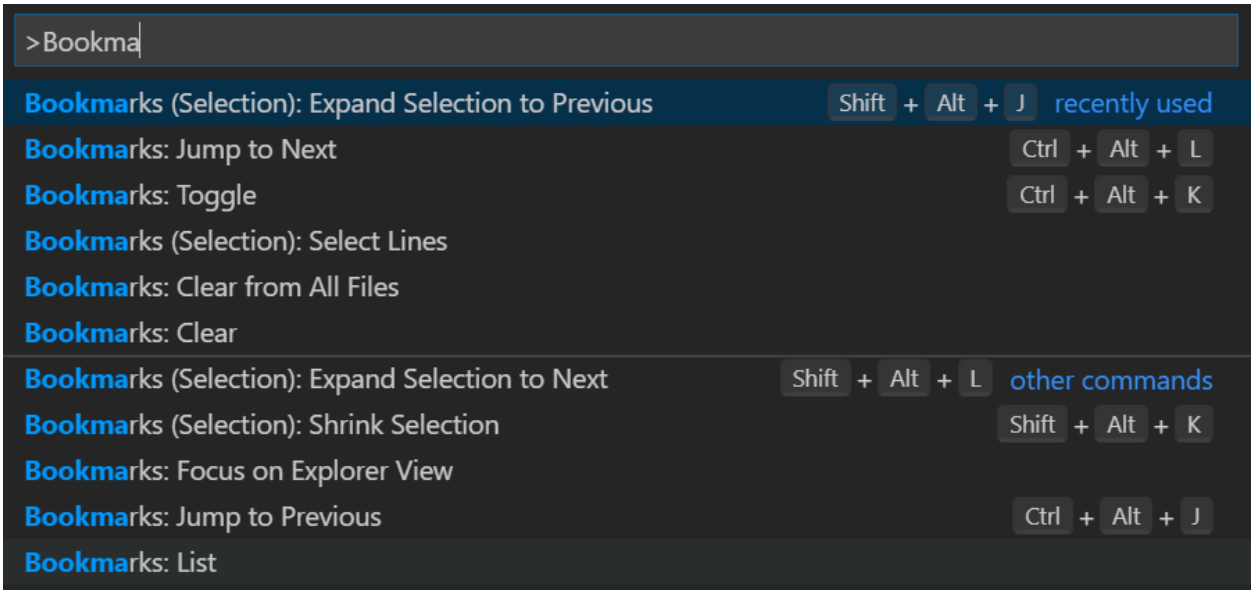
Pada *bookmark explorer* kita bisa melihat *file* apa saja dan alamat kode mana saja yang telah kita *bookmark*. Untuk mempermudah memahami kode yang telah kita *bookmark* kitapun bisa memberikan *label* pada kode yang telah kita *bookmark*.



Gambar *Bookmark – Delete & Edit Label*

Kita juga bisa menghapus kode yang telah kita *bookmark*, atau dengan cara menekan kembali **CTRL+ALT+K** pada baris kode yang telah kita berikan *bookmark*.

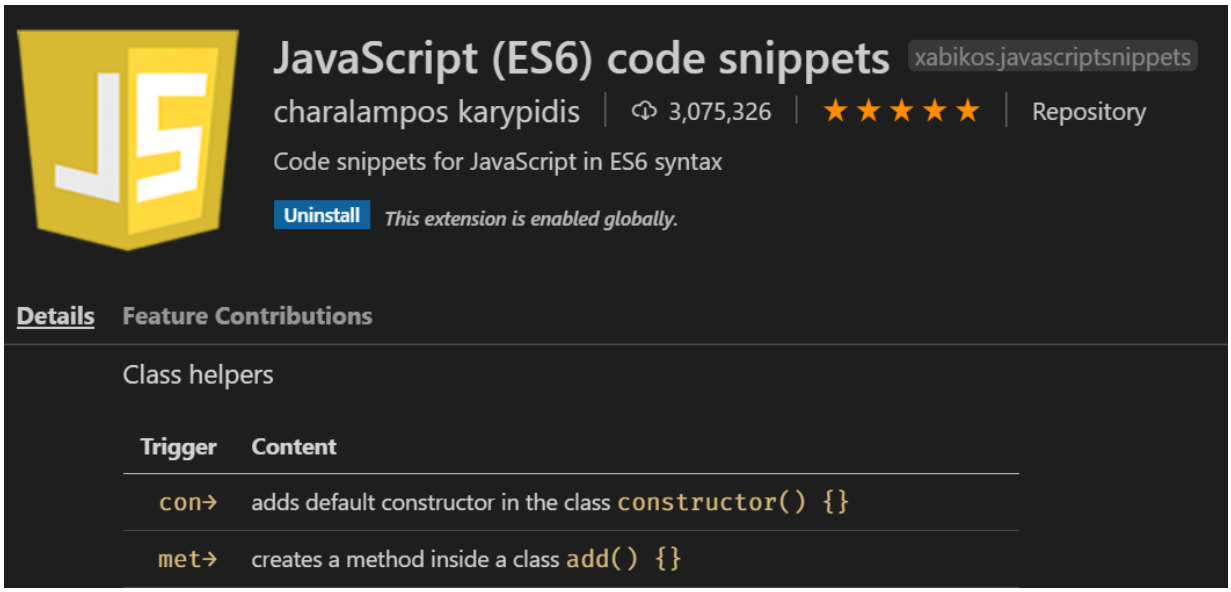
Untuk mengetahui ada fitur *bookmark* apa saja, tekan tombol **F1** kemudian ketikan **bookmarks** seperti pada gambar di bawah ini :



Gambar 104 Bookmark – List Commands

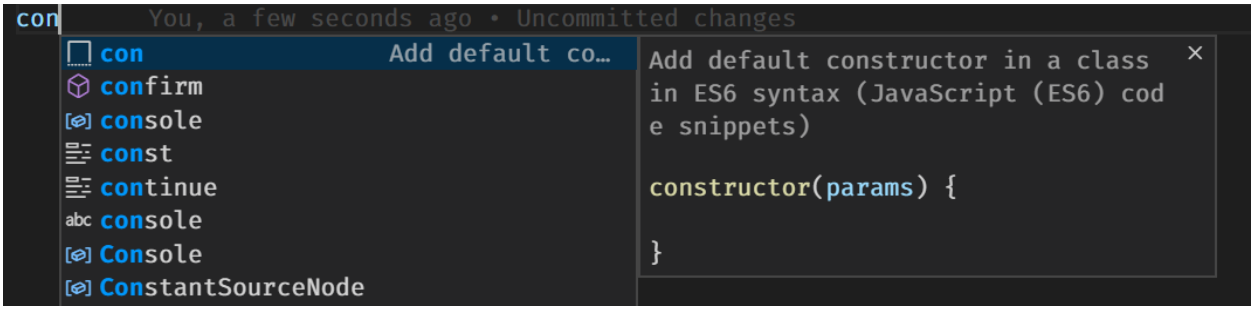
Javascript (ES6) Code Snippets

Ekstensi ini membantu kita untuk mempercepat kode yang akan kita tulis dengan sekumpulan ***snippets*** yang telah disediakan.



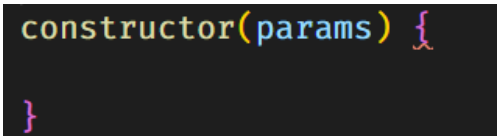
Gambar 105 Javascript ES6 Snippets

Sebagai contoh saat membuat sebuah **class** biasanya kita akan membuat sebuah **constructor** maka kita dapat mempersingkatnya dengan hanya menulis **con** :



Gambar 106 Constructor Snippets

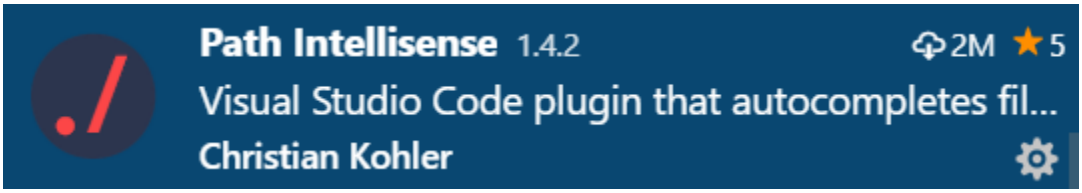
Jika kita enter maka kode akan langsung dibantu dibuat :



Gambar 107 Generated Snippet

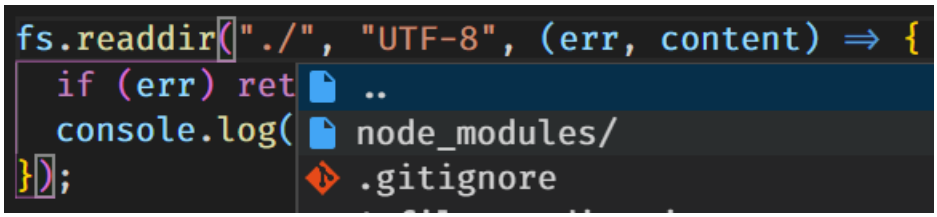
Path Intellisense

Ekstensi ini membantu kita saat kita berhadapan dengan *path*, ekstensi ini akan memberikan fitur ***autocomplete*** ketika ingin mengeksplorasi suatu *path* di dalam *code editor*.



Gambar 108 Path Intellisense Extension

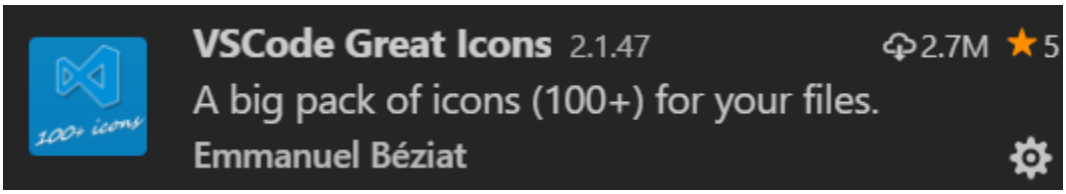
Sebagai contoh saat kita berinteraksi dengan *path* kita dapat melihat *list directory* yang tersedia pada *current directory* seperti pada gambar di bawah ini :



Gambar 109 Detected Path

VSCode Great Icons

Ekstensi ini sangat membantu untuk menampilkan *icon* untuk berbagai macam ekstensi agar menjadi lebih menarik lagi dan klasifikasi *file* menjadi mudah untuk dikenal.



Gambar 110 VSCode Great Icons

Ada banyak *Extension* yang bisa anda gunakan, silahkan berdiskusi, bertanya di komunitas dan forum online jika anda ingin mendapatkan insight lebih banyak lagi.

9. *The Terminal*

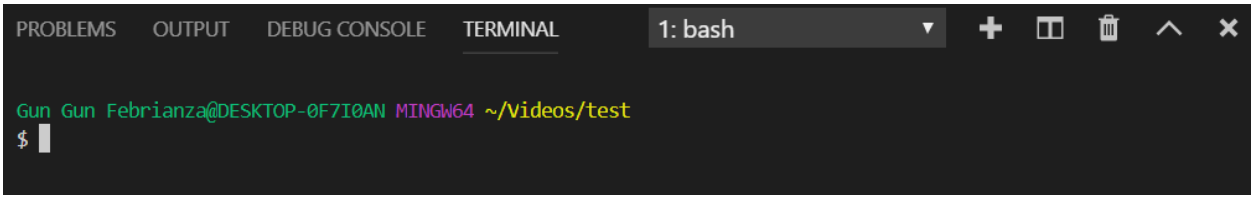
Fitur terminal dalam *code editor* membantu kita untuk bisa mengeksekusi berbagai macam perintah. Seperti mengeksekusi *file node.js*, *install package* atau mengeksekusi program lainya di dalam direktori proyek yang sedang anda bangun.

Pada **Settings.json** tambahkan kode di bawah ini agar terminal kita terintegrasi dengan ***bash*** :

Gambar 111 Shell Integration

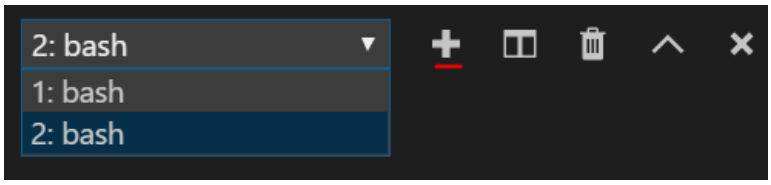
Notes
Pastikan anda sudah melakukan instalasi Git !

Untuk menampilkan terminal dalam *viscode* tekan tombol **CTRL+`**, maka *terminal* akan tampil seperti pada gambar di bawah ini :



Gambar 112 Terminal Visual Studio Code

Menambah *Terminal* Baru



Gambar 113 Add New Terminal

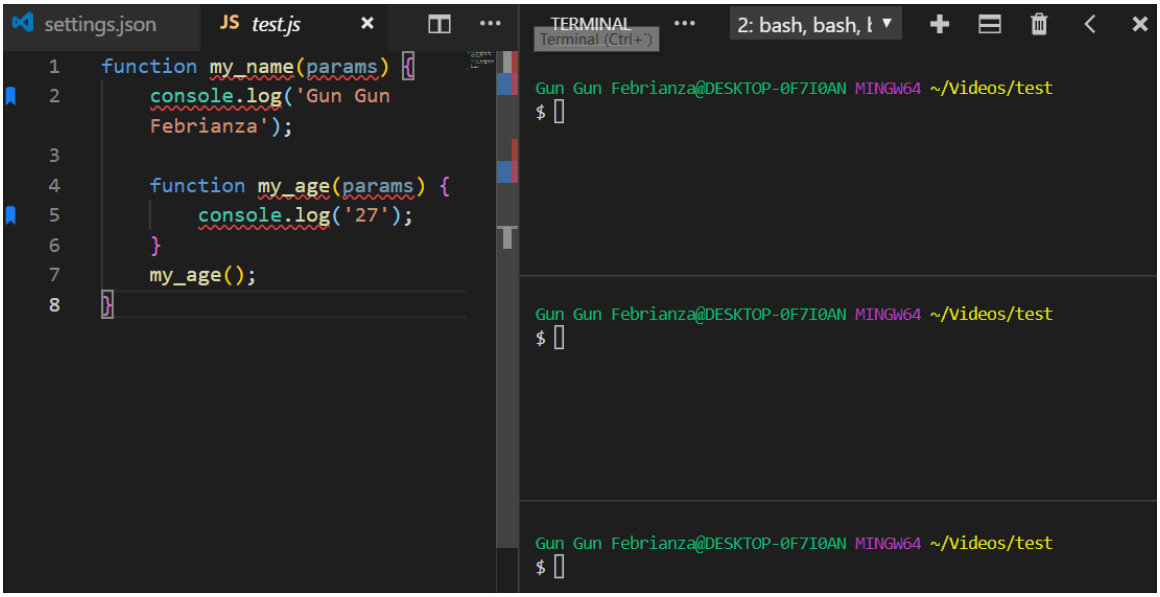
Tekan tombol tambah jika kita ingin menambahkan *terminal* yang baru, anda bisa melakukan *switch* pada *terminal* yang pertama dan kedua.

Melakukan *Split Terminal*

Gambar 114 Add New Terminal

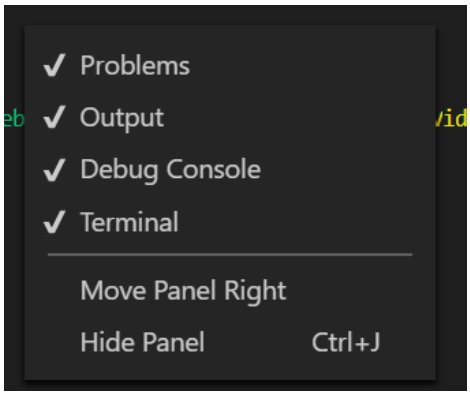
Bagi yang memiliki layar dengan lebar yang sangat panjang fitur ini sangat membantu. Sehingga kita tidak perlu menggunakan ***drop down*** untuk mengganti *channel terminal.*

Mengubah Posisi Terminal



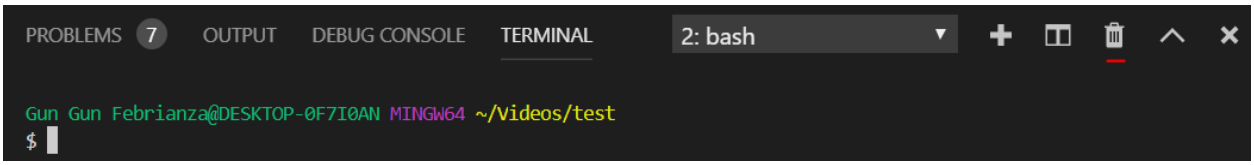
Gambar 115 Change Terminal Position

Kita bisa mengubah posisi terminal menjadi di sebelah kanan dengan cara melakukan klik kanan pada *terminal*, kemudian pilih menu *Move Panel Right* seperti pada gambar di bawah ini :



Gambar 116 Move Panel

Menghapus Terminal

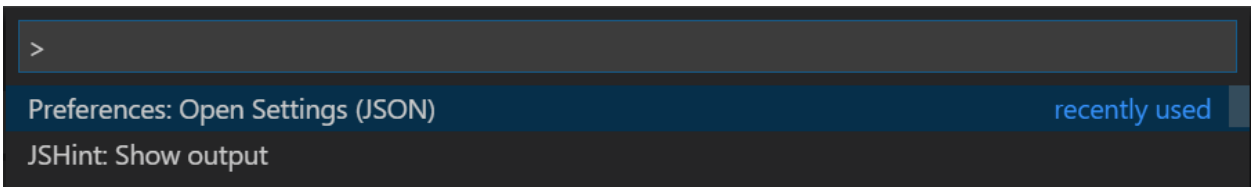


Gambar 117 Delete Terminal

10. *Performance Optimization*

Jika kita melakukan pengembangan aplikasi *node.js*, tentu kita akan menggunakan berbagai *module* sebagai *dependency* proyek yang kita bangun. *Module* tersebut sering kali menjadi penyebab *performance code editor* menjadi *slow*, ada beberapa konfigurasi yang dapat kita optimasi agar ***performance code editor*** kita menjadi lebih baik lagi.

Tekan tombol F1 pada *viscode* hingga muncul kolom perintah kemudian Ketik **Open Settings** seperti pada gambar di bawah ini :



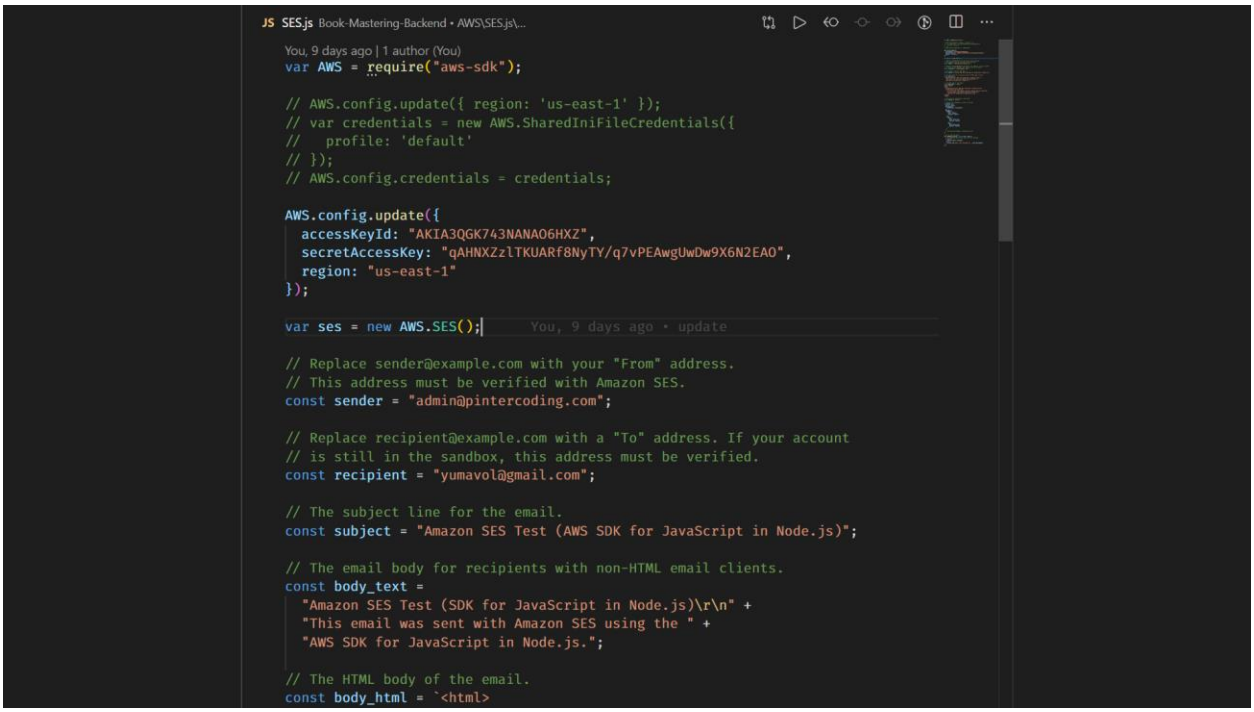
Gambar 118 Viscode Settings

Masukan kode di bawah ini kedalam konfigurasi :

```
"files.exclude": {
   "/.git": true,
   "/.DS_Store": true,
   "/node_modules": true,
   "/node_modules/": true
 },
 "search.exclude": {
   "/node_modules": true
 },
 "files.watcherExclude": {
   "/node_modules/": true
 }
```

11. *Zen Mode*

Agar kita dapat fokus pada kode yang kita tulis, kita bisa memasuki ***zen mode*** dengan menekan tombol **CTRL+K** sekali kemudian klik tombol **Z** :

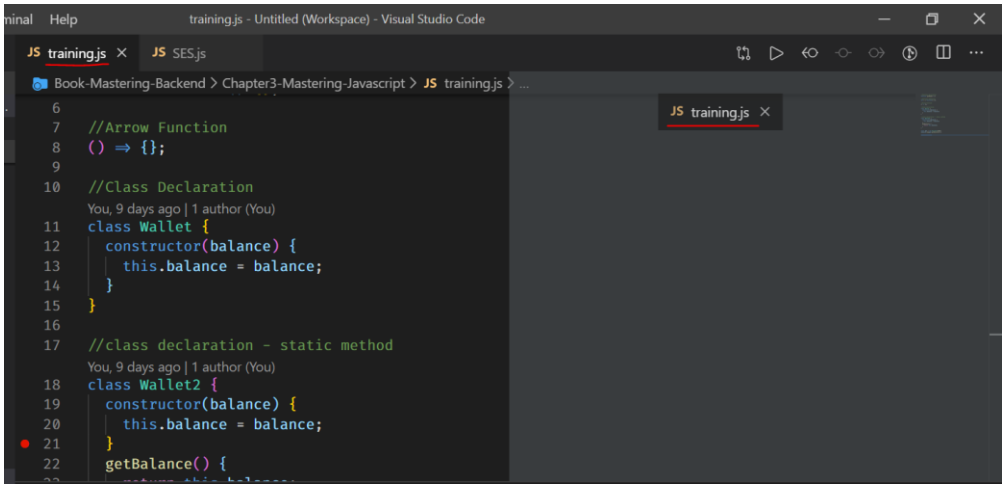


Gambar 119 Zen Mode

Untuk keluar dari ***zen mode*** tekan tombol **CTRL+K** sekali kemudian klik lagi tombol **Z**.

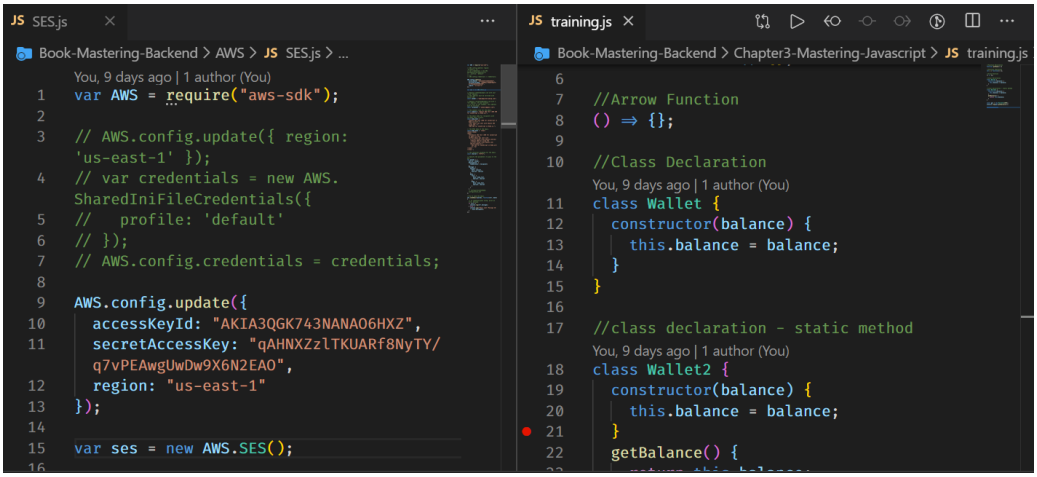
12. *Display Multiple File*

Terkadang ada saatnya kita ingin menampilkan dua kode sekaligus dalam satu ***code editor***, untuk melakukannya *drag* salah satu *file* ke arah kanan sampai muncul ***gradient*** warna yang berbeda membelah ***code editor*** :



Gambar 120 Display Multiple File

Jika sudah kita dapat melihat ***code editor*** kedua untuk mempermudah kita memahami kode, jika layar anda cukup lebar anda dapat menampilkan ***code editor*** ketiga dan seterusnya :



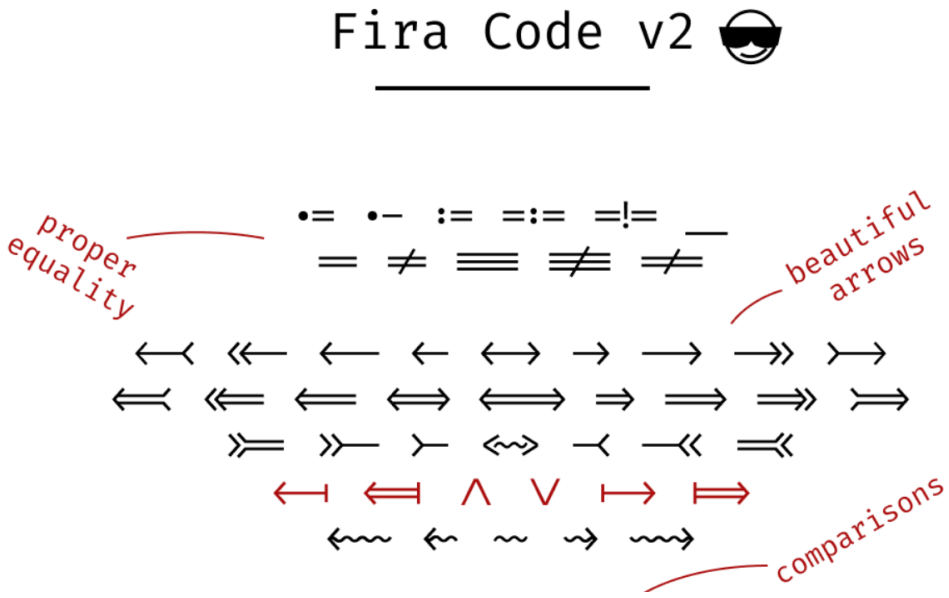
Gambar 121 Display Double File

13. *Font Ligature*

Terdapat font yang bagus untuk *code editor* yaitu ***Fira Code***, bisa anda cek disini :

https://github.com/tonsky/FiraCode

Jika kita menggunakan ***font*** **Fira code** kita akan memiliki efek ***symbol*** yang lebih mudah difahami untuk menulis kode :

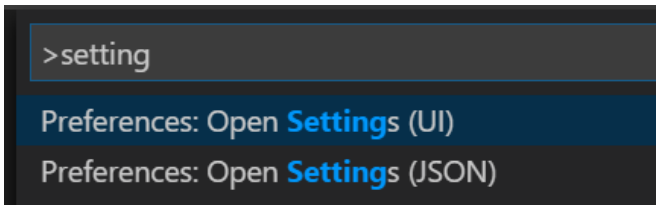


Gambar 122 Font Fira Code

Untuk cara instalasi Font dapat dibaca disini :

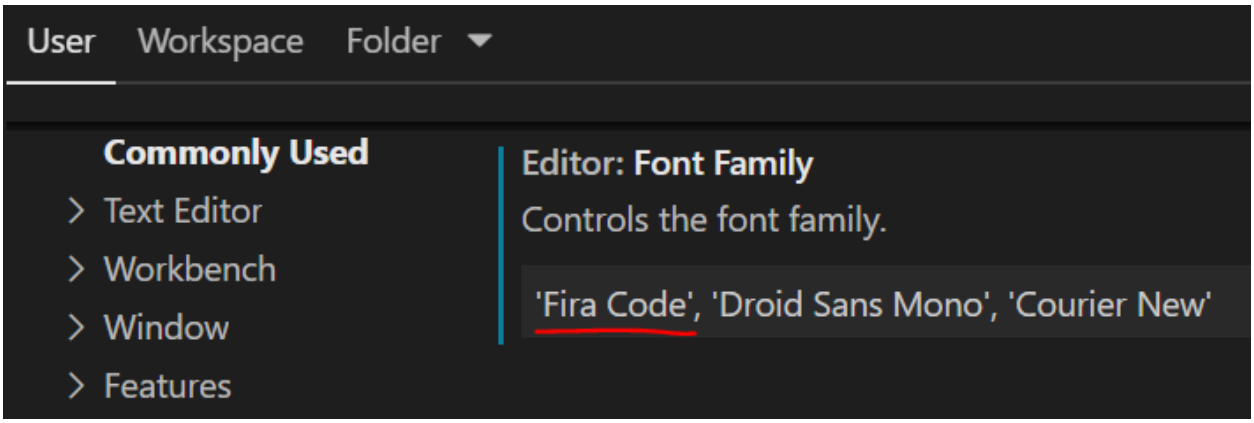
https://github.com/tonsky/FiraCode/wiki/Installing

Setelah anda melakukan instalasi font selanjutnya kita harus melakukan konfigurasi, tekan tombol **F1** kemudian ketik **settings**. Pilih **Open Settings (UI)** :



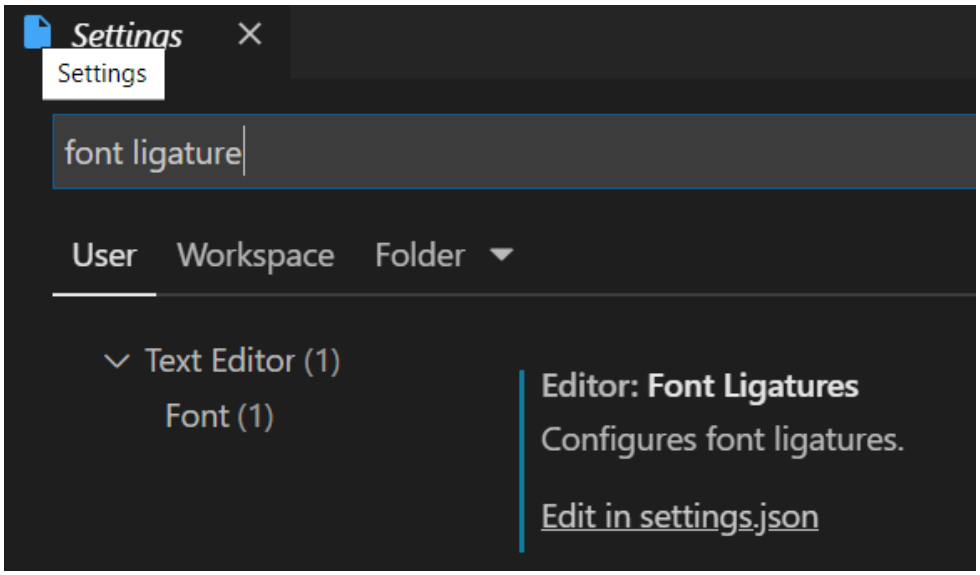
Gambar 123 Open Settings

Pada menu **Font Family** tambahkan **'Fira Code'** seperti gambar di bawah ini :



Gambar 124 Add Fira Code

Selanjutnya pada kolom pencarian ketik **font ligature**, klik **Edit in setting.json** :

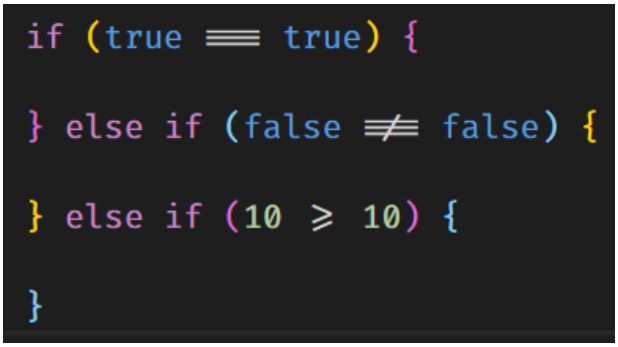


Gambar 125 Configure Font Ligature

Tambahkan pengaturan di bawah ini :

```
"editor.fontFamily": "'Fira Code', 'Droid Sans Mono', 'Courier New'",
"editor.fontLigatures": true,
```

Jika kita menulis kode seperti di bawah ini maka anda dapat melihat efeknya pada operator di dalam **logic if** dan **else if** :



Gambar 126 Fira Code Effect

Subchapter 2 – Web Browser

The original idea of the web was that it should be a collaborative space where you can communicate through sharing information.

— Tim Berners-Lee

Subchapter 2 – Objectives

- Mengenal *Web Browser*
- Mengenal *Web Console* dalam *browser*
- Mengenal *Multiline-editor* dalam *browser*

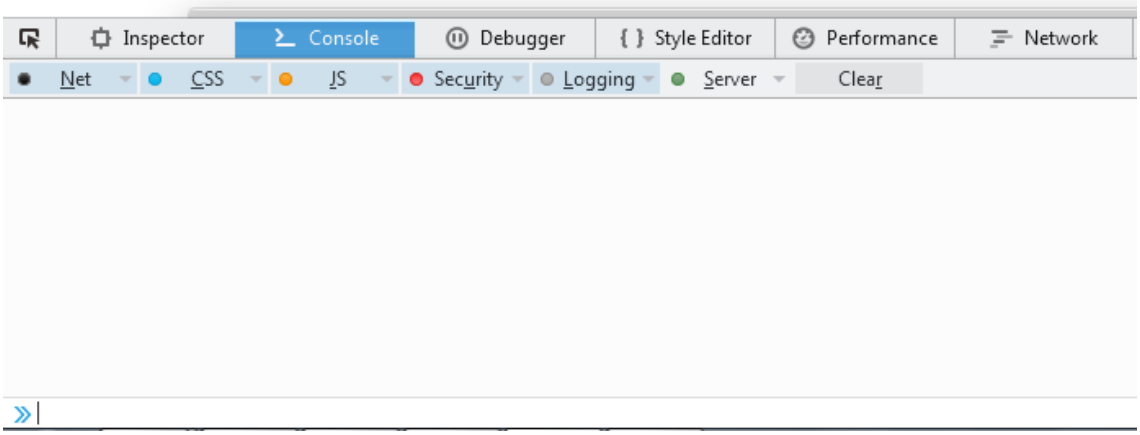
1. *Web Browser*

Anda bisa menggunakan *google chrome* atau *firefox*, namun pada buku ini penulis menggunakan *browser firefox*. Di dalam *firefox* terdapat fitur yang dapat membantu kita dalam mempelajari *javascript* yaitu web console yang menyediakan interpreter agar kita bisa mengeksekusi *javascript* pada *tab firefox*.

2. *WebConsole*

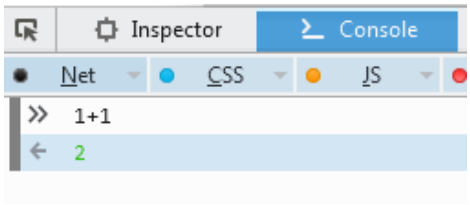
Dengan *web console* kita bisa mengetahui seluruh informasi yang terjadi di dalam suatu halaman *website*. Informasi yang dimaksud adalah *network request, javascript, css, security error* dan pesan peringatan lainya yang bisa anda pelajari lebih lanjut.

Kita bisa memanggilnya dengan menekan **CTRL+SHIFT+K** maka akan muncul sebuah *interpreter* pada *browser* seperti gambar di bawah ini :



Gambar 127 Web Console

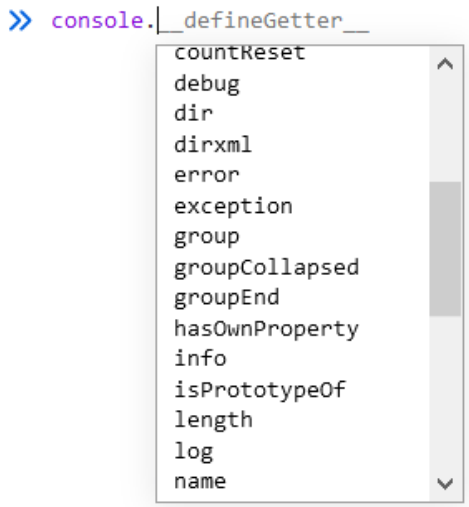
Untuk mengujinya coba ketik : ***1 + 1*** kemudian tekan *enter*. Hasilnya :



Gambar 128 Addition Operation

Autocomplete

Web console juga menyediakan fitur *autocomplete* untuk mempermudah kita belajar :



Gambar 129 Autocomplete

Syntax Highlighting

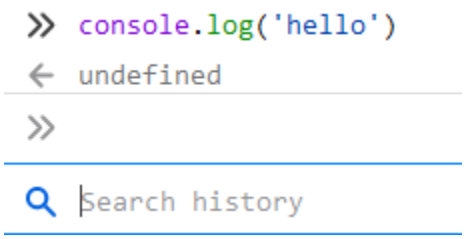
Web Console juga menyediakan fitur *syntax hightlighting* untuk membantu kita dalam membaca kode *javascript* yang ditulis :

```
» function person(firstname, lastname, age, eyecolor) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
    this.eyecolor = eyecolor;
  }

  person.prototype.getName = function () {
    return this.firstname + ' ' + this.lastname
  }

  var hooman = new person('Gun Gun', 'Febrianza', 28, 'brown');

  hooman
```

Gambar 130 Syntax Highlight Feature

Execution History

Setiap kali kita mengeksekusi suatu *statement, statement* tersebut akan dicatat di dalam *history*. Untuk mendapatkan *list statement* yang telah dieksekusi cukup tekan tombol *scroll* ke atas atau menekan tombol **F9**.



Gambar 131 Execution History

3. *Multiline Code Editor*

Dengan ***Web Console*** kita hanya bisa mengeksekusi satu *statements Javascript* saja, untuk lebih *advance* kita bisa menggunakan ***Multi-line code editor*** yang disediakan oleh *firefox*. Untuk memanggilnya tekan tombol **F12** kemudian tekan CTRL+B maka akan muncul seperti pada gambar di bawah ini :

Gambar 132 Multi-line code editor pada Firefox

Tulis kode di atas kemudian tekan **CTRL + ENTER** untuk mengeksekusinya, Maka hasilnya adalah :

Gambar 133 Hello Word Pada Javascript

Subchapter 3 – Javascript REPL

The strength of JavaScript is that you can do anything.
The weakness is that you will.

— Reg Braithwaite

Subchapter 3 – Objectives

- Mengenal Apa itu ***REPL*?**
- Mengenal Apa itu ***Shell*?**
- Mengenal Apa itu ***Node Virtual Machine*?**
- Mengenal Apa itu ***JSBin*?**
- Mengenal Apa itu ***WebConsole*?**
- Mengenal Apa itu ***Multi-line Code Editor*?**

1. *Node.js*

Node.js bekerja sebagai *javascript runtime* yang akan membaca sebuah *javascript code*. *Node.js* berdiri di atas *engine javascript* yang telah dioptimasi untuk melakukan kompilasi dan eksekusi kode *javascript*. *Engine Javascript* itu bernama **Google V8 *Javascript engine*** yang dibuat oleh tim *google developer* menggunakan bahasa pemrograman C++.

Gambar 134 Node Engine Symbol

Apa itu REPL?

Apakah sebelumnya anda pernah mendengar istilah REPL?

Artinya adalah **(*Read – Eval – Print – Loop*)**. REPL sering disebut juga dengan ***language shell***.

Apa itu Shell?

Lalu apa itu ***Shell****? shell* adalah sebuah *user interface* berbasis *CLI* (***Command Line Interface***) yang dapat kita gunakan untuk mengakses kemampuan (*OS API*) sebuah sistem operasi.

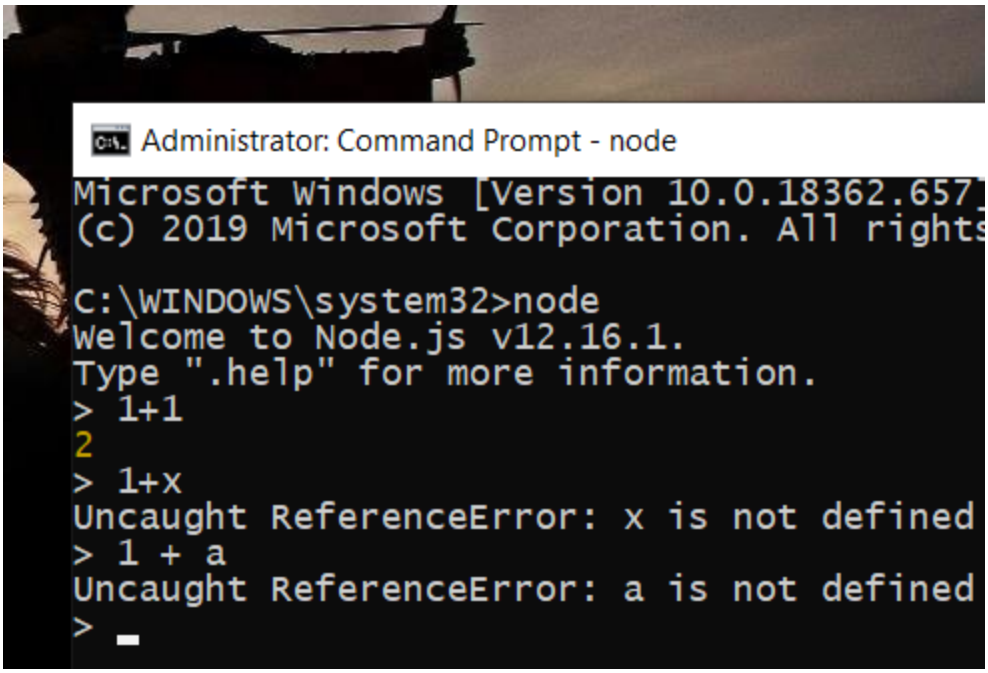
Setiap bahasa *scripting* pasti memiliki *REPL* sebagai tempat interaktif untuk melakukan *computing,* menerima *user input*, melakukan evaluasi dan memberikan hasilnya.

Node Virtual Machine

Buka *command prompt* kemudian pada terminal eksekusi perintah ***node*** :

```
node

Welcome to Node.js v12.16.1.

Type ".help" for more information.
```

Lalu buatlah sebuah *expression* ***1 + 1*** tekan **enter** lagi, maka kita dapat melihat hasil dari operasi penjumlahan tersebut :



Gambar 135 Node.js

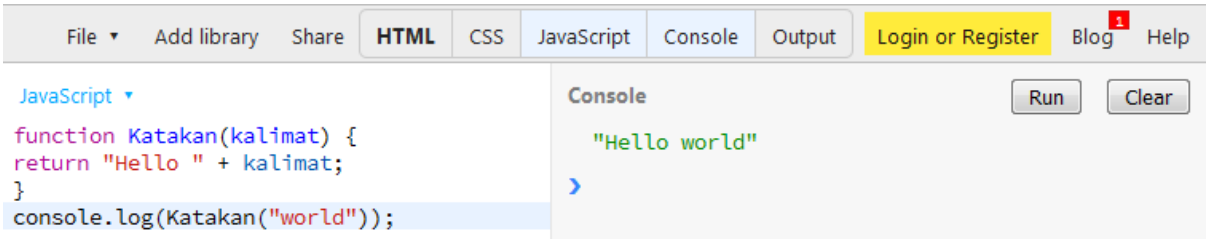
Namun jika kita mencoba melakukan kesalahan dengan membuat *expression* ***1 + a***, maka akan muncul *node error* seperti pada gambar di atas.

Jadi penulis merekomendasikan ada dua tempat menarik untuk mempelajari *REPL*, yaitu kita bisa mempelajarinya secara *online* menggunakan *JSBin* atau secara *offline* menggunakan *Web Console*.

2. *JSBin*

Untuk menggunakan **JSBin** kita bisa mengunjungi :

http://jsbin.com/nayiwopise/edit?js,console



Gambar 136 JSBin

Sekarang kita akan mempelajari ***hello world*** menggunakan layanan **JSBin**, buat fungsi katakan seperti pada gambar di atas kemudian tekan tombol *run*.

JSBin menyediakan banyak sekali *tools* yang bisa kita gunakan untuk mempelajari bahasa *javascript*, tersedia fitur *syntax highlight* agar kode lebih mudah dibaca, dan *runtime error detection*.

Selain menggunakan *JSBin* kita juga bisa mempelajari *javascript* secara *offline* dengan menggunakan *Browser Firefox* melalui ***Web Console*** untuk mempelajari REPL pada *Javascript*.

Chapter 3

Mastering Javascript

Subchapter 1 – *Introduction to Javascript*

JavaScript is the world's most misunderstood programming language.

— Douglas Crockford

Subchapter* 1 – *Objectives

- Mempelajari membuat ***hello world*** *&* ***Comment*** dalam ***javascript***
- Mempelajari ***Expression*** *&* ***Operator*** dalam ***javascript***
- Mempelajari ***Arithmetic Operator & Operation*** dalam ***javascript***
- Mempelajari ***Comparison Operator & Operation*** dalam ***javascript***
- Mempelajari ***Logical Operator & Operation*** dalam ***javascript***
- Mempelajari ***Assignment Operator & Operation*** dalam ***javascript***
- Mempelajari ***Javascript Strict Mode***
- Mempelajari ***Automatic Add Semicolon*** *&* ***Case Sensitivity***
- Mempelajari ***Variable Declaration*** dalam ***javascript***
- Mempelajari ***Reserved Words*** **dalam** ***javascript***
- Mempelajari ***Loosely Typed Language***

Javascript is an ***Interpreted Language****.*

Secara historis penciptaan *javascript* digunakan agar sebuah halaman *web* menjadi lebih hidup. *Javascript* adalah sebuah bahasa pemograman yang termasuk kedalam varian *script language, scripting* memerlukan program ***Interpreter*** agar bisa dimengerti oleh komputer. Saat kita mengeksekusi *javascript code* dalam sebuah *browser* maka sebuah *interpreter* dalam *browser* tersebut akan menerjemahkan *Javascript* menjadi sebuah *machine code*.

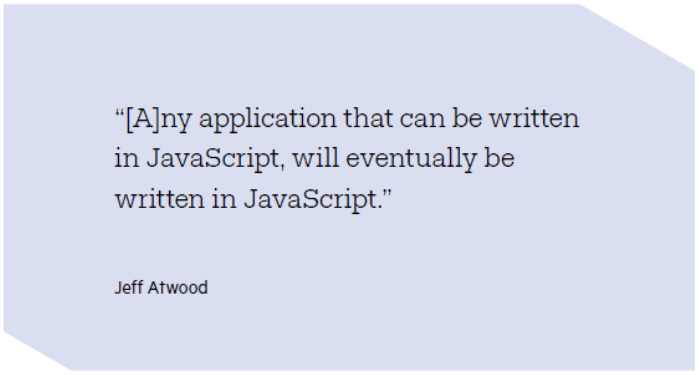
Machine code adalah sebuah bahasa yang dimengerti oleh komputer. *Machine code* adalah sebuah *string* yang terdiri dari bilangan biner 1(s) dan 0(s). Analoginya seperti mengkonversi bahasa inggris kedalam bahasa Indonesia sehingga dapat kita mengerti. Hanya saja agar bisa difahami oleh komputer *Javascript* harus di interpret (terjemahkan) menggunakan program bernama *interpreter* pada *browser* setiap kali kita mengeksekusi sebuah *Javascript code*.

Javascript adalah salah satu bahasa pemograman yang paling popular di dunia *programming*. Bahasa *javascript* adalah bahasa yang *versatile* karena bisa digunakan untuk membuat :

1. ***Web Application,***
2. ***Application Server,***
3. ***Dekstop Application,***
4. ***Mobile Application*** dan
5. ***Embedded System*** menggunakan *microcontroller*.

Artinya kita bisa memiliki potensi dan peluang yang besar jika mau mempelajari dan mengeksplorasinya, bahasa *javascript* memiliki *range of implementation* yang sangat luas.

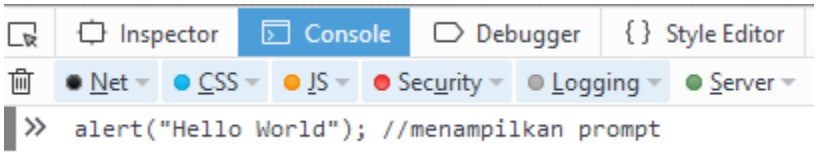
Sebelum menulis buku ini saya sempat membaca *quote* menarik tentang *javascript* dalam salah satu majalah tentang *javascript*.



Gambar 137 Javascript Quote by Jeff Atwood

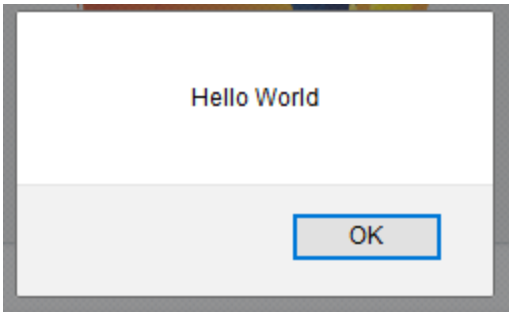
1. *Hello World*

Sudah menjadi tradisi jika kita berkenalan dengan suatu bahasa pemrograman kita akan membuat program yang paling sederhana yaitu **Hello World**. Buka *web console* dengan menekan tombol **CTRL+SHIFT+K** :



Gambar 138 Test Hello World

Tulis kode di atas kemudian tekan enter maka akan muncul *prompt* seperti pada gambar di bawah ini :



Gambar 139 Prompt Hello World

Pada kode di atas kita juga menggunakan ***single line comment***. Kita akan mempelajari cara membuat komentar di halaman selanjutnya.

2. *Comment*

Dalam prinsip dasar *software engineering*, kita harus membuat *code documentation* yang baik melalui komentar.

Manfaat penggunaan komentar adalah ketika program yang kita buat semakin kompleks dan banyak, maka *comments* yang kita buat bisa mempermudah kita untuk memahaminya kembali.

Jika anda bekerja dalam tim maka kode yang anda buat juga dapat mempermudah tim lainya untuk memahami kode yang anda buat.

Comment yang kita buat akan diabaikan oleh *javascript engine*.

Di bawah ini adalah contoh pembuatan *single line* dan *multi line comment* :

```
// Single Line Comment
// Short cut in atom & visual studio code is CTRL + /

/*
      Multi
      Line
      Comment
      */

/* To make multi comment shortcut
in atom & visual studio code is SHIFT + ALT + A */
```

**Link* sumber kode.

3. *Expression & Operator*

Statement

Apa itu *Statement*?

Statement adalah perintah untuk melakukan suatu aksi. Sebuah program terdiri dari sekumpulan *statement*. Sebuah *statement* bisa berupa :

Declaration Statement

Declaration statement untuk membuat sebuah variabel,

Assignment Statement

Assignment statement untuk memberikan nilai pada sebuah variabel,

Invocation Statement

Invocation statement untuk memanggil sebuah *block of code*,

Conditional Statement

Conditional statement untuk mengeksekusi kode berdasarkan kondisi,

Iteration Statement

Iteration statement untuk mengeksekusi kode secara berulang,

Disruptive Statement

Disruptive statement untuk keluar dari sebuah *control flow*.

Expression Statement

Expression statement untuk evaluasi sebuah *expression*,

Sebuah *statement* dapat memiliki sebuah *expression* atau tidak sama sekali (*non-expression statement*), tapi apa itu ***Expression***?

Expression

Expression adalah suatu *statement* yang digunakan untuk melakukan komputasi agar memproduksi suatu nilai. *Expression* adalah kombinasi dari *literal, variable*, *operand* dan *operator* yang dievaluasi untuk memproduksi sebuah *value*. Di bawah ini terdapat dua buah *statement* :

```
let x; //declaration statement
```

Pada *statement* pertama kode di atas terdapat perintah untuk melakukan deklarasi variabel.

```
let x; //declaration statement
x = 2 * 10; //expression statement
```

**Link* sumber kode.

Statement berikutnya di sebut dengan *expression statement* karena terdapat operasi untuk memproduksi suatu nilai.

Operator & Operand

Operator yang digunakan dalam *expression* tersebut adalah *multiplication* (*), angka 2 dan 10 adalah sebuah *operand*.

Operator Precedence

Ketika sebuah *expression* memiliki lebih dari satu *operator* maka terdapat aturan untuk mengatur *operator precedence*.

Pada contoh kode di bawah ini *interpreter* mengetahui mana operasi aritmetika yang harus dilakukan terlebih dahulu, yaitu perkalian dahulu kemudian penjumlahan :

```
function arithmeticOperation(params) {
  let x           //declaration statement
  x = 2 + 2 * 10  //expression statement
  console.log(x); //22
}

arithmeticOperation()
```

**Link* sumber kode.

Arithmetic Operator

Di bawah ini adalah *arithmetic operator* dalam *javascript*. Tidak hanya penjumlahan, pengurangan, perkalian dan pembagian. *Javascript* menyediakan *operator* untuk *remainder*, *unary*, *pre & post increment* dan *pre & post decrement* :

Table 5 Arithmetic Operator

Operator	***Description***	***Example***
+	*Addition (String Concat)* atau penjumlahan	`2+1 //3`
-	*Subtraction* atau pengurangan	`2-1 //1`
/	*Division* atau pembagian	`6/2 //3`
*	*Multiplication* atau perkalian	`3*2 //6`
%	*Remainder*	`3%2 //1`
-	*Unary Negation*	`-x // negative x`
+	*Unary Plus*	`+x // positive x`

++	*Pre-Increment*	++x
++	*Post-Increment*	X++
--	*Pre-Decrement*	--x
--	*Post-Decrement*	x--

Arithmetic Operation

Contoh *arithmetic operation* dalam *javascript* :

```
var a = 3;
var x = (100 + 50) * a;
console.log(x);
```

**Link* sumber kode.

Modulus Operation

Contoh *modulus operation* dalam *javascript* :

```
console.log(12 % 5); // 2
console.log(-1 % 2); // -1
console.log(1 % -2); // 1
console.log(NaN % 2); // NaN
console.log(1 % 2); // 1
console.log(2 % 3); // 2
console.log(-4 % 2); // -0
console.log(5.5 % 2); // 1.5
```

**Link* sumber kode.

Pada baris pertama terdapat 12 mod 5 hasilnya adalah 2, karena angka *integer* 5 maksimum hanya dapat di kalikan dua kali saja (5*2) atau (5+5) yaitu sama dengan 10.

Angka *integer* 5 tidak bisa dikalikan tiga kali karena tidak bisa lebih dari 12. Angka *integer* hanya dapat dikalikan dua kali saja maka hasil mod adalah 12-10 = 2

Anda akan mempelajari *NaN* di bab tentang tipe data, *NaN* artinya *Not a Number* sebuah nilai yang dihasilkan jika kita gagal melakukan operasi aritmetika pada *integer*.

Addition Operation

Contoh *addition operation* dalam *javascript* :

```
  //The addition operator produces the sum of numeric operands
or string concatenation.
  // Number + Number -> addition
  console.log(1 + 2); // 3

  // Boolean + Number -> addition
  console.log(true + 1); // 2

  // Boolean + Boolean -> addition
  console.log(false + false); // 0

  // Number + String -> concatenation
  console.log(5 + 'foo'); // "5foo"

  // String + Boolean -> concatenation
  console.log('foo' + false); // "foofalse"

  // String + String -> concatenation
  console.log('foo' + 'bar'); // "foobar"
```

**Link* sumber kode.

Pada kode di atas jika kita melakukan operasi *addition* antara :

Integer & Integer Addition

Tipe data *integer* dan *integer* hasilnya adalah *integer*.

Boolean & Integer Addition

Tipe data *boolean* dan *integer* hasilnya adalah *integer*.

Tipe data *true* memiliki nilai *integer* 1.

Boolean & Boolean Addition

Tipe data *boolean* dan *boolean* hasilnya adalah *integer*.

Tipe data *false* memiliki nilai *integer* 0.

Integer & String Addition

Tipe data *integer* dan *string* hasilnya adalah *string*.

String & Boolean Addition

Tipe data *string* dan *boolean* hasilnya adalah *string*.

String & String Addition

Tipe data *string* dan *string* hasilnya adalah *string*.

Subtraction Operation

Contoh *subtraction operation* dalam *javascript* :

```
console.log(5 - 3); // 2
console.log(3 - 5); // -2
console.log('foo' - 3); // NaN
```

**Link* sumber kode.

Pada kode di atas kita melakukan operasi pengurangan, hasilnya dapat berupa *integer* positif atau *integer* negatif. Operasi yang tidak legal akan menghasilkan *NaN* (*Not a Number*).

Division Operation

Contoh *division operation* dalam *javascript* :

```
console.log(1 / 2); // returns 0.5 in JavaScript
console.log(1.0 / 2.0); // returns 0.5 in both js & Java
console.log(2.0 / 0); // returns Infinity in JavaScript
console.log(2.0 / 0.0); // returns Infinity too
console.log(2.0 / -0.0); // returns -Infinity in JavaScript
```

**Link* sumber kode.

Pada kode di atas kita melakukan operasi pembagian, operasi pembagian juga dapat dilakukan pada *decimal number*. Jika *decimal number* di bagi dengan angka nol maka akan menghasilkan *Infinity*. Anda akan mengenal lebih dalam apa itu *infinity* nanti di bab tentang tipe data *number*.

Multiplication Operation

Contoh *multiplication operation* dalam *javascript* :

```
console.log(2 * 2); // 4
console.log(-2 * 2); // -4
console.log(Infinity * 0); // NaN
console.log(Infinity * Infinity); // Infinity
console.log('foo' * 2); // NaN
```

**Link* sumber kode.

Pada kode di atas kita melakukan operasi perkalian, operasi perkalian antara *string* dan *integer* akan menghasilkan *NaN*, perkalian *infinity* dengan *infinity* akan menghasilkan *infinity* dan perkalian *infinity* dengan 0 hasilnya akan *NaN*.

Exponentiation

Contoh *exponentiation* dalam *javascript* :

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(3 ** 2.5); // 15.588457268119896
console.log(10 ** -1); // 0.1
console.log(NaN ** 2); // NaN
console.log(2 ** (3 ** 2)); // 512
console.log(2 ** (3 ** 2)); // 512
console.log((2 ** 3) ** 2); // 64
```

**Link* sumber kode.

Pada kode di atas kita melakukan operasi eksponensiasi, 2 pangkat 3 hasilnya adalah 8. Pangkat dapat berupa *integer*, *decimal number* atau *negative number*. Operasi eksponensiasi dengan *NaN* akan menghasilkan *NaN* (*Not a Number*).

Increment & Decrement Operation

Contoh *increment & decrement operation* dalam *javascript* :

```
var a = 5, b = 5;
var a = ++a;
var b = --b;
console.log(a);
console.log(b);
```

**Link* sumber kode.

Operator ++ digunakan untuk menambahkan 1 *integer* dan operator -- digunakan untuk mengurangi 1 *integer*. Penambahan operator diawal *increment* atau *decrement* di awal disebut dengan *pre-increment & pre-decrement*. Penambahan operator *increment* atau *decrement* di akhir disebut dengan *post-increment & post-decrement*.

Comparison Operator

Ada beberapa ***Comparison Operator / Relational Operator*** yang bisa kita gunakan untuk mengatur kondisi dan *control flow*. Bisa kita lihat pada tabel *Comparison Operator* di bawah ini, sebagai contoh diketahui nilai x = 5 maka :

Table 6 Comparison Operator

Operator	***Description***	***Comparing***	***Return***
==	Equal (Setara)	x == 9	*False*
===	*Data Type* & *Value Equal (*Tipe data dan Nilai Setara)	x === 5	*True*
!=	Not Equal (Tidak Setara)	x != 8	*True*
!==	*Data Type* or *Value Not Equal (*Tipe data dan nilai tidak setara*)*	x !== ”5”	*True*
>	Greater than (Lebih besar dari)	x > 8	*False*
<	Less than (Lebih Kecil dari)	x < 3	*False*
>=	Greater than or Equal (Lebih besar atau setara)	x >= 5	*True*
<=	Less than or Equal (Lebih kecil atau setara)	x <= 1	*False*

Equality Operator

Contoh penggunaan *equality operator* dalam *javascript* :

```
const x = 5;
console.log(x == 8); //false

const x = 5;
console.log(x == 5); //true

console.log(x === 5); //true
console.log(x === '5'); //false
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat perbedaan penggunaan *operator* `==` dengan `===`, pada *operator* `===` terdapat pemeriksaan apakah nilai yang dibandingkan memiliki tipe data yang sama atau tidak. Sedangkan pada *operator* `==` hanya membandingkan nilainya saja tanpa memeriksa kesamaan tipe datanya.

Inequality Operator

Contoh penggunaan *inequality operator* dalam *javascript* :

```
var x = 5;
console.log(x != 8); //true
console.log(x != 5); //false

console.log(x !== 5); //false
console.log(x !== '5'); //true
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat perbedaan penggunaan *operator* `!=` dengan `!==`, pada *operator* `!=` jika nilai tidak sama maka hasilnya akan *true*. Pada *operator* `!==` jika

nilai pembandingnya tidak sama dan tipe datanya juga tidak sama maka hasilnya akan *true*.

Greater Than Operator

Contoh penggunaan *Greater than operator* dalam *javascript* :

```
var x = 5;
console.log(x > 6); //false
console.log(x > 5); //false
console.log(x > 4); //true
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat contoh penggunaan *greater than* untuk membandingkan dua buah nilai menggunakan *javascript*.

Less Than Operator

Contoh penggunaan *Less than operator* dalam *javascript* :

```
var x = 5;
console.log(x < 6); //true
console.log(x < 5); //false
console.log(x < 4); //false
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat contoh penggunaan *less than* untuk membandingkan dua buah nilai menggunakan *javascript*.

Greater Than or Equal Operator

Contoh penggunaan *Greater than or Equal operator* dalam *javascript* :

```
var x = 5;
console.log(x >= 6); //false
console.log(x >= 5); //true
console.log(x >= 4); //true
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat contoh penggunaan *greater than or equal* untuk membandingkan dua buah nilai menggunakan *javascript*.

Less Than or Equal Operator

Contoh penggunaan *Less than or Equal operator* dalam *javascript* :

```
var x = 5;
console.log(x <= 6); //true
console.log(x <= 5); //true
console.log(x <= 4); //false
```

**Link* sumber kode.

Pada kode di atas kita dapat melihat contoh penggunaan *Less than or equal* untuk membandingkan dua buah nilai menggunakan *javascript*.

Logical Operator

Selain *comparison operator* terdapat juga ***logical operator.*** Jika terdapat lebih dari satu perbandingan dalam satu kondisi kita bisa menggunakan *logical operator*. Bisa kita lihat pada tabel *Logical Operator* di bawah ini, sebagai contoh diketahui x = 4 dan y = 5 maka :

Table 7 Logical Operator

Operator	***Description***	***Comparing***	***Return***
&&	And	(x > 3 && y = 5)	*True*
||	Or	(x > 3 || y = 5)	*True*
!	Not	!(x!=y)	*False*

Operator OR

Operator OR (**||**) digunakan jika kondisi nilai yang diberikan benar atau dapat diterima oleh salah satu kondisi. Perhatikan contoh kode di bawah ini :

```
let hour = 8;

if (hour < 9 || hour > 17) {
  console.log('Warung masih tutup');
}
```

**Link* sumber kode.

Operator AND

Operator AND (**&&**) digunakan jika kita ingin menguji nilai yang diberikan namun harus memenuhi dua kondisi sekaligus. Perhatikan kode di bawah ini :

```
let hour = 04;
let minute = 30;

if (hour == 04 && minute == 30) {
  console.log('Alarm on!');
```

```
}
```

**Link* sumber kode.

Operator NOT

Operator NOT (**!**) digunakan jika kita ingin mengubah suatu *operand* kedalam *data type boolean*, *return* berupa *true or false* secara kebalikan (*inverse value*). Perhatikan contoh kode di bawah ini :

```
console.log(!true); // false
console.log(!0); // true
```

**Link* sumber kode.

Assignment Operator

Di bawah ini adalah *assignment operator* dalam *javascript*. Kita dapat menggunakanya untuk menetapkan sebuah nilai dengan berbagai cara :

Table 8 Assignment Operator

Operator	***Description***	***Example***
=	Menetapkan sebuah *literal* pada sebuah variabel. *Literal* dapat berupa data tunggal atau sebuah *expression*.	C = A → C = 12 C = A * B → C = 12 * 2
+=	Menetapkan sebuah *literal* pada *operand* kiri, hasil dari *operand* kiri adalah penjumlahan *operand* kiri dengan *operand* kanan.	C += A → C = C + A 12 += 2 → 12 = 12 + 2

-=	Menetapkan sebuah literal pada *operand* kiri, hasil dari *operand* kiri adalah selisih dari *operand* kiri dengan *operand* kanan.	C -= A → C = C - A 12 -= 2 → 12 = 12 - 2
*=	Menetapkan sebuah literal pada *operand* kiri, hasil dari *operand* kiri adalah perkalian *operand* kiri dengan *operand* kanan.	C *= A → C = C * A 12 *= 2 → 12 = 12 * 2
/=	Menetapkan sebuah literal pada *operand* kiri, hasil dari *operand* kiri adalah pembagian *operand* kiri dengan *operand* kanan.	C /= A → C = C / A 12 /= 2 → 12 = 12 / 2
%=	Menetapkan sebuah literal pada *operand* kiri, hasil dari *operand* kiri adalah modulus *operand* kiri dengan *operand* kanan.	C %= A → C = C % A 12 %= 2 → 12 = 12 % 2

4. *Javascript Strict Mode*

Pembuatan kode *javascript* secara *Strict Mode* muncul semenjak ***ECMAScript*** ke **5** muncul, gunanya agar kode *javascript* yang dihasilkan lebih bersih dan baik. Untuk ***normal mode*** seperti yang telah kita lakukan seringkali disebut dengan ***sloppy mode***. Dengan ***strict mode*** pembuat kode *javascript* dengan ***sloppy mode*** bisa diminimalisir, jika kita baru belajar menggunakan bahasa *javascript* sangat disarankan dalam ***strict mode***.

Legacy Code

Jika anda hendak menggunakan sebuah *legacy code* hati hati menggunakan *strict mode* sebab bisa membuat kode menjadi tidak berjalan. *By the way* sudah taukah apa itu ***legacy code***?

Sebuah *term* gaul dunia *developer* yang artinya sebuah kode yang sudah tidak lagi dikembangkan dan dipelihara (*no longer maintenanced*) biasanya kode yang sudah sangat lama sebelum saat versi *javascript* masih jadul dimana terdapat beberapa fitur *javascript* yang sudah *deprecated* dan di *remove* pada *javascript* terbaru.

Perbedaanya dengan *normal mode* atau *sloppy mode* adalah dalam *strict mode* dalam pembuatan variabel kita harus mendeklarasikanya secara eksplisit. Dalam ***sloppy mode*** jika kita melakukan penetapan nilai pada variabel yang belum dideklarasikan akan dianggap sebagai *global variable*. Perhatikan pada gambar di bawah ini :

```
» function sloppyFunction() {
    sloppyVariable = 777;
  }
← undefined
» sloppyFunction()
← undefined
» console.log(sloppyVariable)
← undefined
  777
```

Gambar 140 Sloppy Mode

Pada gambar di atas kita menggunakan *normal mode* atau *sloppy mode* dimana kita bisa tiba tiba saja menetapkan nilai pada variabel `sloppyVariable` tanpa melakukan deklarasi *variable* terlebih dahulu baik itu menggunkan ***keyword*** `var` atau `let`**.** Dalam ***normal mode*** atau ***sloppy mode*** hal ini lazim dan bisa dilakukan tetapi tidak bagi ***strict mode*** sebab tidak aman alias ***unsafe***.

Tapi perhatikan pada gambar di bawah ini jika kita menggunakan `sloppyVariable` dalam keadaan ***strict mode*** hasilnya adalah ***error***.

Hal inilah yang akan terjadi jika kita menggunakan ***legacy code*** dalam keadaan ***strict mode*** boleh jadi ada 1 diantara ribuan atau ratusan baris terdapat `sloppyvariable`.

```
» function strictFunction() {
    'use strict';
    sloppyVar = 77;
  }
← undefined
» strictFunction()
⛔ ▶ ReferenceError: assignment to undeclared variable sloppyVar
»
```

Gambar 141 Strict Mode

Untuk *sample code* penggunaan *strict* dalam *node.js* anda bisa melihat kode di bawah ini :

```
// Javascript kode akan dieksekusi dengan mode strict
'use strict';

// Error akan terjadi karena x belum dideklarasikan
x = 3.14; //ReferenceError: x is not defined

// Use Strict di deklarasikan didalam sebuah function,
// Strict dalam sekup lokal
myFunction();
function myFunction() {
  ('use strict');
  y = 3.14; //Error. ReferenceError: y is not defined
}
```

**Link* sumber kode.

Notes
*Kita bisa menggunakan strict mode dengan memberikan statement "**use strict**" dibaris pertama sebelum kita menulis kode javascript.*

5. *Automatic Add Semicolon*

Dalam ***node.js*** jika kita lupa memberikan simbol ; di akhir sebuah *statement* maka dibelakang layar simbol tersebut akan ditambahkan secara otomatis. Dengan begitu *interpreter* dapat mengenali instruksi yang diberikan.

Hal ini sangat membantu jika anda sebagai *programmer* lupa memberikan tanda titik koma.

```
var A = 10
var B = 20
console.log(A)
console.log(B)
var C = 40;
var D = 60;
console.log(C);
console.log(D);
```

**Link* sumber kode.

7. Variable Declaration

Variable

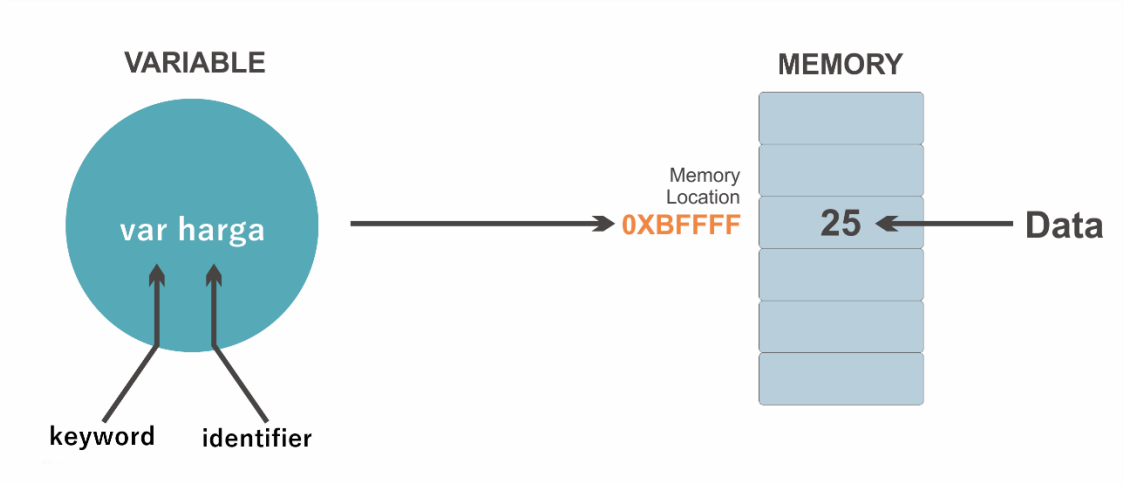
Setiap program pasti membutuhkan tempat untuk menyimpan suatu data. Data tersebut akan disimpan dilokasi memori tertentu. *RAM* (*Random Access Memory*) komputer terdiri dari jutaan sel memori. Ukuran dari setiap sel tersebut sebesar 1 *byte*.

Sebuah *RAM* komputer dengan ukuran 8 (*GigaBytes*) memiliki 8 x 1024MB= 8192 x 1024KB = 8.589.934.592 sel memori.

Saat *javascript engine* membaca *statement code* di bawah ini :

```
var harga = 25;
```

maka secara internal dapat di representasikan sebagai berikut :



Gambar 142 Variabel dalam memory

Variabel adalah sebuah nama yang nyaman & mudah diingat yang merepresentasikan alamat memori tempat suatu data tersimpan. Data pada alamat memori (*memory location*) tersebut dapat diubah (*manipulate*), *variable are the most basic form of data storage*.

Identifier

Variabel mempunyai nama yang kita sebut sebagai ***Identifier***.

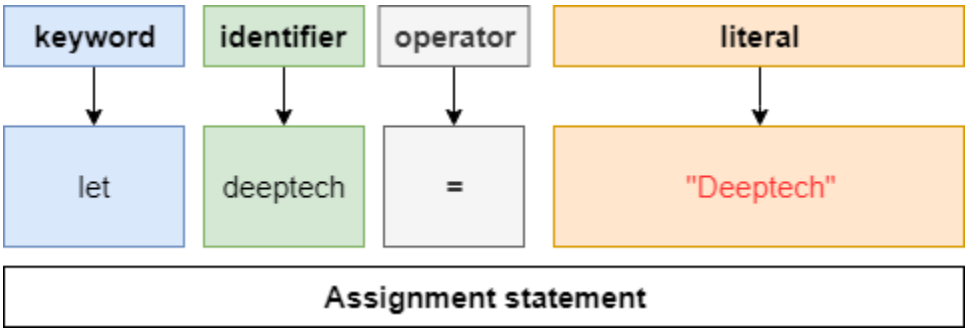
Setiap kali kita membuat *identifier* terdapat aturan dalam pembuatan namanya, tidak boleh menggunakan ***Reserved Keyword***. Pemberian *identifier* bersifat *case sensitive*.

Literal

Nilai yang hendak disimpan pada sebuah variabel disebut dengan ***Literal***. Di bawah ini adalah contoh deklarasi variabel lengkap dengan *reserved keyword, identifier* & *literal* :

```
let deeptech = "DeepTech"

//let <- reserved keyword
//deeptech <- identifier
//"DeepTech" <- literal (string literal)
```

**Link* sumber kode.



Gambar 143 Variable Declaration Syntax Rule

Binding

Proses pemberian *literal* pada variabel disebut dengan ***Binding*** atau ***Assignment Operation***. Pada contoh kode di atas kita melakukan ***binding string literal*** pada variabel dengan *identifier* **deeptech**.

Reserved Words

Dalam *javascript* terdapat 4 grup ***reserved words*** yang tidak boleh digunakan sebagai ***identifier*** diantaranya adalah : ***Keyword***, ***Future Reserved Keyword***, ***Null Literal***, dan ***Boolean Literals***.

Reserved word adalah sebuah ***Token*** yang memiliki makna dan dikenali oleh *javascript engine*. *Token* adalah serangkaian *character* yang membentuk kesatuan tunggal. Misal *keyword* `let` adalah susunan dari *character* l, e dan t.

Keywords

Di bawah ini adalah kumpulan ***token*** yang menjadi *javascript keywords* :

Table 9 Javascript Keyword

break	case	catch	class
const	continue	debugger	default
delete	do	else	export
extend	finnaly	for	function
if	import	in	instanceof
new	return	super	switch
this	throw	try	typeof
var	void	while	yield

Future Reserved Words

Di bawah ini adalah kumpulan ***token*** yang menjadi *javascript future reserved words*, sebuah *keyword* yang akan digunakan pada versi ***ECMAScript*** dimasa depan. Beberapa *reserved word*s hanya dapat digunakan dalam keadaan ***strict mode*** **(**diberi tanda ***)** :

Table 10 Future Reserved Keyword

abstract	implements*	interface*	let*
package*	private*	protected*	public*
static*	yield*		

Null Literal

Null literal adalah ***token*** yang tersusun dari karakter n, u, l, dan l membentuk literal null. Tidak boleh digunakan sebagai ***identifier***.

Boolean Literal

Boolean literal adalah dua ***token*** yang tersusun dari karakter t, r, u, e, dan f, a, l, s, e membentuk *literal* true dan membentuk *literal* false. Tidak boleh digunakan sebagai ***identifier***.

Notes
Jangan gunakan *Keyword* sebagai *Identifier*, *V8 engine* akan memproduksi *error*!

Naming Convention

Seperti yang telah kita fahami sebelumnya, *identifier* adalah nama yang menjadi pengenal pada suatu variabel. Penamaan *identifier* tidak boleh menggunakan *reserved words* yang telah disediakan *javascript*.

Di bawah ini adalah aturan dalam membuat *identifier* dalam *javascript* :

1. Tidak dapat diawali dengan angka misal `7deeptech`.
2. Dapat di awali dengan ***symbol*** $ misal `$deeptech`.
3. Dapat di awali dengan ***symbol*** _ misal `_deeptech`.
4. Dapat di awali dengan ***character*** dalam *unicode (*contoh π atau ö*)*. dilanjutkan dengan *symbol* $, _ angka, atau *character* lagi misal : `deep_tech` atau `deep$tech` atau `deeptech2024`
5. Secara ***naming convention*** gunakan ***camelCase*** untuk membuat *identifier*, misal `deepTech`, `kecilBesar` atau `gunGun` (cirinya kata pertama tanpa huruf kapital disambung kata kedua menggunakan huruf besar).

```
var deep$tech = "deep$tech"
console.log(deep$tech);

var deep_tech = "deep_tech"
console.log(deep_tech);

var deeptech2019 = "deeptech2019"
console.log(deeptech2019);

var namingConvention = "camelCase"
console.log(namingConvention);
```

**Link* sumber kode.

Di bawah ini adalah contoh penamaan *identifier* yang unik diizinkan selama dalam ruang lingkup ***unicode*** yang memiliki **65.534 *character*** lebih :

```
var π = Math.PI;
console.log(π);
```

```
var ಠ_ಠ = "ಠ_ಠ";
console.log(ಠ_ಠ);

var ლ_ಠ益ಠ_ლ = "ლ_ಠ益ಠ_ლ";
console.log(ლ_ಠ益ಠ_ლ);

var ᕙ( ͡° ͜ʖ ͡°)ᕗ = "ᕙ( ͡° ͜ʖ ͡°)ᕗ";
console.log(ᕙ( ͡° ͜ʖ ͡°)ᕗ);

// berbeda:
// ᕙ( ͡° ͜ʖ ͡°)ᕗ << ᕙ( ͡° ͜ʖ ͡°)ᕗ;

// Roman numerals
var Ⅳ = 4;
var Ⅴ = 5;
console.log(Ⅳ + Ⅴ);// 9
```

**Link* sumber kode.

Apa itu Unicode?

Javascript mendukung penamaan *identifier* menggunakan *unicode*. Apakah anda tahu apa itu *unicode*? ***Unicode*** adalah suatu standar industri yang dirancang untuk mengizinkan teks dan simbol dari semua sistem tulisan di dunia untuk dapat ditampilkan dan dimanipulasi secara konsisten oleh komputer.

Case Sensitivity

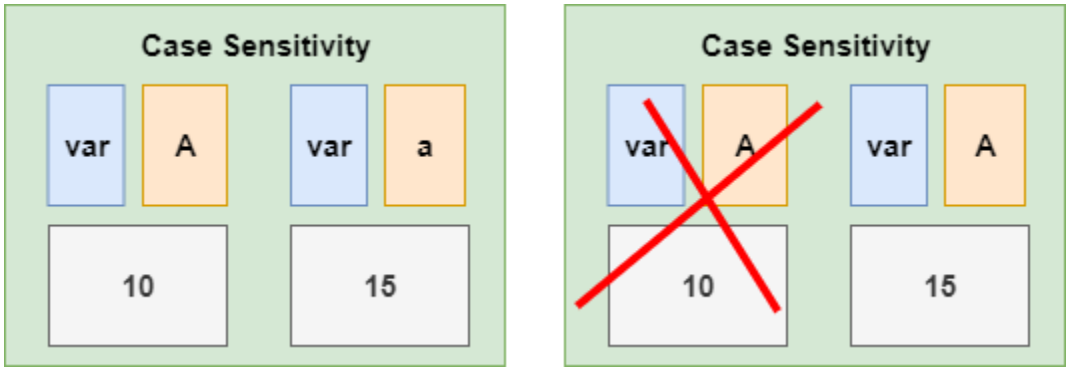
Javascript adalah bahasa pemograman yang bersifat *case sensitive*, jadi anda harus berhati-hati karena dua buah *variable* dengan nama pengenal (*identifier*) yang sama bisa memiliki nilai yang berbeda. Penyebabnya adalah huruf besar dan huruf kecil yang diberikan pada sebuah *identifier*.

```
var A = 10;
var a = 15;
console.log(A); //A Capital
console.log(a); //a small
```

Output :

```
10
15
```

**Link* sumber kode.



Gambar 144 Case Sensitivity Illustration

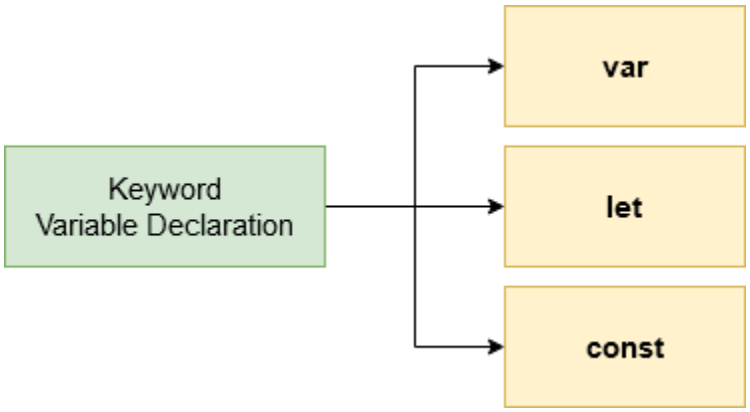
Ilustrasi gambar di atas menjelaskan jika *identifier* yang kita buat berbeda yaitu ***a*** dan ***A***, maka masing-masing akan memiliki nilai yang berbeda. Namun, jika kedua-duanya memiliki *identifier* yang sama yaitu ***A*** dan ***A*** maka *identifier* duplikat yang berada pada baris *statement* terakhir akan menimpa nilai dari *identifier* sebelumnya.

Anda bisa mencobanya dengan mengubah *identifier* ***a*** menjadi ***A***.

Loosely Typed Language

Javascript adalah ***Loosely Typed Language*** yang artinya kita tidak harus secara eksplisit menetapkan ***data types*** ketika membuat suatu *variable*. Jika sebelumnya anda telah mempelajari bahasa *C, C#* atau *Java* maka jika kita ingin membuat sebuah *variable* maka kita harus menentukan *data type* yang dimilikinya. Sehingga disebut dengan ***Strongly Typed Language***.

Berdasarkan ***ECMAScript*** terbaru Ada 3 tipe deklarasi dalam *javascript* yaitu :



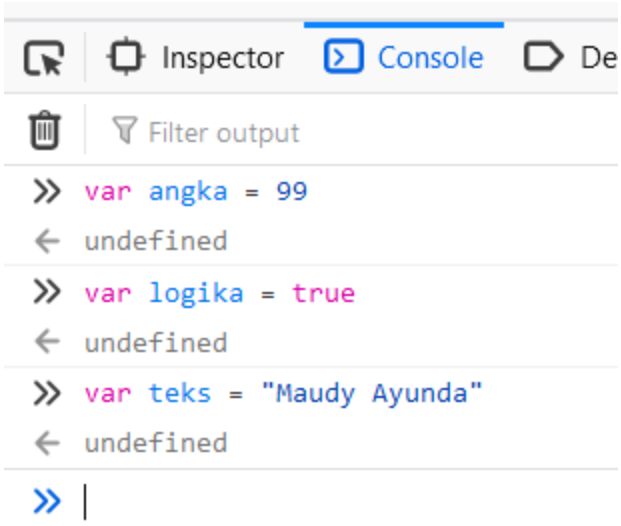
Gambar 145 Variable Declaration Keyword

Var Keyword

Keyword yang digunakan untuk mendeklarasi *variable* semenjak *javascript interpreter* dibuat.

Di bawah ini adalah contoh *variable declaration* menggunakan ***reserved keyword*** **var**.

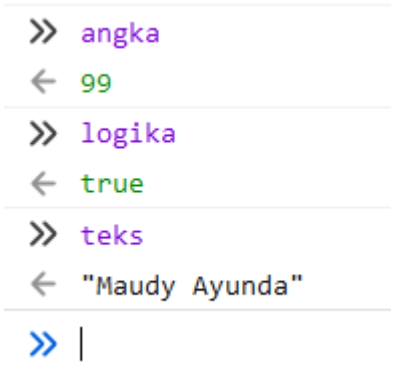
Buka kembali *web console* dengan menekan tombol **CTRL+SHIFT+K** :



Gambar 146 Deklarasi Variable

Terdapat 3 variabel dengan *identifier* **`angka`**, **`logika`** dan **`teks`**, masing-masing memiliki *literal* yaitu **`99`**, **`true`** dan **`“hi maudy ayunda”`**.

