

BELAJAR DENGAN JENIUS DENO



THEORY + VISUALIZATION + CODE

GUN GUN FEBRIANZA



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

Gun Gun Febrianza
Bandung, 03 Juni 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 mengabdikan 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 kedepannya 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.

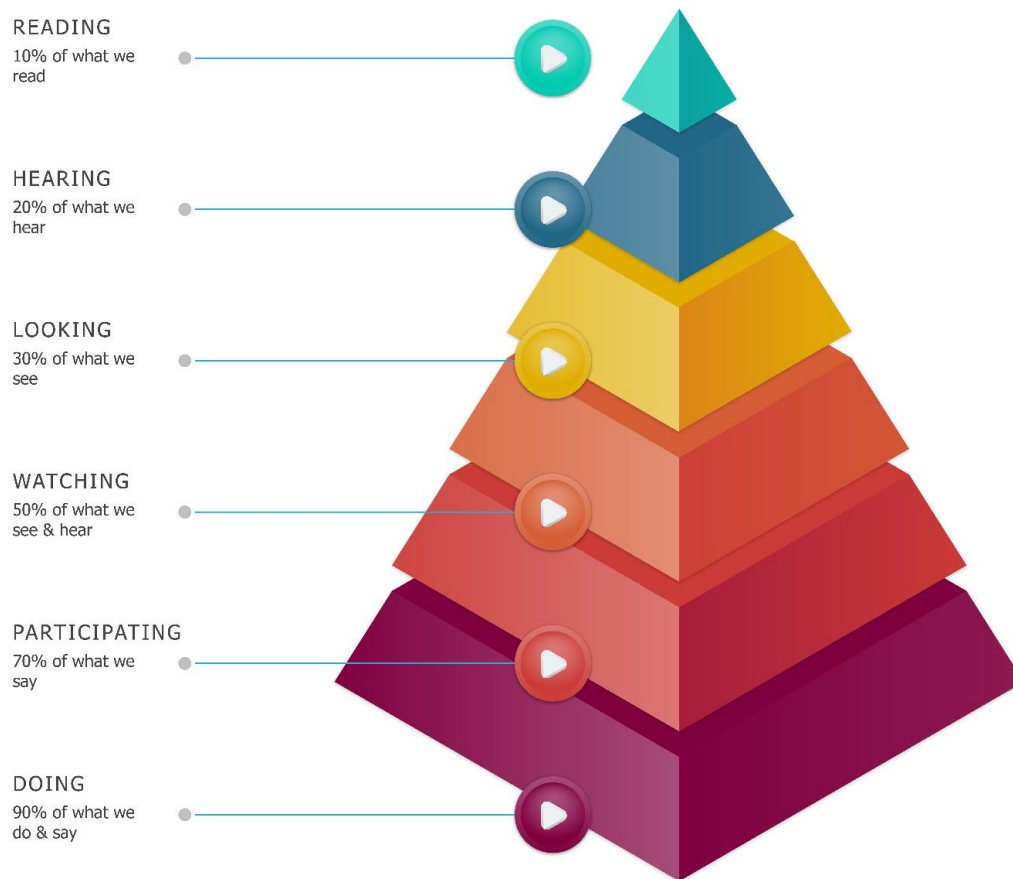
Dari Penulis, yang kelak bercita-cita ingin menjadi seorang guru.

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

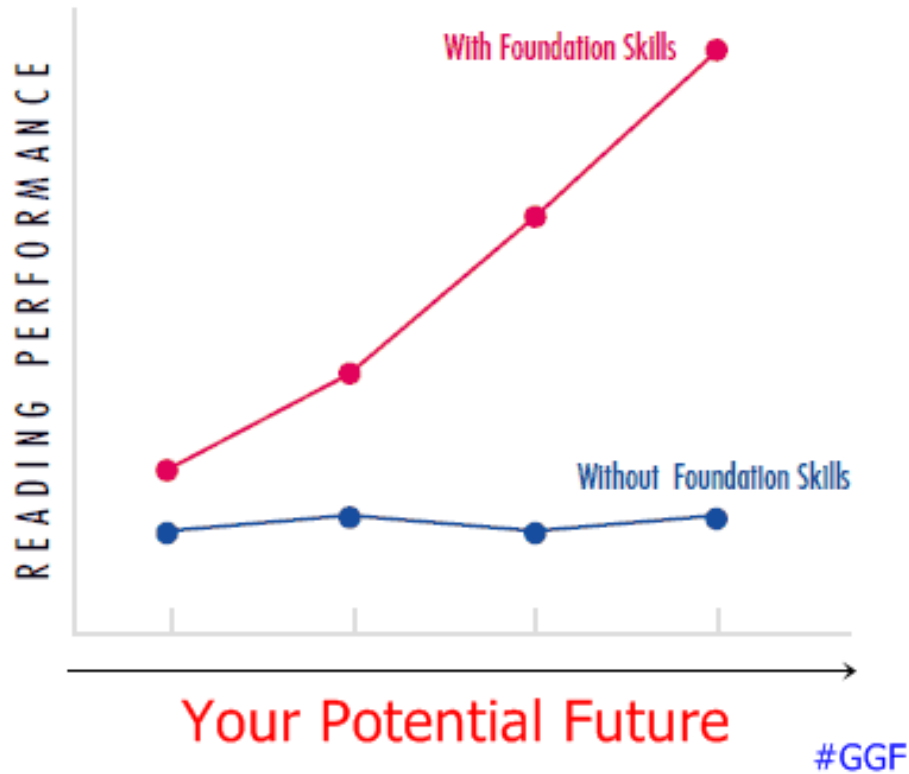
Saat belajar anda harus memahami konsep ***Abstraction Control***. Sebuah daya tahan yang harus anda miliki untuk tetap fokus pada hal-hal yang relevan dan penting untuk anda.

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/>

Downloads

Latest Current Version: 14.2.0 (includes npm 6.14.4)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer <small>node-v14.2.0-x64.msi</small>	 macOS Installer <small>node-v14.2.0.pkg</small>	 Source Code <small>node-v14.2.0.tar.gz</small>

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

Source Code

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
64-bit	
ARMv7	ARMv8
node-v14.2.0.tar.gz	

Install Visual Studio Code

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

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



↓ Windows

Windows 7, 8, 10



↓ .deb

Debian, Ubuntu

↓ .rpm

Red Hat, Fedora, SUSE



↓ Mac

macOS 10.9+

User Installer	64 bit	32 bit
System Installer	64 bit	32 bit
.zip	64 bit	32 bit

.deb	64 bit	32 bit
.rpm	64 bit	32 bit
.tar.gz	64 bit	32 bit

[Snap Store](#)

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left displays a project structure with folders like '1. Basic', '2. CRUD', '3. Querying In-Depth', '4. Query Operator', '5. Update Operator', 'node_modules', 'Training', and '.gitignore'. The '2. CRUD' folder is expanded, showing files like '1.Inserting-save.js', '2.Inserting-insertMany.js', '3.Querying-find.js', '4.Querying-find-criteria.js', '5.Updating-updateOne.js', '6.Updating-updateMany.js', '7.Updating-replaceOne.js', and '8.Deleting-deleteOne.js'. The main editor area shows the 'employees.js' file with JavaScript code. The code includes a REST API definition for a login endpoint using Express.js and Passport.js. The code is as follows:

```

73 }
74 } else {
75   const message = {
76     meta: {
77       status_code: '401',
78       message: 'You are not allowed'
79     }
80   };
81   res.status(401).json(message);
82 }
83 }
84 );
85
86 // @route POST api/employees/login
87 // @desc Login Employee
88 // @access Public
89 router.post('/login', (req, res) => {
90   //Input Validation
91   const { errors, isValid } = validateLoginInput(req.body);

```

The bottom of the screenshot shows the 'TERMINAL' tab with a list of files and their status, such as 'modules/user/mail/verify_email.js' with a status of '2 +-'.

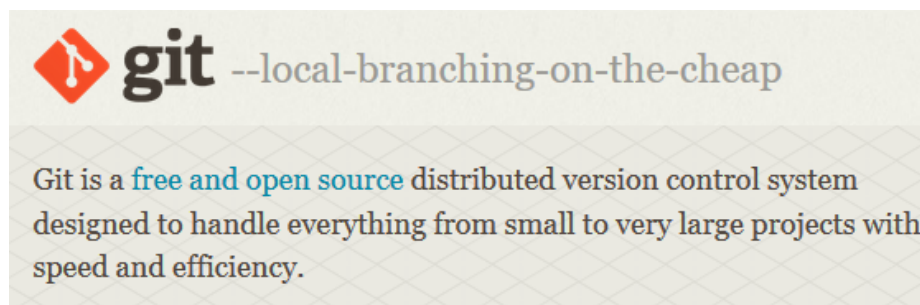
Install Google Chrome atau Firefox.

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 <i>file</i> , 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?

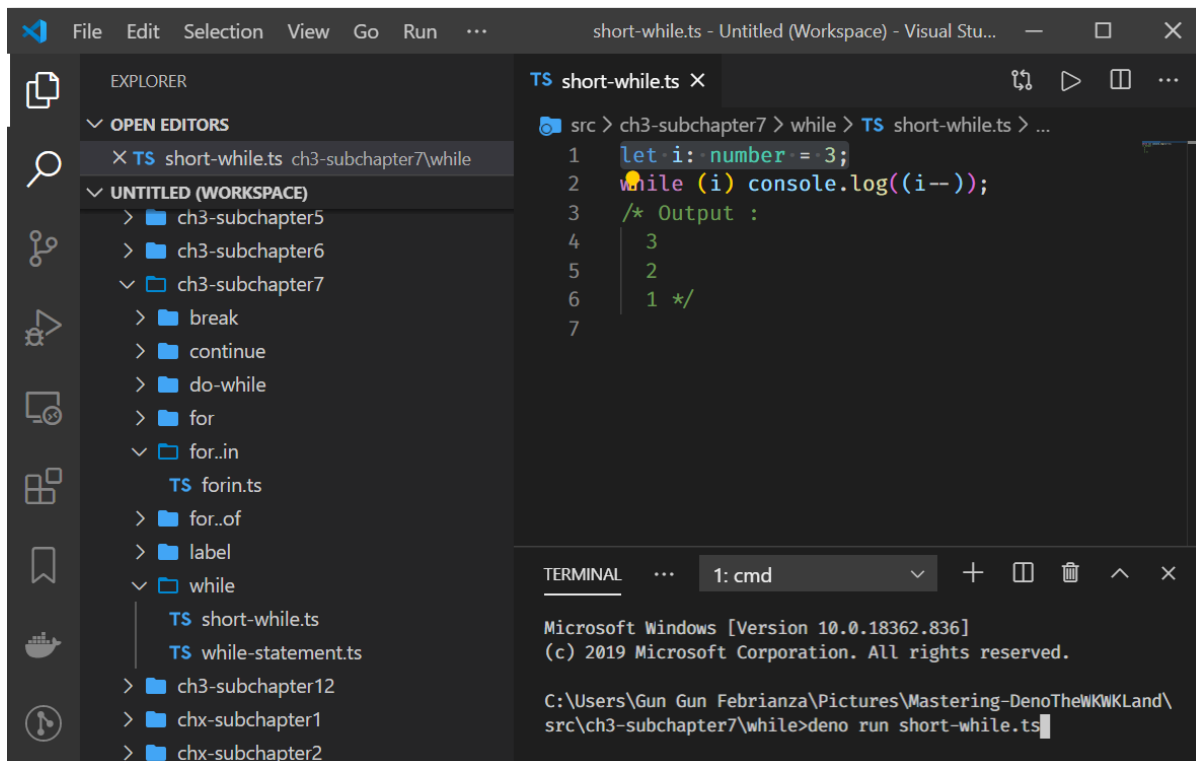
Jika ada *feedback* yang bersifat *private* silahkan diajukan melalui *email* saya gungunfebrianza@gmail.com.

Kode Sumber?

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

<https://github.com/gungunfebrianza/Mastering-DenoTheWKWKLand>

Penggunaan Kode ?



Pilih salah satu *file* yang ingin dieksekusi, misal `short-while.ts` kemudian buka *terminal*.

Eksekusi *file* dengan perintah `deno run [nama file tanpa ekstensi]` contoh :

```
deno run short-while.ts
```

Terdapat Kesalahan?

Silahkan ajukan isu :

<https://github.com/gungunfebrianza/Belajar-Dengan-Jenius-DenoTheWKWKLand>

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 lainnya. Bisa berdiskusi untuk berkenalan dengan developer lainnya 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

<i>Open Library Indonesia</i>	3
<i>Metode Belajar</i>	5
<i>Learning Problems & Abstraction Control</i>	8
<i>Matthew Effect in Reading</i>	9
<i>Persiapan</i>	10
Apa saja yang harus dipersiapkan?	10
<i>Install Node.js</i>	10
<i>Install Visual Studio Code</i>	10
<i>Install Python</i>	Error! Bookmark not defined.
<i>Install Google Chrome atau Firefox</i>	11
<i>Install git</i>	12
<i>Install Postman</i>	12
Konvensi Penulisan?	13
<i>Feedback?</i>	15
Kode Sumber?.....	15
Penggunaan Kode ?.....	15
Terdapat Kesalahan?	16
Pertanyaan, Kritik dan Saran?.....	16

<i>Table of Contents</i>	17
<i>Chapter 1</i>	52
<i>Belajar Open Web Platform</i>	52
<i>Subchapter 1 – Apa itu Open Web Platform</i>	52
1. <i>Technical Specification</i>	54
2. <i>HTML 5.2</i>	54
<i>Semantic Advantage</i>	55
<i>Connectivity Advantage</i>	55
<i>Storage Advantage</i>	55
<i>Multimedia Advantage</i>	55
<i>Performance Advantage</i>	56
<i>Device Access Advantage</i>	56
<i>Specification</i>	56
3. <i>Web Assembly</i>	58
<i>Safe</i>	59
<i>Fast</i>	59
<i>Portable Code</i>	59
<i>Compact Code</i>	59
<i>Specification</i>	60
4. <i>EcmaScript</i>	61
<i>Specification</i>	62

5. Web Socket	63
<i>Specification</i>	63
<i>Application</i>	63
6. WebRTC	65
<i>Specification</i>	65
<i>Application</i>	65
7. WebGL	67
<i>Specification</i>	67
<i>Application</i>	67
Subchapter 2 – Apa itu Web Application?	69
1. Server	71
<i>File Server</i>	72
<i>Mail Server</i>	73
<i>Proxy Server</i>	73
<i>Application Server</i>	74
<i>Database Server</i>	74
<i>Messaging Server</i>	75
2. Virtual Private Server	77
<i>Virtualization</i>	77
<i>Virtual Machine</i>	80
<i>Hypervisor</i>	81

3. Web Server	82
4. Web Page	84
Static Web Page	84
Dynamic Web Page.....	84
Progressive Web Application (PWA).....	85
Single Page Application (SPA).....	86
5. Network	89
Local Area Network (LAN)	89
World Area Network (WAN)	91
Internet Service Provide (ISP).....	91
6. Internet.....	93
Internet Transit	93
Satellite & Fiber Optic	94
7. Internet Exchange Point	95
8. Content Delivery Network (CDN).....	97
9. Cloud Computing	98
Cloud Computing Execution Model	99
Cloud Service Provider	101
Scalability.....	102
Load Balancer	106
10. Serverless Computing	107

<i>FaaS Provider</i>	107
<i>AWS Lambda</i>	107
<i>Subchapter 3 – Bedah Konsep HTTP</i>	109
1. <i>HTTP & URL</i>	109
<i>HTTP</i>	109
<i>Hypertext & Hyperlink</i>	109
<i>Hypermedia</i>	110
<i>World Wide Web (www)</i>	111
<i>Uniform Resources Identifier (URI)</i>	111
<i>URL / Web Resources</i>	112
2. <i>HTTP & DNS</i>	115
<i>IP Address</i>	115
<i>DNS Resolver</i>	116
<i>Root Server & TLD Server</i>	117
3. <i>HTTP Transaction</i>	121
<i>TCP Three-way Handshake</i>	121
4. <i>HTTP Request</i>	124
<i>HTTP Method</i>	124
<i>Message</i>	127
<i>HTTP Header</i>	127
<i>Header Attribute</i>	128