Untuk menampilkan *literal* dari ketiga *identifier* tersebut cukup ketik kembali kemudian tekan *enter* seperti pada gambar di bawah ini :

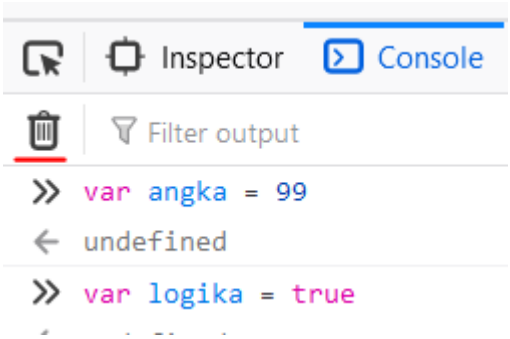


Gambar 147 Cetak Nilai *Variable*

Jangan khawatir saat kita melakukan deklarasi variabel ketika kita menekan *enter* terdapat *output* dengan keterangan ***undefined***, maksudnya adalah saat kita melakukan deklarasi *variable* tidak terdapat ***return***.

Namun saat kita memasukan variabel yang telah kita buat kemudian menekan *enter* maka nilai dari variabel tersebut akan muncul seperti pada gambar di atas.

Untuk melakukan *reset web console* klik gambar tong sampah yang diberi garis warna merah :



Gambar 148 Reset Web Console

Let Keyword

Deklarasi *variable* versi *optimized*.

Dalam deklarasi variabel terdapat istilah ***Variable Scoope***, jika kita melakukan deklarasi variabel diluar sebuah *function* maka *variable* tersebut berada dalam ***scoope global*** yang bisa diakses dimana saja. Lalu saat kita melakukan deklarasi variabel di dalam sebuah *function* maka *variable* tersebut berada dalam ***scoope local*** yang hanya bisa diakses di dalam *function* itu sendiri.

Javascript sebelum ECMA 6 tidak mengenal *block statement scoope* yaitu variabel yang hanya dikenali di dalam *block of code* saja (*scoope local*). Sehingga diciptakanlah **`let`**. Dalam *javascript* variabel yang dideklarasikan diluar *block* bersifat *global* sementara yang berada di dalam sebuah *scoope* bersifat *local*.

Sebagai contoh pada gambar di bawah ini *variable* **x** tetap bisa dibaca diluar *block* karena deklarasinya menggunakan *reserved keyword* **`var`**.

Silahkan buka ***multiline-editor*** :

```
1 if (true) {
2   var x = 5;
3 }
4 console.log(x)
5 x
```

Gambar 149 Var Keyword

Tulis kode di atas kemudian tekan **CTRL+Enter** untuk melihat hasilnya.

Solusinya kita dapat menggunakan *keyword* `let` untuk membuat *local scope variabel* :

```
1  var x = 8;
2  // Here x is 8
3  {
4    let x = 5;
5    // Here x is 5
6    console.log(x);//
7  }
8
9  // Here x is 8
10 console.log(x); // 8
```

Gambar 150 Block scope variable

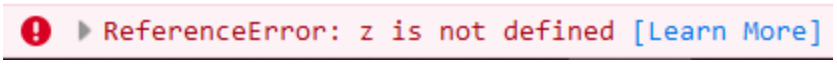
Tulis kode di atas kemudian tekan **CTRL+Enter** untuk melihat hasilnya.

Jika kita menggunakan ***keyword*** `let` yang di desain hanya untuk deklarasi variabel *local scoope* maka jika kita mencoba mengaksesnya dari luar *scoope* maka hasilnya akan *error*. Kita akan membuktikanya, tulislah kode di bawah ini :

```
1 if (true) {
2   let z = 5;
3 }
4 console.log(z)
```

Gambar 151 Let Keyword Scoope

Eksekusi kode di atas dengan cara menekan **CTRL+Enter** untuk melihat hasilnya.



Gambar 152 Error

Constant Keyword

Deklarasi konstanta atau ***read-only variable***

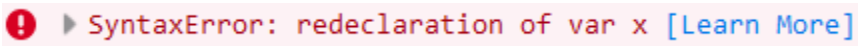
Pada *Javascript* juga terdapat *constant*, sebuah *variable* yang tidak bisa diubah karena hanya untuk dibaca saja (*Read only*). Sebuah *Constant* harus dimulai dengan abjad, *underscore* atau *dollar sign* selanjutnya boleh memiliki lagi konten *alphabetic, numeric* dan *underscore character*.

Di bawah ini adalah contoh deklarasi *constant* dalam *Javascript* :

```
1 const x = 9
2 x = 4
```

Gambar 153 Const Declaration

Pada kode di atas kita akan mendapatkan sebuah *error* ketika dieksekusi karena, sebuah *Constant* tidak dapat lagi diberi nilai atau di deklarasi ulang lagi.



Gambar 154 Redecleration variable x

`const` memiliki karakteristik yang dengan **`let`** yaitu termasuk kedalam *block scoped* hanya saja nilainya tidak dapat diubah.

```
1  var x = 8;
2 // Here x is 8
3 {
4   const x = 5;
5   // Here x is 5
6 }
7 console.log(x)
8 // Here x is 8
```

Gambar 155 Constant Declaration

8. *Clean Code Variable Declaration*

Avoid Global Variable

Penggunakaan *global variable* harus diminimalisir sebab berpotensi tertimpa oleh *script* lainnya.

Declaration on Top

Salah satu *practice* untuk membuat *clean code* adalah penempatan untuk melakukan deklarasi variable, usahakan lokasinya ada di awal penulisan kode.

```
var firstName, lastName, price, discount, fullPrice;

firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

**Link* sumber kode.

Initialize Variable

Pastikan setiap kali anda membuat sebuah variabel inisialisasi terlebih dahulu dengan sebuah nilai untuk mencegah *undefined values* dan salah satu *practice* untuk membuat *clean code*.

```
var firstName = "",
  lastName = "",
  price = 0,
```

```
  discount = 0,
  fullPrice = 0,
  myArray = [],
  myObject = {};
```

**Link* sumber kode.

Use Const or Let

Gunakan **`const`** untuk mencegah *re-assignment* pada *variable* yang bisa menimbulkan *bugs*.

```
// bad
var a = 1;
var b = 2;

// good
const a = 1;
const b = 2;
```

Saran ini mengikuti konvensi standar *Ecma Script Linter* yang memiliki pengaturan *prefer-const* & *no-const-assign*. Jika kita ingin agar *variable* yang kita miliki dapat melakukan operasi *re-assignment* gunakan **`let`** daripada **`var`**, karena let memiliki karakteristik *block scoped* :

```
// bad
var count = 1;
if (true) {
  count += 1;
}

// good, use the let.
```

```
let count = 1;
if (true) {
  count += 1;
}
```

**Link* sumber kode.

Saran ini mengikuti konvensi standar *Ecma Script Linter* yang memiliki pengaturan *no-var*.

Subchapter 2 – Data Types

Without data, you're just another person with an opinion.
— W. Edward Deming

Subchapter 2 – Objectives

- Memahami Apa itu ***Data***?
- Memahami Apa itu ***Types***?
- Memahami Apa itu ***Data Types***?
- Memahami Apa itu ***Strongly Typed Language***?
- Memahami Apa itu ***Dynamically Typed Language***?
- Memahami Apa itu ***Pointer***?
- Memahami Apa itu ***Stack*** & ***Heap***?
- Memahami Apa itu ***Numeric Data Types***?
- Memahami Apa itu ***String Data Types***?
- Memahami Apa itu ***Boolean Data Types***?
- Memahami Apa itu ***Null & Undefined Data Types***?
- Memahami Apa itu ***Symbol Data Types***?

1. *Javascript Data Types*

Untuk memahami konsep ***Data Types*** secara sempurna dalam *javascript* ada beberapa konsep fundamental yang harus kita pelajari. Di antaranya memahami konsep *data*, konsep *types*, konsep lainya seperti *generic variable, pointer, stack & heap.*

Apa itu Data?

Data dalam komputer secara *digital electronics* direpresentasikan dalam wujud ***Binary Digits (bits),*** sebuah unit informasi (*unit of information*) terkecil dalam mesin komputer. Setiap *bit* dapat menyimpan satu nilai dari ***binary number*** yaitu **0 atau 1**, sekumpulan *bit* membentuk konstruksi ***Digital Data***. Jika terdapat ***8 bits*** yang dihimpun maka akan membentuk ***Binary Term*** atau ***Byte***.

Pada *level byte* sudah membentuk unit penyimpanan (*unit of storage*) yang dapat menyimpan *single character*. Satu **data *byte*** dapat menyimpan 1 *character* contoh : 'A' atau 'x' atau '$'.

Serangkaian *byte* dapat digunakan untuk membuat ***Binary Files***, pada *binary files* terdapat serangkaian *bytes* yang dibuat untuk diinterpretasikan lebih dari sekedar *character* atau *text*.

Pada *level* yang lebih tinggi (*kilobyte, megabyte, gigabyte* & *terabyte*) kumpulan *bits* ini dapat digunakan untuk merepresentasikan teks, gambar, suara dan video.

Data dalam konteks pemrograman adalah sekumpulan *bit* yang merepresentasikan suatu informasi.

Apa itu Types?

Types adalah sekumpulan nilai yang dikenali oleh *compiler* atau *interpreter* dan mengatur bagaimana data harus digunakan.

Sebuah *types* menentukan :

1. Data seperti apa saja yang dapat disimpan?
2. Seberapa besar memori yang dibutuhkan untuk menyimpan data?
3. Operasi apa saja yang dilakukan pada data tersebut?

Ada *types integer* terdiri dari angka positif dan negatif, *types boolean* terdiri dari benar (*true*) atau salah (*false*) dan *types string* terdiri dari kumpulan karakter (*character*).

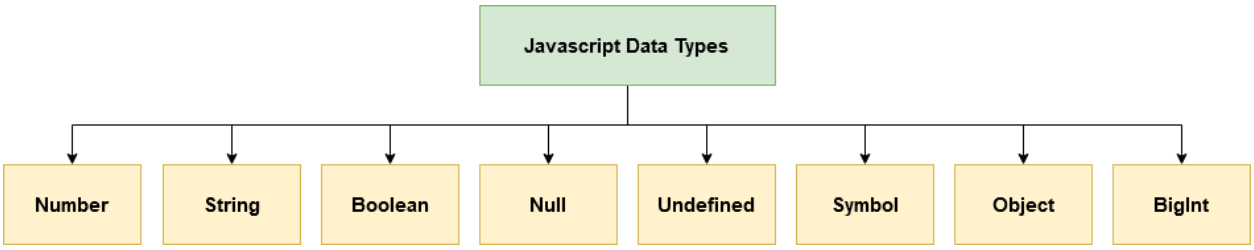
Dalam *javascript* terdapat tiga tipe data dasar yaitu *number, string* dan *boolean*. Sebuah variabel dalam *javascript* memiliki *data type* yang spesifik.

Seperti yang telah dijelaskan sebelumnya *javascript* dikenal dengan *Loosely Typed Language* dikarenakan hanya mampu membuat ***Generic Variable.***

Apa itu Generic Variable?

Apa itu *Generic Variable*? Sebuah variabel yang bisa kita buat tanpa harus menetapkan *data type* secara eksplisit.

Javascript Data Types



Gambar 156 Javascript Data Type

Meskipun begitu *Generic Variable* dalam *Javascript* memiliki 7 *Data Types* jika mengacu pada spesifikasi *ECMAScript* terbaru diantaranya adalah :

1. ***Number***, digunakan untuk merepresentasikan tipe data numerik seperti angka (tanpa *quote*). Misal `42,` `0.5` dan `2e-16`
2. ***String***, digunakan untuk merepresentasikan tipe data tekstual berupa serangkaian *character* dalam *quote* atau *single quote*. Misal `“hello”` / `‘hello’`
3. ***Boolean***, digunakan untuk merepresentasikan sebuah logika. Misal `TRUE` atau `FALSE`
4. ***Null***, sebuah *keyword* yang berarti tidak memiliki nilai
5. ***Undefined,*** sebuah *variable* yang nilainya belum didefinisikan.
6. ***Symbol***, sebuah *data types* baru yang muncul pada *ECMAScript* Tahun 2015 lalu.
7. ***Object***, terdiri dari *function*, *array*, *date*, *regexp* dan sebagainya
8. ***BigInt,*** digunakan untuk merepresentasikan tipe data numerik melebihi *safe integer*.

Apa itu Pointer?

Untuk mengenal representasi *data types* secara ***low-level*** dalam *javascript* kita perlu memahami terlebih dahulu apa itu ***pointer***. Seperti yang telah kita pelajari sebelumnya variabel adalah tempat untuk menyimpan *data*. Setiap variabel memiliki representasi berupa alamat memori (*memory address*).

Pointer adalah sebuah variabel yang nilainya adalah alamat memori suatu variabel. Jika sebelumnya kita telah belajar bahasa pemrograman C, kita dapat membuat sebuah *pointer* untuk menyimpan *memory address* dari sebuah variabel dan mengaksesnya kembali untuk mendapatkan *memory address* atau nilai variabel dari *memory address* tersebut. Contoh di C :

```
#include <stdio.h>
int main () {
   int  var = 26;   /* Deklarasi variabel */
   int  *ip;        /* Deklarasi pointer variabel */
   ip = &var;  /* simpan memory address dari var ke pointer
                  variable*/
   printf("Alamat memori dari var variable: %x\n", &var  );
   /* memory address disimpan di pointer variable */
   printf("Alamat memori disimpan dalam ip variable: %x\n",
ip );
   /* akses nilai yang digunakan dalam pointer */
   printf("Nilai dari *ip variable: %d\n", *ip );
   return 0;
}
/*
Alamat memori dari var variable: bffd8b3c
Alamat memori disimpan dalam ip variable: bffd8b3c
Nilai dari *ip variable: 26
*/
```

**Link* sumber kode.

Apa itu Stack & Heap?

Stack digunakan untuk membuat alokasi ***Static Memory*** dan ***Heap*** digunakan untuk membuat ***Dynamic Memory***, keduanya disimpan di dalam ***RAM (Random Access Memory)***. Variabel yang dialokasikan dalam *stack* disimpan secara langsung di dalam memori dan akses pada *static memory* sangat cepat. Variabel yang dialokasikan dalam *heap* memiliki memori yang dialokasikan saat ***run time*** dan akses pada *dynamic memory* cenderung lamban.

Apa itu Primitive & Reference Values?

Di dalam *javascript* terdapat dua *system types* yaitu *primitive types* atau *reference types*. *Primitive types* disimpan sebagai *data types* sederhana. *Reference types* disimpan sebagai *object* yang menjadi referensi sebuah lokasi dimemori.

Javascript inventor, menciptakan bahasa *javascript* dengan kaidah yang unik yaitu *primitive types* diperlakukan seperti *reference types* tujuanya untuk membuat bahasa *javascript* menjadi lebih konsisten.

Saat bahasa pemrograman lainya membedakan antara *primitive types* dan *reference types*, dengan menyimpan *primitive* dalam *stack* dan *references* dalam *heap*. *Javascript* menghilangkan konsep ini.

Secara *under the hood*, *Javascript* melacak variabel dan sekumpulan variabel yang berada dalam *scope* tertentu dengan sebuah ***Variable Object (VO)*** yang dikemudian hari pada dokumentasi *EcmaScript* 5 disebut sebagai ***Lexical Environment***.

Primitive Values disimpan secara langsung di dalam *variable object* & ***Reference Values*** ditempatkan sebagai *pointer* di dalam *Variable Object (VO),* menjadi referensi lokasi dimemori termpat *object* disimpan.

Primitive Types

Seperti yang telah di bahas sebelumnya, *javascript* memiliki 8 *Data Types* dan 7 diantaranya disebut dengan ***primitive*** *atau primitive value.*

1. *String,*
2. *Number,*
3. *Boolean,*
4. *Null*
5. *Undefined*
6. *BigInt*
7. *Symbol,*

Istilah *primitive* digunakan karena hanya menyimpan satu nilai tunggal, data bukan sebuah *object* dan tidak memiliki *method*. Di bawah ini adalah *primitive types* :

```
// strings
var name = "Gun Gun Febrianza";

// numbers
var age = 25;
var price = 1.51;

// boolean
var isExist = true;

// null
var object = null;
```

```
// undefined
var flag = undefined;
var ref; // setara dengan kode di atas
```

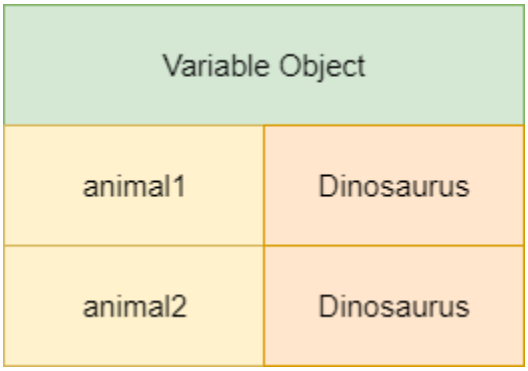
**Link* sumber kode.

Khusus untuk *symbol* anda akan mempelajarinya pada kajian khusus *Symbol Data Types*.

Pada *javascript*, ketika variabel hendak menyimpan *literal* berupa *primitive value* maka variabel tersebut menyimpan nilai secara langsung. Jika kita membuat variabel dengan nilai yang berasal dari variabel yang lain, masing-masing akan mendapatkan salinanya. Sebagai contoh :

```
var animal1 = "dinosaurus";
var animal2 = animal1;
```

Pada kode di atas variabel `animal1` menyimpan *string literal*, kemudian variabel `animal2` melakukan *binding* dengan nilai yang dimiliki oleh variabel `animal1`.



Gambar 157 Variable Object

Meskipun `animal1` dan `animal2` memiliki nilai yang sama, masing masing memiliki nilai secara terpisah. Jika kita mengubah nilai pada variabel `animal1` maka nilai pada variabel `animal2` tidak akan berubah. Perhatikan kode di bawah ini :

```
var animal1 = "dinosaurus";
var animal2 = animal1;

console.log(animal1); //dinosaurus
console.log(animal2); //dinosaurus

animal1 = "godzila"
console.log(animal1); //godzila
console.log(animal2); //dinosaurus
```

**Link* sumber kode.

Dalam *javascript* sebuah *primitive type* bersifat ***immutable***.

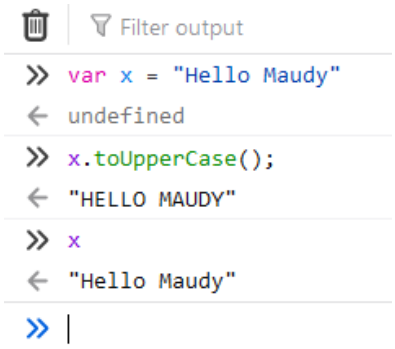
Apa itu Immutable?

Sebagai contoh jika terdapat variabel x dengan nilai *string* "`hello`", maka nilainya akan selalu *string* "`hello`". Sebagai contoh ketika seseorang mengubah string "hello" menjadi huruf besar semua dengan menggunakan *method* `toUpperCase()` menjadi "`HELLO`". Sebagian dari mereka masih mengira maka nilai x sudah menjadi huruf besar semua, padahal tidak.

```
var x = "hello"
console.log(x.toUpperCase()); // HELLO
console.log(x); // hello
```

**Link* sumber kode.

Ini terbukti ketika kita memanggil kembali variabel x, nilai yang dimilikinya tidak berubah. Perhatikan gambar di bawah ini :



Gambar 158 Immutable Behaviour

Meskipun begitu bukan berarti nilai sebuah variabel tidak bisa diubah, kita bisa mengubahnya dengan melakukan ***rebinding*** melalui menetapkan operasi *assignment* :

Gambar 159 Assignment Operation

Reference Types

Dalam *javascript*, *reference types* direpresentasikan dengan sebuah *object*. Sebuah *object* berbeda dengan *primitive, object* bisa memiliki wujud dan nilai yang berbeda-beda. Sebuah *object* mampu menyimpan berbagai nilai secara (*hetererogenous*).

Object bisa menyimpan seluruh nilai yang dimiliki oleh *primitive types*. Sifat fleksibel ini membuat *object* dapat digunakan untuk membangun sebuah *custom data type*.

Ketika kita berinteraksi dengan *web browser* menggunakan *javascript* kita akan berkenalan dengan ***built-in object***, sekumpulan *object* bawaan dari *web browser* yang bisa kita gunakan untuk mempermudah memecahkan masalah dalam bahasa pemrograman. Diantaranya adalah *objects* :

1. *Array*
2. *Date*
3. *RegExp*
4. *Map* dan *WeakMap*
5. *Set* dan *WeakSet*

Built-in Object tersebut juga tersedia dalam *node.js*, di bawah ini adalah contoh *object* sederhana yang memiliki *properties* dan *method* dalam bahasa pemrograman *javascript* :

```
let Gun = {
  name: "GGF",
  ucapSalam: function () {
    alert("Hello World!!");
  }
};
Gun.ucapSalam(); // Hello World!
```

**Link* sumber kode.

Pada kode di atas kita membuat *object* bernama **Gun** yang memiliki *properties* `name` dan ***method*** `ucapSalam`, ***properties*** `name` di isi dengan nilai *string* dan *properties* `ucapSalam` di isi dengan sebuah *function*.

Dalam *javascript* sebuah *function* juga bisa diperlakukan sebagai *object*.

Pada kasus di atas *object* `Gun` dapat menyimpan sebuah *function*. Secara *low level system* sebuah *object* mempunyai kapasitas penggunaan *memory* lebih besar dari *primitive*. Ini dikarenakan kemampuanya untuk bisa menampung banyak *data properties* dan *method*.

Primitive as Object via Object Wrapper

Ada saatnya kita hanya memerlukan *primitive type* saja untuk mengolah data yang proporsional. Terlebih penggunaan *primitive type* lebih cepat dan ringan.

Namun ada saatnya kita ingin memanipulasi nilai suatu *primitive type*. Misal terdapat variabel x yang menyimpan tipe data primitif *string*, jika kita ingin mengubah nilai yang ada di dalam variabel tersebut kita harus melakukan deklarasi ulang.

Misal mengubahnya menjadi *string* dengan huruf kapital semua. Tentu daripada sekedar melakukan deklarasi ulang maka ada konsep yang lebih efisien yaitu menggunakan *object wrapper*.

Dengan *object wrapper* sebuah *primitive string* bisa **diberi akses untuk mengakses kemampuan super** yaitu *method*. Pada *primitive type* sebuah *object wrapper* akan disediakan untuk sementara waktu agar bisa digunakan. Jika penggunaan sudah selesai *object wrapper* tersebut akan kembali dihapus.

Di bawah ini adalah contoh kode agar anda bisa membayangkanya :

```
let str = "Hello";
console.log(str.toUpperCase()); // HELLO
```

**Link* sumber kode.

Pada kode di atas kita memberikan *primitive type* `str` sebuah *object wrapper* yaitu `.toUpperCase()` untuk mengubah nilai *string* menjadi huruf kapital semua. Setelah fungsi `console.log()` dipanggil *object wrapper* akan dihapus kembali. Sehingga keaslianya sebagai *primitive type* tetap terjaga.

2. Data Types Conversion

Javascript selain dikenal dengan sebutan ***Loosely Typed Language*** juga dikenal dengan sebutan ***Dynamically Typed Language*** artinya untuk melakukan konversi *data types* pada *javascript* kita tinggal mengubah nilai suatu *variable* saja. Sebagai contoh :

Gambar 160 Data Types Conversion

Dynamic Typed

Pada gambar di atas pertama kita membuat *variable* dengan *identifier* **angka**, selanjutnya melakukan *assign statement* (penetapan nilai) dengan nilai 99 yang artinya *variable* tersebut merupakan sebuah *data type number*.

Misalkan jika ingin mengganti *data type* **angka** ke dalam *string* cukup melakukan *re-assign statements* (penetapan nilai ulang) kembali dengan nilai sebuah *string*, maka *variable* angka adalah sebuah *data types string*.

String To Number

Untuk menerjemahkan *string* ke dalam *number* tersedia beberapa fungsi yang bisa kita gunakan. Di bawah ini adalah contoh penerapan konversi *data types string* ke dalam *number* :

```
>> var maudy = parseInt("99")
<- undefined
>> maudy
<- 99
>> maudy+1
<- 100
```

Gambar 161 String to Number Conversion

String To Decimal Number

Pada gambar di atas ***function*** **`parseInt()`** digunakan untuk mengubah *string* **"99"** menjadi *number* 99. Selain **`parseInt()`** terdapat juga fungsi **`parseFloat()`** untuk mengubah *string* berupa pecahan kedalam *number*. Misal :

```
>> var maudy = parseFloat("9.88")
<- undefined
>> maudy
<- 9.88
```

Gambar 162 String Float to Number Conversion

Number to String

Kita juga dapat melakukan konversi *number integer* ke dalam *string* :

```
>> var a = 10
<- undefined
>> var b = String(a)
<- undefined
>> b
<- "10"
```

Gambar 163 Number to String Conversion

Decimal Number to String

Kita juga dapat melakukan konversi *decimal number* ke dalam *string* :

```
>> var d = String(c)
<- undefined
>> d
<- "23.2"
```

Gambar 164 Decimal Number to String

Boolean to String

Kita juga dapat melakukan konversi *boolean* ke dalam *string* :

```
>> var x = true
<- undefined
>> var y = String(x)
<- undefined
>> y
<- "true"
```

Gambar 165 Boolean to String

Check Data Type

Ah masa sih? Kita buktikan dengan ***function typeof()*** yang digunakan untuk mengetahui data *types* suatu *variable* :

```
>> typeof x
<- "string"
```

Gambar 166 Typeof Function

Di bawah ini adalah *table* jika kita melakukan operasi ***typeof()*** :

Table 11 Typeof Result

Value	***Result***
Undefined	“undefined”
Null	“null”
Boolean	“boolean”
Number	“number”
String	“string”
Object	“object”

3. *Number Data Types*

Dalam *javascript*, *number* adalah sebuah *primitive type* yang digunakan untuk mengekspresikan sebuah data numerik.

Menurut *ECMAScript standard*, secara internal di dalam *browser engine* satu buah *number* disimpan dalam format 64 ***bit floating point*** **(IEEE 754)** atau biasa disebut 64 ***bit double precision***. Acuan ini mengacu kepada *ECMAScript standard*.

Sebagai tambahan kenapa *javascript* dapat merepresentasikan *floating-point numbers*, maka ada 3 *symbolic value* yang dimiliki *number*:

+Infinity, -Infinity, dan ***NaN (not-a-number).***

Berbicara *number* dalam komputer ada yang bisa direpresentasikan secara akurat atau hampir mendekati akurat (*approximately*). *Number* 6, 66, dan 30,000,000 adalah *number* yang bisa direpresentasikan secara akurat, sementara πtidak tidak bisa direpresentasikan secara akurat.

Dalam *algebra, symbol* πtermasuk kedalam *irrational number*. Berbeda dengan bahasa pemrograman lainya *javascript* hanya memiliki satu *numeric type* yaitu *number* yang bisa diekspresikan sebagai ***decimal*** atau tanpa *decimal*.

Untuk membuat *variable* yang menyimpan nilai *floating point* cukup beri *decimal point* dan 1 *digit* sesudahnya misalkan 99.9. Untuk membuat *variable* yang menyimpan nilai *integer* cukup dengan *number literal* tanpa *decimal point* misal 99.

Gambar 167 Javascript Number

Infinity

Infinity merepresentasikan *mathematical Infinity* dengan notasi ∞. Dalam *javascript* ini adalah nilai spesial yang nilainya selalu paling besar. Untuk mendapatkanya dalam *javascript* kita bisa mengekskusi kode sebagai berikut :

```
// Angka yg dibagi 0 (zero) memproduksi Infinity:
var x = 2 / 0; // x memproduksi Infinity
var y = -2 / 0; // y memproduksi -Infinity
console.log(x);
console.log(y);
```

**Link* sumber kode.

Kita juga dapat melakukan perulangan sampai memperoleh bilangan *infinity* :

```
var myNumber = 2;
while (myNumber != Infinity) {
  // Execute until Infinity
  console.log((myNumber = myNumber * myNumber));
}
/* Output :
```

```
4
16
256
65536
4294967296
18446744073709552000
3.402823669209385e+38
1.1579208923731162e+77
1.3407807929942597e+154
Infinity */
```

**Link* sumber kode.

Untuk memeriksa angka yang diberikan tidak *infinity* kita dapat menggunakan *method* **`isFinite()`** yang dimiliki oleh *object* **`Number`** :

```
function div(x) {
  if (Number.isFinite(1000 / x)) {
    return "Number is NOT Infinity.";
  }
  return "Number is Infinity!";
}

console.log(div(0));
// expected output: "Number is Infinity!"

console.log(div(1));
// expected output: "Number is NOT Infinity."
```

**Link* sumber kode.

NaN

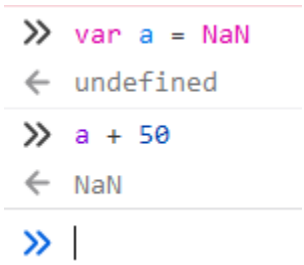
Sebuah *number* bisa menyimpan seluruh kemungkinan nilai *number* termasuk nilai *special* seperti ***Not-a-Number*** (*NaN*), ***positive infinity*** dan ***negative infinity***. Seorang *mathematicians* menyatakan bahwa *infinity is not a number*. Memang bukan, begitu juga dengan NaN. Ini bukan *number* yang bisa digunakan untuk melakukan *computation*. Lalu apa sih *NaN* itu?

Dalam *javascript NaN* adalah nilai spesial yang dihasilkan (*returned*) ketika operasi atau fungsi matematika yang tidak wajar dilakukan sehingga menimbulkan *computation error*.

Gambar 168 NaN Result

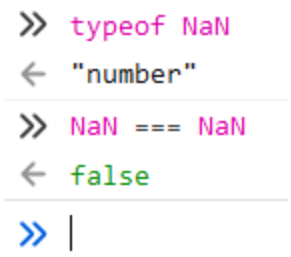
Pada kode di atas hasilnya **NaN** karena kita mencoba membagi *number* dengan *string* sehingga memproduksi *computation error*.

Operasi aritmetika pada **NaN** akan selalu memproduksi **NaN**:



Gambar 169 NaN Characteristic

Meskipun begitu *javascript engine* memperlakukan **NaN** *(Not a Number)* sebagai tipe data *number*, dengan karakteristiknya yang aneh.



Gambar 170 NaN Check Equality

Maximum & Minimum Value

Object Number memiliki *property* **MAX_VALUE** yang bisa diberi nilai maksimum 1.79E+308. Jika nilai lebih dari **MAX_VALUE** maka akan direpresentasikan sebagai *infinity,* perhatikan gambar di bawah ini :

```
function multiply(x, y) {
  if (x * y > Number.MAX_VALUE) {
    return 'Process as Infinity';
  }
  return x * y;
}

console.log(multiply(1.7976931348623157e308, 1));
// expected output: 1.7976931348623157e+308

console.log(multiply(1.7976931348623157e308, 2));
// expected output: "Process as Infinity"
```

**Link* sumber kode.

MIN_VALUE yang dimiliki oleh *number* adalah -1.79E+308.

Max Safe Integer

Object Number juga memiliki ***property*** `MAX_SAFE_INTEGER` ***constant*** yang merepresentasikan nilai maksimum *safe integer* dalam *JavaScript* yaitu (2^{53} - 1).

Property `MAX_SAFE_INTEGER` *constant* memiliki nilai maksimum sebesar 9007199254740991 (9,007,199,254,740,991 atau sekitar ~9 *quadrillion*).

Safe dalam konteks ini adalah kemampuan untuk merepresentasikan *integer* dengan benar dan tepat. *Javascript* dapat menampilkan *numbers* secara aman dengan *range* -(2^{53} - 1) dan 2^{53} - 1.

```
var x = Number.MAX_SAFE_INTEGER + 1;
var y = Number.MAX_SAFE_INTEGER + 2;

console.log(Number.MAX_SAFE_INTEGER);
// expected output: 9007199254740991

console.log(x);
// expected output: 9007199254740992

console.log(x === y);
// expected output: true
```

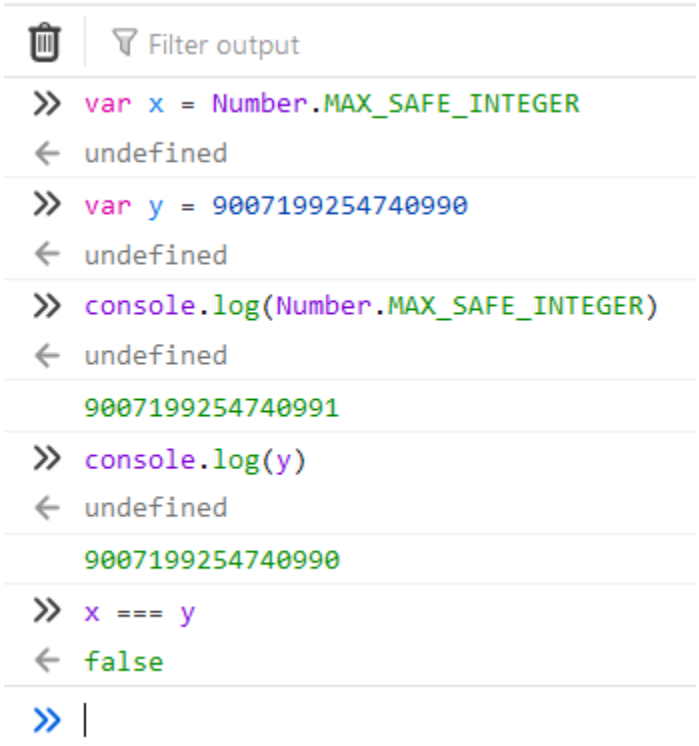
**Link* sumber kode.

Pada gambar di atas kita melihat terdapat operasi :

`Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2`

Statement di atas akan dievaluasi dan mendapatkan nilai *true*, secara matematika ini tidak benar (*incorrect*). Hal ini terjadi karena penjumlahan dilakukan dengan salah satu nilai yang sudah tidak *safe number* lagi.

Pada gambar di bawah ini, operasi dilakukan didalam ranah *safe integer* sehingga bisa melakukan pembandingan matematis secara akurat :



Gambar 171 Max Safe Integer Operation

Safe Integer Checking

Object Number memiliki *method* **`isSafeInteger()`** untuk memeriksa apakah *number* yang diberikan termasuk *safe integer* atau tidak.

```
Number.isSafeInteger(3);                    // true
Number.isSafeInteger(Math.pow(2, 53));      // false
Number.isSafeInteger(Math.pow(2, 53) - 1);  // true
Number.isSafeInteger(NaN);                  // false
Number.isSafeInteger(Infinity);             // false
Number.isSafeInteger('3');                  // false
Number.isSafeInteger(3.1);                  // false
Number.isSafeInteger(3.0);                  // true
```

```
```

*Link sumber kode.

Jika ingin mengelola nilai yang lebih kecil atau lebih besar dari ***~9 quadrillion*** secara presisi kita perlu menggunakan *arbitrary precision arithmetic library* seperti bignumber.js

Positive e Notation

Untuk menghindari kesalahan penulisan *number*, apalagi jika terdapat angka nol dan angka lainya yang sangat panjang. Untuk mengatasi hal ini kita bisa mempersingkatnya dengan menggunakan ***e notation*** :

```
let billion = 1e9; //1 billion (1 & 9 zero)
//Contoh lainya :
// 1e3 = 1*1000
// 1.23e6 = 1.23 * 1000000
```

*Link sumber kode.

Negative e Notation

Di bawah ini contoh untuk menampilkan *negative e notation* :

```
//explicit zero
let ms = 0.000001;
//implicit zero
ms = 1e-6;

//1e-3 = 1 / 1000
//1.23e-6 = 1.23 / 1000000
```

*Link sumber kode.

Rounding

Dalam *javacript number* kita tidak akan pernah lepas dari aktivitas ***rounding*** atau dalam bahasa indonesia disebutnya pembulatan. *Javascript* menyediakan *math object* untuk melakukan pembulatan, ada berbagai *method* yang bisa digunakan diantaranya adalah:

1. ***Math.floor***

   Pembulatan ke bawah (*Rounds down*):

   Angka 3.1 menjadi 3, dan angka -1.1 menjadi -2.

2. ***Math.ceil***

   Pembulatan ke atas (*Rounds up*):

   Angka 3.1 menjadi 4, dan angka -1.1 menjadi -1.

3. ***Math.round***

   Pembulatan ke angka terdekat (*Rounds to the nearest integer*):

   Angka 3.1 menjadi 3, angka 3.6 menjadi 4 dan angka -1.1 menjadi -1.

4. ***Math.trunc*** **(tidak didukung oleh *Internet Explorer*)**

   Pembulatan dengan cara menghapus seluruh nilai setelah *decimal point*:

   Angka 3.1 menjadi 3, angka -1.1 menjadi -1.

Tabel *Rounding Comparison*

Literal	***Math.floor***	***Math.ceil***	***Math.round***	***Math.trunc***
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Precision

Dalam *number object* terdapat *method* **`toPrecision()`** yang bisa kita gunakan untuk menentukan presisi sebuah *number*.

```
>> var num = new Number(27.123456);
<- undefined
>> num.toPrecision()
<- "27.123456"
>> num.toPrecision(1)
<- "3e+1"
>> num.toPrecision(2)
<- "27"
>> num.toPrecision(3)
<- "27.1"
>> num.toPrecision(4)
<- "27.12"
>> num.toPrecision(5)
<- "27.123"
>> num.toPrecision(6)
<- "27.1235"
>> num.toPrecision(7)
<- "27.12346"
>> num.toPrecision(8)
<- "27.123456"
>> num.toPrecision(9)
<- "27.1234560"
>> 3e+1
<- 30
```

Gambar 172 Precision

Exponentiation

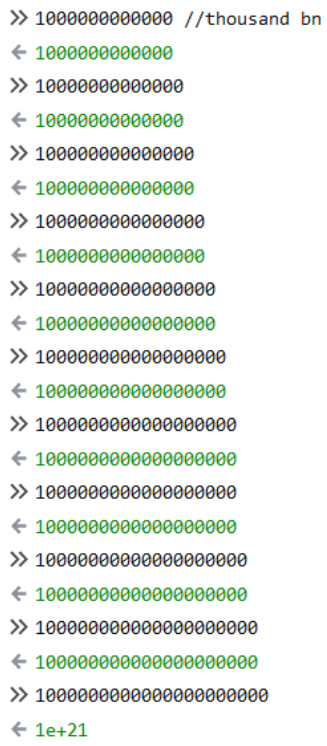
Jika kita ingin membuat *number* menampilkan notasi eksponensial, kita bisa menggunakan *method* **`toExponential()`** dan *e notation* seperti pada gambar di bawah ini :

```
>> var num = 1225.30
<- undefined
>> num.toExponential()
<- "1.2253e+3"
>> num.toExponential(1)
<- "1.2e+3"
>> num.toExponential(2)
<- "1.23e+3"
>> num.toExponential(4)
<- "1.2253e+3"
>> num.toExponential(5)
<- "1.22530e+3"
>> 1.2e+3
<- 1200
>> 1.23e+3
<- 1230
>> 1.2253e+3
<- 1225.3
>> 1.22530e+3
<- 1225.3
```

Gambar 173 Exponentiation

e Notation Trigger

Notasi e dalam *javascript* akan otomatis dibuat ketika jumlah digit sudah lebih dari 20 *digits*. Di bawah ini adalah kode sampel untuk *trigger* notasi e :



Gambar 174 Automated e Notation

Gambar 175 20 & 21 Digit

Jika *number* terlalu besar dieksekusi maka akan terjadi *overflow*, berpotensi mendapatkan *infinity*.

```
>> 1e500
<- Infinity
>>  0.1 + 0.2 == 0.3
<- false
```

Gambar 176 Infinity Result

Number Accuration

Akurasi ***integer*** dalam *javascript* adalah ***15 digits***. Jika lebih dari itu terdapat *problem loss of precision*. *Javascript* tidak memperlakukan nilai yang sudah tidak akurat atau lebih 15 digit sebagai *error*. Di bawah ini adalah contoh uji keakuratan *javascript*, ketika kita membuat angka 9 dengan total 16 digit maka hasilnya menjadi tidak akurat :

```
>>  var x = 999999999999999;   // x will be 999999999999999
    var y = 9999999999999999;  // y will be 10000000000000000
<- undefined
>> x
<- 999999999999999
>> y
<- 10000000000000000
```

Gambar 177 e Notation Trigger

Di bawah ini adalah representasi internal *64 bit format IEEE-754* dalam *javascript*, ada 64 *bits* yang dapat digunakan untuk menyimpan *number*. 52 *bits* digunakan untuk menyimpan *digits*, 11 *digits* digunakan untuk menyimpan posisi *decimal point*, dan 1 bit digunakan untuk memberikan ***sign*** positif atau ***sign*** negatif.

Namun secara *internal* sering kali 52 *bits* ini tidak cukup sehingga ada *digits* yang hilang :

Value (aka Fraction/Mantissa)	Exponent	Sign
52 bits (0 - 51)	11 bits (52 - 62)	1 bit (63)

Gambar 178 Number Representation

Maksimum jumlah *digits* ada angka desimal adalah 17 *digits*, namun *floating point arithmetic* tidak selalu menghasilkan penjumlahan yang 100% akurat:

Gambar 179 Wrong Addition Result

Imprecise Calculation

Kenapa operasi penjumlahan 0.2 + 0.1 sebelumnya tidak akurat? Jawabanya karena *number* disimpan dalam wujud biner dalam memori (serangkaian angka 1 dan nol). Pecahan seperti 0.1 dan 0.2 terlihat sederhana dalam sistem bilangan desimal, namun sebernarnya pecahan tersebut adalah representasi **pecahan yang tidak berujung (*unending fractions*)** dalam wujud binernya.

Analoginya adalah, Pecahan 0.1 adalah bentuk lain dari 1 dibagi 10 atau notasi pecahanya 1/10. Dalam sistem bilangan desimal angka ini mudah sekali direpresentasikan. Namun jika dibandingkan dengan 1/3 maka hasilnya adalah **pecahan yang tidak berujung (*endless fraction*)** 0.33333(3)...n

Pembagian dengan angka 10 berjalan dengan baik dalam sistem bilangan desimal namun tidak saat kita membaginya dengan angka 3. Alasan inilah yang menyebabkan sistem bilangan biner menjamin kepastian pembagian dengan angka 2, namun jika pembagian dengan angka 10 maka akan menghasilkan pecahan biner yang tidak berujung (*endless binary fraction*).[binary]

Artinya tidak ada jalan untuk menyimpan atau menulis angka 0.1 atau angka 0.2 menggunakan sistem biner, sama seperti angka 1 dibagi 3 dalam sistem bilangan desimal tidak ada cara untuk menulis dan mengetahui seluruh jumlahnya secara akurat.

IEEE-754 menyelesaikan permasalahan ini dengan cara melakukan pembulatan (*rounding*) ke angka terdekat (*nearest possible number*). Pembulatan ini dilakukan dibelakang layar sehingga kita tidak mengetahui ada presisi kecil yang hilang.

```
>> 0.1.toFixed(20)
<- "0.10000000000000000555"
```

Gambar 180 Tiny Precision Loss

Itulah alasan kenapa ketika kita menjumlahkan 01 + 02 hasilnya bukan 0.3

Ini isu menarik terkenal dengan sebutan *What Every Computer Scientist Should Know About Floating-Point Arithmetic*.

Solution to Imprecise

Untuk mengatasi permasalahan ketidakakuratan, kita perlu mengubah *number* ke dalam integer untuk melakukan operasi matematika. Ini dapat digunakan dengan cara melakukan operasi perkalian antara 0.1*10 dan 0.2*10=2 , kedua angka telah menjadi *integer* sehingga tidak terjadi permasalahan *loss precision*. Di bawah ini adalah precision.js yang bisa anda gunakan untuk menghadapi permasalahan *loss precision* :

https://gist.github.com/gungunfebrianza/c63617d7afe2559faf5844e55da8970e

Di bawah ini adalah bukti operasi penjumlahan, pengurangan, pembagian dan perkalian menjadi akurat :

High-precision calculation for javascript,You can call add,sub,mul,div to calculate you variables.

precision.js

Gambar 181 Precision.js Result

Fixed Number

Format *number* dengan membuat *digit* yang spesifik. *Method* di bawah ini menghasilkan *return* berupa *string* :

```
>> 0.1 + 0.2
<- 0.30000000000000004
>> (0.1 + 0.2).toFixed(2)
<- "0.30"
```

Gambar 182 toFixed() Method

Jika ingin mengubahnya kedalam *number* gunakan *unary plus* :

```
>> let sum = 0.1 + 0.2;
<- undefined
>> +sum.toFixed(2)
<- 0.3
>> typeof +sum.toFixed(2)
<- "number"
>> typeof sum.toFixed(2)
<- "string"
```

Gambar 183 Coerce String to Number

Numeric Conversion

Di bawah ini adalah ***method*** `parseInt()` yang dapat kita gunakan untuk mengkonversi *string* ke dalam *integer* :

```
parseInt('100'); //= 100
parseInt('2019@marketkoin.com'); //= 2018
parseInt('marketkoin@2019'); //= NaN
parseInt('3.14'); //= 3
parseInt('21 7 2018'); //= 21
parseInt('100', 10); //= 100
parseInt('8', 8); //= NaN
parseInt('15', 8); //= 13
parseInt('16', 16); //= 22
parseInt(' 100 '); //= 100
parseInt('0x16'); //= 22
parseInt('10'); //= 10
parseInt('10.33'); //= 10
parseInt('10 20 30'); //= 10
parseInt('10 tahun'); //= 10
parseInt('tahun ke 10'); //= NaN
```

**Link* sumber kode.

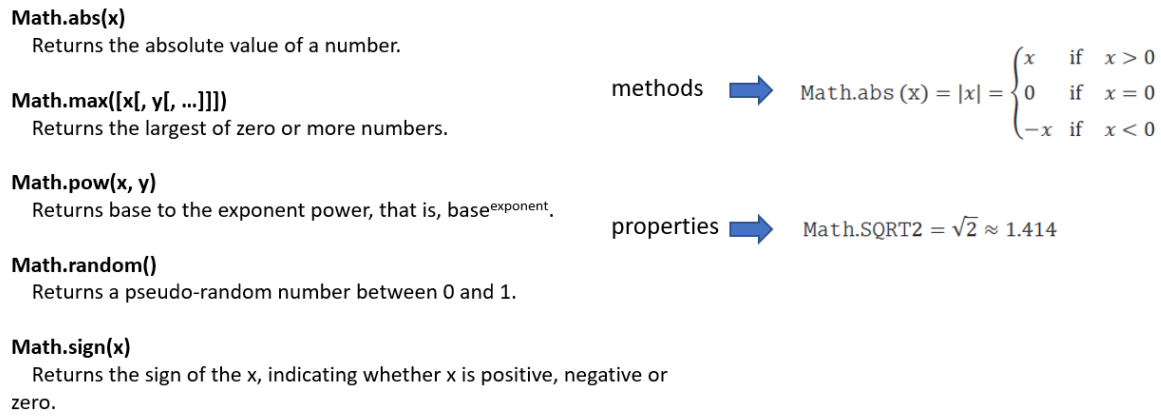
Di bawah ini adalah ***method*** `parsefloat()` yang dapat kita gunakan untuk mengkonversi *string* ke dalam *float* :

```
parseFloat("10");        // returns 10
parseFloat("10.33");     // returns 10.33
parseFloat("10 20 30");  // returns 10
parseFloat("10 tahun marketkoin");  // returns 10
parseFloat("tahun ke 10");  // returns NaN
```

**Link* sumber kode.

Math Object

Math adalah *object* bawaan yang telah disediakan di dalam *browser* atau *javascript engine node.js*. *Math object* memiliki *properties* dan *methods* untuk melakukan operasi matematika :



Gambar 184 Math Objects

Hexadecimal, Binary dan Octadecimal

Jika kita ingin menggunakan *hexadecimal* gunakan *prefix* **0x** :

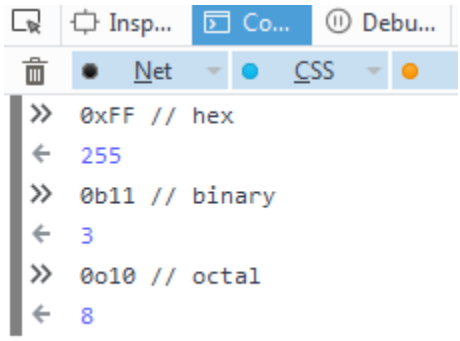
Misal `alert(0xff);` **//hasilnya adalah 255**

Jika kita ingin menggunakan *binary* gunakan ***prefix*** `0b` jika ingin menggunakan *octal* gunakan ***prefix*** `0o` :

```
    let a = 0b11111111; // binary form dari 255
    let b = 0o377; // octal form dari 255

    console.log(a == b); // hasilnya true, karena nilainya setara
atau sama.
```

**Link* sumber kode.

Dalam *javascript* juga terdapat ***numeral system*** lainya yang didukung yang akan kita pelajari selanjutnya. Semenjak *ECMAScript* 6, kita sudah bisa menulis *integer* dengan notasi *binary* dan *octal*. Pada *binary* diawali dengan *prefix* 0b dan untuk *octal* 0o.



Gambar 185 New binary and octal notation

4. *String Data Types*

Sebuah *string* adalah sebuah data teks, kata *string* sendiri berasal dari kata "*string of character*" kata ini digunakan pada abad 19 di akhir tahun 1800 oleh para *typesetter* yang kemudian didukung para matematikawan untuk merepresentasikan serangkaian simbol.

Dalam *javascript* sebuah *string* adalah serangkaian *Unicode* karakter yang membentuk suatu kesatuan, *unicode* sendiri adalah *computing industry standard* untuk merepresentasikan data teks termasuk setiap karakter atau simbol yang difahami oleh manusia seperti kanji hingga ke emoji.

Setiap karakter dalam *unicode* memerlukan memori penyimpanan sebesar 16 bit. Setiap karakter bisa diakses melalui posisi *index* yang dimilikinya, *index* pada karakter pertama dimulai dengan nol.

Sebuah *string literal* bisa menggunakan :

1. ***double quotation*** mark (" ... "),
2. ***single quotation mark*** (' ... ') atau
3. *backticks*.

Double Quote String

Kita dapat mendeklarasikan *literal string* menggunakan *double quote* :

```
» let hi = "hi maudy"
← undefined
» hi
← "hi maudy"
»
```

Gambar 186 Javascript String

Single Quote String

Kita juga dapat membuat *variable string* dengan *literal string* menggunakan *single quote* :

```
>> let gun = 'hi gun'
<- undefined
>> gun
<- "hi gun"
>>
```

Gambar 187 Single Quote String

Jika kita perhatikan pada *output variable* **gun**, *literal* yang ditampilkan memiliki *double quote*.

String Concatenation

Jika kita menjumlahkan *literal number* dalam bentuk *string*, maka akan memproduksi *string concatenation* :

```
>> let x = "5"
<- undefined
>> let y = "15"
<- undefined
>> x + y
<- "515"
```

Gambar 188 String Concatenation

Begitu juga jika *variable number* dijumlahkan dengan *variable string* hasilnya adalah *string concatenation* :

```
» var a = 5
← undefined
» var b = "15"
← undefined
» a + b
← "515"
```

Gambar 189 Number Addtion String

Numeric String Characteristic

Jika kita mencoba melakukan operasi aritmetika pada *literal string* yang nilainya *numeric* maka akan berhasil :

```
» var a = "100"
← undefined
» var b = "10"
← undefined
» var c = a / b
← undefined
» c
← 10
» typeof c
← "number"
»
```

Gambar 190 String Division

Begitupun pada operasi perkalian dan pengurangan juga berhasil :

```
» a * b
← 1000
» a - b
← 90
»
```

Gambar 191 Multiplication & Subtraction String

Namun pada penjumlahan akan memproduksi *string concatenation* :

```
» a + b
← "10010"
```

Gambar 192 String Addition

Escaping

Javascript juga memiliki *escape sequences* yang bisa kita gunakan pada REPL, pertama kita bisa membuat *newline* :

```
Net  CSS  JS  Security
» console.log("Ayunda \n Faza \n Maudya ")
  Ayunda
   Faza
   Maudya
← undefined
```

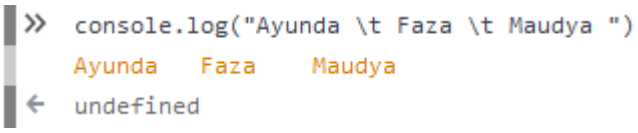
Gambar 193 Newline Character

Di bawah ini *escape sequences* untuk membuat *single* atau *double quatation mark character* :

```
» console.log("Ayunda \" Faza \" Maudya ")
  Ayunda " Faza " Maudya
← undefined
» console.log("Ayunda \' Faza \' Maudya ")
  Ayunda ' Faza ' Maudya
← undefined
```

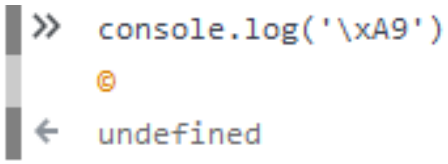
Gambar 194 Single and Double Quotation Mark Character

Di bawah ini *escape sequences* untuk membuat tab :



Gambar 195 Tab Character

Kita juga bisa menggunakan *special character* dan *unicode* literal :



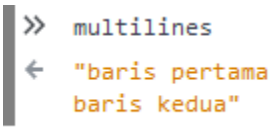
Gambar 196 Copyright Character

Template String

Untuk melakukan *backticks* :

```
const multilines = `baris pertama
baris kedua`;
```

**Link* sumber kode.



Gambar 197 Contoh Backticks

Dengan *template string* kita juga dapat mengeksekusi *expression statement* di dalam *string* :

```
>> var z = `Result ${2+2*10}`
<- undefined
>> z
<- "Result 22"
>>
```

Gambar 198 Expression Inside String

Untuk menampilkan *expression* & *variable* dalam *template string* gunakan notasi **${…}** :

```
>> var m = 60
<- undefined
>> console.log(`result ${m}`)
<- undefined
   result 60
>> console.log(`result m`)
<- undefined
   result m
```

Gambar 199 Variable Inside String

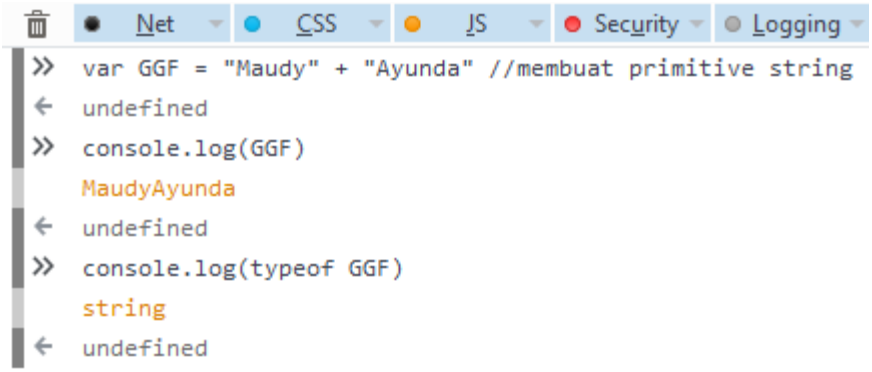
String Objects & Primitives

Membuat *string object* dalam *javascript* :

```
Net  CSS  JS  Security  Logging  Server
>> var GGF = new String("Ayunda Faza Maudya") //membuat string object
<- undefined
>> console.log(GGF)
   String { "Ayunda Faza Maudya", 19 more… }
<- undefined
>> console.log(typeof GGF)
   object
<- undefined
```

Gambar 200 String Object on Javascript

Membuat *primitive string* dalam *javascript* :

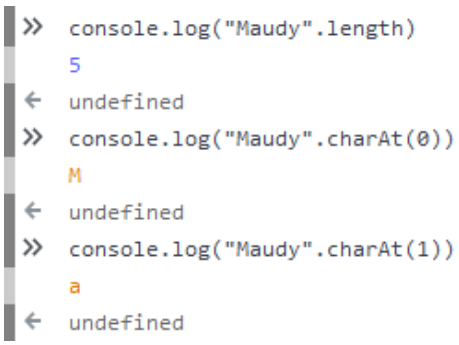


Gambar 201 Primitive String on Javascript

Setiap tipe data *primitive* seperti *string, boolean, number* di dalam bahasa *javascript* masing-masing memiliki *objects* yang *equivalent*. Sangat disarankan untuk tidak menggunakan *string object*.

String Function

Pada gambar di bawah ini kita dapat mengetahui jumlah karakter menggunakan *properties length* dan menggunakan *function charAt* untuk mengetahui karakter berdasarkan *index*.



Gambar 202 Properties Length & charAt Function

Guna dari *Function* **`indexOf`** untuk membaca *index* berdasarkan karakter, **`startWith`** untuk membaca karakter awal dan **`endsWith`** membaca karakter akhir dengan *return* berupa *boolean* (*true or false*). *Function* **`includes`** untuk memastikan karakter tersebut terdapat pada *string*.

```
» console.log("Maudy".indexOf("u"))
  2
← undefined
» console.log("Maudy".startsWith("M"))
  true
← undefined
» console.log("Maudy".endsWith("y"))
  true
← undefined
» console.log("Maudy".includes("d"))
  true
← undefined
```

Gambar 203 String Functions

Selain itu juga terdapat *function* **`split`** untuk memisahkan *string* kedalam bentuk *array*, **`toUpperCase`** untuk mengubah kehuruf kapital dan **`toLowerCase`** untuk mengubah kehuruf kecil. Bisa kita lihat pada gambar di bawah ini :

```
» console.log("Maudy Ayunda".split(" "))
  Array [ "Maudy", "Ayunda" ]
← undefined
» console.log("Maudy Ayunda".split(""))
  Array [ "M", "a", "u", "d", "y", " ", "A", "y", "u", "n", 2 more… ]
← undefined
» console.log("Maudy Ayunda".toUpperCase())
  MAUDY AYUNDA
← undefined
» console.log("Maudy Ayunda".toLowerCase())
  maudy ayunda
← undefined
```

Gambar 204 String Functions

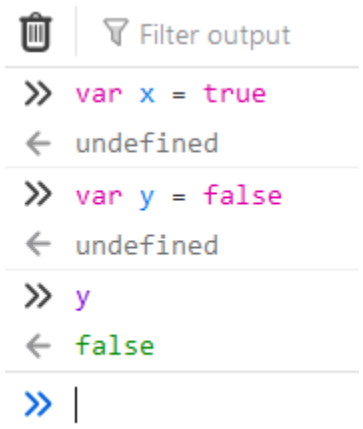
Di dalam *javascript* juga bisa melakukan *string interpolation* seperti dalam *python* menggunakan *syntax* di bawah ini :

```
» var x = 2, y = 3
← undefined
» console.log("Jumlah dari x * y adalah : " + (x*y))
  Jumlah dari x * y adalah : 6
← undefined
```

Gambar 205 String Interpolation

5. *Booleans Data Types*

Di dalam *javascript* sebuah *primitive boolean* hanya memiliki dua nilai yaitu *True* dan *False Keywords*. Kita bisa membuat *primitive boolean* dengan memberinya nilai *true* atau *false* pada sebuah variabel seperti pada gambar di bawah ini :



Gambar 206 Javascript Booleans

Seluruh nilai di bawah ini akan menjadi *false* ketika dikonversi (*coerced*) ke dalam *boolean* :

1. *NaN*
2. *null*
3. *undefined*
4. ''
5. 0 & -0
6. *false*

Sebagai catatan selain *undefined* atau *null* termasuk *boolean object* yang nilainya **`false`** akan di evaluasi sebagai *true* jika digunakan di dalam *multiconditonal if else statement* :

```
const isExcuted1 = new Boolean(false)
```

```
if(isExcuted1) {
  console.log('isExecuted1 executed!');
} else {
  console.log('isExecuted1 not executed!');
}
// output : isExecuted1 executed!
```

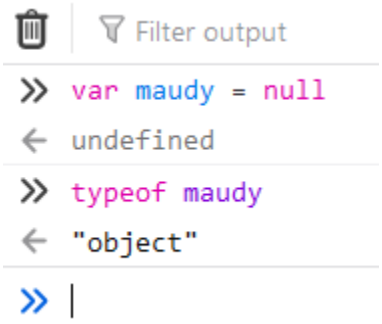
Namun jika kita memberikan nilai *primitive* **`false`** ketika di evaluasi dalam *multiconditional if else statement* maka akan menghasilkan nilai **`false`** :

```
const isExcuted2 = false

if (isExcuted2) {
  console.log('isExecuted2 executed!');
} else {
  console.log('isExecuted2 not executed!');
}
// output : isExecuted2 not executed!
```

**Link* sumber kode.

6. *Null Data Types*

Dalam *Javascript null* artinya *nothing* yaitu sesuatu yang tidak ada, meskipun begitu dalam *javascript null* adalah sebuah *object*.



Gambar 207 Javascript Null

Sebelumnya kita mengetahui dalam kajian *javascript data type* bahwa *null* adalah *primitive* kenapa disini adalah sebuah *object*?

Hal yang menarik karakteristik ini sudah diprogram dalam *interpreter* semenjak *javascript* pertama kali dikembangkan. Inilah alasan kenapa *javascript* disebut sebagai *the world's most misunderstood programming language* oleh Douglas Crockford.

```
// This stands since the beginning of JavaScript
typeof null === 'object';
```

Sebuah proposal untuk memperbaiki permasalahan ini sempat diajukan untuk ECMAScript namun ditolak (detailnya dapat dilihat disini), agar ketika kita memeriksa tipe dari null adalah **`'null'`** bukan **`'object'`** seperti :

```
typeof null === 'null';
```

Dalam series buku You Don't Know JS dikatakan bahwa ini adalah sebuah *bug* yang sepertinya tidak akan pernah dibetulkan, sebab banyak *applications* bergantung pada *bug* tersebut, membetulkanya akan menimbulkan lebih banyak *bug* baru.

> *This is a long-standing bug in JS, but one that is likely never going to be fixed. Too much code on the Web relies on the bug and thus fixing it would cause a lot more bugs!*

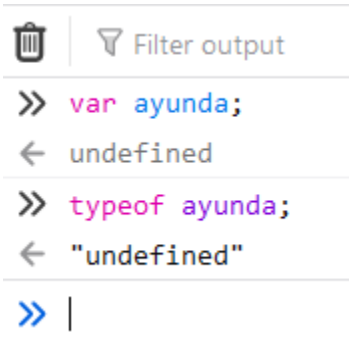
Begitulah ceritanya, namun jika anda masih memaksa bertanya kenapa harus *object*?

Jawabanya karena spesifikasi dalam *ECMAScript* telah mengaturnya harus demikian :

http://www.ecma-international.org/ecma-262/5.1/#sec-11.4.3

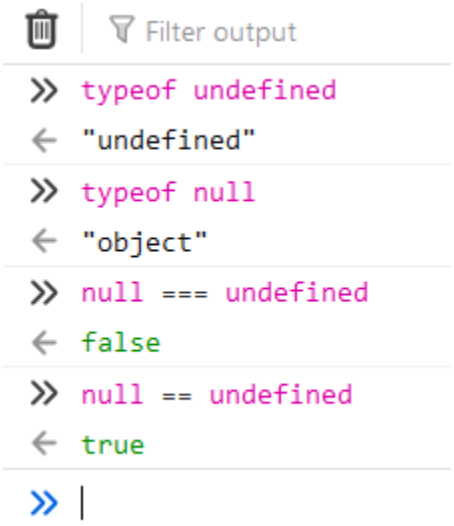
7. *Undefined Data Types*

Dalam *Javascript* sebuah *variable* yang tidak memiliki nilai secara otomatis memiliki *Data Types Undefined*.



Gambar 208 Javascript undefined

Perbedaan antara *null* dan *undefined* adalah *null variable* harus dideklarasikan secara eksplisit sementara *undefined* hanya ada pada *variable* yang tidak dideklarasikan :

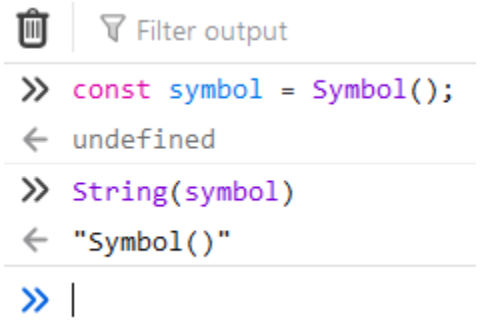


Gambar 209 Differences Null and Undefined

8. *Symbol Data Types*

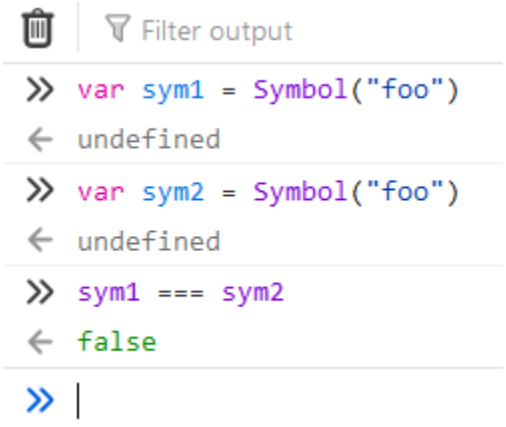
Symbol adalah *Data types* baru dalam *ECMAScript* 6 yang merepresentasikan *token* unik dan memiliki karakteristik *immutable*.

Pada gambar di bawah ini kita membuat *variable constant* dengan *identifier symbol,* setelah itu melakukan operasi *assignment* memberinya nilai *return* dari *symbol function. Return* yang dihasilkan berupa *id unique*.



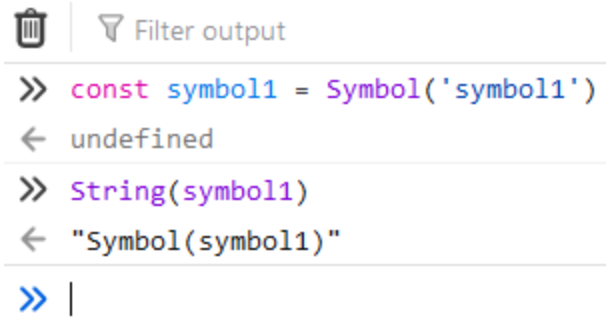
Gambar 210 Javascript Symbol

Kenapa *unique*? Karena setiap *symbol* menghasilkan identitas berbeda (yang tidak bisa kita lihat) kita buktikan dengan kode pada gambar di bawah ini :



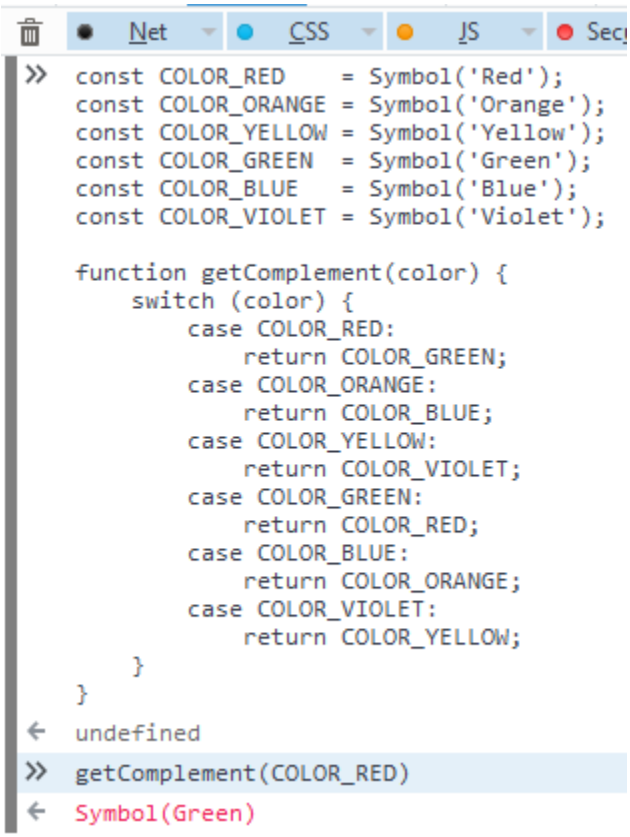
Gambar 211 Symbols Equality Test

Untuk memudahkan pembacaan (*Code Readability*) kita bisa menggunakan *parameter* yang dimiliki *symbol function,* kita bisa melakukanya seperti pada gambar di bawah ini:



Gambar 212 Parameter Symbol Function

Agak sedikit kompleks untuk dicerna memang, karena penggunaanya memang untuk yang sudah *advance* namun tidak ada salahnya kan kenalan dahulu biar bisa persiapan kedepanya. Implementasi selanjutnya penggunaan *symbol* bisa kita pelajari pada gambar di bawah ini :



Gambar 213 Sample Symbol Implementation

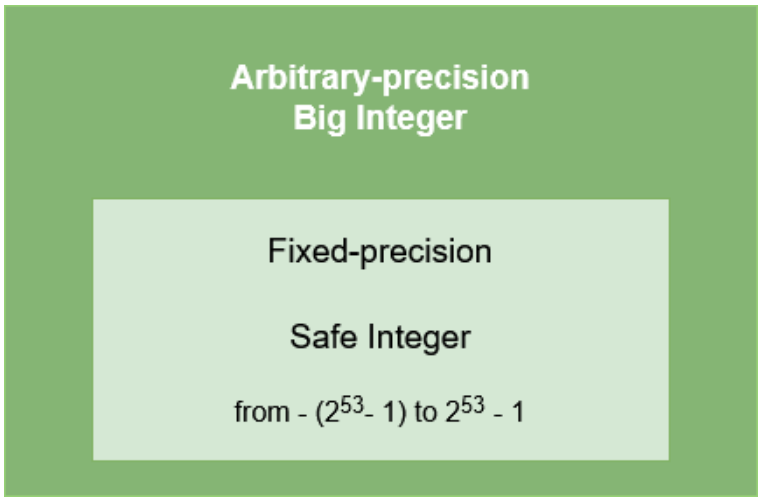
9. *BigInt Data Types*

Dalam *javascript*, *BigInt* atau *Big Integer* seringkali disebut dengan *arbitrary-precision integer*. Apa sih yang dimaksud dengan ***Arbitrary-precision***?

Sebelumnya kita mengetahui bahwa *javascript* dapat menampilkan *numbers* secara aman dengan *range* -(2^{53} - 1) dan 2^{53} – 1 karena *number* secara internal disimpan dalam format 64 ***bit floating point.*** Batasan *range* antara -(2^{53} - 1) dan 2^{53} – 1 menegaskan bahwa *number* bersifat ***fixed-precision*** *type*.

Arbitrary Precision

Arbitrary-precision adalah sifat yang dimiliki oleh *Big Integer* agar kita dapat menyimpan dan melakukan komputasi *integer* melebihi *safe limit* yang dimiliki oleh *integer number javascript*.



Gambar 214 Arbitrary & fixed Precision

Untuk mendeklarasikan variabel dengan tipe data *big integer* perhatikan kode di bawah ini :

```
const bigInt1 = 12345678901234567890123456789012345678890n;
const bigInt2 = BigInt("12345678901234567890123456789012345678890");
```

```
const bigInt3 = BigInt(12345678901234567890123456789012345678901234567890);
```

Pada variable **bigInt1** kita dapat menyimpan *integer number* yang diakhiri *character* **n**, simbol **n** menegaskan bahwa nilai yang disimpan secara implisit ditujukan untuk membuat *big integer*.

Pada variable **bigInt2** kita dapat mengkonversi *string* yang merepresentasikan *integer number* menjadi sebuah *big integer*.

Pada variable **bigInt3** kita dapat mengkonversi *integer number* menjadi sebuah *big integer*.

Big Integer adalah pritimitif *type* berbasis *numeric* dalam *javascript* untuk merepresentasikan *integer* dengan *arbitrary-precision*.

```
console.log(typeof 123); // 'number'
console.log(typeof 123n); // 'bigint'
```

Kode di bawah ini terdapat maksimum *safe integer* yang dimiliki oleh *javascript* sebelum kemunculan *big integer* :

```
console.log(Number.MAX_SAFE_INTEGER);
// output : 9007199254740991
console.log(12345678901234567890123456789012345678901234567890n);
// output : 12345678901234567890123456789012345678901234567890n
console.log(typeof 12345678901234567890123456789012345678901234567890n);
// output : bigint
```

**Link* sumber kode.

Arithmetic Operation

Seperti pada tipe data *number, big integer* juga dapat digunakan untuk melakukan operasi aritmetika :

```
console.log(50n*2n); //100n
console.log(50n / 2n);// 25n
console.log(5n / 2n); // 2n
```

Pembagian antara 5 dan 2 akan menghasilkan *decimal number*, namun akan dibulatkan ke dalam *integer* sehingga akan kehilangan *fractional digit*.

Selain itu kita tidak bisa menggabungkan operasi aritmetika antara *big integer* dan *number* :

```
console.log(100n+25);
// TypeError: Cannot mix BigInt and other types,
   use explicit conversions
```

Jika ingin tetap melakukan operasi penjumlahan, salah satu *number* harus dikonversi dulu ke dalam tipe data *big integer* atau *number*.

Eksistensi *Big Integer* membawa dunia baru dalam operasi aritmetika tanpa mengalami *overflowing*, membuka beberapa kemungkinan baru dalam pengembangan aplikasi. Salah satunya adalah operasi matematika dalam jumlah besar dalam dunia teknologi keuangan.

Comparison

Untuk operasi perbandingan antara big integer dan number, kita dapat melakukanya :

```
console.log(1n < 2); // true
console.log(2n > 1); // true
```

```
console.log(2n > 2); // false
console.log(2n >= 2); // true
```

Secara nilai sama namun tipe data nya berbeda :

```
console.log(0n === 0); // false
console.log(0n == 0); // true
```

**Link* sumber kode.

10. *Clean Code Data Types*

Declare Primitive Not Object

Untuk deklarasi *string, number* atau *boolean* sangat disarankan menggunakan *primitive data type* bukan *object*, karena dapat memproduksi kode yang akan dieksekusi dengan lambat dan memberikan hasil komputasi yang berbeda :

```
var x = "Gun Gun Febrianza";
var y = new String("Gun Gun Febrianza");
console.log((x === y)); // false
```

**Link* sumber kode.

Pada kode di atas hasilnya *false* karena `x` merupakan *primitive* dan `y` adalah *object* (*reference type*).

Stop using new Keyword

Berhenti menggunakan keyword new :

1. Untuk membuat *object* baru gunakan `{}` jangan gunakan `new Object()`
2. Untuk membuat *object* baru gunakan `""` jangan gunakan `new String()`
3. Untuk membuat *object* baru gunakan `0` jangan gunakan `new Number()`
4. Untuk membuat *object* baru gunakan `false` jangan gunakan `new Boolean()`
5. Untuk membuat *object* baru gunakan `[]` jangan gunakan `new Array()`
6. Untuk membuat *object* baru gunakan `/()/` jangan gunakan `new RegExp()`
7. Untuk membuat *object* baru gunakan `function (){}` jangan gunakan `Function()`

Seperti kode yang ada di bawah ini :

```
var v1 = {};           // new object
```

```
var v2 = "";            // new primitive string
var v3 = 0;             // new primitive number
var v4 = false;         // new primitive boolean
var v5 = [];            // new array object
var v6 = /()/;          // new regexp object
var v7 = function () { }; // new function object
```

**Link* sumber kode.

Subchapter 3 – Control Flow

The most important property of a program is
Whether it accomplishes the intention of its user.

— C.A.R Hoare

Subchapter 3 – Objectives

- Memahami Apa itu ***Block Statements***?
- Memahami Apa itu ***Conditional Statements***?
- Memahami Apa itu ***Ternary Operator***?
- Memahami Apa itu ***Multiconditional Statement***?
- Memahami Apa itu ***Switch Style***?

Control Flow menjelaskan mana urutan *expression* & *statements* yang akan dieksekusi. Sehingga *Web Application* yang dibuat dengan *javascript* menjadi lebih interaktif. Ada beberapa hal yang akan kita bahas disini di antaranya adalah :

1. *Block Statements*

Block statements atau terkadang disebut *compound statement* adalah sekumpulan *statements* yang akan dieksekusi secara berurutan di dalam sebuah kurung kurawal buka dan kurung kurawal tutup. ({ … }). Perhatikan contoh *syntax* di bawah ini :

```
{
  statement_1;
  statement_2;
  ...
  statement_n;
}
```

2. *Conditional Statements*

Conditional statements adalah sekumpulan perintah yang akan dieksekusi jika kondisi bernilai *true*. Pada kode di bawah ini terdapat dua *block statements,* **statement_1** ada pada *block statements* pertama dan **statement_2** ada pada *block statements* kedua.

Perhatikan contoh *syntax* di bawah ini :

```
if(condition) {
  statement_1;
} else {
  statment_2
}
```

Contoh penggunaan *conditional statement* bisa kita lihat pada gambar di bawah ini :

```
var time = 20;
if (time < 20) {
  x = "statement_1"
} {
  x = "statement_2"
}
//output
//"statement_2"
```

**Link* sumber kode.

Jika nilai ***time*** di bawah 20 maka *block statement* pertama akan dieksekusi namun pada kasus di atas nilai ***time*** adalah sama sehingga kondisi yang dihasilkan adalah *false*, karena angka 20 tidak lebih kecil dari 20. Dengan begitu ***block statements*** **yang kedua** dieksekusi.

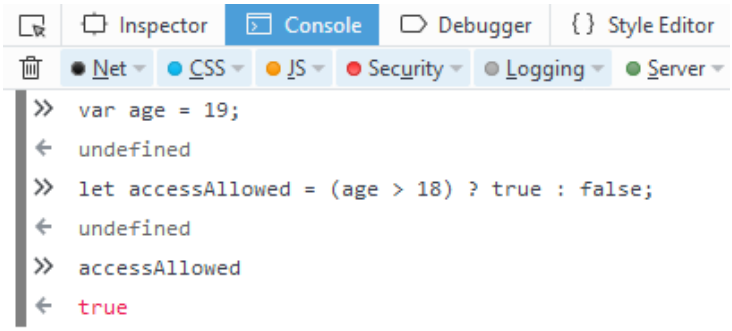
3. *Ternary Operator*

Dengan *ternary operator* menggunakan *question mark* (?), kita bisa membuat *Control Flow* yang lebih pendek dan sederhana. Terminologi *ternary* artinya kita menggunakan operator yang memiliki 3 *operand*. Perhatikan contoh *syntax* di bawah ini :

```
let result = condition ? value1 : value2
```

Gambar 215 Syntax Ternary Opertor

Variabel **`result`** akan mendapatkan nilai berdasarkan kondisi yang diberikan, jika hasil kondisi bernilai **`true`** maka kode yang akan di eksekusi adalah **`value1`** dan jika kondisi bernilai **`false`** maka kode yang akan dieksekusi adalah **`value2`**.



Gambar 216 Contoh kode Ternary Operator

Bagaimana jika kita ingin menggunakan lebih dari satu kondisi menggunakan *ternary operator*? Kita bisa melakukanya perhatikan kode di bawah ini :

```
let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello! Sweety' :
    (age < 100) ? 'Hello Pretty!' : 'Who you are?!';

alert(message);
```

**Link* sumber kode.

4. *Multiconditional Statement*

Untuk menghadapi *multiple condition* kita bisa menggunakan ***If***.. ***Else If*** dan ***Else***, *block statement* ke ***n*** akan dieksekusi jika kondisinya memenuhi syarat. Perhatikan contoh *syntax* di bawah ini :

```
if (condition) {
statement_1;
} else if (condition) {
statement_2;
} else if (condition) {
statement_3;
} else {
statement_last;
}
```

Contoh penggunaan *multiple condition statements* bisa kita lihat pada gambar di bawah ini :

```
var time = 44
if (time < 20) {
  x = "Statement_1"
} else if (time == 20) {
  x = "Statement_2"
} else {
  x = "Statement_3"
}
//output
```

```
//"Statement_3"
```

**Link* sumber kode.

Pada gambar di atas ***Statement*_3** dieksekusi karena hanya *block statement* yang ketiga yang memenuhi syarat.

5. *Switch Style*

Selanjutnya kita akan mempelajari *Switch Statements* yang akan mengeksekusi *statements* berdasarkan *label* yang sama. Perhatikan contoh *syntax* di bawah ini :

```
switch (expression) {

  case label_1:
    statements_1
    [break;]

  case label_2:
    statements_2
    [break;]

    ...

  default:
    statements_def
    [break;]
}
```

Contoh penggunaan *Switch Statements* bisa kita lihat pada gambar di bawah ini :

```
var day = 2
switch (day) {
  case 0:
    x = "Hari ini minggu"
    break;
  case 1:
    x = "Hari ini senin"
    break;
  case 2:
    x = "Hari ini selasa"
    break;
}
//ouput
```

```
    //"Hari ini selasa"
```

**Link* sumber kode.

Label yang digunakan sebagai kondisi adalah ***number***, karena nilai *number* adalah adalah 2 maka *statement* di dalam *case* 2 yang akan di eksekusi. *Keyword* **`break`** digunakan untuk menghentikan *statement* agar *case* berikutnya tidak dieksekusi.

Subchapter 4 – Loop & Iteration

Programmers are not to be measured by their ingenuity and their logic
but by the completeness of their case analysis.

— Alan J. Perlis

Subchapter 4 – Objectives

- Memahami Apa itu ***While Statements***?
- Memahami Apa itu ***Do..While Statements***?
- Memahami Apa itu ***For Statements***?
- Memahami Apa itu ***Break Statement***?
- Memahami Apa itu ***Continue Style***?
- Memahami Apa itu ***Labeled Style***?

Ada saatnya kita ingin mengulang kode yang sama untuk dieksekusi berkali kali, pada *javascript* kita bisa menggunakan beberapa cara diantaranya adalah :

1. *While Statement*

Pada *while statement* selama kondisi bernilai *true* maka *block statements* akan terus dieksekusi. Eksekusi di dalam *loop body* disebut dengan ***iteration***. Kita bisa membuat sebuah *expression* di dalam *loop body*. Perhatikan contoh *syntax* di bawah ini :

```
while (condition) {
// bagian "loop body"
}
```

Contoh penggunaan *While Statement* dengan tiga *iteration* bisa kita lihat pada kode di bawah ini :

```
var n = 0;
while (n < 3) {
```

```
  console.log(n++);
}
/* output :
0
1
3 */
```

**Link* sumber kode.

Selain cara di atas juga terdapat *shorthand-while*, perulangan menggunakan *while* dalam satu *statement* :

```
let i = 3;
while (i) console.log((i--));
/* output
3
2
1 */
```

**Link* sumber kode.

2. *Do ... While Statement*

Pada *do ... while statement*, sebuah *block statement* akan dieksekusi terlebih dahulu sebelum kondisinya dievaluasi. Jika kondisi bernilai *true*, *block statement* akan dieksekusi kembali, saat kondisi sudah bernilai *false*, *block statement* akan kembali dieksekusi sekali lagi untuk melanjutkan eksekusi baris kode berikutnya. Perhatikan contoh *syntax* di bawah ini :

```
do
    statement
while (condition)
```

Contoh penggunaan *Do ... While Statement* bisa kita lihat pada kode di bawah ini :

```
var i = 0;
do {
  i += 1;
  console.log(i);

} while (i < 5)
/* Output
1
2
3
4
5 */
```

**Link* sumber kode.

3. *For Statement*

```
for ([initialExpression]; [condition]; [incrementExpression];)
      statement
```

Di atas adalah contoh *syntax for statement*.

Pada *for statement*, perulangan akan terus terjadi sampai hasil evaluasi ***condition* mendapatkan nilai *false*.** Untuk melakukan perulangan kita harus mengatur ***initialExpression*** terlebih dahulu, sebuah nilai awal untuk melakukan perulangan. Selama *condition* bernilai *true* maka *statement* di dalam *block for statement* akan terus dieksekusi. Setiapkali *statement* di dalam *block for statement* dieksekusi ***incrementExpression*** akan terus meningkat.

Contoh penggunaan *For Statements* bisa kita lihat pada gambar di bawah ini :

```
for (let index = 0; index < 5; index++) {
  console.log("Perulangan ke " + index);
}
/* Output :
Perulangan ke 0
Perulangan ke 1
Perulangan ke 2
Perulangan ke 3
Perulangan ke 4
*/
```

**Link* sumber kode.

Catatan, bukan hanya *incrementExpression* yang akan terus bertambah satu setiap kali perulangan dilakukan bisa juga *decrementExpression* yang akan terus berkurang satu setiap kali perulangan dilakukan.

```
for (let index = 10; index > 5; index--) {
```

```
    console.log("Perulangan ke " + index);
}

/* Output :
Perulangan ke 10
Perulangan ke 9
Perulangan ke 8
Perulangan ke 7
Perulangan ke 6
*/
```

**Link* sumber kode.

4. *Break Statement*

Sebelumnya anda sudah menggunakan ***keyword*** **break** saat mempelajari *Switch*. Pada perulangan *break* juga dapat digunakan untuk keluar dari suatu perulangan dan terus mengeksekusi kode setelah keluar dari perulangan.

```
for (i = 0; i < 10; i++) {
  if (i === 3) { break; }
  console.log("Perulangan ke " + i);
}
/* Output :
Perulangan ke 0
Perulangan ke 1
Perulangan ke 2
*/
```

**Link* sumber kode.

5. *Continue Statement*

Penggunaan **continue** dilakukan jika kita ingin menghentikan eksekusi suatu ***statement*** dalam suatu ***iteration*** dan melanjutkan kembali perulangan dengan ***iteration*** selanjutnya sampai selesai.

```
for (i = 0; i < 6; i++) {
  if (i === 3) { continue; }
  console.log("Perulangan Ke " + i);
}

// Output
// Perulangan Ke 0
// Perulangan Ke 1
// Perulangan Ke 2
// Perulangan Ke 4
// Perulangan Ke 5
```

**Link* sumber kode.

6. *Labeled Statement*

Dengan *label statement* kita dapat membuat sebuah *identifier* sebagai sebuah *statement* yang menjadi acuan saat menggunakan *break* atau *continue statement*.

```
var str = "";

loop1:
for (var i = 0; i < 5; i++) {
  if (i === 1) {
    continue loop1;
  }
  str = str + i;
  console.log(str);

}

//output :
// 0
// 02
// 023
// 0234
```

**Link* sumber kode.

Subchapter 5 – Function

Any application that can be written in JavaScript
will eventually be written in JavaScript.

—Atwood's Law, by Jeff Atwood

Subchapter 5 – Objectives

- Memahami Apa itu ***Function***?
- Memahami Apa itu ***First-class Function***?
- Memahami Apa itu ***Function Parameter***?
- Memahami Apa itu ***Function Return***?
- Memahami Apa itu ***Function with Local & Outer Variable***?
- Memahami Apa itu ***Callback Function***?
- Memahami Apa itu ***Arrow Function***?
- Memahami Apa itu ***Multiline Arrow Function***?
- Memahami Apa itu ***Function Constructor***?
- Memahami Apa itu ***Function As Expression***?
- Memahami Apa itu ***Nested Function***?
- Memahami Apa itu ***Argument Object***?
- Memahami Apa itu ***Call & Apply Function***?
- Memahami Apa itu ***Call & Apply Function Argument***?
- Memahami Apa itu ***Bind***?
- Memahami Apa itu ***This Keyword***?
- Memahami Apa itu ***IIFE***?

1. Apa itu *Function*?

Function adalah sebuah *subprogram* yang didesain untuk menyelesaikan suatu pekerjaan. *Function* akan dieksekusi jika telah kita panggil, fenomena memanggil fungsi disebut dengan ***Invoking***. Sebuah *function* selalu menghasilkan sebuah *return*, dalam *javascript* jika sebuah *function* tidak memiliki ***return*** maka akan menghasilkan *return undefined*.

Secara garis besar sebuah *function* dapat ditulis dalam bentuk :

1. *Function Declaration*
2. *Function Expression*

Function Declaration

Function Declaration digunakan jika kita ingin membuat sebuah fungsi. Saat melakukan deklarasi kita perlu menggunakan *function keyword* diikuti nama fungsi yang ingin dibuat. Di bawah ini adalah contoh *syntax function* :

```
function name(parameters) {
  statements
}
```

Function Expression

Function Expression digunakan jika kita ingin membuat *anonymous function*, sebuah *anonymous function* tidak memiliki *identifier* atau nama. *Syntax* di bawah ini ini adalah contoh membuat *anonymous function* yang disimpan ke dalam variabel **`name`** :

```
let name = function (parameters) {
  statements
}
```

Arrow Function Expression

Kita juga dapat menggunakan *Arrow Function Expression* untuk mempersingkat penulisan kode *function expressions*. Di bawah ini adalah contoh *syntax* penggunaan ***Arrow Function Expression*** yang setara dengan *Function Expression* sebelumnya :

```
let name = (parameters) => {
  statements
}
```

2. First-class Function

Fungsi atau *Function* adalah hal *fundamental* dalam *javascript*. Memahami *function* dalam *javascript* artinya kita memperkuat persenjataan kita untuk memahami *javascript*. Dalam *javascript* sebuah *function* diperlakukan seperti *object*, direferensikan sebagai sebuah variabel, dideklarasikan secara *literal*, dan juga bisa diperlakukan sebagai *argument* sebuah *function*. *Javascript* adalah bahasa pemrograman yang mendukung *first class-function*.

Sebuah bahasa pemrograman dikatakan memiliki ***First-class function*** saat *function* dalam bahasa pemograman tersebut dapat diperlakukan seperti sebuah variabel.

Sebagai contoh *function* dapat digunakan :

1. Sebagai sebuah ***argument*** untuk sebuah fungsi
2. Sebagai sebuah ***return*** dari sebuah *function* dan
3. *Function* dapat disimpan sebagai sebuah nilai ke dalam sebuah variabel.

What is Execution Context (EC)?

Dalam *javascript* saat kita membuat sebuah variabel atau *function* terdapat ***Execution Context*** yang menjadi konsep abstrak tempat seluruh kode *javascript* dievaluasi dan dieksekusi. Terdapat 2 *Execution Context* :

Global Execution Context

Kode *javascript* yang tidak berada di dalam suatu *function* maka kode tersebut berada di dalam *global excution context*. Di dalam satu program *javacript* hanya terdapat 1 *Global Execution Context*.

```
var a = 10; // variabel berada dalam global context
```

```
(function () {
  var b = 20; // variabel lokal berada dalam function
context
})();

console.log(a); // 10
console.log(b); // "b" belum didefinisikan
```

**Link* sumber kode.

Functional Execution Context

Setiap kali suatu fungsi dipanggil sebuah *execution context* baru dibuat, setiap fungsi memiliki *execution context* masing-masing.

Execution Stack Theory

Execution Stack adalah sebuah *Data Structure Stack* untuk menampung seluruh *execution context* saat kode *javascript* sedang dieksekusi. *Javascript Engine* akan membangun terlebih dahulu *global execution context* dan memasukanya terlebih dahulu kedalam *execution stack*.

Jika *Javascript Engine* menemukan terdapat *invoke* pada suatu *function*, maka *execution context* baru akan dibuat dan di *push* ke dalam *execution stack* diposisi paling atas.

Jika *function* tersebut telah selesai dieksekusi, maka *execution stack* akan mencabut *execution context* paling atas. Menandakan *execution context* selanjutnya siap dievaluasi sampai *execution stack* kembali kosong yang menandakan program telah selesai. Untuk lebih memahami kita akan membuat simulasinya :

Execution Stack Simulation

Di bawah ini adalah sebuah program *javascript* yang terdiri dari deklarasi variabel, deklarasi fungsi :

```
let hello = 'Hello World!';

function executionContextPertama() {
  console.log('Di dalam function');
  executionContextKedua();
  console.log('Di dalam function');
}

function executionContextKedua() {
  console.log('Di dalam function');
}

executionContextPertama();
console.log('Di dalam Global Execution Context');
```

**Link* sumber kode.

Ketika kode di atas dieksekusi, *javascript engine* akan membuat *global excution context* terlebih dahulu dan memasukanya kedalam *execution stack*.

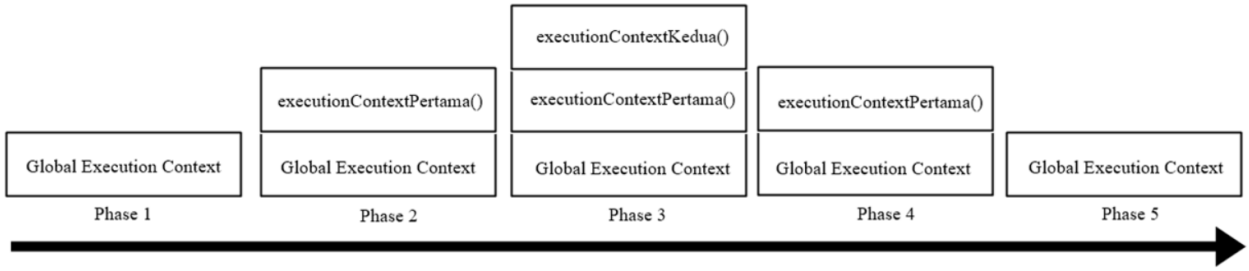
Ketika fungsi **`executionContextPertama()`** dipanggil maka *execution context* baru akan dibuat dan menyimpanya (*push*) ke dalam *exection stack* di posisi paling atas.

Ketika **`executionContextKedua()`** dipanggil maka di dalam

`executionContextPertama()` *javascript engine* akan membuat *execution context* baru dan menyimpanya ke dalam *execution stack* di posisi paling atas lagi.

Jika **`executionContextKedua()`** selesai dieksekusi *execution context function* tersebut akan dicabut dari *execution stack* (*pop*), hingga mencapai **`executionContextPertama()`** yang juga akan dicabut dalam *execution stack* (*pop*).

Di bawah ini adalah visualisasi proses yang terjadi di dalam *execution stack* :



Gambar 217 Execution Stack Simulation

3. *Simple Function*

Di bawah ini adalah bentuk paling sederhana untuk membuat fungsi :

```
function sayHello() {
  console.log('Hello!');
  console.log('Gun Gun Febrianza!');
}
```

**Link* sumber kode.

Untuk menggunakan fungsi tersebut kita hanya perlu memanggil nama fungsinya atau *identifier* fungsinya lengkap dengan kurung buka dan kurung tutupnya `sayHello()` :

```
sayHello()
/*
Output
Hello!
Gun Gun Febrianza! */
```

Kita dapat menggunakan fungsi tersebut lagi dan lagi :

```
sayHello()
sayHello()
/*
Output
Hello!
Gun Gun Febrianza!
Hello!
Gun Gun Febrianza!  */
```

4. *Function Parameter*

Di bawah ini adalah sebuah *function* yang memiliki *parameter*, terdapat dua *parameter* yaitu **`from`** dan **`text`** :

```
function tampilkanPesan(from, text) {
  // arguments: from, text
  console.log(from + ': ' + text);
}

tampilkanPesan('Kaiz', 'Hello! Maudy Ayunda!');
tampilkanPesan('Maudy Ayunda', "What's up? kaiz!");
/*
Output
Kaiz: Hello! Maudy Ayunda!
Maudy Ayunda: What's up? kaiz! */
```

**Link* sumber kode.

Pada kode di atas kita dapat memanfaatkan *parameter* agar bisa memberikan *output* yang berbeda. Pada *function body* kita membuat sebuah *statement* yang membutuhkan *parameter* agar dapat dieksekusi.

Tapi apa jadinya jika kita hanya memberikan satu *parameter* saja ? misal :

```
tampilkanPesan('Kaiz');
tampilkanPesan('Maudy Ayunda', "What's up? kaiz!");
/*
Output
Kaiz: undefined
Maudy Ayunda: What's up? kaiz! */
```

Jawabannya adalah pada *parameter* kedua nilainya adalah *undefined*. Saat pengembangan *software* di *production* kita harus memastikan terlebih dahulu agar setiap *input parameter* yang diberikan tidak boleh kosong.

5. *Function Return*

Di bawah ini adalah sebuah *function* yang memiliki sebuah *return* :

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
console.log(result); // 3
```

Pada kode di atas kita menggunakan ***keyword*** `return` di dalam ***body function***, setiap ***function*** hanya dapat memiliki satu buah ***keyword*** `return`.

Saat *statement* yang mengandung *keyword* `return` dieksekusi maka kode di bawahnya tidak akan dieksekusi.

```
function sum(a, b) {
  console.log('message 1');
  return a + b;
  console.log('message 2');
}

let result = sum(1, 2);
console.log(result); // 3

/*
Output
message 1
3
```

Statement ***console.log('message 2');*** tidak akan pernah dieksekusi.

Function yang memiliki *return* dapat digunakan untuk melakukan *assignment operation* untuk menyimpan nilai komputasinya di dalam sebuah *variable* seperti. **`let result = sum(1,2);`**

6. *Function For Function Parameter*

Di bawah ini adalah contoh kode *function* yang digunakan sebagai *parameter* :

```
function tampilkanPesan(from, text = test()) {
  console.log(from + ': ' + text);
}

function test() {
  return 'Hello';
}
tampilkanPesan('Maudy Ayunda');
```

**Link* sumber kode.

Pada kode di atas kita menggunakan *function* `test()` sebagai salah satu *parameter* di dalam *parameter* yang dimiliki oleh *function* `tampilkanPesan()`.

7. Function & Local Variable

Di bawah ini jika kita membuat sebuah *variable* yang sekupnya berada di dalam *function* maka *variable* tersebut hanya dapat digunakan di dalam *function* tersebut.

```
  function tampilkanPesan() {
    let message = "Hello, I'm Message Variable Inside Function";
// local
    console.log(message);
  }

  tampilkanPesan();
  console.log(message);
// <-- Error! ReferenceError: message is not defined
```

**Link* sumber kode.

Pada kode di atas jika kita mencoba memanggil *local variable* yang dimiliki suatu *function* di luar *function* tersebut maka *error* akan terjadi.

8. *Function & Outer Variable*

Di bawah ini adalah contoh kode di mana *statement* di dalam *javascript function* mencoba mengakses variabel diluar sekup *function*. Operasi ini tetapdapat dilakukan di dalam *javascript* :

```
let userName = 'Gun Gun Febrianza';

function tampilkanPesan() {
  let message = 'Hello, ' + userName;
  console.log(message);
}

tampilkanPesan(); // Hello, Gun Gun Febrianza
```

**Link* sumber kode.

9. *Callback Function*

Di bawah ini adalah contoh kode dimana *function* **`greeting()`** digunakan sebagai *callback* di dalam *function* **`processUserInput()`**.

Penggunaan *Callback* menggunakan *keyword callback* :

```
function greeting(name) {
  console.log('Hello ' + name);
}

function processUserInput(callback) {
  var name = 'Gun Gun Febrianza';
  callback(name);
}

processUserInput(greeting);
//Hello Gun Gun Febrianza
```

**Link* sumber kode.

10. *Arrow Function*

Arrow function pertama kali diperkenalkan dalam ES6, dengan *arrow function* kita dapat membuat sebuah *function* dengan *syntax* yang lebih singkat.

Di bawah ini adalah contoh kode penggunakan *arrow function* :

```
let sum = (a, b) => a + b;

/* setara dengan:

  let sum = function(a, b) {
  return a + b;
  };
  */

console.log(sum(1, 2)); // 3
```

**Link* sumber kode.

Karakteristik lain dari *arrow function* adalah secara otomatis memberikan **`return`** tanpa harus menambahkan *keyword* **`return`**. Pada kode di atas operasi penjumlahan secara otomatis memberikan sebuah *return* dari hasil komputasinya.

11. *Multiline Arrow Function*

Di bawah ini adalah contoh kode *multiline arrow function*, kita hanya perlu menggunakan *curly brace* dan *keyword return* :

```
  let sum = (a, b) => { // curly brace untuk membuat multiline fu
nction
    let result = a + b;
    return result; // gunakan return untuk memproduksi hasil
  };

  console.log(sum(1, 2)); // 3
```

**Link* sumber kode.

12. *Anonymous Function*

Anonymous function adalah sebuah ***function*** yang tidak memiliki ***identifier***. Sebelumnya setiap kali kita membuat sebuah ***function***, kita selalu memberikan nama pada ***function*** tersebut. ***Anonymous function*** adalah ***function*** yang tidak memiliki nama.

Di bawah ini adalah contoh kode penggunaan *anonymous function* :

```
var x = function(a, b) {
  return a * b;
};
console.log(x(4, 3));//12
```

**Link* sumber kode.

13. *Function Constructor*

Kita dapat membuat menggunakan *function constructor* untuk membuat *function object*.

```
function UserCredential(username, password) {
  this.username = username;
  this.password = password;
}
```

Untuk membuat *object* menggunakan *function constructor* gunakan *keyword* **new** :

```
const user = new UserCredential("Maudy", "indonesia2020");
```

Pada kode di atas kita membuat *object* **UserCredential** lengkap dengan *parameter* untuk **username** dan **password**. Untuk mendapatkan *values* yang dimiliki oleh *object* cukup akses *property* dari *object* tersebut :

```
 console.log(user);
 //UserCredential { username: 'Maudy', password: 'indonesia2020'
}
 console.log(user.username); //Maudy
 console.log(user.password); //indonesia2020
```

Cara ini dapat kita gunakan jika ingin membuat *function* yang dinamis, namun memiliki permasalahan keamanan dan *performance*.

```
var rekomendasiFunction = function expressions(a, b) {
  return a * b;
};
```

**Link* sumber kode.

14. *Function As Expression*

Pada kode di bawah ini kita dapat memperlakukan sebuah *function* sebagai ***operand*** untuk membuat sebuah *expression* :

```
function myFunction(a, b) {
  return a * b;
}
var x = myFunction(4, 3) * 2; //expression
console.log(x);
```

**Link* sumber kode.

15. *Nested Function*

Kita juga dapat membuat *function* di dalam sebuah *function* sering kali disebut dengan fungsi bersarang atau *nested function*. Contoh kode :

```
function add() {
  var counter = 0;

  function plus() {
    counter += 1;
  }
  plus();
  return counter;
}
console.log(add());
```

**Link* sumber kode.

16. *Argument Object*

Javascript Function memiliki *object* bawaan di dalamnya yang dikenal dengan sebutan ***Arguments Object*****.** Pada *argument object* terdapat *array* dari *arguments* yang digunakan saat fungsi di panggil. Di bawah ini adalah contoh penggunaan *argument object* :

```
function sumAll() {
   var i;
   var sum = 0;

   for (
     i = 0;
     i < arguments.length;
     i++
   ) {
     sum += arguments[i];
   }
   return sum;
 }
 console.log(sumAll(1, 123, 500, 115, 44, 88));
 //871
```

**Link* sumber kode.

17. *This Keyword*

Setiap kali kita membuat sebuah *function*, sebuah *keyword* bernama **this** juga ikut dibuat di dalamnya secara *under the hood* (kita tidak bisa melihatnya).

Implicit Binding

Bisa kita buktikan jika kita mengesekusi kode di bawah ini :

```
// define a function
var myFunction = function () {
  console.log(this);
};

// call it
myFunction();
undefined
▸ Window about:newtab
```

Gambar 218 this keyword in the global scope

Pada kode diatas kita membuat sebuah *function*, ketika kita mengeksekusi kode di atas tanpa *strict mode* dan kita memanggil **this**, maka munculah **Windows** sebagai *root object* di dalam sebuah *tab web browser*.

```
var myObject = {
  myMethod: function () {
    console.log(this);
  }
};
undefined
myObject
{…}
  ▾ myMethod: myMethod()
      arguments: null
      caller: null
      length: 0
      name: "myMethod"
    ▸ prototype: Object { … }
    ▸ <prototype>: function ()
  ▸ <prototype>: Object { … }
```

Gambar 219 this in myObject scope

Pada kode di atas kita membuat sebuah *object* bernama **myObject** dan *object* tersebut memiliki sebuah *property* bernama **myMethod** dan nilainya adalah suatu *function*. Ketika kita mengeksekusi kode di atas hasil dari **this** adalah **myObject** itu sendiri. Fenomena ini di dalam javascript disebut dengan *implicit binding* :D

Jika anda membaca kode di bawah ini, maka anda akan memahami *keyword* **this** dengan baik :

```
function account(username, password) {
  this.username = username;
  this.password = password;
  this.captcha = 9998;
}

let myaccount = new account('gun@gmail.com', 'test1234');
console.log(myaccount);
console.log(typeof myaccount);
console.log(myaccount.captcha);
```

**Link* sumber kode.

Penggunaan *keyword* **this**, digunakan agar kita memiliki akses terhadap *property* yang dimiliki oleh sebuah *function*. Pada *function* di bawah ini terdapat penggunaan *keyword* **this** yang menegaskan bahwa *function* **account** memiliki 2 *properties* yaitu **username** dan **password**.

Di halaman selanjutnya kita akan belajar cara melakukan ***explicit binding***.

18. *Call & Apply Function*

Call adalah sebuah *method* yang menjadi *prototype* dari *function object*, digunakan untuk melakukan *explicit binding*. Kita dapat menerapkan sebuah *context* ke dalam sebuah *function*. *Context* yang dimaksud dapat berupa suatu *object* & *properties*-nya dapat digunakan oleh sebuah *function*.

Pada kode di bawah ini *context* yang dimaksud adalah *object* dengan *identifier* **`person1`**, *properties* yang dimiliki oleh *object* tersebut dapat digunakan oleh *function* **`fullName()`** yang dimiliki oleh *object* **`person`**.

```
var person = {
  fullName: function () {
    return this.firstName + ' ' + this.lastName;
  }
};
var person1 = {
  firstName: Gun Gun,
  lastName: 'Febrianza'
};

var x = person.fullName.call(person1);
console.log(x); //Gun Gun Febrianza
```

**Link* sumber kode.

Pada contoh kode di bawah ini, kita dapat menggunakan *properties* dari suatu *object* di dalam suatu *object*. Pada *object* **`Food`** kita mengeksekusi *method* **`call`** menggunakan *object* **`Product`** sehingga *object* **`Food`** memiliki *properties* yang dimiliki oleh *object* Product.

```
function Product(name, price) {
  this.name = name;
```

```
    this.price = price;
  }

  function Food(name, price) {
    Product.call(this, name, price);
    this.category = 'food';
  }

  var x = new Food('cheese', 5);

  console.log(x.name); //cheese
  console.log(x.category); //food
```

Explicit Binding

Cara kerja *Call & Apply* memang identik.

perbedaanya pada *method* **`Call`**, *parameter* yang dapat diterima adalah *argument list* :

```
var x = person.fullName.call(person1, 'Bandung', 'Indonesia');
```

Sementara pada *method* **`apply`**, *parameter* yang dapat diterima adalah *argument list* :

```
var x = person.fullName.apply(person1, ['Antares', 'Denmark']);
```

Untuk melihat perbedaanya lebih detail lagi silahkan tulis, eksekusi dan pelajari kode di bawah ini, terkait perbedaan *Call & Apply* :

Call

Pada kode di bawah ini, kita dapat menggunakan *method* **`Call`** agar *properties* di dalam sebuah *object* dapat digunakan *method* **`fullName()`** yang dimiliki oleh *object* **`person`**.

```
  var person = {
    fullName: function (city, country) {
      return this.firstName + ' ' + this.lastName + ',' + city +
',' + country;
    }
  };

  var person1 = { firstName: 'Gun Gun', lastName: 'Febrianza' };

  var x = person.fullName.call(person1, 'Bandung', 'Indonesia');
  console.log(x); // Gun Gun Febrianza,Bandung,Indonesia
```

**Link* sumber kode.

Pada *method* **`call`** terdapat *parameter* **`person1`** dan *argument list* artinya kita akan membuat *keyword* this baru di dalam **`fullname()`** *method* yang nilainya dapat digunakan.

Apply

Pada kode di bawah ini, kita dapat menggunakan *method* **`apply`** agar *properties* di dalam sebuah *object* dapat digunakan *method* **`fullName()`** yang dimiliki oleh *object* **`person`**.

```
  var person = {
    fullName: function (city, country) {
      return this.firstName + ' ' + this.lastName + ',' + city +
',' + country;
    }
  };
  var person1 = {
    firstName: 'Nikolaj',
    lastName: 'Vestorp'
  };
  var x = person.fullName.apply(person1, ['Antares', 'Denmark']);
```

```
  console.log(x);
```

**Link* sumber kode.

Pada *method* **`apply`** terdapat *parameter* **`person1`** dan *array*, artinya kita akan membuat *keyword this* baru di dalam **`fullname()`** *method* yang nilainya dapat digunakan.

19. *IIFE*

Immediately-invoked Function Expression atau disingkat IIFE adalah cara lain untuk mengeksekusi sebuah fungsi. Ketika *javascript engine* selesai membaca IIFE maka fungsi akan dieksekusi.

```
(function () {
  console.log('hello');
}());
```

IIFE juga dapat menggunakan *arrow function* :

```
( () => {
  console.log('hello');
}());
```

**Link* sumber kode.

20. *Clean Code Function*

Always Declare Local Variable

Seluruh variabel yang hanya akan digunakan di dalam sebuah *function*, wajib di deklarasikan di dalam *function* tersebut. Minimalisir penggunaan *global variable*.

Use Named Function Expression

Saat membuat function sangat disarankan kita tidak menggunakan function declaration tetapi menggunakan named function expression. Saran ini mengikuti konvensi standar *Ecma Script Linter* yang memiliki pengaturan *func-style*.

```
// bad
function foo() {
  // ...
}

// bad
const foo = function () {
  // ...
};

const short = function longUniqueMoreDescriptiveLexicalFoo() {
  // ...
};
```

Saran ini mengikuti konvensi standar *Ecma Script Linter* yang memiliki pengaturan *func-style*.

Use Default Parameter

Setiap kali sebuah *function* menerima *missing argument*, maka akan mengganggu komputasi di dalam *function* tersebut. Oleh karena itu perlu dibuatkan *default parameter* untuk mencegah nilai *undefined* dari *missing arguments* mengganggu komputasi *function*.

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

**Link* sumber kode.

Function is not statement

ECMA – 262 menjelaskan bahwa *block of code* adalah sekumpulan *statements*, *function declaration* bukanlah *statement* kode di bawah ini tidak sesuai dengan konvensi :

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}
```

Untuk memperbaikinya kita dapat menggunakan konvensi di bawah ini :

```
// good
let test;
if (currentUser) {
  test = () => {
```

```
    console.log('Yup.');
  };
}
```

**Link* sumber kode.

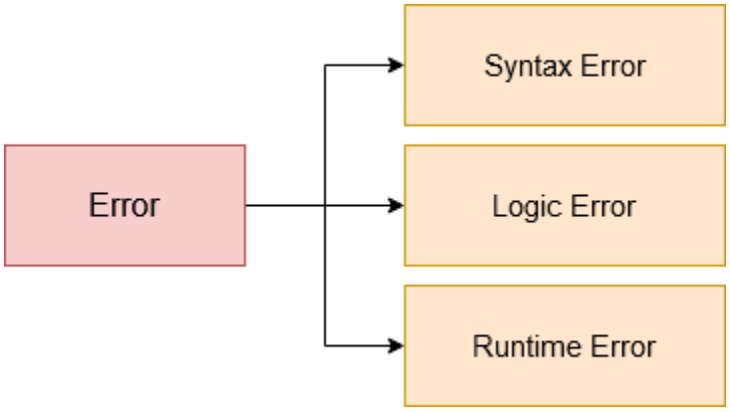
Subchapter 6 – Error Handling

The best error message is the one that never shows up.

—Thomas Fuchs

Subchapter 6 – Objectives

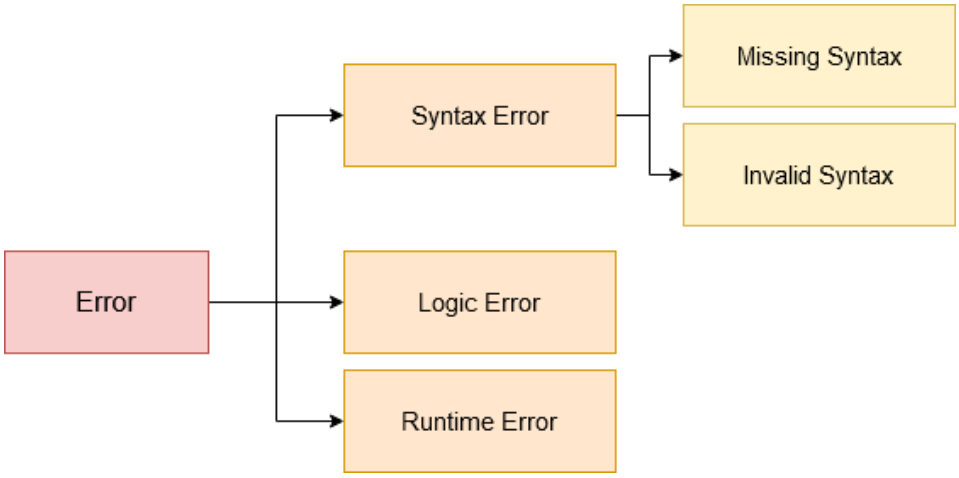
- Memahami Apa itu ***Syntax Error***?
- Memahami Apa itu ***Logical Error***?
- Memahami Apa itu ***Runtime Error***?
- Memahami Apa itu ***Reference Error***?
- Memahami Apa itu ***Range Error***?
- Memahami Apa itu ***Type Error***?
- Memahami Apa itu ***Try & Catch Statement***?
- Memahami Apa itu ***Finnaly Statement***?
- Memahami Apa itu ***Error Object Properties***?
- Memahami Apa itu ***Stack Trace***?
- Memahami Apa itu ***Custom Error***?



Gambar 220 Error Type

Sebuah program memiliki tiga kemungkinan *errors* yaitu *syntax error*, *logical error* dan *runtime error*. [59]

1. *Syntax Error*



Gambar 221 Syntax Error

Syntax Error adalah kesalahan yang paling sering terjadi. Kesalahan ini dapat dideteksi oleh interpreter, karena interpreter memiliki sebuah program internal yang disebut dengan *syntax analyzer*.

Faktor yang paling mempengaruhinya adalah kesalahan *typing* dan kesalahan aturan penulisan bahasa pemrograman yang disebut dengan ***syntax rule***.

Missing Syntax

Sebagai contoh kode *javascript* di bawah ini adalah susunan *declaration statement* yang benar :

```
var x = 10
```

Apa yang terjadi jika kita menghapus *identifier* **x** pada kode di atas?

```
>> var = 10
SyntaxError: missing variable name [Learn More]
```

Gambar 222 Syntax Error Example

Invalid Syntax

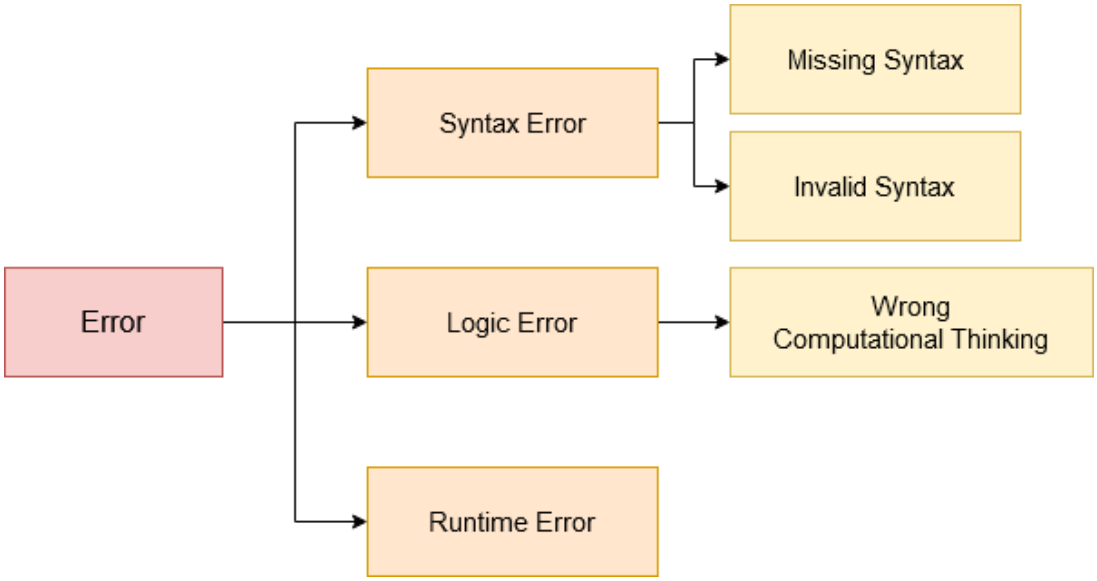
Invalid syntax terjadi ketika kita menambahkan sebuah *syntax* yang susunanya tidak sesuai dengan *syntax rule* yang dimiliki oleh *javascript*. Pada kode di bawah ini kita menambahkan lagi *keyword* **`var`** setelah *literal* **`10`**, akibatnya menimbulkan *syntax error*.

```
>> var x = 10var
SyntaxError: identifier starts immediately after numeric literal [Learn More]
```

Gambar 223 Invalid Syntax Example

Pada *syntax error* di atas *interpreter javascript* memperlakukan **`10var`** sebagai sebuah *identifier*, menolak *syntax rule* tersebut karena *identifier* tidak boleh di awali dengan angka. Pesan *error* ini telah di atur oleh pembuat *javascript engine*.

2. *Logical Error*



Gambar 224 Logic Error

Logical Error adalah kesalahan yang terjadi pada logika kode pemrograman yang ditulis oleh seorang *programmer*. Kesalahan ini tidak bisa dideteksi oleh kompiler, kesalahan ini hanya bisa diketahui ketika seorang *programmer* mulai melakukan evaluasi lagi pada kode pemrograman yang dibuatnya.

Kita coba dengan *study case* yang sederhana :

```
var x = 10;
var xx = 20
var result = 10 * x;
console.log(result);
```

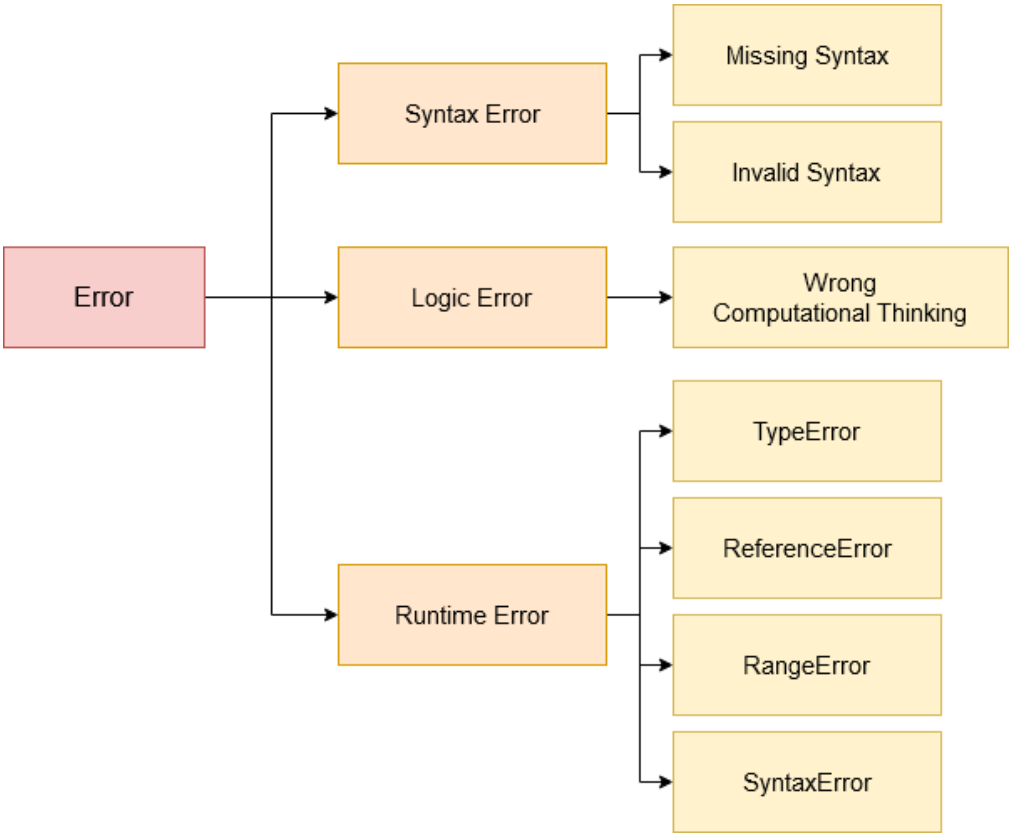
Kita asumsikan anda telah menulis kode hingga ratusan atau ribuan baris, kemudian anda berpikir bahwa seharusnya nilai dari variabel **`result`** adalah **`200`**. Karena anda yakin nilai **`x`** adalah **`20`**, padahal nilai **`x`** bukanlah **`20`** melainkan **`10`**, namun anda tidak menyadarinya sama sekali.

Pada kasus di atas terdapat dua kemungkinan *logic error* :

1. Kita salah memberikan *variable*, seharusnya variable **xx** agar hasilnya adalah **200**.
2. Jika nilai **x** adalah hasil dari suatu proses komputasi tertentu dan hasilnya tidak sesuai dengan perhitungan anda, maka terdapat kemungkinan kesalahan rumus komputasi. terdapat kemungkinan kesalahan dalam cara berpikir kita untuk menghitung (*human error*).

Dalam pemrograman kesalahan seperti ini bisa menjadi sangat sulit untuk di deteksi, *perhaps you will blamming your computer*. Lol

3. *Runtime Error*



Gambar 225 Runtime Error

Runtime Error adalah kesalahan yang dapat terjadi saat sebuah program sedang dalam keadaan dieksekusi. *Javascript engine* akan memberikan sebuah ***error object*** jika terjadi kesalahan saat ***runtime error***.

Error object adalah bagian dari *built-in object* dalam *javascript engine* yang dikategorikan sebagai *fundamental object*. Anda akan mempelajari lebih banyak tentang *built-in object* dan mengenal *fundamental object* di chapter selanjutnya.

Referrence Error

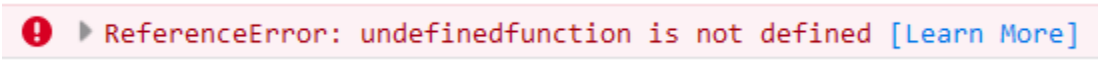
Javascript memiliki salah satu *type error* yang disebut dengan *reference error*.

Pada kode di bawah ini jika nilai dari variable **a** adalah **true** maka program akan tetap berjalan dengan baik, namun ternyata jika nilai dari hasil komputasi yang akan diberikan kepada *variable* **a** adalah **false** maka *error* akan terjadi :

```
var a = false
if (a === true) {
  console.log('var a = true');
} else {
  undefinedfunction()
}
```

**Link* sumber kode.

Jika kode di atas kita eksekusi maka kita akan mendapatkan pesan *Reference Error,* yang menegaskan kepada kita bahwa **undefinedFunction** belum dideklarasikan :

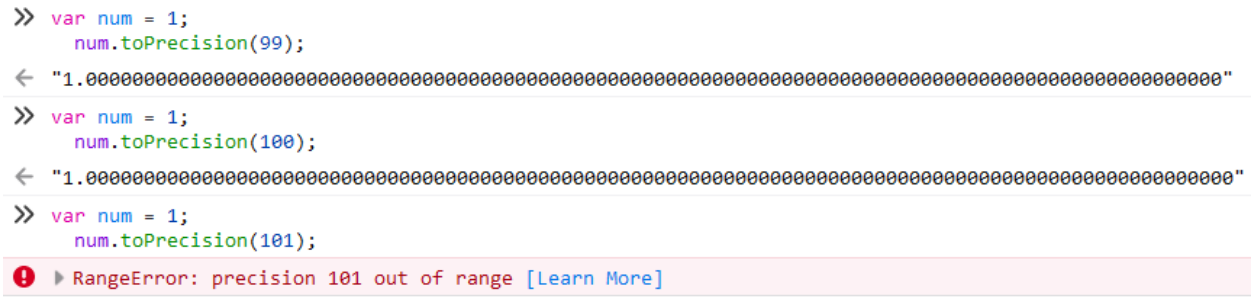


Gambar 226 Reference Error

Range Error

Range error terjadi karena kita memberikan sebuah ***argument*** yang jarak nilai nya diluar batasan kemampuan *interpreter*.

Sebagai contoh jika kita mengubah *number* **1** agar memiliki presisi lebih dari **100** maka *error* akan terjadi, pada contoh kasus di bawah ini kita menggunakan *method* **toPrecision()** :

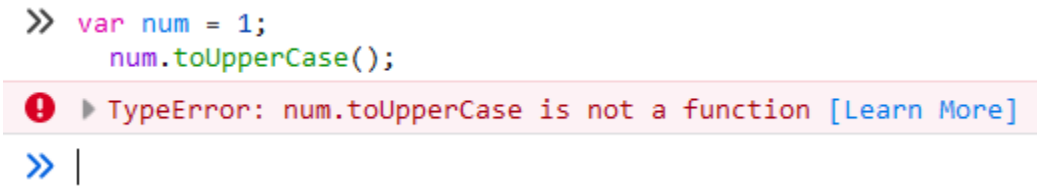


Gambar 227 Range Error

Type Error

Type error terjadi karena ketika kita memperlakukan suatu tipe data seperti menggunakan tipe data lainnya.

Contoh pada kode di bawah ini kita memperlakukan *number* dengan nilainya **1** seperti *string*, dengan cara memanggil *method* **toUpperCase()** yang hanya dimiliki oleh *object string*. Akibatnya interpreter akan memberikan pesan *type error* :



Gambar 228 Type Error

Syntax Error

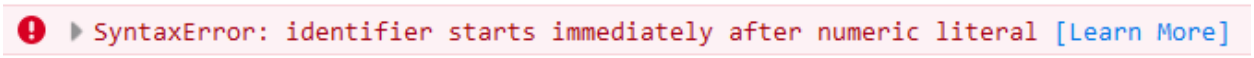
Syntax Error juga dapat terjadi saat *runtime*, sebagai contoh jika kita menggunakan *method* **eval()** yang dapat membaca *string javascript* untuk di eksekusi :

```
var a = false
if (a === true) {
  eval('2 + 2') // 4
} else {
  eval('2 + 2asd') //syntax error
```

```
}
```

**Link* sumber kode.

Jika hasil komputasi memberikan nilai `false` pada variabel `a` maka ***syntax error*** akan terjadi :



Gambar 229 Syntax Error

4. *Try & Catch*

Untuk mengatasi *error* dengan baik kita dapat menggunakan *keyword* **`try`** & **`catch`** :

```
try {
  test()
} catch (ex) {
}
```

Pada kode di atas, kita mencoba memanggil fungsi **`test()`** yang sama sekali belum kita deklarasikan, tentu saja ini akan mengakibatkan *error*. Namun, karena fungsi tersebut di deklarasikan di dalam **`try`** & **`catch`** akibatnya *error* yang dapat membuat program berhenti untuk berjalan tidak terjadi.

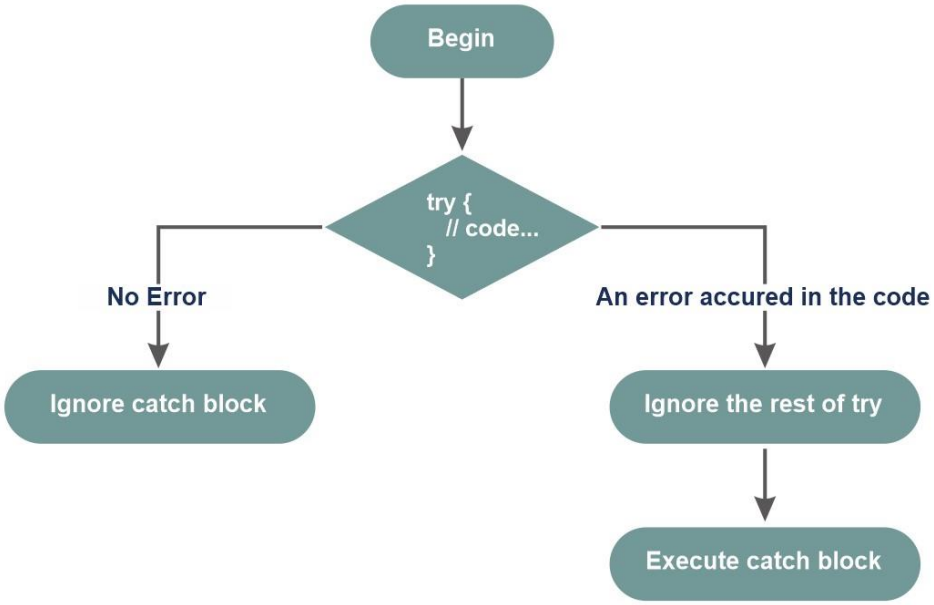
Meskipun begitu *error* tetaplah terjadi namun kita berhasil mengatasinya, untuk mengetahui tipe *error* dan pesan *error* kita dapat memanggil *error object*. Lalu melihat *properties name* & *message* yang dimiliki oleh *error object*.

Seperti pada kode di bawah ini :

```
try {
  test()
} catch (ex) {
  console.log(ex.name); //ReferenceError
  console.log(ex.message); //test is not defined
}
```

**Link* sumber kode.

Pada kode di atas di dalam **`try`** kita dapat memasukan *block of code* yang kita khawatirkan atau kita ketahui akan terjadi *error* dan pada **`catch`** kita dapat memasukan *block of code* yang dapat kita gunakan untuk mengatasi *error* tersebut.

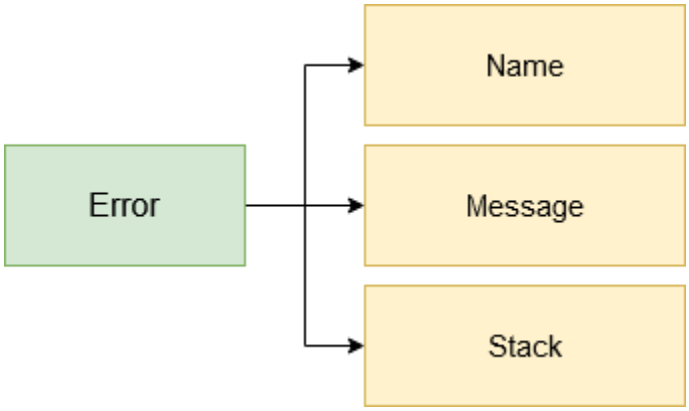


Gambar 230 Try & Catch Statement

Jadi jika dalam **`try`** *statement* tidak terdapat *error* maka *block of code* di dalam **`catch`** *statement* akan di abaikan, namun jika terdapat *error* di dalam **`try`** *statement* maka *statement* selanjutnya akan di abaikan dan seluruh *block of code* dalam **`catch`** *statement* akan dieksekusi.

Error Object Properties

Saat kita melakukan *handling error* menggunakan **`try`** & **`catch`**, kita dapat mengakses *properties* yang dimiliki oleh *object error* yaitu **`name`**, **`message`** & **`trace`** *properties*.



Gambar 231 Error Object Properties

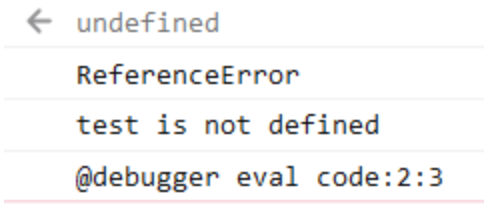
Property **`name`** akan memberikan informasi tipe *error* yang terjadi dan *property* **`message`** akan memberikan informasi pesan *error* yang terjadi.

Stack Trace

Pada kode di bawah ini kita memanggil *property* **`stack`** yang dimiliki oleh *error object* :

```
try {
  test()
} catch (ex) {
  console.log(ex.name); //ReferenceError
  console.log(ex.message); //test is not defined
  console.log(ex.stack);
}
```

Akan memberikan hasil seperti pada gambar di bawah ini :



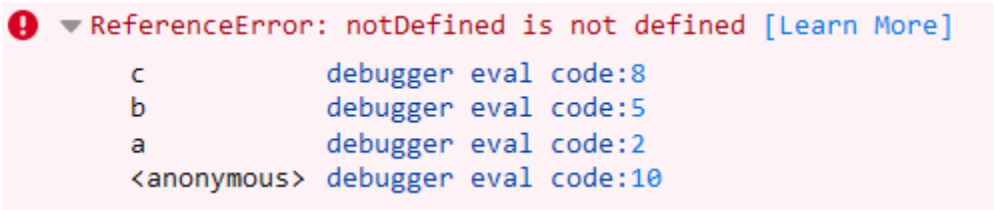
Gambar 232 property stack result

Property **`stack`** memberitahukan kita lokasi baris kode dan kolom kode tempat terjadinya *error* yaitu baris kode kedua dan kolom ketiga.

Perhatikan kode di bawah ini :

```
a = () => {
  b();
}
b = () => {
  c();
}
c = () => {
  notDefined();
}
a();
```

Jika kita eksekusi kode di atas, maka kita akan mendapatkan *reference error* lengkap dengan informasi *stack trace* yang terjadi :



Gambar 233 Stack

Pada *stack trace* di atas kita dapat mengetahui tempat *error* yang terjadi yaitu pada *function* **`c()`**, dan runutan pemanggilan fungsi yang terjadi. Pada error di atas kita dapat melihat baris kode tempat terjadinya *error*.

Finnaly

Apapun hasil dari **`try`** & **`catch`** *statement*, **`finnaly`** *statement* akan selalu mengeksekusi kode. Anda bisa membuktikanya dengan menulis kode di bawah ini :

```
try {
  let age = 25 //change to 1 or 6 or 40 or string or {}
  if (age === undefined) throw "age undefined!"
  if (age < 2) throw "reject boolean!"
  if (age < 10) throw "too young!"
  if (age > 35) throw "too old!"
  if (typeof age === 'string') throw "not a number!"
  if (typeof age === 'null') throw "not a number!"
  if (typeof age === 'object') throw "not a number!"
} catch (ex) {
  console.log('Error : ' + ex);
}
finally {
  console.log("Always executed");
}
```

Silahkan ubah nilai variabel **`age`** menjadi :

1. *literal number* yaitu **`1`**, **`6`** atau **`50`**
2. *literal boolean* yaitu **`true`** or **`false`**
3. *literal string* yaitu **`'he who is humble shall be raised'`**
4. *literal null* yaitu **`null`**
5. *literal object* yaitu **`[]`** atau **`{}`**

Anda akan selalu melihat statement di dalam finnaly akan selalu di eksekusi.

**Link* sumber kode.

5. *Custom Error*

Selain mendapatkan *built-in error* dari *javascript engine* kita juga dapat membangun *custom error* sendiri, kita akan membuat sebuah studi kasus.

Pada **Wallet** *function* di bawah ini kita membutuhkan 1 *parameter* dengan *identifier* **id**, kita ingin agar *parameter* yang diberikan adalah *primitive number*. Jika *parameter* yang diberikan adalah **undefined** atau kosong maka program harus melakukan *trigger error*.

Bagaimana cara melakukanya?

```
  function Wallet(id) {
    this.id = id
  }
```

Kita akan membuat *function* baru yang akan kita gunakan untuk melakukan validasi dan memberikan *custom error* jika *input* yang diberikan tidak sesuai dengan keinginan kita.

```
  function walletValidation(id) {
    if (id === undefined) {
      try {
        throw new Error("Wallet Validation Error: Cant Create Obj
ect Withoud ID !");
      } catch (err) {
        console.log(err.message);
        return false
      }
    }
    return true
  }
```

Pada kode di atas kita membuat *custom error* jika *input* yang diberikan adalah **undefined**, maka *statement* di bawah ini akan dieksekusi :

```
  throw new Error("Wallet Validation Error: Input ID must be numb
er !");
```

Kita menggunakan *keyword* **throw** dan *object error* agar bisa membuat *customer error*.

Jika sudah membuat kode di atas, kita akan menambahkan fungsi **walletValidation()** ke dalam **Wallet()** *function* sebagai *validator* :

```
function Wallet(id) {
  if (walletValidation(id)) {
    this.id = id
  }
}
```

Sekarang kita akan mengujinya :

```
var mywallet = new Wallet()
console.log(mywallet.id); // undefined
```

Jika kita membuat *object* **Wallet** tanpa memberikan *parameter* maka kita akan mendapatkan pesan *error* dan *return undefined,* seperti pada gambar di bawah ini :

```
Wallet Validation Error: Cant Create Object Withoud ID !
undefined
```

Gambar 234 Custom Validation Error 1

Selamat anda berhasil membuat *custom error* sendiri, jika kita memberikan *parameter number* maka kita juga berhasil membuat *object* baru bernama **mywallet**. Akses terhadap *property* pun berhasil :

```
var mywallet = new Wallet(99)
console.log(mywallet.id); // 99
```

Bagaimana jika user memberikan *input string*? Bagaimana cara kita untuk memvalidasinya? Tentu kita harus melakukan *upgrade* kemampuan fungsi **`walletValidation()`** dengan cara menambahkan *validator* untuk *string* :

```
    else if (typeof id === 'string') {
      try {
        throw new Error("Wallet Validation Error: Input ID must b
e number !");
      } catch (err) {
        console.log(err.message);
        return false;
      }
    }
    return true
```

Kita akan mengujinya :

```
var mywallet = new Wallet('2000')
console.log(mywallet.id); // undefined
```

**Link* sumber kode.

Jika kita membuat *object* **`Wallet`** dengan memberikan *parameter string* maka kita akan mendapatkan pesan *error* dan *return undefined,* seperti pada gambar di bawah ini :

```
Wallet Validation Error: Input ID must be number !
undefined
```

Gambar 235 Custom Validation Error 2

Subchapter 7 – Object

Good code is its own best documentation.

—Steve Mconnel

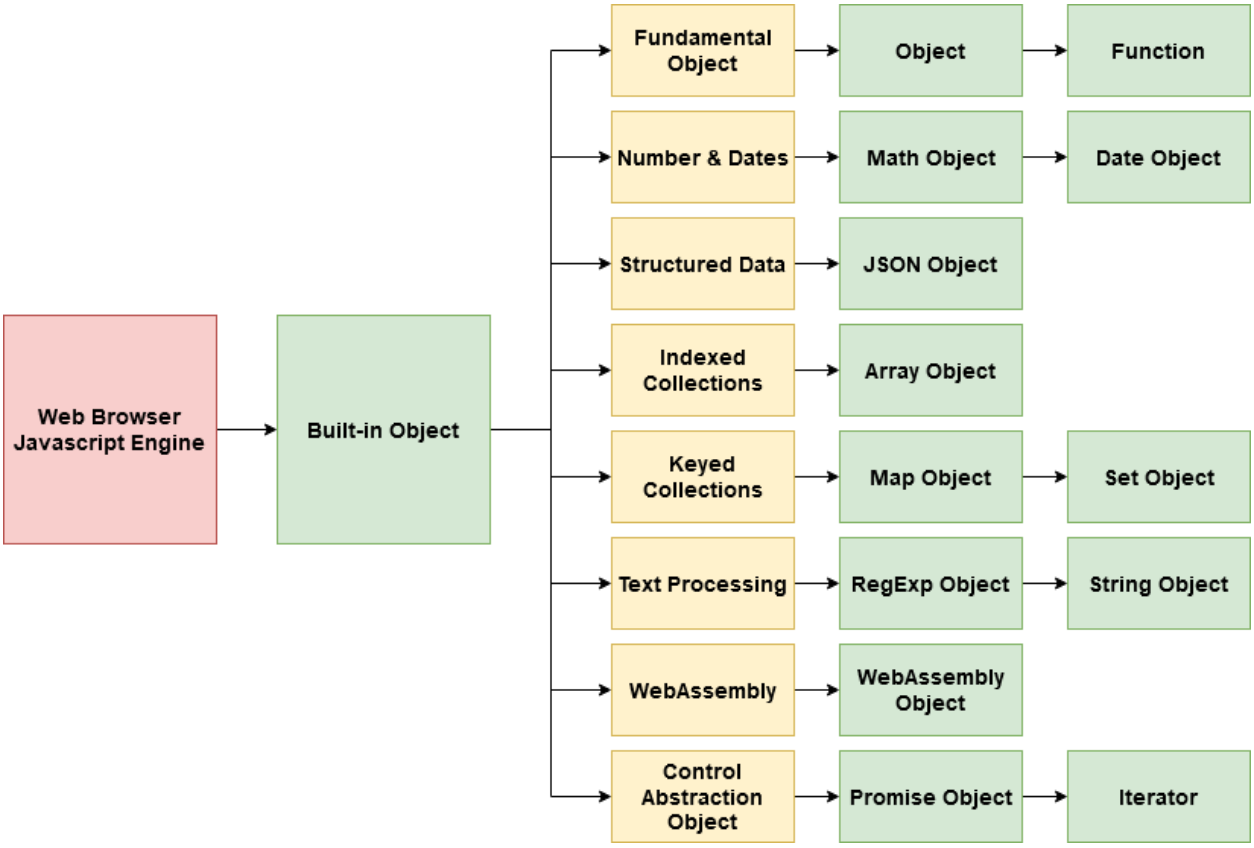
Subchapter 7 – Objectives

- Memahami Apa itu ***Fundamental Object***?
- Memahami Apa itu ***Object Initializer***?
- Memahami Apa itu ***Object Property***?
- Memahami Apa itu ***Object Method***?
- Memahami Apa itu ***Object Constructor***?
- Memahami Apa itu ***Object Prototype***?
- Memahami Apa itu ***Getter & Setter***?
- Memahami Apa itu ***JSON***?

Jika pada bab ini anda lupa apa itu definisi *object*? Maka anda harus kembali ke *chapter* sebelumnya agar bisa melanjutkan *chapter* ini dengan baik.

Sebelumnya kita telah mempelajari apa itu *object* dalam konteks *data types*, namun sebelum mengeksplorasi pembuatan dan pemanfaatan *object*. Tahukah anda definisi *object* benar-benar sangat ambigu dalam *javascript* jika kita amati secara detail.

Object yang telah kita pelajari sebelumnya adalah sebuah *Fundamental Object* yang menjadi bagian dari ***built-in Object*** dalam sebuah *javascript engine*.



Gambar 236 Built in-object

1. Apa itu *Fundamental Objects*?

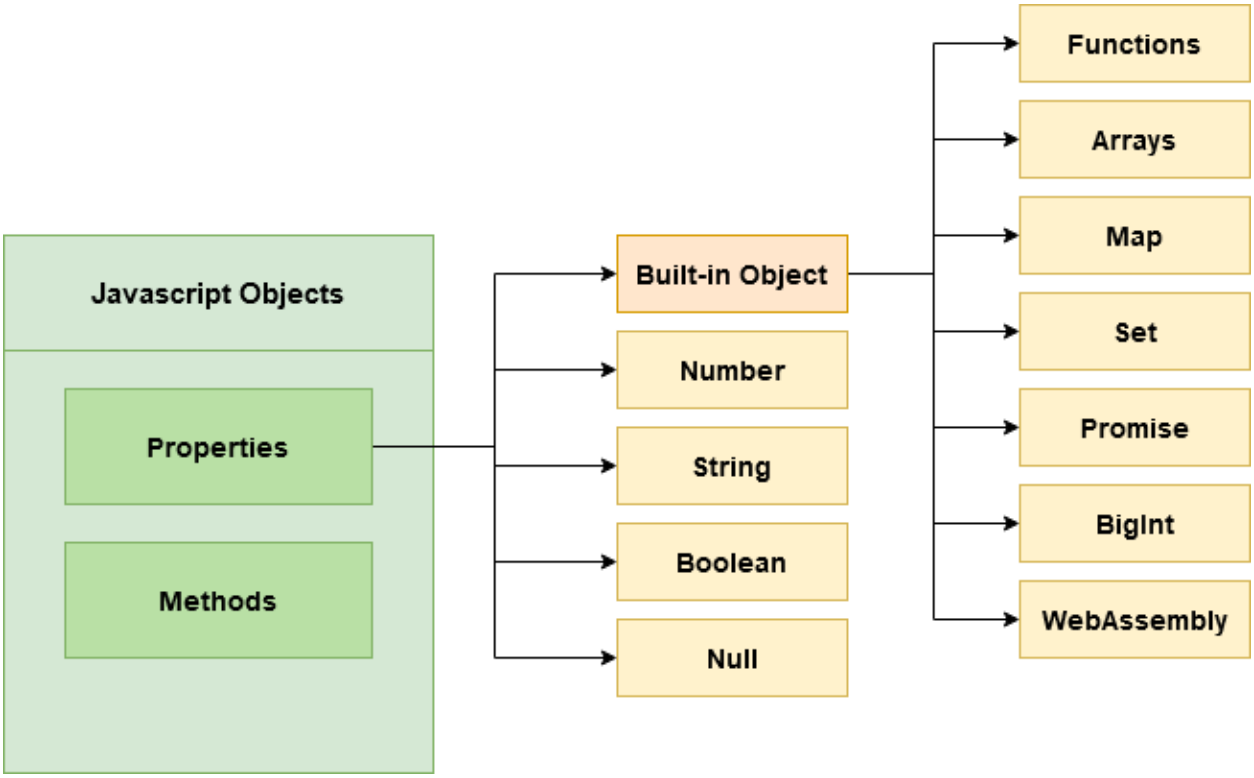
Pada dasarnya segala sesuatu yang ada didalam *javascript* adalah sebuah *object*, namun begitu dalam *javascript* terdapat *fundamental object(s)* yang menjadi dasar *object* semua *objects* yang ada di dalam *javascript*. Di antaranya adalah :

1. *Object*
2. *Function*
3. *Boolean*
4. *Symbol*
5. *Error*
6. *EvalError*
7. *InternalError*
8. *RangeError*
9. *ReferenceError*
10. *SyntaxError*
11. *TypeError*
12. *URIError*

Sebelumnya kita telah mempelajari cara membuat *function object* menggunakan *function constructor*. *Function* adalah salah satu dari bagian *fundamental object* yang dimiliki oleh *javascript*.

2. *Custom Object*

Saya tegaskan sekali lagi pada dasarnya segala sesuatu yang ada didalam *javascript* adalah sebuah *object*, meskipun begitu kita tetap mempunyai kesempatan untuk membuat *custom object* buatan kita sendiri.



Gambar 237 Custom Object Possibilities

Ada tiga cara membuat *custom object*, menggunakan *Object Initializer, Object Constructor,* dan *function style* :

Object Initializer

Object Initializer atau notasi *initializer* adalah cara membuat *object* tanpa menggunakan *constructor*, *object* dibuat dengan gaya *literal* di dalam kurung kurawal (*curly brace*) dan sering kali disebut dengan *object literal*.

Buka *web console* dengan menekan tombol **CTRL+SHIFT+K** :

```
>> var cantik = {firstname:'Maudy', lastname:'Ayunda Faza', age:23, haircolor:'black'}
<- undefined
>> cantik
<- {…}
     age: 23
     firstname: "Maudy"
     haircolor: "black"
     lastname: "Ayunda Faza"
   ▶ <prototype>: Object { … }
```

Gambar 238 Object Initializer

Key & Value

Object dapat memiliki sebuah *property* yang memiliki ***key & value***, pada kasus di atas *object* memiliki *keys* yaitu :

1. `age`
2. `firstname`
3. `haircolor`
4. `lastname`.

Setiap *keys* memiliki *values* yaitu :

1. `23`
2. `“maudy”`
3. `“black”`
4. `“Ayunda Faza”`.

Value dapat berupa *function*, *primitive* atau *object* lagi.

Object Property

Pada gambar di bawah ini kita membuat sebuah *object* bernama cantik dengan *properties* `firstname`, `lastname`, `age` dan `eyecolor`.

```
>> var mod = { firstname: 'Maudy', lastname: 'Ayunda Faza', age: 23, haircolor: 'black' }
<- undefined
>> mod
<- ▶ Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 23, haircolor: "black" }
```

Gambar 239 Object Properties

Object Method

Selain membuat *properties* dalam *object*, kita juga bisa membuat *function* di dalam *object*. Di bawah ini kita membuat *function* dengan *identifier* **`fullname`** di dalam *object*.

```
>> var mod = { firstname: 'Maudy', lastname: 'Ayunda Faza', age: 23, haircolor: 'black',
   fullname: function () { console.log("Maudy Ayunda Faza"); } }
<- undefined
>> mod.fullname()
   Maudy Ayunda Faza                                    debugger eval code:1:119
<- undefined
```

Gambar 240 Object Method

Object Constructor

Selain membuat *object* dengan *literal* kita juga bisa membuat *object* dengan memanfaatkan *constructor*. *Keyword* **`new`** digunakan untuk membuat *constructor*. *Constructor* memiliki ciri menggunakan huruf kapital.

```
>> var person = new Object()
<- undefined
>> person.firstname = "Maudy"
<- "Maudy"
>> person.lastname = "Ayunda Faza"
<- "Ayunda Faza"
>> person.age = 23
<- 23
>> person.eyecolor = "black"
<- "black"
>> person
<- ▶ Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 23, eyecolor: "black" }
```

Gambar 241 Object Constructor

Seperti yang telah kita pelajari sebelumnya pembuatan *object* dengan cara seperti ini tidak disarankan.

Function Constructor

Untuk membuat ***object*** lebih ringkas kita bisa melakukanya dengan *function style* atau biasa disebut dengan *function constructor*. Sebelumnya kita membuat *object* dalam satu *statement,* ada cara lain membuat *object* dengan memanfaatkan *function* dan **`this`** *keyword* :

```
>> function Person(firstname, lastname, age, eyecolor) {
     this.firstname = firstname;
     this.lastname = lastname;
     this.age = age;
     this.eyecolor = eyecolor;
   }
<- undefined
>> var maudy = new Person("Maudy", "Ayunda Faza", 23, "Black")
<- undefined
>> maudy
<- ▶ Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 23, eyecolor: "Black" }
```

Gambar 242 Function Style

Apa itu Mutable?

Javascript object bersifat *mutable* artinya *properties* dapat diubah, sebagai contoh pada gambar di bawah ini kita akan mengubah umur maudy.

```
» maudy
← Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 23, eyecolor: "Black" }
» maudy.age = 22
← 22
» maudy
← Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 22, eyecolor: "Black" }
```

Gambar 243 Mutable

Object Prototype

Sebelumnya kita membuat *object* menggunakan *function style*, namun kita tidak bisa menambahkan *property* yang baru. Hal ini membuat penulisan kode menjadi tidak ***expressive***, mari kita buktikan :

```
» function Person(firstname, lastname, age, eyecolor) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
    this.eyecolor = eyecolor;
  }
← undefined
» Person.skill = "singing";
← "singing"
» var maudy = new Person("Maudy", "Ayunda Faza", 23, "Black")
← undefined
» maudy.skill
← undefined
```

Gambar 244 Reason Prototype

Pada gambar di atas kita hendak menambahkan *property* **`skill`** pada *object constructor* yang dibangun dengan *function style*. Saat kita membuat *instance object* dari **`Person`** dan akses properti **`skill`** maka hasilnya **`undefined`**.

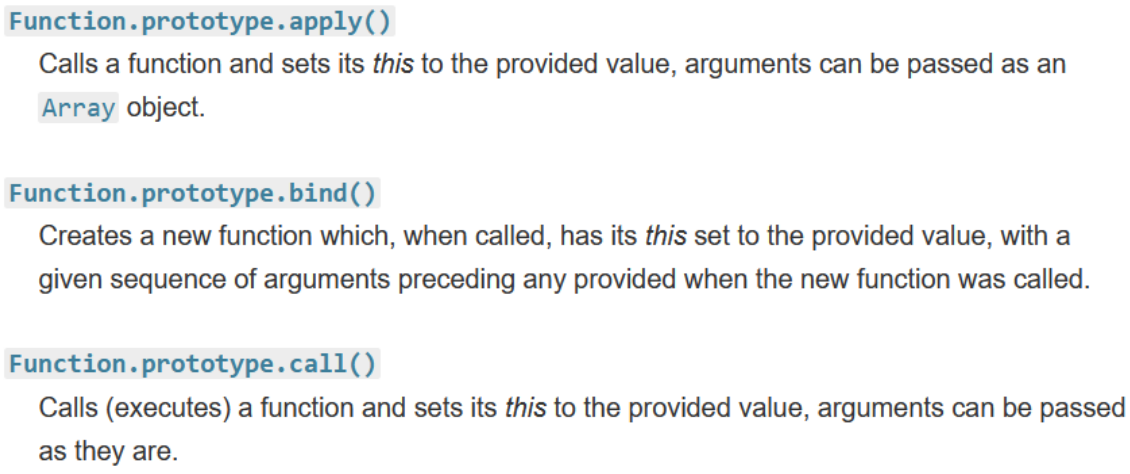
Dari permasalahan tersebut ada masanya kita ingin menambah *properties* atau *methods* pada sebuah *object constructor* atau menambah *properties* atau *methods* pada seluruh *object* yang telah dibuat. *Javascript* memberikan solusi atas permasalahan ini dengan konsep *prototype*.

```
function Person(firstname, lastname, age, eyecolor) {
   this.firstname = firstname;
   this.lastname = lastname;
   this.age = age;
   this.eyecolor = eyecolor;
 }

 Person.prototype.skill = "singing"; // <-- Prototype

 var maudy = new Person("Maudy", "Ayunda Faza", 23, "Black")

 console.log(maudy.skill); //singing
```

**Link* sumber kode.

Dalam *javascript*, seluruh *object* akan menerima atau mewarisi (***Inherit***) *methods* dan *properties* dari sebuah ***Prototype***.

Sebagai contoh ***Function Object*** akan mewarisi *Function.prototype* :



Gambar 245 Function Object Prototype

Sehingga setiap kali kita membuat *function object* maka kita akan memiliki 3 *method* di atas (*apply*, *bind* & *call*) yang telah kita pelajari sebelumnya pada *chapter* tentang *function*.

Getter & Setter

Fitur *Getter & Setter* diperkenalkan tahun 2009 dalam *EcmaScript* 5, dibuat untuk mempermudah kita agar dapat melakukan komputasi *properties* pada *object literal* :

```
var mod = {
  firstname: 'Maudy',
  lastname: 'Ayunda Faza',
  age: 23,
  haircolor: 'black',
  get getFirstName() {
    return this.firstname
  },
  set setFirstName(name) {
    this.firstname = name;
  }
}
```

Pada kode di atas untuk *getter* kita menggunakan *keyword* **get** dan untuk *setter* kita menggunakan *keyword* **set**. Untuk menggunakan *getter* eksekusi kode di bawah ini :

```
console.log(mod.getFirstName); // Maudy
```

Untuk menggunakan *setter* agar kita bisa mengubah *property* dalam *object* **mod**, eksekusi kode di bawah ini :

```
mod.setFirstName = 'Rindu';
console.log(mod.getFirstName); // Rindu
```

**Link* sumber kode.

Object Destructure

Javascript sudah mendukung operasi *destructure* untuk *object literal*, sebagai contoh kita memiliki *object* **mod** :

```
const mod = {
  firstname: "Maudy",
  lastname: "Ayunda Faza",
  age: 23,
  haircolor: "black"
};
```

Untuk melakukan operasi *extract properties* atau *destructure* yang dimiliki oleh *object* **mod** tulis kode di bawah ini :

```
const { firstname, age } = mod;
```

Untuk memeriksa hasilnya tulis kode di bawah ini dan eksekusi :

```
console.log(firstname, age);
// Output Maudy 23
```

3. *Custom Object Property*

Sebuah *object* memiliki *property* yang dapat kita gunakan untuk menyimpan data. Di bawah ini kita membuat sebuah *object* dengan 2 *property* yaitu **`firstname`** dan **`age`**.

```
var person = {
  firstname: 'Gun Gun',
  age: 28,
};
```

**Link* sumber kode.

Jika kode di atas kita visualisasikan maka :

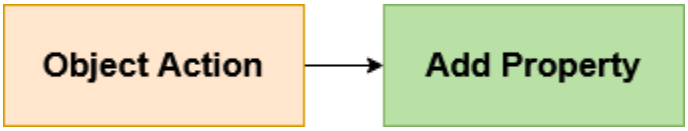
Gambar 246 Visualisasi Person Object

Object bersifat *programmable* artinya kita dapat :

1. Menambahkan *property* baru
2. Menghapus sebuah *property*
3. Mengakses sebuah *property*
4. Memeriksa sebuah *property*

Add Object Property

Selalu ingat setiap kali kita membuat sebuah *object* kita dapat melakukan sebuah *action* yaitu menambah suatu *property* :



Gambar 247 Add Property

Pada kode di bawah ini kita membuat sebuah *object* bernama **`person`**.

```
var person = {
  firstname: 'Gun Gun',
  lastname: 'Febrianza',
};
```

Seandainya kita lupa memberikan sebuah *property*, bagaimana menambahkan cara menambahkan *property* yang baru?

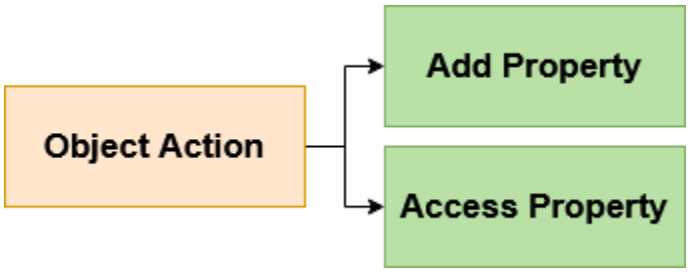
```
person.nationality = 'Indonesia';
console.log(person);
```

**Link* sumber kode.

Pada kode di atas kita menambahkan sebuah *property* baru. Pada kode di atas **`nationality`** adalah key dan **`'Indonesia'`** adalah ***value*** yang diberikan.

Access Object Property

Setelah kita menambahkan sebuah *property* baru kita juga dapat mengaksesnya.



Gambar 248 Access Property

```
var person = {
  firstname: 'Gun Gun',
  lastname: 'Febrianza',
  age: 28,
  eyecolor: 'Red Brown',
  'super loyal': true
};
```

Jika kita ingin mendapatkan nilai dari *key* **`firstname`** terdapat dua cara :

```
console.log(person.firstname + ' is ' + person.age + ' years old.');
// Gun Gun is 28 years old.
console.log(person['firstname'] + ' is ' + person['age'] + ' years old.');
// Gun Gun is 28 years old.
```

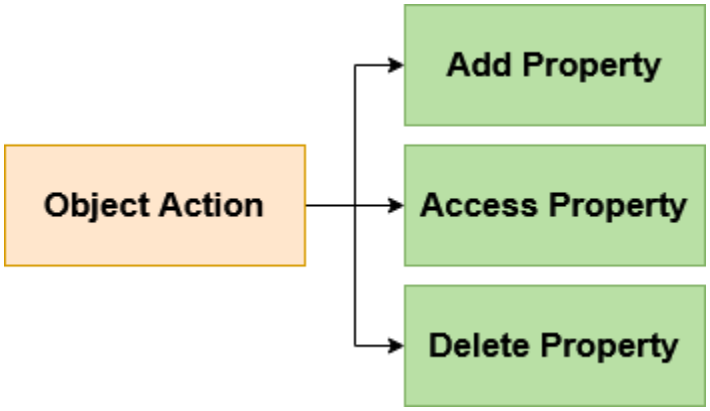
Lalu bagaimana cara mendapatkan nilai dari *key* **`‘super loyal’`** yang bersifat *multiword key*? Eksekusi kode di bawah ini

```

```

```
console.log(person['super loyal']);  // true
```

Delete Object Property

Selain kita menambahkan *property* baru, kita juga dapat menghapus *property* yang sudah kita buat.



Gambar 249 Delete Property

Pada kode di bawah ini kita membuat sebuah *object* bernama `person`.

```
var person = {
  firstName: 'Gun Gun',
  lastName: 'Febrianza',
  age: 28,
};
```

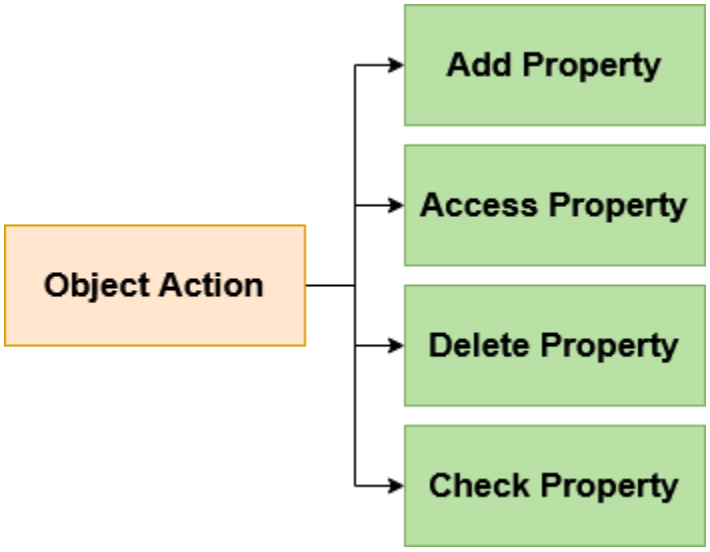
Jika kita ingin menghapus *property* `age`, maka kita perlu mengeksekusi perintah di bawah ini :

```
delete person.age;
console.log(person);
```

**Link* sumber kode.

Check Object Property

Selain kita menambahkan *property* baru, kita juga dapat menghapus *property* yang sudah kita buat.



Gambar 250 Check Property

Pada kode di bawah ini kita membuat sebuah *object* bernama `person`.

```
var person = {
  firstName: 'Gun Gun',
  lastName: 'Febrianza',
  language: 'en',
};
```

Jika kita ingin memeriksa apakah suatu *object* memiliki *property* atau tidak, kita dapat menggunakan *keyword* in seperti pada kode di bawah ini :

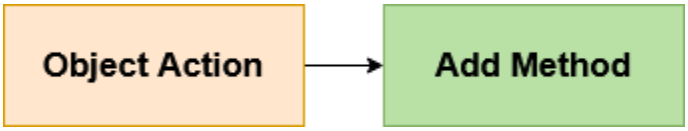
```
console.log('language' in person);// true
console.log('languages' in person);// false
```

Atau memeriksanya menggunakan *conditional if statement* :

```
if ('languages' in person === false) {
  console.log('Properties is not exist');
}
```

**Link* sumber kode.

4. *Custom Object Method*



Gambar 251 Add Object Method Action

Selain membuat *property* dalam *object* kita juga dapat membuat sebuah *method* dalam *object*. Pada kode di bawah ini kita membuat sebuah *method* dengan *identifier* `fullName()` :

```
var person = {
  firstName: "Gun Gun",
  lastName: "Febrianza",
  fullName() {
    return this.firstName + " " + this.lastName;
  }
};
```

Pada kode di atas kita membuat *object* `person` yang memiliki *property* `firstName`, `lastName` dan memiliki *method* `fullName()`. Saran untuk membuat *object method* ini mengikuti konvensi standar *Ecma Script Linter* yang memiliki pengaturan *object-shorthand*.

Access Object Method

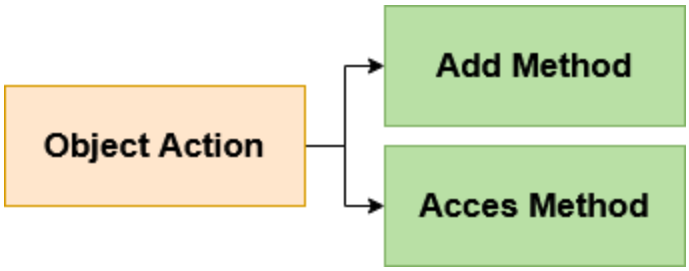
Jika kita ingin memanggil *method* `fullName()` yang dimiliki oleh *object* `person` maka perhatikan kode di bawah ini :

```
console.log(person.fullName());
// Output : Gun Gun Febrianza
```

*Link sumber kode.

Kita menggunakan `console.log()` karena *keyword* `return` dalam *method* `fullName()` tidak akan mencetak *output*.

Add Object Method



Gambar 252 Access Object Method Action

Kita juga dapat menambahkan *method* baru pada sebuah *object*, pada kode di bawah ini kita mencoba menambahkan *function* baru dengan *identifier* `name` :

```
person.name = function() {
  return this.firstName + " " + this.lastName;
};
```

Untuk memanggil *method* yang baru kita buat barusan, eksekusi kode di bawah ini :

```
console.log(person.name());
// Output Gun Gun Febrianza
```

*Link sumber kode.

5. *Custom Object Looping*

Ada kalanya kita ingin melakukan operasi looping pada suatu object literal, baik itu untuk mendapatkan seluruh property ataupun value yang dimilikinya. Untuk melakukanya perhatikan kode di bawah ini :

```
var person = {
  fname: "Gun Gun",
  lname: "Febrianza",
  age: 28
};
```

Untuk mendapatkan *list key* dari *object literal* :

```
var x;
for (x in person) {
  console.log(x);
}
//fname
//lname
//age
```

Untuk mendapatkan *list value* dari *object literal* :

```
var x;
for (x in person) {
  console.log(person[x]);
}
//Gun Gun
//Febrianza
//28
```

**Link* sumber kode.

6. *JSON*

JSON adalah singkatan dari ***Javascript Object Notation***. *Syntax* pada *JSON* digunakan untuk menyimpan dan kegiatan bertukar data antara *browser* dengan *server* atau sebaliknya.

Kita dapat mengubah sebuah *javascript object* menjadi sebuah *JSON* dan mengirimkan data *JSON* tersebut menuju *server*. Sebaliknya kita juga dapat mengubah *JSON* yang diterima dari *server* untuk diubah kedalam *javascript object*.

JSON & Object Literal

Jika kita tidak teliti Notasi *object literal* dan *JSON* tidaklah sama, perbedaanya terletak pada pembuatan *property*. Pada *JSON property* diharuskan menggunakan *double-quote*, nilai yang bisa digunakan dalam *JSON* adalah *strings, numbers, arrays, true, false, null*, atau *JSON object* lainya.

Di bawah ini adalah *object literal* :

```
{ name: "Gun Gun Febrianza" }
```

Jika representasi *object literal* di atas di ubah kedalam *JSON* maka ini hasilnya :

```
{ "name": "Gun Gun Febrianza" }
```

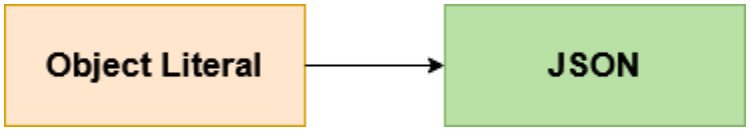
Ada perbedaan yang cukup signifikan antara *Object Literal* dan *JSON*, pada *JSON* kita memiliki aturan yang berbeda yaitu :

1. Penggunaan *function* sebagai *value* untuk *properties* tidak diizinkan di dalam *JSON*.
2. Penggunaan *date object* sebagai *value* untuk *properties* tidak diizinkan di dalam *JSON*.

3. Penggunaan *undefined* sebagai *value* untuk *properties* tidak diizinkan di dalam *JSON*.

Stringify

Proses untuk mengubah *object literal* ke dalam *JSON* seringkali disebut dengan *stringify*. Di bawah ini adalah *object literal* yang akan dikonversi ke dalam *JSON*.



Gambar 253 Transform Object Literal to JSON

Pada kasus di dunia nyata jika kita ingin mengirimkan data berbasis *object literal* ke *server* kita perlu mengubahnya terlebih dahulu ke dalam *JSON* sebagai *data exchange format* yang efisien.

```
var objectLiteral = {
  name: "Gun Gun Febrianza",
  age: 28,
  city: "Bandung"
};
```

Untuk mengubah *object literal*, kita akan menggunakan ***built-in object*** yang dimiliki oleh *javascript engine* yaitu ***JSON object***. Pada *JSON object* kita dapat menggunakan *method* `stringify()` untuk mengubah *object literal* ke dalam *JSON* :

```
var JSONData = JSON.stringify(objectLiteral);
```

Setelah di ubah ke dalam *JSON*, tipe data akan berubah menjadi string dan ini adalah hasilnya :

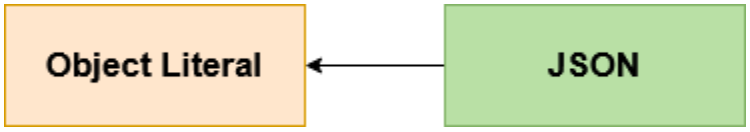
```
console.log(typeof JSONData) // string
console.log(JSONData)
```

```
// {"name":"Gun Gun Febrianza","age":28,"city":"Bandung"}
```

**Link* sumber kode.

Parse JSON

Proses untuk mengubah *JSON* ke dalam *object literal* seringkali disebut dengan *parsing JSON*. Di bawah ini adalah *JSON* yang akan dikonversi ke dalam *object literal*.



Gambar 254 Transform JSON into Object Literal

Pada kasus di dunia nyata kita ingin menerima data berbasis *JSON* dari *server* kita dan kita perlu mengubahnya ke dalam *object literal* agar bisa diproses oleh aplikasi *client*.

```
var JSONData = '{"name":"Gun Gun Febrianza","age":28,"city":"Ba
ndung"}'
```

Kita asumsikan variabel **`JSONData`** mendapatkan data *JSON* dari *server*, untuk mengubahnya kembali ke dalam *object literal* gunakan *method* **`parse()`**yang dimiliki oleh *JSON Object* :

```
var objLiteral = JSON.parse(JSONData);
```

Setelah di ubah ke dalam *object literal*, tipe data akan kembali menjadi *object* dan ini adalah hasilnya :

```
console.log(typeof objLiteral);
console.log(objLiteral); // object
```

```
  // { name: 'Gun Gun Febrianza', age: 28, city: 'Bandung' }
```

*Link sumber kode.

Parse Date in JSON

Di bawah ini adalah strategi untuk mengatasi data tahun, bulan dan tanggal yang ada di dalam *JSON*.