<i>MIME</i>	129
5. <i>HTTP Response</i>	131
6. <i>HTTP Status Message</i>	133
<i>Subchapter 4 – Web Security</i>	136
1. <i>Data in The Low Level</i>	136
<i>Host</i>	136
<i>Socket</i>	136
<i>Bit</i>	138
<i>Byte</i>	138
<i>Bytes</i>	139
<i>Character</i>	139
<i>ASCII</i>	139
<i>Data Transmission</i>	141
<i>Base64 Encoding</i>	141
2. <i>Cryptography</i>	143
<i>Cryptanalysis</i>	144
<i>Information Security</i>	145
<i>Ciphertext</i>	145
<i>Symmetric Cryptography</i>	148
<i>Hash Function</i>	152
<i>Message Authentication Codes (MAC)</i>	153

<i>Assymmetric Cryptography</i>	156
<i>Cryptography Protocol</i>	158
3. <i>Man In The Middle (MITM) Attack</i>	159
<i>Eavesdropping</i>	159
4. <i>HTTPS</i>	162
<i>Perbedaan HTTP & HTTPS</i>	163
<i>Manfaat HTTPS</i>	164
5. <i>Secure Socket Layer (SSL)</i>	166
<i>Transport Socket Layer (TLS)</i>	166
<i>SSL Handshake</i>	167
<i>Chapter 2</i>	170
<i>Setup Learning Environment</i>	170
<i>Subchapter 1 – Visual Studio Code</i>	170
1. <i>Install Programming Language Support</i>	174
2. <i>Install Keybinding</i>	177
3. <i>Install & Change Theme Editor</i>	179
4. <i>The File Explorer</i>	181
5. <i>Search Feature</i>	183
6. <i>Source Control</i>	185
7. <i>Debugger</i>	186
8. <i>Extension</i>	186

<i>Deno</i>	187
<i>Auto Fold</i>	188
<i>Better Comment</i>	189
<i>Bookmarks</i>	189
<i>Javascript (ES6) Code Snippets</i>	192
<i>Path Intellisense</i>	193
<i>VSCode Great Icons</i>	193
9. <i>The Terminal</i>	195
<i>Menambah Terminal Baru</i>	195
<i>Melakukan Split Terminal</i>	196
<i>Mengubah Posisi Terminal</i>	196
<i>Menghapus Terminal</i>	197
10. <i>Performance Optimization</i>	198
11. <i>Zen Mode</i>	199
12. <i>Display Multiple File</i>	200
13. <i>Font Ligature</i>	201
<i>Subchapter 2 – Web Browser</i>	204
1. <i>Web Browser</i>	204
2. <i>WebConsole</i>	204
<i>Autocomplete</i>	205
<i>Syntax Highlighting</i>	206

<i>Execution History</i>	207
3. <i>Multiline Code Editor</i>	208
<i>Chapter 3</i>	209
<i>Mastering Deno</i>	209
<i>Subchapter 1 – Install Deno</i>	209
1. <i>Installation</i>	209
<i>Deno For Windows</i>	209
<i>Install Deno For Linux</i>	211
<i>Install Deno For MacOS</i>	213
<i>Check Deno Version</i>	213
<i>Subchapter 2 – Introduction to Deno</i>	214
1. <i>Why Deno?</i>	214
<i>Promise Dilemma</i>	214
<i>Security</i>	215
<i>The Build System (GYP)</i>	215
<i>Package.json</i>	216
<i>Node_Modules</i>	216
<i>Module & Extension</i>	217
<i>Index.js</i>	217
2. <i>Deno Runtime</i>	218
<i>Typescript</i>	218

<i>Package Manager</i>	218
<i>Browser Object Support</i>	218
<i>Built-in Tooling</i>	219
<i>Sandbox</i>	220
3. <i>Deno Infrastructure</i>	223
<i>Deno Front-end</i>	223
<i>Deno Middle-end</i>	226
<i>Deno Back-end</i>	226
3. <i>Deno Program</i>	227
<i>REPL Mode Execution</i>	227
<i>Script Execution</i>	228
<i>Eval Mode</i>	228
4. <i>Deno Command</i>	229
<i>Deno Options</i>	229
<i>Deno Sub Command</i>	230
<i>Deno Help</i>	231
<i>Run Javascript & Typescript Files</i>	232
<i>Deno Permissions</i>	232
<i>Deno Bundling</i>	234
<i>Deno Testing</i>	235
<i>Deno Reload Module</i>	235

<i>Deno Install</i>	235
<i>Deno Formatting</i>	235
<i>Deno Upgrade Command</i>	236
5. <i>Deno Standard Module</i>	237
<i>Module fs</i>	238
<i>Module http</i>	238
<i>Module datetime</i>	239
<i>Module node</i>	239
<i>Module ws</i>	240
6. <i>Deno Third-party Modules</i>	241
<i>Subchapter 3 – Typescript</i>	242
1. <i>Javascript</i>	242
2. <i>Node.js</i>	244
3. <i>Typescript</i>	247
<i>Compilation</i>	248
<i>Static Typing</i>	249
<i>Typescript Compiler</i>	249
<i>Compile Typescript</i>	250
<i>Typescript Compiler Options</i>	251
<i>Typescript Playground</i>	254
<i>Subchapter 4 – Fundamental Deno</i>	255

1. <i>Deno Task Runner</i>	255
<i>Edit Environment Variable</i>	256
<i>Create Training Directory</i>	256
<i>Create index.ts</i>	258
2. <i>Hello World</i>	259
3. <i>Comment</i>	259
4. <i>Variable Declaration</i>	261
<i>Variable</i>	261
<i>Binding</i>	263
<i>Reserved Words</i>	263
<i>Naming Convention</i>	265
<i>Case Sensitivity</i>	267
<i>Loosely Typed Language</i>	269
<i>Typescript Static Typing</i>	269
<i>Var Keyword</i>	269
<i>Let Keyword</i>	270
<i>Constant Keyword</i>	272
5. <i>Expression & Operator</i>	274
<i>Statement</i>	274
<i>Expression</i>	275
<i>Operator & Operand</i>	276

Operator Precedence	276
Arithmetic Operator.....	276
Arithmetic Operation.....	277
Comparison Operator.....	282
Logical Operator	285
Assignment Operator	287
6. Javascript Strict Mode	289
Legacy Code	289
7. Automatic Add Semicolon	292
8. Clean Code Variable Declaration	293
Avoid Global Variable	293
Declaration on Top.....	293
Subchapter 5 – Deno Data Types	294
1. Javascript Data Types	294
Apa itu Data?.....	294
Apa itu Types?.....	295
Apa itu Generic Variable?	296
Javascript Data Types.....	296
Apa itu Pointer?.....	297
Apa itu Stack & Heap?.....	298
Primitive Types	299

<i>Apa itu Primitive & Reference Values?</i>	302
<i>Reference Types</i>	303
<i>Primitive as Object via Object Wrapper</i>	305
2. <i>Typescript Data Type</i>	307
<i>Typescript Type Annotation</i>	307
<i>Declare Explicit</i>	308
<i>Declare Implicit</i>	308
3. <i>String Data Types</i>	309
<i>String Type</i>	309
<i>Template String</i>	310
<i>Escaping</i>	310
<i>String Concatenation</i>	311
<i>String Interpolation</i>	311
<i>String Object</i>	312
<i>String Property</i>	312
<i>String Methods</i>	313
4. <i>Number Data Types</i>	316
<i>Number Type</i>	317
<i>Infinity</i>	317
<i>NaN</i>	319
<i>Number Object</i>	321

<i>Number Property</i>	321
<i>Number Methods</i>	324
<i>Number Accuration</i>	329
<i>Imprecise Calculation</i>	331
<i>Solution to Imprecise</i>	332
<i>Fixed Number</i>	332
<i>Numeric Conversion</i>	333
<i>Math Object</i>	334
<i>Hexadecimal, Binary dan Octadecimal</i>	335
5. <i>Booleans Data Types</i>	336
<i>Boolean Type</i>	336
6. <i>Inferred Type</i>	337
<i>Dynamic Typed</i>	337
<i>Static Typed</i>	338
7. <i>Type Conversion</i>	339
<i>String To Number</i>	339
<i>String To Decimal Number</i>	339
<i>Number to String</i>	339
<i>Decimal Number to String</i>	340
<i>Boolean to String</i>	340
8. <i>Check Data Type</i>	341

<i>Primitive Types</i>	341
<i>Reference Types</i>	343
9. <i>Any Type</i>	344
<i>Parameter Any Type</i>	346
<i>Option noImplicitAny</i>	346
10. <i>Type Union</i>	347
<i>Parameter Union Type</i>	347
11. <i>Symbol Type</i>	348
12. <i>Type Widening</i>	350
<i>Undefined</i>	350
<i>Null</i>	351
<i>Option strictNullCheck</i>	352
13. <i>BigInt Data Types</i>	353
<i>Arbitrary Precision</i>	353
<i>Arithmetic Operation</i>	355
<i>Comparison</i>	355
14. <i>Custom Type</i>	357
15. <i>Clean Code Data Types</i>	359
<i>Declare Primitive Not Object</i>	359
<i>Subchapter 6 – Control Flow</i>	360
1. <i>Block Statements</i>	360

2. Conditional Statements	361
3. Ternary Operator	363
4. Multiconditional Statement	364
5. Switch Style	366
Subchapter 7 – Loop & Iteration	368
1. While Statement	369
2. Do ... While Statement	371
3. For Statement	373
4. For ... Of	376
5. For..in	377
6. Break Statement	378
7. Continue Statement	380
8. Labeled Statement	382
Subchapter 8 – Function	383
1. Apa itu Function?	383
Function Declaration	384
Function Expression	384
2. First-class Function	385
What is Execution Context (EC)?	385
3. Simple Function	389
Typescript Version	390

4. <i>Function Parameter</i>	391
<i>Parameter Type</i>	392
<i>Optional Parameter</i>	393
<i>Default Parameter</i>	394
<i>Rest Parameter</i>	395
5. <i>Function Return</i>	397
6. <i>Function For Function Parameter</i>	399
7. <i>Function & Local Variable</i>	400
8. <i>Function & Outer Variable</i>	401
9. <i>Callback Function</i>	402
10. <i>Arrow Function</i>	404
11. <i>Multiline Arrow Function</i>	405
12. <i>Anonymous Function</i>	406
13. <i>Function Constructor</i>	407
14. <i>Function As Expression</i>	408
15. <i>Nested Function</i>	409
16. <i>Argument Object</i>	410
17. <i>This Keyword</i>	411
<i>Implicit Binding</i>	411
18. <i>Call & Apply Function</i>	414
<i>Explicit Binding</i>	415

<i>Call</i>	416
<i>Apply</i>	416
19. <i>IIFE</i>	418
20. <i>Clean Code Function</i>	419
<i>Always Declare Local Variable</i>	419
<i>Use Named Function Expression</i>	419
<i>Use Default Parameter</i>	420
<i>Function is not statement</i>	420
<i>Subchapter 9 – Error Handling</i>	422
1. <i>Syntax Error</i>	423
<i>Missing Syntax</i>	423
<i>Invalid Syntax</i>	424
2. <i>Logical Error</i>	425
3. <i>Runtime Error</i>	427
<i>Reference Error</i>	427
<i>Range Error</i>	428
<i>Type Error</i>	429
<i>Syntax Error</i>	429
4. <i>Try & Catch</i>	431
<i>Error Object Properties</i>	433
<i>Stack Trace</i>	433

<i>Finally</i>	435
5. <i>Custom Error</i>	437
<i>Subchapter 10 – Object</i>	441
1. <i>Apa itu Fundamental Objects?</i>	443
2. <i>Custom Object</i>	444
<i>Object Initializer</i>	444
<i>Object Property</i>	445
<i>Object Method</i>	446
<i>Object Constructor</i>	446
<i>Function Constructor</i>	447
<i>Object Prototype</i>	448
<i>Getter & Setter</i>	450
<i>Object Destructure</i>	451
<i>Typescript Type Template</i>	452
<i>Complex Object Type</i>	453
3. <i>Custom Object Property</i>	455
<i>Add Object Property</i>	456
<i>Access Object Property</i>	457
<i>Delete Object Property</i>	458
<i>Check Object Property</i>	459
4. <i>Custom Object Method</i>	461

<i>Access Object Method</i>	461
<i>Add Object Method</i>	462
5. <i>Custom Object Enumeration</i>	464
6. <i>JSON</i>	466
<i>JSON & Object Literal</i>	466
<i>Stringify</i>	468
<i>Parse JSON</i>	469
<i>Parse Date in JSON</i>	470
<i>Subchapter 11 – Classes</i>	471
1. <i>Class-based language</i>	475
<i>Function Constructor</i>	475
<i>Class ECMAScript2015</i>	476
<i>Class Typescript</i>	477
2. <i>Class Inheritance</i>	480
3. <i>Class Access Modifier</i>	482
<i>Public</i>	482
<i>Private</i>	482
<i>Protected</i>	482
<i>Public Behaviour</i>	483
<i>Protected Behaviour</i>	484
<i>Private Behaviour</i>	484

<i>Readonly Property</i>	485
4. <i>Class Constructor</i>	486
<i>Spread Argument</i>	487
5. <i>Static Keyword</i>	489
6. <i>Super Method</i>	490
7. <i>Method Override</i>	491
8. <i>Accessor Getter & Setter</i>	493
9. <i>Abstract Class</i>	495
10. <i>Class Declaration</i>	498
<i>Subchapter 12 – Interface</i>	499
1. <i>Design Interface</i>	499
<i>Create Malware Interface</i>	500
<i>Implement Malware Interface</i>	501
<i>Class Type & Interface Type</i>	504
<i>Create Ransomware Interface</i>	505
<i>Implement Ransomware Interface</i>	506
<i>Interface Extend Multi-Interface</i>	508
<i>Create Trojan Interface</i>	508
<i>Implement Multi-interface</i>	509
2. <i>Interface & Class</i>	510
<i>Interface Extends Interface</i>	510

<i>Interface Extends Class</i>	510
<i>Interface Cant Implements Interface</i>	510
<i>Interface Implements Class</i>	511
<i>Class Cant Extends Interface</i>	511
<i>Class Extends Class</i>	511
<i>Class Implements Interface</i>	511
<i>Class Implements Class</i>	511
<i>Subchapter 13 – Collection</i>	512
1. Apa itu <i>Collection</i> ?	512
<i>Iterable</i>	513
<i>Keyed</i>	513
<i>Destructurable</i>	513
2. Apa itu <i>Indexed Collections</i> ?	514
<i>Array</i>	514
<i>Create Array</i>	515
<i>Iterate Array</i>	519
<i>Array Property & Method</i>	524
<i>Array Properties</i>	524
<i>Multidimensional Array</i>	528
<i>Matrix</i>	530
<i>Multi-type Array</i>	531

<i>Array of Interface</i>	532
3. <i>Keyed Collections</i>	534
<i>Map</i>	534
<i>Set</i>	540
<i>Subchapter 14 – Tuples</i>	544
<i>Create Tuple</i>	544
<i>Tuple Property & Method</i>	548
<i>Application</i>	550
<i>Subchapter 15 – Enum</i>	554
1. <i>Create Enum</i>	554
2. <i>Access Enum</i>	555
3. <i>Enum Function Argument</i>	556
<i>Subchapter 16 – Generic</i>	557
1. <i>Generic Function</i>	557
<i>Type Union Way</i>	558
<i>Generic Type</i>	558
2. <i>Reverse Array Element</i>	561
<i>Create Generic Function</i>	562
<i>Create Parameter For Generic Function</i>	562
<i>Create Return For Generic Function</i>	562
<i>Create Internal Array Data Structure</i>	562

<i>Create Internal Algorithm</i>	563
<i>Create Return Inside Generic Function</i>	564
<i>Subchapter 17 – Module</i>	567
1. <i>Module Concept</i>	567
<i>Module</i>	567
<i>Module Format</i>	569
<i>Module Loaders</i>	571
<i>Module Bundlers</i>	571
2. <i>Node.js Module</i>	573
<i>Create & Export Module</i>	575
<i>Use Module</i>	576
<i>Export Multiple Method & Value</i>	577
<i>Export Style</i>	578
<i>Destructure Assignment</i>	578
<i>Export Class</i>	579
3. <i>Deno Module</i>	580
<i>Create & Export Module</i>	580
<i>Use Module</i>	581
<i>Compile to Javascript</i>	582
<i>Export Visualization</i>	583
<i>Import Module</i>	585