```
  var data = '{"name":"Gun", "born":"1992-12-
14", "city":"Bandung"}';
```

Di bawah ini adalah strategi untuk mengatasi data tahun, bulan dan tanggal yang ada di dalam JSON. Lakukan *parsing JSON* terlebih dahulu ke dalam *object literal* :

```
  var obj = JSON.parse(data);
```

Kemudian akses kembali *property* tempat data tanggal, bulan dan tahun, ubah data *string* tersebut ke dalam ***date object*** dengan *syntax* ***new Date()*** seperti kode di bawah ini :

```
  obj.born = new Date(obj.born);
  console.log(obj.born.getFullYear()) //1992
```

*Link sumber kode.

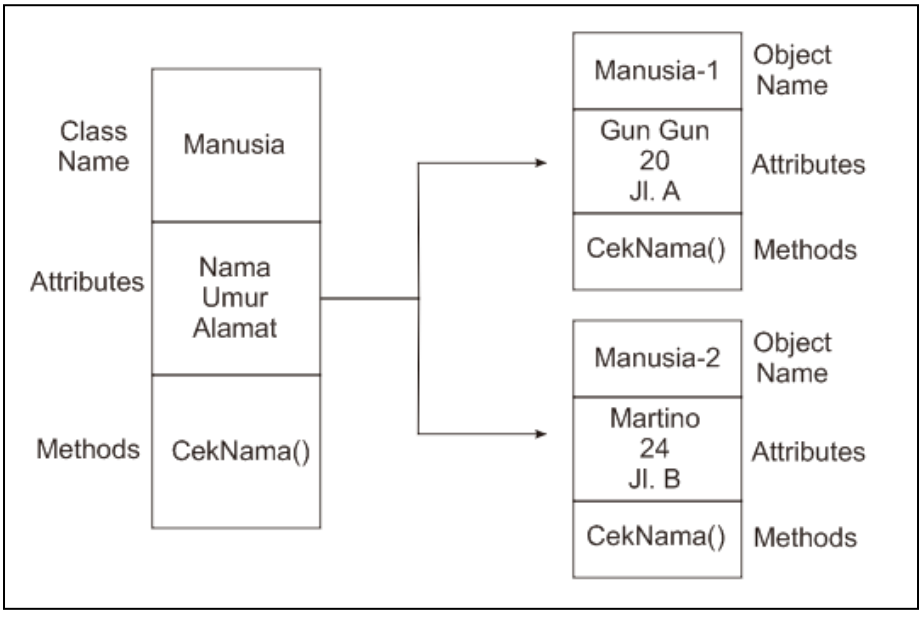
Subchapter 8 – Classes

Learning to code is learning to create and innovate.
—Enda Kenny

Subchapter 8 – Objectives

- Memahami Apa itu ***Class-based Language***?
- Memahami Apa itu ***Class Declaration?***
- Memahami Apa itu ***Class Constructor?***
- Memahami Apa itu ***Class Static Method?***
- Memahami Apa itu ***Class Getter & Setter?***
- Memahami Apa itu ***Class Expression?***
- Memahami Apa itu ***Class Inheritance?***
- Memahami Apa itu ***Method Override?***
- Memahami Apa itu ***Constructor Override?***

Class adalah sebuah *template* atau *blueprint* dari *object* yang akan dibuat. Sebuah kelas menjelaskan seluruh atribut sebuah *objects*, termasuk *methods* yang akan menentukan sifat dari *member objects*.



Gambar 255 Class Illustration

Konsep *class* dalam *javascript* diperkenalkan pertama kali dalam *EcmaScript* 2015. Diciptakan untuk mempermudah kita dalam membuat sebuah *object* dan membantu kita untuk membangun *design pattern code* yang baik.

Javascript adalah *prototype-based language* dan konsep *class* sering kali disebut dengan *syntactical sugar* dari *function constructor*. Maksud dari *syntactical sugar* adalah kita memiliki alternatif yang lebih baik, *syntax* yang *human readable* dan lebih *expressif* dalam hal ini untuk membuat sebuah *object*.

1. *Class-based language*

Konsep *class* menyediakan *syntax* yang merepresentasikan sebuah *prototypal inheritance* dalam paradigma pemrograman berbasis *class*. Kita akan belajar bagaimana cara mentransformasikan sebuah *function constructor* menggunakan *classes*.

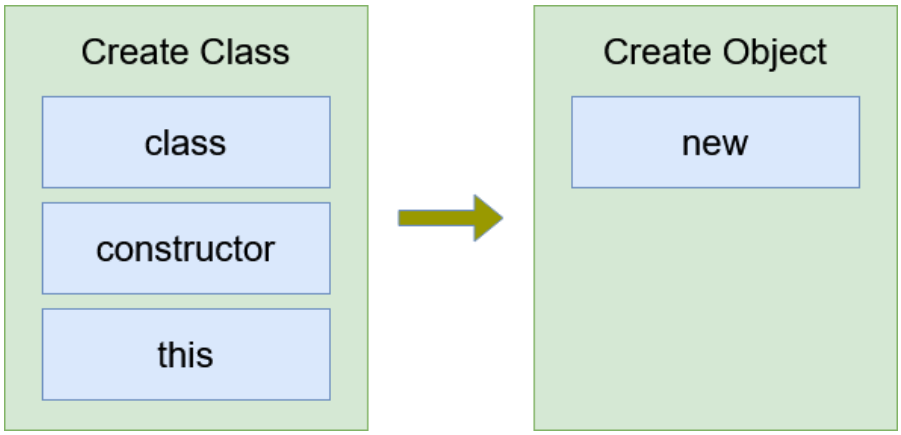
Pada kode di bawah ini kita membuat sebuah *function constructor* untuk **`person`** dan menambahkan sebuah *method* **`getName`** ke dalam *prototype* :

```
 function person(firstname, lastname, age, eyecolor) {
  this.firstname = firstname;
  this.lastname = lastname;
  this.age = age;
  this.eyecolor = eyecolor;
}

person.prototype.getName = function() {
  return this.firstname + " " + this.lastname;
};
```

Untuk mentrasformasikan *function constructor* di atas ke dalam *class* perhatikan kode di bawah ini :

```
class person {
  constructor(firstname, lastname, age, eyecolor) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
    this.eyecolor = eyecolor;
  }
  getName() {
    return this.firstname + " " + this.lastname;
  }
}
```

Jika kita perhatikan dari kode di atas untuk transformasi ke dalam sebuah *class* kita membutuhkan 3 *keywords*, yaitu **`class`**, **`constructor`** dan **`this`** *keyword*.



Gambar 256 Create Class & Object

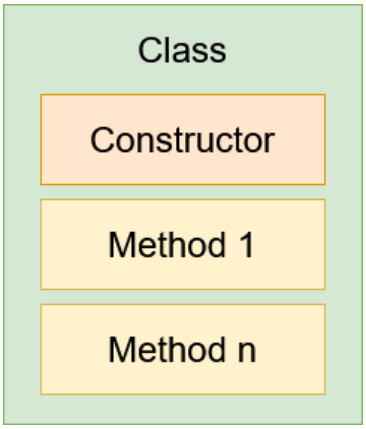
Setelah sebuah *class* dibuat kita dapat membuat sebuah *object*, masih dengan cara yang sama menggunakan *keyword* **`new`** untuk membuat sebuah *object*.

Untuk membuat *object* baru menggunakan *function constructor* ataupun *class* tulis kode di bawah ini :

```
var hooman = new person("Gun Gun", "Febrianza", 28, "brown");
console.log(hooman.getName());
//Gun Gun Febrianza
```

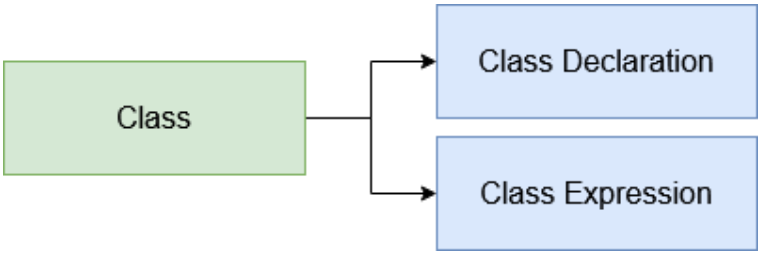
**Link* sumber kode.

Di dalam sebuah *class* kita dapat membuat sebuah *constructor* dan sekumpulan *methods* :



Gambar 257 Class Structure

Terdapat dua cara untuk membuat sebuah *class*, yaitu menggunakan *class declaration* atau *class expression* :



Gambar 258 Class Declaration & Expression

2. Class Declaration

Kita akan membuat sebuah *class* menggunakan *class declaration* :

```
class wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
}
```

Untuk menggunakan *class* tersebut perhatikan kode di bawah ini :

```
var a = new wallet(1,2000)
console.log(a);
//wallet { id: 1, balance: 2000 }
```

**Link* sumber kode.

Strict Mode

Setiap *statements* yang ditulis di dalam *class body* akan di eksekusi dengan *strict mode* untuk meningkatkan performa.

Constructor

Constructor adalah sebuah *method* spesial yang digunakan untuk membuat *object* di dalam *class*.

Sebuah *class* hanya memiliki *constructor* tunggal, membuat *constructor* lebih dari satu akan memproduksi *syntax error*. Jika kita tidak membuat *constructor* di dalam *class* maka *default constructor* akan diberikan secara otomatis.

Jika kita ingin memiliki *constructor* dengan *parameter* berupa sekumpulan *arguments* secara *arbitrary* tanpa batas gunakan *spread syntax* seperti kode di bawah ini :

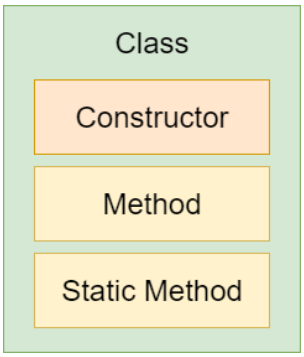
```
class Log {
  constructor(...args) {
    console.log(args);
  }
}

new Log('data 1', 'data 2', 'data 3')
// Output [ 'data 1', 'data 2', 'data 3' ]
```

**Link* sumber kode.

Output yang diproduksi adalah *array of arguments*.

Static Method

Selain kita dapat membuat *constructor* dan *method* di dalam *class*, kita juga dapat membuat *static method*. Dengan *static method* kita dapat memanggil *method* yang dimiliki oleh suatu *class* tanpa perlu membuat representasi *object* dari suatu *class*.



Gambar 259 Static Method

Perhatikan kode di bawah ini, di dalam *class* **`wallet`** terdapat *static method* dengan nama **`getBalance().`** Untuk membuatnya kita akan menggunakan *keyword* **`static`** :

```
class wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
  static getBalance() {
    return 'Your Balance is 0';
  }
}
```

Untuk menggunakan *static method* perhatikan kode di bawah ini :

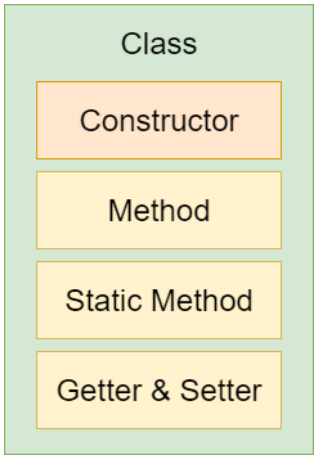
```
console.log(wallet.getBalance());
// Your Balance is 0
```

**Link* sumber kode.

Untuk memanggil *method* `getBalance()` tidak perlu membuat *object* terlebih dahulu.

Getter & Setter

Di dalam sebuah *class* kita dapat membangun *getter & setter* untuk melakukan komputasi pada *properties* dari *object* yang akan diciptakan.



Gambar 260 Getter & Setter

Selain dalam *object literal*, kita juga dapat menggunakan *getter & setter* di dalam *class* :

```
class Wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
  get Balance() {
    return this.balance;
  }
  set Balance(balance) {
    this.balance += balance;
  }
}
```

Pada kode di atas kita memiliki **`get`** & **`set`** untuk **`Balance`** untuk dapat berinteraksi dengan *getter & setter* kita harus membuat *object* dari *class* tersebut terlebih dahulu :

```
const duitku = new Wallet(1, 2000);
```

Di bawah ini contoh untuk menggunakan *setter* dari *object* **`duitku`** :

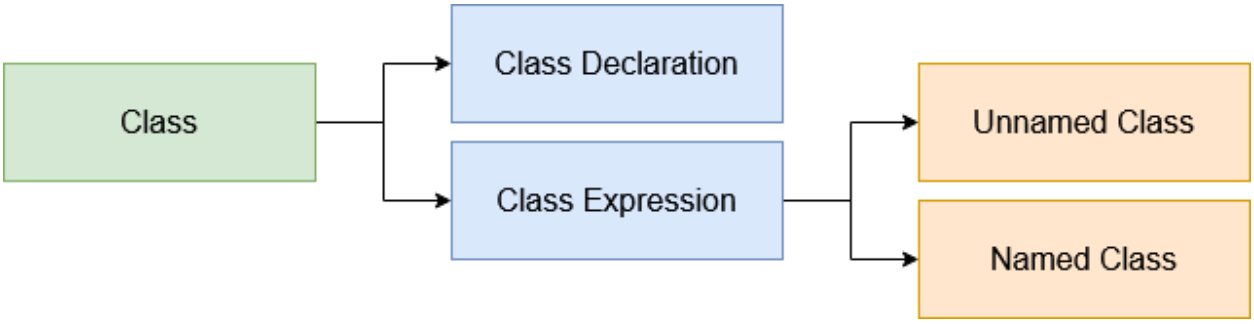
```
duitku.Balance = 2000;
```

Pada kode di atas kita memberikan nilai 2000 kepada *setter* untuk *object* **`duitku`** :

```
console.log(duitku.Balance); // 4000
```

3. Class Expression

Selain membuat *class* dengan *class declaration*, kita juga dapat membuat sebuah *class* dengan *class expression*. Terdapat dua *class expression* yaitu *unnamed class & named class*.



Gambar 261 Named & Unnamed Class

Unnamed Class

Di bawah ini adalah contoh membuat *class expression* yang berbasis *unnamed class* :

```
let mywallet1 = class {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
};

console.log(mywallet1.name); //mywallet1
console.log(typeof mywallet1); //function
```

**Link* sumber kode.

Pada kode di atas *class* tidak diberi nama sama sekali dan ditulis dengan gaya *assignment statement*. Variabel **`mywallet1`** menjadi tempat untuk menyimpan *class* yang secara *internal* adalah sebuah *function*.

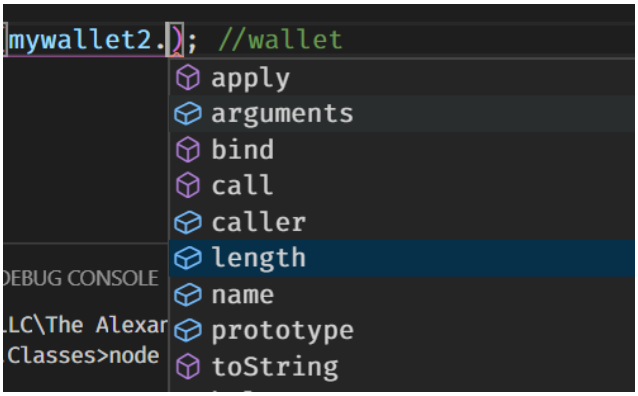
Named Class

Di bawah ini adalah contoh membuat *class expression* yang berbasis *named class* :

```
let mywallet2 = class wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
};
```

**Link* sumber kode.

Pada kode di atas *class* diberi nama **`wallet`** dan masih ditulis dengan gaya *assignment statement*. Sebuah *class* memiliki *property arguments*, *called*, *length* dan *name* :



Class Property

Sehingga jika kita ingin mengetahui nama sebuah *class* kita dapat membaca *property* yang dimiliki oleh *class* tersebut :

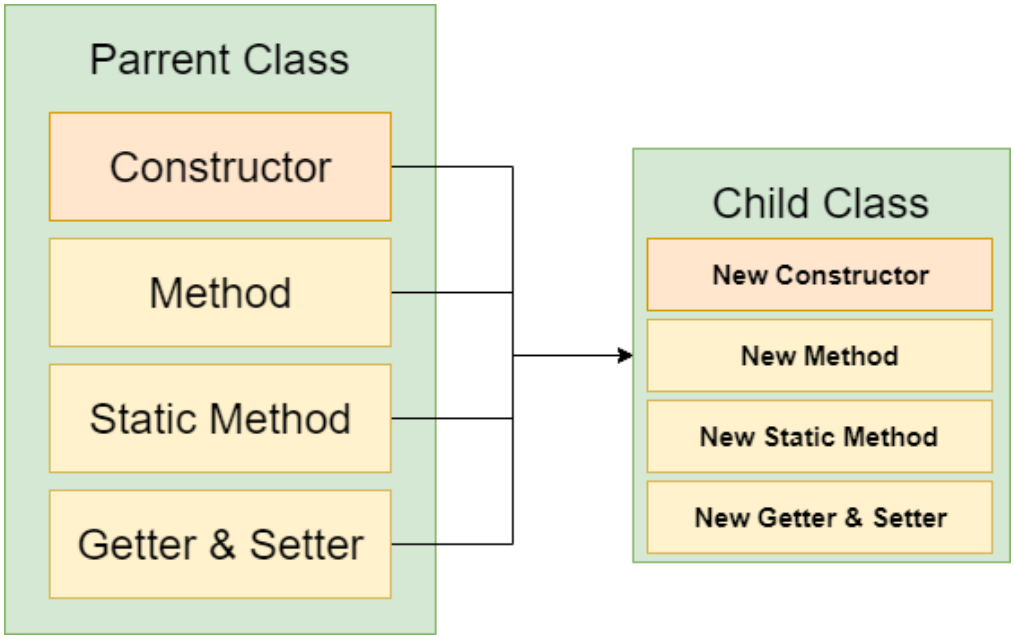
```
console.log(mywallet2.name); //wallet
```

Class yang merupakan representasi lain dari *function*, memiliki beberapa *method* yaitu *apply*, *bind*, *call* dan *toString* yang telah kita pelajari sebelumnya.

4. Class Inheritance

Secara teori *Class inheritance* adalah kemampuan suatu *class* untuk memberikan karakteristik yang dimilikinya pada *class* turunannya.

Secara teknis dalam pemrograman *javascript*, *property*, *method, static method, getter & setter* yang dimiliki oleh suatu *class* dapat diturunkan pada suatu *class* atau *child class* :



Gambar 262 Illustration of Inheritance

Kita akan belajar membuat sebuah *parent class* yang akan digunakan oleh sebuah *child class*.

Di bawah ini adalah sebuah *class* dengan nama **`Wallet`**, di dalamnya terdapat *constructor, method, static method, getter & setter* :

```
class Wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
```

```
    getID() {
      return this.id;
    }
    static getInfo() {
      return `this class contains property, method, static method
, getter & setter`;
    }
    get PropBalance() {
      return this.balance;
    }
    set PropBalance(balance) {
      this.balance += balance;
    }
  }
```

Jika kita ingin membuat *child class* yang memiliki karakteristik seperti yang dimiliki oleh *parent class*, maka kita perlu menggunakan *keyword* **`extend`** :

```
  class Ewallet extends Wallet {
    constructor(id, balance, type) {
      super(id, balance);
      this.type = type;
    }
  }
```

Pada kode di atas kita membuat *class* baru (*child class*) bernama **`Ewallet`** yang memiliki karakteristik dari *class* **`Wallet`**. Pada *class* yang baru di dalam *constructor* kita memiliki *keyword* **`super`** yang akan menerima *parameter* untuk *parent class*.

Kita akan membuktikannya dengan cara membaca *property* yang disimpan di dalam *parent class,* untuk melakukannya buat *object* **`Ewallet`** terlebih dahulu :

```
```

```
const mywallet = new Ewallet(1, 2000, "gopay");
console.log(mywallet.id); // Output : 1
console.log(mywallet.balance); // Output : 2000
```

Kita juga dapat mengakses *method* **`getID()`** yang dimiliki oleh *parent class* :

```
console.log(mywallet.getID()); // Output : 1
```

Kita juga dapat mengakses *static method* yang dimiliki oleh *parent class* :

```
console.log(ewallet.getInfo());
//this class contains property, method, static method, getter &
setter
```

Kita juga dapat menggunakan *getter & setter* yang dimiliki oleh *parent class* :

```
mywallet.PropBalance = 1000;
console.log(mywallet.PropBalance); //3000
```

**Link* sumber kode.

Method Override

Jika kita ingin mengganti atau menimpa *method* yang dimiliki oleh *parent class* maka kita dapat menggunakan *method overriding strategy*. Pada kode di bawah ini *parent class* memiliki *method* **`getID()`** :

```
class Wallet {
  constructor(id, balance) {
    this.id = id;
    this.balance = balance;
  }
```

```
  getID() {
    return this.id;
  }
}
```

Jika pada *child class* kita ingin memiliki *method* dengan nama yang sama namun fungsi yang berbeda, maka kita dapat mendeklarasikan ulang *method* **`getID()`** pada *child class* :

```
class Ewallet extends Wallet {
  constructor(id, balance, type) {
    super(id, balance);
    this.type = type;
  }
  getID() {
    return `Your id is ${this.id}`;
  }
}
```

Pada kode di atas *method* **`getID()`** pada *child class* akan menimpa fungsi milik *parent class*, mari kita buktikan dengan menulis dan mengeksekusi kode di bawah ini :

```
const mywallet = new Ewallet(1, 2000, "gopay");
console.log(mywallet.getID()); // Your id is 1
```

Lalu bagaimana jika kita ingin tetap memiliki fungsi *original* dari *parent class*?

```
class Ewallet extends Wallet {
  constructor(id, balance, type) {
    super(id, balance);
    this.type = type;
```

```
  }
  getID() {
    return `Your id is ${super.getID()}`;
  }
}
```

**Link* sumber kode.

Perhatikan pada *method* **`getID()`** di atas kita menggunakan *keyword* **`super`** untuk mengakses *method* **`getID()`** yang dimiliki oleh *parent class*.

Constructor Override

Selain melakukan *overriding* pada *method* kita juga dapat melakukan *override* pada *constructor*, jika anda sadari anda telah melakukannya sebelumnya. Pada kode di bawah ini kita membuat *constructor* pada *child class* yang menerima 3 *parameter* :

```
class Ewallet extends Wallet {
  constructor(id, balance, type) {
    super(id, balance);
    this.type = type;
  }
}
```

Dua *parameter* yaitu **`id`** dan **`balance`** akan di *passing* menuju *constructor* yang dimiliki oleh *parrent class* dengan memanfaatkan *keyword* **`super`**, sementara *parameter* **`type`** digunakan di dalam *constructor* milik *child class* yaitu **`Ewallet`**.

Subchapter 9 – Collection

Learning to write programs stretches your mind,
and helps you think better,
creates a way of thinking about things
that I think is helpful in all domains.

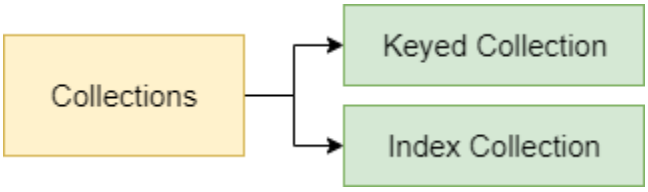
—Bill Gates

Subchapter 9 – Objectives

- Memahami Apa itu ***Collection***?
- Memahami Apa itu ***Iterable?***
- Memahami Apa itu ***Keyed?***
- Memahami Apa itu ***Destructurable?***
- Memahami Apa itu ***Indexed Collections?***
- Memahami Apa itu ***Array?***
- Memahami Apa itu ***Multidimensional-array?***
- Memahami Apa itu ***Keyed Collections?***
- Memahami Apa itu ***Map Collection?***
- Memahami Apa itu ***Set Collection?***

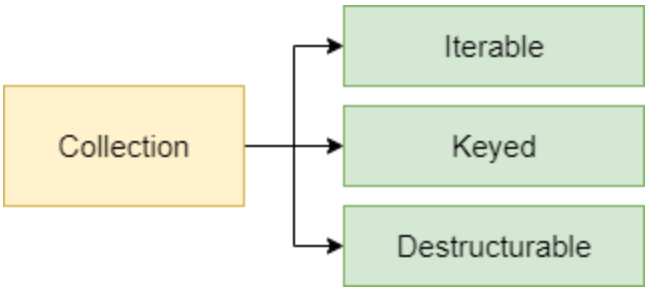
1. Apa itu *Collection*?

Collection adalah *object* tunggal yang merepresentasikan kumpulan kumpulan *object*. *Collections* juga dapat disebut sebagai *containers*[26]. Dalam *javascript* sendiri terdapat dua *collections*, yaitu ***Keyed Collections*** dan ***Indexed Collections***.



Gambar 263 Collections Type

Masing-masing memiliki kekurangan dan kelebihan. Pada *keyed collections* terdapat *map* dan *set object* dan pada *indexed collection* terdapat *array* dan *typed array*.



Gambar 264 Collection Properties

Terdapat 3 faktor utama yang menentukan *collection* sebelum menggunakannya untuk mengatasi masalah yang kita hadapi :

Iterable

Dapatkah kita melakukan *looping* pada *collection* secara langsung dan mendapatkan akses pada setiap data yang ada di dalamnya?

Keyed

Dapatkah kita mendapatkan suatu nilai dengan memanfaatkan *key* yang dimiliki oleh nilai tersebut?

Destructurable

Dapatkah kita dengan mudah dan cepat untuk mendapatkan potongan *collection* dari *collection* yang ingin kita dapatkan?

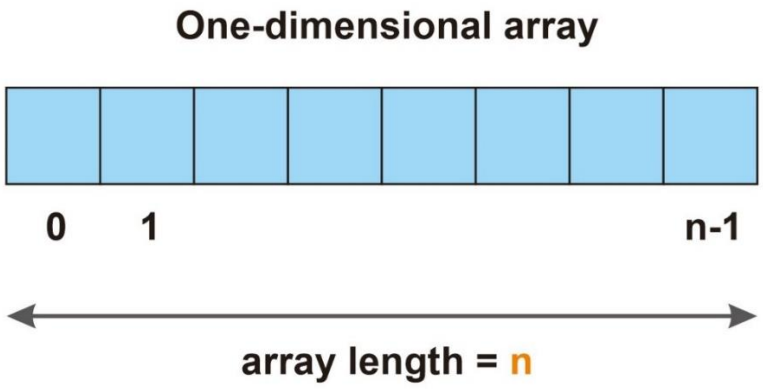
2. Apa itu *Indexed Collections*?

Pada *indexed collection, data* yang disimpan akan diurut berdasarkan *index*.

Array

Sebelum anda ingin membuat dan menggunakan sebuah *array*, anda harus memahami terlebih dahulu bahwa *array* memiliki karakteristik ***iterable***, ***destructurable*** dan tidak memiliki karakteristik ***keyed***.

Dalam buku berjudul ***Javascript – The Definitive Guide*** karya **David Flanagan** yang diterbitkan pada tahun 2002, dikatan bahwa *Array* adalah sebuah *data type* untuk menyimpan nilai yang diberi nomor (*numbered values*). Nilai yang diberi nomor disebut dengan ***element*** dan nomor yang diberikan pada sebuah *element* disebut dengan ***index***[27]. Sebuah *array* dapat menyimpan *primitive value* dan *object*, termasuk *array* yang dapat menyimpan *array*.



Gambar 265 One-dimensional Array

Create Array

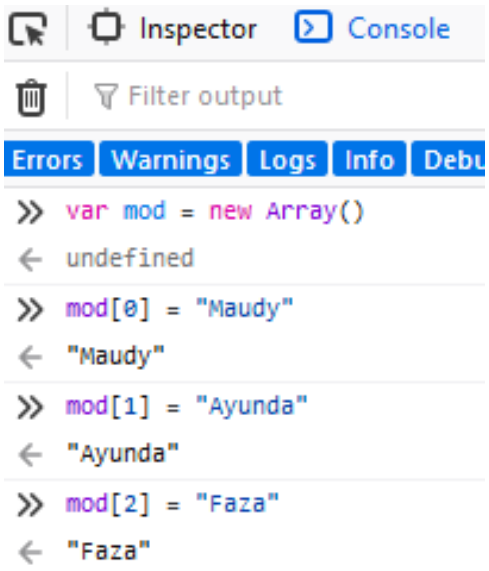
Dalam *Javascript* dengan sebuah *array* kita bisa menyimpan nilai lebih dari satu dalam satu variabel. Untuk membuat sebuah *array* dalam *Javascript* ada 3 cara :

Index & Element

Cara yang pertama adalah cara *regular* sebagai contoh kita akan membuat sebuah *array* yang menampung lebih dari satu *data types string*.

Pada kasus di bawah ini **`mod[0]`**, **`mod[1]`** dan **`mod[2]`** adalah *index* dari *array* mod dan **`“Maudy”`**, **`“Ayunda”`** dan **`“Faza”`** adalah *element* dari *array* mod. *Index* pada *array* dimulai dari angka nol.

Buka *web console* dengan menekan tombol **CTRL+SHIFT+K** :



Gambar 266 Sample Regular Way

Notes
Pada kode di atas kita membuat *array* menggunakan *keyword* ***new*** sangat tidak disarankan untuk membuat *array* dengan cara di atas, baca lagi konvensi *stop using new keyword*.

Array Constructor

Cara yang kedua adalah *condensed way*, memanfaatkan *constructor* yang dimiliki *array*. Bisa kita lihat pada gambar di bawah ini :

```
>> var mod = new Array("Maudy", "Ayunda", "Faza")
<- undefined
>> mod
<- ▶ Array(3) [ "Maudy", "Ayunda", "Faza" ]
```

Gambar 267 Sample Condensed Way

Notes
Pada kode di atas kita membuat *array* menggunakan *keyword* ***new*** sangat tidak disarankan untuk membuat *array* dengan cara di atas, baca lagi konvensi *stop using new keyword*.

Array Literal

Cara yang ketiga adalah *literal way*, bisa kita lihat pada gambar di bawah ini :

```
>> var mod = ["Maudy","Ayunda","Faza"]
<- undefined
>> mod
<- ▶ Array(3) [ "Maudy", "Ayunda", "Faza" ]
```

Gambar 268 Sample Literal Way

Akses *Array Element*

Selain membuat sebuah *array* kita juga bisa mengakses salah satu *element* yang ada di dalam sebuah *array*. Sebagai contoh pada gambar di bawah ini kita bisa mengakses *array* yang ada pada indeks pertama (dimulai dari nol).

```
>> var mod = ["Maudy","Ayunda","Faza"]
<- undefined
>> mod
<- ▶ Array(3) [ "Maudy", "Ayunda", "Faza" ]
>> mod[0]
<- "Maudy"
```

Gambar 269 Access an Array

Modify Array Element

Selain itu kita bisa mengubah salah satu *element* yang ada di dalam *array*, caranya bisa kita lihat pada gambar di bawah ini :

```
>> mod[2] = "cantik"
<- "cantik"
>> mod
<- ▶ Array(3) [ "Maudy", "Ayunda", "cantik" ]
```

Gambar 270 Modify an Array

Array mod pada indeks kedua nilainya diubah dari faza menjadi **cantik**, untuk membuktikanya kita bisa mengakses lagi *element* tersebut seperti pada gambar di atas.

Array Destructuring (ES6)

Destructuring mempermudah pekerjaan kita jika ingin melakukan ekstrasi suatu data dalam sebuah *array object*. Seperti yang telah dijelaskan sebelumnya bahwa array memiliki karakteristik *destructurable*. Di bawah ini adalah contoh penggunaan *destructure* pada *array* :

```
>> var mod = new Array("Maudy", "Ayunda", "Faza")
<- undefined
>> var [satu,dua] = mod;
<- undefined
>> satu
<- "Maudy"
>> dua
<- "Ayunda"
```

Gambar 271 Destructure Example

Array Looping

Seperti yang telah dijelaskan sebelumnya bahwa *array* memiliki karakteristik *iterable*. Kita dapat melakukan perulangan pada suatu *array* untuk mengetahui *element* di dalamnya menggunakan perulangan **`for`** :

```
>> for (var i = 0; i < mod.length; i++ ) {
     console.log(i + " " + mod[i])
   }
   0 Maudy
   1 Ayunda
   2 Faza
```

Gambar 272 For Looping in Array

Array Looping ES 6

Untuk mendapatkan *index* dan *element* dalam suatu *array*, kita bisa menggunakan *method* **`forEach()`**. *Method* ini akan mengeksekusi *callback* dan selalu menghasilkan *return undefined*. Pada gambar di bawah ini kita membuat *callback* untuk menampilkan *index* dan *element* yang dimiliki oleh *array* **`mod`** :

```
>> mod.forEach(function(item, index, array){
     console.log(item, index);
   });
   Maudy 0
   Ayunda 1
   Faza 2
```

Gambar 273 Loop Over an Array

Array Iterator Object

Untuk melakukan *looping* kita juga bisa membuat sebuah *iterator object* dengan *method* **`entries()`**. Di bawah ini adalah contoh kode yang bisa anda pelajari :

```
var soul = [ "Maudy", "Ayunda Faza", "Gun Gun", "Febrianza" ]
undefined
var x = soul.entries()
undefined
for (n of x) {console.log(n=n)}
▶ Array [ 0, "Maudy" ]
▶ Array [ 1, "Ayunda Faza" ]
▶ Array [ 2, "Gun Gun" ]
▶ Array [ 3, "Febrianza" ]
```

Gambar 274 Array Iterator Object Example

Untuk membuktikan bahwa x adalah *iterator object*, cukup dipanggil *variable* tersebut :

```
» x
← ▶ Array Iterator {  }
```

Gambar 275 Array Iterator Object

Array Map

Method map digunakan jika kita ingin mendapatkan *array* baru setelah kita mengeksekusi sebuah *callback* untuk setiap *element* yang ada di dalamnya. Pada gambar di bawah ini kita membuat *callback* dengan kemampuan untuk mengubah huruf kecil menjadi huruf besar untuk setiap *element* yang dimiliki oleh *array* **mod** :

```
» var mod = new Array("Maudy", "Ayunda", "Faza")
← undefined
» var gun = mod.map(function(item) {return item.toUpperCase();})
← undefined
» gun
← ▶ Array(3) [ "MAUDY", "AYUNDA", "FAZA" ]
```

Gambar 276 Array.map Example

Array Filter

Method **`filter`** digunakan jika kita ingin mendapatkan *array* baru yang dihasilkan dari *callback* dengan *return* bernilai *true*.

```
>> var array1 = ['maudy', 100, 'ayunda', 200, 'faza', 300];
<- undefined
>> var array2 = array1.filter(function(item) { return typeof item === 'string'; });
<- undefined
>> array2
<- ▶ Array(3) [ "maudy", "ayunda", "faza" ]
```

Gambar 277 Array.filter Example

Array Property & Method

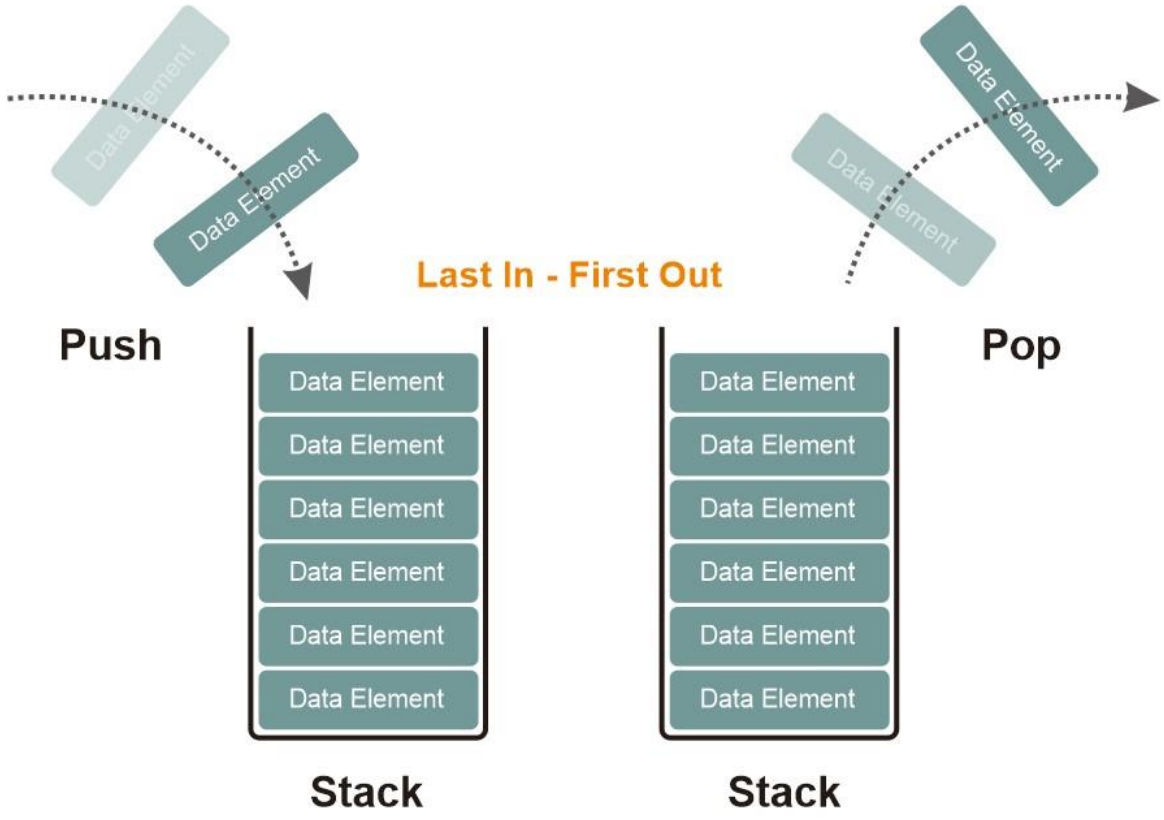
Array Properties

Sebuah *array* memiliki *properties* **`length`** untuk mengetahui jumlah *element* yang dimilikinya. Pada gambar di bawah ini didapatkan jumlah *element* dari *array mod*.

```
>> var mod = new Array("Maudy", "Ayunda", "Faza")
<- undefined
>> mod.length
<- 3
```

Gambar 278 Array Length

Push Array



Gambar 279 Push & Pop Method Visualization

Kita bisa menggunakan **`push`** *function* yang dimiliki *array* untuk menambahkan *element* baru diakhir elemen sebuah *array*, caranya bisa kita lihat pada gambar di bawah ini :

```
>> mod.push("cantik")
<- 4
>> mod
<- ▸ Array(4) [ "Maudy", "Ayunda", "Faza", "cantik" ]
```

Gambar 280 Push Array

Pop Array

Pop function digunakan untuk menghapus *element* terakhir di dalam sebuah *array*. Bisa kita lihat pda gambar di bawah ini :

```
>> mod.pop()
<- "cantik"
>> mod
<- ▸ Array(3) [ "Maudy", "Ayunda", "Faza" ]
```

Gambar 281 Pop Array

Shift Array

Dalam *array* kita bisa menggunakan **`shitft`** *function* untuk menghapus *element* yang posisinya di awal dalam sebuah *array*. Pada gambar di bawah ini *element* awal atau indeks pertama dari *array mod* di hapus.

```
>> mod.shift()
<- "Maudy"
>> mod
<- ▸ Array [ "Ayunda", "Faza" ]
```

Gambar 282 Shift Array

Unshift Array

Untuk menambahkan sebuah *element* di awal elemen sebuah *array* kita bisa menggunakan **`unshift`** *function*, cara penggunaanya bisa kita lihat pada gambar di bawah ini :

```
>> mod.unshift("Maudy")
<- 3
>> mod
<- ▸ Array(3) [ "Maudy", "Ayunda", "Faza" ]
```

Gambar 283 Unshift Array

Find Index Array

Untuk mengetahui *index* sebuah *element* kita bisa menggunakan **`indexOf`** *function* yang dimiliki sebuah *array*. Bisa kita lihat pada gambar di bawah ini :

```
>> mod
<- ▶ Array(3) [ "Maudy", "Ayunda", "Faza" ]
>> mod.indexOf("Ayunda")
<- 1
```

Gambar 284 IndexOf Array

Remove Item by Index

Untuk menghapus salah satu *element* dalam sebuah *array* berdasarkan *index* kita bisa menggunakan ***splice function***, kita akan mencoba menghapus *array* yang berada pada indeks ke dua. Bisa kita lihat pada gambar di bawah ini :

```
>> mod.splice(2)
<- ▶ Array [ "Faza" ]
>> mod
<- ▶ Array [ "Maudy", "Ayunda" ]
```

Gambar 285 Splice Array

Copy an Array

Dalam *Javascript* untuk mendapatkan salinan sebuah *array* kita bisa menggunakan ***slice function***, bisa kita lihat pada gambar di bawah ini :

```
<- ▶ Array [ "Maudy", "Ayunda" ]
>> var firstname = mod.slice(1)
<- undefined
>> firstname
<- ▶ Array [ "Ayunda" ]
```

Gambar 286 Slice Array

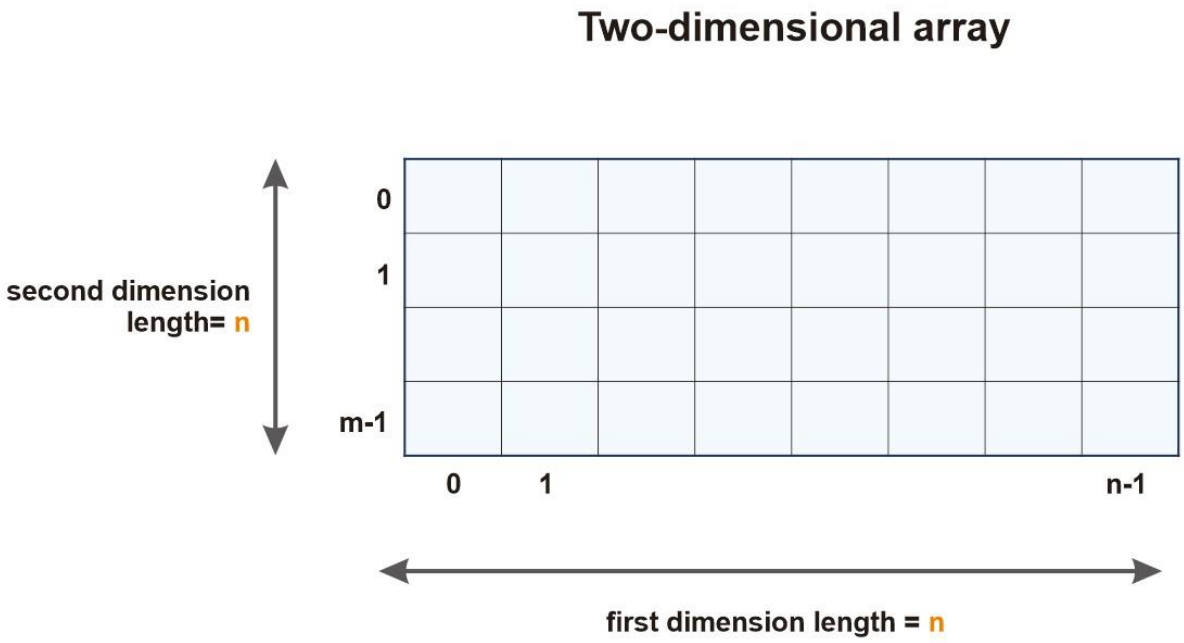
Merge Array

Untuk menggabung dua buah *array* atau lebih menjadi satu kita bisa menggunakan *method concat()*. Di bawah ini adalah contoh dua hati* yang di gabungkan :

```
>> var mod = ["Maudy", "Ayunda Faza"]
<- undefined
>> var gun = ["Gun Gun", "Febrianza"]
<- undefined
>> var together = mod.concat(gun)
<- undefined
>> together
<- ▶ Array(4) [ "Maudy", "Ayunda Faza", "Gun Gun", "Febrianza" ]
```

Gambar 287 Merge Array

Multidimensional Array



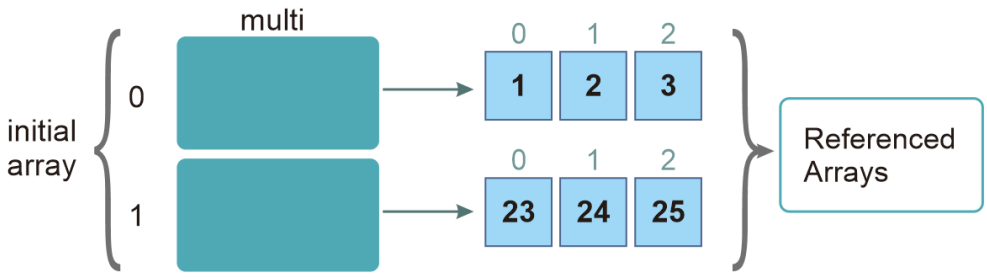
Gambar 288 Multidimensional Array

Sebuah *array element* yang menjadi referensi acuan untuk mendapatkan nilai pada *array* lainya disebut dengan ***Multi-dimensional Arrays***. Di bawah ini adalah contoh sederhana dari *Multi-dimensional Arrays* :

```
var multi = [[1,2,3],[23,24,25]]
console.log(multi[0][0])
console.log(multi[0][1])
console.log(multi[0][2])
console.log(multi[1][0])
console.log(multi[1][1])
console.log(multi[1][2])
```

Gambar 289 Example Multi-dimensional Array

Pada contoh kode di atas, kita membuat sebuah *array object* yang di dalamnya terdapat dua *elements*. Masing-masing *element* memiliki acuan yang terhubung satu sama lain. Untuk memudahkan visualisasi di bawah ini adalah representasi *array multi-dimension* :



Gambar 290 Visualized Multi-dimensional Array

Matrix

Indexed Collection seperti *array* dapat digunakan untuk merepresentasikan sebuah *matrix* sehingga kita bisa mengeksplorasi dunia aljabar dan aljabar linear. Kita dapat menggunakan *multi-dimensional array* untuk membuat sebuah *matrix* :

```
» let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
← undefined
» matrix[1][1]
← 5
```

Gambar 291 Matrix Representation

3. *Keyed Collections*

Map adalah sekumpulan *mapping*, asosiasi antar *object*. *Set* adalah *collection* yang tidak bisa memiliki nilai duplikat

Map

Map adalah tipe *collection* baru yang ditambahkan ke dalam *javascript*. Di desain untuk menyimpan *key & value pair* manggantikan *object*. Sebelum anda ingin membuat dan menggunakan sebuah *map*, anda harus memahami terlebih dahulu bahwa *map* memiliki karakteristik *iterable*, *keyed* dan tidak memiliki karakteristik *destructurable*.

Create Map

Untuk membuat sebuah *map* kita perlu menggunakan *keyword* **`map()`** seperti pada gambar di bawah ini :

```
>> var map = new Map();
<- undefined
```

Gambar 292 Create Map

Add Key & Value

Setelah kita membuat *map*, selanjutnya kita bisa menambahkan ***item*** ke dalam *map*. *Item* tersebut berupa ***key & value pair***. *Key* dapat menggunakan *string, number* atau pun *boolean*, pada kasus di bawah ini *key* '1' memiliki *value* **`"Maudy"`**, *key* 1 memiliki *value* **`"Ayunda"`** dan *key* **`true`** memiliki *value* **`"Faza"`**. Perhatikan gambar di bawah ini :

Gambar 293 Add Key & Value to Map

Get Map Item By Key

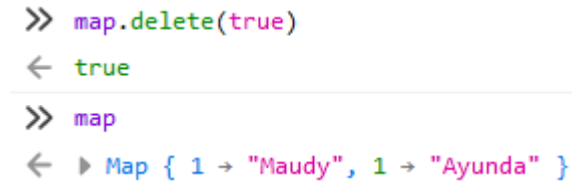
Untuk mendapatkan *value* dari sebuah *map* kita dapat menggunakan *key* sebagai *parameter* untuk *method get*. Perhatikan contoh kode di bawah ini :

```
console.log(map.get('1')); // Maudy
console.log(map.get(1)); // Ayunda
console.log(map.get(true)); // Faza
```

Gambar 294 Get Value on Map Item

Delete Map Item By Key

Untuk menghapus *item* di dalam *collection map* kita bisa menggunakan *method delete* dan *key* dari *map item* yang ingin dihapus. Perhatikan contoh kode di bawah ini :



Gambar 295 Delete Map Item

Delete All Map Item

Untuk menghapus semua *item* yang berada di dalam *map collection*, kita dapat menggunakan *method* **`clear()`**. Perhatikan contoh kode di bawah ini :

```
>> map.clear()
<- undefined
>> map
<- ▸ Map(0)
```

Gambar 296 Delete All Map Items

Check Map Item By Key

Jika kita ingin mengetahui sebuah *item* di dalam *map collection*, kita dapat menggunakan *method* **`has()`** yang dimiliki *map*. Perhatikan contoh kode di bawah ini :

```
>> map
<- ▸ Map(3) { 1 → "Maudy", 1 → "Ayunda", true → "Faza" }
>> map.has(1)
<- true
>> map.has(99)
<- false
```

Gambar 297 Check Map Item

Count Map Item

Jika kita ingin mengetahui jumlah *item* yang berada di dalam *map collection*, kita dapat menggunakan *property* **`size`** yang dimiliki oleh *map*. Perhatikan contoh kode di bawah ini :

```
← ▶ Map(3) { 1 → "Maudy", 1 → "Ayunda", true → "Faza" }
>> map.size
← 3
```

Gambar 298 Map Size

Iterate Map Keys

Untuk mengetahui seluruh *key* yang dimiliki oleh setiap *item* di dalam *map collection*, kita dapat menggunakan *method* **`keys()`** yang dimiliki oleh *map*. Perhatikan contoh kode di bawah ini :

```
← ▶ Map(3) { 1 → "Maudy", 1 → "Ayunda", true → "Faza" }
>> for (const k of map.keys()) {
     console.log(k)
   }
   1
   1
   true
```

Gambar 299 Iterate Map Keys

Iterate Map Values

Untuk mengetahui seluruh *value* yang dimiliki oleh setiap *item* di dalam *map collection*, kita dapat menggunakan *method* **`values()`** yang dimiliki oleh *map*. Perhatikan contoh kode di bawah ini :

```
>> for (const v of map.values()) {
     console.log(v)
   }
   Maudy
   Ayunda
   Faza
```

Gambar 300 Iterate Map Keys

Iterate Map Items

Jika kita ingin melakukan iterasi pada *map collection* untuk mendapatkan *key* dan *value* dari setiap *item*, maka kita bisa menggunakan *method* **`entries()`**. Perhatikan contoh kode di bawah ini :

```
>> map
<- ▶ Map(3) { 1 → "Maudy", 1 → "Ayunda", true → "Faza" }
>> for (const [k, v] of map.entries()) {
     console.log(k, v)
     }
   1 Maudy
   1 Ayunda
   true Faza
```

Gambar 301 Iterate Key & Values

Set

Set adalah tipe *collection* baru yang ditambahkan ke dalam *javascript*. Di desain untuk menyimpan nilai yang unik dan mencegah duplikat. Kita dapat menyimpan *primitive* dan *object* kedalam *set*. Sebelum anda ingin membuat dan menggunakan sebuah *map*, anda harus memahami terlebih dahulu bahwa *map* memiliki karakteristik *iterable*, *destructurable* dan tidak memiliki karakteristik *keyed*.

Create Set

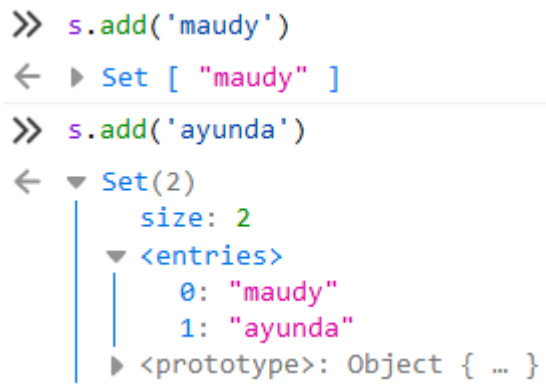
Untuk membuat sebuah *set* kita perlu menggunakan *keyword* **`set()`** seperti pada gambar di bawah ini :

```
>> const s = new Set()
<- undefined
```

Gambar 302 Create Set Collection

Add Items

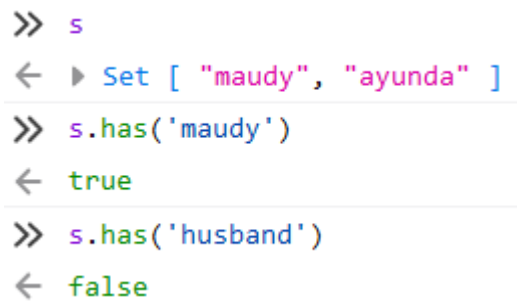
Setelah kita membuat *set*, selanjutnya kita bisa menambahkan *item* ke dalam *set*. Kita bisa menambahkan sebuah *primitive value* atau *object*. Pada kasus ini kita akan menambahkan dua buah *primitive string* ke dalam *set* :



Gambar 303 Add Items to Set

Check Item

Jika kita ingin mengetahui sebuah *item* di dalam *set collection*, kita dapat menggunakan *method* `has()` yang dimiliki *set*. Perhatikan contoh kode di bawah ini :



Gambar 304 Check Set Item

Delete Item

Untuk menghapus *item* di dalam *set collection* kita bisa menggunakan *method delete* dan *value* dari *set item* yang ingin dihapus. Perhatikan contoh kode di bawah ini :

```
>> s.delete('ayunda')
<- true
>> s
<- ▶ Set [ "maudy" ]
```

Gambar 305 Delete Set Item

Count Set Item

Jika kita ingin mengetahui jumlah *item* yang berada di dalam *set collection*, kita dapat menggunakan *property* **`size`** yang dimiliki oleh *set*. Perhatikan contoh kode di bawah ini :

```
>> s.size
<- 1
>> s
<- ▶ Set [ "maudy" ]
```

Gambar 306 Set Size

Delete All Set Items

Untuk menghapus semua *item* yang berada di dalam *set collection*, kita dapat menggunakan *method* **`clear()`**. Perhatikan contoh kode di bawah ini :

```
>> s.clear()
<- undefined
>> s
<- ▼ Set []
     size: 0
   ▶ <entries>
   ▶ <prototype>: Object { … }
```

Gambar 307 Delete All Set Item

Iterate Set Items

Untuk mengetahui seluruh *value* yang dimiliki oleh setiap *item* di dalam *map collection*, kita dapat menggunakan *method* **`values()`** yang dimiliki oleh *map*. Perhatikan contoh kode di bawah ini :

```
>> s
<- ▶ Set [ "maudy", "ayunda" ]
>> for (const k of s.values()) {
     console.log(k)
   }
   maudy
   ayunda
```

Gambar 308 Iterate Set Items

Chapter 4

Mastering Node.js

Subchapter 1 – *Re-introduction Javascript*

You can never understand everything.
But, you should push yourself to understand the system
— Ryan Dahl

Subchapter* 1 – *Objectives

- Memahami Apa itu ***Runtime Environment***?
- Memahami ***Abstraction*** pada bahasa pemrograman.
- Memahami *Javascript* sebagai ***System Language***.
- Memahami aplikasi apa saja yang dapat dibuat dengan ***System Language***.
- Memahami Apa itu ***I/O Scaling Problem***?
- Memahami Apa itu ***Process & Thread***?
- Memahami Apa itu ***Node.js System***?

Hari ini *javascript* adalah bahasa yang berhasil berevolusi menjadi sebuah ekosistem utuh. *Javascript* bukan lagi menjadi bahasa yang digunakan agar *website* menjadi interaktif. Kini dengan *javascript* kita bisa membuat sebuah *single page application* untuk *front-end development* dan *back-end development*.

"*Anything that can be written in JavaScript will eventually be written in JavaScript*!"

Node.js adalah sebuah ***Runtime Environment*** untuk *javascript*, *runtime environment* tersebut bernama *V8 engine*. Kita bisa menulis dan mengeksekusi *javascript* diberbagai *platform* yang didalamnya tersedia *node.js*.

Tahun 2009 *perspective* tentang *javascript* telah berubah, semenjak **Ryan Dahl** mengemukakan *node.js* dalam presentasinya di **JSConf.** Presentasinya mendapatkan

feedback positif dari komunitas *javascript*. Ryan Dahl terinspirasi membuat *node.js* setelah melihat sebuah *simple file upload progress bar* di situs flickr (*Image Sharing Site*).

Ryan melihat terdapat kesalahan didalamnya yang membuatnya terinspirasi untuk memberikan solusi yang lebih baik. V8 *engine* dalam *node.js* dikembangkan oleh Google dan telah dibuat *open source* pada tahun 2008.

Di bawah ini adalah 10 perusahaan besar yang menggunakan *node.js* :

1. Paypal
2. Netflix
3. Trello
4. Linkedin
5. Walmart
6. Uber
7. Medium
8. Groupon
9. Ebay
10. NASA

1. *System Programming*

Javascript adalah raja di dalam dunia *browser*, namun kali ini *javascript* siap menaklukan dunia di luar *browser*. Artinya sekarang kita mampu membuat berbagai macam aplikasi di luar konteks *browser* menggunakan bahasa *javascript*.

Berbicara pemrograman diluar konteks *browser* jika kita ingat lagi dalam sejarah bahasa pemrograman, ***C*** adalah bahasa yang paling *powerful*. Dari bahasa ***low level programming language*** ini terlahir bahasa ***high-level programming language*** seperti ***C++.***

Lalu bermunculan bahasa-bahasa pemrograman yang lebih tinggi lagi dalam memberikan kapabilitas *abstraction* (menyembunyikan kerumitan dan kedetailan), seperti bahasa pemograman *C#, Java, Ruby, Python* dan sebagainya.

Hal ini membuat predikat bahasa *C++* mendapatkan klasifikasi sebagai *middle level programming laguage*, beberapa literatur mengklasifikasikanya kembali sebagai bahasa *low-level programming language*.

Namun, hal yang paling penting disini adalah kemampuan bahasa pemrograman C sebagai *System Language* melahirkan banyak sekali turunan dalam dunia pemograman dan sistem komputer.

Dalam *javascript*, jika kita ingin menggabungkan dua buah *string* sangatlah mudah :

```
const string1 = "Hello";
const string2 = "Gun Gun Febrianza";
let combine = string1 + string2;
```

Pada bahasa *low-level* seperti C mendefinisikan *string* sangat mudah namun sesudahnya akan menjadi sedikit rumit, tanpa tersedianya ***Automatic Memory Management*** kita harus menentukan berapa besar *memory* yang kita butuhkan, mengalokasikanya,

kemudian menulis ke dalam memori tanpa menimpa *buffer* hingga membersihkan kembali *memory*.

Dalam kasus yang lebih besar manajemen penting ini digunakan untuk mencegah *memory leak*.