<i>Subchapter 18 – Namespace</i>	586
1. <i>Create Namespace</i>	586
<i>Nested Namespace</i>	587
<i>Namespace Reference</i>	588
2. <i>Application – Stack Data Structure</i>	589
<i>Stack Data Structure</i>	589
<i>Create Interface Data Structure</i>	589
<i>Create Class Stack Data Structure</i>	590
<i>Create Stack Object</i>	591
<i>Access Peek Method</i>	591
<i>Access Push Method</i>	592
<i>Access Pop Method</i>	592
3. <i>Application – Stack Data Structure</i>	593
<i>Queue Data Structure</i>	593
<i>Create Interface Data Structure</i>	593
<i>Create Class Queue Data Structure</i>	594
<i>Create Queue Object</i>	595
<i>Access Peek Method</i>	595
<i>Access Enqueue Method</i>	595
<i>Access Dequeue Method</i>	596
<i>Chapter 4</i>	598

<i>Deno Standard Module</i>	598
<i>Subchapter 1 – Filesystem Programming</i>	598
1. <i>File & Directory</i>	599
<i>Create Directory</i>	599
<i>Check Directory</i>	601
<i>Delete Directory Content</i>	602
<i>Move File & Directory</i>	602
<i>Copy File & Directory</i>	603
<i>WriteJson</i>	604
<i>ReadJson</i>	605
<i>writeFileStr</i>	605
<i>ReadFileStr</i>	606
<i>Walk</i>	606
<i>Subchapter 2 – HTTP Server</i>	608
<i>Subchapter 3 – Testing</i>	610
1. <i>Why Software Testing</i>	610
2. <i>Type of Software Testing</i>	612
<i>Bug</i>	612
<i>Defect Level</i>	612
<i>Functional Testing</i>	613
<i>Unit Testing</i>	614

Code Refactoring.....	615
Code Coverage	615
Integration Testing	615
System Testing	616
Use Acceptance Testing	617
Alpha & Beta Testing.....	617
3. Deno Testing	618
Assertion	618
assert Function.....	619
equal Function	620
4. Deno Benchmarking.....	622
bench Function	622
runBenchmarks Function	622
Subchapter 4 – Debugging	624
Chapter 5.....	629
REST API.....	629
Subchapter 1 – Oak Framework	629
1. Create Basic Server	630
Class Application.....	630
Register Middleware.....	631
Start Listening.....	632

2. Middleware.....	633
Control Execution.....	633
Context Object.....	634
Cookies.....	634
Request.....	635
Request Properties.....	636
Request Headers.....	637
Request Methods.....	637
Response.....	639
Response Properties.....	639
Response Method.....	640
3. Routing.....	642
Class Router.....	643
Route Method.....	643
Route Path.....	645
Route Parameter.....	645
Route Handler.....	646
Register Router Middleware.....	646
4. Error Handling.....	648
Funtion isHttpError.....	649
Enum Status.....	649

<i>Try & Catch</i>	650
5. <i>Static Content</i>	651
<i>Subchapter 2 – Introduction to Database</i>	652
1. <i>Database Function</i>	654
<i>Data Management</i>	654
<i>Scalability</i>	655
<i>Data Heterogenity</i>	655
<i>Efficiency</i>	656
<i>Persistence</i>	656
<i>Reliability</i>	656
<i>Consistency</i>	656
<i>Non-redundancy</i>	657
2. <i>Use Case Database</i>	658
<i>Aplikasi Penjualan (Sales)</i>	658
<i>Aplikasi Accounting</i>	658
<i>Aplikasi HR (Human Resources)</i>	658
<i>Aplikasi Manufaktur</i>	658
<i>Aplikasi e-Banking</i>	658
<i>Aplikasi Keuangan</i>	659
3. <i>Data Analytic</i>	660
<i>Subchapter 3 – PostgreSQL</i>	661

1. Intro PostgreSQL.....	661
Enable External Access.....	662
psql Program.....	662
pgAdmin Program.....	663
2. Database Administration.....	670
Create Role.....	670
Create Database	671
Grant Database	671
List Database.....	671
Test Role Access Database.....	672
Setup Server.....	672
Create Table	674
Insert Data	674
Subchapter 4 – Blog App.....	676
Chapter 6.....	677
Mastering Node.js.....	677
Subchapter 1 – Node.js	677
1. Node.js System	677
Test Node.js Executable.....	678
2. I/O Scaling Problem	680
3. Process & Thread.....	682

<i>Multithread</i>	683
4. <i>Core Modules & libuv</i>	687
<i>Subchapter 2 – V8 Javascript Engine</i>	689
1. <i>The Call Stack</i>	691
<i>Synchronous Program</i>	692
<i>Asynchronous Program</i>	694
<i>Event Loops</i>	699
<i>Blocking</i>	700
<i>Non-blocking</i>	700
2. <i>Javascript Compilation Pipeline</i>	702
<i>Interpreter & Compiler</i>	702
<i>Machine Code</i>	707
<i>Ignition & Turbofan</i>	709
<i>Intermediate Representation (IR)</i>	711
<i>Bytecode</i>	712
<i>Just-in-Time Compilation</i>	713
<i>Compiler Development Philosophy</i>	713
3. <i>Memory Management</i>	714
<i>Memory Lifecycle</i>	714
<i>Allocation Example</i>	714
<i>Garbage Collector</i>	715

<i>Mark-and-Sweep Algorithm</i>	715
<i>Subchapter 3 – Node.js Application</i>	724
1. <i>Running Javascript File</i>	724
2. <i>Node REPL</i>	726
3. <i>Package Manager</i>	728
4. <i>Node Package Manager</i>	730
<i>npm commands</i>	730
5. <i>Node Package Registry</i>	733
6. <i>Create Node.js package</i>	736
<i>package.json</i>	738
<i>Directive</i>	739
<i>Search Package</i>	741
<i>Install Package</i>	741
<i>Remove Package</i>	743
<i>View Package</i>	743
<i>Publish Package</i>	743
<i>Create Package</i>	744
7. <i>Publish Node.js Package</i>	745
8. <i>Node.js Application</i>	749
<i>Subchapter 4 – Debugging Node.js</i>	751
1. <i>Debug on Visual Studio Code</i>	751

2. Built-in Node.js Debugger	755
Subchapter 5 – Asynchronous.....	759
1. Callback.....	759
2. Promise.....	761
Promise Constructor	761
Resolve.....	762
Reject.....	764
Example Promise.....	765
3. Fetch	767
Fetch Kawal Korona.....	767
Promise Chaining	768
4. Async Await.....	769
Async Function.....	770
Await.....	771
Error Handling	772
5. Top Level Await.....	774
Daftar Pustaka.....	775
Tentang Penulis.....	777
My Books.....	780
Belajar dengan Jenius Golang.....	780
Belajar Dengan Jenius AWS & Node.js.....	781

Belajar Dengan Jenius Amazon IAM	782
Develop Security Software With C#	783

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

- Mengetahui Apa itu **Open Web Platform?**
 - Mengetahui Apa itu **Technical Specification?**
 - Mengetahui Apa itu **HTML 5.2?**
 - Mengetahui Apa itu **Web Assembly?**
 - Mengetahui Apa itu **EcmaScript?**
 - Mengetahui Apa itu **WebSocket?**
 - Mengetahui Apa itu **WebRTC?**
 - Mengetahui 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**) lainnya 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**. Terdapat *localStorage* dan *sessionStorage*. Lalu terdapat **IndexedDB API** untuk menyimpan 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** :

W3C TR Specifications (133 Specs)

Recommendations (39 Specs)

- [Accessible Rich Internet Applications \(WAI-ARIA\) 1.0](#)
- [Content Security Policy Level 2](#)
- [Cross-Origin Resource Sharing](#)
- [Encrypted Media Extensions](#)
- [Geolocation API Specification](#)
- [HTML 5.1](#)
- [HTML 5.2](#)
- [HTML Canvas 2D Context](#)
- [HTML Image Description Extension \(longdesc\)](#)
- [HTML Media Capture](#)
- [HTML5 Web Messaging](#)
- [High Resolution Time](#)
- [Indexed Database API](#)
- [Indexed Database API 2.0](#)
- [Mathematical Markup Language \(MathML\) Version 3.0](#)
- [Media Source Extensions](#)
- [Micropub](#)
- [Navigation Timing](#)
- [Page Visibility](#)
- [Performance Timeline](#)

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* :

Part of the open web platform

WebAssembly is designed to maintain the versionless, feature-tested, and backwards-compatible *nature of the web*. WebAssembly modules will be able to call into and out of the JavaScript context and access browser functionality through the same Web APIs accessible from JavaScript. WebAssembly also supports *non-web* embeddings.

Gambar 3 Web Assembly As Open Web Platform[2]

Initially 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 *performance* 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

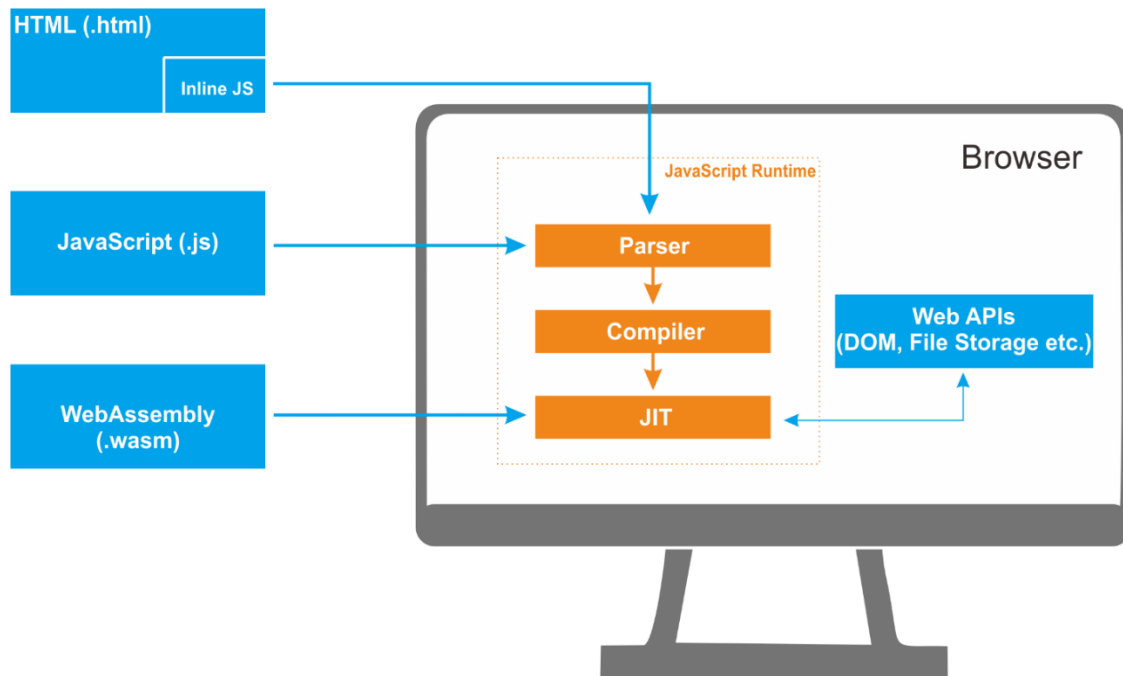
Mampu memproduksi *native code* yang telah dioptimasi sehingga dapat memanfaatkan secara optimal seluruh kemampuan *performance* yang dimiliki mesin.

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

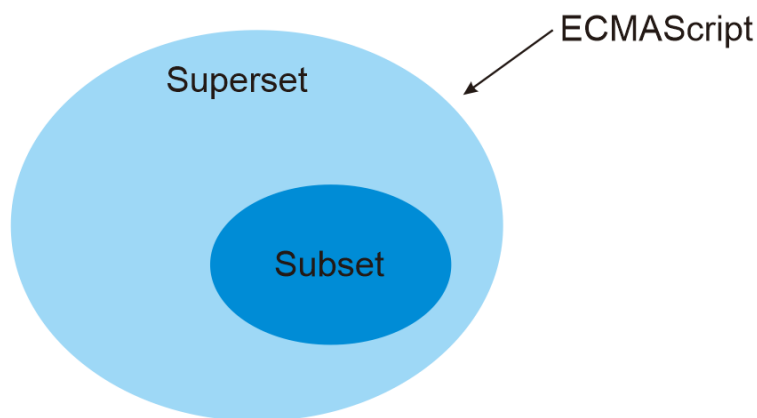
Draft ECMA-262 / June 16, 2019

ECMAScript® 2020 Language Specification



Gambar 5 Ecma 262

Javascript adalah bahasa yang telah dijelaskan 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 pengembangannya selaras. Saat ini **WebSocket Protocol** sudah bisa digunakan hampir diseluruh *browser* seperti *Microsoft Edge, Firefox, Chrome, Internet Explorer, Safari* dan *Opera*.

Specification

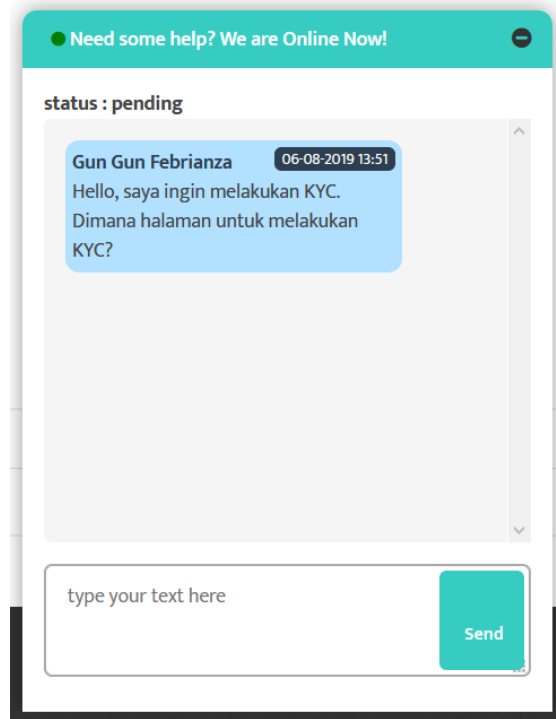
Spesifikasi dapat dilihat di :

<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 (*Marketskoin 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.

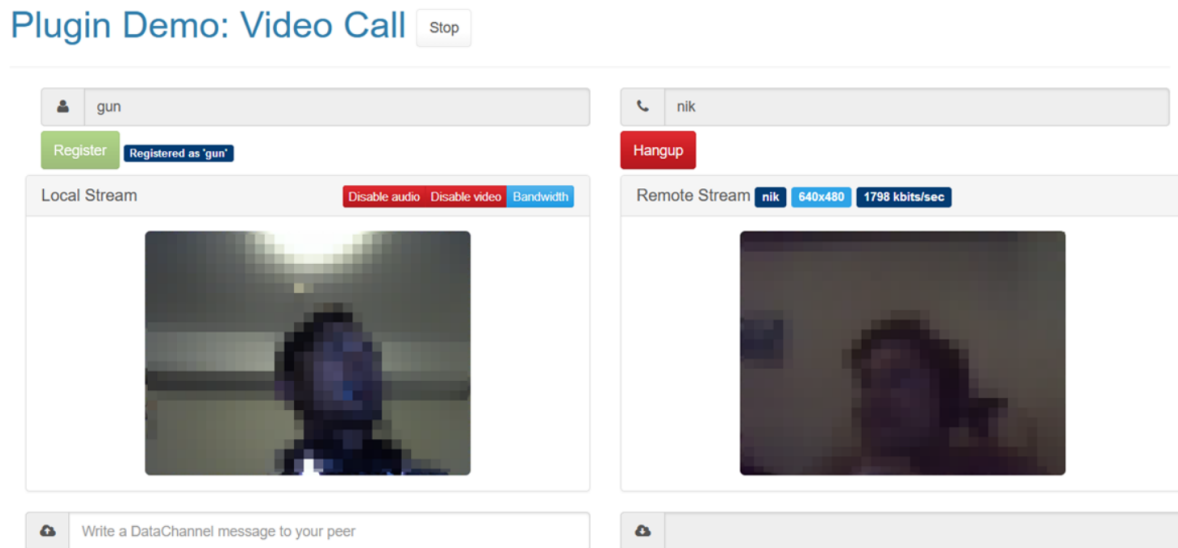
Specification

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)*.

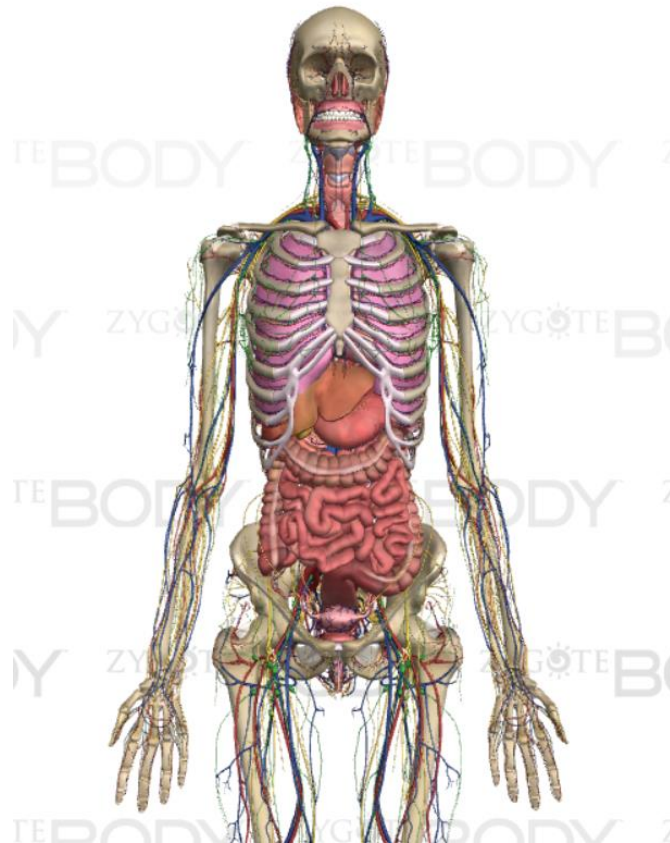
Specification

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.

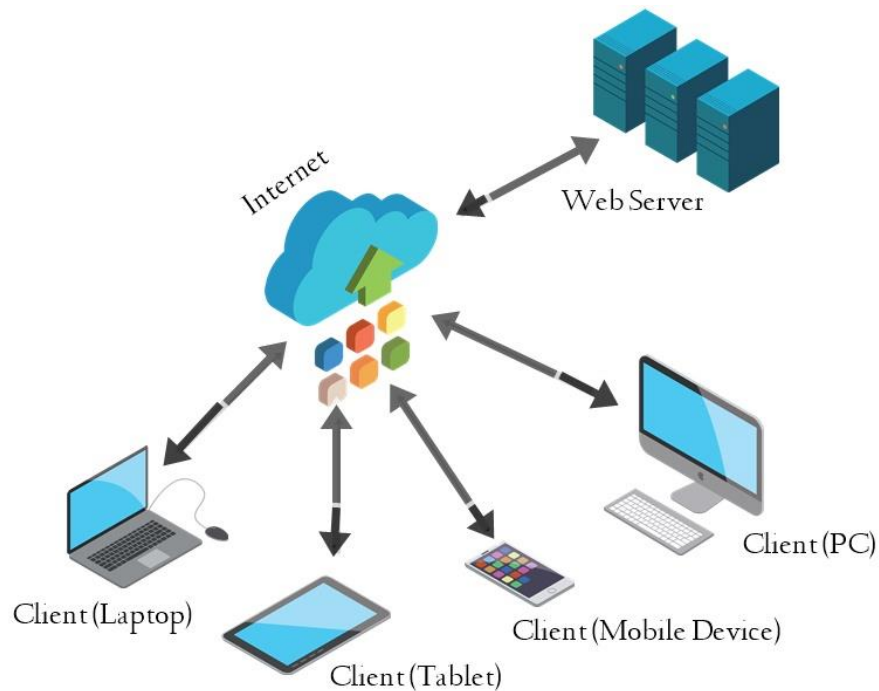
— Christopher Dayagdag

Subchapter 2 – Objectives

- Mengetahui Apa itu **Server**?
 - Mengetahui Apa itu **Virtual Private Server**?
 - Mengetahui Apa itu **Web Server**?
 - Mengetahui Apa itu **Web Page**?
 - Mengetahui Apa itu **Network**?
 - Mengetahui Apa itu **Internet**?
 - Mengetahui Apa itu **Internet Exchange Point**?
 - Mengetahui Apa itu **Content Delivery Network (CDN)** ?
 - Mengetahui Apa itu **Cloud Computing**?
 - Mengetahui 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 memungkinkan user untuk bisa mengeksekusi **business logic** dengan sebuah *browser*.^[8]

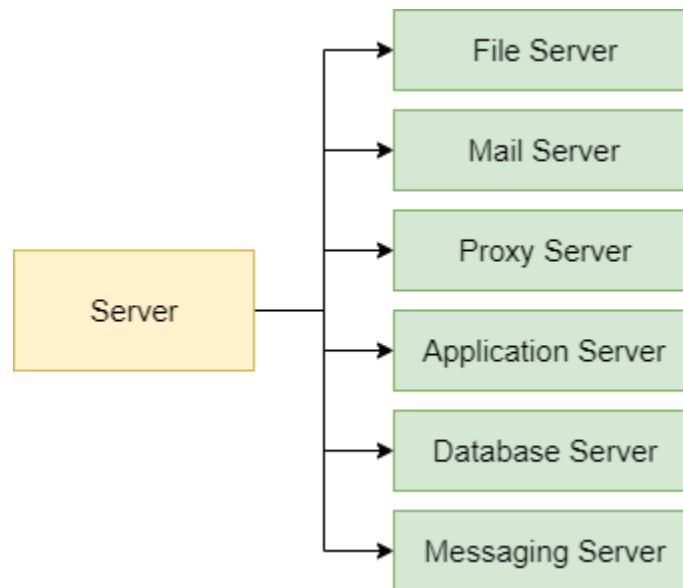
Saat ini pengembangan teknologi *web* telah distandarisasi dalam **Open Web Platform (OWP)**. Setiap **Web Application** terdiri dari beberapa elemen yang saling bekerja satu sama lain agar bisa memberikan suatu layanan. Sebuah *web application* terdiri dari **web server** dan **clients** yang bisa anda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



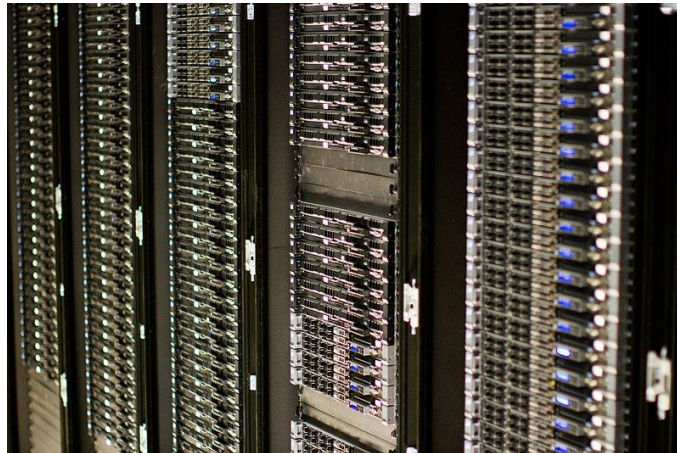
Gambar 11 Server Classification

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 layanannya 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 penggunaanya 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***,

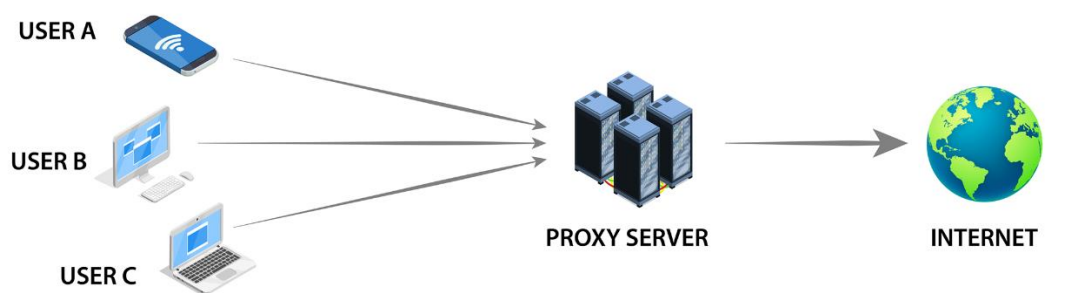
3. **HTTP server**, atau
4. **Network File System (NFS) server**.

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

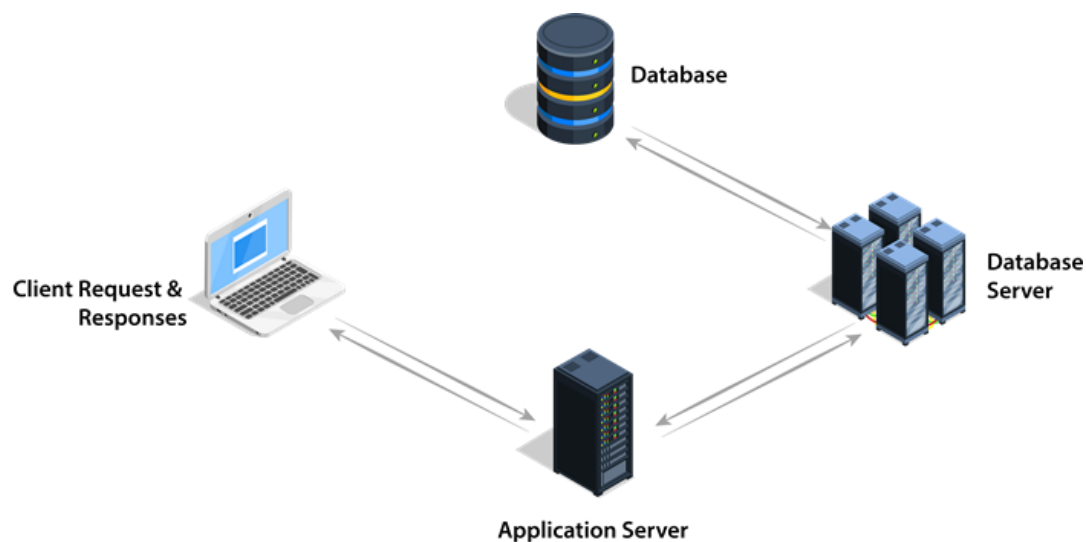
Proxy server adalah *server* yang menjadi perantara dalam jaringan komputer untuk melayani permintaan dari *client* agar bisa melakukan akses *resources* dari suatu *remote server*. Salah satu *proxy server* adalah *Reverse Proxy Server*, *server* akan menerima *request*

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 *subscribe*.

2. *Subscriber*

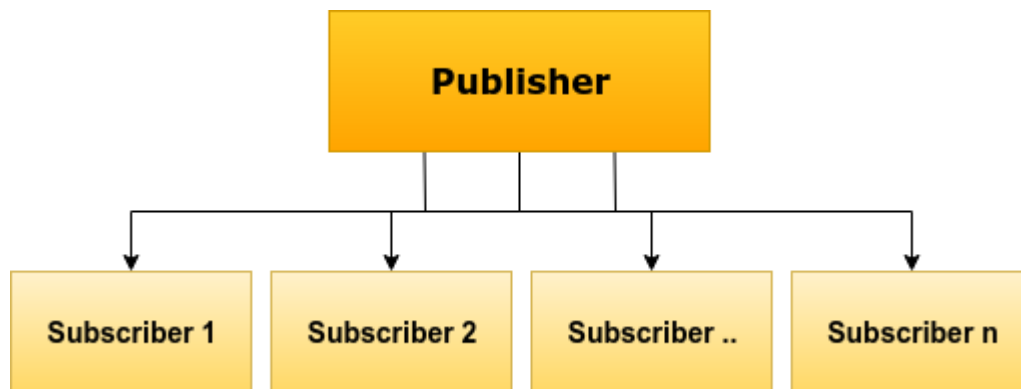
Sebuah *service* yang akan mendapatkan pesan yang di *broadcast* oleh *publisher*.

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 layanannya 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.

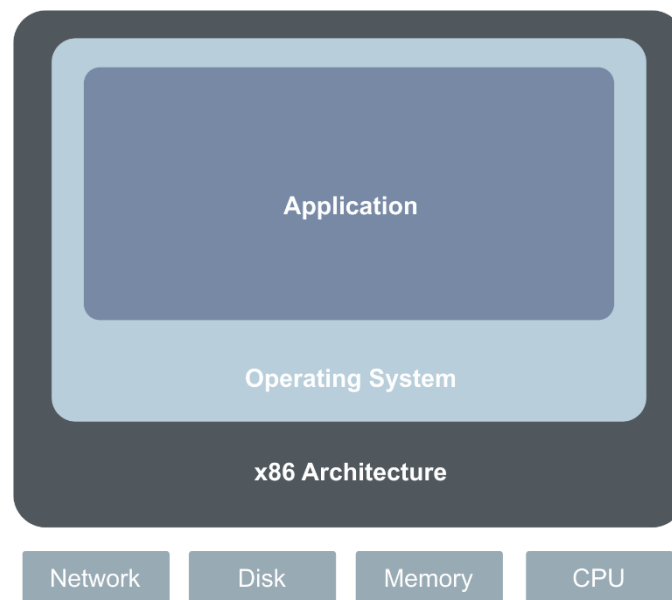
Virtualization

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

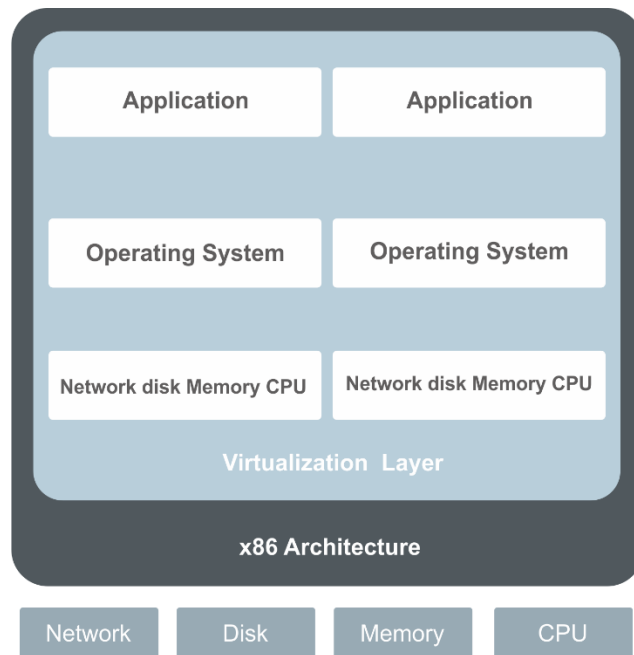
Apa sih yang dimaksud *Multiple Execution Environment*?

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



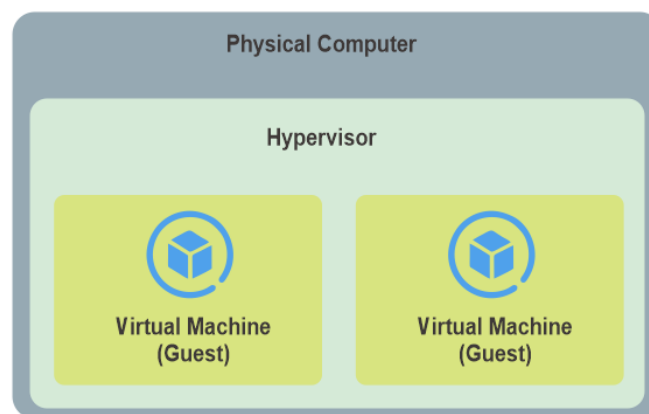
Gambar 16 OS Image pada suatu mesin komputer

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*, *desktop virtualization*, *network virtualization*, *storage virtualization* dan masih banyak lagi.