```
const char *string1 = "Hello";
const char *string2 = "Gun Gun Febrianza";
int size = strlen(string1) + strlen(string2);
char *buffer = (char *) malloc(size + 1);
strcpy(buffer, s1);
strcat(buffer, s2);
free(buffer);
```

Sebelum *node.js* muncul, *javascript* tidak bisa digunakan untuk melakukan pencarian blok memori untuk mendapatkan *byte* tertentu. Di dalam *browser*, *javascript* tidak dapat mengalokasikan *buffer* dan tidak memiliki tipe data untuk menangani *byte*. Namun dengan *Node.js* hal tersebut kini dapat dilakukan, karena *node.js* telah membuat *javascript* hidup seperti *system language*.

Jadi kira kira aplikasi apa saja yang mampu dibuat dengan *node.js*?

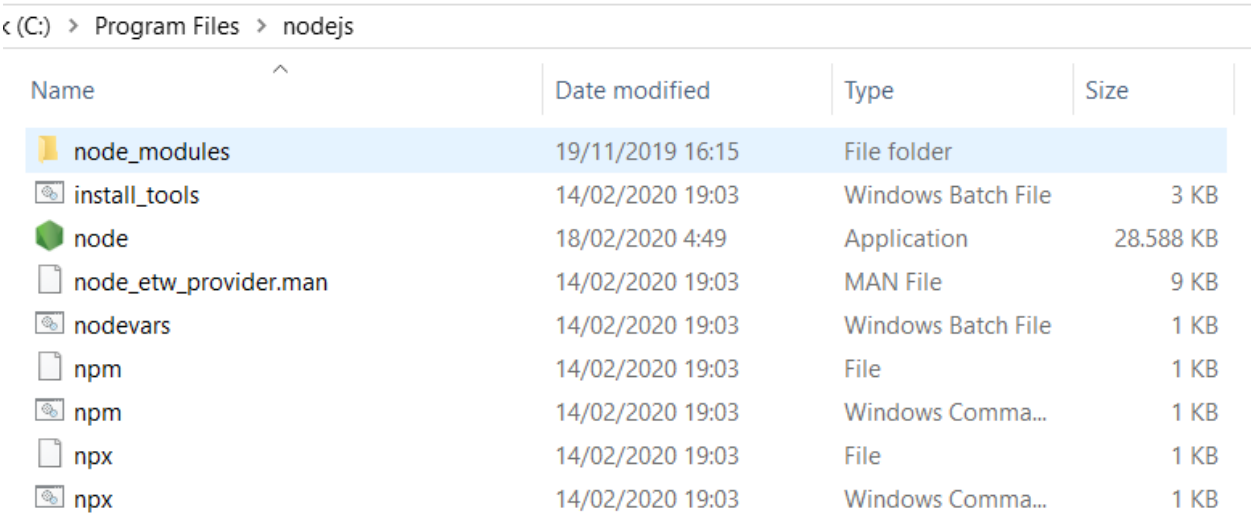
Dengan ***V8 Javascript engine*** kita dapat memanipulasi :

1. ***Buffer***, berinteraksi dengan *binary data* untuk membaca *file* atau *packet* dalam *network*.
2. ***Processes***, berinteraksi dengan *process* dalam *operating system*.
3. ***Filesystem,*** berinteraksi dengan *file* untuk mengolah informasi.
4. ***Stream,*** berinteraksi dengan *streaming data*.
5. ***Database***, berinteraksi dengan *SQL* & *NoSQL*.
6. ***Web Server***, berinteraksi dengan *Application Server*.

2. *Node.js System*

Node.js adalah perangkat lunak *open source* dan sebuah *cross platform javascript runtime*. *Node.js* telah menjadi *platform* baru untuk mengembangkan berbagai *application* seperti *web application* dan *application server*.

Setelah melakukan instalasi *node.js*, di bawah ini adalah *list program* pada sistem operasi *windows* yang disediakan dalam *node.js* dengan alamat pada *drive* **C:\Program Files\node.js** :



Gambar 309 Node.js System

Terdapat beberapa komponen yaitu :

1. ***node.exe***

   Program untuk memulai ***Javascript V8 Engine***, kita bisa menggunakanya untuk mengeksekusi kode *javascript*.

2. ***Folder node_modules***

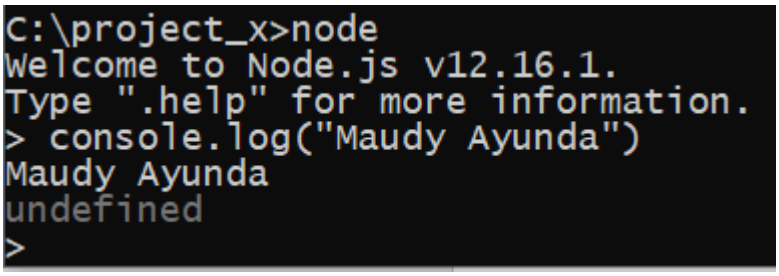
   Tempat untuk menyimpan *node.js packages*. Di dalamnya terdapat sekumpulan *packages* yang bekerja seperti *library* untuk memperluas kapabilitas *node.js*.

3. ***npx***

Program baru dalam *node.js* semenjak versi 5 ke atas disediakan untuk membantu mempermudah penggunaan *CLI Tools* dan *executable* lainya yang ada di dalam *Node Package Registry*.

Test Node.js Executable

Buka *command prompt*, kemudian eksekusi *Node Virtual Machine* dan eksekusi kembali *script hello world* seperti gambar di bawah ini :

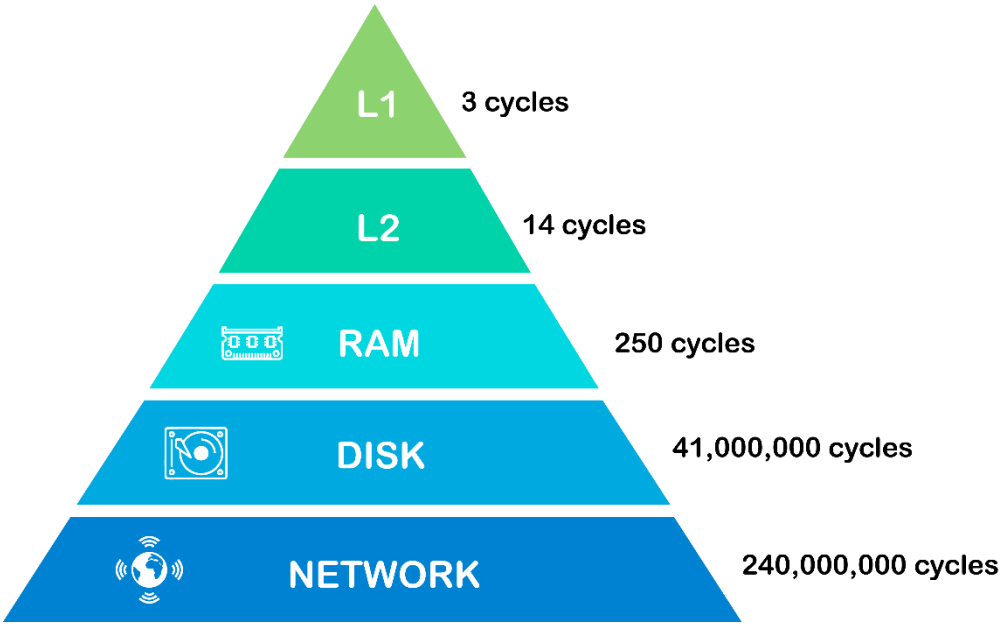


Gambar 310 Node Virtual Machine

Jika berhasil maka anda melakukan instalasi dengan benar.

3. *I/O Scaling Problem*

Selain berfokus pada pertimbangan terkait kecepatan *processor* dan *bandwidth memory*, terdapat area lainnya yang juga sangat penting yaitu *input* dan *output mechanism* untuk *secondary storage* dan *network*. Terdapat konsep dasar ***I/O Scaling Problem***. Perhatikan pada gambar di bawah ini :



Gambar 311 I/O Scaling Problem

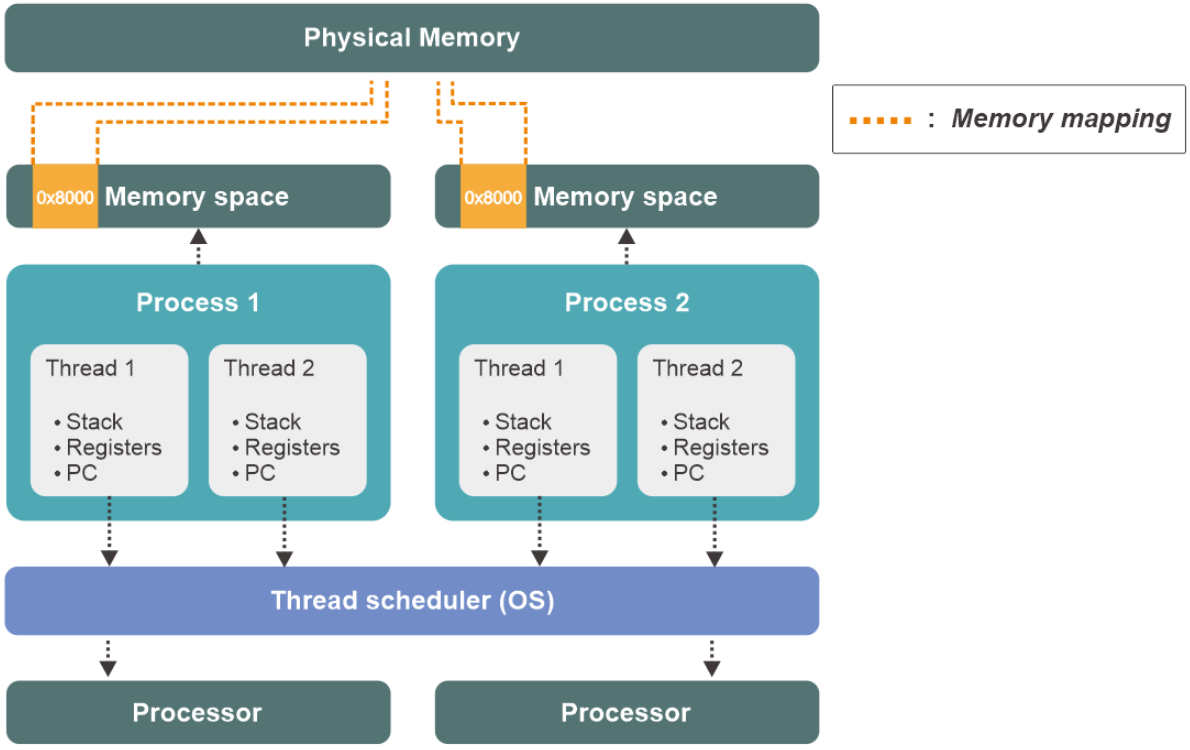
Input/Output Operation pada *disk* dan *network* memerlukan ***CPU Cycles*** yang sangat besar. Terminologi *CPU Cyle* mengacu pada ***Clock Cycle***, kecepatan sebuah *processor* komputer. Akses pada *disk* dan *network* memerlukan waktu yang lebih lama jika dibandingkan dengan ***RAM*** dan *cache* dalam *CPU (L1 & L2)*.

Sebagian besar *web application* akan membaca data di dalam *disk* atau data di dalam jaringan (*network*) komputer lainya, misal *database server*. Setiap kali terdapat *request* diterima oleh suatu *web application* untuk memuat data di dalam *database* maka *request* tersebut harus menunggu proses *reading data*.

Setiap kali terdapat *pending request* maka *server* akan mengkonsumsi *resources* (*memory* & *CPU*). Ketika terjadi *request* dalam jumlah yang sangat besar maka kita akan menghadapi *I/O Scaling Problem*.

4. *Process & Thread*

Pada *server* tradisional ***process*** baru akan dibuat setiap kali terdapat *web request*, alokasi *CPU* dan *memory* dibutuhkan untuk setiap proses. Untuk itu kita akan mengkaji secara singkat apa itu *process* dan *thread*.



Gambar 312 Process & Thread

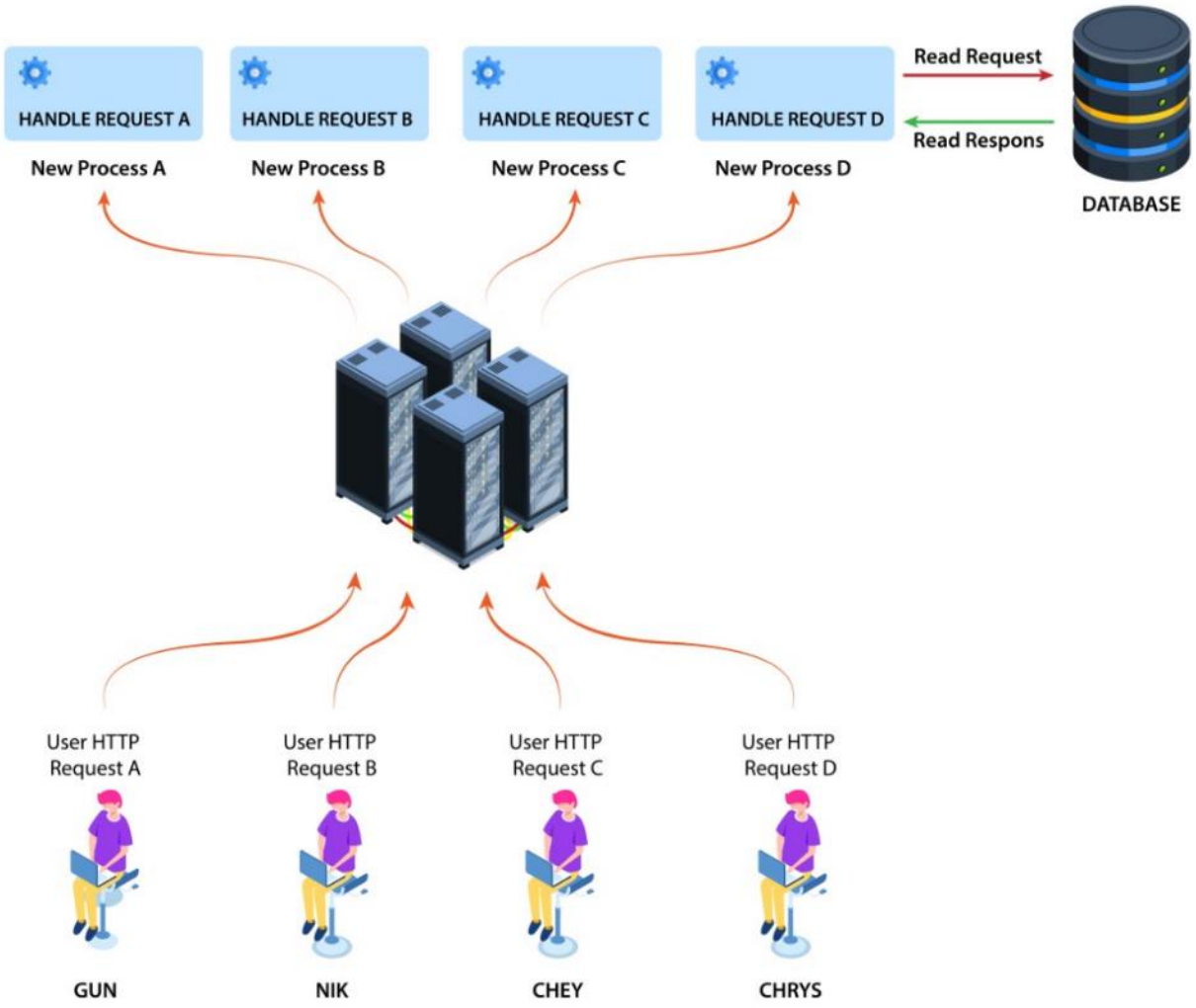
Sebuah program dalam sistem operasi yang dieksekusi akan disebut sebagai ***Process***. Setiap *process* menyediakan *resources* yang dibutuhkan agar program bisa dieksekusi. *Process* selalu disimpan di dalam *main memory/primary memory (RAM)*, sebuah program bisa dibangun dari beberapa *process*, masing-masing *process* dimulai dengan satu *thread tunggal* disebut dengan ***primary thread***, sebuah *process* dapat memiliki beberapa *thread*.

Multithread

Sebuah *thread* dapat disebut dengan ***Lightweight Process*** (LWP), untuk mencapai *parallelism* sebuah *process* akan dibagi menjadi sekumpulan *thread*. Sebagai contoh :

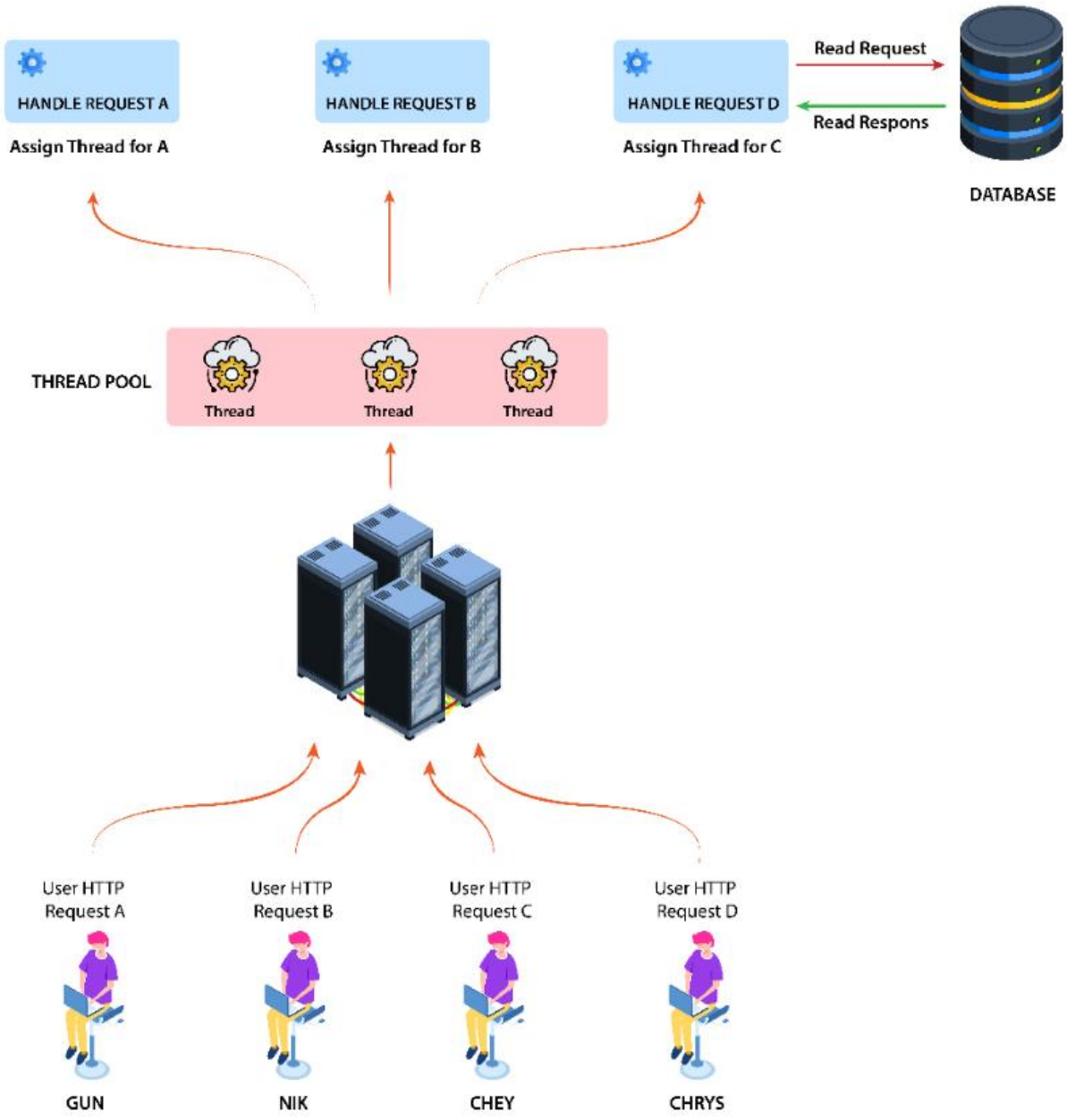
1. Pada aplikasi *browser* masing-masing *tab* yang kita gunakan memiliki *thread* masing-masing.
2. Pada aplikasi *microsoft word* terdapat *multiple thread*, 1 *thread* untuk format teks, 1 *thread* untuk memproses *input* dan 1 *thread* untuk untuk melakukan *autosave* dengan interval waktu tertentu.

Pada gambar di bawah ini terdapat ilustrasi beberapa *user* yang melakukan *request*, dimana pada *server* tradisional *process* akan dibuat untuk masing-masing *user* yang melakukan *request* :



Gambar 313 Web Server Using Processes

Beberapa *server modern* akan menggunakan *thread* pada *thread pool* untuk melayani setiap *request*, setiap kali *request* datang kita menetapkan *thread* untuk memproses *request*. *Thread* tersebut disediakan untuk *request* selama *request* sedang dilayani. Di bawah ini adalah ilustrasi *server modern* yang menggunakan *thread pool* :



Gambar 314 Web Server using Thread Pool

Ketika pembuatan *thread* atau *process* dilakukan berdasarkan *request* dari *client* maka, tantangan yang sering timbul menjadi masalah adalah ketika *concurrent request* terjadi. *Server* harus membuat *thread* baru untuk *handling* setiap *request* sehingga mengkonsumsi *resources* yang sangat besar.

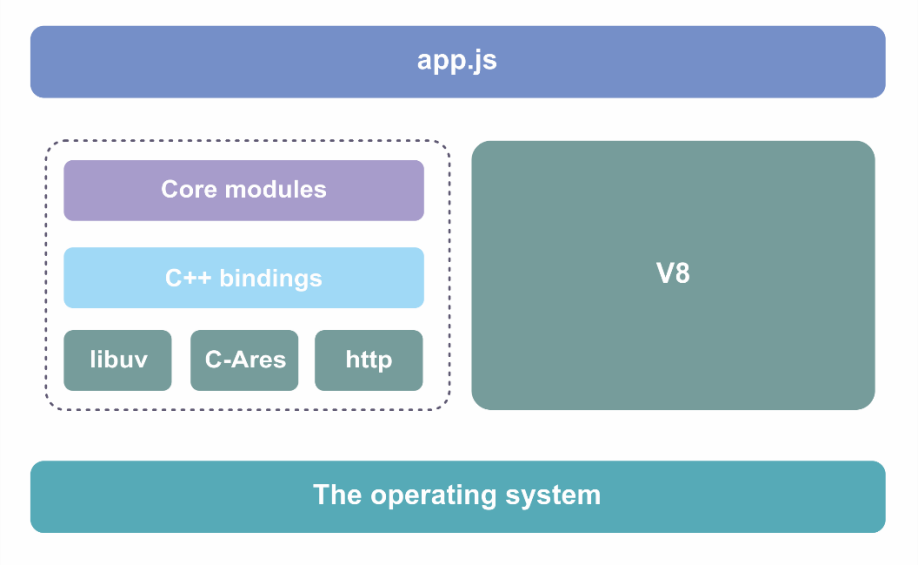
Penambahan *server* memang solusi agar tetap bisa mempertahankan *scalability* dari aplikasi namun ini justru menambah beban biaya dan tanggung jawab.

Untuk itu *node.js* hadir membawa solusi dari masalah tersebut. *Node.js* memiliki kemampuan *asynchronous processing* dalam sebuah *single thread* untuk menyediakan *performance* yang lebih baik dan kemampuan *scalability* untuk menghadapi *massive traffic*. Keuntungan dari operasi secara *asynchronous* adalah penggunaan *central processing unit* (*CPU)* tunggal dan memori menjadi lebih maksimal.

Anda sudah mempelajari bagaimana cara *node.js* bekerja menggunakan *single thread* pada *subchapter* sebelumnya tentang *The Call Stack*.

5. *Core Modules & libuv*

Node.js dibangun dari *standard library* dari bahasa C yang bersifat *open source*. Kelebihan dari *node.js* adalah *standard library* yang dimilikinya, pada *standard library* terdiri dari dua bagian yaitu sekumpulan ***Binary Libraries*** dan ***Core Modules***.



Gambar 315 Node.js Internal.

Binary Libraries

Salah satu *binary libraries* yaitu ***libuv*** didesain untuk *node.js* sebagai solusi atas permasalahan ***I/O Scaling Problem*** yang sering mengalami *Bottlenect*. Secara internal *node.js* menggunakan *libuv* yang menyediakan dukungan *asynchronous I/O* menggunakan *event loops*. Hal ini membuat *node.js* sangat *powerful* dalam *non-blocking I/O* untuk *networking* dan *file system*.

c-Ares adalah *library* yang digunakan *node.js* untuk melakukan *asynchronous DNS request*. Selain itu juga terdapat *HTTP Library* untuk membangun *HTTP Server* dan *HTTP Parser* yang digunakan untuk *parser HTTP Request & Response*.

Core Modules

Pada *Node.js core modules* sepenuhnya dibuat menggunakan *javascript*, untuk *developer* yang sudah *advance* jika terdapat sesuatu hal yang tidak kita fahami dan ingin diketahui secara detail kita tinggal melihat *source code node.js*.

Diantaranya terdapat *core modules* seperti ***fs*** untuk memanipulasi *file system*, *stream* untuk memanipulasi data *streaming* dan masih banyak lagi yang akan kita pelajari pada bab selanjutnya.

Source code Node.js dapat dilihat disini :

https://github.com/nodejs/node/tree/master/lib/internal

C++ Binding

Dengan ***C++ Binding*** kita mampu mengeksekusi *C/C++ code* atau *native library* ke dalam aplikasi *node.js*. *Node.js* menyediakan *Node.js Addons* yang menjadi *interface* antara *javascript* yang sedang berjalan dalam *V8* dan *C/C++ libraries*.

Subchapter 2 – V8 Javascript Engine

The Analytical Engine weaves algebraic patterns,
just as the Jacquard loom weaves flowers and leaves.

— Ada Lovelace

Subchapter 2 – Objectives

- Memahami apa itu ***EcmaScript*** pada bahasa pemrograman *javascript*.
- Memahami bagaimana bahasa pemograman *javascript* di buat.
- Memahami ***Javascript Engine V8*** pada *node.js*.
- Memahami istilah ***Single-threaded*** dan ***Call Stack*** pada *node.js*.
- Memahami ***Synchronous*** dan ***Asynchronous program***.
- Memahami ***Javascript Compilation Pipeline*** pada *node.js*.
- Memahami ***Memory Management*** & ***Memory Leak*** pada *javascript*.

Pernahkah kita bertanya dibuat atau ditulis dengan bahasa pemrograman apakah *javascript*?

Javascript sendiri sebenarnya adalah sebuah *standard*, yang spesifikasi standarnya di atur oleh ***EcmaScript***. Sederhananya, *EcmaScript* mengatur bahasa formal dalam *javascript* seperti :

1. *Syntax* (struktur bahasa pemrograman),
2. *Semantic* (makna dari bahasa pemograman) dan
3. *Pragmatic* (implementasi dari bahasa pemograman) .

Javascript bisa dibuat menggunakan bahasa pemrograman apa saja.

Bahkan *Javascript* bisa dibuat menggunakan bahasa *javascript*, salah satu contoh *projectnya* adalah ***narcissus*** :

https://github.com/mozilla/narcissus/

Narcissus adalah sebuah *Javascript Interpreter* yang dibuat menggunakan bahasa pemrograman *javascript*. Dalam *computer science* fenomena ini disebut dengan ***self-hosting interpreter***.

Jadi pertanyaanya bisa kita ubah menjadi :

Dibuat atau ditulis dengan bahasa pemograman apakah *javascript interpreter*?

V8 Javascript Engine sendiri yang digunakan oleh *node.js* ditulis menggunakan bahasa pemrograman *C++*. *V8 Javascript Engine* juga digunakan pada ***Google Chrome***. ***V8*** dapat mengenali *ECMAScript* yang telah dispesifikasikan dalam **ECMA-262**.

Browser lainya seperti *firefox* menggunakan *javascript engine* bernama ***TraceMonkey*** yang juga ditulisa menggunakan bahasa pemograman *C++*.

V8 Javascript Engine juga digunakan di dalam :

1. *MongoDB*
2. *Couchbase*
3. *Electron*
4. *NativeScript*

V8 sering kali disebut sebagai ***engine***, bukan *interpreter, compiler* ataupun *hybrid* kenapa?

Karena V8 adalah sekumpulan program yang menyediakan :

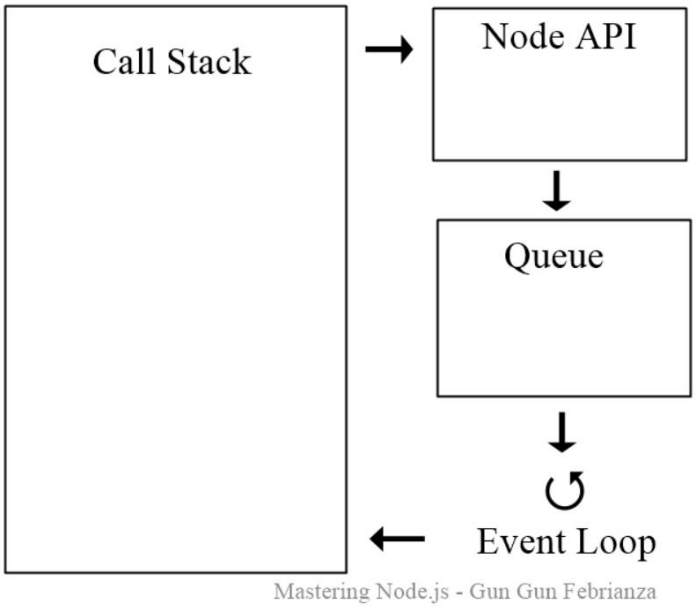
1. *Interpreter*
2. *Compiler*
3. *Memory Management*
4. *Garbage Collection*

1.*The Call Stack*

Javascript sering kali disebut ***Single Threaded*** *programming language*, maksudnya adalah *engine* untuk *javascript* hanya memiliki ***stack*** tunggal dan hanya bisa melakukan satu tugas dalam satu waktu.

Call Stack adalah sebuah *data structure* tempat merekam jejak kode yang dieksekusi dalam *V8 Javascript Engine*, *call stack* akan terus merekam jejak *javascript function* yang sedang dieksekusi dan *statement code* dalam *function* yang telah di eksekusi. Cara kerjanya adalah *Last In, First Out* (LIFO), yang terakhir masuk kedalam *stack* akan keluar lebih awal. Sehingga *Call Stack* memberikan dua layanan :

1. Kita bisa memasukan *javascript* ke dalam *stack*.
2. Kita hanya bisa mencabut *javascript* paling atas di dalam *stack*.



Gambar 316 Single Thread & The Call Stack

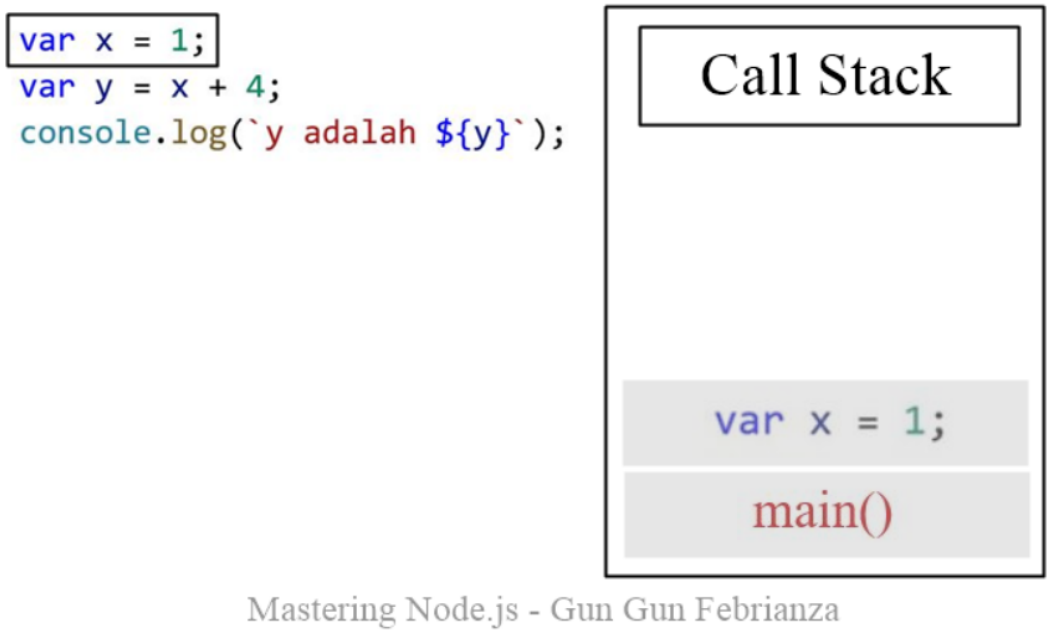
Kita akan mempelajari *call stack* dari 2 bentuk program dalam *javascript*, ***Synchronous*** dan ***Asynchronous program***. Pada *synchronous* eksekusi kode harus dilakukan secara berurutan dan pada *asynchronous* eksekusi kode dapat dilakukan tanpa harus berurutan.

Synchronous Program

Synchronous atau ***Synchronized*** memiliki makna terhubung (*connected*), ketika kita mengeksekusi suatu *task* secara *synchronous* maka kita akan menunggu *task* pertama selesai sebelum mengeksekusi *task* selanjutnya.

Pada ilustrasi gambar di bawah ini terdapat ***Main Function*** yang menjadi pembungkus (*wrapper*) sekumpulan *statement code*. *Main function* adalah *function* yang anda buat ketika menulis kode *javascript*, terdiri dari nama *function* dan *statement code* di dalam *function*. Nama *function* dan isi *statement code* di dalam *function* bersifat ***arbitrary***, anda bisa memberikan nama *function* dengan bebas.

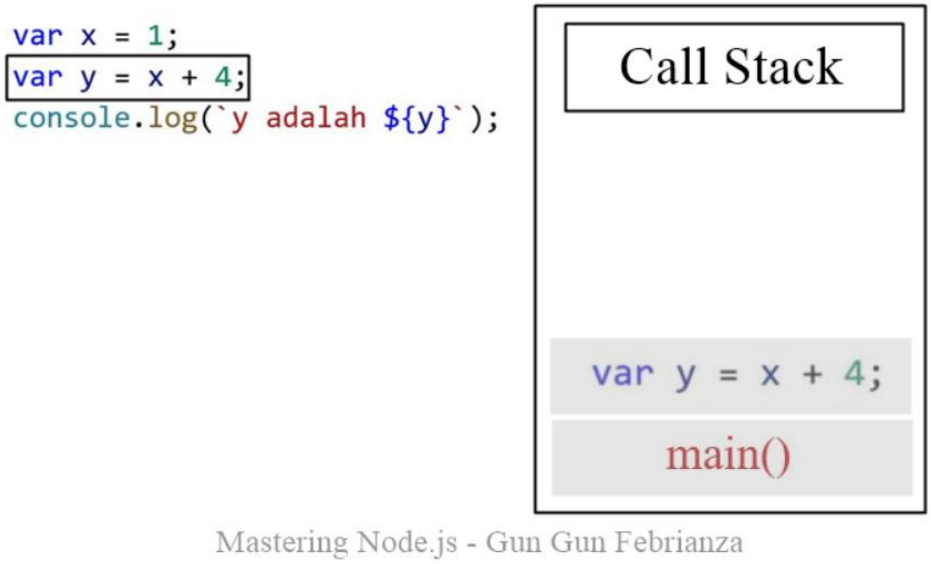
Main Function adalah perumpamaan untuk memberikan ilustrasi, jika kita perhatikan dari kode *javascript* di bawah ini. Kita membuat variabel **`x`** dan memberikan nilai berupa *literal* 1, ini adalah *statement code* pertama yang berjalan.



Gambar 317 Statement Code 1 On The Call Stack

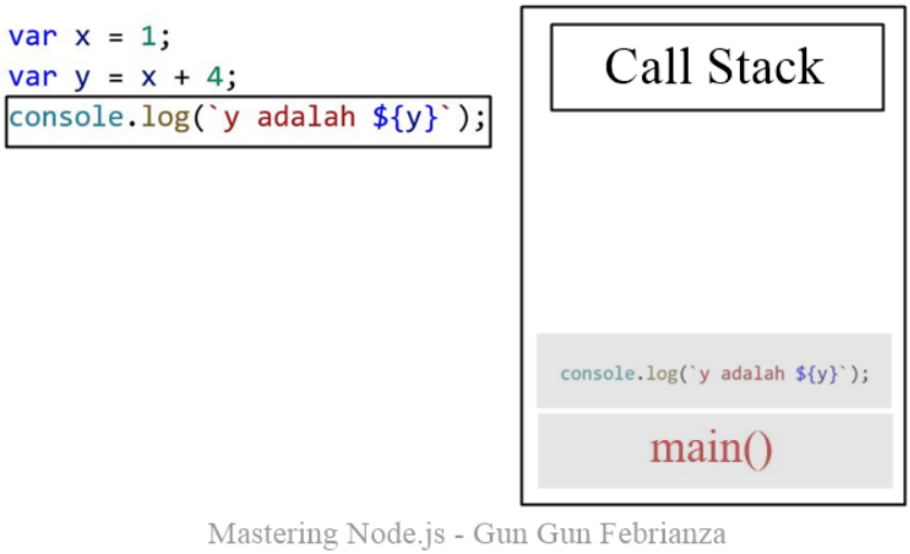
Statement tersebut masuk ke dalam *stack* setelah *main function* karena *statement* **`var x = 1`** dibungkus atau berada di dalam *main function*. Setelah *statement* **`var x = 1`**

dieksekusi, *statement* tersebut berada di posisi paling atas dan akan dicabut digantikan oleh *statement code* selanjutnya.



Gambar 318 Statement Code 2 On The Call Stack

Pada fase ini terjadi operasi penjumlahan, hasil penjumlahan disimpan pada variabel ***y***. Selanjutnya *statement* tersebut akan dicabut dari *stack* diganti *statement* yang terakhir.



Gambar 319 Statement Code 3 On The Call Stack

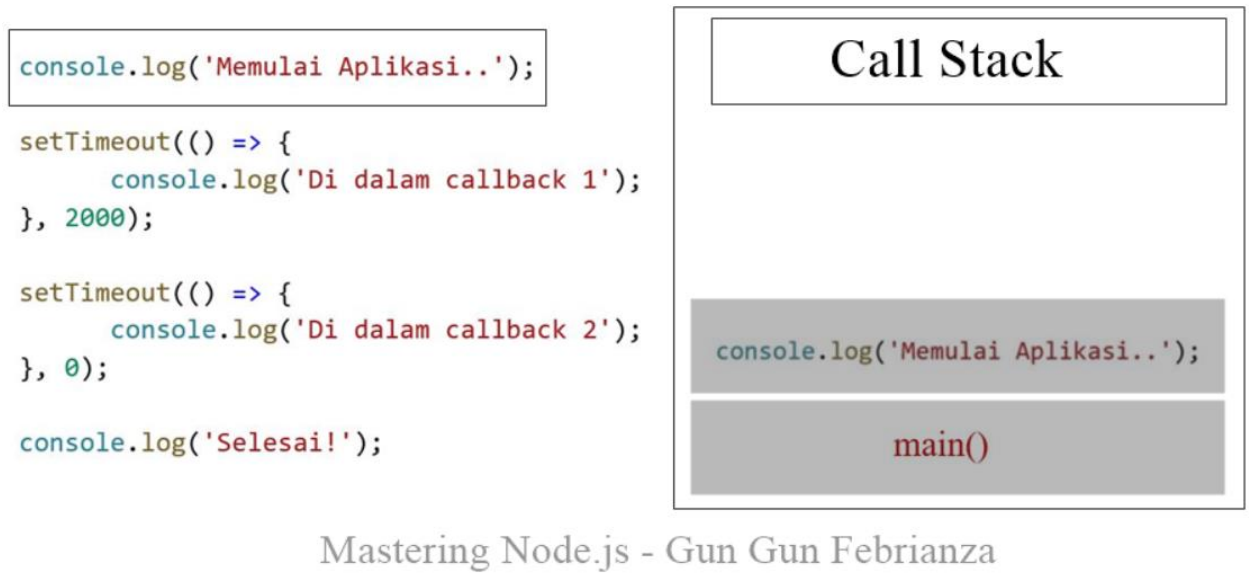
Pada fase ini *statement* yang terakhir akan menampilkan nilai dari ***y***, setelah itu *statement* ini akan dicabut dari *call stack* diikuti dengan *main function* sampai *stack* menjadi kosong kembali.

Asynchronous Program

Ketika kita mengeksekusi suatu *task* secara ***Asynchronously***, kita bisa mengeksekusi *task* selanjutnya tanpa harus menunggu *task* sebelumnya selesai. Eksekusi secara *Asynchronous* membutuhkan komponen lain yaitu *callback queue,* fungsi *callback queue* untuk mengetahui kapan waktu yang tepat untuk mengeksekusi *task* atau sekumpulan *task* yang telah selesai.

Pada contoh kali ini kita akan menggunakan ***Call Stack***, ***Node.js API***, ***Callback Queue*** dan ***Event Loop***. Semua komponen tersebut akan digunakan ketika *javascript engine* berhadapan dengan ***Asynchronous Program***.

Seperti biasa kita akan memasukan terlebih dahulu *main function* ke dalam *stack*. Di ikuti *statement code* yang pertama di dalam *function* tersebut.

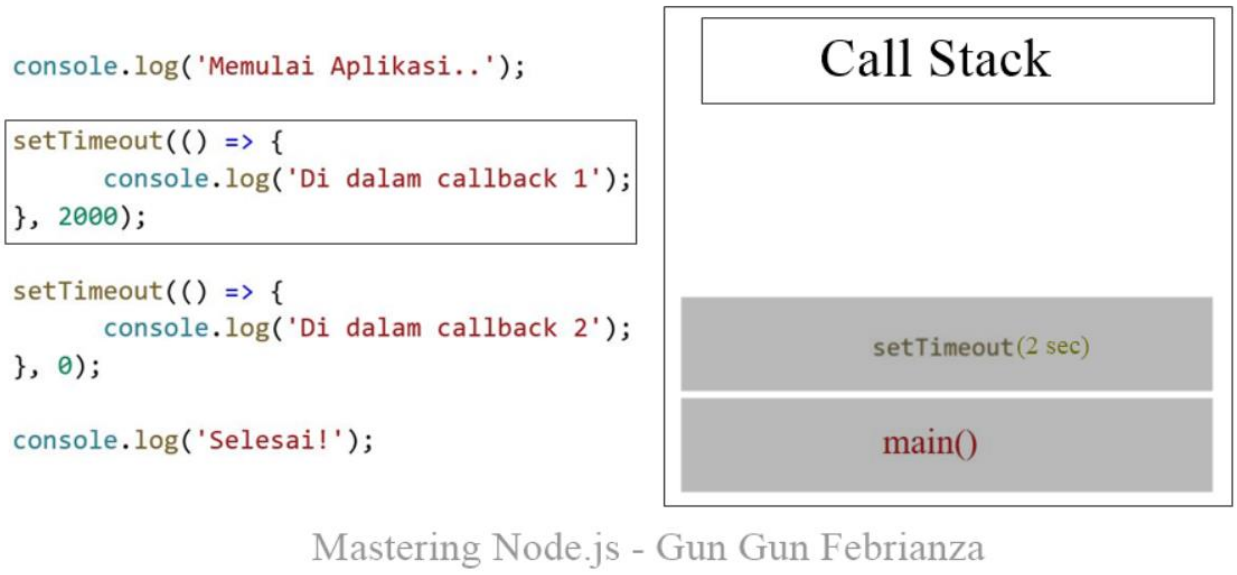


Gambar 320 Asynchronous Program on The Stack

Single Thread for Concurrent Connection

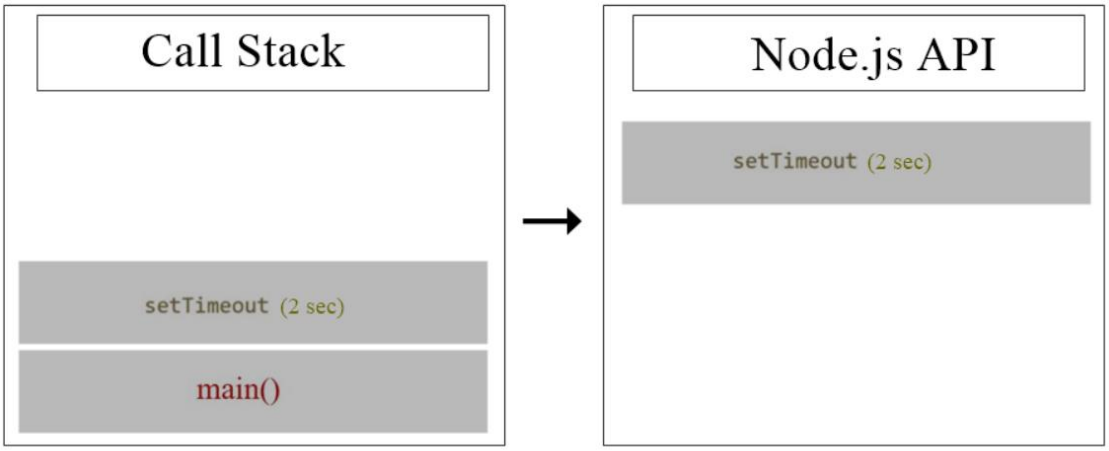
Javascript Engine dalam *Node.js* memiliki kemampuan *asynchronous processing* dalam sebuah *single thread* untuk menyediakan *performance* yang baik dan kemampuan *scalability* untuk menghadapi ***massive traffic***.

Jika diperjelas lagi *javascript Engine* dalam *Node.js* memiliki *single threaded asynchronous processing model* memiliki manfaat untuk mengatasi ***concurrent request*** dengan pengunaan *resources* yang lebih hemat.



Gambar 321 First Asynchronous Code

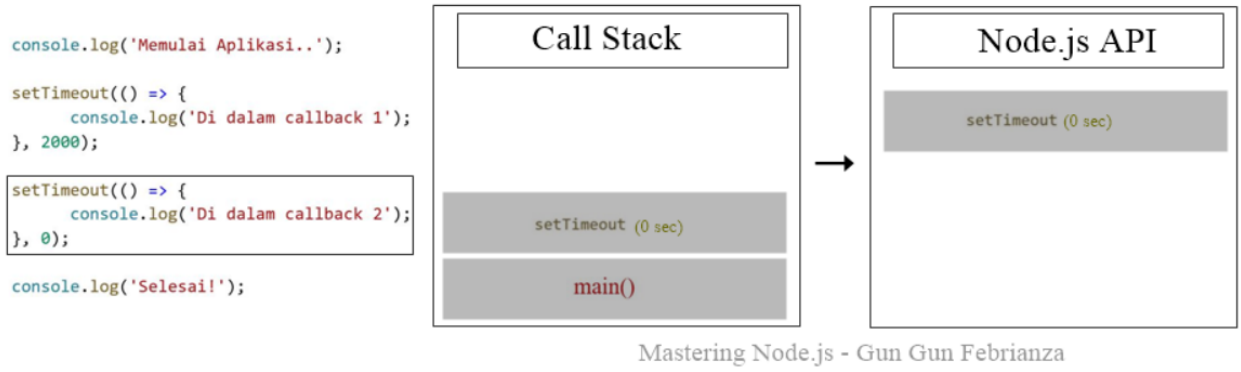
Saat kita memanggil **`setTimeout`** ***(2 sec) function*** ke dalam *call stack*, maka kita juga memerintahkan untuk mendaftarkan kedalam ***node.js API***. Pada contoh kode di atas ***event*** adalah sebuah *event* sederhana untuk menunggu 2 detik sebelum mengeksekusi kode di dalam ***callback***.



Gambar 322 Register Event

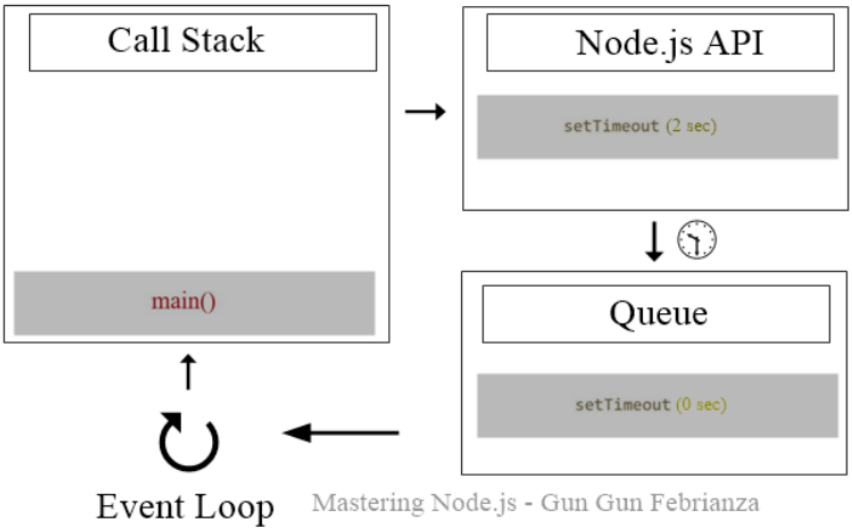
Setelah mendaftarkan *event*, *Call Stack* terus berlanjut, dan fungsi ***setTimeout*** terus berjalan menghitung waktu mundur *(counting down*). Mungkin anda berpikir kita akan menunggu selama dua detik di dalam *call stack*, namun ternyata tidak.

Call stack memang bisa melaksanakan satu tugas dalam satu waktu namun bukan berarti *call stack* tidak bisa melanjutkan pekerjaanya untuk membaca *statement code* berikutnya. Jadi saat *call stack* terus bekerja begitu juga ***event*** yang kita daftarkan bekerja dalam waktu yang sama secara bersamaan. Inilah yang disebut dengan ***asynchronous***.



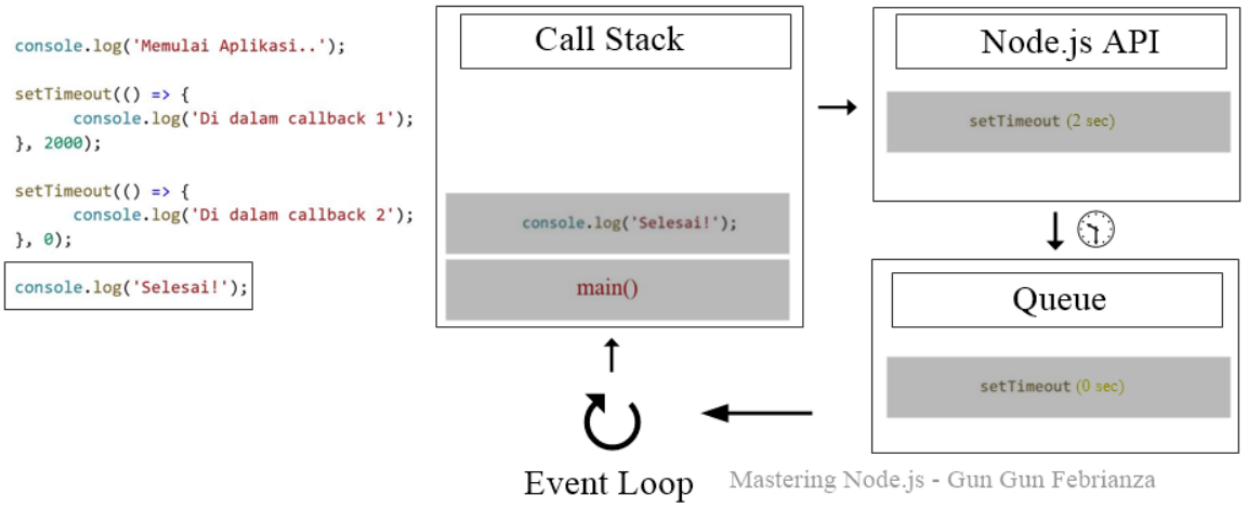
Gambar 323 Register Last Event

Pada fase ini kita mendaftarkan **`setTimeout`** ***(0 sec) function,*** dan *Call Stack* akan mencabutnya setelah selesai dieksekusi. **`setTimeout`** ***(0 sec) function*** akan selesai lebih awal karena memiliki *zero delay*. Saat selesai, *event* tersebut tidak akan langsung dieksekusi. *Event* tersebut akan dipindahkan terlebih dahulu ke dalam *Queue*, di dalam *queue* terdapat sekumpulan ***callback function*** yang telah siap untuk dieksekusi.



Gambar 324 Queue Activity

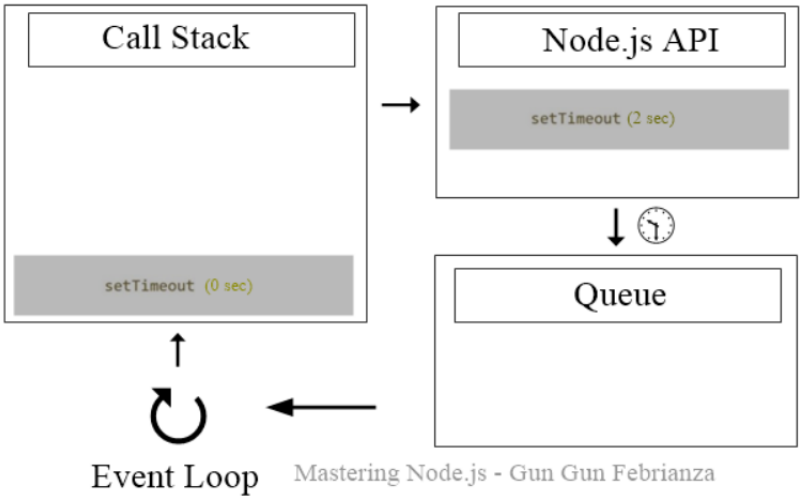
Queue adalah tempat dimana *callback function* yang kita buat menunggu sampai ***call stack*** menjadi kosong. ***Event Loop*** akan melihat *Call Stack*, jika *call stack* masih belum kosong kode maka *callback function* dalam *queue* belum bisa dieksekusi. Masih tersisa *console.log('selesai'), statement* tersebut akan dicabut dalam *stack* diikuti *main function()*.



Gambar 325 Last Asynchronous Statement

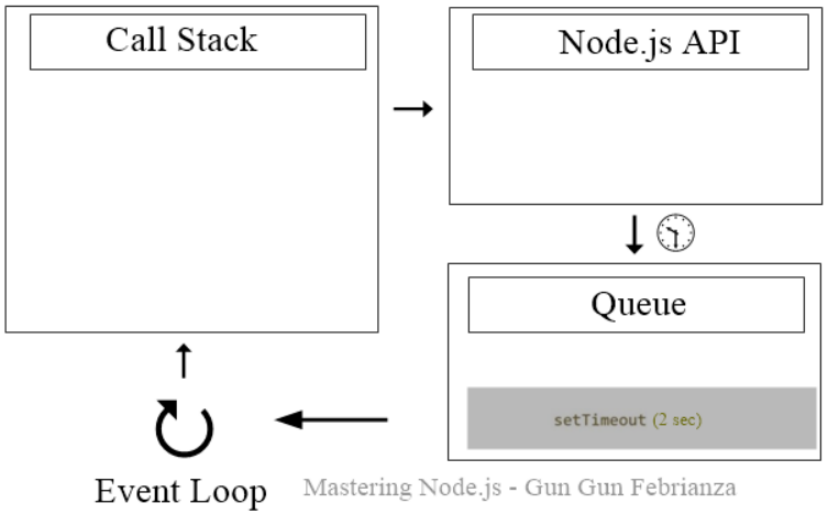
Event Loops

Melanjutkan dari ilustrasi gambar sebelumnya pada ilustrasi gambar di bawah ini, ***event loop*** melihat kesempatan untuk mengambil *callback function* dalam *queue* ke dalam *call stack* :



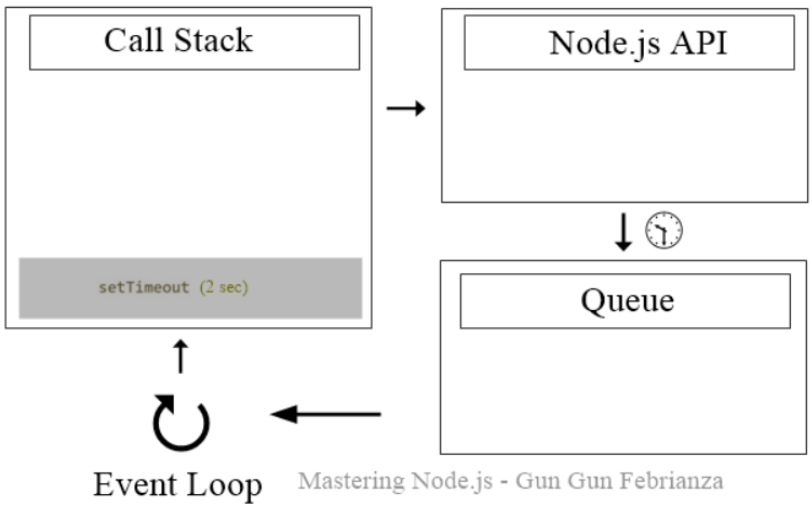
Gambar 326 Extract Callback Function

Pada fase tidak ada satupun dalam *Call Stack* & *Queue*, namun masih tersedia satu *event listener* terdaftar di dalam *Node APIs*. Dua detik kemudian ***setTimeout (2 sec) function*** siap menuju *Queue*.



Gambar 327 Event go to Queue

Pada fase ini ***event loop*** melihat posisi *stack* telah kosong dan mengambil ***callback function*** di dalam *queue*. Operasi dilakukan hingga *stack* kembali kosong.



Gambar 328 Extract Last Callback Function

Begitulah cara kerja *stack* di belakang layar. *Event Loops* membuat *node.js* mampu melakukan operasi *non-blocking* I/O.

Blocking

Istilah *blocking* mengacu pada operasi menghentikan *task* yang akan di eksekusi di masa mendatang sampai salah satu *task* selesai. Sehingga eksekusi *task* dilakukan secara berurutan.

Non-blocking

Istilah *non-blocking* adalah lawan kata dari *blocking*, terminologi *non-blocking* digunakan secara spesifik. Terminologi *non-blocking* biasanya digabung bersama I/O (*Input* atau *Ouput*) sementara *asynchronous* bersifat general dan mencakup beberapa operasi yang sangat luas.

2.Javascript Compilation Pipeline

Sebelumnya pada *chapter* 3 kita belajar bahwa *javascript* adalah *interpreted language*. *Javascript* adalah bahasa yang memerlukan *interpreter*. Setiap *browser* seperti *chrome, firefox* dan *opera* memiliki *interpreter* untuk menterjemahkan kode *javascript*.

Kini mengandalkan *interpreter* saja sudah tidak efisien. Alasan tersebut membuat *browser* mengadopsi *compiler* untuk mengatasi kekurangan *interpreter*. Sehingga *javascript* tidak lagi disebut sebagai *interpreted language*.

V8 dan sebagian besar ***modern javascript engine*** sudah menggunakan ***Just-in-time compilation***, Untuk memahami apa itu *JIT Compilation* ada beberapa sub kajian dasar tentang *compiler construction* yang harus kita pelajari terlebih dahulu:

Interpreter & Compiler

Selalu ingat, komputer hanya memahami satu bahasa yaitu ***Machine Language***.

Bahasa pemrograman di desain ***human-readable*** untuk mempermudah kita dalam memberikan instruksi kepada mesin komputer. Dalam pemograman terdapat dua cara untuk menterjemahkan ke dalam bahasa mesin yaitu :

1. Menggunakan ***Interpreter***
2. Menggunakan ***Compiler***

Sebuah *interpreter* menerjemahkan satu *statement* dari *high level language* kedalam *machine code* dan langsung mengeksekusinya, kemudian menerjemahkan kembali *statement* selanjutnya sampai selesai.

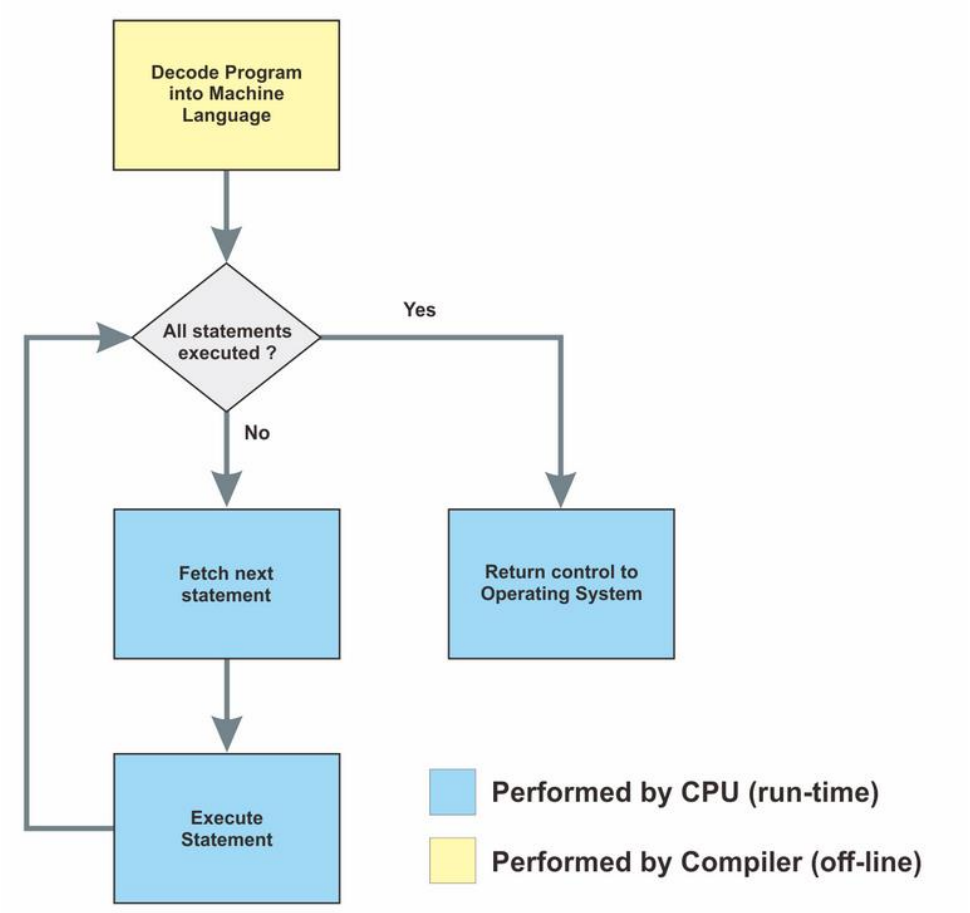
Gambar 329 Interpreter Illustration

Sebuah *interpreter* dan *compiler* secara umum akan menterjemahkan sumber kode (*source code*) dalam bahasa tingkat tinggi (*high-level language*) kedalam bahasa mesin (*machine language*) agar bisa dieksekusi oleh CPU.

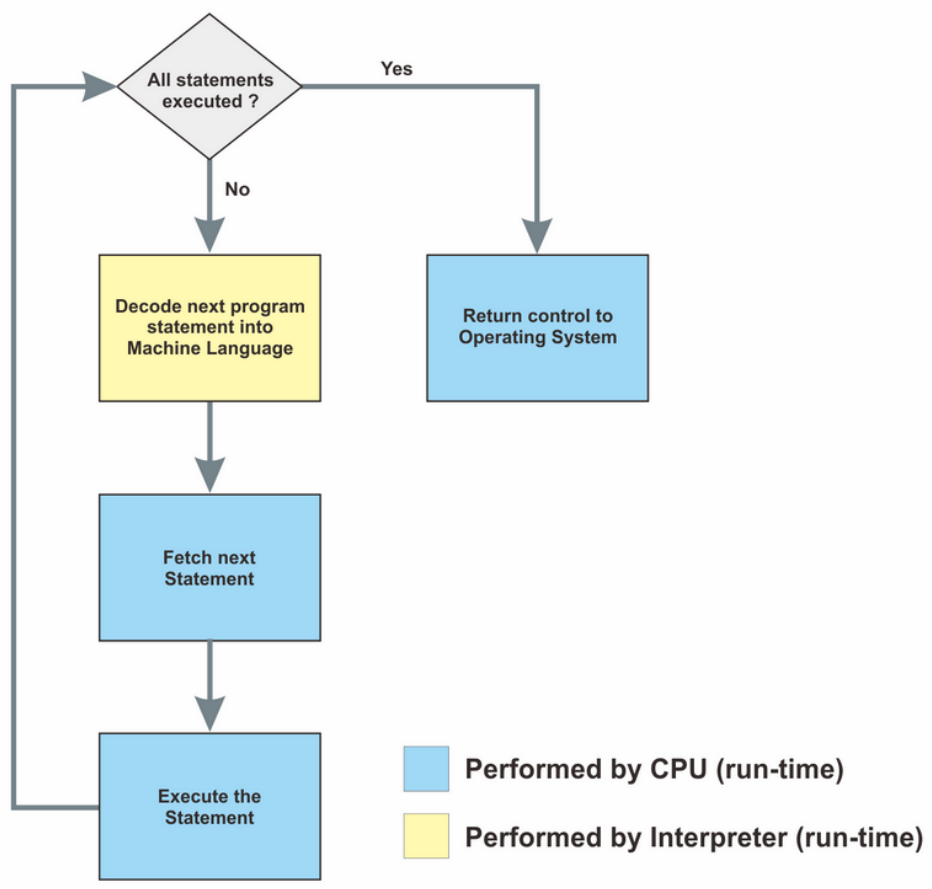
Kompiler menterjemahkan seluruh sumber kode sekaligus dalam satu fase kompilasi, hasil program yang telah dikompilasi dieksekusi dengan mode *fetch-execute cycle*.

Pada *interpreter*, kode sumber harus dilakukan penterjemahan terlebih dahulu secara *statement by statements*. Program yang dibuat menggunakan *interpreter* dieksekusi dengan mode *decode-fetch-execute cycle*, proses *decode* dilakukan oleh *interpreter* sendiri selanjutnya *fetch-execute* dilakukan oleh CPU. Proses *decode* pada *interpreter* membuat eksekusi program menjadi lebih lambat jika dibandingkan dengan *compiler*.

Di bawah ini adalah *flowchart* perbandingan eksekusinya.



Gambar 330 Proses eksekusi pada kompiler



Gambar 331 Proses eksekusi pada *interpreter*.

Pada fakta *flowchart* di atas *interpreter* harus melewati tahap *decode* pada setiap *statement* terlebih dahulu agar bisa dieksekusi oleh CPU.

Table 12 Perbedaan Kompiler dan Interpreter

Compiler	***Interpreter***
Menerjemahkan seluruh sumber kode sekaligus.	Menerjemahkan sumber kode baris perbaris.
Proses penerjemahan sumber kode cenderung lebih lama namun waktu eksekusi cenderung lebih cepat.	Proses penerjemahan sumber kode cenderung lebih cepat namun waktu eksekusi cenderung lebih lambat.

Membutuhkan penggunaan memori lebih banyak untuk menampung keluaran *intermediate object*.	Memori lebih efisien karena tidak memproduksi *intermediate object*.
Keluaran program yang dihasilkan bisa digunakan tanpa harus melakukan penerjemahan ulang.	Tidak ada keluaran program.
Kesalahan sumber kode ditampilkan setelah sumber kode diperiksa.	Kesalahan sumber kode ditampilkan setiap kali instruksi dieksekusi.

Machine Code

Machine Language adalah bahasa yang mampu difahami secara langsung oleh mesin komputer. *Machine code* atau *Machine Language* adalah sekumpulan instruksi atau *set of instruction* yang langsung dieksekusi oleh CPU (*Central Processing Unit*). Seluruh instruksi informasinya direpresentasikan dalam bentuk angka 1 dan 0 yang diterjemahkan dengan cepat oleh komputer.

V8 Javascript Engine juga memiliki fitur ***Code Caching***, kode mesin (*machine code*) yang telah dikompilasi disimpan secara lokal. Sehingga ketika kode *javascript* dalam suatu program yang sama dieksekusi kembali proses *parsing* dan *compiling* bisa di *skip*.

Sebelumnya ***V8 Javascript Engine*** menggunakan 2 *compiler* sekaligus, yaitu :

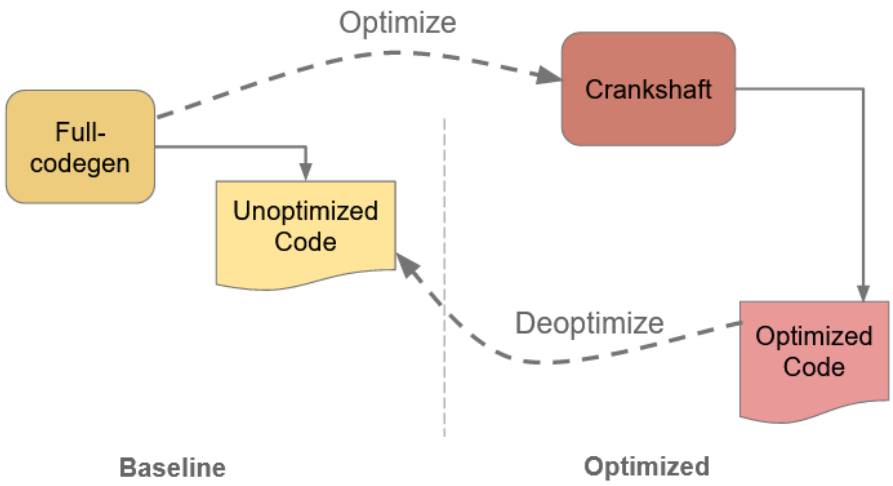
1. ***Full-codegen*** (*Baseline Compiler*) — Kompiler yang dapat memproduksi kode mesin yang belum dioptimasi (*non-optimized machine code*).
2. ***Crankshaft*** (*Optimized Compiler*)— Kompiler yang dapat memproduksi kode mesin yang telah dioptimasi (*optimized machine code*).

V8 Javascript Engine menggunakan beberapa ***thread*** dibelakang layar:

1. *Thread* utama membaca kode, mengkompilasi dan mengeksekusi *machine code*.
2. Saat *Thread* utama sedang mengeksekusi *machine code*, *Thread* kedua juga melakukan kompilasi untuk memproduksi *optimized machine code*.
3. *Thread* yang ketiga mengeksekusi program ***Profiler*** untuk menganalisa kode mesin yang sedang berjalan (*runtime*), informasi ini akan dikirimkan ke *Crankshaft* untuk memproduksi *optimized machine code*.
4. *Thread* yang kelima meenjalankan ***Garbage Collector Sweeps*** untuk membersihkan kembali memori.

Javascript di kompilasi pertama kali menggunakan ***Baseline Compiler*** yang mampu memproduksi ***machine code* yang belum di optimasi** secara cepat.

Machine code tersebut kemudian di analisa saat sedang berjalan menggunakan program bernama *profiler* dan secara opsional bisa di *re-compil* secara dinamis menggunakan kompiler yang didesain untuk melakukan optimasi (*optimized compiler*) agar bisa memproduksi *machine code* yang mampu memberikan *peformance* sampai puncak (*peak performance*).



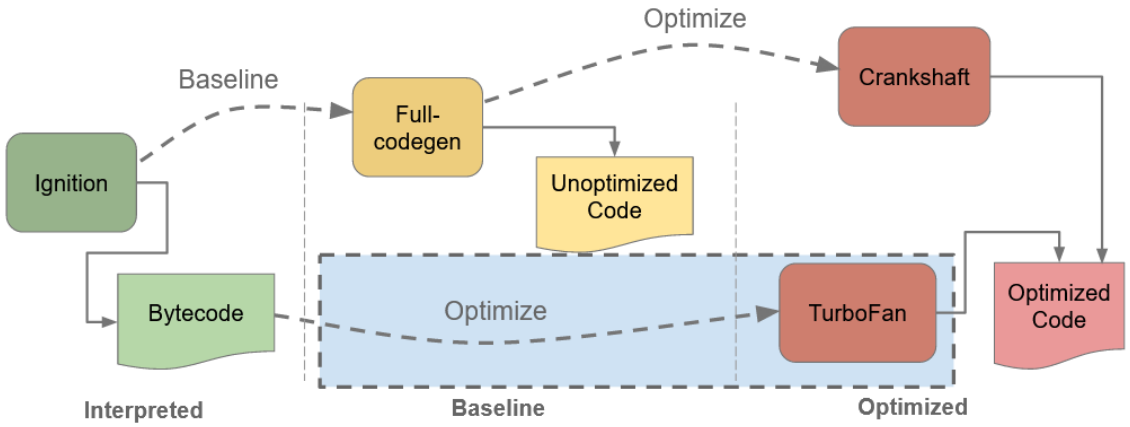
Gambar 332 V8 Compilation Pipeline

Pada gambar di atas adalah *Compilation Pipeline* yang digunakan dalam *V8 Javascript Engine* dari tahun 2010 sampai tahun 2015, di tahun 2015 juga.

Salah satu masalah besar dalam pendekatan yang digunakan di atas adalah *machine code* yang diproduksi dari *Full-codegen* mengkonsumsi memori yang amat besar. Sehingga semenjak **versi 5.9** penggunaan *Full-codegen* dan *Crankshaft* sudah tidak digunakan lagi.

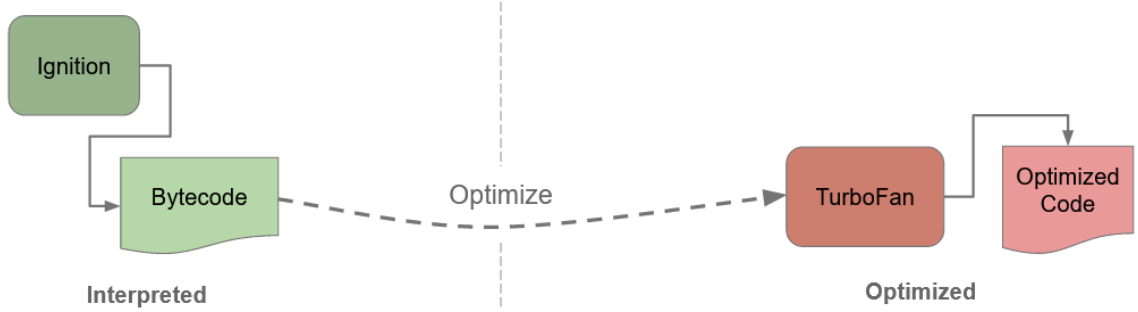
Ignition & Turbofan

Permasalahan kompleksitas, penggunaan memori yang berlebihan dan kecepatan masih menjadi masalah serius pada versi sebelumnya. Untuk mengatasi hal tersebut tim pengembang *v8 javascript engine* membuat *javascript interpreter* terbaru bernama ***Ignition***.



Gambar 333 Baseline Compiler Removed

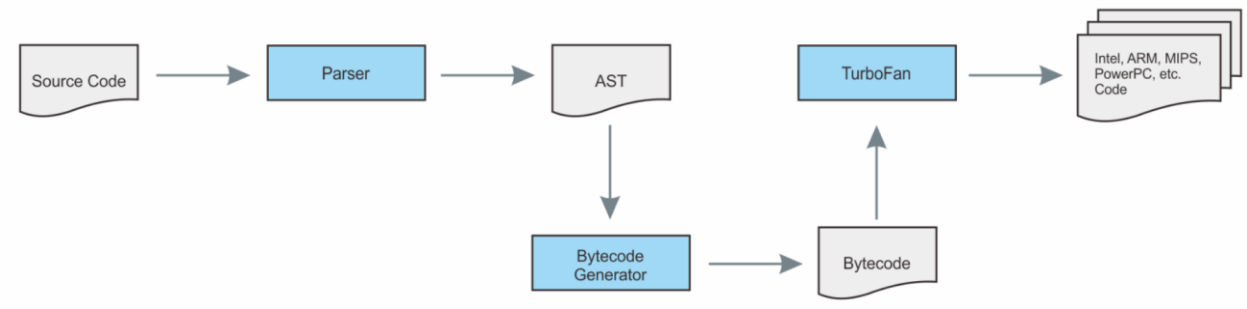
Baseline compiler akan diganti oleh *ignition* agar eksekusi kode dapat mengurangi konsumsi memori dan menyederhanakan alur eksekusi pada *pipeline*. Selain itu *Optimized Compiler* seperti *Crankshaft* akan digantikan oleh *TufboFan Compiler*.



Gambar 334 Brand New Pipeline

Dengan ***ignition***, *V8 Javascript Engine* akan memproduksi kode *javascript* kedalam bentuk ***intermediate representation*** yang disebut dengan ***bytecode***. *Intermediate*

representation (IR) adalah *data structure* yang merepresentasikan sebuah program antara *high-level programming language* dan *machine code*. Keuntungan memiliki *intermediate representation* adalah ***Turbofan*** dapat memproduksi *machine code* ke target arsitektur yang diinginkan.



Gambar 335 Compilation Process

Saat *V8 Javascript Engine* melakukan kompilasi kode *javascript*, program *parser* yang dimiliki *V8 Javascript Engine* akan memproduksi ***Abstract Syntax Tree (AST)***. *Syntax tree* merepresentasikan *syntactic structure* dari kode *javascript* yang telah dikompilasi dalam bentuk ***tree data structure***.

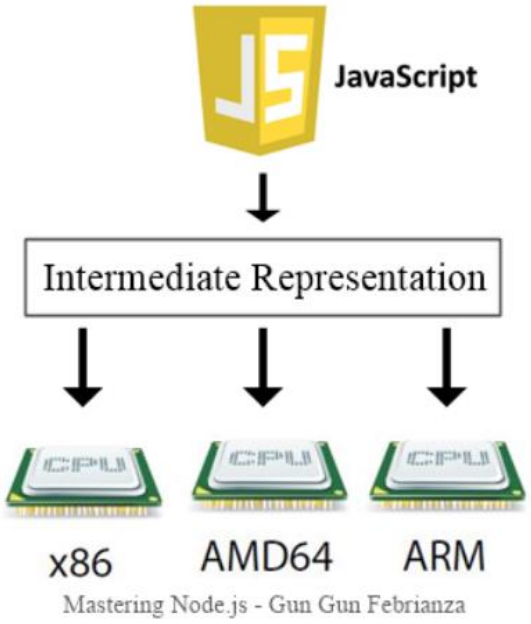
Ignition sebagai program *interpreter*, akan menggunakan *syntax tree* tersebut untuk memproduksi *bytecode*. ***TurboFan*** sebagai *optimized compiler*, akan mengambil *bytecode* dan memproduksi kode mesin yang telah di optimasi (*optimized machine code*).

TurboFan telah didesain dari awal untuk mendukung seluruh bahasa *javascript* saat ini, mulai dari ES 5 dan memperkenalkan desain kompiler yang memisahkan antara *high-level* dan *low-level compiler optimization*.

Pemisahan yang rapih pada *turbofan* melalui *layer* yang dibagi menjadi 3, dimana bahasa javascript sebagai *source-level language* (JavaScript), kemampuan *javascript runtime engine* (V8) dan arsitektur komputer memberikan banyak keuntungan.

Seluruh *programmer* yang ingin ikut mengembangkan dapat fokus pada masing-masing *layer* yang sudah di segmentasi. *Engineer* yang bekerja di *ARM*, *Intel, MIPS*, dan *IBM* bisa ikut berkontribusi dalam pengembangan *TurboFan*.

Intermediate Representation (IR)



Gambar 336 Intermediate Representation

Sebelumnya saat *V8 Javascript Engine* masih menggunakan *crankshaft* pada versi 5.8, tim pengembang V8 harus menulis *architecture-specific code* atau bahasa *assembly* untuk 9 *platform* yang didukung seperti *windows x86, windows x64, linux x64, linux-arm x64, solaris x64* dan lain lainya yang anda bisa lihat di dokumentasi *node.js*.

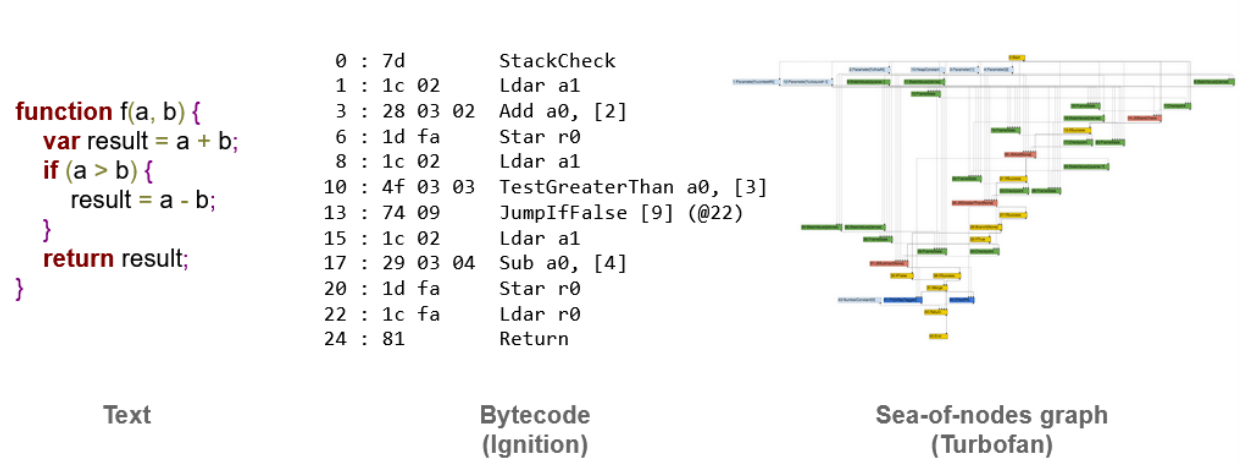
Tim pengembang V8 juga harus memelihata lebih dari 10 ribu baris kode untuk masing-masing *chip architecture,* setiap kali pengembangan baru dibuat semuanya harus di *port* ke *architecture* yang didukung. Artinya setiap ***bahasa assembly*** untuk setiap *architecture* harus disediakan ketika pengembangan baru dibuat. Benar benar merepotkan ya?

Jadi langkah memproduksi *bytecode* sebagai *intermediate representation* sudah tepat, kita bisa dengan mudah mentargetkan *optimized machine code* untuk *platform* yang didukung oleh *V8 Javascript Engine*.

Memproduksi *bytecode* juga lebih cepat jika dibandingkan dengan *full-codegen* pada *baseline compiler* sebelumnya. *Bytecode* dapat digunakan untuk memproduksi *optimized machine code* menggunakan *turbofan* secara langsung. *Bytecode* menyediakan eksekusi model yang lebih *clean* dan tidak rentan kesalahan (*less error-prone*) saat melakukan *deoptimization*.

Ukuran ***bytecode*** lebih ringan 50% sampai 25% jika dibandingkan dengan ukuran *machine code* yang diproduksi dari kode *javascript* yang ***equivalent***.

Bytecode



Gambar 337 Javascript Code, Bytecode, & TurboFan

Tertarik untuk mengembangkan bahasa pemograman tingkat tinggi dengan *bytecode node.js*? di bawah ini adalah *list* lengkap *bytecode node.js* :

https://github.com/v8/v8/blob/master/src/interpreter/bytecodes.h

Just-in-Time Compilation

Disini anda dapat memahami bahwa, *JIT Compilation* memiliki ciri khas yaitu proses kompilasi cenderung menggunakan sebuah *intermediate representation* (*bytecode*) ke dalam bahasa mesin (*machine code*). Ekesekusi dilakukan segera setelah kompilasi dilakukan. *V8* Sebagai *Javascript Engine* memiliki *profiler* yang dapat menganalisa kode yang sedang dieksekusi dan melakukan optimasi *machine code* dengan cara melakukan *re-compilation*.

Compiler Development Philosophy

Kompiler akan selalu tumbuh dan berkembang menjadi suatu program yang kompleks setiap kali fitur baru untuk bahasa pemrograman ditambahkan dan munculnya arsitektur komputer yang baru. Dalam hal ini *ECMAScript* sering kali melakukan penambahan fitur pada bahasa *javascript*.

Lebih detail lagi dapat dilihat dokumentasinya disini :

https://github.com/thlorenz/v8-perf/blob/master/compiler.md#sea-of-nodes

3.*Memory Management*

Pada bahasa pemrograman seperti C, ***memory management*** dilakukan secara manual menggunakan *method* **`malloc(), calloc(), realloc()`** dan **`free().`** Pada *javascript, engine* akan secara otomatis mengalokasikan memori pada ***Heap*** saat ***object*** dibuat. *Heap* adalah daerah memori tempat alokasi untuk menyimpan *objects*. Ketika *object* sudah tidak digunakan lagi memori dalam *heap* akan dibersihkan oleh ***Garbage Collector.***.

Memory Lifecycle

Apapun bahasa pemrogramanya cara kerja *memory lifecycle* selalu sama :

1. Membuat alokasi memori dalam heap
2. Menggunakan memori yang telah dialokasikan untuk membaca (*read*) & menulis (*write*)
3. Memori yang telah dialokasikan akan dibebaskan kembali.

Allocation Example

Javascript mempermudah *programmer* untuk tidak pusing memikirkan masalah alokasi memori, semuanya dilakukan secara otomatis dibelakang layar. Alokasi memori dilakukan saat anda memberikan nilai kedalam variabel yang telah anda buat.

Perhatikan kode di bawah ini :

```
var g = 123; // membuat alokasi memori untuk number
var f = 'Maudy Ayunda; // membuat alokasi memori untuk string

var object = {
  a: 1,
  b: null
}; // alokasi memori untuk object dan nilai yang digunakan
```

```
// alokasi memori untuk array dan elemen yang digunakanya
var array = [1, null, 'hazna'];

function f(x) {
  return x + 10;
} // alokasi memori untuk fungsi
```

Garbage Collector

Javascript Engine memiliki ***Garbage Collector*** yang bertugas untuk mengetahui dan melacak memori yang sudah tidak digunakan lagi sehingga *memory leak* tidak terjadi. *V8 Javascript Engine* menggunakan algoritma ***Mark-and-Sweep*** untuk menyelesaikan permasalahan *automatic memori management*.

Mark-and-Sweep Algorithm

Mark-and-Sweep algorithm digunakan untuk membebaskan memori saat *object* sudah tidak lagi bisa di jangkau, pertama *garbage collector* akan mencari ***global object/root object*** lalu mencari semua *references object* yang mengarah ke *global object*, setiap *references object* yang ditemukan akan dicari lagi *references* yang mengarah ke *object* tersebut dan seterusnya.

Dengan algoritma tersebut, *garbage collector* dapat mengidentifikasikan mana *object* yang bisa dijangkau (*reachable*) dan *object* yang tidak bisa dijangkau (*unreachable*). Seluruh *unreachable objects* akan secara otomatis dibersihkan.

Pada tahun 2012, seluruh *javascript engine* yang ada sudah memanfaatkan *mark-and-sweep algorithm* untuk *garbage collector*.

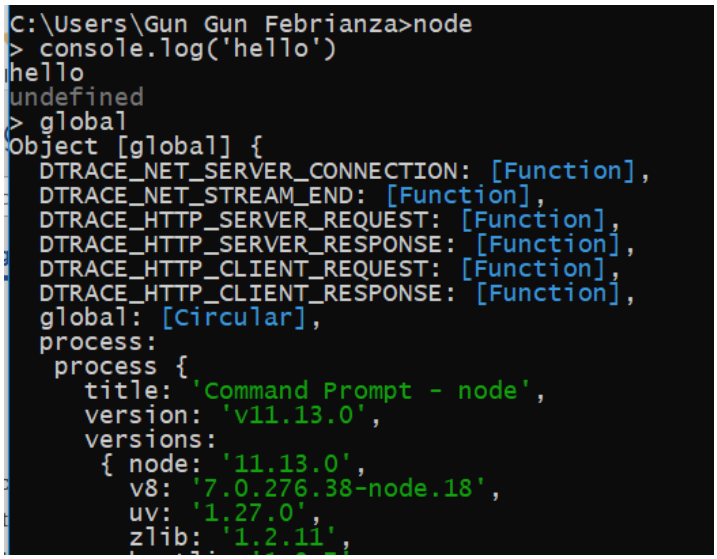
Root Object

Pada *browser* yang dimaksud dengan *global* atau *root object* adalah *"window" object* dan pada *Node.js* adalah "global". Bagi *programmer* yang sebelumnya pernah mempelajari dan sering melakukan ***DOM Manipulation*** pasti akan sangat mengenal *window object*.

Jika anda ingin mengetahui lebih lengkap buka *browser firefox* kemudian tekan tombol **CTRL + SHIFT + K**, kemudian ketik `console.log(window)` maka akan muncul *object* tertinggi dalam *browser* yaitu *window object* :

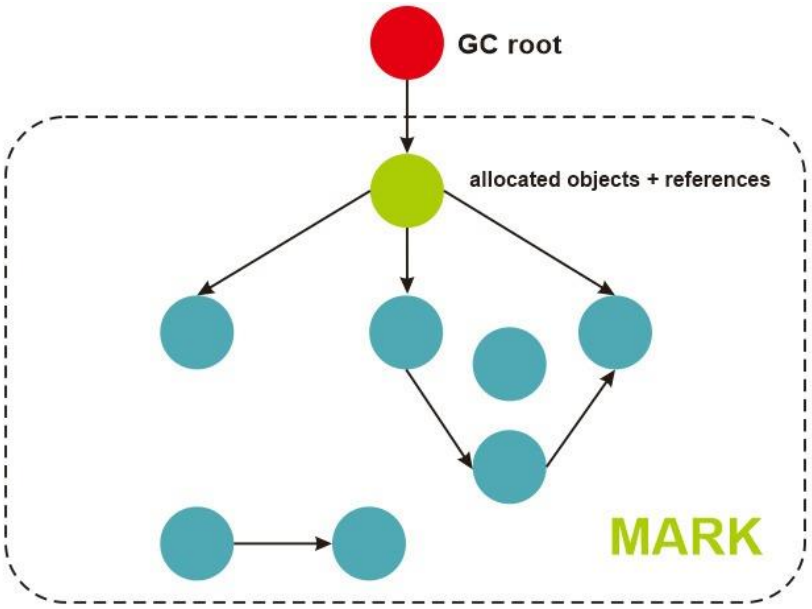
Gambar 338 Window Object

Pada *node.js* melalui *command prompt* ketik saja langsung global :

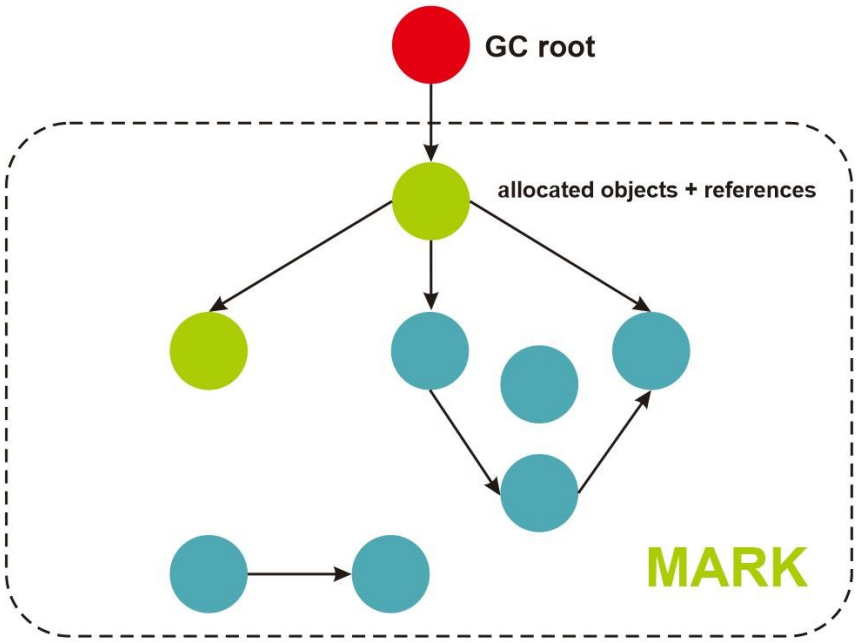


Gambar 339 Global Object

Pada gambar di bawah ini terdapat visualisasi bagaimana cara kerja *garbage collector* dalam melacak *object* yang *unreachable*.



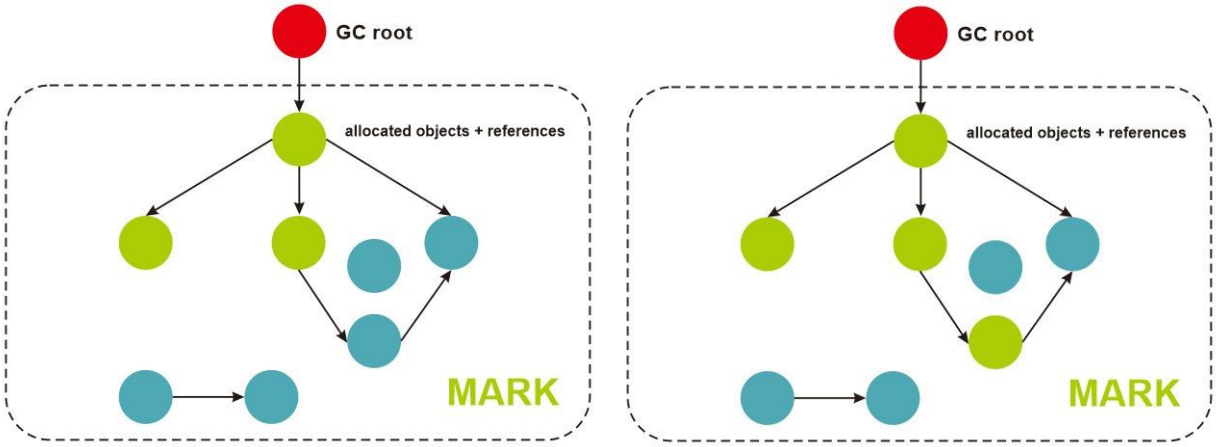
Gambar 340 Mark Phase 1



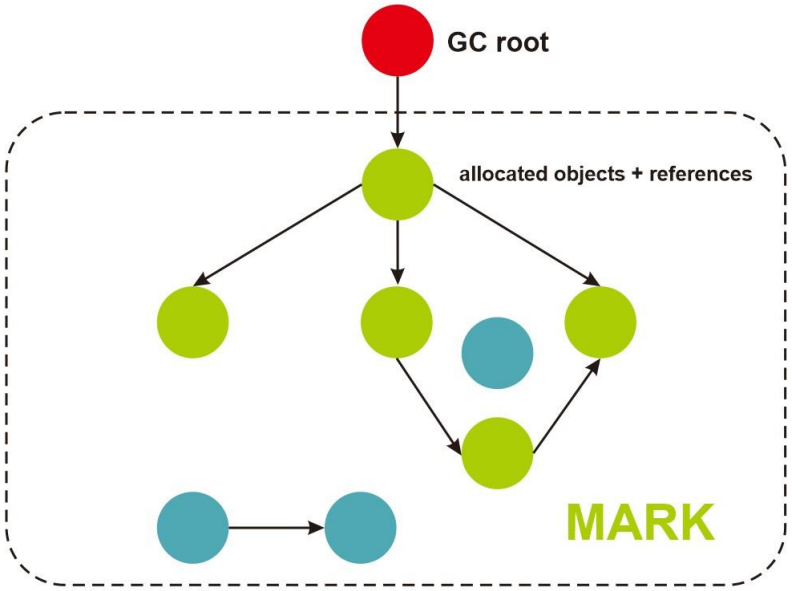
Gambar 341 Mark Phase 2

Setiap *root* akan diinspeksi sampai ke *childern* dan menandainya sebagai *object active*.

Seluruh *childern* akan di tandai sebagai *object active*.

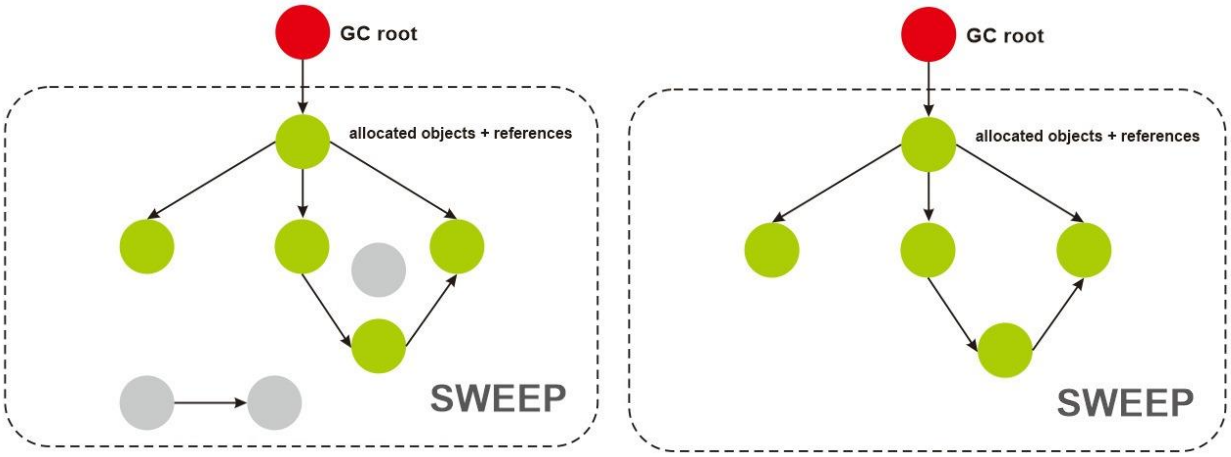


Gambar 342 Mark Phase 3



Gambar 343 Mark Phase 2

Seluruh *object* yang tidak terjangkau (*unreachable*) akan dibersihkan, memori dibebaskan dan dikembalikan lagi pada sistem operasi.



Gambar 344 Sweep Phase

Begitulah cara kerja algoritma *mark-and-sweep* secara *high level view*.

Memory Leak

Memory leak adalah kumpulan memori yang telah digunakan oleh suatu aplikasi, namun memori tersebut sudah tidak dibutuhkan lagi dan tidak kita bebaskan kembali untuk diberikan kepada *pool of free memory* dalam sistem operasi.

Memory leak juga dapat bermakna memori yang gagal dibebaskan setelah alokasi dilakukan diakibatkan dari ***bug*** suatu program atau *poor code* yang dibuat oleh seorang *programmer*, sehingga terjadi konsumsi memori yang tidak diinginkan / tidak diketahui.

Case Study - Global Variable

Pada *javascript* sendiri *memori leak* bisa terjadi pada *global variable* :

Jika kita membuat variabel yang tidak dideklarasikan di dalam *javascript*, lalu variabel tersebut digunakan (bahasa teknisnya *referenced*) maka *javascript engine* akan memperlakukan variabel tersebut sebagai *global object*. Perhatikan kode di bawah ini :

```
function foo(arg) {
  bar = "some text";
}
```

Pada kode *javascript* di atas, variabel bar digunakan tanpa dideklarasikan terlebih dahulu. Jika kita menggunakan kode di atas dalam *browser* maka *javascript engine* akan menganggapnya sebagai *window object* yang memiliki ukuran memori yang cukup besar. Pada *node.js, javascript engine* akan menganggapnya sebagai *global* sehingga menajdi redundan.

Kode di atas akan bekerja dibelakang layar menjadi seperti ini :

```
function foo(arg) {
```

```
    window.bar = "some text";
}
```

Case Study – this keyword

```
function foo() {
  this.bar = "potential accidental global";
}
```

Pada kasus di atas karena **`bar`** tidak dideklarasikan maka **`this`** akan mengacu pada *global object*. Secara dasar harusnya menjadi ***undefined***, tapi begitulah cara kerja *semantic analysis* pada *javascript engine*.

Notes
Untuk mencegah hal ini terjadi pastikan anda menggunakan *Strict Mode* dalam menulis kode *javascript*. *Strict mode* akan membuat *javascript engine* mencegah kita membuat global variabel.

Subchapter 3 – Node.js Application

Think twice, code once.

— Waseen Latif

Subchapter 3 – Objectives

- Memahami Cara Eksekusi ***Javascript File***?
- Memahami Cara Eksekusi ***Node REPL***?
- Memahami Apa itu ***Module Concept***?
- Memahami Apa itu ***Node.js Module***?
- Memahami Apa itu ***Package Manager***?
- Memahami Apa itu ***Node Package Manager***?
- Memahami Cara Membuat ***Node.js Package***?
- Memahami Cara Publich ***Node.js Package***?
- Memahami Apa Menggunakan ***Node.js Package***?

1. *Running Javascript File*

Untuk mengeksekusi *javascript file* dengan *node.js* sangatlah mudah.

Buatlah sebuah *file* dengan nama **`1.hello.js`**, kemudian tulis kode seperti di bawah ini :

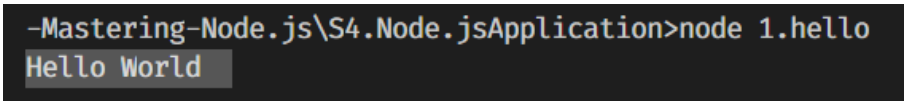
```
console.log("Hello World");
```

Untuk mengeksekusi kode *javascript* di atas dengan *node.js* eksekusi perintah di bawah ini :

```
node 1.hello
```

Node.js Javascript engine akan membaca *file javascript* dan memberikan *output* pada *terminal*.

Di bawah ini adalah contoh hasil dari eksekusi kode di atas :



Gambar 345 Node.js First App

Selamat anda berhasil mengeksekusi sebuah *node.js application*.

2. Node REPL

Buatlah sebuah *file* dengan nama **2.list-command** dan tulis kode di bawah ini :

```
  let node_command = [".help", ".load", ".save", ".break", ".exit
"];
```

Pada terminal eksekusi perintah **node** :

```
node

Welcome to Node.js v12.16.1.

Type ".help" for more information.
```

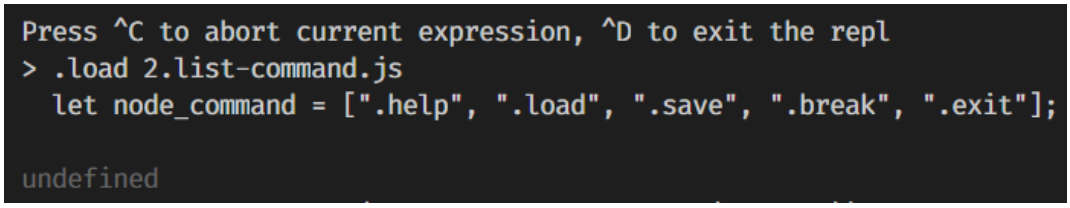
Kemudian ketik ***.help*** agar kita bisa mengetahui seluruh perintah yang tersedia :

```
> .help

.break    Sometimes you get stuck, this gets you out

.clear    Alias for .break

.editor   Enter editor mode

.exit     Exit the repl

.help     Print this help message

.load     Load JS from a file into the REPL session

.save     Save all evaluated commands in this REPL session to a file


Press ^C to abort current expression, ^D to exit the repl
```

Dengan perintah **.load** kita akan memuat *variable* **node_command** kedalam *REPL* :

```
> .load 2.list-command.js
```

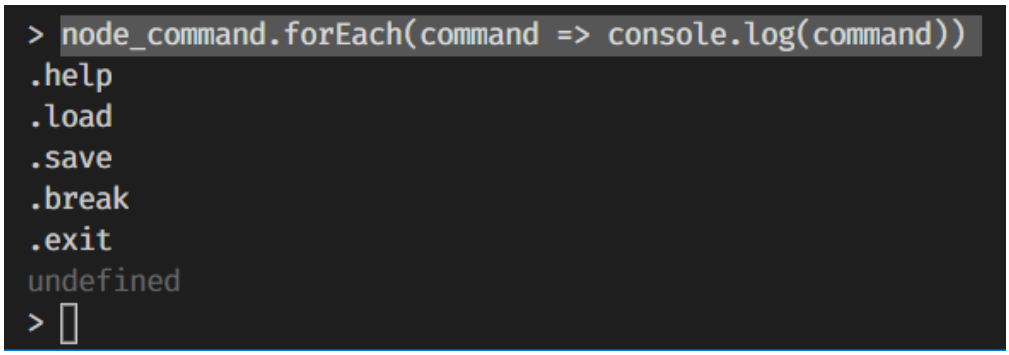
Eksekusi perintah tersebut di dalam *terminal* maka akan memproduksi *output* sebagai berikut :



Gambar 346 .load Command

```
> node_command.forEach(command => console.log(command))
```

Variable **`node_command`** merupakan sebuah *array* sehingga kita dapat mengeksekusi *method* **`forEach()`**. Di bawah ini adalah hasilnya :



Gambar 347 forEach Method Result

3. *Module Concept*

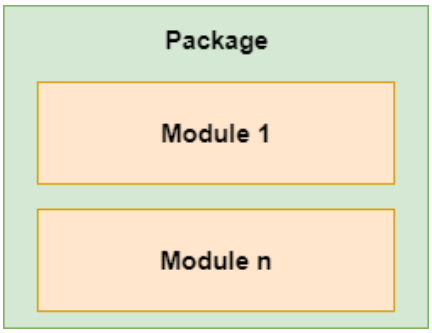
Dalam konteks buku ini terdapat beberapa terminologi :

Modules

Modules adalah sebuah *file javascript* tunggal yang merepresentasikan suatu fungsionalitas atau *library*.

Packages

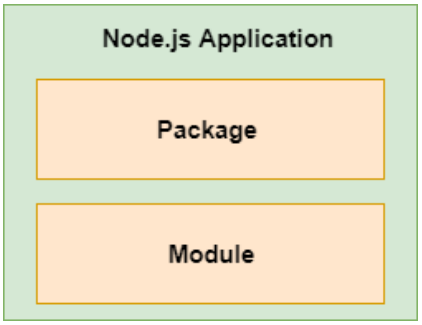
Sebuah *packages* dapat memiliki sekumpulan *modules* atau hanya memiliki *module* tunggal.



Gambar 348 Package Illustration

Dependencies

Dependency artinya diperlukan, sebuah *node.js application* dapat memiliki sebuah *dependencies* yang terdiri dari *module* atau *package*. Jika sebuah *package* dikategorikan sebagai *dependency* suatu *node.js application* maka *package* tersebut harus ditanam (*install*) agar aplikasi *node.js* bisa berjalan.



Gambar 349 Dependencies Illustration

4. *Node.js Module*

Modules is a reusable piece of code that encapsulates implementation details.

Module sederhananya adalah sekumpulan kode yang dapat kita gunakan lagi dan lagi untuk memempercepat pengembangan aplikasi. Sebuah *node.js application* dibangun oleh sekumpulan *module*, sehingga *module* adalah *basic block* sebuah *node.js application*.

Sebuah *modules* ditulis oleh :

1. *Node.js Developer* lainnya
2. *Third-party Library*
3. Kita membuat dan mempublikasi *module* untuk orang lain

Sekumpulan *modules* yang ditulis dapat di muat kedalam *node.js application* agar dapat digunakan.

Sebuah *module* harus memberikan kita :

1. ***Abstraction***

   Kita dapat mendelegasikan sebuah fungsionalitas pada sekumpulan module yang diciptakan khusus untuk permasalahan tertentu, tanpa harus memahami kompleksitas cara untuk implementasinya secara aktual.

2. ***Encapsulation***

   Untuk membungkus dan mengklasifikasikan sekumpulan kode (*block of code*) atau sekumpulan *statements* tunggal dalam sebuah *module*.

3. ***Reuse Code***

   Untuk mencegah penulisan kode yang sama lagi dan lagi.

4. ***Manage Dependencies***

   Memudahkan kita untuk memanajemen *dependencies* tanpa harus menulis ulang kode.

Module Format

Sebuah *module* juga memiliki *module format* yaitu *syntax* yang digunakan untuk membuat sebuah *module*. Terdapat *module format* yang dapat digunakan dalam *node.js* yaitu :

Universal Module Definition (UMD)

Format ini dapat digunakan di dalam *browser* dan *node.js*.

CommonJS (CommonJS)

Format ini digunakan di dalam *Node.js* cirinya penggunaan *keyword* `require` dan `module.exports` untuk menentukan sebuah *dependencies* dan *modules*.

Module Loaders

Module loader adalah sebuah program yang akan membaca sebuah *module* yang memiliki format tertentu. Terdapat dua *module loader* yang sangat populer :

RequireJs

Sebuah *module loader* untuk *module* yang ditulis dengan format *AMD* (*Asynchronous Module Definition).*

SystemJs

Sebuah *module loader* untuk *module* yang ditulis dengan format *AMD* (*Asynchronous Module Definition)*), *CommonJS, UMD* atau *system.register format*.

Dalam buku ini kita akan belajar membuat *module* menggunakan *CommonJS Format*.

Module Bundlers

Sebuah *module bundler* akan mengganti peran *module loader*, sebuah *module bundler* berjalan saat *build time*. Kita akan menggunakan program *module bundler* untuk memproduksi *bundle file*.

Terdapat dua *module bundlers* yang populer :

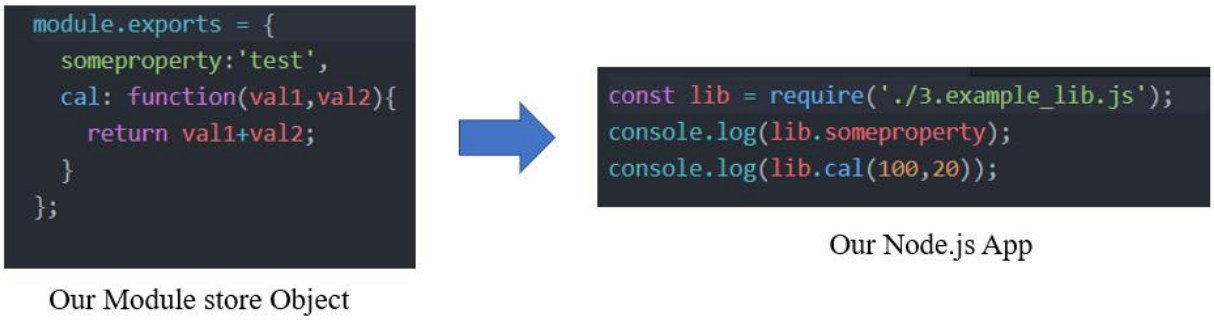
Browserify

Module bundler untuk *CommonJS Modules*

Webpack

Module bundler untuk *AMD, CommonJS, & ES 6 Module*.

Perhatikan pada gambar di bawah ini :



Gambar 350 Module Implementation

Pada gambar sebelah kiri kita membuat sebuah *module* yang memiliki *property* dan *method* tunggal. Pada gambar sebelah kanan terdapat *node.js application* yang kita buat sedang memuat *module* dan menggunakan *property* dan *method* dalam *module* tersebut.

Selain membuat *module* sendiri *node.js* juga telah menyediakan *built-in module* yang siap untuk kita gunakan dalam pengembangan aplikasi.

Tujuan dari konsep *modules* adalah konsep *reusable code, Node.js* sendiri telah menyediakan sekumpulan *built-in modules* yan dapat kita lihat disini :

https://nodejs.org/api/

Seluruh informasi seperti dokumentasi kode untuk setiap *modules* telah disediakan disana.

Gambar 351 Node.js Built-in Module

Sebagai contoh pada gambar di atas terdapat *built-in module file system* yang dapat kita gunakan untuk mengelola *file* dan berinteraksi dengan sistem operasi.

Module file system memiliki *method* **`readdir`** untuk membaca suatu *directory*, kita dapat menggunakan *module* tersebut untuk membuat sebuah aplikasi *node.js* sederhana untuk membaca suatu *directory* :

```
const fs = require('fs');

fs.readdir('./', 'UTF-8', (err, content) => {
  if (err)
    return err;
  console.log(content);
})
```

Di bawah ini adalah *output* yang dihasilkan dari *script* di atas pada komputer penulis :

```
e:\13. The Kaizer Arsenal -
[
  '.gitignore',
  '1.file_reading.js',
  '2.file_writting.js',
  '3.ensureDir.js',
  '4.remove.js',
  '5.move.js',
  'global.html',
  'node_modules',
```

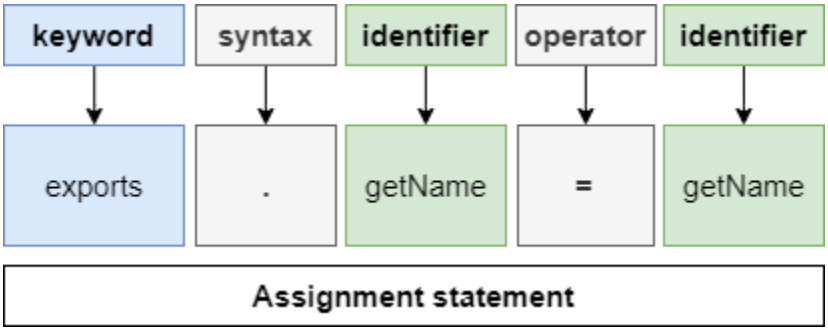
Gambar 352 Output readdir method

Create & Export Module

Kita akan membuat sebuah *module* dengan nama ***simple-module*** :

```
const getName = () => {
  return "Maudy Ayunda";
};

exports.getName = getName;
```

Dalam *module* di atas kita hanya memiliki satu *method* yaitu **`getName()`** dan kita akan menggunakan *keyword* **`exports.getName = getName`** untuk menegaskan bahwa **`getName()`** adalah *method* yang akan di *export* agar bisa digunakan oleh aplikasi *node.js* lainnya.



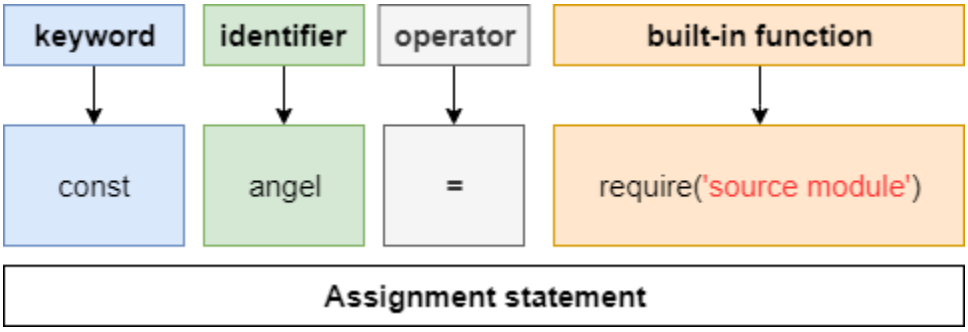
Gambar 353 Syntax Rule Export Module

Use Module

Kita akan membuat aplikasi *node.js* yang akan menggunakan ***simple-module***, buatlah *file* ***simple-app.js*** dan tulis kode di bawah ini :

```
const angel = require("./3.simple-module.js");
```

Pada kode di atas *assignment statement* digunakan agar *module* dapat disimpan dalam sebuah *variable*. Kita akan menggunakan *built-in function* **`require`** digunakan untuk menentukan dimana lokasi *module* disimpan.



Gambar 354 Module Syntax

Kita menggunakan *keyword* **`const`** untuk deklarasi variabel sebagai *best practice* untuk mencegah kesalahan *coding* yang dapat memodifikasi *object* tempat *module* akan disimpan. Selanjutnya dengan *variable* **`angel`** kita dapat menggunakan *method* yang dimiliki oleh *simple-module* :

```
console.log(angel.getName());
// Maudy Ayunda
```

**Link* sumber kode.

Export Multiple Method & Value

Pada *simple-module* kita dapat menambahkan sekumpulan *method* dan *value*. Perhatikan kode di bawah ini :

```
const getName = () => {
  return "Maudy Ayunda";
};

const getHerSong = () => {
  return "Perahu Kertas";
};

const birthday = "19 December 1994";

exports.getName = getName;
exports.getHerSong = getHerSong;
exports.birthday = birthday;
```

Pada kode di atas kita menambahkan *variable* **getHersong** dan **birthday** agar dapat digunakan oleh aplikasi. Sebelum dapat digunakan variabel tersebut harus di *export* terlebih dahulu.

Untuk menggunakan kode di atas ***simple-app.js*** sekarang dapat menggunakan *variable* **getHersong** dan **birthday** :

```
const angel = require("./3.simple-module.js");
console.log(angel.getName());
// Maudy Ayunda
console.log(angel.birthday);
// 19 December 1994
console.log(angel.getHerSong());
```

```
// Perahu Kertas
```

**Link* sumber kode.

Export Style

Terdapat dua cara yang bisa kita gunakan untuk melakukan *export* :

```
const getName = () => {
  return "Maudy Ayunda";
};

exports.getHerSong = () => {
  return "Perahu Kertas";
};

exports.getName = getName;
```

**Link* sumber kode.

Destructure Assignment

Kita juga dapat menggunakan *destructure* untuk menggunakan suatu *module* :

```
const { birthday, getName } = require("./3.simple-module.js");
console.log(`Maudy birthday is ${birthday}`);
// Maudy birthday is 19 December 1994
console.log(getName());
// Maudy Ayunda
```

**Link* sumber kode.

Export Class

Selain *method* dan *primitive value* kita juga dapat melakukan *export* sebuah *class*. Pada contoh kode di bawah ini kita mencoba melakukan *export* sebuah *class* :

```
module.exports = class Wallet {
  constructor(name, balance) {
    this.name = name;
    this.balance = balance;
  }
  topUp(newbalance) {
    return (this.balance += newbalance);
  }
};
```

**Link* sumber kode.

Buatlah sebuah *file* dengan nama ***class-module.js*** kemudian tulis kode di atas, selanjutnya buatlah *file* dengan nama ***wallet-app.js*** dan tulis kode di bawah ini :

```
const ClassWallet = require("./6.class-module.js");
const userGun = new ClassWallet("Gun", 1000);
userGun.topUp(1000);
console.log(userGun);
// Wallet { name: 'Gun', balance: 2000 }
console.log(userGun.balance);
// 2000
```

**Link* sumber kode.

Pada kode di atas kita membuat sebuah *object* bernama **userGun** yang berasal dari *class* **Wallet** dalam *file 6.class-module.js*. Selanjutnya kita bisa memanggil *method* **topup()** dan menggunakan *property* **balance**.

5. *Package Manager*

Berbicara tentang *package manager* ada kalimat yang menarik dari seorang pelukis terkenal asal belanda, pelukis yang hampir dalam satu dekade mampu menciptakan 2.100 karya. Bahwa :

Great things are done by a series of small things brought together.

—Vincent Van Gogh

Konsep tentang *package manager* sering juga disebut dengan *dependency manager*.

Dengan sebuah *package manager* kita bisa mengelola kumpulan *libraries* dalam sebuah *project*. Ada beberapa *package manager* di antaranya adalah :

1. *npm*: sebuah *package manager* untuk *Node.js*
2. *Composer*: sebuah *package manager* untuk *PHP*
3. *pip*: sebuah *package manager* untuk *Python*
4. *NuGet*: sebuah *package manager* untuk untuk *Microsoft development platform*

Package manager mempermudah cara berbagi kode (*share*) dan menggunakan kembali suatu kode (*re-use*). Banyak sekali *developer* berbagi kode yang mereka buat untuk menyelesaikan masalah tertentu, sehingga *developer* lainya bisa ikut menggunakan dan mengembangkanya.

Saat kita telah bergantung pada kode yang dibuat oleh seorang *developer*, *package manager* akan mempermudah *developer* lainya untuk menerima informasi jika ada pembaharuan (*update*) dalam kode tersebut.

Setiap *reusable code* yang dibagikan disebut dengan ***package*** atau ***module***. Sebuah *package* terdiri dari 1 atau lebih *directory* yang di dalamnya juga bisa terdapat 1 atau lebih *file*. Sebuah *package* juga memiliki ***file metadata***, pada **npm** milik *node.js file metadata*

dibuat dalam bentuk JSON. Sebuah aplikasi dalam suatu *project* pasti tersusun dari ratusan atau ribuan *packages*.

Sebuah *package manager* selalu memiliki *package repository* yang bisa kita gunakan untuk mencari *packages* yang dibuat oleh para *developer*. Di dalamnya kita bisa mengeksplorasi kumpulan *packages* yang bisa anda gunakan dalam *project* yang anda buat.

Pada *Node.js* juga terdapat *package repository* yang disebut dengan *Node Package Registry* :

https://www.npmjs.com/

6. *Node Package Manager*

Gambar *NPM Logo*

npm adalah singkatan dari *Node Package Manager* dan ***npm*** adalah *package manager* untuk *javascript platform*. Dengan *Node Package Manager* kita bisa melakukan instalasi sebuah *module*. Sebuah *module* bekerja seperti *package library* yang dengan mudah bisa dibagikan (*shared*), digunakan ulang (*reused*) dan di tanam (*installed*) diberbagai *project* yang akan anda buat.

Setiap *module* yang telah kita *install* akan tersimpan dalam sekumpulan *node_modules*, yang dapat membantu kita mengatasi konflik *libraries* dengan menyediakan *dependency management*.

Jika *node.js* sudah terinstal anda dapat mengujinya dengan mengeksekusi perintah berikut **:**

```
> node –version

v12.16.1
```

Perintah dasar pada ***npm*** memiliki *format* sebagai berikut :

```
> npm <command> [args]
```

npm commands

Di bawah ini adalah sekumpulan *npm commands* yang akan kita pelajari :

Table 13 npm commands

npm command	***Description***	***Example***
npm init	Inisialisasi *node.js application* dan membuat *file package.json*.	npm init
npm install	Memasang sebuah *module* dalam sekup lokal	npm install typescript
npm install -g	Memasang sebuah *module* dalam sekup global	npm install typescript -g
npm search	Mencari sebuah *modules node.js*	npm search typescript
npm remove	Menghapus *module* yang telah dipasang	npm remove typescript
npm update	Memperbaharui *module* yang telah dipasang	npm update typescript -g
npm list	Menampilkan daftar *modules* yang telah dipasang	npm list
npm view	Menampilkan informasi *module* secara detail.	npm view typescript
npm start	Menjalankan aplikasi *node.js* yang telah dikofigurasi di dalam *package.json*	npm start
npm stop	Menghentikan aplikasi *node.js* yang sedang berjalan	npm stop
npm publish	Publikasikan *module* ke *Node Package Registry*.	npm publish mymodule

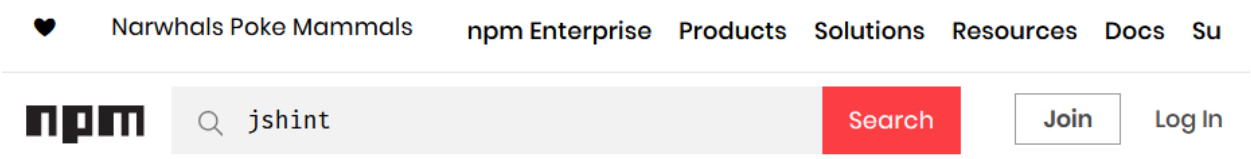
npm unpublish	Batalkan *module* yang telah dipublikasikan.	npm unpublish mymodule
npm docs	Membaca dokumentasi dari sebuah *module*.	npm docs typescript

7. Node Package Registry

Jika kita ingin mengetahui *package* apa saja yang tersedia di dalam *node.js* kita bisa mengunjungi *Node Package Registry* di :

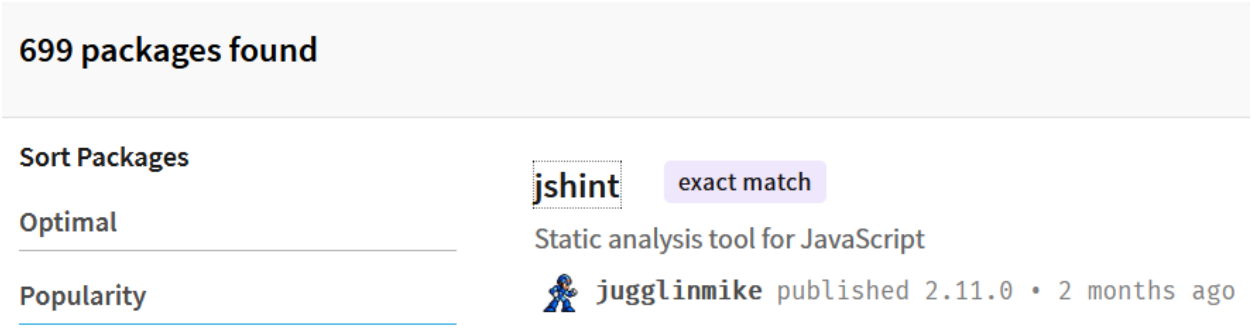
https://www.npmjs.com/

Jika kita ingin mencari sebuah *node.js modules*, cukup ketik nama *node.js modules* yang ingin diketahui seperti pada gambar di bawah ini :



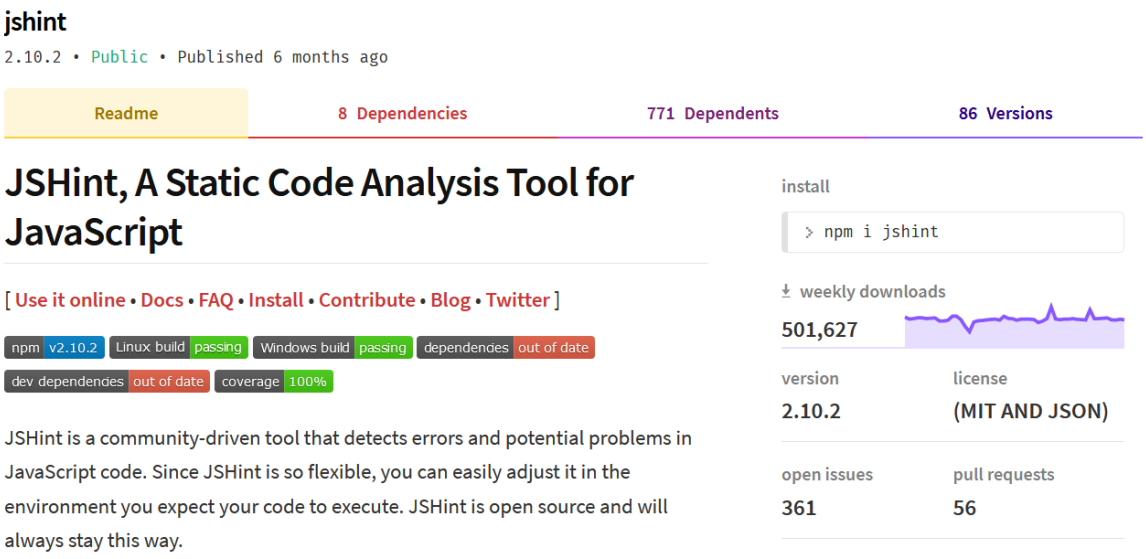
Gambar 355 Npmjs.com

Anda akan menemukan *module* tersebut dalam urutan pertama :



Gambar 356 jshint module

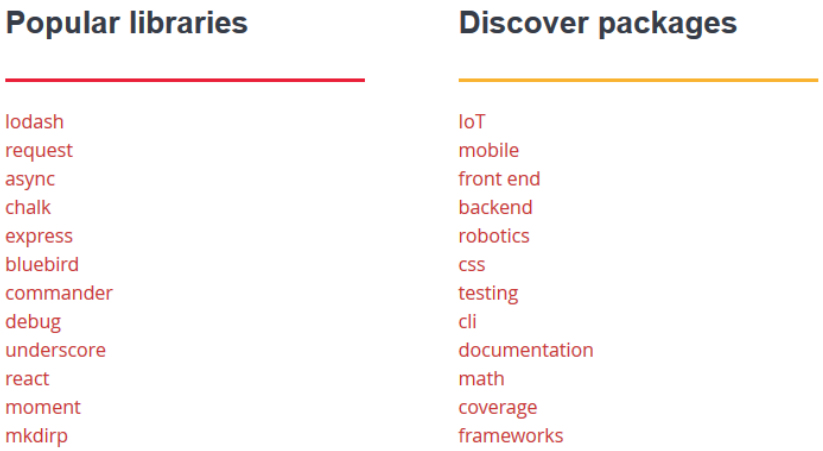
Setelah anda mendapatkan *node.js module jshint* selanjut anda bisa melihat lebih detail informasi dari *node.js module* tersebut :



Gambar 357 node.js module jshint

Node Package Registry akan menampilkan *modules* terbaru dan terpopuler. Setiap *package* terbaru jika ingin tampil disana harus didaftarkan terlebih dahulu.

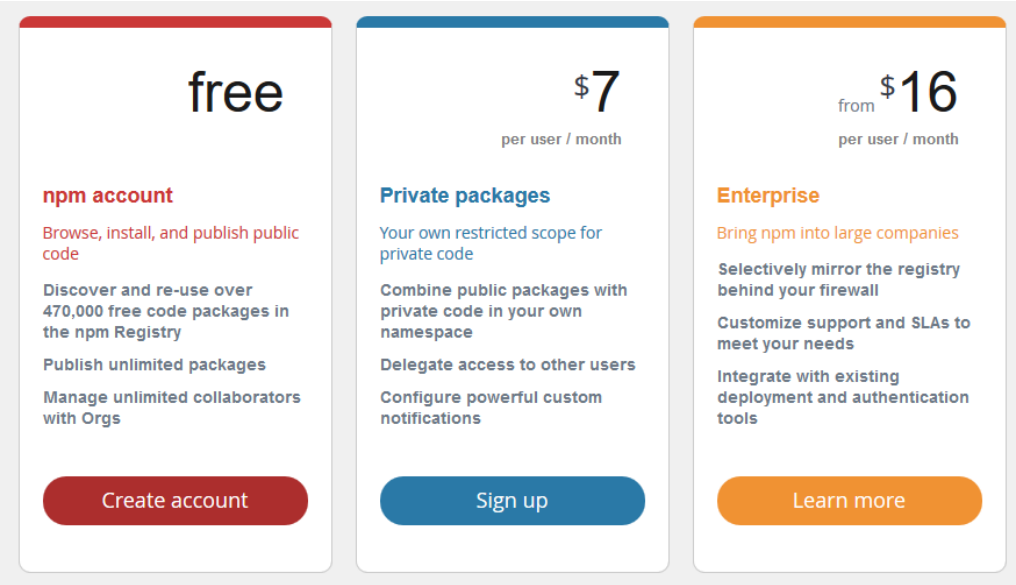
Di bawah ini adalah contoh *package* yang popular dan klasifikasinya :



Gambar 358 Registered Package

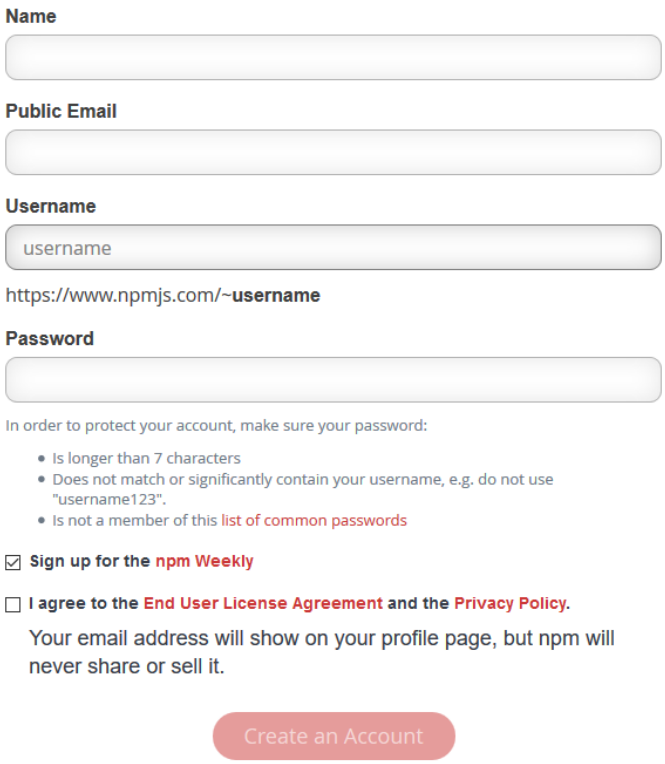
Untuk bisa mengeksplorasinya anda harus membuat akun terlebih dahulu di npmjs.com. Anda juga bisa membuat *private package* yang hanya bisa diketahui oleh anda dan orang-orang tertentu saja dengan biaya 7 *Dollar* perbulan/perpengguna.

Untuk membuat akun klik tombol ***create account*** :



Gambar 359 Buat Akun di Node Package Registry

Isi formulir pendaftaran dan konfirmasi pendaftaran akan dilakukan melalui *email*.



Gambar 360 Form Register npmjs

8. *Create Node.js package*

Setiap kali kita membuat aplikasi dengan *node.js*, *file package.json* akan selalu dibutuhkan untuk memanajemen *package* dalam aplikasi *node.js* kita.

Buatlah sebuah *folder* dengan nama **`project_x`**, kita akan belajar sebuah aplikasi *node.js*.

Pada *folder* **`project_x`** eksekusi perintah di bawah ini pada *command prompt* :

```
npm init
```

Jika berhasil maka anda akan menemukan *output* seperti pada gambar di bawah ini :

```
C:\project_x>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.
```

Gambar 361 npm init command

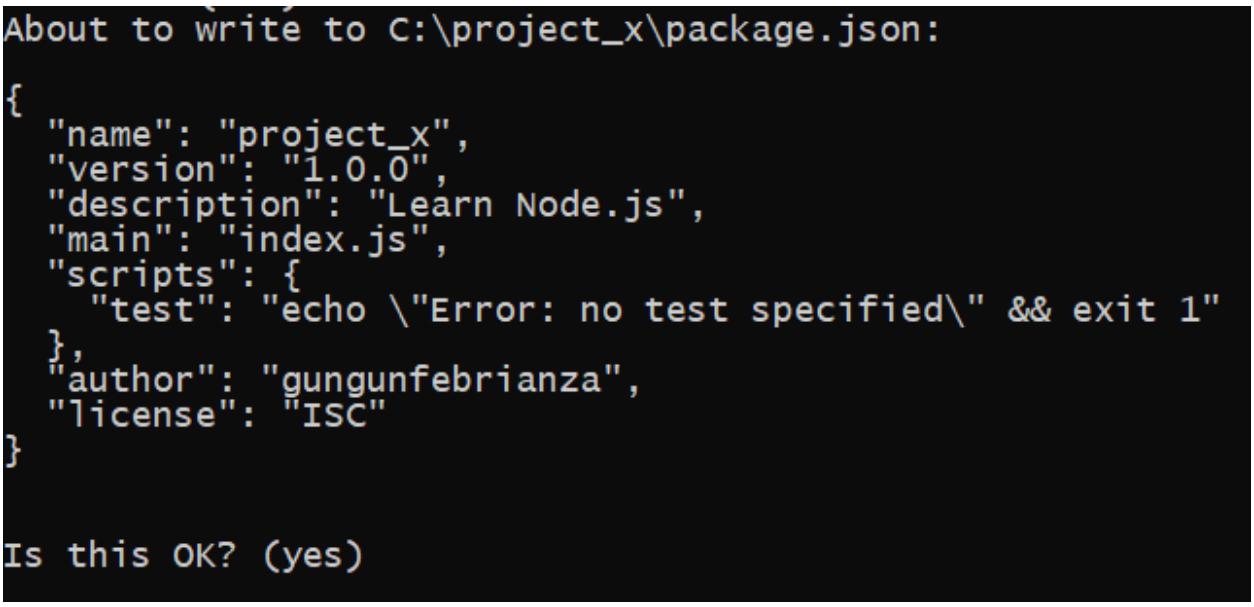
Saat pertama kali kita akan diberi kesempatan untuk mengisi *package name*, silahkan ketik nama *package* jika ingin *custom* namun pada kali ini kita ingin menggunakan nama *default* yaitu **`project_x`** jadi langsung saja tekan *enter* :

```
Press ^C at any time to quit.
package name: (project_x)
version: (1.0.0)
description: Learn Node.js
entry point: (index.js)
test command:
git repository:
keywords:
author: gungunfebrianza
license: (ISC)
```

Gambar 362 package.json setup

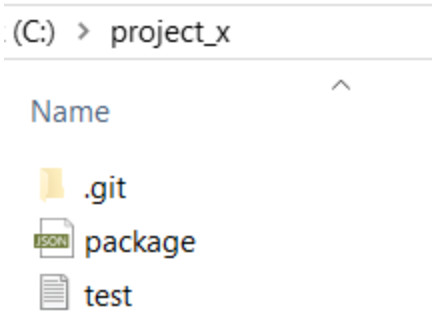
Terdapat informasi seperti *version* untuk menentukan versi dari *package* yang akan kita buat silahkan tekan *enter* untuk menyetujui versi 1.00, kemudian anda akan diminta untuk mengisi informasi *description* untuk menjelaskan dari *package* yang akan anda buat. Selanjutnya anda akan diminta untuk mengisi informasi dari *entrypoint package* yaitu tempat pertama kali *file javascript* akan dieksekusi jika *package* dijalankan, pada *test command, git repository* dan *keyword* silahkan dikosongkan dan *author* isi dengan nama anda, untuk *license* pilih *default* yaitu *ISC*.

Silahkan isi seperti pada gambar di atas :



Gambar 363 Package.json file

Ketik *yes* kemudan tekan tombol *enter*, maka di dalam *directory repository* **`project_x`** akan muncul *file package.json* :



Gambar 364 Generated Package.json

package.json

Isi dari *file package.json* adalah sebagai berikut :

```
{
  "name": "project_x",
  "version": "1.0.0",
  "description": "Learn Node.js",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "gungunfebrianza",
  "license": "ISC"
}
```

Isi dari *package.json* adalah sekumpulan *directive*.

Directive

Di bawah ini adalah tabel yang menjelaskan *directive* dalam *package.json* :

Table 14 Directive on Package.json

Directive	***Description***	***Example***

name	Nama untuk *package* yang dibuat, wajib diisi dengan huruf kecil semua tanpa spasi dan maksimal tidal lebih dari 214 karakter.	`"name": "name-must-unique"`
version	Versi dari *package* yang dibuat, wajib diisi dengan aturan ***Semantic Versioning***. Format X.X.X dengan urutan pertama untuk ***Major Update***, urutan kedua untuk ***Minor Update*** dan urutan ketiga untuk ***Bug***. Contoh versi 3.2.5	`"version": "1.0.0"`
description	Penjelasan dari *package* yang dibuat, masukan beberapa *keyword* yang dapat mempermudah orang lain mencari *package* anda.	`"description": "new package"`
main	Sebuah *entrypoint* untuk aplikasi yang dibuat, bisa berupa program biner atau *javascript file*.	
author	Nama dari pembuat *package,* dibuat untuk satu orang saja.	`"author": "Gun"`

contributors	Nama untuk para *contributor* yang ikut mengembangkan *package,* dibuat untuk lebih dari satu orang.	`{ "name" : "nikolaj"`
		`, "email" :`
		`"nikolaj@marketkoin.com"`
		`, "url" :`
		`"https:// marketkoin.com/"`
		`}`

dependencies	*Modules* dan versi dari *module* yang bergantung pada *package* yang dibuat. Kita bisa menggunakan notasi *wilcard* dengan * dan x	{ "dependencies" : { "jshint" : "1.0.0" , "cli" : "~1.2.3" , "openssl" : "2.x" } }
license	Lisensi yang akan digunakan untuk *package*.	"license": "MIT"
bin	Program biner yang ingin ikut tertanam bersama *package* yang dibuat.	
repository	Tipe *repository* yang digunakan dan informasi *package location*.	
homepage	Alamat pengenal untuk *package* yang dibuat.	"homepage": "https://www.marketkoin.com"

Setiap *node.js module* memiliki *package.json* yang menjelaskan *node.js module* itu sendiri. Pada *file* tersebut terdapat informasi ***metadata*** mengenai *name*, *version*, *author* dan *contributor*.

Selain itu didalamnya juga terdapat informasi *dependencies* dan informasi lainya yang dibutuhkan oleh *node package manager* (*NPM*) untuk melakukan instalasi dan publikasi *node.js module* agar bisa digunakan oleh pengguna lainya.

Search Package

Dengan *npm* kita dapat melakukan pencarian sebuah *node.js modules,* silahkan eksekusi perintah di bawah ini pada *command prompt* :

```
npm search underscore
```

Pastikan anda terhubung ke internet untuk mendapatkan hasilnya :

```
NAME              | DESCRIPTION         | AUTHOR         | DATE       | VERSION  | KEYWORDS
underscore        | JavaScript's…       | =jashkenas…    | 2020-01-06 | 1.9.2    | util functional server clie
object.assign     | ES6 spec-compliant… | =ljharb…       | 2017-12-21 | 4.1.0    | Object.assign assign ES6 ex
….
```

Anda dapat melihat *node.js module* lainya yang memiliki *string underscore*.

Install Package

Terdapat dua mode untuk melakukan instalasi *node.js module*, yaitu instalasi secara lokal per *directory* atau instalasi secara global.

Instalasi secara lokal membuat *node.js module* hanya dapat digunakan pada satu *directory* saja dan instalasi secara *global* membuat anda dapat menggunakan *node.js module* di *directory* manapun.

Untuk eksekusi perintah instalasi secara *global* cukup diberi *parameter* **`-g`** seperti dibawah ini:

```
npm install underscore -g
```

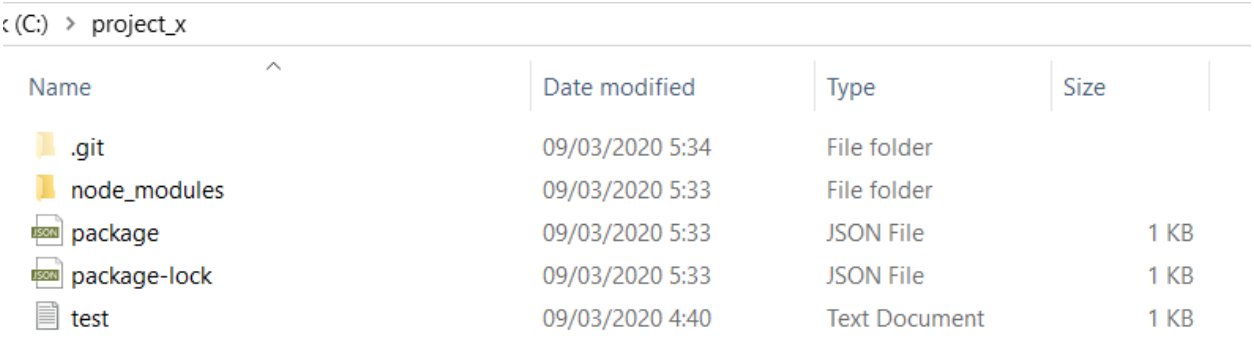
Masih di dalam *repository* **`project_x`** eksekusi perintah instalasi secara lokal di dalam *repository* tersebut.

```
npm install underscore
```

Jika berhasil maka akan muncul informasi *node.js module underscore* lengkap dengan versi terakhir yang di instalasi :

```
+ underscore@1.9.2
added 1 package from 1 contributor and audited 1 package in 1.391s
found 0 vulnerabilities
```

Kalimat ***found 0 vulnerabilities*** menyatakan bahwa *node.js module* tersebut aman dari kerentanan keamanan yang dapat ditimbulkan. Pada *directory* akan terdapat *folder* baru dengan nama *node modules* dan *package-lock.json* :



Gambar 365 underscore node_modules

Remove Package

Untuk menghapus atau mencabut *node.js module* yang telah kita tanam terdapat perintah sebagai berikut :

```
npm remove underscore
```

View Package

Untuk melihat suatu *node.js module* lebih detail lagi kita dapat mengeksekusi perintah sebagai berikut pada *command prompt* :

```
npm view underscore
```

Hasilnya adalah :

```
underscore@1.9.2 | MIT | deps: none | versions: 36
JavaScript's functional programming helper library.
https://underscorejs.org
keywords: util, functional, server, client, browser
```

```
...
```

Kita dapat mengetahui informasi terkait *package* yang ingin kita gunakan.

Publish Package

Selain menggunakan *node.js module* yang dibuat oleh orang lain, anda juga dapat membuat *node.js module* sendiri dan mempublikasikanya pada ***Node Package Registry***.

Sehingga *node.js module* yang anda buat dapat digunakan oleh orang lain. Anda akan belajar cara mempublikasikan *node.js module* yang anda buat sendiri dihalaman berikutnya.

Create Package

Masih di dalam *repository* **`project_x`**

Jika sudah buatlah sebuah *file index.js* dengan kode di bawah ini :

```
const _ = require("underscore");

function getFirstIndex(array) {
  let x = _.first(array);
  return x;
}

function getLastIndex(array) {
  let x = _.last(array);
  return x;
}

exports.getFirstIndex = getFirstIndex;
exports.getLastIndex = getLastIndex;
```

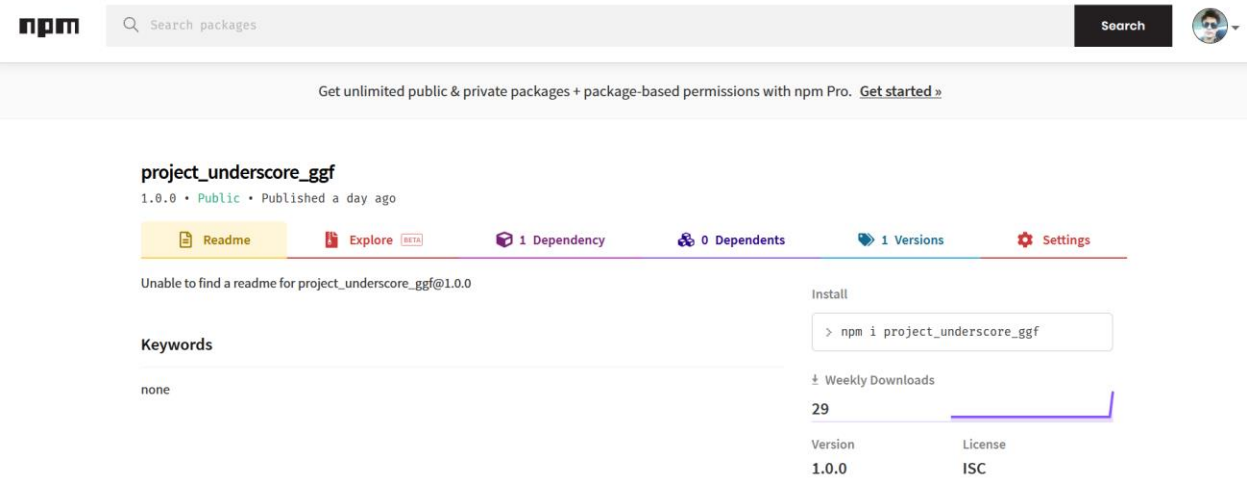
Pada kode di atas kita memanfaatkan *method first* yang dimiliki oleh *library underscore* untuk mendapatkan *index* pertama dari suatu *array* dan *method last* untuk mendapatkan *index* terakhir dari suatu *array*.

Kemudian kita membuat *function* **`getFirstIndex`** *&* **`getLastIndex`** yang akan di *export* sebagai *node.js module*.

9. *Publish Node.js Package*

Ketika sebuah *module* dipublikasikan ke *NPM Registry* di http://npmjs.org , siapapun bisa melihat dan mengaksesnya. Ini mengapa *Node Package Registry* sangat membantu dalam ekosistem *node.js* agar seluruh *developer* bisa saling berbagi dan berkontribusi.

Di bawah ini adalah contoh *module* milik penulis di **npmjs**.



Gambar 366 Publish Package

Untuk melakukan publikasi pastikan anda telah membuat akun di *Node Package Registry*. Jika sudah eksekusi perintah di bawah ini :

```
npm adduser
```

Isi *username* dan *password* sampai *login* ke *Node Package Registry* :

```
Username: gungunfebrianza
Password:
Email: (this IS public) gungunfebrianza@gmail.com
Logged in as gungunfebrianza on https://registry.npmjs.org/.
```

Gambar 367 Login Ke NPM Registry

Buka kembali *file package.json* dan tambahkan *directive* baru yaitu *repository* dan *keywords*, agar orang lain bisa mengetahui lokasi *repository* dari *project* anda di *github*. Selanjutnya kita akan mengeksekusi perintah untuk melakukan *publish* :

```
npm publish
```

Jika muncul pesan *error* seperti ini, salah satu penyebabnya adalah nama *package* sudah terdaftar sebelumnya, sehingga anda perlu mengubah *directive* ***name*** dalam *package .json* misal ***project_underscore_namaanda***.

```
C:\project_x>npm publish
npm notice
npm notice package: project_x@1.0.0
npm notice === Tarball Contents ===
npm notice 395B beginner-module.js
npm notice 285B package.json
npm notice 29B  test.txt
npm notice === Tarball Details ===
npm notice name:          project_x
npm notice version:       1.0.0
npm notice package size:  521 B
npm notice unpacked size: 709 B
npm notice shasum:        d10469a6b227a02780846c74deb1289793008a62
npm notice integrity:     sha512-uxo7Q2tdZVsti[...]HYji5f1FzAh2A==
npm notice total files:   3
npm notice
npm ERR! code E403
npm ERR! 403 403 Forbidden - PUT https://registry.npmjs.org/project_x - Package name too similar to existing
t_x' and publishing with 'npm publish --access=public' instead
npm ERR! 403 In most cases, you or one of your dependencies are requesting
npm ERR! 403 a package version that is forbidden by your security policy.

npm ERR! A complete log of this run can be found in:
npm ERR!     C:\Users\Gun Gun Febrianza\AppData\Roaming\npm-cache\_logs\2020-03-08T23_00_57_437Z-debug.log
```

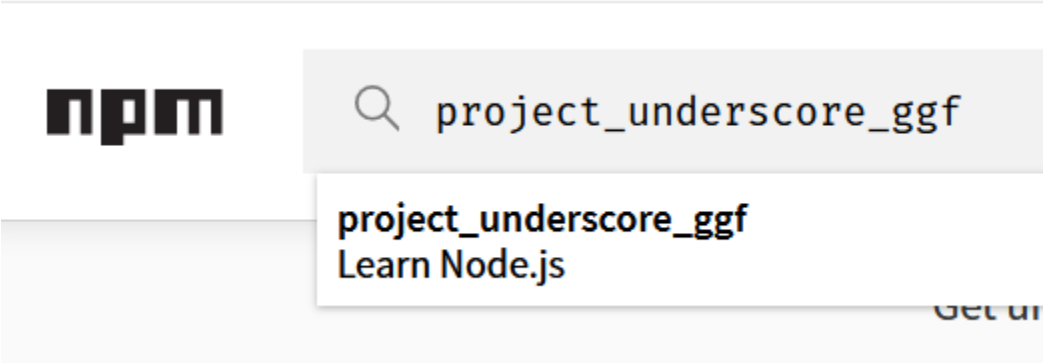
Gambar 368 Publish Module Failed

Jika berhasil maka akan muncul informasi seperti pada gambar di bawah ini :

```
C:\project_x>npm publish
npm notice
npm notice package: project_underscore_ggf@1.0.0
npm notice === Tarball Contents ===
npm notice 395B beginner-module.js
npm notice 298B package.json
npm notice 29B  test.txt
npm notice === Tarball Details ===
npm notice name:          project_underscore_ggf
npm notice version:       1.0.0
npm notice package size:  527 B
npm notice unpacked size: 722 B
npm notice shasum:        d3f691f498e5577afd40d7277b6cef174a9c70cf
npm notice integrity:     sha512-RKzcsZrJQwGVP[...]HvDTZotdz8apw==
npm notice total files:   3
npm notice
+ project_underscore_ggf@1.0.0
```

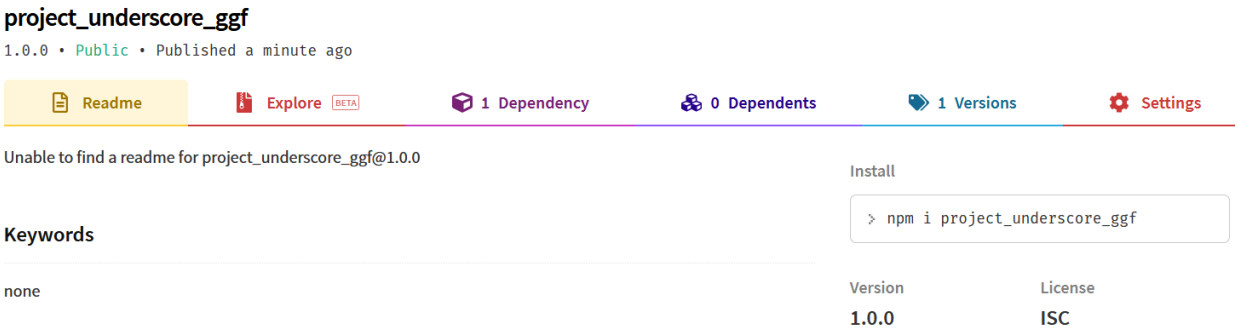
Gambar 369 Publish Module Success

Anda bisa mencarinya di kolom pencarian yang disediakan oleh **npmjs** :



Gambar 370 Search Published Package

Jika kita buka secara detail *package* tersebut maka kita akan memiliki halaman khusus dimana orang-orang dapat mengetahui informasi dari *package* yang kita buat secara lengkap. Pada *sidebar* sebelah kanan terdapat informasi untuk melakukan installasi *package* yang telah kita buat.



Gambar 371 Package Page Details

Untuk membatalkan *module* yang telah dipublikasikan anda bisa menggunakan perintah di bawah ini :

```
npm unpublish
```

10. *Node.js Application*

Sebelumnya kita telah belajar bagaimana cara membuat dan mempublikasikan *package*. Sekarang kita akan belajar bagaimana cara menggunakan *package* yang telah kita *publish* kedalam aplikasi *node.js* yang akan kita buat.

Untuk melakukanya sangat mudah, kita hanya perlu menanamkan *module* ke dalam aplikasi *node.js* yang akan kita buat. Melakukan instalasi *module* dan memuat *module* menggunakan *method* **`require()`**.

Pertama kita akan membuat *folder* bernama **`project_y`** **:**

```
mkdir project_y
```

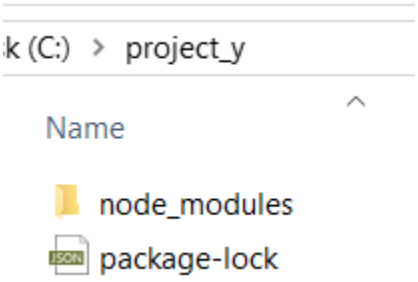
Selanjutnya masuk kedalam *directory* tersebut :

```
cd project_y
```

Selanjutnya *install module* yang telah kita *publish* :

```
npm i project_underscore_ggf
```

Pastikan setelah melakukan instalasi terdapat *folder node_modules* :



Gambar 372 Installed Modules

Setelah itu buatlah sebuah *file* bernama ***index.js*** dengan isi kode di bawah ini :

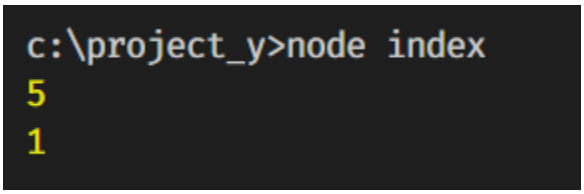
```
const arrayapp = require("project_underscore_ggf");
```

```
console.log(arrayapp.getFirstIndex([5, 4, 3, 2, 1]));
// Output : 5

console.log(arrayapp.getLastIndex([5, 4, 3, 2, 1]));
// Output : 1
```

Eksekusi kode di atas dengan perintah di bawah ini :

```
node index.js
```

Maka akan muncul *output* seperti pada gambar di bawah ini :



Gambar 373 Execute node.js application

Silahkan beri anda waktu untuk memahami kodenya baik-baik.

Subchapter 4 – Debugging Node.js

Teach a man how to debug,
and you teach them for a lifetime.

— Gun Gun Febrianza

Subchapter 4 – Objectives

- Memahami cara *debugging* dalam ***Visual Studio Code***
- Memahami cara *debugging* dengan ***built-in node.js debugger***

1. *Debug on Visual Studio Code*

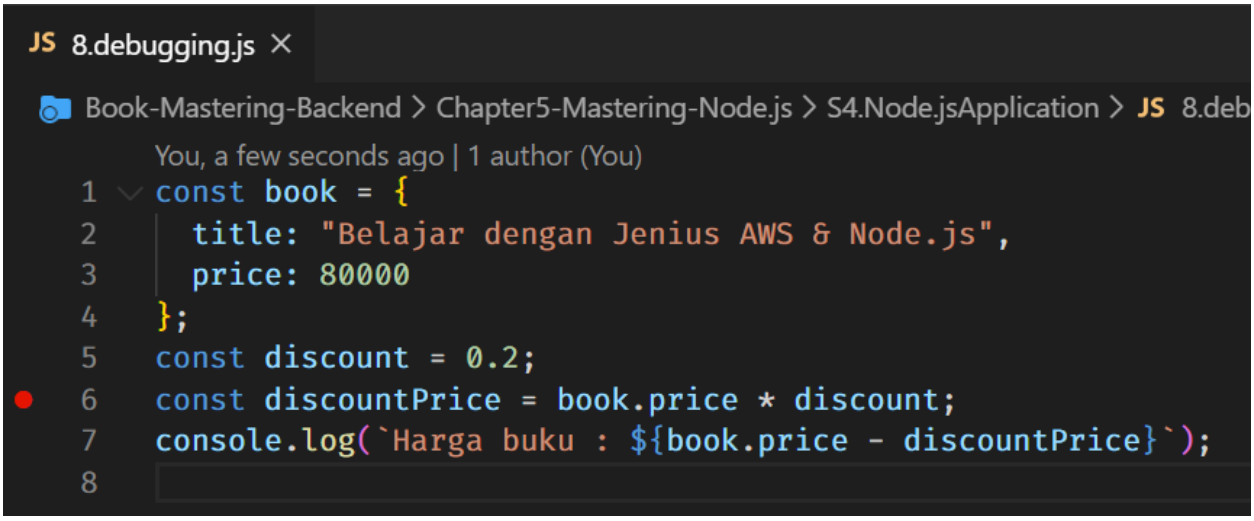
Kelebihan dari *Visual Code* sebagai *editor* untuk *node.js* adalah tersedianya ***built-in debugger***, kita akan belajar melakukan ***debugging*** menggunakan *visual studio*.

Buatlah sebuah *file* dengan nama *debugging.js* kemudian tulis kode di bawah ini :

```
const book = {
  title: "Belajar dengan Jenius AWS & Node.js",
  price: 80000
};
const discount = 0.2;
const discountPrice = book.price * discount;
console.log(`Harga buku : ${book.price - discountPrice}`);
```

**Link* sumber kode.

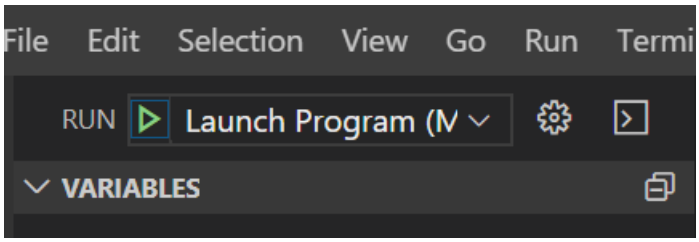
Arahkan *cursor mouse* anda pada baris kode nomor 6 sampai muncul ikon bulat berwarna merah, jika sudah klik baris kode tersebut sampai ikon bulat berwarna merah muncul seperti pada gambar di bawah ini :



Gambar 374 Add Breakpoint

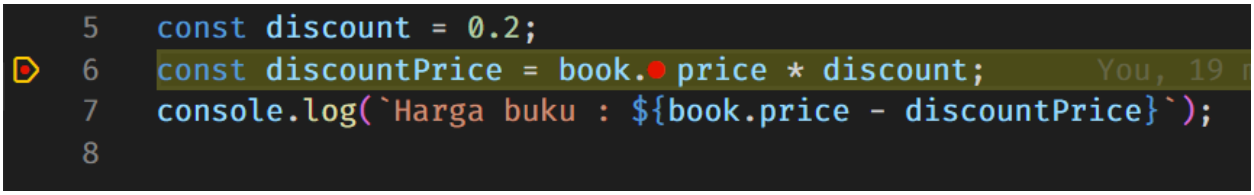
Ikon bulat berwarna merah tersebut adalah sebuah *breakpoint*, program akan berhenti pada baris tersebut dan kita dapat mengamati apa yang terjadi pada *statement code* di baris ke 6.

Untuk menggunakan *debug panel* pada *visual code* tekan **CTRL+SHIFT+D**, *panel* tersebut menampilkan seluruh informasi terkait *debugging* yang kita lakukan.



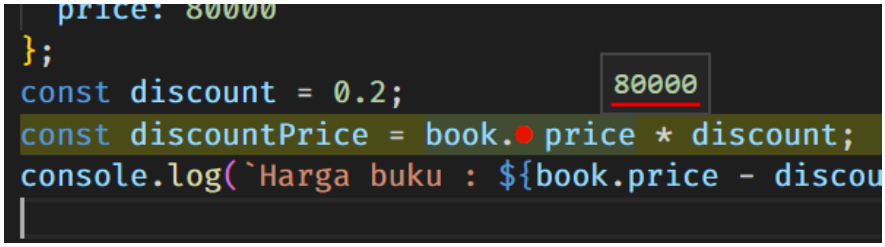
Gambar 375 Launch Program

Klik tombol **run** berwarna hijau dan pilih **launch program** pada tempat dimana aplikasi node.js kita disimpan. Anda akan melihat terjadi perubahan warna pada kode *editor* seperti pada gambar di bawah ini :



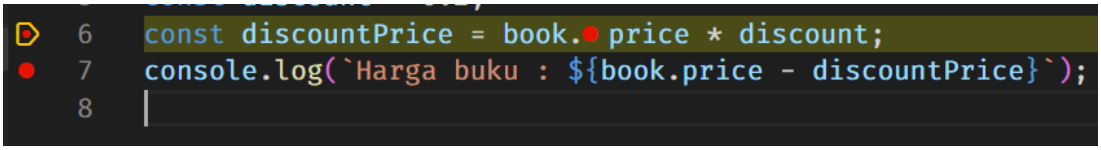
Gambar 376 Debugging Process

Arti dari baris yang diberi background warna kuning adalah kita akan segera mengeksekusi baris tersebut. Saat ini posisi program sedang dalam keadaan *pause*, jika kita arahkan *cursor* kita menuju *object* **book.price** maka anda bisa melihatnya seperti pada gambar di bawah ini :



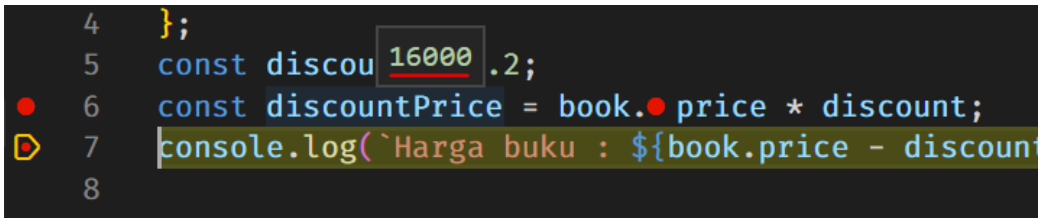
Gambar 377 Peek Value

Jika kita ingin mengamati pada baris berikutnya klik lagi baris ke 7 sampai kita berhasil membangun *breakpoint* dengan tanda berupa ikon bulat berwarna merah :



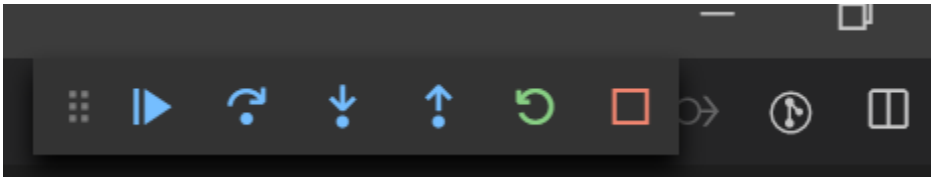
Gambar 378 Add New Breakpoint

Selanjutnya tekan tombol F5 dan jika kita arahkan kembali cursor kita pada variabel **discountPrice** maka kita bisa melihat hasil dari komputasi yang telah terjadi. Proses *debugging* inilah yang sangat membantu kita untuk melacak *logic error* atau sumber masalah yang terjadi di dalam program.



Gambar 379 Next Breakpoint

Selain itu ada dapat melihat terdapat *Debugger Panel* pada *visual studio code* :



Gambar 380 Debugger Panel

Panel ini membantu kita untuk melakukan *debugging* terdapat *control* :

1. *Continue (F5)*
2. *Step Over (F10)*
3. *Step Into (F11)*
4. *Step Out (Shift+F11)*
5. *Restart (CTRL+Shift+F5)*
6. *Stop (Shift+F5)*

Control Continue dapat kita gunakan untuk memeriksa setiap baris kode yang diberi *breakpoint*. Sementara *Control Step Over* digunakan untuk memeriksa setiap baris *statement code*.

2. *Built-in Node.js Debugger*

Sekarang kita akan melakukan ***debugging*** menggunakan ***built-in debugger*** yang dimiliki oleh *node.js*. Untuk memulainya buatlah sebuah *file* dengan nama ***debugging.js***, kemudian tulislah kode di bawah ini :

```
var person = {
  name: 'Nikolaj'
}
person.age = 25
person.name = 'Vestorp'
console.log(person);
```

**Link* sumber kode.

Untuk melakukan *debugging* pada *script* di atas, eksekusi perintah di bawah ini :

```
> node inspect debugging
```

Perintah di atas akan memproduksi *output* sebagai berikut :

```
< Debugger listening on ws://127.0.0.1:9229/c4891f76-ad83-4167-a23e-3c8b4414d64b
< For help, see: https://nodejs.org/en/docs/inspector
Break on start in debugging.js:1
> 1 (function (exports, require, module, __filename, __dirname) { var person = {
  2   name: 'Nikolaj'
  3 };
debug>
```

Gambar 381 Built-in Debugger

Saat kita mencoba menjalankan aplikasi dalam mode *debug*, posisi *pause* dimulai sebelum kita akan mengeksekusi *statement* pertama. Di tandai dengan *icon little caret* (>) seringkali disebut dengan *line break*.

Saat posisi *pause* ada pada baris pertama artinya baris tersebut belum dieksekusi sehingga kita belum memiliki variabel **book**. Kita dapat mengeksekusi perintah n yang artinya memberikan instruksi ***next***.

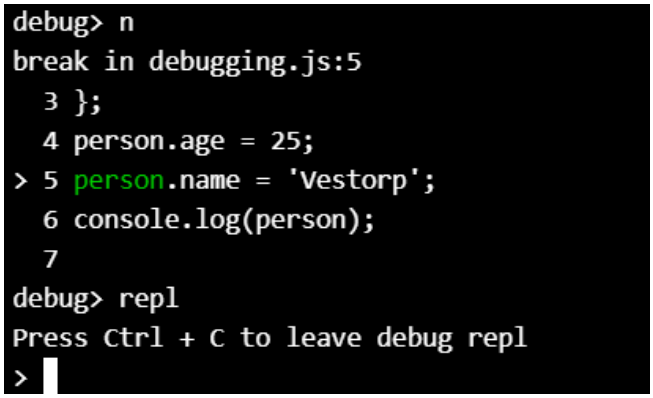
```
debug> n
```

Perintah c artinya memberikan instruksi *Continue* untuk melanjutkan sampai akhir program.

```
debug> c
```

Untuk keluar dari *debugger* tekan tombol **CTRL+C**.

Beri perintah n sampai anda menuju statement pada baris ke 5 :

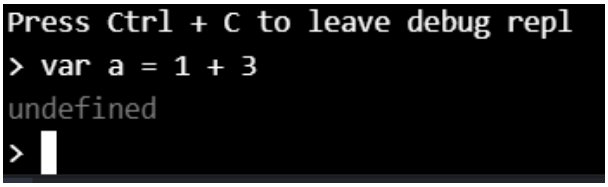


Gambar 382 REPL on Debug Mode

Ketik ***repl*** *command* agar kita bisa memasuki mode REPL :

```
debug> repl
```

Selanjutnya anda bisa bermain-main seperti mencoba mengeksekusi sebuah *expression* :



Gambar 383 Test Expression

Kita juga dapat membaca variabel **person** di dalam REPL :

```
> var a = 1 + 3
undefined
> person
{ name: 'Nikolaj', age: 25 }
>
```

Gambar 384 Read Variable

Pada kode ***debugging.js*** kita dapat menambahkan *statement* **debugger**; tujuannya adalah untuk memberikan instruksi pada ***node.js debugger*** agar berhenti pada baris tersebut :

```
person.age = 25;
debugger;
person.name = "Vestorp";
```

Setelah itu lakukan *debugging* lagi pada *script* di atas, eksekusi perintah di bawah ini :

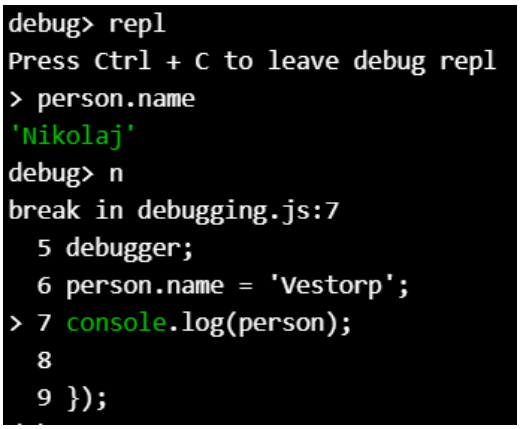
```
> node inspect debugging
```

Gunakan perintah ***continue*** agar kita bisa langsung menuju baris ke 5, tanpa harus menggunakan n yang membuat kita harus membaca per *statement*.

```
debug> c
break in debugging.js:5
  3 };
  4 person.age = 25;
> 5 debugger;
  6 person.name = 'Vestorp';
  7 console.log(person);
debug>
```

Gambar 385 debugger keyword

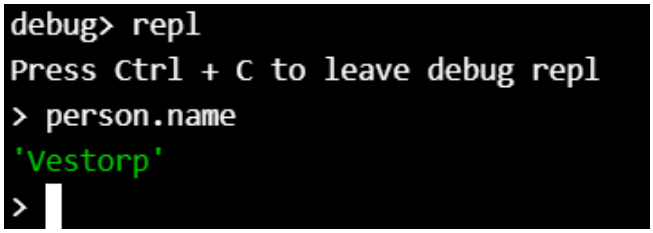
Kita bisa kembali memasuki mode REPL dan memeriksa nilai yang dimiliki oleh suatu variabel, pada gambar di bawah ini kita mencoba membaca nilai *property* **name** yang dimiliki oleh *object* **person** :



Gambar 386 Before name value changed

Untuk keluar dari mode REPL silahkan tekan secara bersamaan CTRL + C.

Setelah melewati baris kode ke 6 nilai dari *property* **name** pada *object* **person** telah berubah, untuk memeriksanya kita harus kembali ke mode REPL dan memeriksanya secara manual :



Gambar 387 After Value Changed

Lebih mudah melakukan *debugging* menggunakan *visual studio code* yah?

Subchapter 5 – *Asynchronous*

1. **Callback**

2. **Promise**

3. **Async Await**

Chapter 6

Amazon Web Service

Sebelum menggunakan layanan ***Amazon Web Services*** pastikan anda membuat akun *AWS* terlebih dahulu. Kunjungi Halaman *AWS Click here*.

Untuk tutorial bagaimana cara mendaftarkan akun di AWS, silahkan baca tutorial *Click here*.

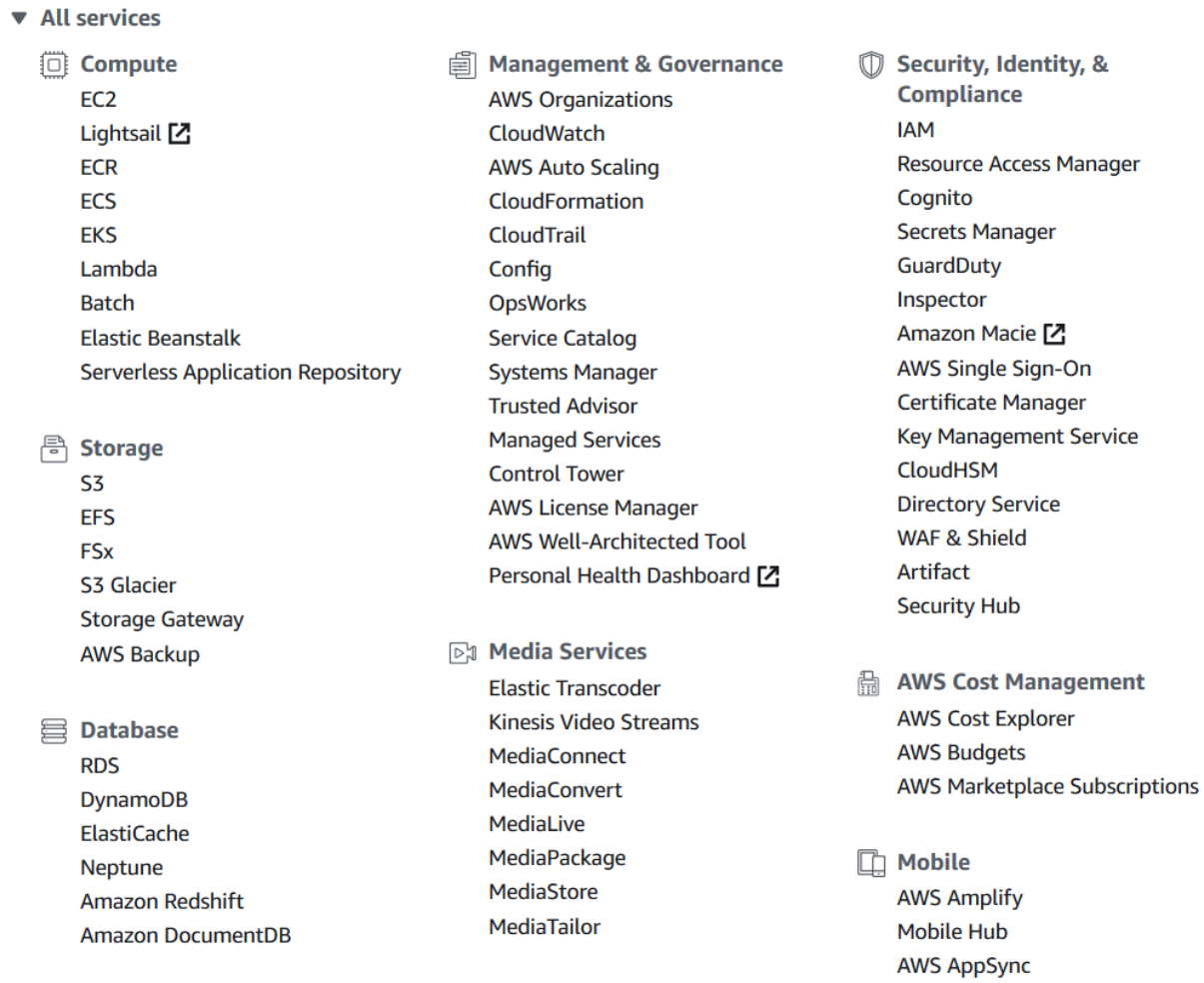
AWS menyediakan layanan yang sangat luas seperti layanan komputasi *(compute)*, penyimpanan *(storage), databases, networking, analytics, machine learning* dan *artificial intelligence (AI), Internet of Things (IoT), security* dan masih banyak lagi. Layanan yang memberikan ***non-stop solution*** untuk menggunakan *product AWS*.

Harapan penulis semoga tahun ini atau tahun depan *Amazon* bisa datang ke Indonesia agar layananya bisa menjadi lebih *reliable* dan *pricing* menjadi lebih terjangkau. Sempat beredar kabar dari Gubernur Jawa Barat bapak Ridwan Kamil bahwa perusahaan *amazon* akan berinvestasi membangun ***Data Center*** di daerah Jawa Barat.[30]

Gambar 388 AWS In Jakarta/Indonesia

Cloud adalah kumpulan teknologi yang bisa diakses melalui ***web portal***, fungsi dari *web portal* sebagai *interface* untuk mengendalikan infrastruktur yang dimiliki oleh *cloud service provier*. Saat ini penulis mengandalkan *Amazon Web Service* sehingga penulis bisa berbagi pengalaman kepada pembaca melalui *dashboard AWS* penulis.

Saat buku ini ditulis *AWS* memiliki 165 layanan yang bisa kita gunakan. Di bawah ini adalah kumpulan teknologi yang dimiliki oleh *AWS* :



Gambar 389 AWS Resources

Ketika anda memasuki *dashboard AWS* maka anda akan melihat menu tersebut. Kita akan mengkajinya secara ringkas agar anda mengetahui apa saja teknologi yang dimiliki oleh *AWS*.

Subchapter 1 – *AWS Resources*

A brand for a company is like a reputation for a person.
You earn reputation by trying to do hard things well.

— Jeff Bezos

Subchapter* 1 – *Objectives

- Mengenal ***Computing Power*** pada **AWS**.
- Mengenal Layanan ***Amazon Lightsail***
- Mengenal Layanan ***Elastic Compute Cloud (EC2)***
- Mengenal Layanan ***Elastic Container Service (ECS)***
- Mengenal ***Storage Power*** pada **AWS**.
- Mengenal Layanan ***Simple Storage Service (S3)***
- Mengenal Layanan ***Amazon Glacier***
- Menganal Layanan ***Elastic Block Store (EBS)***
- Mengenal Layanan ***Elastic File System (EFS)***

1. *Computing Power*

Terdapat layanan *computing* yang dapat kita gunakan untuk membuat *server* dengan spesifikasi seperti yang kita inginkan. Di antaranya adalah :

Amazon Lightsail

Gambar 390 Amazon Lightsail Logo

Lightsail adalah salah satu *cloud-platform* yang paling mudah digunakan, cocok untuk yang baru belajar teknologi *cloud*. Harganya terjangkau dan dapat diprediksi. Biaya dengan spesifikasi paling minimum perbulan sebesar $3.50 atau ~50 ribu rupiah. Segala hal yang diperlukan untuk membuat *Web Application* sudah tersedia seperti *Virtual Server*, *Storage*, *Database*, *DNS Management*, dan *Static IP Address*.

Untuk membuat *web application* sederhana sangat mudah, pada *lightsail* cukup dengan beberapa klik saja kita bisa menggunakan ***pre-configured development stack*** seperti *LAMP*, *Nginx*, *MEAN* dan *Node.js*. Membuat aplikasi *web* berbasis *CMS* menggunakan *wordpress, magento* dan *joomla* menjadi lebih muda.

Lightsail juga cocok untuk *development environment* atau testing aplikasi yang kita buat.

Amazon Elastic Compute Cloud (EC2)

Gambar 391 Amazon EC2 Logo

Layanan ***Amazon Elastic Compute Cloud*** adalah layanan yang memberikan fleksibilitas tinggi untuk mengatur kapabilitas *cloud* yang ingin kita miliki sehingga sering kali disebut *resizable compute capacity*. Kita dapat menyewa mesin (*instance*) *EC2* yang setara dengan ***virtual server***. Dalam hitungan menit kita bisa meningkatkan atau menurunkan kapasitas mesin yang kita miliki.

Fleksibilitas untuk memiliki kontrol penuh atas mesin yang kita miliki, seperti akses *root* untuk mengatur segala hal yang kita inginkan. Saat hendak membangun mesin kita bisa menentukan kapasitas *CPU, Memory, Storage* dan sistem operasi, tersedia berbagai distribusi *linux* dan *microsoft windows server*. Sangat cocok untuk membangun *enterprise application*. Untuk *large scale project* kita bisa membangun ratusan hingga ribuan mesin sekaligus dalam hitungan menit. Pelajari *AWS* jika anda ingin memiliki kemampuan ini.

Untuk *pricing* kita bisa membayar secara ***on-demand***, sewa server perjam sesuai dengan kemampuan komputasi yang kita inginkan.

Jika sebelumnya anda menggunakan ***Amazon Lightsail*** anda dapat dengan mudah melakukan migrasi untuk *scaling* ke *Amazon Elastic Compute Cloud* untuk mendapatkan fleksibilitas maksimum.

Amazon Elastic Container Service (ECS)

Gambar 392 Amazon ECS Logo

Amazon Elastic Container Service (Amazon ECS) adalah layanan yang memberikan *high-performance container orchestration,* layanan ini mendukung ***Docker containers*** sehingga kita bisa menjalankan *containerized applications* di *AWS*.

Layanan ini memberikan kemudahan kepada kita untuk tidak melakukan instalasi dan mengoperasikan perangkat lunak *container orchestration* sendiri. Berikut dengan manajemen dan *scaling cluster of virtual machine* sampai ke *schedule container* di dalam *cluster of virtual machine*.

Dalam hitungan detik kita bisa membangun ratusan hingga ribuan *docker container*.

Untuk *pricing* ada dua model pembayaran mengikuti *pricing* seperti EC2 atau *pricing* berdasarkan sumber *vCPU* dan *memory* yang digunakan oleh *containerized application.*

Dengan ***Container*** kita bisa melakukan ***packaging*** kode, konfigurasi, *dependencies* dan *runtime engine* yang kita butuhkan. Sehingga layanan *ECS* sangat cocok untuk membangun ***microservice***.

2. Storage Power

Saat ini *cloud storage* adalah salah satu komponen penting dalam *cloud computing*, tempat menyimpan informasi yang digunakan untuk menjalankan sebuah aplikasi. Aplikasi seperti *Big data analytics, data warehouses, Internet of Things* atau *backup* dan *archive* sangat tergantung pada tempat penyimpanan data.

Cloud storage yang *reliable, scalable,* dan aman. *AWS* memiliki layanan *cloud storage* untuk aplikasi dengan *traffic* yang tinggi dan *storage* untuk *archive* dalam jumlah besar :

Amazon Simple Storage Service (S3)

Gambar 393 Amazon S3 Logo

Amazon Simple Storage Service (Amazon S3) menyediakan layanan yang hampir tanpa batas (*virtually limitless*) sebagai tempat penyimpanan data di internet. Dengan ***Amazon S3*** kita dapat menyimpan dan meminta kembali data yang kita miliki, kapanpun dan dimanapun.

Kita dapat menyimpan dan melindungi data, menggunakannya untuk *web application, mobile application, backup & restore, archive, enterprise applications, IoT devices & big data analytics.*

Amazon S3 menyebut data (foto, dokumen, dan video) yang akan kita simpan atau *upload* dengan sebutan ***objects***. *Objects* tersebut akan disimpan didalam sebuah *buckets*. Masing-masing *object* dapat disimpan dengan maksimum 5 TB.

Objects yang kita *upload* di dalam sebuah *bucket* dapat kita atur perizinanya (*permission*). Pemberian izin akses (*grant*) atau penolakan izin akses (*deny*) untuk upload dan download juga disediakan untuk memastikan data kita tidak bisa diakses oleh pihak yang tidak berwenang (*unauthorized access*).

Versioning

Amazon S3 memiliki fitur *versioning* untuk mempertahankan dan mengembalikan kembali setiap versi *objects* yang telah dihapus atau dari *human error* dan kegagalan pada aplikasi yang kita bangun. *Amazon S3* juga menyediakan fitur *Multi-factor Authentication* untuk mencegah ***acidental deletion***.

S3 Cross-Region Replication (CRR)

Amazon S3 juga memiliki ***S3 Cross-Region Replication (CRR)*** yang dapat kita gunakan untuk mereplikasi *object* ke beberapa ***AWS Region*** lainya dibeberapa negara. Manfaatnya untuk *disaster recovery, security, reduced latency* dan *compliance*.

Amazon Glacier

S3 Glacier adalah salah satu *storage class* yang dimiliki *Amazon S3*, digunakan untuk menyimpan arsip dalam jumlah besar dan data penting yang sudah jarang diakses. Biayanya lebih murah dari layanan penyimpanan *Amazon S3* versi standar. Dalam hitungan menit hingga jam kita dapat mengembalikan seluruh data yang kita miliki.

Pada level industri layanan seperti ini sangat cocok untuk perusahaan media yang menyimpan data *image* dan *video*, institusi finansial untuk menyimpan milyaran data transaksi, dunia kesehatan seperti rumah sakit, perpustakaan untuk *digital preservation* dan sains untuk *scientific data storage* seperti proyek astronomi hingga *human sequence genome project* yang memerlukan penyimpanan data yang besar.

Amazon Elastic Block Store (EBS)

Amazon Elastic Block Store (EBS) adalah layanan yang bisa kita gunakan untuk membuat *persistent storage* untuk *Amazon Elastic Compute Cloud (EC2). Amazon EBS* cocok untuk aplikasi yang memerlukan *low-latency storage*.

Dikatakan ***persistent*** karena data akan direplikasi ke ***Availability Zone*** lainya yang dimiliki oleh *AWS* untuk melindungi data anda dari kegagalan komponen pada aplikasi atau mesin yang kita bangun.

Amazon EBS dapat digunakan untuk *relational databases (MySQL* atau *Microsoft SQL Server)* dan *NoSQL databases (MongoDB* atau *Cassandra)*, *data warehousing* seperti *teradata, stream and log processing* seperti *kafka*, *Big Data processing* menggunakan Hadoop, atau *backup and recovery*.

Amazon Elastic File System (EFS)

Amazon EFS menyediakan *elastic file system* untuk *linux* yang dapat digunakan oleh satu atau lebih mesin ***Amazon EC2***. Skalabilitas sampai ukuran *petabytes* dapat dibangun dengan Amazon EFS dan dapat diakses oleh ribuan mesin *Amazon EC2* secara *parallel*. Kita dapat membuat *EFS file system* dengan cara memberikan konfigurasi mesin *Elastic Compute Cloud* untuk memasang (*mount*) ***Amazon EFS***.

Subchapter 2 – AWS CLI V1 & V2

If you double the number of experiments you do per year
you're going to double your inventiveness.

— Jeff Bezos

Subchapter 2 – Objectives

- Mengenal *AWS Command-line Interface (CLI)*
- Belajar *Install AWS CLI V2* di *OS Linux*
- Belajar *Install AWS CLI V2* di *OS MacOS*
- Belajar *Install AWS CLI V2* di *OS Windows*
- Belajar *Install AWS CLI V1*

AWS Command Line Interface (CLI) adalah sebuah *tool* untuk memanajemen seluruh layanan *AWS* yang ingin kita gunakan. Melalui *command line* kita bisa mengendalikan berbagai layanan *AWS* sekaligus baik secara manual atau ***automation*** melalui ***script***.

1. *Command Line Interface (CLI)*

Linux Shell

Pada sistem operasi ***linux*** atau ***macOS*** kita dapat berinteraksi dengan ***AWS CLI*** menggunakan program seberti ***bash***, ***zsh*** dan ***tesh*** untuk mengeksekusi suatu perintah.

Windows Command Line

Pada sistem operasi ***windows*** kita dapat berinteraksi dengan ***AWS CLI*** menggunakan program ***windows command-prompt (CMD)*** atau melalui ***powershell***.

Remote

Mengeksekusi perintah pada mesin ***Amazon Elastic Compute Cloud (Amazon EC2)*** secara ***remote*** melalui terminal yang disediakan dalam program ***PuTTY*** dan ***SSH*** atau melalui ***AWS System Manager***.

2. AWS CLI V2

Saat ini ***AWS CLI*** sedang dalam tahap upgrade dengan menyediakan ***AWS CLI*** Versi ke 2, versi pertama tidak akan dikembangkan lagi namun masih tetap dapat digunakan. Saat ini versi ke 2 disarankan tidak digunakan untuk ***production***, ***AWS CLI*** Versi ke 2 disediakan khusus untuk uji coba.

Pada ***AWS CLI*** Versi ke 2, program sudah tidak lagi menggunakan *dependencies* dari *software package* lainnya. Sebelumnya pada versi pertama terdapat *dependency* yaitu keharusan untuk instalasi *python* terlebih dahulu.

Pada ***AWS CLI*** Versi ke 2 sudah disediakan dukungan untuk ***Linux Distribution*** seperti ***CentOS, Fedora, Ubuntu, Amazon Linux 1,*** dan ***Amazon Linux 2***. Untuk **MacOS**, ***AWS CLI*** Versi ke 2 sudah disediakan dukungan untuk ***High Sierra (10.13), Mojave (10.14),*** dan ***Catalina (10.15)***.

Install AWS CLI V2 on Linux

Eksekusi perintah di bawah ini :

```
$ curl "https://d1vvhvl2y92vvt.cloudfront.net/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
```

Kemudian :

```
$ sudo ./aws/install
```

Install AWS CLI V2 on MacOS

Eksekusi perintah di bawah ini :

```
$ curl "https://d1vvhvl2y92vvt.cloudfront.net/awscli-exe-macos.zip" -o "awscliv2.zip"
unzip awscliv2.zip
```

Kemudian :

```
$ sudo ./aws/install
```

Install AWS CLI V2 on Windows

Pada sistem Operasi ***Windows***, ***download MSI Installer*** untuk ***AWS CLI V2*** disini :

https://d1vvhvl2y92vvt.cloudfront.net/AWSCLIV2.msi

atau anda dapat mengunjungi halaman di bawah ini :

https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2-windows.html

Setelah selesai download, lakukan instalasi kemudian eksekusi perintah di bawah ini :

```
C:\> aws2 --version

aws-cli/2.0.0dev3 Python/3.7.5 Windows/10 botocore/2.0.0dev2
```

3. *AWS CLI V1*

Sebelum melakukan instalasi ***AWS CLI***, pastikan sistem operasi anda sudah melakukan installasi bahasa pemrograman *python* versi 3.7 ke atas, kita akan melakukan instalasi ***AWS CLI*** melalui *python package manager* yang bernama *pip*.

Install AWS CLI

Untuk memulai instalasi eksekusi perintah di bawah ini :

```
pip install awscli
```

Upgrade AWS CLI

Jika sebelumnya anda sudah memiliki *AWS CLI*, direkomendasikan untuk melakukan upgrade ke versi yang terakhir untuk alasan keamanan. Untuk *upgrade AWS CLI*, eksekusi perintah di bawah ini :

```
pip install --upgrade awscli
```

Verify AWS CLI

Lakkan verifikasi untuk memastikan *AWS CLI* sudah terpasang dalam sistem operasi anda :

```
aws --version

aws-cli/1.16.309 Python/3.7.4 Windows/10 botocore/1.13.45
```

Eksekusi perintah di atas untuk mendapatkan informasi versi *AWS CLI* yang kita miliki.

Saat menggunakan *AWS CLI*, kita memerlukan *AWS Credentials* untuk melakukan ***authentication*** pada layanan ***AWS***.

Terdapat beberapa cara diantaranya adalah :

1. ***Environment Credentials*** berupa :

AWS_ACCESS_KEY_ID dan **AWS_SECRET_KEY**

2. ***The Shared Credentials File***
3. ***IAM Roles***, jika anda menggunakan *AWS CLI* di sebuah mesin ***EC2*** maka kita tidak perlu lagi mengatur *credential files* di *production*.

Subchapter 3 – AWS IAM

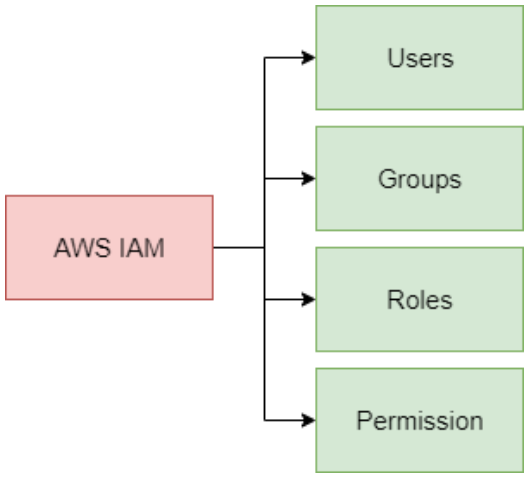
If you don't understand the details of your business you are going to fail.

— Jeff Bezos

Subchapter 3 – Objectives

- Mengenal *AWS IAM*
- Belajar Membuat ***IAM User***
- Belajar Mengkonfigurasi *User Credential*
- Belajar Membuat ***IAM Role***

AWS IAM (*Identity and Access Management*) adalah layanan untuk mengamankan akses pada *AWS Resources*. *IAM* juga mendukung konsep standar keamanan seperti *users, groups, roles* dan *permissions*.



Gambar 394 Standard Security Conceptual

Users adalah seorang pengguna yang ingin mengelola seluruh layanan *AWS, users* dapat ditambahkan ke dalam sebuah *group* yang telah kita klasifikasi, setiap *users* dan *groups* dapat diberikan *permissions* dan *roles* digunakan oleh suatu *AWS Services* untuk akses

AWS Services lainnya. Dengan *IAM* kita dapat melakukan kontrol siapa saja yang melakukan *authentication* (*signed in*) dan memiliki izin *authorization* pada *AWS Resources*.

Saat pertama kali kita membuat *AWS Account*, kita menggunakan sebuah *root account* yang memiliki akses pada seluruh layanan *AWS Resources*.

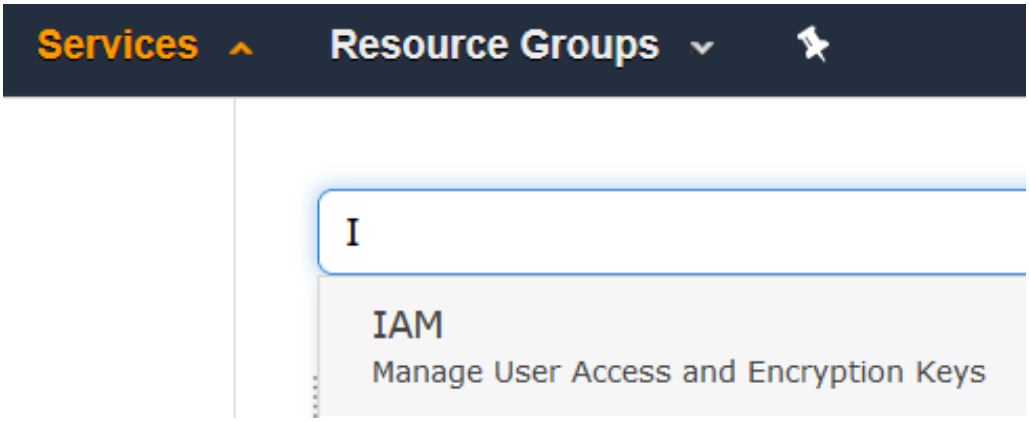
Untuk memulai aktivitas sangat tidak disarankan menggunakan ***AWS Root Account***, karena akun tersebut memiliki kapabilitas untuk membuat dan menghapus ***IAM users,*** mengubah *billing*, menghapus *account* dan segala aksi dalam akun *AWS* anda.

1. *Create IAM User*

Jadi sebaiknya kita membuat akun *IAM user* yang baru, dengan batasan akses hanya *pada AWS Lambda* saja. Untuk memulai membuat user baru, silahkan masuk kembali ke halaman ***AWS Management Console*** :

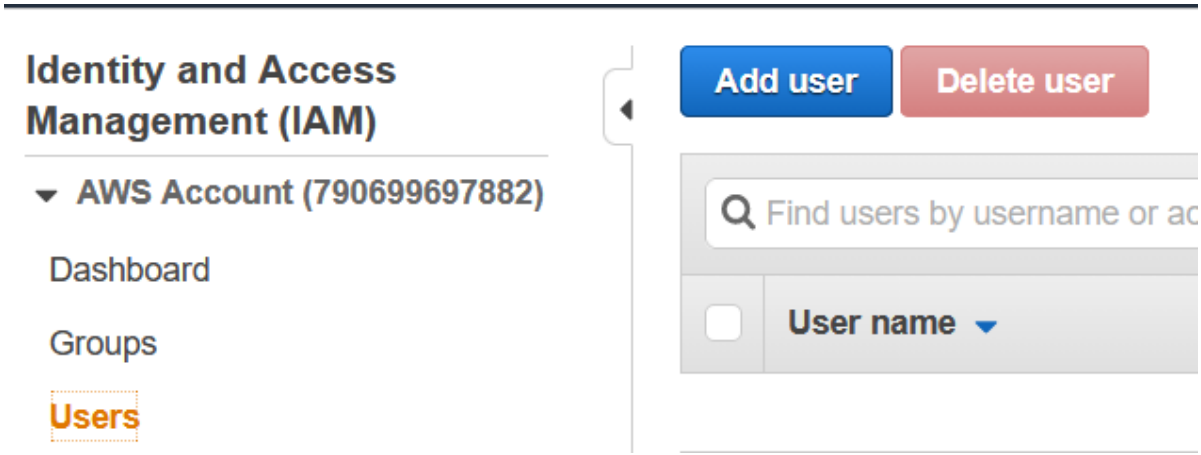
https://aws.amazon.com/console/

Kemudian cari ***IAM*** dalam kolom ***services*** :



Gambar 395 Search IAM Service

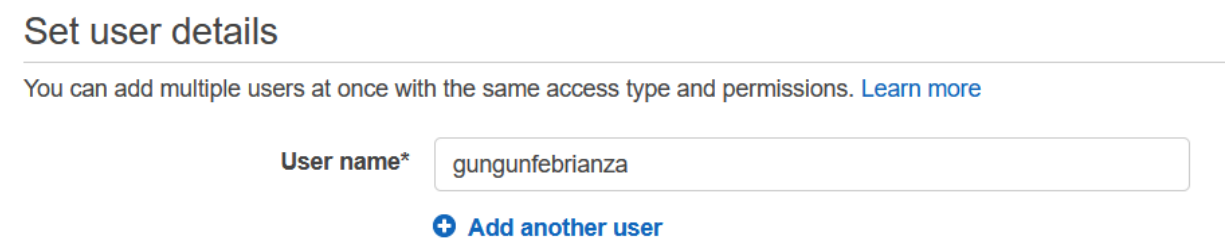
Pilih menu ***Users*** dan klik tombol ***Add User*** :



Gambar 396 Add IAM User

Set User Details

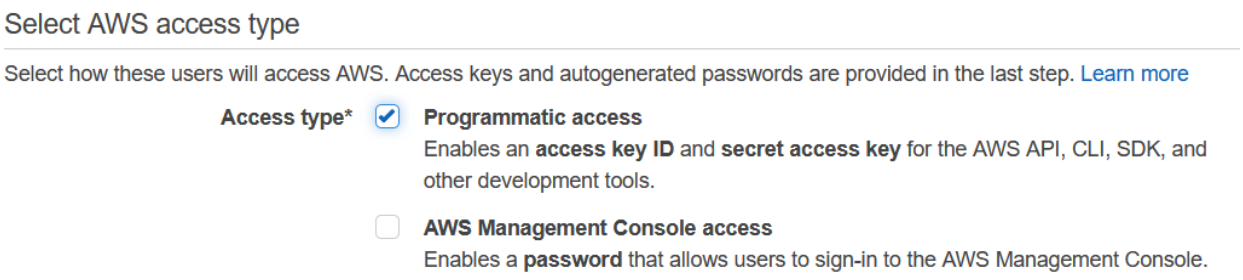
Pada kolom ***Set user details*** masukan nama ***username*** yang anda ingin gunakan :



Gambar 397 Set User Details

AWS Access Type

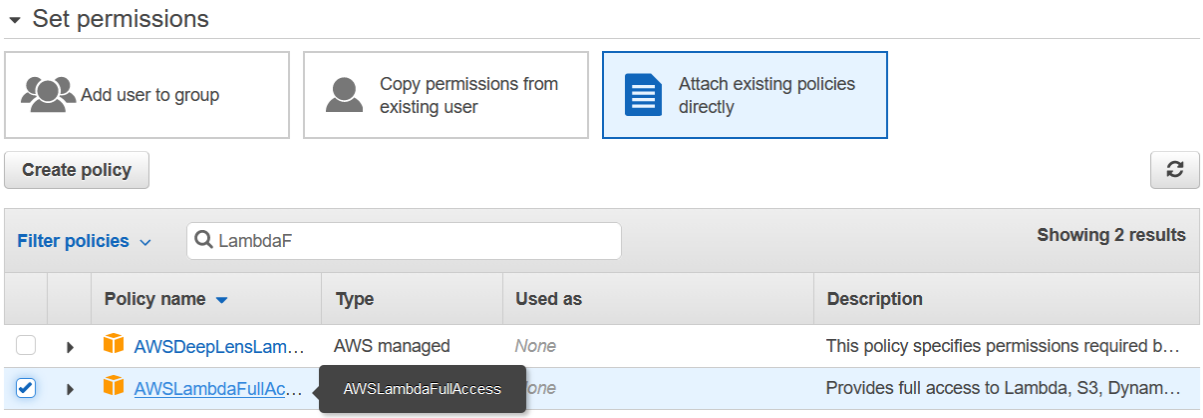
Pada kolom ***Select AWS access type***, *checklist* ***Programmatic access*** seperti gambar di bawah ini :



Gambar 398 Setup AWS Access Type

Set Permission

Pada Kolom ***Set Permissions***, pilih menu ***Attach existing policies directly*** kemudian pada kolom ***Filter policies*** cari **AWSLambdaFullAccess** :

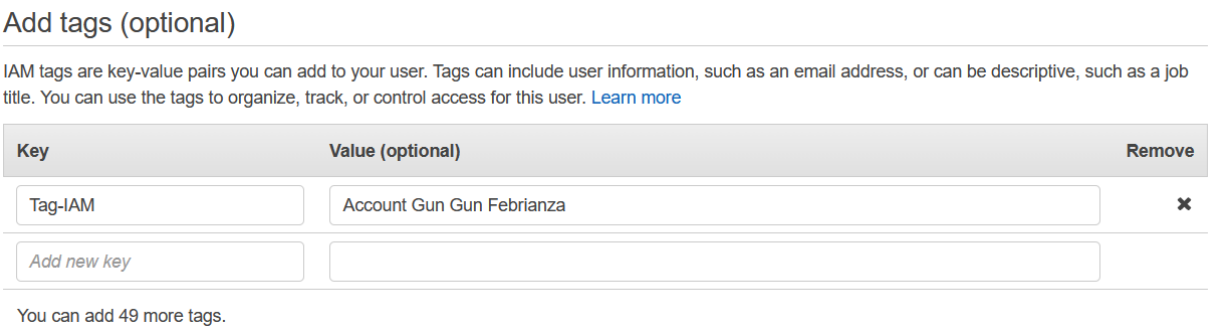


Gambar 399 Attach Existing Policies

Kita menggunakan ***policies*** **AWSLambdaFullAccess** agar kita bisa berinteraksi dengan layanan ***AWS Lambda*** di ***chapter*** berikutnya. ***User gungunfebrianza*** akan memiliki izin untuk akses layanan ***AWS Lambda***.

Tags

Pada menu *tags* anda bisa melewatkanya atau mengisinya sama seperti penulis :

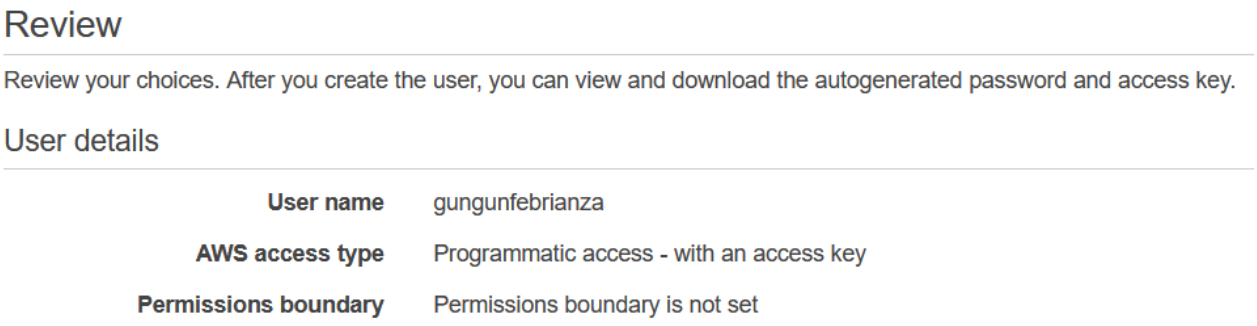


Gambar 400 Create Tags

Setelah kita isi klik tombol ***Next: Review*** di pojok kanan bawah.

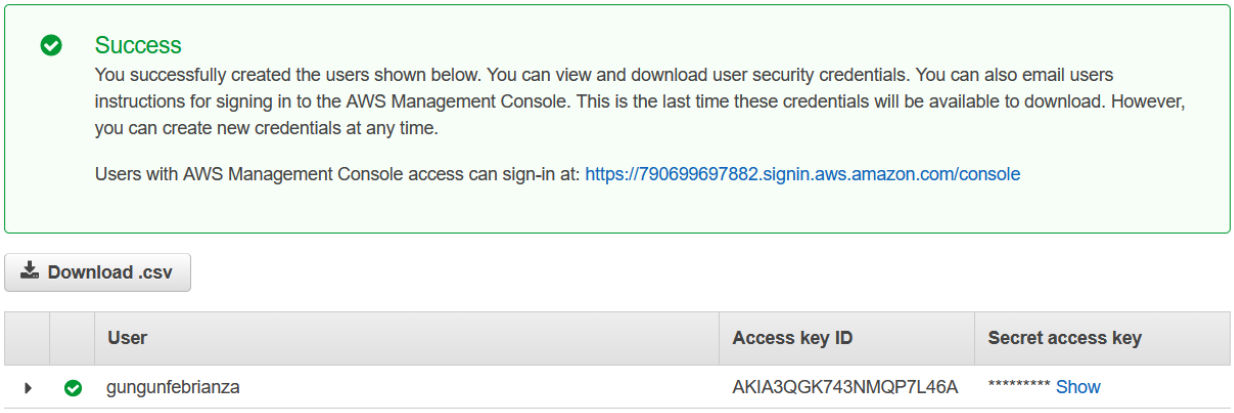
IAM User Credential

Kita memasuki halaman ***Review*** untuk memastikan kembali data dan konfigurasi kebijakan (*policy*) yang telah kita terapkan pada akun yang dibuat.



Gambar 401 Review User Credential

Setelah kita melakukan ***review*** selanjutnya di pojok kanan bawah klik tombol ***Create User***.



Gambar 402 IAM User Created

Pastikan kita mencatat dan menyimpan ditempat yang aman dan lakukan manajemen *backup* untuk mengantisipasi lupa atau kehilangan. Hal ini dikarenakan kita tidak akan lagi mendapatkan kedua data di atas.

2. *AWS Configuration*

Setelah kita membuat ***IAM user***, *Access Key* dan *Secret* digunakan untuk konfigurasi ***AWS CLI*** dalam mesin komputer kita. Eksekusi perintah **`aws configure`** kemudian masukan *access key* dan *secret key* yang telah kita dapatkan sebelumnya.