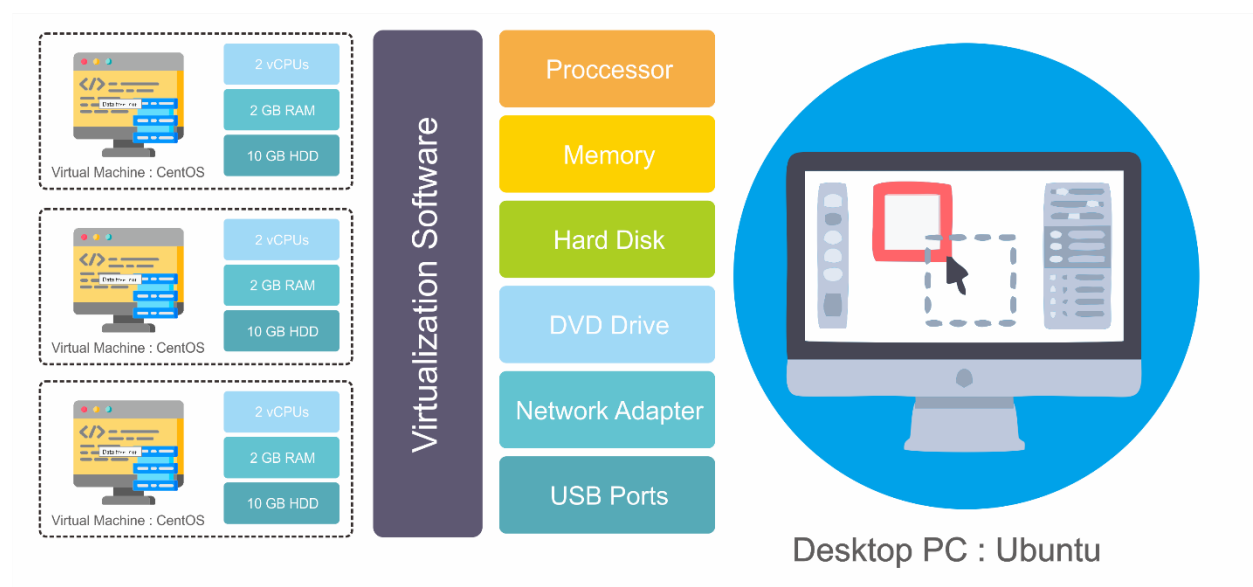
Gambar 18 Virtual Machine

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 mewujudkannya 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].

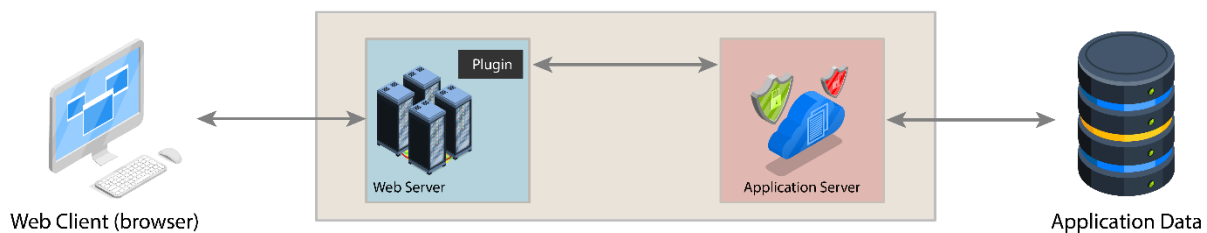
3. Web Server

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

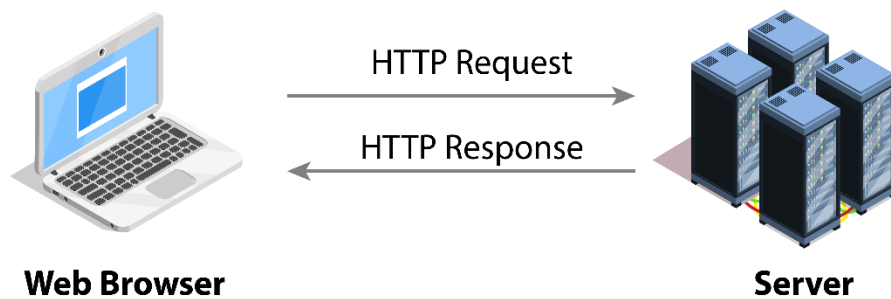
Ada beberapa *Web Server* yang sering digunakan di antaranya adalah :

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

4. Web Page

Static Web Page

Static Web Page, adalah sebuah halaman yang kontennya 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*.



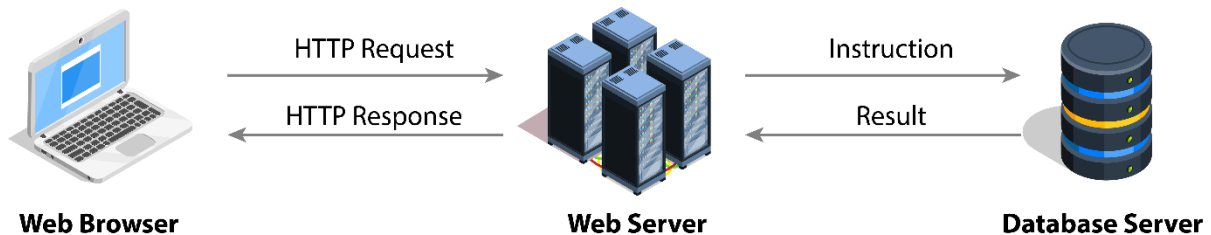
Gambar 21 Static Web Page Processing

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

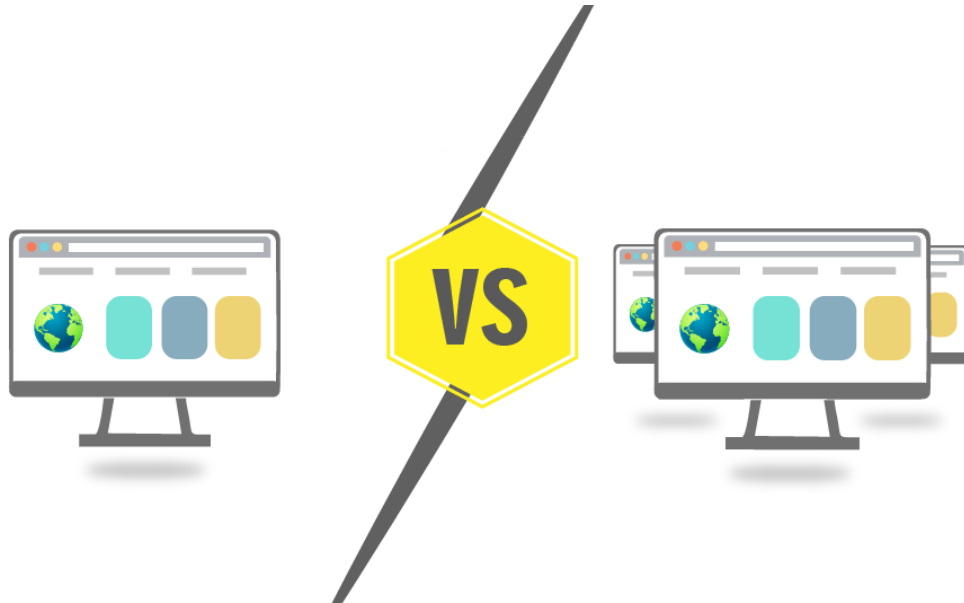
1. **Web Worker**, sebuah *javascript* yang berjalan di dalam *background process* melalui *thread* terpisah sehingga kita bisa melakukan pekerjaan dibelakang layar.
2. **Background Synchronization** untuk mengirimkan *request* ke *server* saat terhubung kembali ke internet
3. **Push Notification** untuk 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 adalah **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*

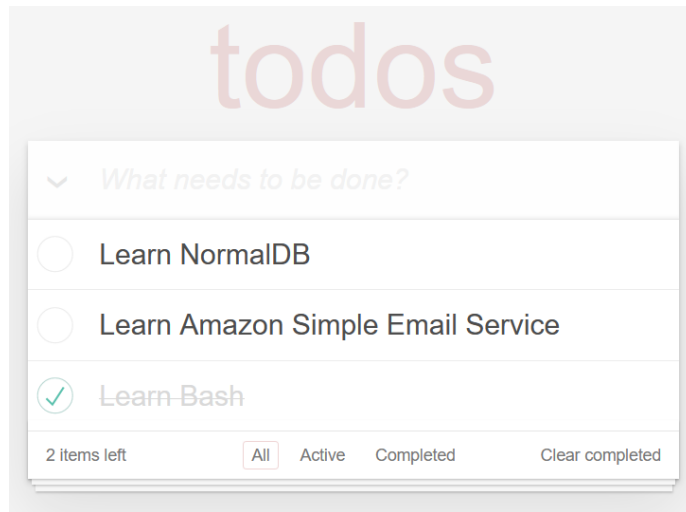
update tanpa harus melakukan **reload** halaman. *Server* hanya mengirimkan satu halaman saja kepada *client*, selanjutnya aktivitas *routing* dan *paging* dilakukan di dalam *browser*.



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* lainnya 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 :

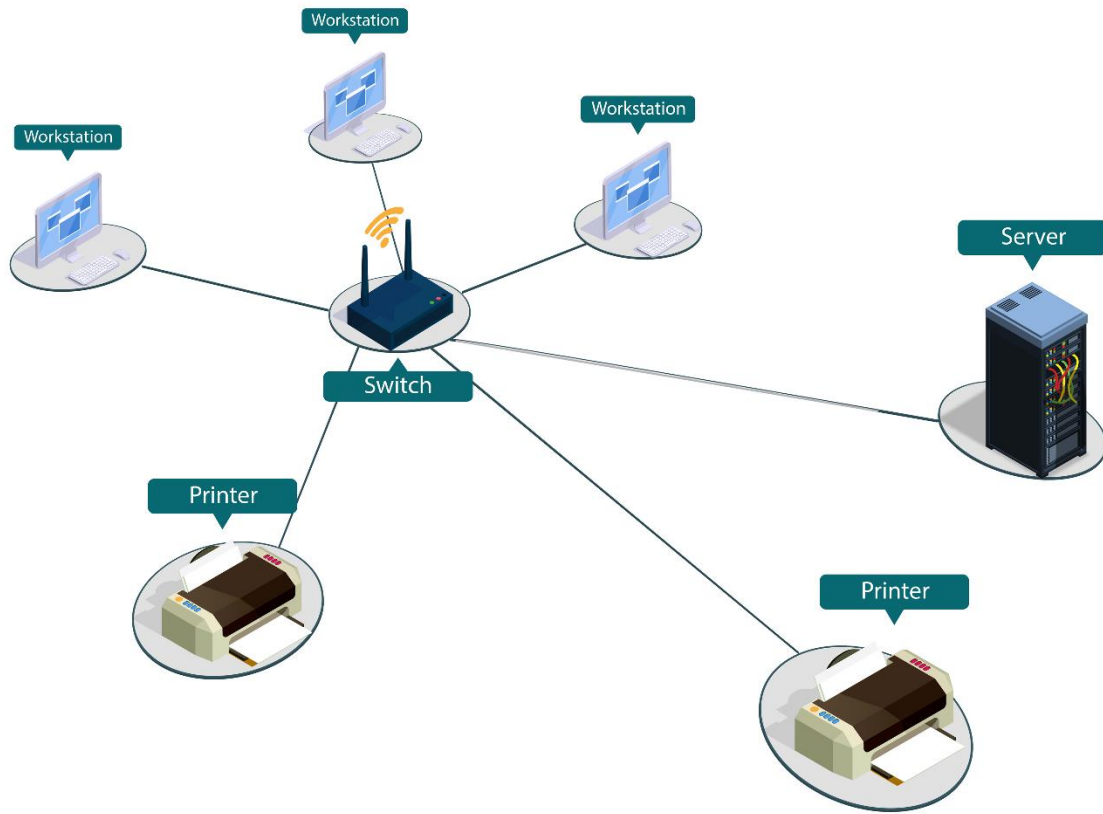
Personal Area Network (PAN)



Gambar 25 Personal Area Network (PAN)

Personal Area Network (PAN) terbentuk ketika terdapat dua atau lebih komputer atau **smart phone** saling terhubung menggunakan **wireless** dengan jarak yang cukup dekat.

Local Area Network (LAN)

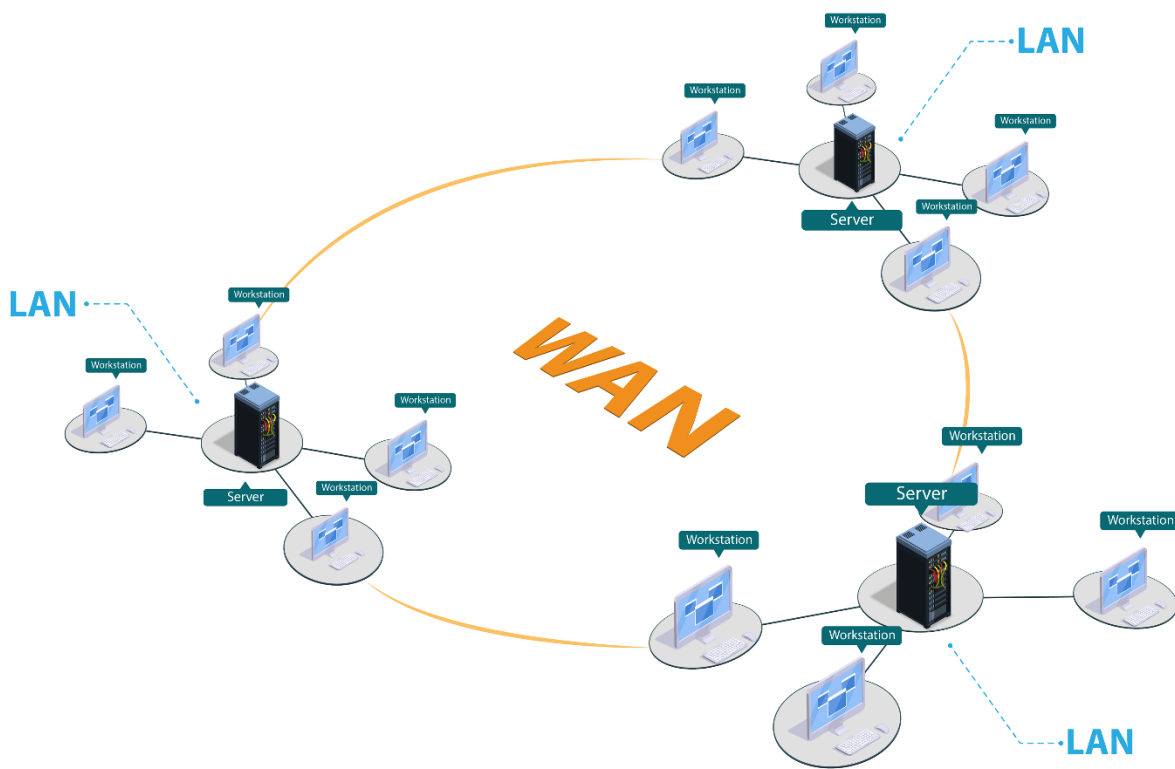


Gambar 26 Local Area Network (LAN)

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)



Gambar 27 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 lainnya sebuah *router* akan memastikan *network* mana yang paling dekat ke target komputer dan mengirimnya melalui *network* tersebut.

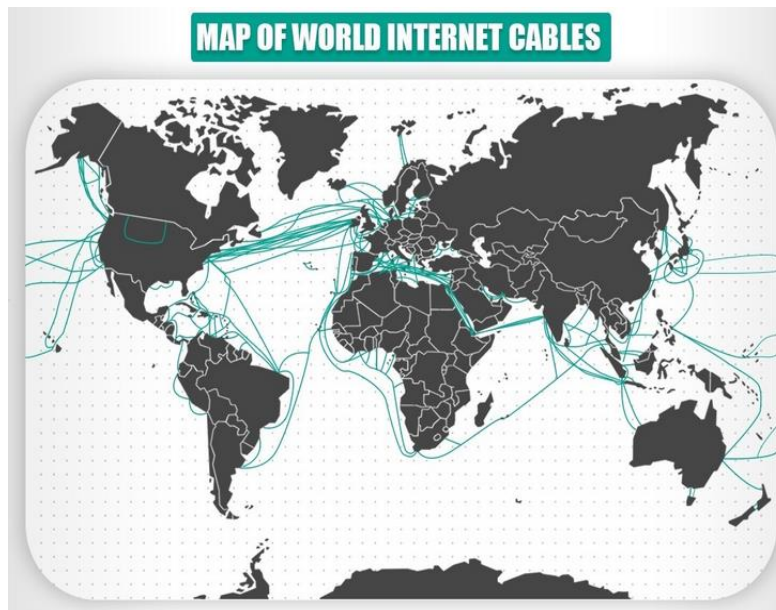
Sebuah WAN bisa dimiliki oleh sebuah perusahaan privat atau lebih dari satu perusahaan privat.

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 28 Internet Cables

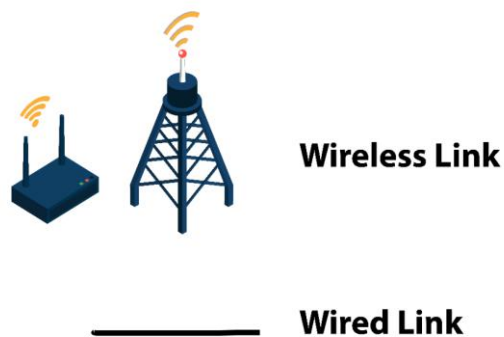
Internet adalah sebuah **network** skala besar yang terdiri dari sekumpulan *networks* kecil yang saling terhubung. Hari Ini *Internet* telah terhubung hampir keseluruhan penjuru dunia, kabel *Fiber Optic* telah terpasang mengelilingi bumi melalui lautan, daratan & pegunungan.

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 29 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 redundansi data yang lebih tinggi.

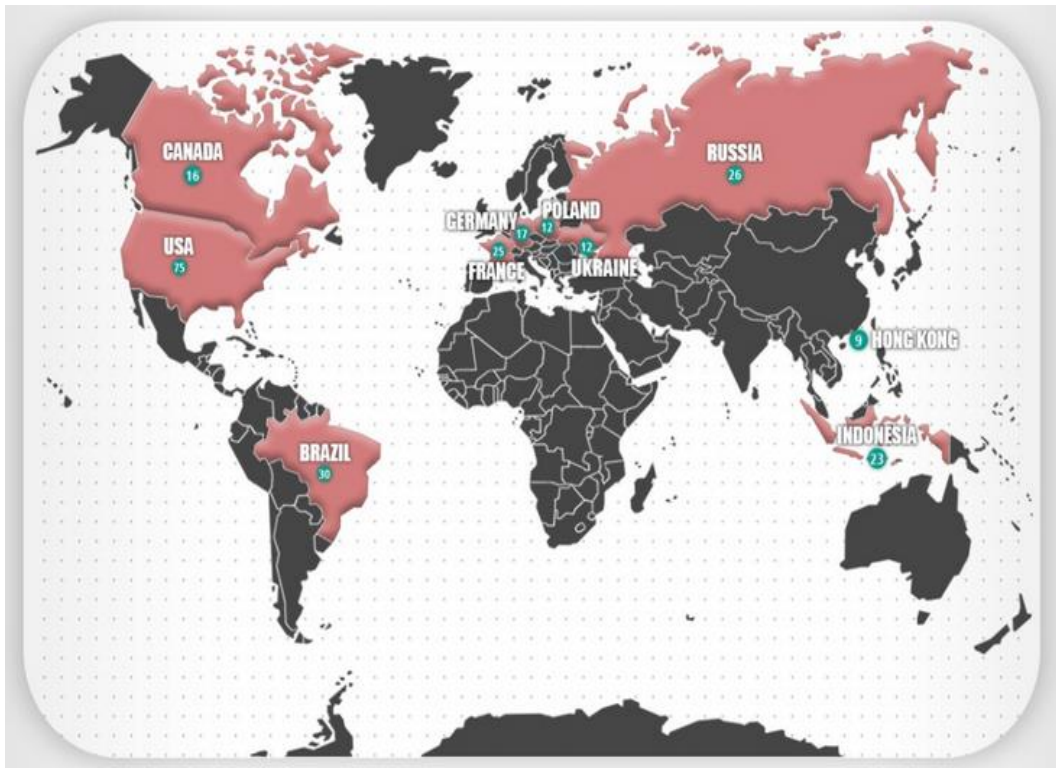
Pada tahun 1950 terdapat lebih dari 5.500 satelit yang didominasi oleh USA dan Rusia. Digunakan untuk aplikasi militer, digunakan oleh perusahaan telekomunikasi, korporasi multinasional, lembaga keuangan, *broadcast* televisi dan radio.

Teknologi *satellite* mulai termarginalisasi 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 Commission (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]

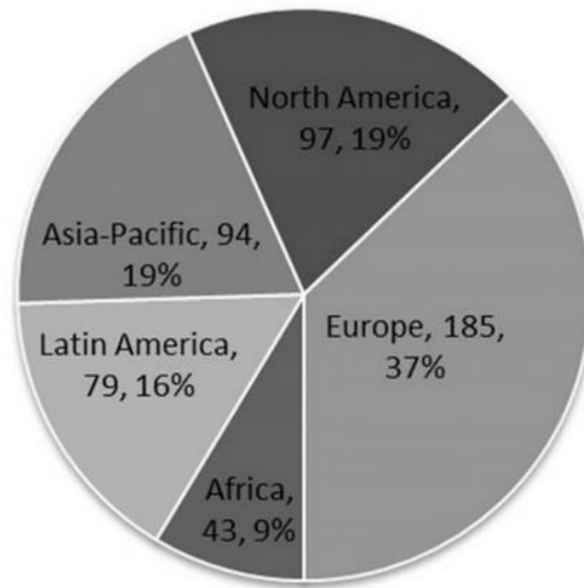
7. Internet Exchange Point



Gambar 30 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 31 Jumlah Internet Exchange Point Tahun 2017[17]

8. Content Delivery Network (CDN)

Content Delivery Network (CDN) atau **Content Distribution Network (CDN)** 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?

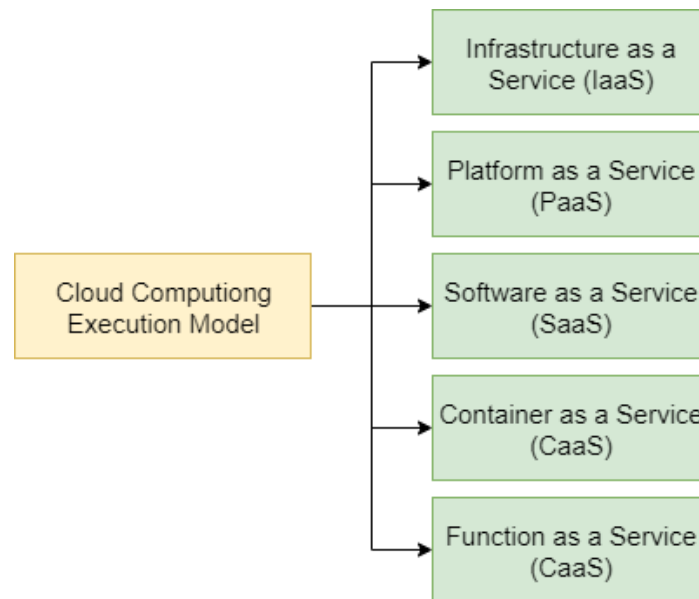
Penjelasan yang ideal untuk menjelaskan cloud computing adalah dengan memberikannya sebuah terminologi yang disebut dengan *Everything as a Service*, disingkat XaaS^[18].

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 32 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 populer saat *docker* pertama kali dirilis ke publik pada tahun 2013. Aktivitas *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 bisa meraih kemampuan *fault-tolerance*, *high-availability* dan *scalability* sebuah *orchestration tool* seperti *docker swarm*, *kubernetes* atau *apache mesos* di sediakan untuk manajemen sekumpulan *container* di dalam sebuah *cluster*.

Hal ini membuat layanan **CaaS** diperkenalkan kepada publik agar kita bisa melakukan *Build*, *Ship* dan *Run Container* secara cepat dan efisien. Juga dapat digunakan untuk pekerjaan berat lainnya 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 perusahaan besar yang memberikan layanan *computing*, *storage* dan *network resources*. Di antaranya adalah perusahaan *Google*, *Microsoft* dan *Amazon*. Penggunaan *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*).

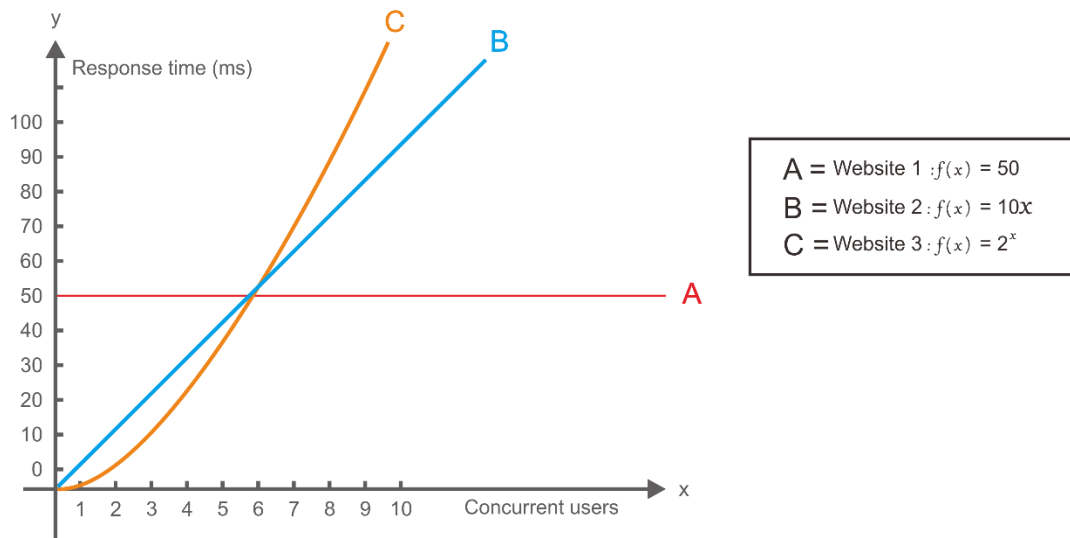
2. Linear Scaling

Mengacu pada *performance system* yang berubah secara proporsional mengikuti beban (*workload*) yang diterima. Pada kasus di atas jika beban permintaan 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 33 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

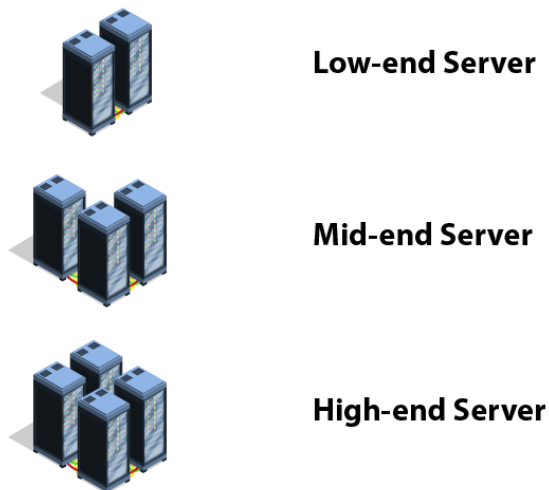
dengan baik saat kita menambahkan kemampuan komputasi, penyimpanan dan sumber jaringan untuk meningkatkan kapasitas ketika *demand* telah muncul.

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 34 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 35 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 dikelola 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

- Mengetahui Apa itu **HTTP & URL?**
 - Mengetahui Apa itu **HTTP & DNS?**
 - Mengetahui Apa itu **HTTP Transaction?**
 - Mengetahui Apa itu **HTTP Request?**
 - Mengetahui Apa itu **HTTP Response?**
 - Mengetahui 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 lainya 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*).

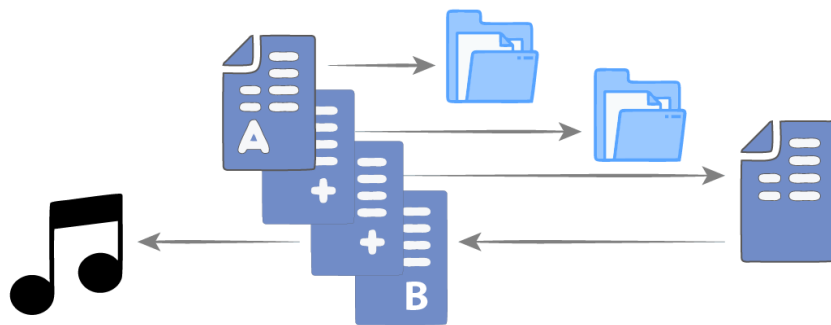
HTML

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

Hypermedia

Sebelumnya pada *hypertext* kita dapat berpindah dari satu teks dokumen ke teks dokumen lainnya. **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 36 Akses Secara Nonlinear

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

World Wide Web (www)

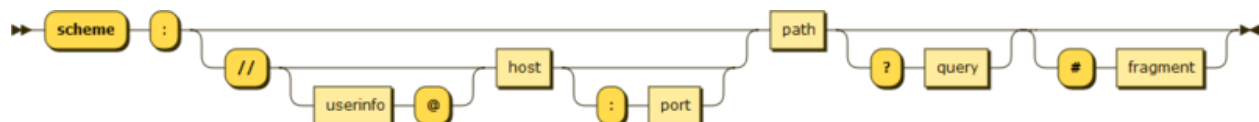
HTTP adalah fondasi untuk komunikasi data dengan **world wide web** atau lebih sering dikenal dengan singkatan (www). Lalu apa sih *world wide web* itu sendiri?

World wide web adalah sebuah dunia maya (*cyberspace*) dimana sekumpulan dokumen dan sumber *web* lainnya 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)

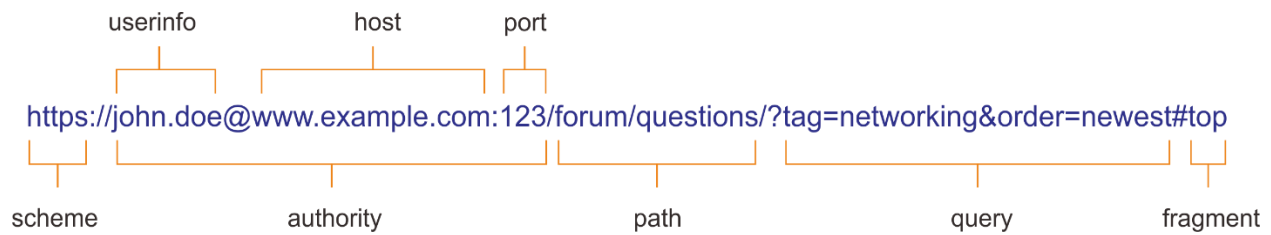
Uniform Resources Identifier (URI) adalah sekumpulan *character* yang membentuk *string* untuk identifikasi suatu **Web Resources** di dalam *world wide web (www)*. Agar bisa diterima secara universal terdapat aturan yang harus diikuti. Aturan yang dibuat menentukan **protocol** yang akan digunakan untuk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

Dibawah ini adalah *syntax diagram* untuk merepresentasikan URI :



Gambar 37 URI Syntax Diagram

Contoh lengkapnya :



Gambar 38 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 triliun *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 mengabaikannya 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

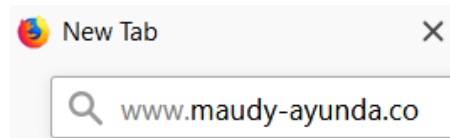
Dalam *URL* juga terdapat istilah ***unsafe character***. Sebuah *character* yang tidak direkomendasikan untuk digunakan untuk *URL*. Sebagai contoh penggunaan simbol # dalam *URL*, sebab telah digunakan untuk membuat *fragments*. Simbol lainnya adalah simbol ^ dan *space*. Meskipun begitu kita tetap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

<http://www.maudy-ayunda.co/%5Data%20saya.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 39 Domain

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



Gambar 40 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?