```
$ aws configure
AWS Access Key ID [none]: AKIA3QGK743NMQP7L46A
AWS Secret Access Key[none]: HvytDJpkw**********************
Default region name [us-east-1]:
Default output format [none]:
```

Pada sistem operasi *Linux*, *AWS CLI* akan menyimpan konfigurasi *credentials* di :

`~/.aws/credentials`

Sementara pada sistem operasi *windows* :

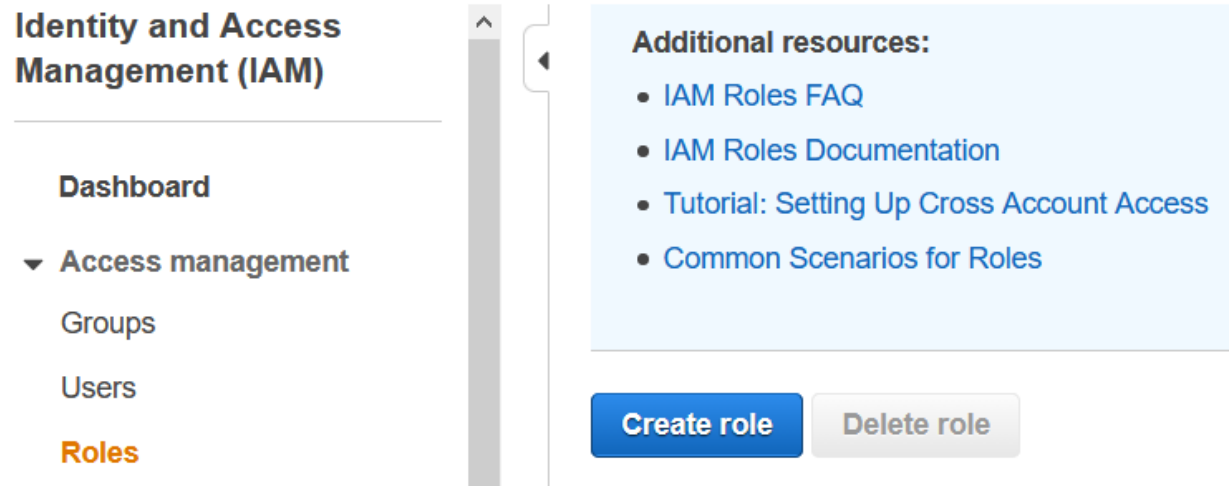
`%UserProfile%\.aws`

3. *Create IAM Role*

Kita membuat *IAM Roles* agar bisa menyediakan izin dan jalur yang aman untuk setiap entitas yang kita percaya. Entitas yang dimaksud adalah :

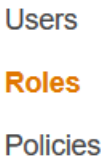
1. ***IAM User*** dalam akun *AWS* yang lain.
2. Sebuah ***application*** yang berjalan di mesin EC2 kemudian membutuhkan akses pada ***AWS Resources***.
3. Sebuah ***AWS Service*** yang membutuhkan akses pada *AWS resources* lainnya dalam akun yang kita miliki agar bisa menyediakan layanan tertentu.

Untuk membuat *IAM Roles* kita harus kembali lagi ke halaman tempat layanan ***AWS Identity Access Management (IAM)***. Perhatikan gambar di bawah ini pada ***dashboard*** pilih menu ***Roles***, kemudian klik tombol ***Create Role*** :



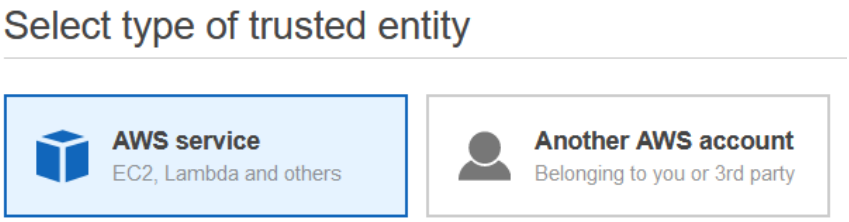
Gambar 403 IAM Dashboard

Masuk ke dalam *IAM Console*, *click here*. Kita harus membuat *Role* terlebih dahulu agar bisa menggunakan *Amazon Lambda*. Untuk membuatnya pada menu sebelah kiri dalam *IAM Console* pilih *Roles* :



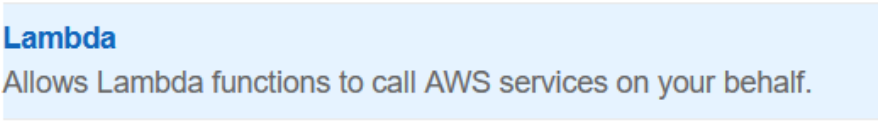
Gambar 404 Create Roles

Selanjutnya klik tombol **Create role**. Pada kolom **Select type of trusted entity** pilih ***AWS service*** :



Gambar 405 Trusted Entity

Lalu pada kolom **Choose a use case** pilih *lambda* seperti pada gambar di bawah ini :



Gambar 406 Role Use Case

Kemudian klik tombol **Next: Permissions**.

Add Policy to Role

Sebuah ***lambda function*** memiliki sebuah ***policy*** yang disebut dengan ***execution role***. *Role* ini akan diberikan setiap kali kita membuat sebuah ***lambda function***.

Execution role memberikan izin kepada ***lambda function*** agar dapat mengakses berbagai ***AWS Services & Resources***.

Di level yang paling minimum ***lambda function*** membutuhkan akses menuju layanan ***Amazon CloudWatch Log*** untuk melakukan ***log streaming***.

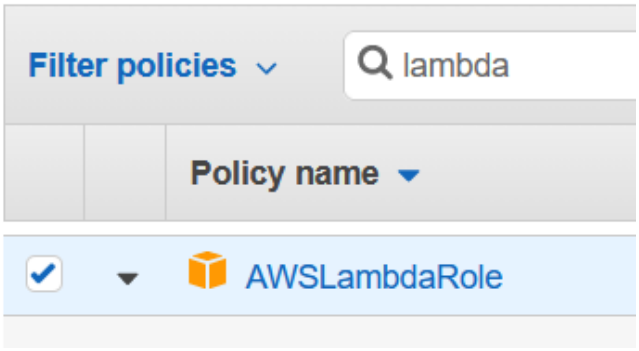
Jika kita ingin menggunakan ***AWS X-Ray*** untuk melakukan ***tracing*** pada ***lambda function*** yang kita eksekusi maka kita perlu memberikan izin (***grant permission***) agar dapat menggunakanya di dalam ***execution role***.

Jika kita ingin menggunakan ***lambda function*** agar dapat mengakses suatu ***service*** melalui ***AWS SDK***, maka kita perlu memberikan izin (***grant permission***) agar dapat menggunakanya di dalam ***execution role***.

AWS Lambda Role

Kita membutuhkan *policy* **AWSLambdaRole** agar bisa mengeksekusi ***lambda function***.

Pada menu **Filter policies** ketik **lambda**, lalu anda akan melihat **Policy name** bernama **AWSLambdaRole** muncul. Ceklis ***policy*** tersebut.



Gambar 407 Setup Policy

Ketika kita memilih menggunakan **Policy name** **AWSLambdaRole** maka kita menggunakan konfigurasi *policy statement* sebagai berikut:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
```

```
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

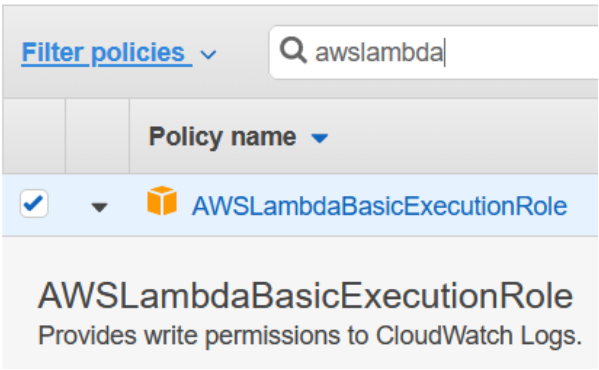
Arti dari *Policy Statement* di atas adalah kita menggunakan :

1. ***Version* element** tahun 2012 yang menentukan aturan *syntax* untuk memproses *policy statement*.
2. ***Statement* element** adalah komponen utama dalam *policy statement*, elemen ini dapat memiliki satu *statement* tunggal atau sekumpulan individual *statement* dalam sebuah *array*. Setiap individual *statement* harus dibungkus didalam (). Pada *policy statement* di atas terdapat satu *statement* saja dengan tanda warna hijau.
3. ***Effect* element** bersifat *required* digunakan untuk mengizinkan atau menolak suatu akses terhadap suatu *resources* secara eksplisit.
4. ***Action* element** menjelaskan aksi apa saja yang diizinkan atau ditolak. Setiap layanan *AWS* memiliki sekumpulan aksi yang bisa diizinkan untuk dieksekusi atau dibatasi. Pada *policy statement* di atas kita mengizinkan *AWS lambda* untuk bisa melakukan pemanggilan (*invoke*) fungsi yang ada di dalam *AWS Lambda*. Baca lebih detail di sini.
5. ***Resource* element** menjelaskan keseluruhan *statement* yang dibuat dalam *policy statement* dapat digunakan pada *resource* mana saja. Dapat digunakan untuk sebuah user tertentu atau layanan AWS tertentu. Simbol * artinya kita dapat menggunakannya di seluruh *resources*.

AWS Lambda Basic Execution Role

Kita akan menerapkan *policy* bernama **AWSLambdaBasicExecutionRole** agar bisa melakukan *upload* data menuju *AWS CloudWatch* untuk analisa logs dari lambda function yang kita buat.

Pada menu **Filter policies** ketik **lambda**, lalu anda akan melihat **Policy name** bernama **AWSLambdaBasicExecutionRole** muncul. Ceklis ***policy*** tersebut.



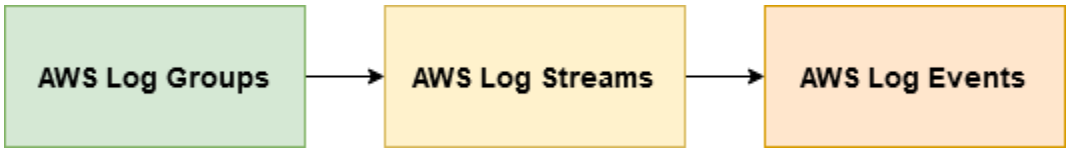
Gambar 408 Setup AWSLambdaBasicExecutionRole Policy

Ketika kita memilih menggunakan **Policy name AWSLambdaBasicExecutionRole** maka kita menggunakan konfigurasi ***policy statement*** sebagai berikut:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
         ],
         "Resource": "*"
      }
   ]
}
```

Konfigurasi ***policy statement*** di atas menyediakan izin agar kita dapat menulis ***logs*** pada layanan ***Amazon CloudWatch***.

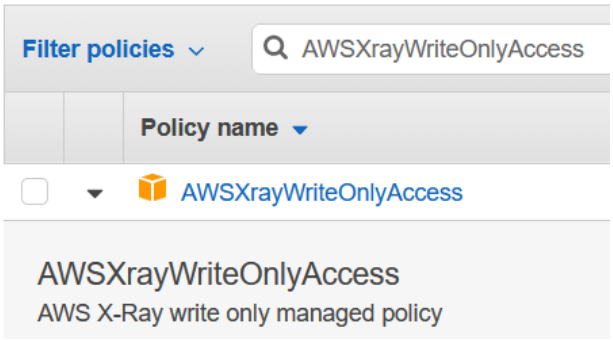
1. Aksi CreateLogGroup digunakan untuk membuat ***log groups*** maksimum sampai 20 ribu *logs per account*.
2. Aksi CreateLogStream digunakan untuk membuat ***log stream*** pada ***log groups*** tertentu dan
3. Aksi PutLogEvents digunakan untuk sekumpulan ***log events*** pada ***log streams*** tertentu.
4. Ketiga aksi tersebut membentuk hirarki ***logs*** :



Gambar 409 Hierarchy Logs

AWS Xray Write Only Access

Pada menu **Filter policies** ketik **lambda**, lalu anda akan melihat **Policy name** bernama **AWSXrayWriteOnlyAccess** muncul. Ceklis ***policy*** tersebut.



Gambar 410 AWS X-Ray Policy

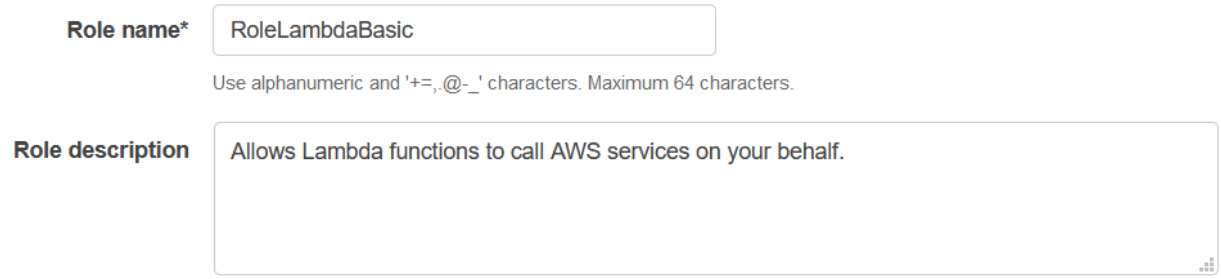
Ketika kita memilih menggunakan **Policy name** **AWSXrayWriteOnlyAccess** maka kita menggunakan konfigurasi *policy statement* sebagai berikut:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "xray:PutTraceSegments",
            "xray:PutTelemetryRecords",
            "xray:GetSamplingRules",
            "xray:GetSamplingTargets",
            "xray:GetSamplingStatisticSummaries"
         ],
         "Resource": [
            "*"
         ]
      }
   ]
}
```

Konfigurasi *policy statement* di atas menyediakan izin agar kita dapat mengirimkan *trade* data menuju *AWS X-ray*.

Tag & Review

Di pojok kanan bawah klik tombol **Next:Tags**, isi *tag* sesuai dengan keinginan anda. Lalu klik lagi tombol **Next: Review** untuk melanjutkan pengaturan.



Gambar 411 Role Name & Description

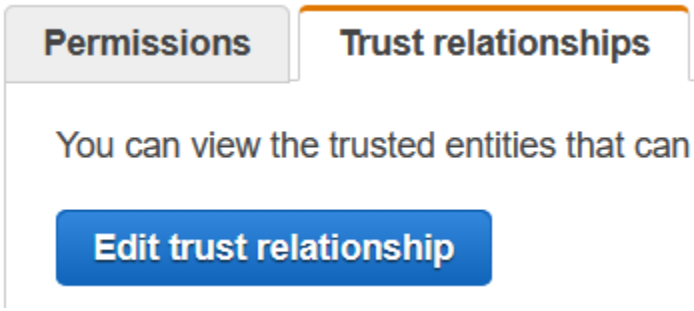
Isi **Role name*** dan **description** sesuai dengan gambar di atas, kemudian klik tombol **Create role** di pojok kanan bawah.

Jika berhasil kembali lagi *dashboard*, klik menu **Roles** perhatikan dikolom *Role* anda akan menemukan *Role* baru bernama **RoleLambdaBasic** seperti pada gambar di bawah ini :

Gambar 412 New Role for Lambda

Trust Relationships

Klik *Role* tersebut, kemudian pilih *tab* **Trust relationship** dan klik tombol **Edit trust relationship** seperti pada gambar di bawah ini :



Gambar 413 Trust relationship Configuration

Pada **Policy Document** tambahkan **apigateway.amazonaws.com** seperti pada gambar di bawah ini, perhatikan **attribute Service** :

Policy Document

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "lambda.amazonaws.com",
          "apigateway.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Gambar 414 Update Trust relationship

Sekarang kita sudah bisa menggunakan *lambda* dan *API Gateway* dengan aman, selanjutnya klik tombol **Update Trust Policy**.

Subchapter 4 – AWS Lambda

The best customer service is if the customer doesn't need to call you, doesn't need to talk to you, it just works.

— Jeff Bezos

Subchapter 4 – Objectives

- Mengenal ***AWS Lambda***
- Belajar Membuat ***Lambda Function***

Apa sih *Amazon Lambda*?

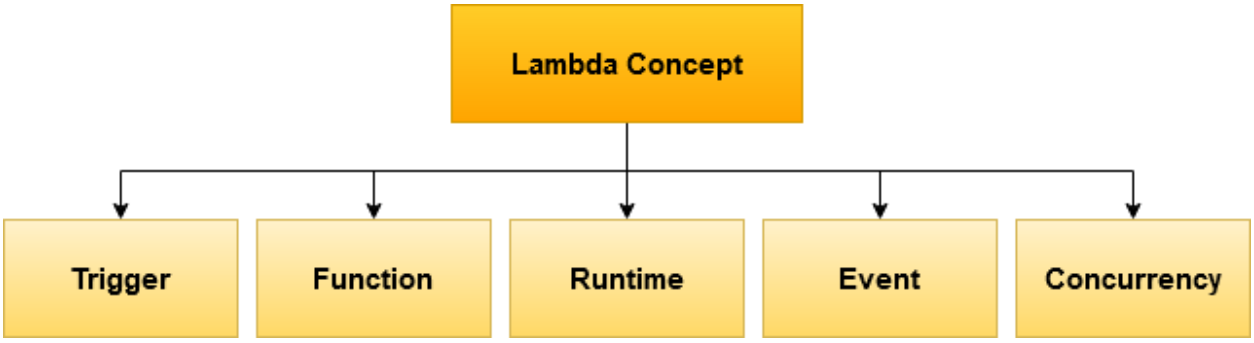
Sebuah *Function* di eksekusi di dalam sebuah ***containers***. ***Containers*** adalah sebuah metode *server virtualization* dimana *kernel* dari suatu sistem operasi mengimplementasikan sekumpulan tempat yang terisolasi (*multiple isolated environments*). *Server* fisik milik ***AWS*** mengelola sekumpulan *containers* tempat fungsi-fungsi yang kita buat dieksekusi.

Sesuai dengan jargonya ***execute the code without thinking the server***, artinya kita bisa fokus pada pengembangan kode tanpa memikirkan pengelolaan *server* inilah yang disebut dengan *serverless*.

1. *Lambda Concept*

Saat membuat sebuah ***function*** dengan ***AWS Lambda*** kita harus membuat nama untuk fungsi, menulis kode untuk fungsi dan melakukan konfigurasi pada ***execution environment*** tempat fungsi akan dieksekusi. Ada 3 Hal yang harus dikonfigurasi :

1. Maksimum ***Memory Size*** yang akan digunakan oleh setiap ***function***.
2. Pengaturan ***timeout*** setelah ***function*** selesai dieksekusi atau belum selesai dieksekusi.
3. Sebuah ***Role*** dalam *Identity and Access Management (IAM)* yang menjelaskan apa yang bisa dilakukan oleh ***function*** dan pada ***AWS resources*** mana saja ***function*** dapat digunakan.



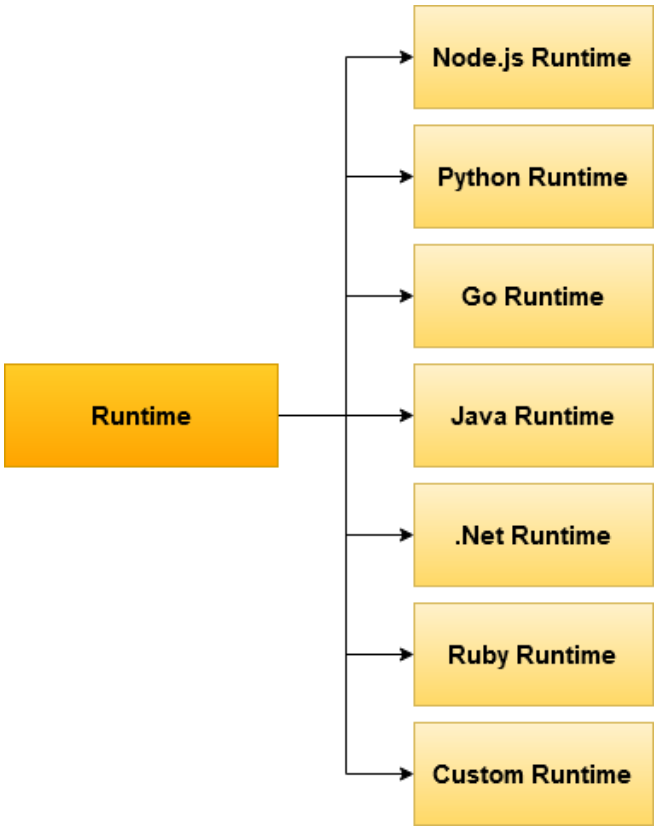
Gambar 415 Lambda Concept

Handler

Handler adalah sebuah ***method*** dalam fungsi ***lambda*** yang anda buat untuk memproses suatu ***event***. Ketika kita memanggil (***invoke***) suatu ***function*** sebuah ***runtime*** akan berjalan untuk mengeksekusi ***method*** yang menjadi ***entrypoint***.

Gambar 416 Runtime for Lambda Function

Runtime



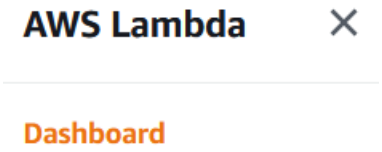
Gambar 417 Runtime Support

AWS Lambda adalah pusat untuk mengendalikan beberapa layanan agar bisa menciptakan ***serverless application***.

2. *Lambda Function*

Create Lambda Function

Setelah selesai mempersiapkan ***Role*** sekarang kita akan menuju ***Lambda Console***.

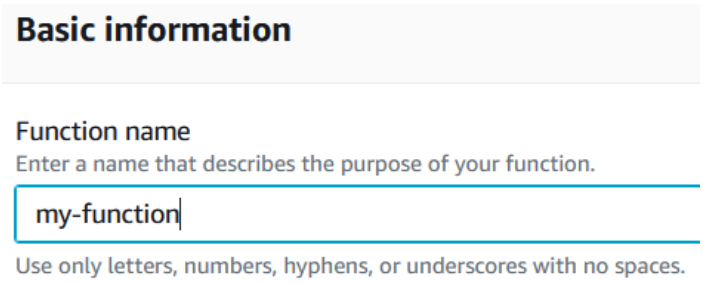


Gambar 418 Dashboard Lambda

Klik tombol **Create function** :

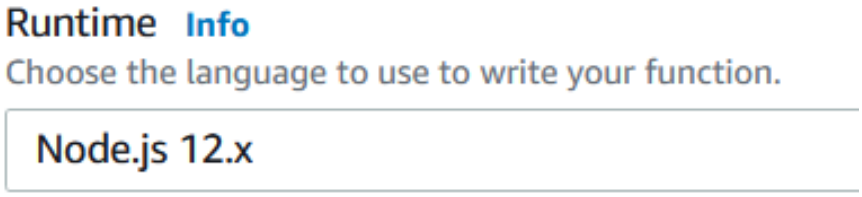
Gambar 419 Create Lambda Function

Pada kolom **Function** name isi **Calc** :



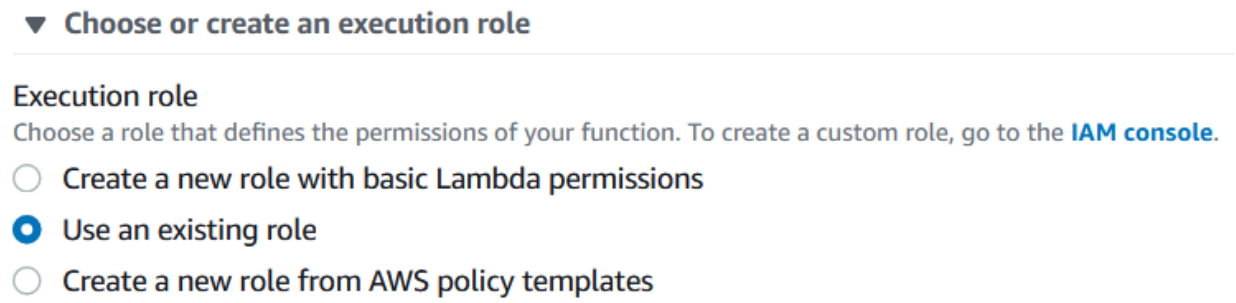
Gambar 420 Create Function Name

Pada kolom **Runtime** pilih **Node.js 12.x** :



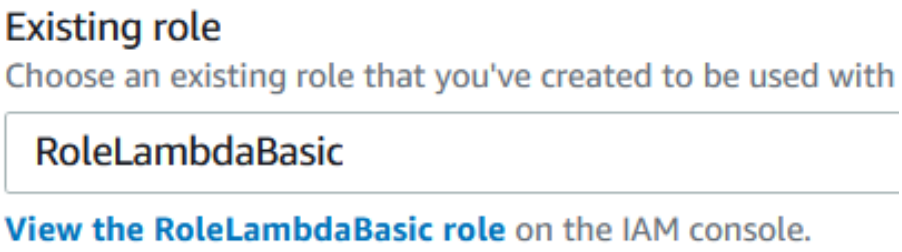
Gambar 421 Choose Lambda Runtime

Pada kolom **Execution role** pilih *Use an existing role* :



Gambar 422 Choose Execution Role

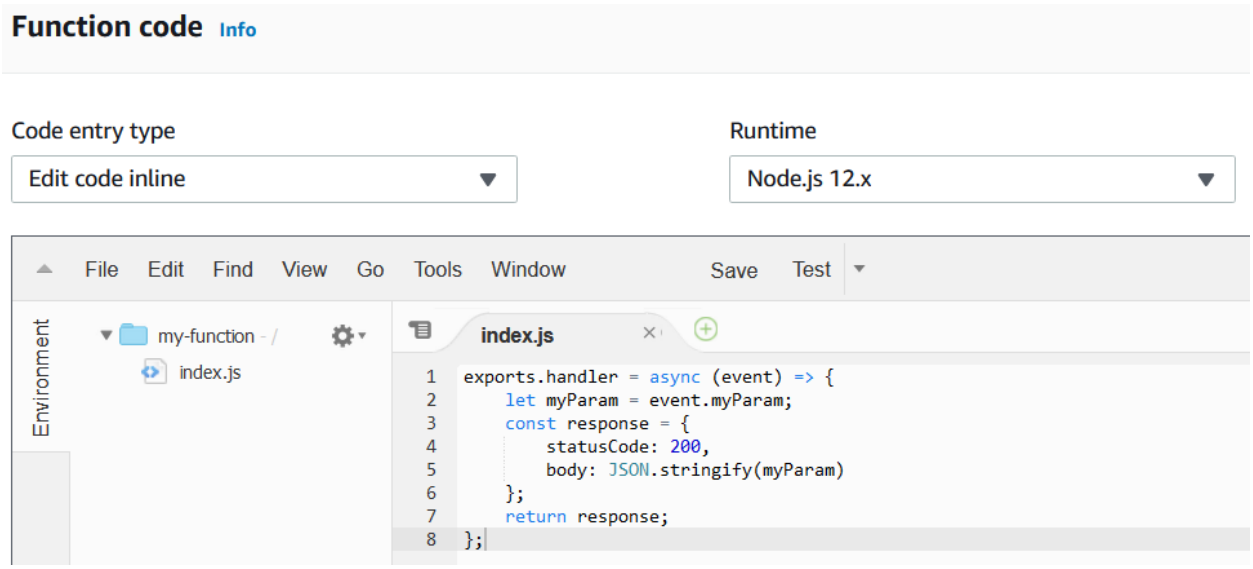
Pilih *IAM Role* yang telah kita buat sebelumnya yaitu **RoleLambdaBasic** :



Gambar 423 Using RoleLambdaBasic

Klik tombol **Create function** untuk menyelesaikan.

Scroll ke bawah, pada kolom *code editor* hapus semua kode yang ada di sana.



Gambar 424 Lambda Function Code Editor

Masukan kode *javascript* di bawah ini :

```
  console.log('Loading the Calc function');

  exports.handler = function(event, context, callback) {
      console.log('Received event:', JSON.stringify(event, null, 2));
      if (event.a === undefined || event.b === undefined || event.op === u
ndefined) {
          callback("400 Invalid Input");
      }

      var res = {};
      res.a = Number(event.a);
      res.b = Number(event.b);
      res.op = event.op;

      if (isNaN(event.a) || isNaN(event.b)) {
          callback("400 Invalid Operand");
      }

      switch(event.op)
      {
          case "+":
          case "add":
```

```
            res.c = res.a + res.b;
            break;
        case "-":
        case "sub":
            res.c = res.a - res.b;
            break;
        case "*":
        case "mul":
            res.c = res.a * res.b;
            break;
        case "/":
        case "div":
            res.c = res.b===0 ? NaN : Number(event.a) / Number(event.b);
            break;
        default:
            callback("400 Invalid Operator");
            break;
    }
    callback(null, res);
};
```

Jika sudah di pojok kanan atas klik tombol **Save**.

Fungsi tersebut membutuhkan dua *operands* (a & b) dan satu *operator* (op) dari *input event*. *Input* adalah sebuah *JSON Object* dengan format sebagai berikut :

```
{
  "a": "Number" | "String",
  "b": "Number" | "String",
  "op": "String"
}
```

Gambar 425 Input Format

Fungsi tersebut akan memberikan sebuah return berupa hasil kalkulasi operasi aritmetika (c) beserta *input* yang diberikan. Jika input yang diberikan tidak sesuai maka fungsi akan

memberikan sebuah *return* berupa nilai **`null`** atau **`“invalid op”`** *string* sebagai hasil akhir. Contoh keluaran menggunakan *JSON Format* :

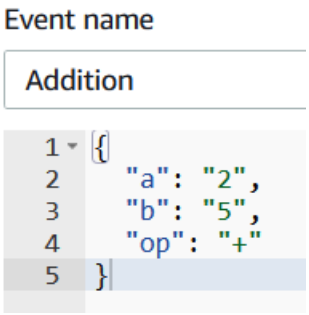
```
{
  "a": "Number",
  "b": "Number",
  "op": "String",
  "c": "Number" | "String"
}
```

Gambar 426 Output Format

Selanjutnya klik tombol **Test** di pojok kanan atas :

Gambar 427 Test Function

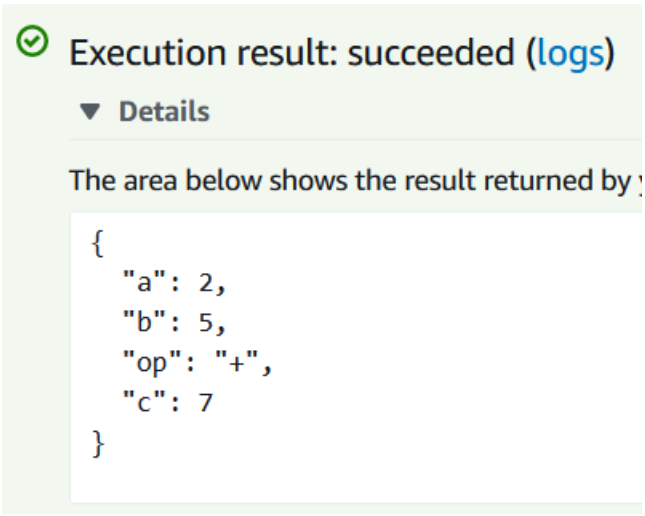
Pada kolom **Event name** beri nama *Addition* dan masukan data *json* seperti pada gambar di bawah ini :



Gambar 428 Create Test Event

Jika sudah klik tombol **Create** di pojok kanan bawah.

Klik kembali tombol **Test** maka anda akan menemukan hasilnya seperti pada gambar di bawah ini :



Gambar 429 Lambda Function Test Result

Subchapter 5 – AWS API Gateway

In business, what's dangerous is not to evolve.

— Jeff Bezos

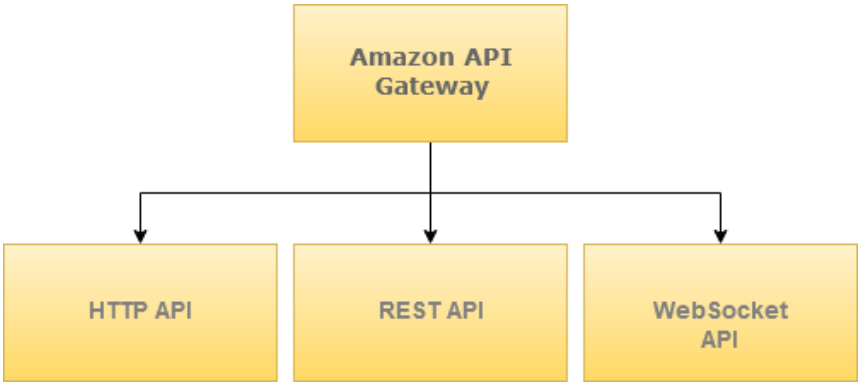
Subchapter 5 – Objectives

- Mengenal *Amazon API Gateway*
- Mengenal *API Developer*
- Mengenal *App Developer*
- Mengenal *Features API Gateway*
- Belajar Membuat ***REST API***

Apa sih itu ***Amazon API Gateway***?

API Gateway adalah sebuah layanan ***AWS*** untuk membuat, mempublikasi, memelihara, memonitor dan mengamankan ***REST API*** dan ***Websocket API***. Dengan ***API Gateway*** proses *deployment* ***REST API*** dan ***Websocket API*** menjadi lebih mudah dan cepat.

Dengan ***API Gateway*** kita dapat menangani ratusan hingga ribuan ***Concurrent API Calls*** sekaligus, memanajemen *Traffic HTTP Request* dengan mudah, dukungan pengelolaan ***CORS***, pembangunan ***authorization*** dan ***access control***, manajemen ***throttling***, dan fitur untuk memanajemen versi ***API*** yang akan dibuat.



Gambar 430 Amazon API Gateway Diagram

1. *API Gateway Service*

HTTP API

Kita dapat menggunakan ***HTTP APIs*** untuk membangun ***high-performance RESTful API*** yang membutuhkan fungsionalitas ***API Proxy*** tanpa manajemen fitur. ***HTTP APIs*** telah dioptimasi untuk ***serverless application*** dan ***HTTP Backend***, dan menawarkan 70% biaya yang lebih hemat dibandingkan ***REST API***.

REST API

Kita dapat menggunakan ***REST API*** untuk menangani pekerjaan yang membutuhkan fungsionalitas ***API Proxy*** dan manajemen fitur, seperti pelacakan (***tracking***) dan penggunakan ***quota*** dengan ***API Keys***, publikasi ***APIs*** dan monetisasi ***APIs***.

WebSocket API

Kita dapat menggunakan ***WebSocket API*** untuk membangun aplikasi ***real-time*** dengan komunikasi dua arah. Seperti pembuatan aplikasi ***chatting***, *game multiplayer* dan ***streaming dashboard***. ***API Gateway*** akan membuat ***persistent connection*** untuk menangani transfer data antara *client* dan *backend*.

2. *API & App Developer*

Terdapat dua *developer* yang akan menggunakan ***Amazon API Gateway*** yaitu :

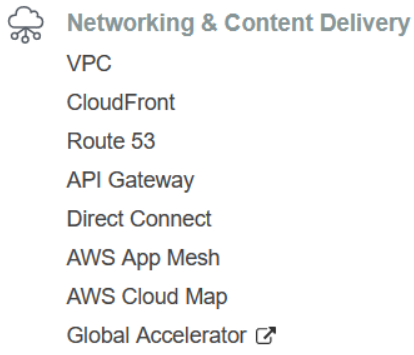
API Developer

API Developer adalah pengembang ***HTTP API, REST API*** atau ***WebSocket API*** di dalam ***Amazon API Gateway***. ***API Developer*** membuat dan melakukan *deployment* sebuah ***APIs*** agar dapat menyediakan sekumpulan fungsi di dalam ***Amazon API Gateway***.

Untuk membuat dan memanajemen *API*, seorang ***API developer*** dapat mengelolanya melalui ***Amazon API Gateway Console, Amazon API Gateway SDK*** dan ***Amazon CLI*** :

API Gateway Console

Untuk masuk ke dalam layananan ***API Gateway Console***, pada menu ***Service*** cari kolom ***Networking & Content Delivery***. Di dalamnya terdapat *link* menuju halaman ***API Gateway Console***.



Gambar 431 API Gateway Console Service

API Gateway SDK

Untuk ***Software Development Kit (SDK)*** telah disediakan untuk berbagai bahasa pemrograman dan ***platform*** agar kita bisa berinteraksi dengan layanan ***Amazon API Gateway***. Pada bahasa pemograman terdapat dukungan untuk bahasa pemrograman

seperti ***C++, Javascript, Python, PHP*** dan seterusnya seperti pada gambar di bawah ini :

Gambar 432 Supported SDK

Untuk mendapatkan informasi lebih lengkap mengenai *SDK* kunjungi halaman ini :

https://aws.amazon.com/getting-started/tools-sdks/

App Developer

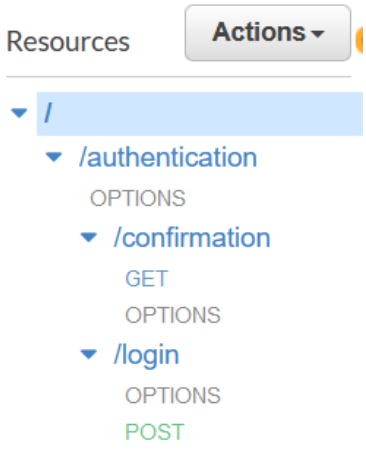
App Developer adalah pengembang aplikasi ***Web, Dekstop, Mobile*** dan ***System*** yang memanfaatkan fungsi-fungsi yang disediakan dalam ***REST API*** atau ***WebSocket API*** pada ***Amazon API Gateway*** yang dikembangkan oleh ***API Developer***.

3. *API Gateway Features*

Di bawah ini adalah fitur-fitur yang disediakan dalam *API Gateway* :

Resources Management

Fitur untuk membuat *Web Resources* di dalam *API*, fitur untuk membuat *Methods* yang disediakan di dalam sebuah *web resources*.



Gambar 433 Resources & Methods

Method Execution Management

Method Execution management membantu kita untuk melakukan kontrol terhadap *input* dan *output data*.

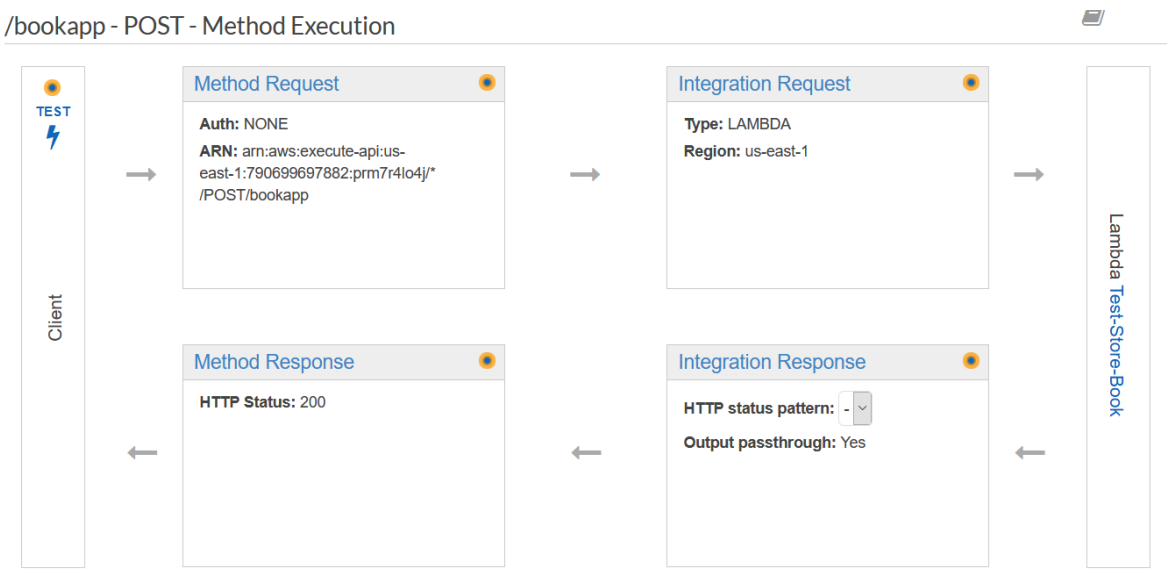
Pada kotak **Client** terdapat ikon **TEST** yang dapat digunakan untuk merepresentasikan sebuah *client* yang akan melakukan pemanggilan API. Fitur ini memberikan kita gambaran besar untuk melakukan simulasi jika *client* mencoba memanggil API melalui *browser, mobile apps, dekstop apps* atau *server apps*.

Pada kotak **Method Request** digunakan untuk merepresentasikan setiap *HTTP Request* dari *client* yang akan diterima oleh *API Gateway*. Di dalamnya terdapat fitur untuk

melakukan validasi *input* pada *HTTP Request* seperti *HTTP Header, HTTP Body & Query String Parameter* sebelum menuju *backend* yaitu *Integration Request*.

Pada kotak **Integration Request** terdapat fitur untuk mengelola *integration type* yang akan menjadi sumber *backend*, kita dapat mengatur agar setiap *HTTP Request* dapat di integrasikan dengan *Lambda Function, External endpoint* atau *AWS Service* lainmya.

Kita juga dapat melakukan ***Mapping Data*** dari ***Method Request*** untuk transformasi data jika diperlukan, sehingga kita dapat mengatur agar *backend* hanya menerima *input data* sesuai dengan yang dibutuhkan. Dengan begitu aplikasi yang dibuat bisa menjadi lebih aman.



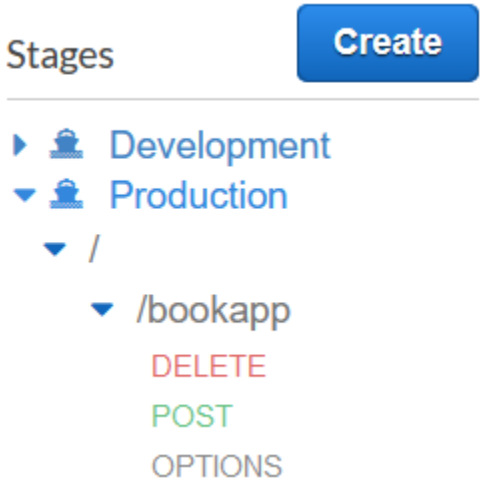
Gambar 434 Method Execution

Pada kotak **Integration Response** digunakan untuk merepresentasikan *response* dari *backend* sebelum dikirimkan menuju *client* sebagai *Method Response*.

Pada kotak **Method Response** digunakan untuk merepresentasikan *HTTP Response* yang akan diterima oleh *client*.

Staging Management

Manajemen ***Staging API*** untuk *development*, *production*, dan *staging* lainya.



Gambar 435 Staging Management

Models Management

Manajemen *Models* menggunakan *JSON Schema* & *Velocity Template Language* agar kita bisa mendapatkan *input data* dari setiap *HTTP Request* sesuai dengan yang kita ingin. Juga kontrol *output data* yang akan diberikan kepada *client* melalui *HTTP Response* sehingga aplikasi yang kita buat lebih aman.

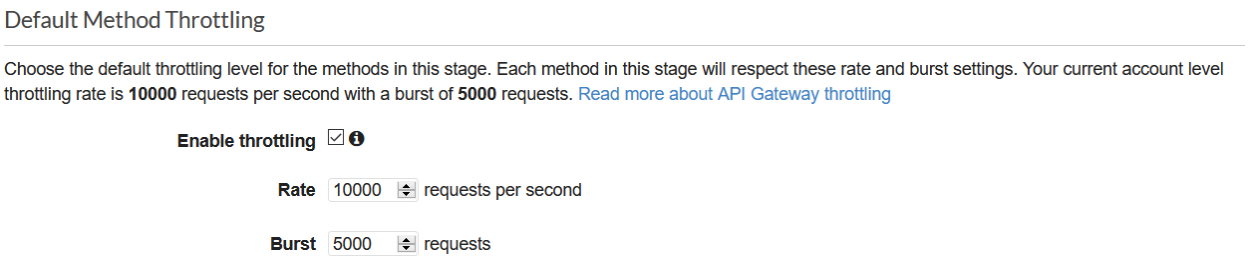
Model schema*

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Login",
    "type": "object",
    "properties": {
        "email": {
            "type":"string",
            "minLength": 8,
            "maxLength": 255
        },
        "password": {
            "type":"string",
            "minLength": 8,
            "maxLength": 150
        }
    },
    "required": ["email", "password"]
}
```

Gambar 436 Models Management

Throttling Management

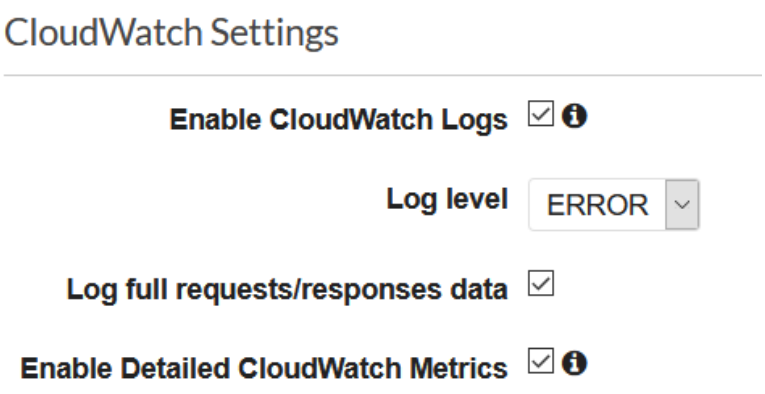
Manajemen ***Throttling*** untuk mencegah *API* yang kita buat untuk tidak digunakan secara berlebihan dan tidak wajar.



Gambar 437 Throttling Management

AWS CloudWatch Integration

Integrasi dengan *AWS CloudWatch* untuk melakukan *logging*, sehingga seluruh data terkait *request & response* akan tercatat.



Gambar 438 Setting CloudWatch Logging

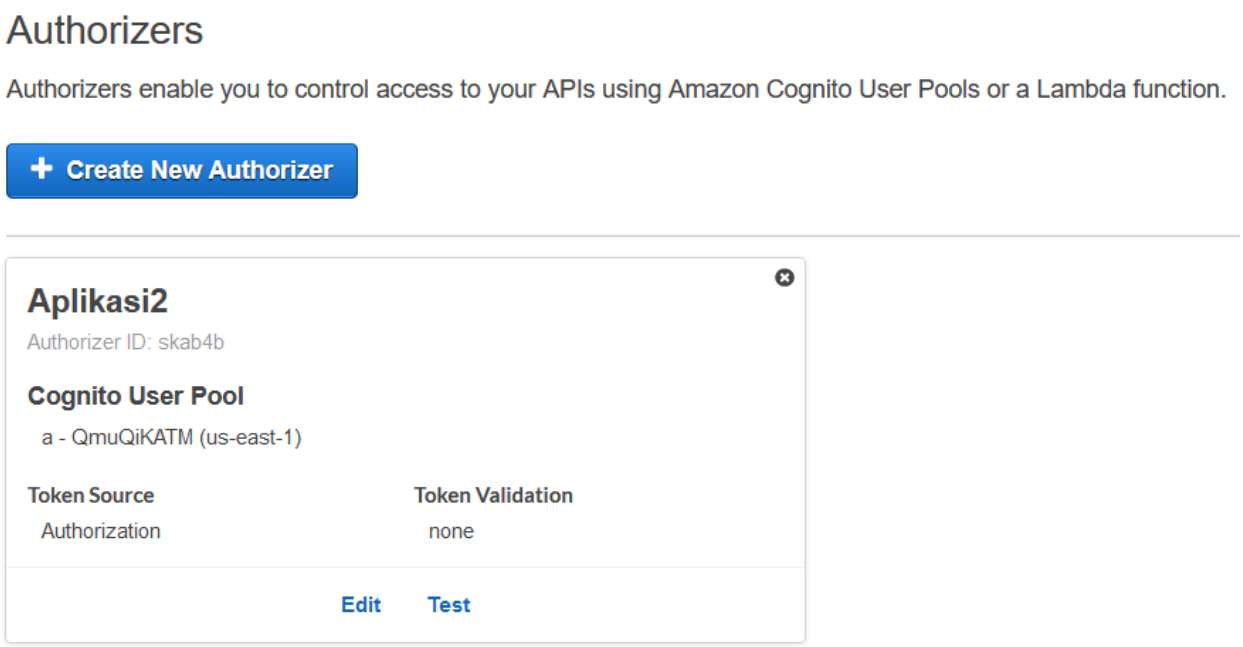
AWS X-Ray Integration

Kita dapat memanfaatkan *AWS X-Ray service maps* dan *trace view* dengan *API Gateway*.

Gambar 439 Enable X-Ray Tracing

AWS Cognito Integration

Integrasi dengan *AWS Cognito* agar *API* yang kita buat dapat dibatasi hanya untuk *user* saja.

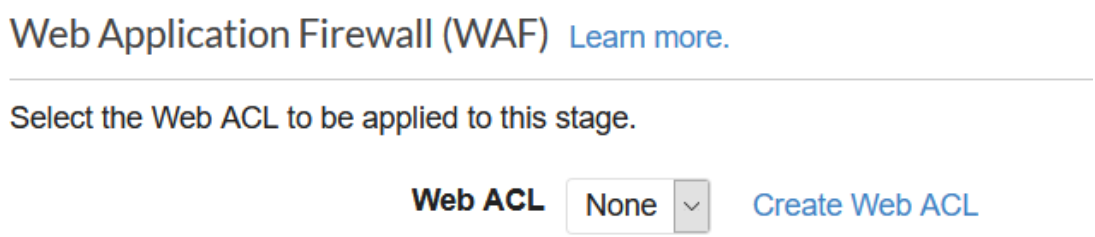


Gambar 440 AWS Cognito Integration

AWS WAF Integration

Integrasi dengan *AWS WAF* (*Web Application Firewall*) untuk mencegah *API Gateway* yang kita buat dieksploitasi menggunakan *Cross-site scripting (XSS) Attack & SQL Injection*. Jika *HTTP Request* yang diberikan oleh *client* mengandung *malicious SQL Code & Malicious script*, *request* tersebut akan di *block*.

Fitur Unggulan lainya seperti *block* berdasarkan *IP, block* berdasarkan lokasi seperti negara dan benua, *block* berdasarkan *rules-based* agar kita bisa mencegah berdasarkan *user agent* tertentu, mencegah *bad bot* dan *content scrapper*.



Gambar 441 Web Application Firewall Configuration

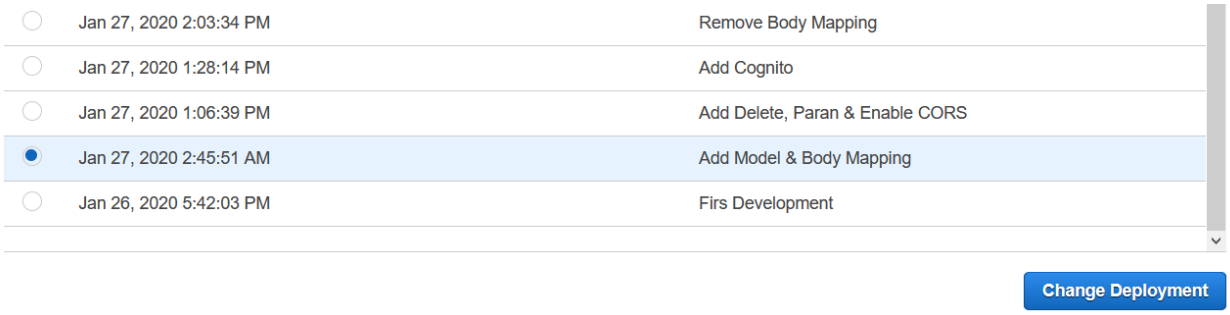
Export API

Export API sebagai *Swagger, Postman* dan *API Gateway Extension* membuat produktifitas menjadi lebih cepat dalam mengembangkan dan menguji API.

Gambar 442 Export as Swagger

Deployment History

Terdapat fitur *Deployment History* kita dapat melakukan *Roll Back API* untuk kembali ke versi *API* sebelumnya atau versi *API* yang lebih lama.



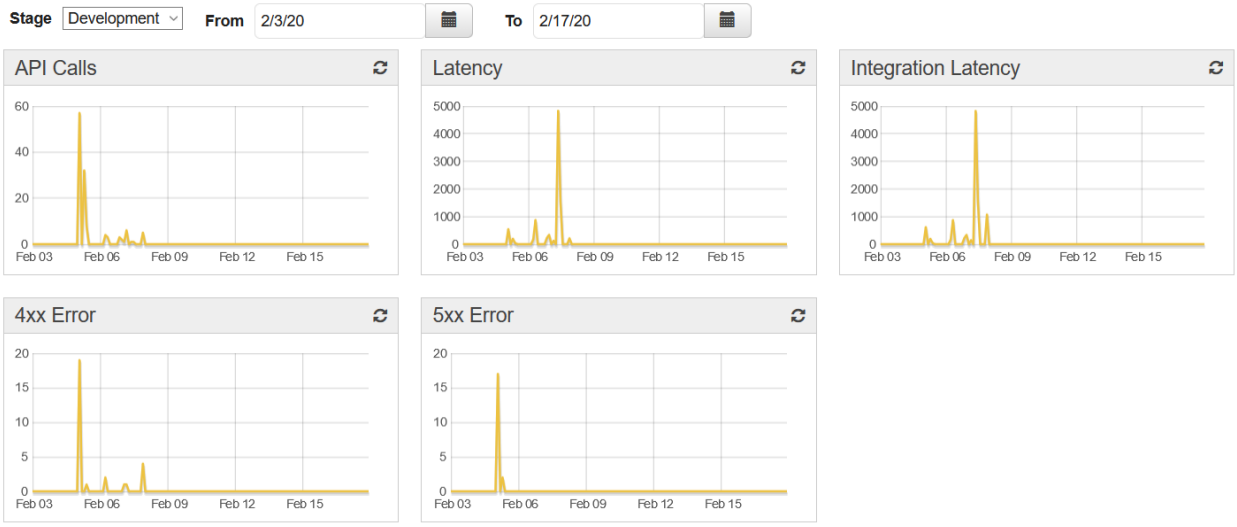
Gambar 443 Deployment History

Documentation

Terdapat fitur untuk membuat dan mempublikasikan *documentation* agar *developer* lainya dapat berinteraksi dengan API.

Dashboard Metrics

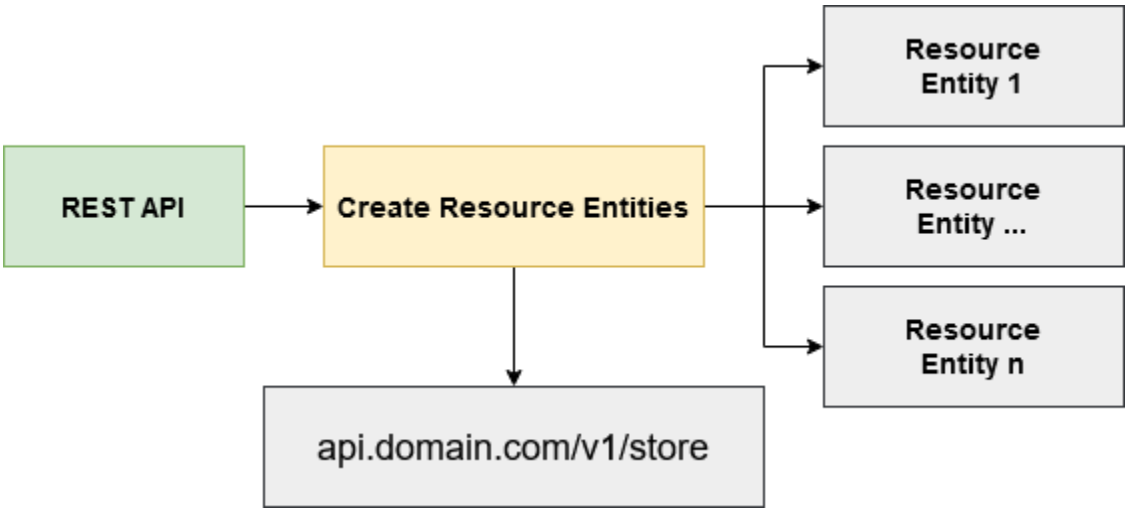
Terdapat *fitur dashboard* yang digunakan untuk melihat *metrics* penggunaan *API* seperti, *API Calls, Latency, Integration Latency, 4xx & 5xx Error*. *Metrics* tersebut dalam dilihat berdasarkan *Stage* dan Tanggal penggunaan.



Gambar 444 Dashboard Metrics

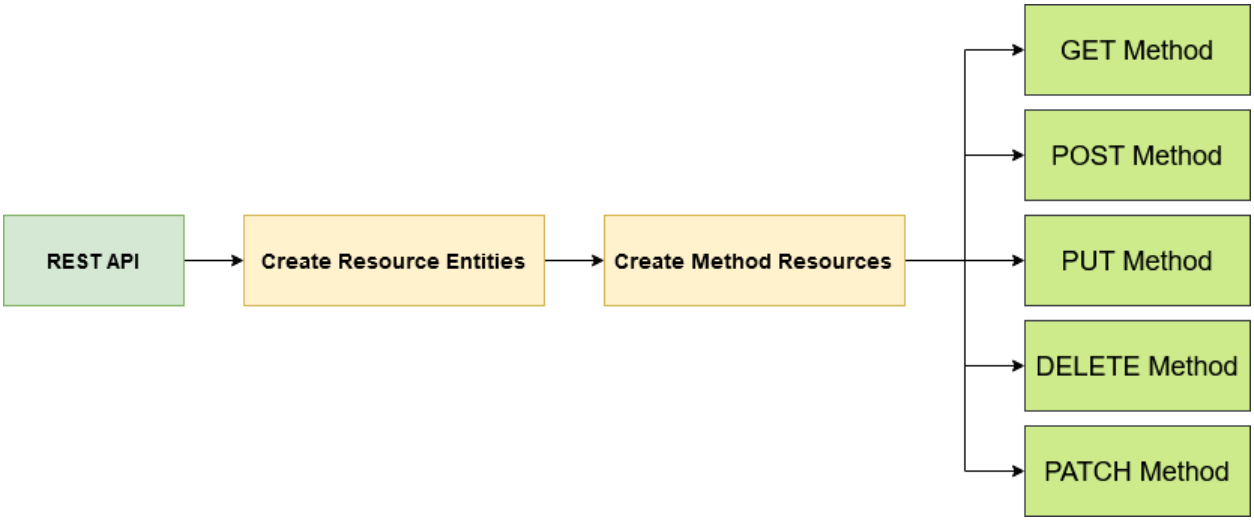
4. *REST API*

Saat kita membuat sebuah *REST API* menggunakan *Amazon API Gateway*, kita akan membuat sebuah *Web Resources*. Dalam suatu *REST API* terdapat sekumpulan *Resources Entities* yang akan kita buat, sebagai contoh pada gambar di bawah ini kita dapat membuat sebuah *Resources Entity* dengan *path* api.domain.com/v1/store :



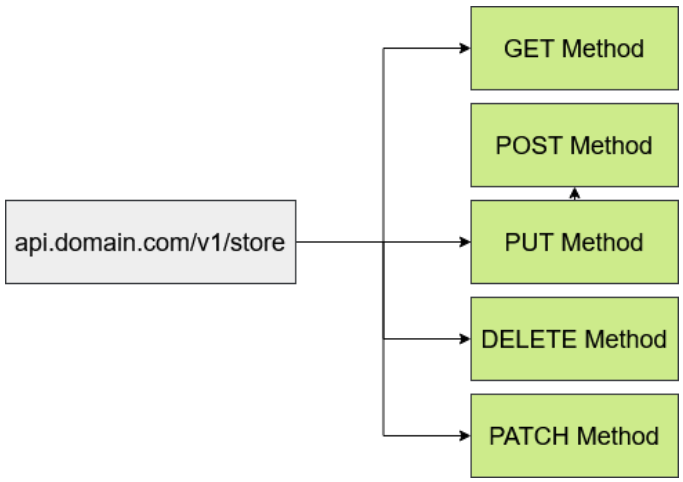
Gambar 445 Resource Entity

Setiap *resource entity* dapat memiliki satu atau lebih *method resources* :



Gambar 446 Method Resources

Sebagai contoh pada *resource entity* api.domain.com/v1/store kita dapat menerapkan salah satu atau seluruh *HTTP Method* :



Gambar 447 Method Resources

Terdapat *HTTP Method* lainya yang dapat digunakan sebagai *resources*.

HTTP Method	RFC	Request Has Body	Response Has Body	Safe	Idempotent	Cacheable
GET	RFC 7231	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 7231	No	No	Yes	Yes	Yes
POST	RFC 7231	Yes	Yes	No	No	Yes
PUT	RFC 7231	Yes	Yes	No	Yes	No
DELETE	RFC 7231	No	Yes	No	Yes	No
CONNECT	RFC 7231	Yes	Yes	No	No	No
OPTIONS	RFC 7231	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	No

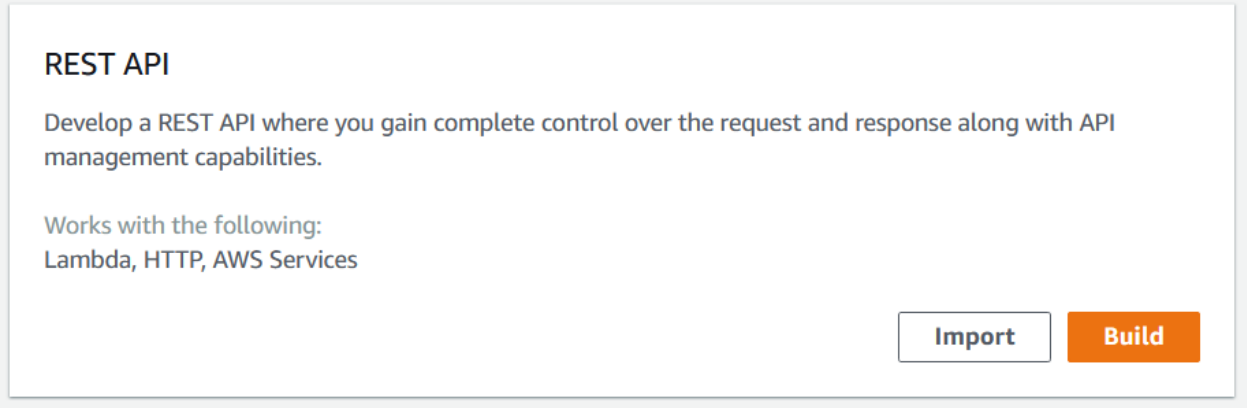
Untuk mengendalikan bagaimana *clients* dapat memanggil API, kita dapat menggunakan :

1. *IAM Permissions*
2. *Lambda Authorizer*
3. *Amazon Cognito User Pool*

Untuk membatasi penggunaan API kita dapat mengatur *usage plans* agar bisa melakukan *throttle* untuk setiap *API Request*.

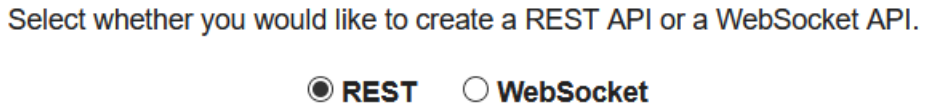
Create REST API

Masuk ke dalam *API Gateway Console*, *click here*. Klik Tombol **Create API**, kemudian cari kolom *REST API* dan klik tombol **Build** seperti pada gambar di bawah ini :



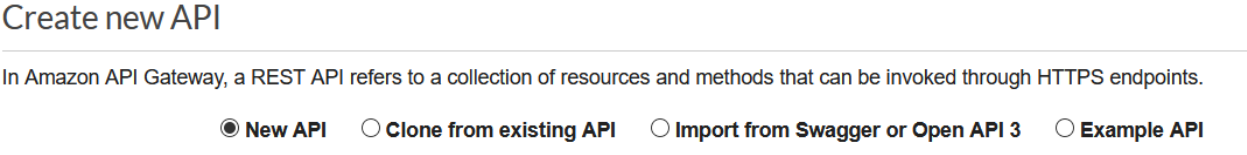
Gambar 448 Build REST API

Disini terdapat dua protokol yang dapat kita pilih. Pilih ***REST*** :



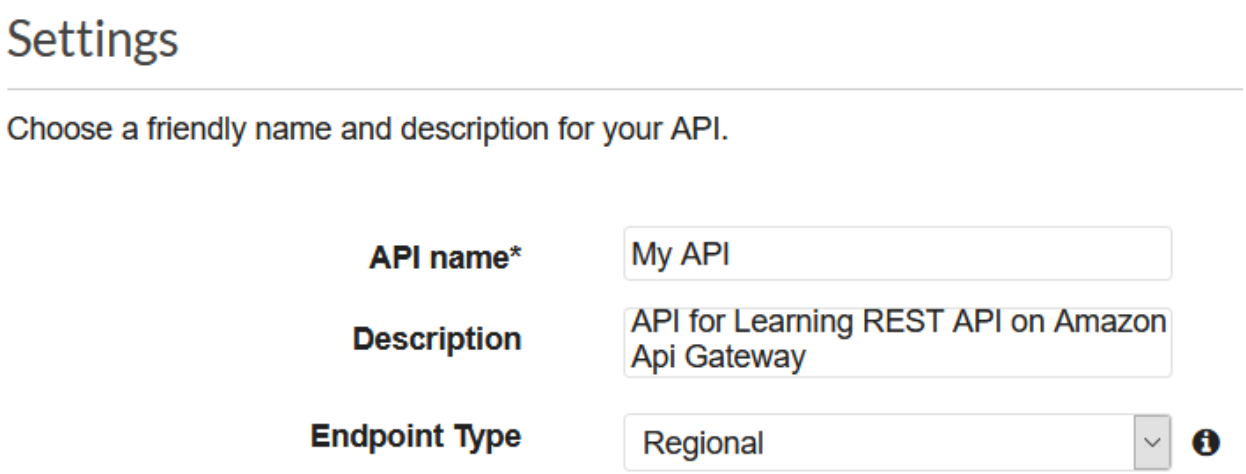
Gambar 449 Protocol for API

Disini terdapat pilihan untuk membuat API baru, kloning dari API yang telah dibuat, *import* API melalui *Swagger* atau *Open API 3* atau menggunakan *Example API* yang telah disediakan pihak *AWS* sebagai sumber belajar kita. Pada kasus ini pilih *New API* :



Gambar 450 Create New API

Pada kolom **API name*** isi nama *API* seperti pada gambar di bawah ini dan pada kolom **Endpoint Type** pilih ***Regional*** :

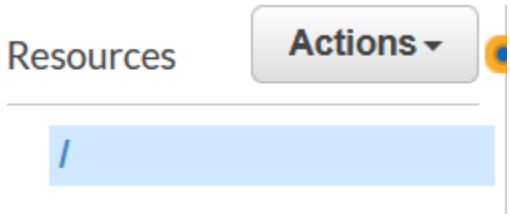


Gambar 451 API Settings

Jika sudah klik tombol *Create API* di pojok kanan bawah.

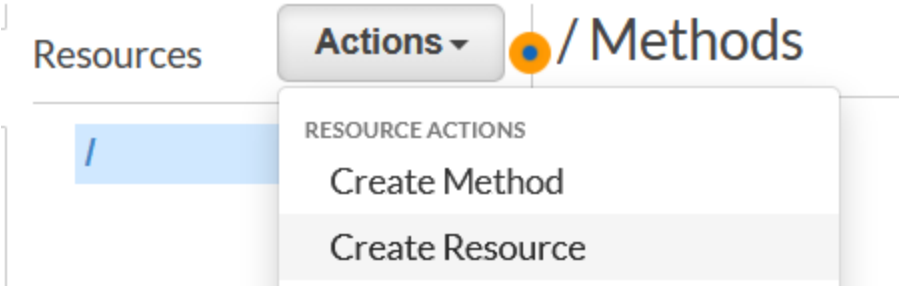
Create Resource

Saat pertama kali kita membuat *REST API* kita akan melihat pada kolom **Resources** hanya terdapat satu ***path*** saja yang disebut dengan ***Root Resource*** (/).



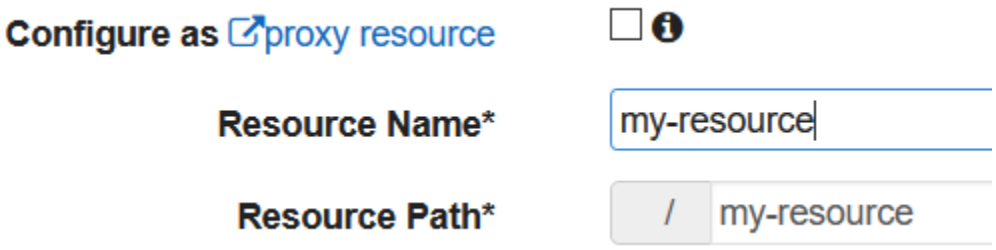
Gambar 452 Root Resource

Klik tombol **Actions** kemudian pilih **Create Resource** :



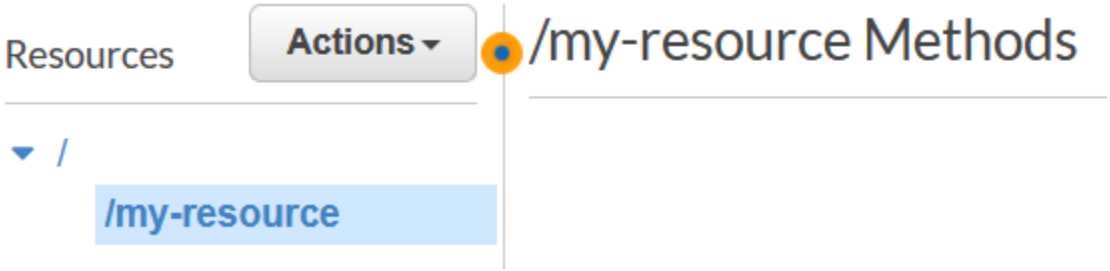
Gambar 453 Create Resource

Pada kolom **Resource Name*** isi sesuai dengan gambar di bawah ini :



*Gambar 454 Create Resource Name**

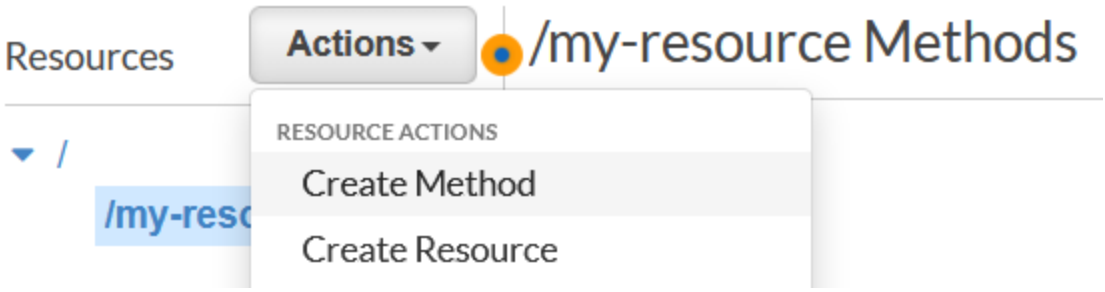
Jika berhasil maka kita memiliki *resource* baru dengan path ***/my-resource***



Gambar 455 New Resource

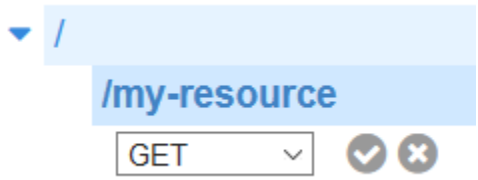
Create Method

Klik kembali tombol **Actions** kemudian pilih **Create Method** :



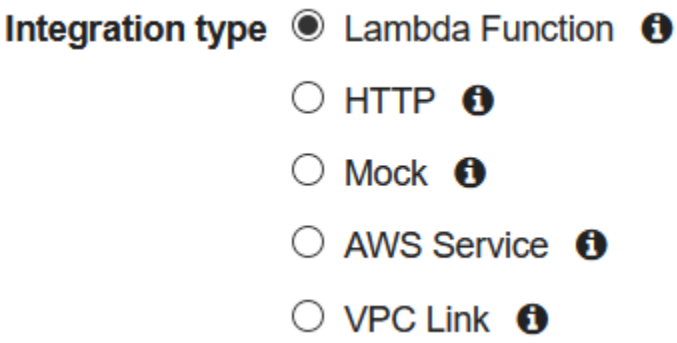
Gambar 456 Create Method

Selanjutnya pilih ***GET*** *Method* :



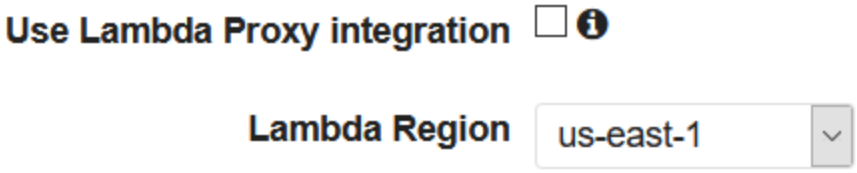
Gambar 457 Create GET Method

Pada pilihan **Integration type** pilih ***Lambda Function*** :



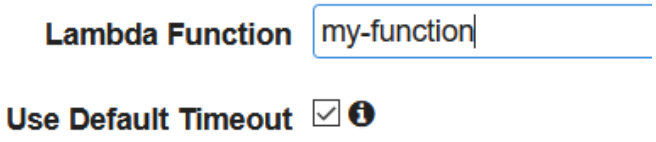
Gambar 458 Lambda Function Integration

Jangan centang **Use Lambda Proxy Integration** kemudian pilih **Lambda Region** sesuai dengan lokasi tempat anda membuat ***lambda function***, pada kasus kali ini penulis menggunakan ***us-east-1*** *region* :



Gambar 459 Use Lambda Proxy Integration

Pada kolom **Lambda Function**, sebelum bisa memilih menggunakan ***my-function*** kita harus membuat fungsi tersebut terlebih dahulu. Untuk membuat fungsi tersebut silahkan anda kunjungi halaman di bawah ini. *Click here*.

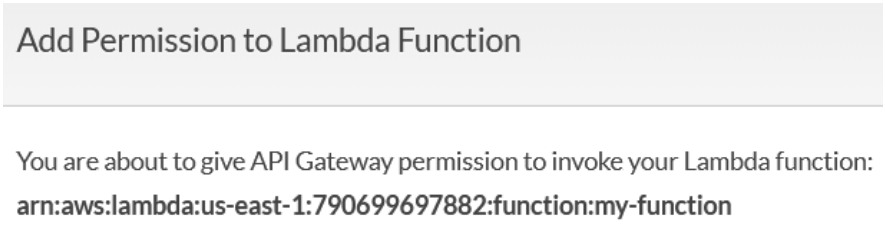


Gambar 460 Lambda Function

Jika sudah kembali lagi ke halaman ini dan *reload API Gateway Console* anda lalu pada kolom **Lambda Function** anda sudah bisa memilih ***my-function***. *Check List* pengaturan **Use Default Timeout**.

Klik tombol **Save** di pojok kanan bawah.

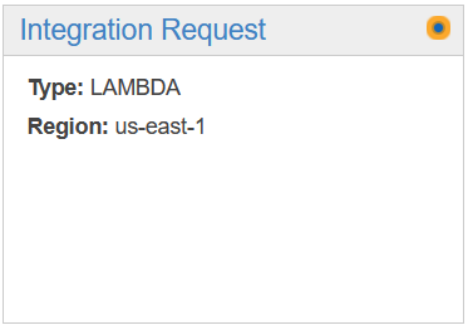
Jika berhasil maka akan muncul informasi agar kita mengizinkan *API Gateway* untuk bisa memanggil *lambda function* :



Gambar 461 Permission Alert

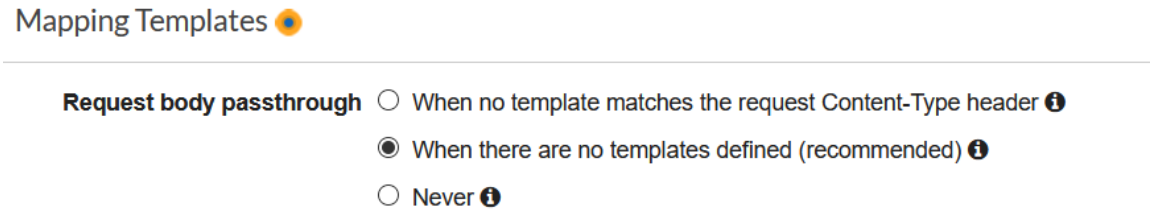
Integration Request

Pada menu *Method Execution Management* klik **Integration Request** :



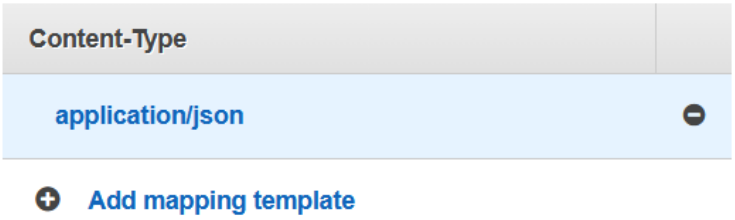
Gambar 462 Integration Request

Pada menu **Mapping Templates** pilih menu seperti pada gambar di bawah ini :



Gambar 463 Request body passthrough configuration

Pada menu **Content-Type** jadikan ***application/json*** dengan cara klik *link* **Add maping template** :



Gambar 464 Add mapping template

Pada kolom **Generate template** masukan *code* di bawah ini :

Gambar 465 Velocity Template Language Example

Pada gambar di atas, kita akan membuat representasi ***JSON object*** yang akan diterima oleh *lambda function*. Sebuah *object* yang hanya memiliki satu *property* saja yaitu ***parameter***. Nilai dari *property* tersebut akan diambil dari *query string* ***myparam***.

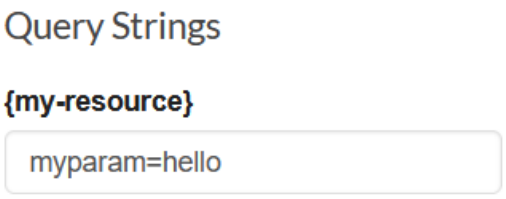
Kita akan mempelajari *Velocity Template Language* di *chapter* berikutnya. Untuk saat ini sampai disini dulu, jika sudah klik tombol **Save**.

Test API

Kembali ke menu *Method Execution Management* klik **TEST**.

Gambar 466 TEST environment

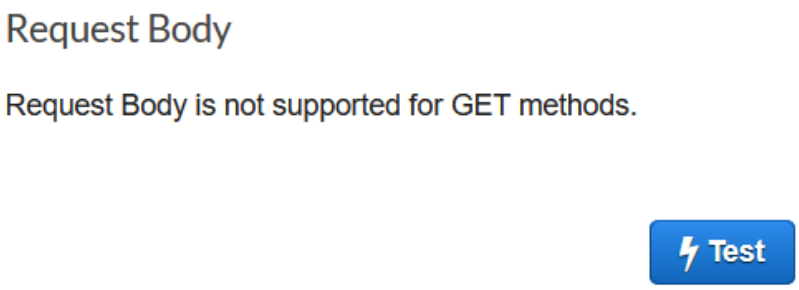
Pada kolom *Query Strings* masukan nilai ***myparam=hello*** seperti pada gambar di bawah ini :



Gambar 467 Query String

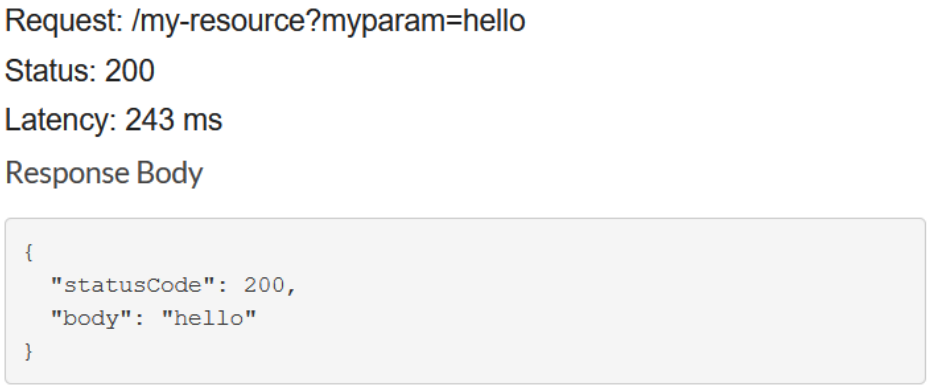
Nilai ***hello*** dari myparam akan dibaca dalam *integration request*, nilai ***hello*** akan disimpan di dalam *property* ***parameter*** yang selanjutnya akan di baca oleh *lambda function*.

Scroll ke bawah klik tombol **Test** untuk menguji hasilnya :



Gambar 468 Test GET Request

Di bawah ini adalah hasil nya :

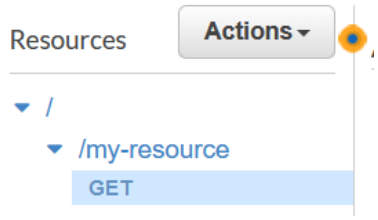


Gambar 469 HTTP Response Body

Jika kita mengunjungi *web resource* dengan *path* ***/my-resource*** lengkap dengan *query string* ***myparam=hello*** maka kita akan mendapatkan *HTTP Response Body* seperti pada gambar di bawah di atas.

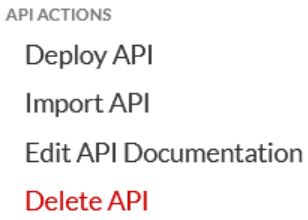
Deploy API

Sekarang kita akan melakukan proses *deployment*, klik tombol **Actions**.



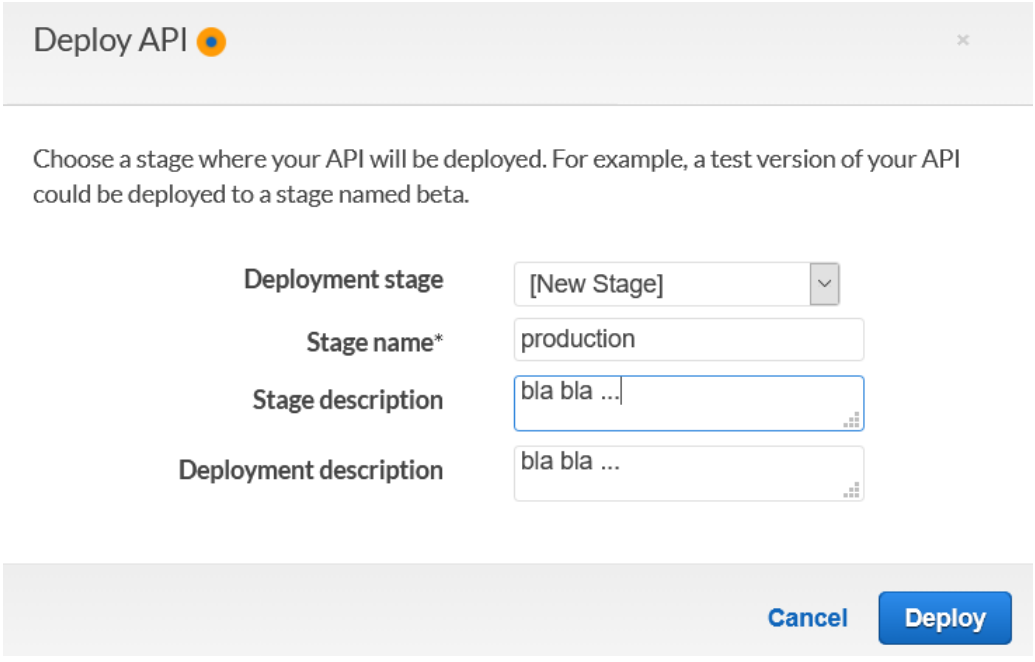
Gambar 470 Actions Deployment

Pilih **Deploy API** :



Gambar 471 Deploy API

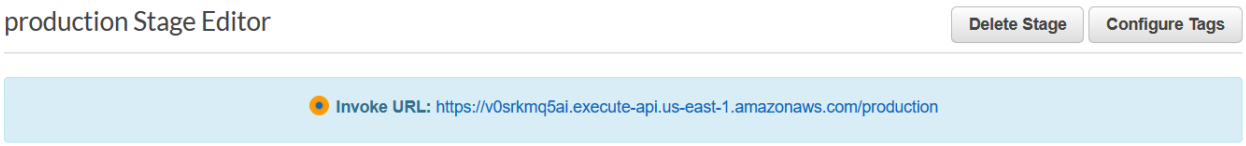
Pada kolom **`Deployment stage`** pilih **`[New Stage]`** dan pada kolom **`Stage name*`** isi dengan nilai **`production`**, pada kolom *description* untuk kedepannya bisa anda isi sesuai dengan kebutuhan :



Gambar 472 Deploy API

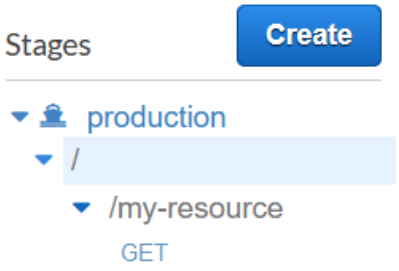
Jika sudah klik **Deploy**.

Jika berhasil maka anda akan melihat **production Stage Editor** seperti pada gambar di bawah ini :



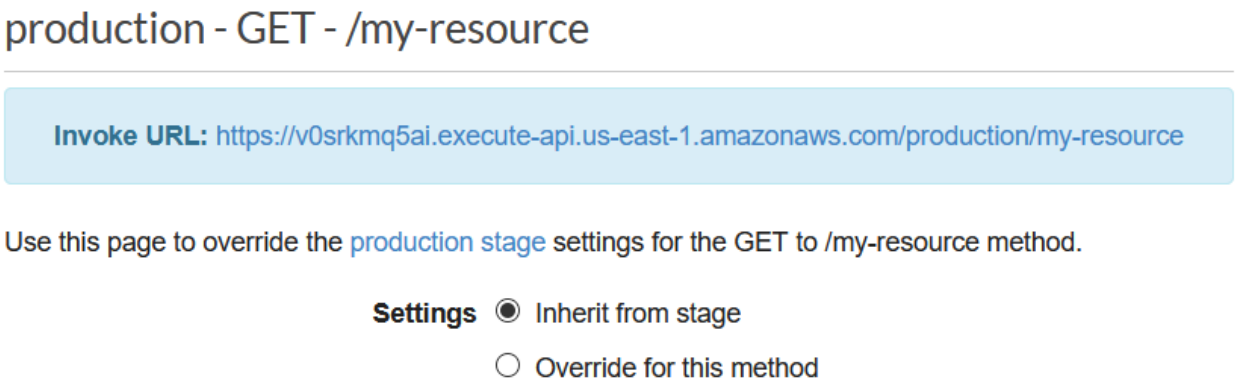
Gambar 473 Production Stage Editor

Pada kolom **stages** anda akan melihat ada *stage* baru bernama **production** :



Gambar 474 Stage API

Klik **GET method** di dalam **/my-resource**, anda akan menemukan *link* seperti pada gambar di bawah ini :

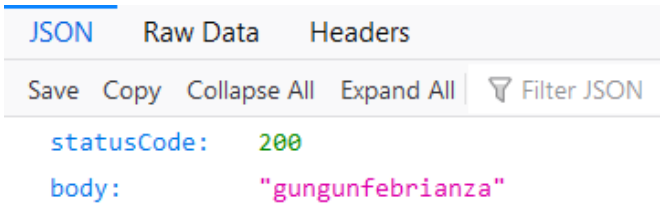


Gambar 475 GET URL my-resource

Jika *link* tersebut kita beri *query string* **`myparam=gungunfebrianza`** menjadi :

https://v0srkmq5ai.execute-api.us-east-1.amazonaws.com/production/my-resource?myparam=gungunfebrianza

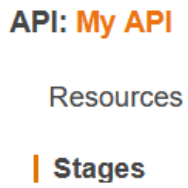
Maka kita akan mendapatkan hasil seperti pada gambar di bawah ini :



Gambar 476 HTTP Response Body in the browser

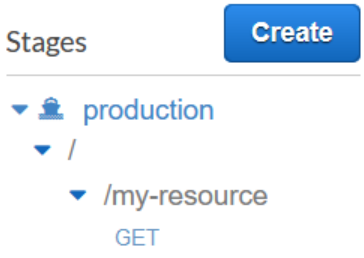
Export to Postman

Selanjutnya kita akan melakukan *export API* yang telah kita buat agar dapat digunakan di dalam Postman. Klik menu **Stages** :



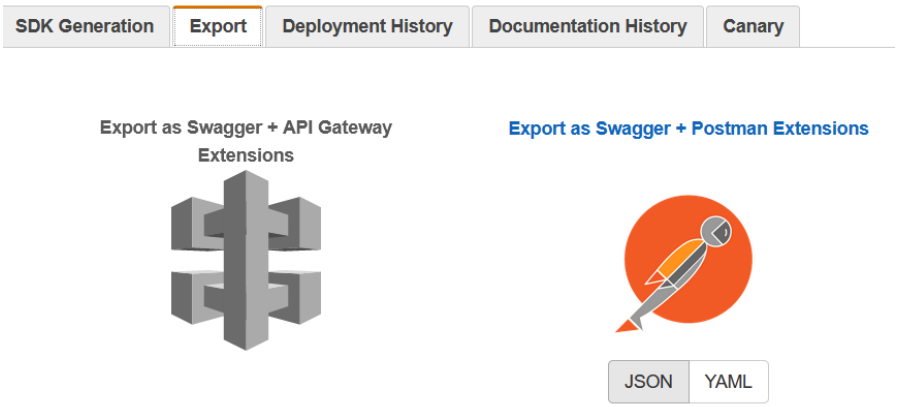
Gambar 477 Stages menu

Klik label **production** :



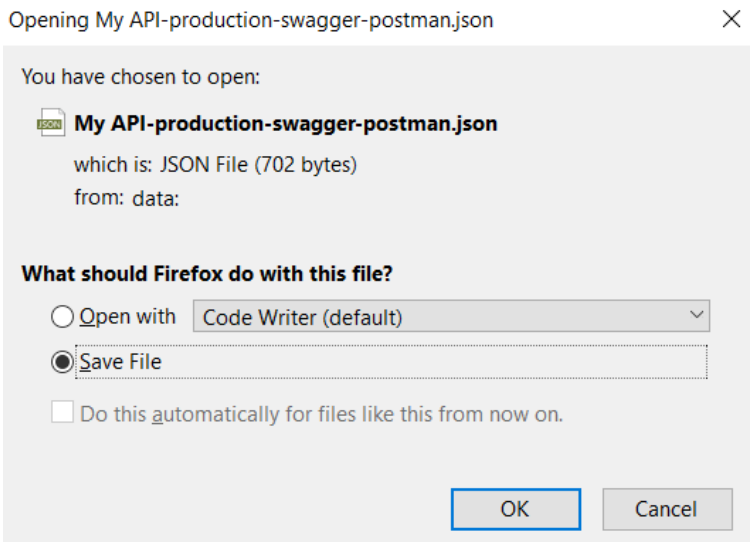
Gambar 478 production stage

Selanjutnya di kolom sebelah kanan akan muncul **production Stage Editor**, pilih *tab* **Export** kemudian klik *JSON* pada gambar *Postman* :



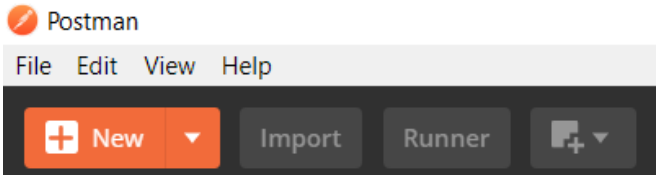
Gambar 479 Export Tab

Simpan *file swagger* yang kita *export* :



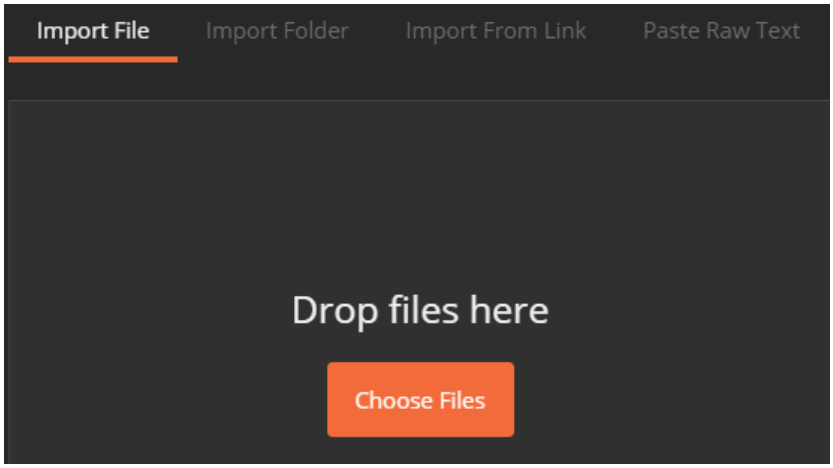
Gambar 480 Save Swagger Schema

Buka aplikasi *Postman* kemudian klik tombol **Import** :



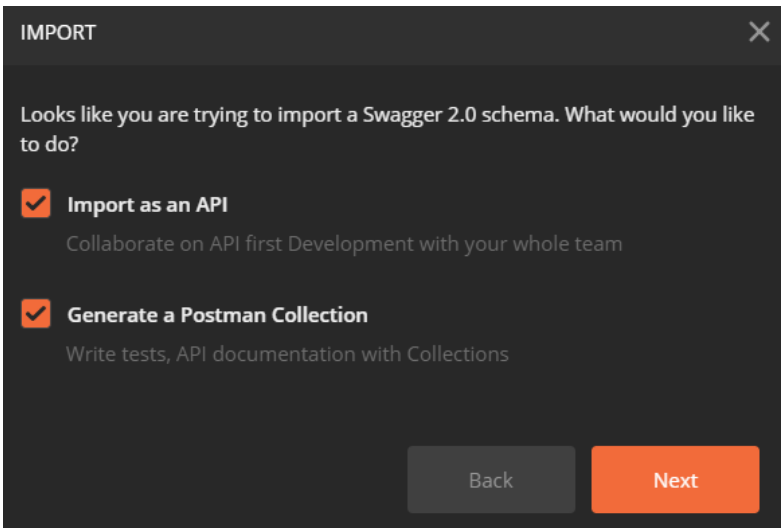
Gambar 481 Import Postman File

Klik tombol **Choose Files** :



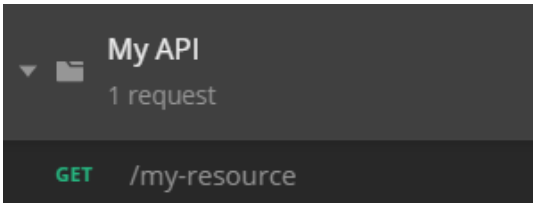
Gambar 482 Choose Files

Pada kolom dialog di bawah ini klik tombol **Next** :



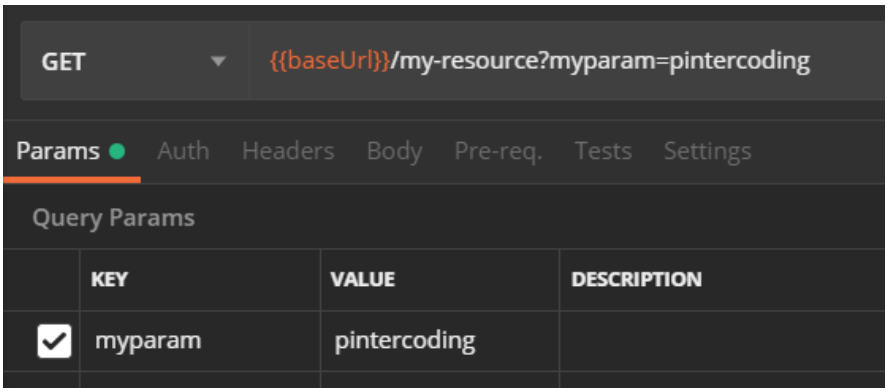
Gambar 483 Import as an API

Maka pada kolom *collection* pada aplikasi *Postman* akan muncul *collection* baru seperti pada gambar di bawah ini :



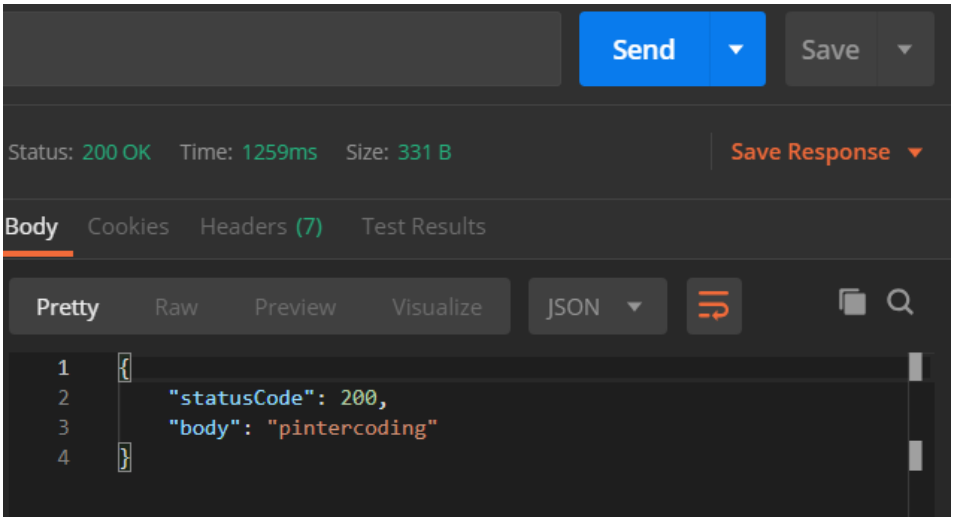
Gambar 484 My API Collection

Kita akan menguji API ***/my-resource*** dengan *GET Method* seperti pada gambar di bawah ini :



Gambar 485 GET Method Testing

Pada kolom **Params**, masukan **Query Params** dengan ***Key myparam*** dan ***Value pintercoding*** seperti gambar di atas, kemudian klik tombol **Send** untuk melihat hasilnya :

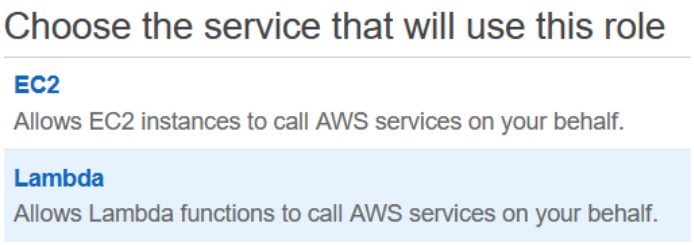


Gambar 486 HTTP Response on Postman

5. *Debugging & Troubleshooting*

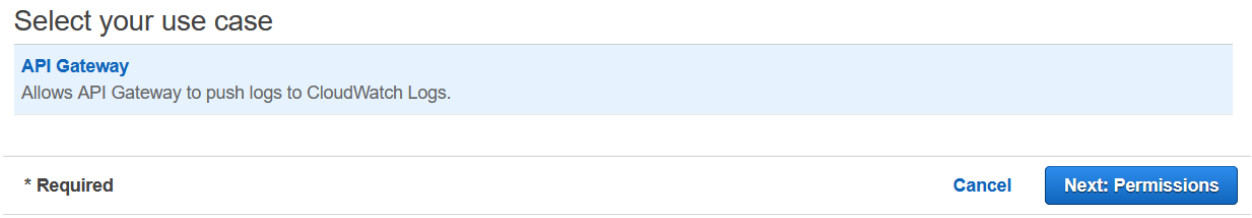
Hal pertama yang harus kita lakukan agar dapat melakukan inspeksi *error* pada *server API Gateway*, kita harus mengizinkan layanan *API Gateway* agar dapat memiliki akses menuju layanan *CloudWatch*. Dengan begitu layanan *API Gateway* dapat melakukan *push data* pada *CloudWatch Logs*. Untuk dapat melakukanya kita perlu membuat *role* baru lagi khusus untuk *API Gateway* agar bisa berinteraksi dengan *CloudWatch Logs*.

Masuk ke halaman *dashboard Amazon Identity Access Management* (IAM), kemudian pilih ***Create Role***. Pada menu ***Choose the service that will use this role***, pilih *Lambda* seperti pada gambar di bawah ini :



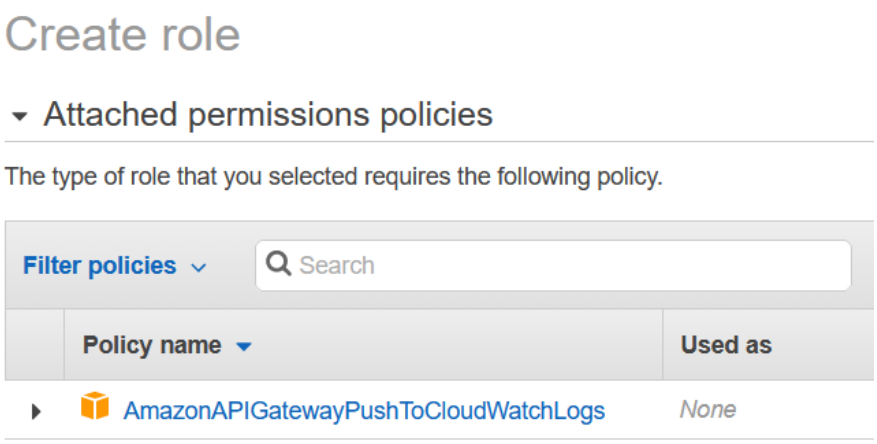
Gambar 487 Setup Role for Lambda

Jika sudah klik tombol ***permission*** di pojok kanan bawah, seperti gambar di bawah ini :



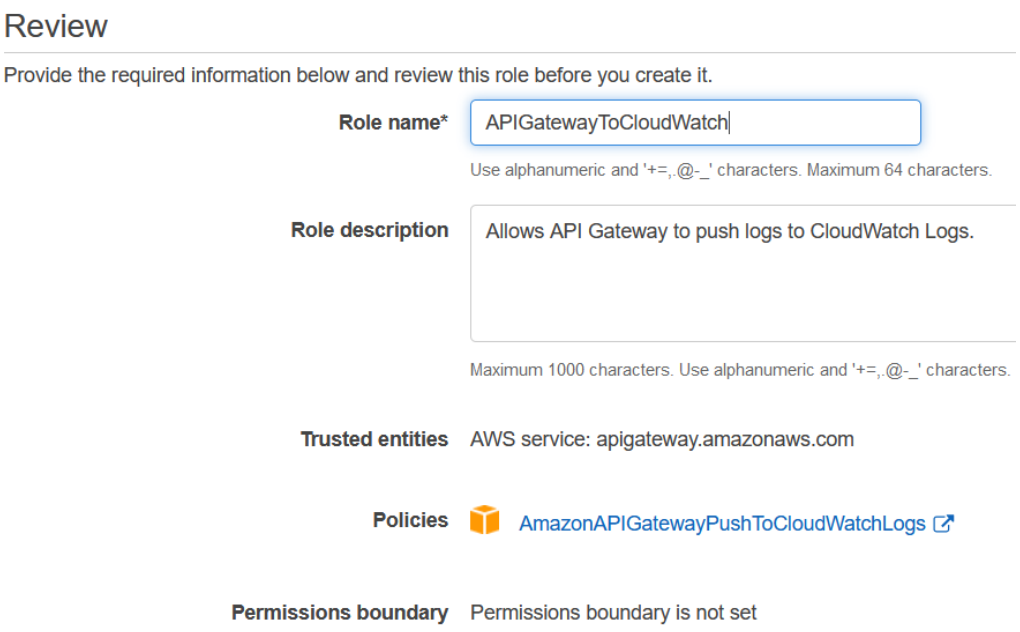
Gambar 488 Setup Permission for Lambda

Pada kolom ***Create Role***, pilih kebijakan yang akan diterapkan pada ***API Gateway***. Pada kolom ***Filter policies*** ketik A***mazonAPIGatewayPushToCloudWatchLogs*** seperti gambar di bawah ini :



Gambar 489 Add Policy to Role

Jika sudah klik tombol ***Tag*** di pojok kanan bawah, isi *tag* anda kemudian klik lagi tombol ***Review*** untuk memeriksa ulang kebijakan yang telah kita buat untuk *role* baru. Pada kolom ***Role name*** isi dengan nama ***APIGatewayToCloudWatch*** dan pada ***Role Description*** isi dengan penjelasan tujuan dari *role* yang sedang kita buat.

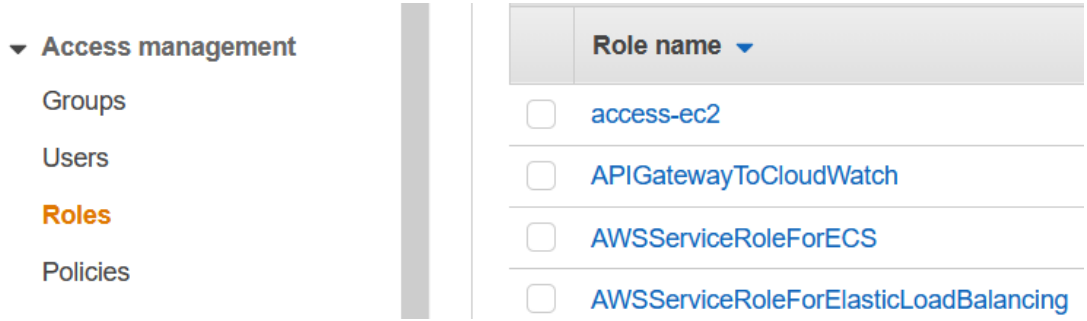


Gambar 490 Role Review

Jika sudah klik tombol ***Create role*** di pojok kanan bawah. Jika berhasil maka akan muncul pesan seperti pada gambar di bawah ini :

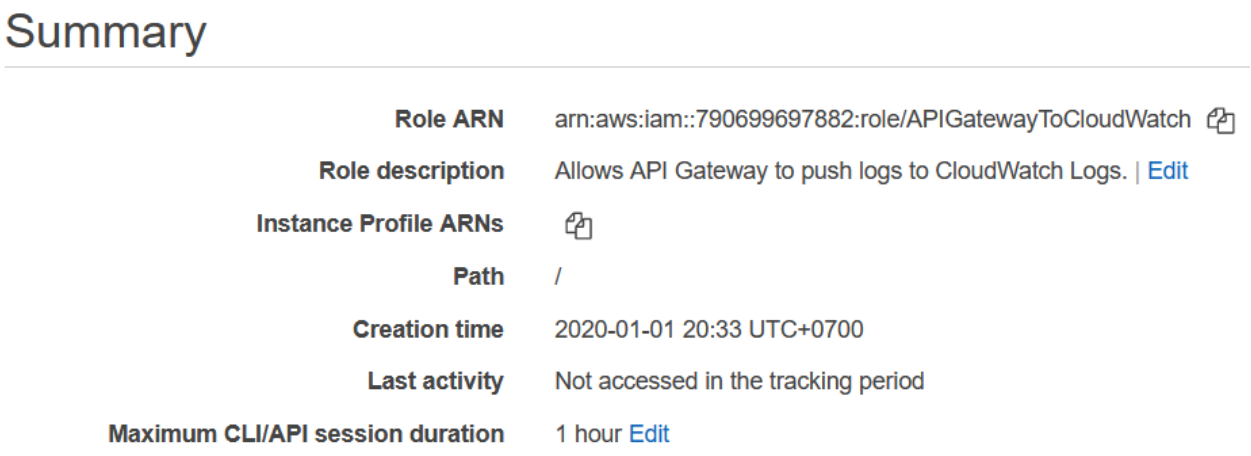
Gambar 491 Success Message For Role Creation

Pada kolom ***Role name*** di dalam *dashboard IAM* akan muncul *Role* baru bernama ***APIGatewayToCloudWatch*** seperti pada gambar di bawah ini :



Gambar 492 List Roles on IAM Dashboard

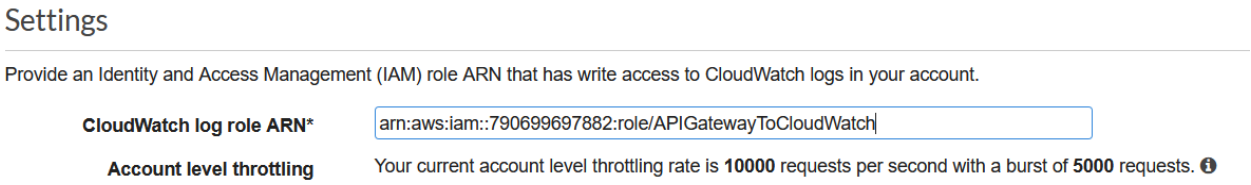
Klik ***APIGatewayToCloudWatch*** kemudian anda akan melihat ***Summary*** dari *Role* tersebut seperti pada gambar di bawah ini :



Gambar 493 Summary of APIGateway Role

Terdapat informasi ***Role ARN***, salin data tersebut kemudian anda harus masuk ke halaman tempat layanan *Amazon Api Gateway*. Di dalam *dashboard API Gateway* pilih

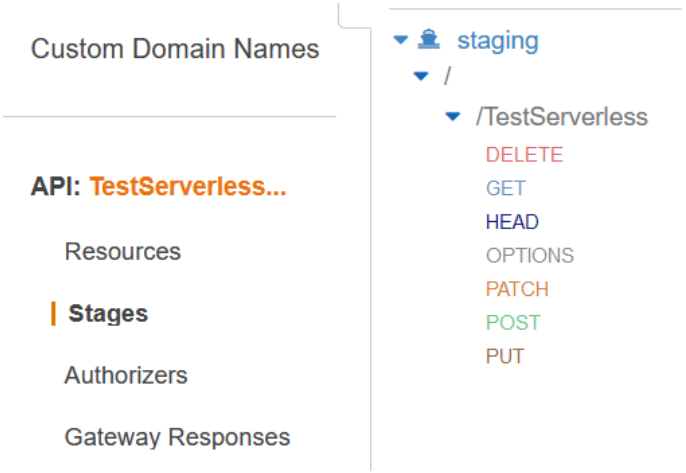
Settings, kemudian anda akan melihat pengaturan ***Settings*** seperti pada gambar di bawah ini :



Gambar 494 Amazon API Gateway Settings

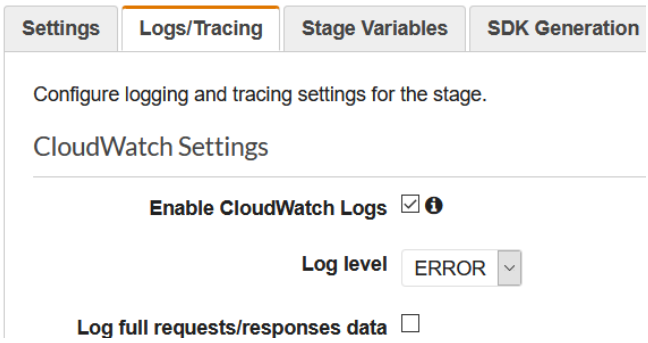
Lakukan paste ***Role ARN*** yang telah kita salin sebelumnya disini, di dalam kolom ***CloudWatch log role ARN**** seperti pada gambar di atas. Jika sudah klik tombol ***save*** di pojok kanan bawah.

Selanjutnya masih di dalam *dashboard API Gateway* pilih menu ***Stages***,



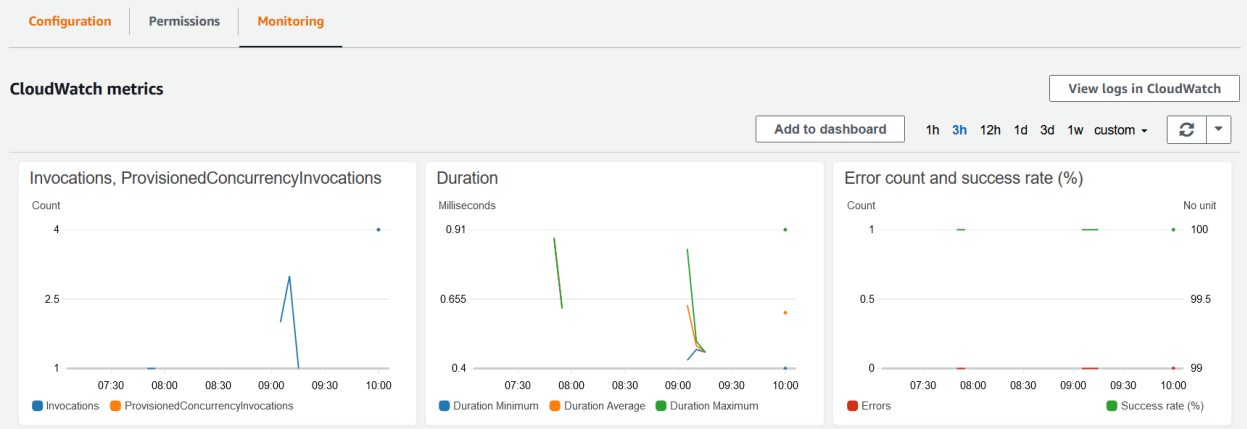
Gambar 495 Stages Menu

Klik ikon ***staging*** maka anda akan melihat ***list HTTP Method*** yang tersedia, dan di menu sebelah kanan klik **Tab *Logs/Tracing***. Pada kolom ***CloudWatch Settings***, ***check list Enable CloudWatch Logs***.



Gambar 496 Enable CloudWatch Logs

Jika sudah klik tombol ***Save Changes*** di pojok kanan bawah. Setelahnya pada ***dashboard AWS Lambda*** anda dapat memonitor ***AWS lambda*** melalui Tab ***Monitoring***, seperti pada gambar di bawah ini :



Gambar 497 CloudWatch Metrics for Lambda

Subchapter 6 – API Gateway & Lambda

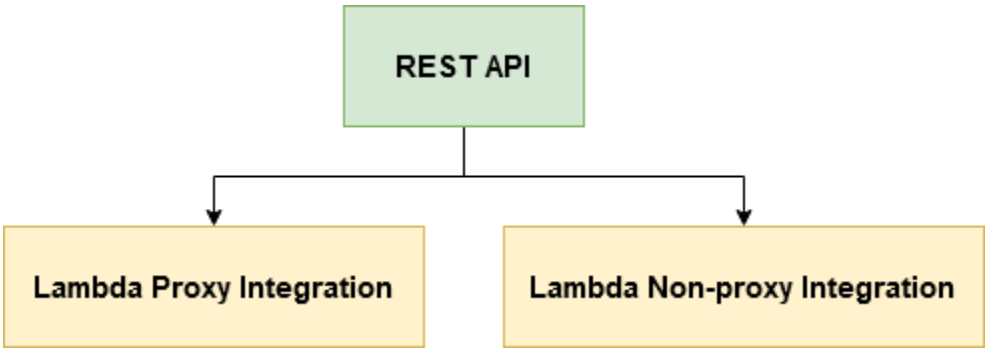
Be stubborn on vision but flexible on details.

— Jeff Bezos

Subchapter 6 – Objectives

- Mengenal *Amazon API Gateway*
- Mengenal *API Developer*
- Mengenal *App Developer*
- Mengenal *Features API Gateway*
- Belajar Membuat ***REST API***

Dengan *Amazon API Gateway* kita dapat membuat sebuah *REST API* menggunakan *Lambda Proxy Integration* dan *Lambda Non-proxy Integration*.



Gambar 498 Lambda Proxy & Non-Proxy Integration

Pada *Lambda proxy integration*, seluruh *client request* terkirim menuju *backend* yaitu sebuah *Lambda function*. *API Gateway* akan melakukan pemetaan seluruh *client request* langsung kedalam *event parameter* dalam *Lambda function*. *Ouput* yang dihasilkan dari *Lambda function's* termasuk *status code, headers,* dan *HTTP Body*, akan dikembalikan lagi kepada klien.

Pada *Lambda non-proxy integration* (disebut juga dengan "*custom integration*"), kita akan mengkonfigurasi *parameters, HTTP Headers*, dan *HTTP Body* yang diminta oleh klien ke

dalam format yang dibutuhkan oleh *backend* (*lambda function*). Kita juga akan mengkonfigurasi keluaran dari *lambda function* sesuai dengan format yang dibutuhkan oleh klien.

Chapter 7

Big Data

Subchapter 1 – *Introduction to Database*

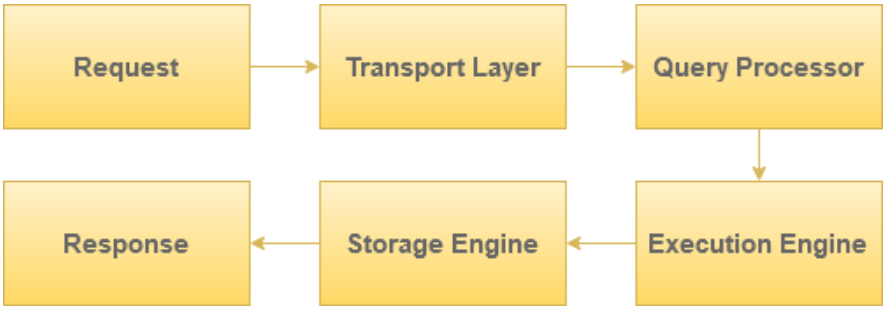
The old question 'Is it in the database?',
will be replaced by 'Is it on the blockchain?.

— William Mougayar

Subchapter* 1 – *Objectives

- Memahami Apa itu *Database-management System (DBMS)*
- Memahami Apa itu *Database*
- Memahami Apa itu *Storage Engine*
- Memahami Fungsi *Database*
- Memahami *Use Case Database*

Pertama kali sistem *database* muncul di awal tahun 1960 untuk memanajemen data komersil menggunakan komputer. ***Database-management system (DBMS)*** adalah sekumpulan data dan sekumpulan program untuk mengelola data tersebut. Sekumpulan data yang dikelola disebut dengan ***Database***. Dalam buku ini penulis menegaskan terminologi *database* dan sistem *database* adalah sesuatu yang sama.



Gambar 499 Database Under The Hood

Database terdiri dari beberapa sistem *modular* seperti ***Transport Layer*** untuk untuk menerima *request*, sebuah ***Query Processor*** untuk menentukan hasil *query* yang efisien, sebuah ***Execution Engine*** yang melaksanakan operasi dan sebuah ***Storage Engine***.[28]

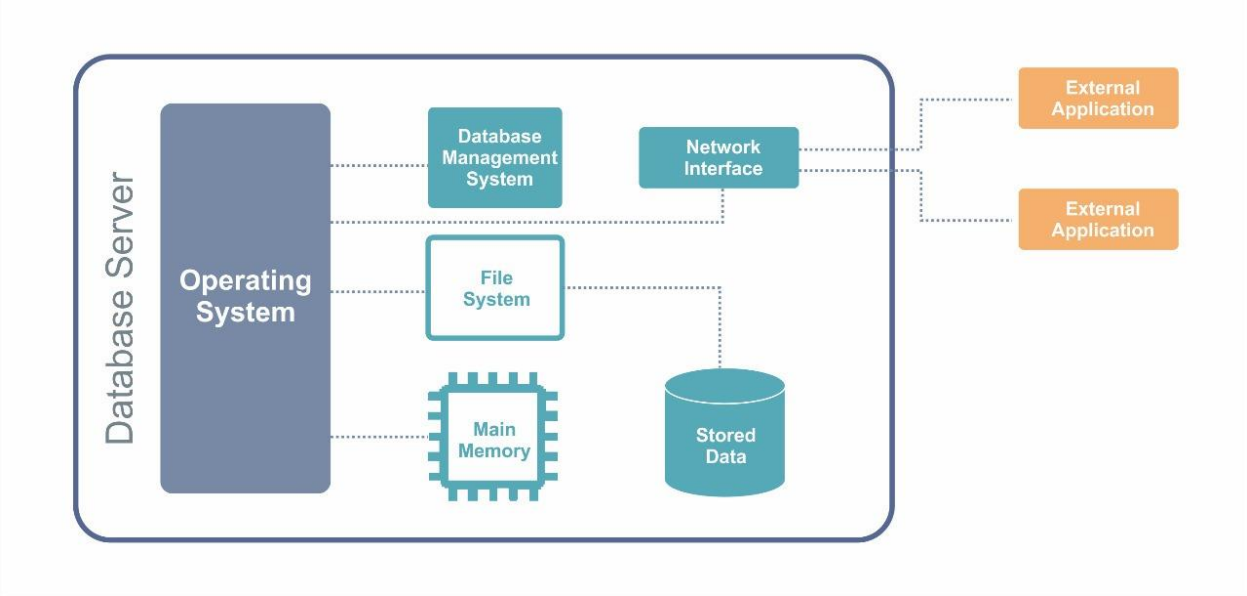
Storage Engine adalah salah satu komponen dalam *DBMS* yang bertugas untuk menyimpan, membaca, dan memanajemen data dalam memori & *disk*.

Salah satu *DBMS* yang populer seperti ***MySQL*** memiliki beberapa *storage engines* sekaligus, diantaranya adalah ***InnoDB***, ***MyISAM*** dan ***RockDB***. Pada *MongoDB*, kita dapat memilih menggunakan ***WiredTiger***, ***In-memory*** dan ***MMAPv1*** yang kini telah usang.

1. *Database Function*

Data Management

Sebuah sistem *database* digunakan untuk menyimpan, membaca, mencari dan memodifikasi data. Proses pengelolaan data tersebut digunakan untuk mendapatkan informasi. Sistem *database* harus memastikan bahwa informasi yang disimpan aman, meskipun sistem operasi mengalami *crash* atau terjadi percobaan akses data yang tidak diizinkan.



Gambar 500 Ekosistem *Database Management System*

Untuk membangun *interoperability* dengan aplikasi eksternal, sebuah sistem *database* harus menyediakan sebuah *inferface* atau API agar dapat diprogram.

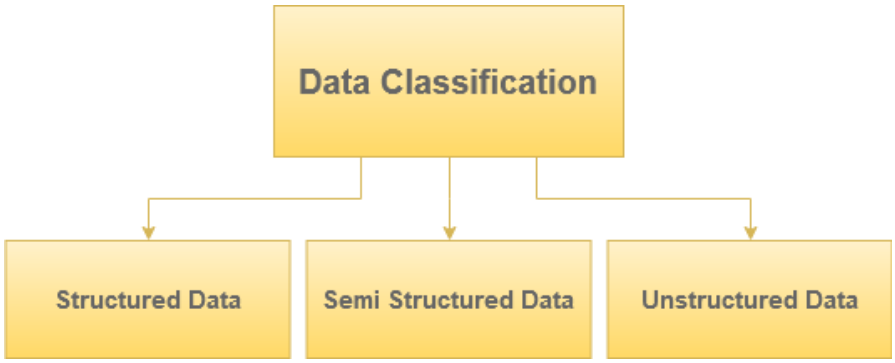
Sistem *database* juga harus mendukung ***Transaction***, sebuah *transaction* terdiri dari serangkaian operasi pada sebuah data di dalam suatu *database* yang tidak boleh diinterupsi. Jika salah satu operasi tidak terlaksana atau serangkaian operasi tidak sampai selesai, maka *trasanction* gagal.

Scalability

Sistem *database* harus bereaksi secara fleksibel dan beradaptasi dengan beban kerja yang lebih tinggi. Data dalam jumlah besar diproses dengan distribusi data dalam jaringan komputer yang berisi sekumpulan *database server* dengan tingkat paralelisasi (*parallelization*) yang tinggi.

Data Heterogenity

Terdapat 3 klasifikasi data yang dapat disimpan di dalam *database* :



Gambar 501 Data Classification

Structured Data

Sistem *database* harus dapat menyimpan data yang terstruktur, sebuah data yang harus memiliki skema. Skema data harus dipatuhi agar data yang disimpan adalah data dengan struktur data yang sesuai dengan skema data.

Semi-Structured Data

Data yang memiliki struktur fleksible disebut dengan *semi-structured data*.

Unstructured Data

Data yang tidak memiliki struktur (*arbitrary data*)

Eficiency

Sistem *database* yang cepat dalam memberikan respon untuk setiap permintaan.

Persistence

Sistem *database* harus dapat memberikan layanan penyimpanan untuk jangka panjang (*long-term*). Pada kasus *stream processing*, fungsi *persistence* bersifat selektif hanya *output* yang telah direkayasa akan disimpan untuk jangka panjang.

Reliability

Sistem *database* harus mencegah insiden kehilangan data dan integritas data. Salinan data disimpan pada server lain, penyimpanan secara redundan atau mekanisme seperti *replication*, agar bisa melakukan *data recovery* jika terjai kegagalan dalam *database server*.

Consistency

Sistem *database* harus memastikan tidak ada data yang salah atau kontradiktif dalam sistem. Verifikasi dilakukan secara otomatis untuk mencegah kendala konsistensi (*constraints*) dengan memanfaatkan kunci primer (*primary key*) atau (*foreign key*) dan update otomatis terhadap salinan data baru secara terdistribusi (*replica*).

Non-redundancy

Physical-redundancy menentukan kehandalan (*reliability*) sistem *database,* namun tidak dengan *logical-redundancy*. Jika terjadi *logical-redundancy* sekumpulan data akan tersimpan secara duplikat menghabiskan ruang untuk menyimpan data. Sekumpulan data yang mengalami *logical-redundancy* sangat rentan membentuk anomali yang dapat menimbulkan kesalahan dan data yang tidak konsisten. Normalisasi adalah salah satu cara untuk mengubah sekumpulan data kedalam format yang tidak redundan.

2. *Use Case Database*

Kita dapat menggunakan *database* untuk keperluan *enterprise* seperti membangun :

Aplikasi Penjualan (Sales)

Kelola *customer*, produk, dan informasi pembelian.

Aplikasi Accounting

Untuk pembayaran, kwitansi, saldo, aset dan informasi akunting lainya.

Aplikasi HR (Human Resources)

Untuk informasi mengenai pegawai, gaji dan pajak.

Kita juga dapat menggunakan *database* untuk dunia manufaktur seperti :

Aplikasi Manufaktur

Untuk manajemen *supply chain* dan melacak produksi barang di pabrik, persediaan barang di gudang dan toko, dan pesanan untuk barang.

Kita juga dapat menggunakan *database* untuk dunia perbankan & keuangan seperti :

Aplikasi e-Banking

Untuk mengelola informasi *customer*, akun, pinjaman, transaksi perbankan, pembelian secara kredit hingga ke mutasi.

Aplikasi Keuangan

Untuk menyimpan informasi seperti kepemilikan, penjualan, dan pembelian instrument keuangan seperti saham dan obligasi.

3. *Data Analytic*

Sebuah database juga digunakan untuk keperluan ***Data Analytic***, data diproses untuk menghasilkan sebuah kesimpulan, menyimpulkan aturan, prosedur keputusan dan informasi lainya yang dapat membantu menentukan *decision* yang efisien.

Contoh, ketika sebuah bank perlu memutuskan untuk memberikan pinjaman atau tidak sama sekali kepada calon peminjam dengan cara membaca data dari calon si peminjam.

Contoh lainya adalah perusahaan *facebook* yang harus menentukan suatu iklan sesuai dengan *target audience* yang diinginkan oleh si pengiklan. Untuk melakukan hal ini tehnik *Data Analysis* digunakan untuk menemukan aturan dan *pattern* dari data dan membuat ***predictive model***.

Model tersebut akan menerima *input* sebuah *attribute* atau sering kali disebut dengan *features* dan memproduksi sebuah prediksi yang dapat digunakan untuk menentukan keputusan.

Subchapter 2 – AWS Database

You have to be willing to be misunderstood

if you're going to innovate.

— Jeff Bezos

Subchapter* 2 – *Objectives

- Mengenal *Managed Relational Database*
- Mengenal *Nonrelational Database*
- Mengenal *Data Warehouse Database*
- Mengenal *In-memory Database*
- Mengenal *Time-series Database*
- Mengenal *Ledger Database*
- Mengenal *Graph Database*
- Mengenal *Database Migration Service*

AWS telah menyediakan berbagai teknologi *database* untuk berbagai keperluan seperti :

1. *Managed Relational Database*

Pada *managed relational database, AWS* menyediakan *Amazon RDS* dan *Aurora*.
Layanan ini sangat cocok jika kita ingin membangun *Transactional Application* seperti ERP (*Enterprise Resource Planning*), SCM (*Supply Chain Management*), CRM (*Customer Relationship Management*) dan *ecommerce* untuk mencatat transaksi dan data yang terstruktur. :

Amazon Relational Database Service (RDS)

Sebuah layanan yang menyediakan *relational database* untuk *MySQL, PostgreSQL, Oracle, SQL Server,* dan *MariaDB*.

Amazon Aurora

Sebuah layanan yang menyediakan *relational database* yang *compatible* dengan *MySQL* dan *PostgreSQL*.

2. *Nonrelational Database*

Pada *nonrelational database AWS* menyediakan *Amazon DynamoDB* dan *DocumentDB*.
Layanan ini sangat cocok jika kita ingin membangun *internet-scale applications* seperti *ride sharing, dating, hospitality,* menyediakan konten, menyimpan data terstruktur dan data tidak terstruktur.

Amazon DynamoDB

Sebuah layanan yang menyediakan ***Managed NoSQL Database*** yang secara konsisten memberikan *single-digit millisecond latency* sebesar apapun skala yang dibutuhkan.

Amazon DocumentDB

Sebuah layanan yang menyediakan *Managed Document Database* yang *compatible* dengan *MongoDB*.

3. *Data Warehouse Database*

Pada *Data Warehouse Database AWS* menyediakan *Amazon Redshift*.
Layanan ini sangat cocok jika kita ingin membangun *analytic application* untuk membuat sebuah *report* dan melakukan operasi *querying* dalam skala *terabyte* hingga ke *exabyte*.

Amazon Redshift

Pada *Data Warehouse Database, AWS* menyediakan ***Amazon Redshift*** agar kita bisa membangun sebuah *dataware house* sampai skala ***petabyte-scale***.

4. *In-memory Data store Database*

Pada *In-memory Data Store Database AWS* menyediakan *Amazon ElastiCache*.

Layanan ini sangat cocok untuk membangun *Realtime-Application* yang membutuhkan ***submillisecond latency***, membantu untuk pengembangan *online games, chat, tracking, streaming* dan *internet of things application*.

Amazon ElastiCache

Pada *In-memory Data Store Database, AWS* menyediakan ***Amazon ElastiCache*** untuk melakukan *deployment, operation,* dan *scaling in memory database* dengan *backbone* ***Memcached*** atau ***Redis*** dibelakangnya.

5. *Time-series Database*

Pada *Time-series Database AWS* menyediakan *Amazon TimeStream*.
Layanan ini sangat cocok untuk membangun aplikasi yang akan menyimpan jutaan data perdetik dalam format *time-series*.

Amazon TimeStream

Pada *Time-series Database, AWS* menyediakan ***Amazon TimeStream*** untuk membangun *managed time series database service* yang dapat mendukung operasi pada lingkungan *Internet of Things* (IOT) dan aplikasi yang akan menyimpan & menganalisa triliunan *events* setiap harinya.

6. *Ledger Database*

Pada *Ledger Database AWS* menyediakan *Amazon Quantum Ledger Database*.
Layanan ini sangat cocok untuk membangun aplikasi yang membutuhkan keaslian data dan sejarah transaksinya. Contoh melacak catatan transaksi *credit* dan *debit* dalam transaksi perbankan.

Amazon Quantum Ledger Database (QLDB)

Pada *Ledger Database, AWS* menyediakan ***Amazon Quantum Ledger Database*** untuk menyediakan sebuah *database* yang transparan, tidak dapat dimanipulasi (*immutable*) dan dapat diverifikasi menggunakan sebuah algoritma *cryptography*.

7. Graph Database

Amazon Neptune

Sebuah layanan yang menyediakan *Managed Graph Database* agar kita dapat menyimpan sebuah *data set* yang saling terhubung.

8. *Database Migration Service*

Amazon Database Migration Service (DMS)

Sebuah layanan yang membantu anda untuk melakukan migrasi *database* secara aman dan terjangkau untuk meminimalisir *downtime*.

Meskipun begitu kita tetap memiliki kebebasan untuk membangun *database* sendiri melalui *Amazon Elastic Compute Cloud (EC2)*. *Amazon Web Services (AWS)* juga menyediakan *AWS Database Migration Services (AWS DMS)*.

Subchapter 3 – Introduction to Big Data

You can have data without information?
but you cannot have information without data.

— Napoleon Bonaparte

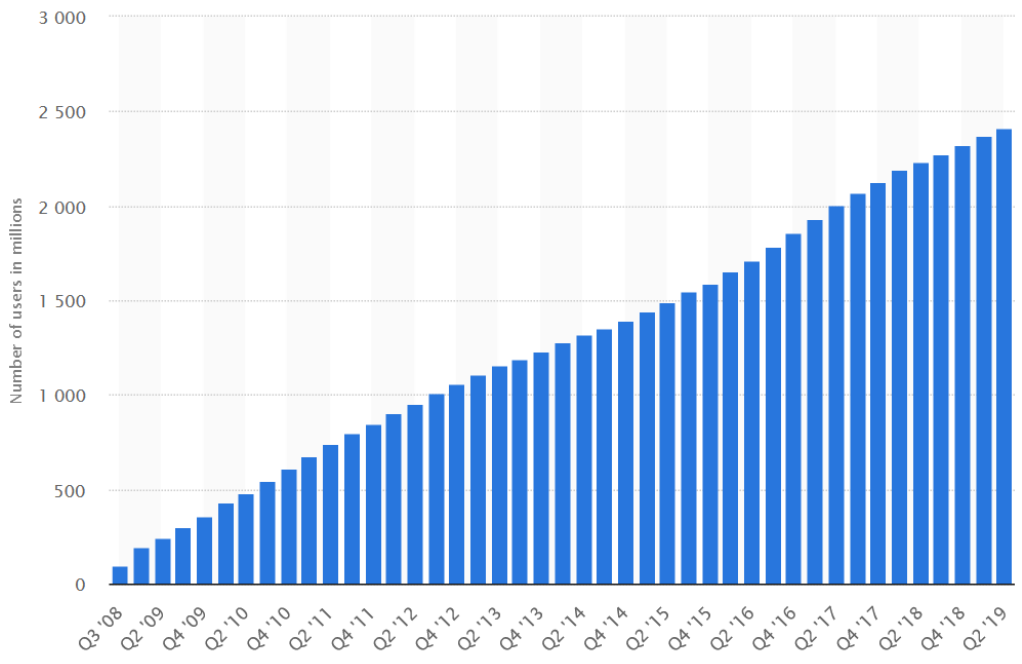
Subchapter 3 – Objectives

- Memahami Apa itu *Database & DBMS*
- Memahami Fungsi Database
- Memahami *Use Case Database*

Big Data adalah terminologi yang menjelaskan sebuah fenomena data dengan *volume* yang sangat besar (*High Volume Data*), data dengan struktur yang bervariasi (*High Varied Data*) dan data diproduksi dengan kecepatan yang sangat tinggi (*High Velocity Data*).

Fenomena *Big Data* menciptakan tantangan baru untuk sistem *database* tradisional seperti *RBMS* (*Relational Database Management System*).

Ketersediaan internet, kecepatan internet, *web application* dan produksi perangkat elektronik seperti *smartphone*, komputer, laptop dan *tablet* menciptakan ledakan data dengan pertumbuhan yang sangat cepat.



Gambar 502 Facebook Monthly Active User[29]

Dengan 2,4 milyar pengguna aktif pada kuarter kedua tahun 2019 *facebook* menjadi *platform* jejaring sosial (*social media*) terbesar didunia. Sebuah *platform social media*, dapat memproduksi berbagai jenis data seperti gambar (*image*), vidio (*video*), teks (*text*) dan suara (*voice*) dengan kecepatan yang sangat tinggi.

Subchapter 4 – Introduction to NoSQL

You can have data without information?
but you cannot have information without data.

— Napoleon Bonaparte

Subchapter 4 – Objectives

- Memahami Apa itu *NoSQL*
- Memahami Kelebihan *NoSQL*
- Memahami Klasifikasi *NoSQL Database*
- Memahami Relevansi *Big Data & NoSQL*

Selama hampir 50 tahun kita telah menggunakan konsep ***Relational Database*** untuk menyimpan data yang kita sebut dengan ***Structured Data***. Data dibagi menjadi sekumpulan grup yang kita sebut sebagai ***Table***, sebuah *table* memiliki ***Column*** dan ***Row***.

Terminologi *NoSQL* artinya kita dapat mengelola *database* tanpa menggunakan ***Structured Query Language*** (*SQL*). Menekankan bahwa *NoSQL* dapat memanajemen data tanpa harus mengikuti prosedur seperti yang ada di dalam konsep *RDBMS* (*Relational Database Management System*) seperti *fixed schema, table, column* dan *row*.

1. *CAP Theorem*

CAP Theorem menyatakan bahwa setiap sistem terdistribusi (*distributed system*) hanya dapat memiliki 2 *properties* dari 3 *properties* yang ada :

Consistency

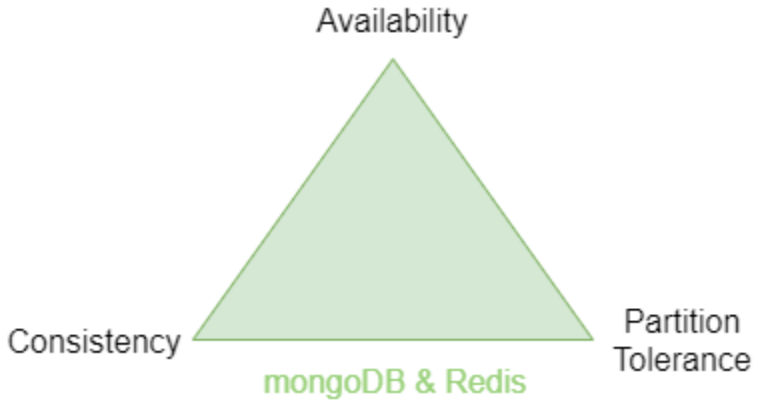
Setiap kali data dibaca adalah data yang terakhir ditulis.

Availability

Setiap kali membaca dan menulis data selalu berhasil.

Partition Tolerance

Sistem akan tetap berjalan meskipun terjadi kegagalan sistem atau kehilangan data.



Gambar 503 CAP Theorem

CAP Theorem terdiri dari 3 kategori :

1. *Consistency & Availability*
2. *Consistency & Partition Tolerance*
3. *Availability & Partition Tolerance*

2. *BASE Approach*

NoSQL Databases di dasari atas pendekatan *BASE* yaitu :

Basic Availability

Database harus tersedia (*available*) di setiap saat. Jaminan untuk mendapatkan respon dari setiap permintaan meskipun status data masih dalam keadaan *stale*.

Soft State

State inkonsisten secara sementara diizinkan.

Eventual Consistency

Sistem akan segera memasuki *state* konsisten setelah beberapa waktu tertentu.

Konsep *NoSQL* membawa gaya baru dalam memanajemen data lalu apa keunggulanya?

3. Keunggulan *NoSQL*?

Schemaless

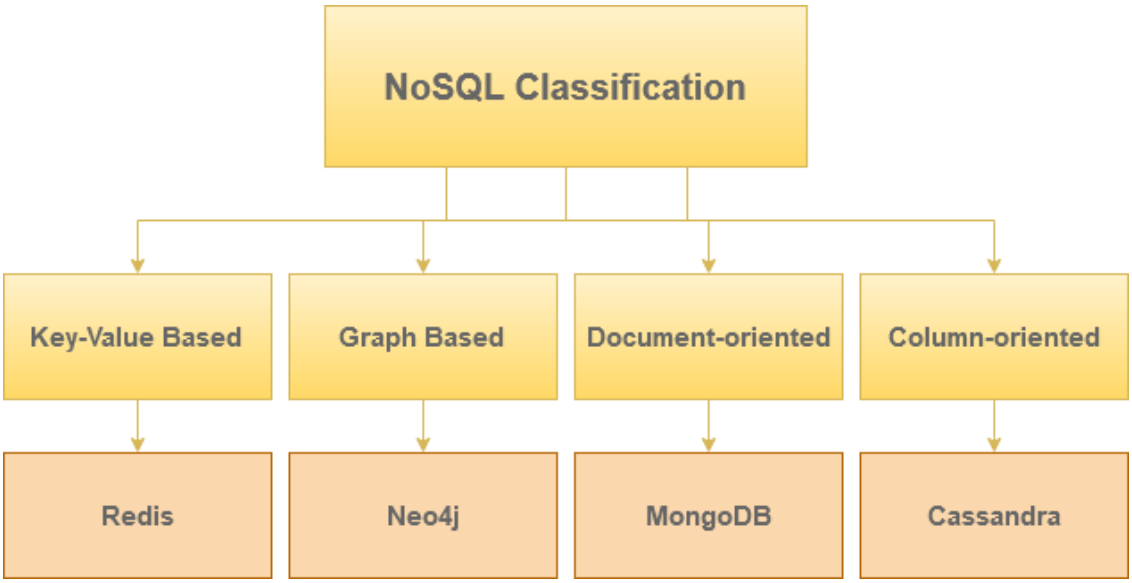
Representasi data yang akan disimpan pada *NoSQL Database* dapat disimpan tanpa perlu menentukan syarat struktur data terlebih dahulu, struktur data yang disimpan dapat bermacam-macam dan berubah-ubah.

Scalable

NoSQL database di desain agar bisa melakukan *scaling* lebih baik dari pada *relational database*.

4. Klasifikasi *NoSQL Database*

Terdapat beberapa klasifikasi *NoSQL Database* :



Gambar 504 NoSQL Classification

Key-value Store

Data disimpan dengan struktur data berbasis *keys* & *values*. Dianalogikan seperti *multidimensional array* yang bisa kita gunakan untuk mendapatkan suatu nilai (*values*) dengan cepat, melalui sebuah *keys*. Salah satu *NoSQL Database* yang menggunakan penyimpanan berbasis *Key/Value* adalah ***Redis***.

https://redis.io/

Di bawah ini adalah contoh bagaimana menyimpan data dan menampilkan data dalam *redis* :

```
redis 127.0.0.1:6379> SET author gungunfebrianza
OK
redis 127.0.0.1:6379> GET author
```

```
"gungunfebrianza"
```

Column-oriented

Data disimpan dengan struktur data berbasis kolom. Dianalogikan seperti sebuah *array* dengan sepasang *key/value*. Pada element ketiga terdapat sebuah ***timestamp***. Salah satu *NoSQL Database* yang menggunakan penyimpanan berbasis *column-oriented* adalah ***Cassandra***.

http://cassandra.apache.org/

Di bawah ini adalah contoh bagaimana menyimpan data dalam *apache cassandra* :

```
INSERT INTO writer id, name, city, phone) VALUES(1,'gungunfebrianza', 'Surabaya', 081313190101);
```

Graph

Data disimpan dengan struktur data visual. Data direpresentasikan dalam bentuk *nodes* yang terhubung melalui *edges*, sebuah garis yang menghubungkan antar *nodes*. Masing-masing *nodes* memiliki *properties* yang menjadi *metadata* dari *nodes* tersebut. Salah satu *NoSQL Database* yang menggunakan penyimpanan berbasis *nodes* dan *edges* adalah ***Neo4j***.

https://neo4j.com

Di bawah ini adalah contoh membuat *nodes* bernama ggf dengan label *programmer* dan *gamer*, pada kasus yang lebih rumit masing masing label dapat memiliki *properties* :

```
CREATE (ggf:programmer:gamer)
```

Document Oriented

Data disimpan dengan struktur data dalam bentuk ***object notation*** yang disebut dengan *document*. Sekumpulan *document* disimpan dalam sebuah *collection*. Setiap *document*

dapat memiliki *fields* yang berbeda-beda tanpa skema yang kaku atau *fixed*. Salah satu *NoSQL Database* yang menggunakan penyimpanan berbasis *document-style* adalah ***MongoDB***.

https://www.mongodb.com

5. *Big Data & NoSQL*

Salah satu *driver* yang menjadi kemunculan ***NoSQL*** adalah *RDBMS (Relational Database Management System)* sudah tidak lagi efisien untuk menangani ***Big Data*** yang memiliki karakteristik ***Semi-structured***, dan ***Unstructured data***.

Big Data memberikan tantangan yang sangat sulit untuk dihadapi *RDBMS*, *NoSQL* membawa paradigma baru dengan menegaskan bahwa konsep *RDBMS* sudah tidak lagi *reliable* untuk menangani *Big Data*.

Sebagai contoh pada *data-intensive processing* seperti ***Human Genome Project*** untuk analisis *strand* sebuah *DNA* memerlukan tempat penyimpanan data yang besar dan struktur data yang kompleks.

Chapter 8

Web Service

Web Service adalah sebuah layanan yang disediakan suatu perangkat elektronik agar dapat digunakan oleh perangkat elektronik lainya, komunikasi antar perangkat elektronik dilakukan melalui *world wide web* atau *Web service* lainya yang dikembangkan oleh *W3C*.

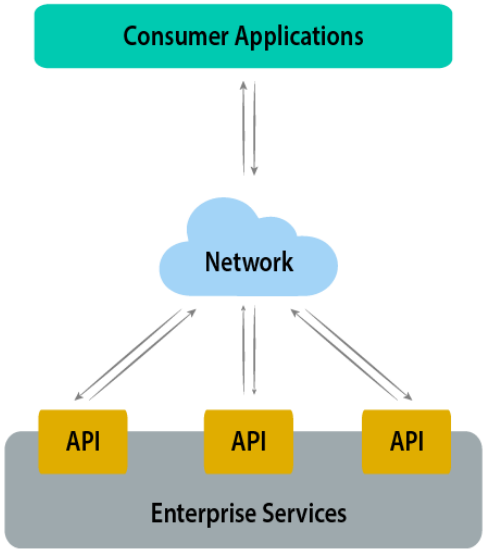
Melalui *Web Service* teknologi *web* seperti *HTTP* dapat digunakan sebagai alat komunikasi manusia kepada mesin (*human-to-machine communication*) agar dapat melakukan komunikasi antar mesin komputer (*machine-to-machine communication*). Pada *HTTP* komunikasi data yang dikirimkan dan diterima adalah data yang dapat dibaca antar mesin (*machine-readable file format*) seperti *XML* & *JSON*.

Subchapter 1 – API

Application Programming Interface (API) adalah sebuah *interface* yang memiliki spesifikasi bagaimana sebuah *software* dapat berinteraksi.

Terdapat sekumpulan *method* yang dapat kita gunakan untuk mengeksplorasi berbagai layanan melalui fitur yang disediakan pada sebuah *software*.

Pada gambar di bawah ini *consumer* dapat menggunakan API untuk berinteraksi dengan layanan yang diberikan suatu perusahaan.



Gambar 505 API Illustration

Sebagai contoh pada ***Google maps*** terdapat *API* yang menyediakan *directions* agar kita bisa mengetahui rute sebuah perjalanan.

Subchapter 2 – Remote Procedure Call

RPC atau ***Remote Procedure Call (RPC)*** adalah salah satu kajian dalam sistem terdistribusi (***Distributed Computing***). *Remote Procedure Call (RPC*) terjadi ketika sebuah program komputer mengirimkan permintaan pada komputer lainya di dalam suatu jaringan komputer untuk mendapatkan suatu layanan.

Permintaan yang dimaksud diajukan pada suatu program komputer untuk mengeksekusi *procedure* (*subroutine*) atau *function* di dalam komputer tersebut secara *remote*.

1. JSON-RPC

JSON-RPC adalah salah satu *RPC* yang menggunakan *JSON* untuk bertukar pesan. *JSON-RPC* mendukung ***Notification***, mengirimkan pesan menuju *server* tanpa perlu menerima respon. Saat ini *JSON-RPC* masih digunakan, salah satunya diimplementasikan sebagai *interface* dalam ***Bitcoin Core Software*** yang dapat digunakan memanajemen dompet lokal (*wallet*) untuk bertransaksi *bitcoin*.

Dalam *JSON-RPC*, *request* adalah usaha untuk memanggil (*invoke*) sebuah *method* yang disediakan oleh *remote system*. Untuk melakukanya terdapat 3 *Properties* yang harus disediakan :

1. ***method***

   Sebuah *string* yang menjadi nama *method* yang ingin kita *invoke*.

2. ***params***

   sebuah *object* atau *array* yang akan digunakan oleh *method*.

3. ***id***

   Sebuah *String* atau *Number* yang digunakan sebagai identitas untuk mengetahui *response* yang dihasilkan dari sebuah *request*.

Dalam *JSON-RPC*, *response* adalah hasil yang didapatkan setelah melakukan sebuah *request*. Sebuah *response* harus memiliki beberapa properties di bawah ini :

1. result

   Data yang dihasilkan dari *method* yang telah di *invoke*. Jika terjadi error saat melakukan *invoke* maka nilai yang dihasilkan harus null.

2. error

   Error kode diperlukan jika terjadi *error* saat melakukan *invoke* sebuah *method*.

3. id

   id yang menjadi identitas untuk mengetahui hasil dari sebuah *request*.

Di bawah ini adalah contoh *Remote Procedure Call (RPC)* pada **api.calculator.com**, penulis sebagai *client* ingin mengeksekusi *method subtract* lengkap dengan *parameter* yang berada dalam suatu *remote system* :

```
POST /subtract HTTP/1.1
HOST: api.calculator.com
Content-Type: application/json
{"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42,
"subtrahend": 23}, "id": 3}
```

Maka di dalam *remote system*, kode di bawah ini akan dieksekusi :

```
function subtract(minuend, subtrahend) {
  // 42-23 = 19
}

subtract(minuend, subtrahend);
```

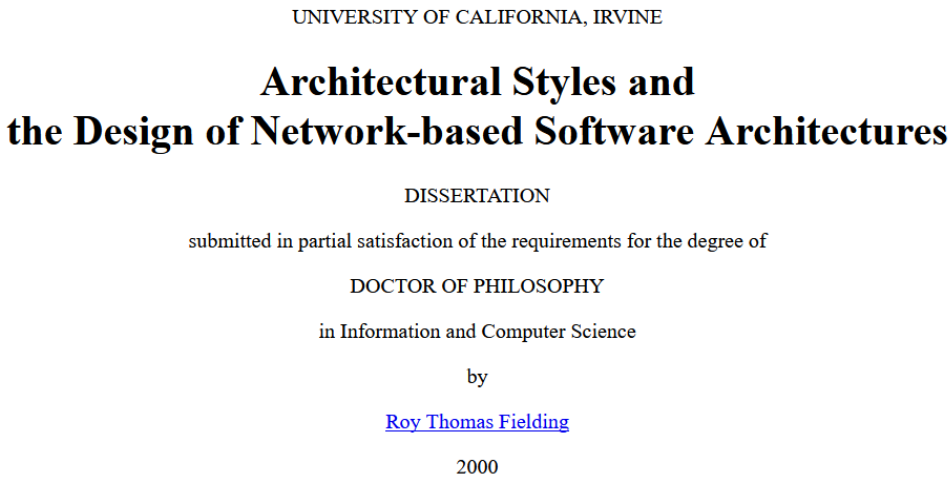
Setelah berhasil dieksekusi respon akan diterima oleh penulis dengan format :

```
{"jsonrpc": "2.0", "result": 19, "id": 3}
```

Sebelum ***REST*** menjadi populer, sebagian besar *API* dibangun menggunakan *XML-RPC* atau *SOAP*. *RPC-based API* mendominasi *landscape*.

Subchapter 3 – REST

REST adalah singkatan dari ***Representational State Transfer***, pertama kali dijelaskan oleh seorang *computer scientist* bernama **Roy Fielding** dalam disertasinya yang berjudul ***Architectural Styles and the Design of Network-based Software Architectures***.



UNIVERSITY OF CALIFORNIA, IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

2000

Gambar 506 Roy Fielding Dissertation

Jika ingin membaca penelitianya anda bisa mendapatkanya disini :

https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

RESTful Web Service

REST adalah sebuah gaya dalam ***Software Architecture*** yang dapat digunakan untuk membangun sebuah *web service*. Sebuah *web service* yang menggunakan REST disebut dengan ***RESTful Web Service (RWS)***.

Ada 6 ***Constraint*** untuk memastikan bahwa sebuah *web service* telah mendukung *RESTful system*. *Constraint* yang dimaksud adalah batasan yang dimiliki oleh *server* dalam memproses dan merespon permintaan dari *client*, jika operasi dilakukan di dalam batasan

maka sistem mampu memperoleh sifat-sifat yang di inginkan seperti *performance, scalability, simplicity, modifiability, visibility, portability,* dan *reliability*.

Constraint yang dimaksud adalah :

Uniform Interface

Uniform Interface menjadi kunci dalam *constraint* yang menjadi pembeda antara ***REST API*** dan ***Non-REST API***. *Uniform Interface* mengusung interaksi yang harus dapat diterima secara universal antara *server* dan perangkat atau aplikasi yang terhubung, baik itu *web application* atau *mobile application*. Ada 4 Prinsip dalam ***Uniform Interface*** :

Resource-Based

Masing-masing *resources* dapat diidentifikasi melalui *HTTP Request*.

Contoh: *API/users*.

Manipulation of Resouces

Manipulasi *resources* dilakukan melalui representasi *resources* yang dimiliki oleh *client* dan tersedia informasi yang cukup untuk memodifikasi atau menghapus suatu *resource* di dalam *server*.

Contoh: Pengguna dapat melakukan aksi *HTTP Request* untuk mendapatkan user ID, lalu menggunakanya misal untuk mendapatkan informasi detail lainya yang dimiliki oleh *user* ID tersebut dan memodifikasi atau menghapus *user* ID tersebut.

Self-descriptive Messages

Setiap pesan mengandung informasi yang cukup untuk menjelaskan bagaimana pesan harus diproses yang dapat memudahkan *server* untuk melayani permintaan.

HATEOAS

Hypermedia as the Engine of Application State (HATEOAS), sebelumnya kita telah mempelajari apa itu *hypermedia*. Maksud dari *HATEOAS* adalah setiap *links* harus disertakan di dalam respon agar *client* dapat menemukan *resources* lainya dengan mudah.

Client-Server Architecture

Client & Server prinsip dasar yang digunakan disini adalah ***Separation of Concern (SoC)*****.** *Client application* dan *server application* harus dapat dikembangkan secara independen, terpisah. Untuk dapat terhubung dengan *server, client* hanya membutuhkan *Uniform Resources Identifier (URIs)*. Perubahan dalam *client* atau *server* tidak boleh memberikan impak negatif pada salah satunya.

Stateless

Roy Fielding terinspirasi dari *HTTP* yang bersifat ***Stateless***, interaksi antara *client application* dan *server application* bersifat *stateless*. Maksud dari *stateless* adalah *server* tidak akan menyimpan informasi *HTTP Request* yang dilakukan oleh client. Setiap kali ada *HTTP Request* baru akan diperlakukan sebagai *request* yang baru tanpa ada *session* dan *history*.

Terkecuali jika *client application* menginginkan layanan *stateful application*, dimana pengguna dapat melakukan ***log in*** dan mendapatkan ***authorization*** untuk melakukan operasi misal mengubah *database*. Maka setiap kali terdapat *HTTP Request* dari *client* harus memiliki seluruh informasi yang dibutuhkan oleh *server application*.

Cacheable

Caching memberikan peningkatan kemampuan pada *performance* yang dapat dimiliki oleh *client* dan *server* untuk mereduksi beban permintaan. Di dalam *REST*, *caching* harus diterapkan pada *resources* ketika memang bisa diterapkan dan *resouces* tersebut harus mendeklarasikan bahwa *resources* tersebut dapat di *caching*. Manajemen *caching* yang baik dapat membantu ketergantungan interaksi antara *client* dan *server* secara parsial atau penuh.

Layered System

REST mendukung penggunaan sistem arsitektur yang menggunakan layer. Maksudnya secara praktis anda dapat membangun sebuah API pada *server* X, menyimpan data pada *server* Y dan permintaan izin ***authentication*** dilakukan di *server* Z.

Code on demand

Fitur ini bersifat opsional, fitur ini diusung agar kode dapat dikirimkan melalui API agar dapat digunakan oleh *application server*. Sehingga *application server* tidak lagi tergantung pada struktur kode yang dimilikinya. Fitur ini memiliki keunggulan untuk membangun ***smart application*** namun juga membangkitkan ***security concern***.

Daftar Pustaka

[1] Hegaret, Philippe "100 Specifications for the Open Web Platform and Counting," w3.org, 2011 [Online]. Tersedia : https://www.w3.org/blog/2011/01/100-specifications-for-the-ope/ [Diakses : 8 Maret 2020]

[2] https://webassembly.org [Diakses : 11 Maret 2020]

[3] Zakas, Nicholas, *Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers*, No Starch Press USA, 2006.

[4] Vanessa, Frank, & Peter, *The Definitive Guide to HTML5 WebSocket*, : Apress, 2003.

[5] Salvatore & Simon, *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*, USA : O'reilly media, (2014).

[6] Parisi, Toni, *WebGL: Up and Running: Building 3D Graphics for the Web*, USA : O'reilly media, (2012).

[7] https://www.zygotebody.com/ [Diakses : 11 Maret 2020]

[8] Connalen, Jim, *Building Web Applications with UML*, USA : Addison-Wesley, 2002.

[9] Price, Ron, *CompTIA Server+ Certification Guide*, UK : Packt Publishing, (2019).

[10] *Virtualization Security – EC-Council*, Cengage Learning, 2011.

[11] James, Jim & Ravi, *Virtual Machines: Versatile Platforms for Systems and Processes*, USA : Elsevier, 2005.

[12] Dinkar & Geetha, *Moving To The Cloud: Developing Apps in the New World of Cloud Computing*, USA : Elsevier, 2012.

[13] Nancy, & Robert, *Web Server Technology*, California : Morgan Kaufmann Publisher, 1996.

[14] Heckmann, Oliver, *The Competitive Internet Service Provider*, USA : John Wiley, 2016.

[15] *Global Geographies of the Internet Barney Warf*, USA : Springer, 2013.

[16] https://futurism.com/the-byte/amazon-launch-3200-internet-satellites [Diakses : 6 Agustus 2019]

[17] Ping, Peng, *World Internet Development Report 2017*

[18] Wang, Ranjan, Chen & Benatallah, *Cloud Computing: Methodology, Systems, and Applications*, USA : CRC Press, 2012.

[19]

[20] Mueler, Millton, *Ruling the Root: Internet Governance and the Taming of Cyberspace*, USA : MIT, 2002.

[21] http://primaryfacts.com/5484/enigma-machine-facts-and-information/ [Diakses : 11 Maret 2020]

[22] Jacobs, Stuart, *Engineering Information Security: The Application of Systems Engineering Concept to Accept Information Assurance*, USA : Wiley, 2011.

[23] Hawker, Andrew, *Security and Control in Information Systems: A Guide for Business and Accounting*, USA : Routledge, 2002.

[24] Oppliger, Rolf, *SSL and TLS Theory and Practice*, USA, 2016.

[25] https://floating-point-gui.de/formats/binary/ [Diakses : 11 Maret 2020]

[26] Hunt, John, *Java and Object Orientation: An Introduction*, UK : Springer, 2002.

[27] Flanagan, David Flanagan, *Javascript – The Definitive Guide karya*, 2002

[28] Alex, Petrov, *Database Internals : A Deep Dive into How Distributed Data Systems Work*, USA : O'Reilly Media, 2019.

[29] https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/ [Diakses : 21 Agustus 2019]

[30] https://bandung.kompas.com/read/2019/04/25/12402021/ridwan-kamil-pastikan-amazon-web-service-investasi-di-jawa-barat [Diakses : 11 Maret 2020]

[59]

Tentang Penulis

Penulis adalah Mahasiswa lulusan Universitas Komputer Indonesia (UNIKOM Bandung). Semenjak masuk ke bangku kuliah sudah memiliki *habit* untuk membuat karya tulis di bidang pemrograman, *habit* ini mengantarkan penulis untuk membangun skripsi di sektor ***Compiler Construction*** dengan judul "Kompiler untuk pemrograman dalam Bahasa Indonesia".

Membuat bahasa pemrograman berbahasa Indonesia, berbasis *object-oriented programming*, membuat Kompiler untuk bahasa pemrograman berbahasa indonesia dan IDE Kompiler untuk memproduksi *executable* (***exe***) dan *dynamic link library* (***dll***).

Penulis juga seorang *Founder* sekaligus *Chief Technology Officer (CTO) Market Koin Indonesia*, sebuah *platform Trading Engine* tempat masyarakat dapat membeli dan menjual *bitcoins, ethereum* dan *alternative cryptocurrency* lainnya. Dari tahun 2017 penulis sudah mendapatkan investasi dan pembiayaan *equity financing* dari pengusaha-pengusaha di Eropa untuk mengembangkan *platform* Market Koin & Blockchain.

Riset-riset yang sedang penulis kembangkan adalah teknologi *Cross-border Payment, Cryptocurrency Arbitrage System, High-Frequency Trading (HFT) Engine,* dan *platform* terbaru yang sedang dikembangkan adalah ***Lightning Bank***.

Sebuah teknologi yang penulis kembangkan untuk membantu perbankan di *Denmark* dan Indonesia agar bisa bertransaksi secara instant dan biaya transaksi di bawah 3% sesuai target *Sustainable Development Goals (SDG) United Nation*.

Penulis juga aktif dalam kegiatan literasi finansial untuk masyarakat dan pengembangan Industri Maritim Indonesia, tempat penulis membangun usaha di sektor Industri Udang.

Terimakasih untuk Ibu ku tercinta, Guru-guruku, Nikolaj Vestorp my lifetime partner dan sahabatku Chrysta Agung Winarno yang sudah ada sejak zaman poerba koepi doea riboe.


BELAJAR DENGAN JENIUS
GOLANG
THEORY + VISUALIZATION + CODE
GUN GUN FEBRIANZA


DEVELOP SECURITY SOFTWARE
WITH C#
C#
THEORY + VISUALIZATION + CODE
GUN GUN FEBRIANZA