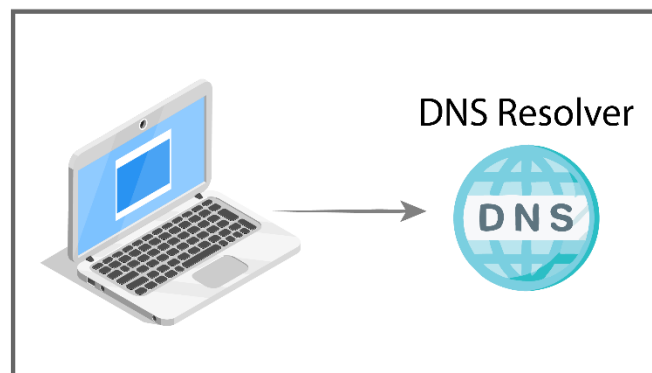
Ada beberapa proses yang terjadi sebelum berinteraksi dengan *web server*. Tentu sebuah koneksi antara *client* dan *server* harus 'didirikan' terlebih dahulu agar keduanya bisa saling berkomunikasi. Lokasi *web server* sendiri bisa berada dalam sekup intranet atau internet.

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*.

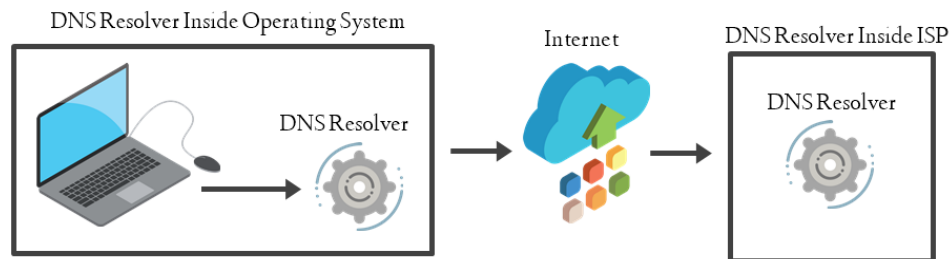
DNS Resolver Inside Operating System



Gambar 41 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 42 DNS Resolver dalam ISP

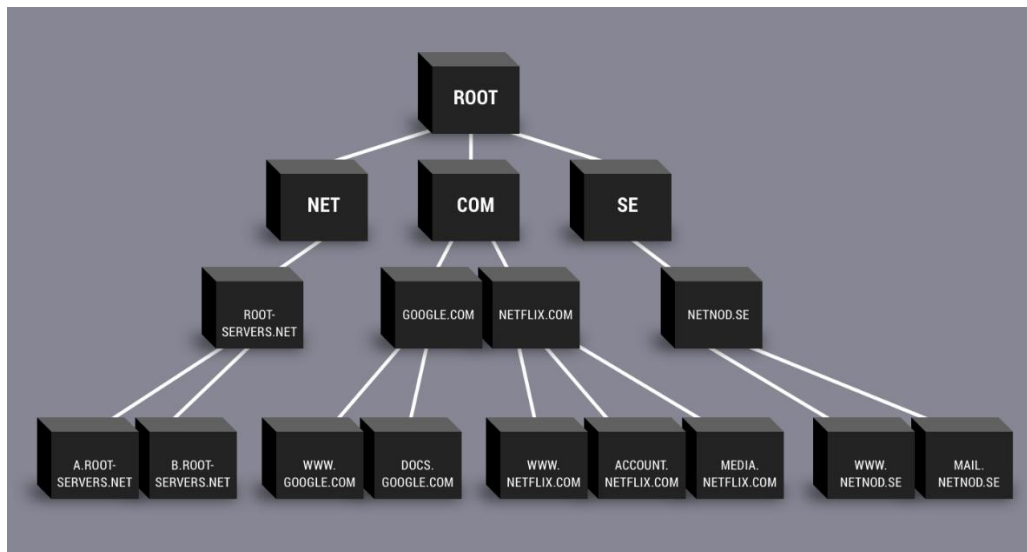
Root Server & TLD Server

Tak banyak materi yang membahas tentang **root server** terkecuali jika anda mempelajari sejarah pembangunan internet. *Root Server* adalah tulang belakang yang menjadi penyokong 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 43 Root Server

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

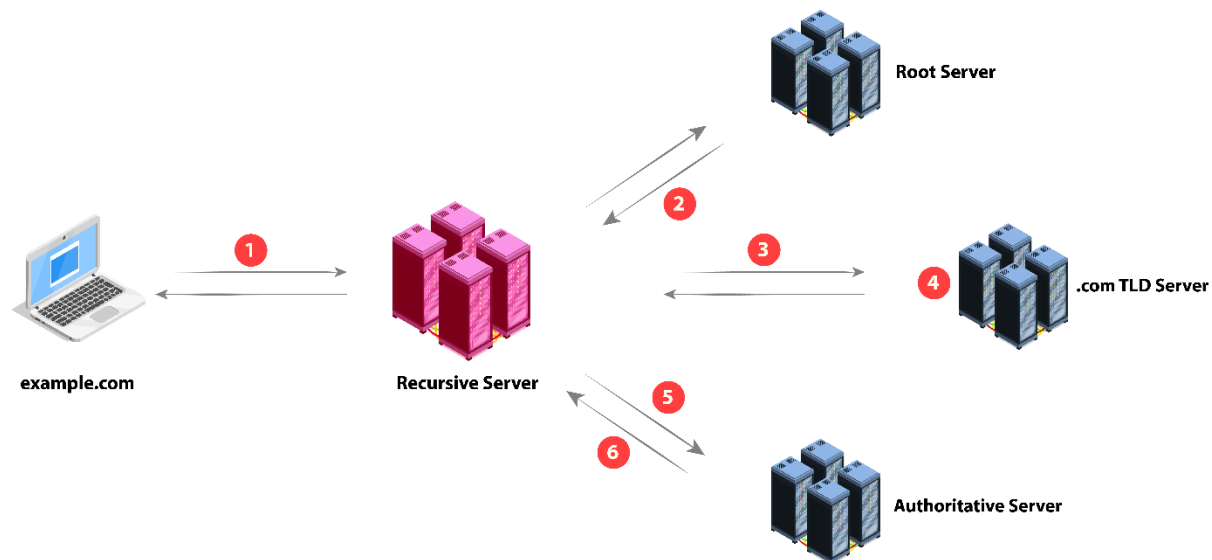
Root Server akan menerima permintaan dari *Recursive Resolver*, respon yang diberikan adalah informasi **DNS** dari **Top-level Domain**. Pada kasus yang diangkat dalam buku ini *top level domain* yang dicari adalah .co (www.maudy-ayunda.co). *Root Server* akan memberikan lokasi TLD Server untuk .co 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 mencari **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 44 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 manajemennya 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 Three-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 45 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 Exchange

Dilakukan antara *client* dan *server*.



Gambar 46 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 <i>web resource</i>
2	POST	Untuk update <i>web resources</i>
3	DELETE	Untuk menghapus <i>web resources</i>
4	PUT	Untuk menyimpan <i>web resources</i>

5	HEAD	Untuk mendapatkan <i>header web resources</i>
---	------	---

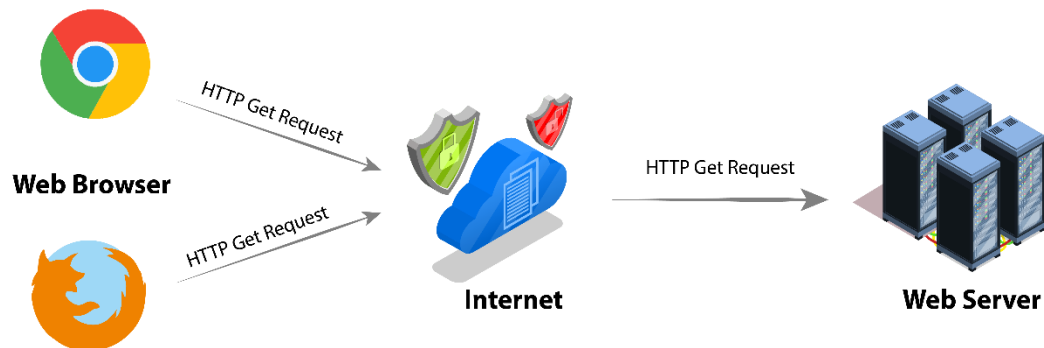
Setiap kali sebuah *HTTP Request* dilakukan di dalamnya harus terdapat salah satu *HTTP Method*.

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 47 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 melakukan *login* kedalam sebuah *website* tentu kita harus mengirimkan data akun berupa *username* dan *password*.

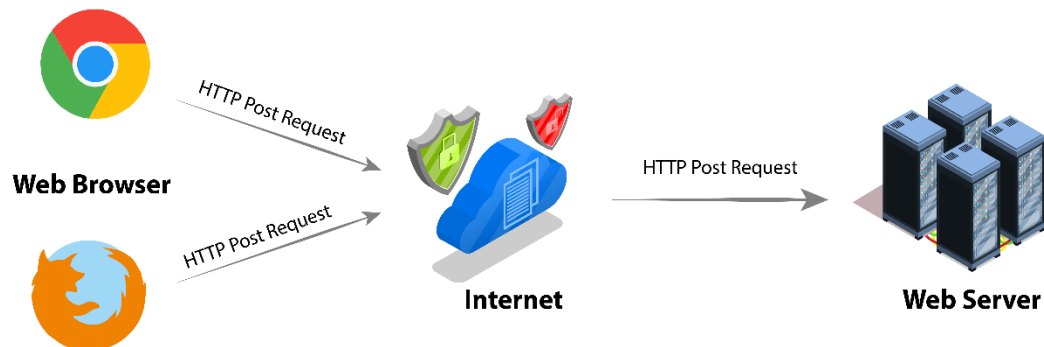
Di bawah ini adalah contoh kode untuk *form* halaman *login* menggunakan PHP :

```
<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 48 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 menggunakannya 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 lainnya. 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 <i>http header</i> .
2	User Agent	Informasi tentang <i>software</i> yang melakukan <i>request</i> . Di gunakan untuk mengetahui <i>browser</i> apa yang digunakan oleh <i>client</i> untuk melakukan <i>request</i> .
3	Accept	Menjelaskan <i>media type</i> yang dimana <i>user agent</i> akan menerima. Selain itu juga terdapat <i>accept-language</i> , <i>accept-encoding</i> , <i>accept-charset</i> dan sebagainya.
4	Cookie	Informasi <i>cookies</i> yang bisa digunakan <i>server</i> untuk mengenali <i>user</i> .
5	If-Modified-Since	Tanggal ketika <i>user agent</i> 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**.

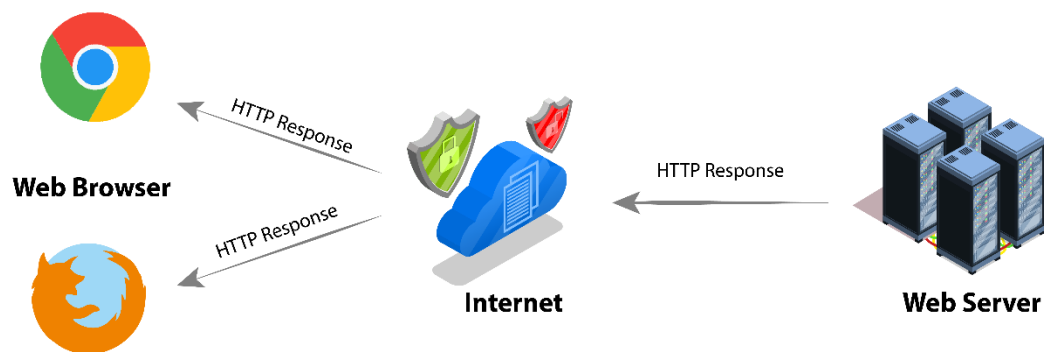
Jika kita perhatikan pada *HTTP Request* di atas terdapat "**q**" di beberapa *attribute header*. Nilai q akan selalu berada di antara 0 sampai dengan 1 yang merepresentasikan *quality value*. Nilai *quality value* ini didapatkan dari seberapa besar *user* atau *user agent* seringkali melakukan permintaan 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 ilustrasinya pada gambar di bawah ini :



Gambar 49 HTTP Post Request

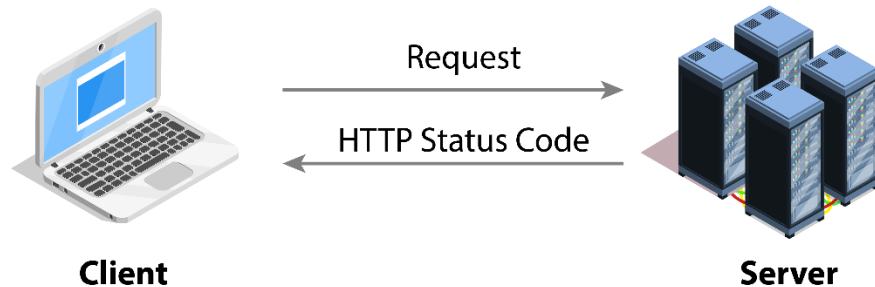
Di bawah ini adalah contoh pesan secara penuh pada *HTTP Response* :

```
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 50 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	<i>Ok</i>	Sebuah status yang ingin dilihat siapapun, <i>status code</i> dengan nilai 200 artinya semua berjalan dengan lancar.

301	<i>Moved Permanently</i>	<i>Web Resources</i> yang anda akses sudah tidak ada karena berpindah alamat secara permanen dari <i>URL</i> lama yang anda akses ke <i>URL</i> baru yang lain.
302	<i>Moved Temporarily</i>	<i>Web Resources</i> yang anda akses telah berpindah <i>URL</i> untuk sementara waktu, sehingga <i>URL</i> original harus tetap diketahui oleh <i>client</i> .
304	<i>Not Modified</i>	<i>Server</i> akan memberi tahu <i>client</i> bahwa <i>resources</i> tidak berubah semenjak terakhir kali <i>client</i> menerima <i>resources</i> sehingga bisa menggunakan data yang telah di <i>cached</i> secara <i>local</i> .
400	<i>Bad Request</i>	<i>Server</i> tidak mampu memahami permintaan yang diberikan <i>client</i> karena terdapat kesalahan dalam <i>HTTP Message</i> yang diberikan.
403	<i>Forbidden</i>	<i>Server</i> menolak permintaan anda untuk mengakses <i>web resources</i>
404	<i>Not Found</i>	<i>Web Resources</i> yang anda minta tidak ada
500	<i>Internal Service Error</i>	<i>Server</i> mengalami masalah saat memproses data yang diminta, biasanya karena kesalahan <i>programming</i> dalam <i>application server</i> .
503	<i>Service Unavailable</i>	<i>Server</i> tidak mampu memberikan layanan pada permintaan yang diberikan. Ini terjadi ketika <i>server</i> 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	<i>Informational</i>
200-299	<i>Successful</i>
300-399	<i>Redirection</i>
400-499	<i>Client Error</i>
500-599	<i>Server Error</i>

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

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

1. Data in The Low Level

Host



Gambar 51 End Point

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

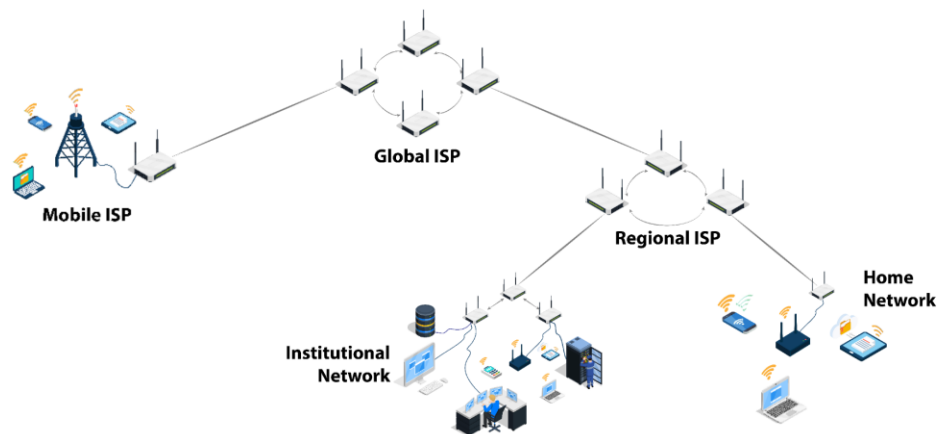
Socket

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 keseluruhan 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 52 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 lainnya.

ABCD天地玄黃
色は匂へあゝうん

Gambar 53 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 54 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 menejemahkannya 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

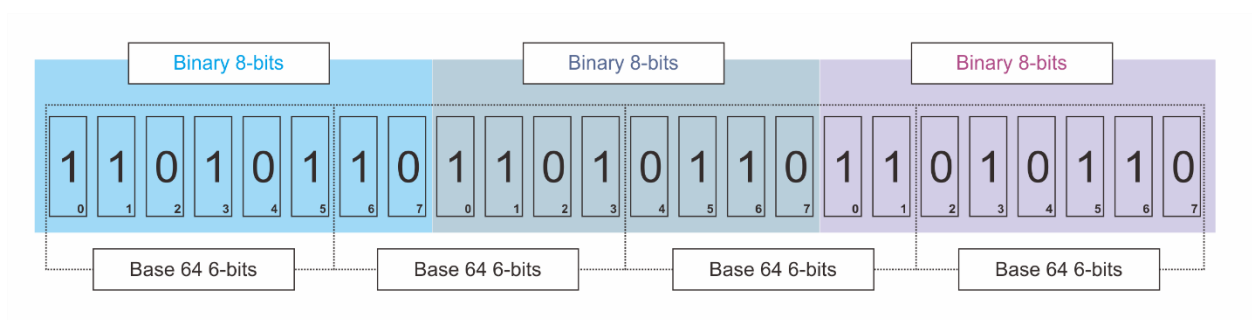
Encoding adalah proses untuk mengubah suatu data menjadi sebuah format yang lebih efisien untuk disimpan atau ditransmisikan.

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 menggunakannya untuk menyisipkan gambar dan data biner lainnya 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*.

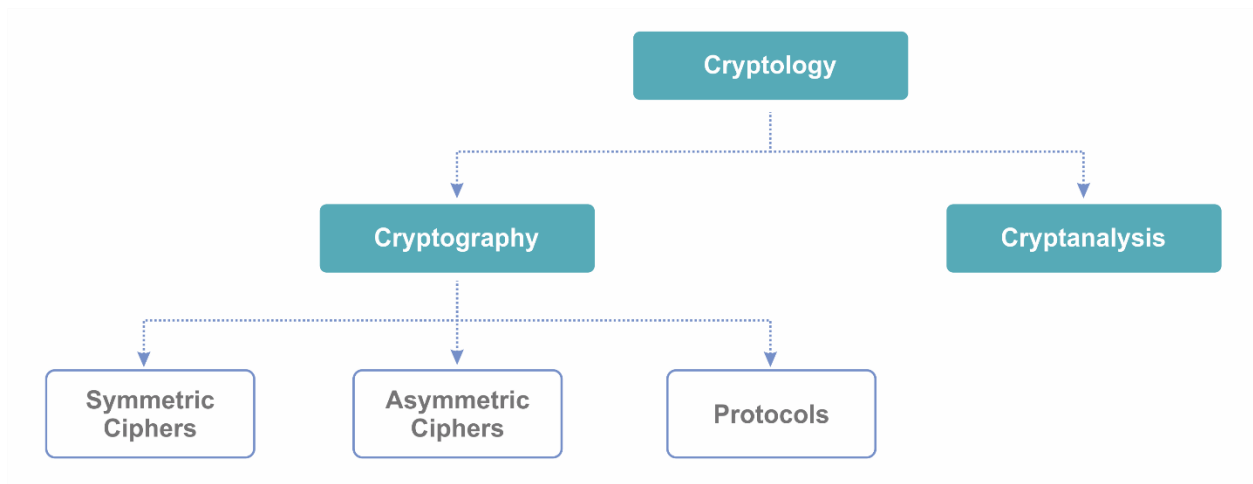


Gambar 55 Base64 Representation

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 56 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.

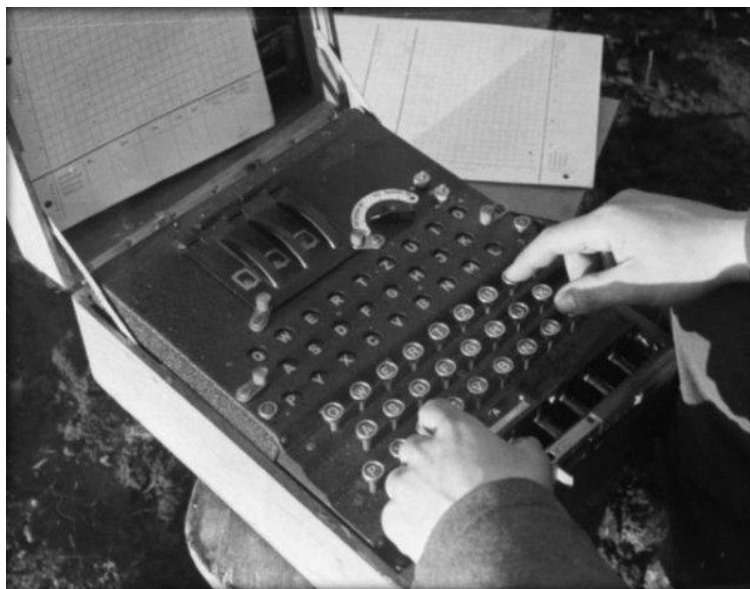
Hari ini *cryptography* lebih dari sekedar seni, *Modern Cryptography* sudah menggunakan pendekatan ilmu matematika untuk melindungi informasi digital, sistem dan komputasi terdistribusi dari serangan pihak ketiga.

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 menggunakannya.

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 57 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 (**ciphertext**), 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

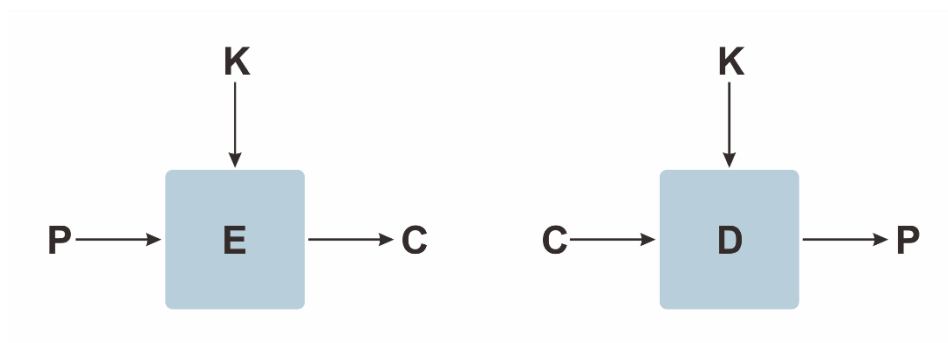
Setiap *encryption* menggunakan suatu algoritma yang disebut dengan **Cipher** dan memiliki nilai rahasia yang disebut dengan **secret key**. Jika kita tidak memiliki *secret key* maka kita tidak akan bisa melakukan proses **Decryption** untuk membaca sebuah *encrypted information*.

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 58 Encryption & Decryption

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 lainnya, 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.

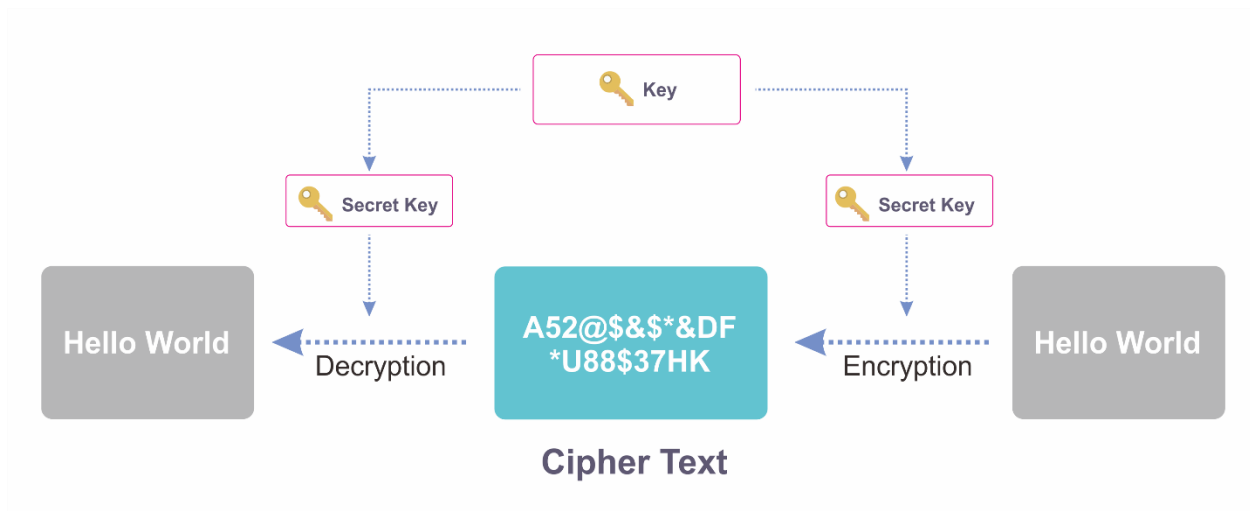
Bagaimana cara menghitungnya?

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 59 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 frekuensi 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)*.

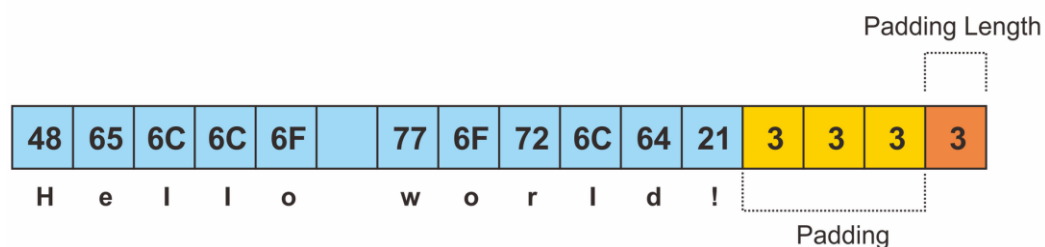
Padding

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*.

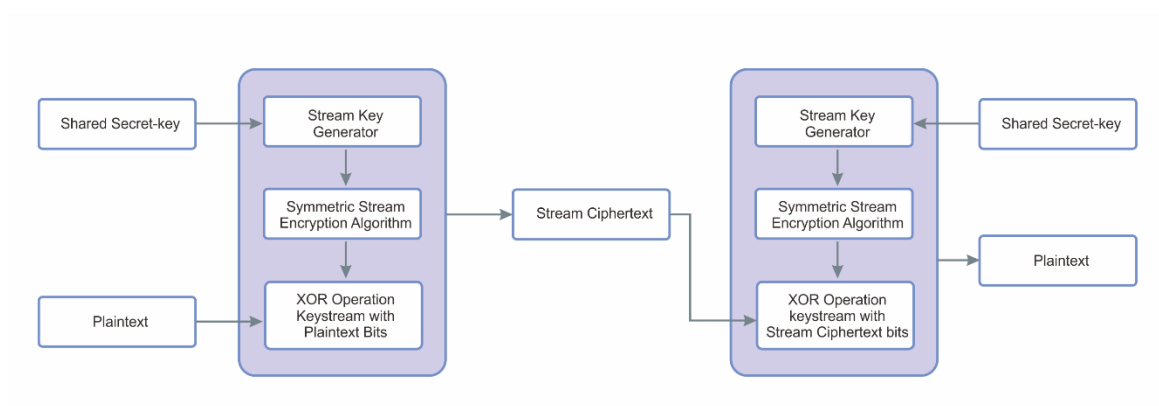


Gambar 60 Padding dalam SSL/TLS

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 61 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)*.

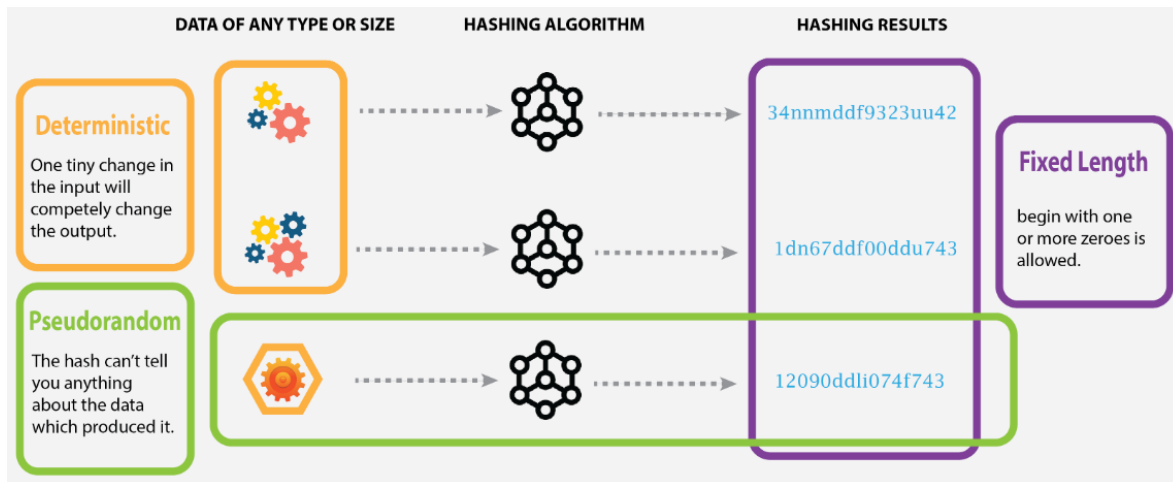
Stream Cipher memiliki keunggulan dalam hal kecepatan sehingga sangat cocok untuk melakukan komputasi pada *resources* yang terbatas, seperti dalam komunikasi *cellphones*.

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 62 Ilustrasi 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*). Sebagai contoh pada hash algorithm SHA256 kita dapat memproduksi 40 karakter unik untuk setiap *arbitrary input length*.

Hashing adalah proses yang terjadi saat *hash function* mengolah *input* untuk memproduksi *message digest* atau *hash value*. **Hashing** digunakan untuk menyembunyikan data & memeriksa integritas suatu data.

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* atau 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)* diciptakan.

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

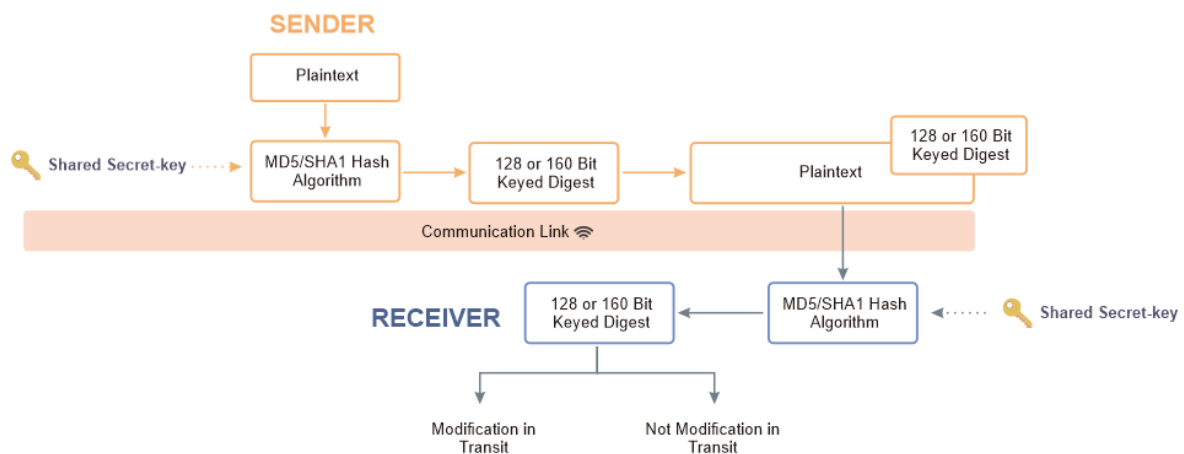
Cryptographic Hash

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 63 MAC Menggunakan Cryptographic Hash

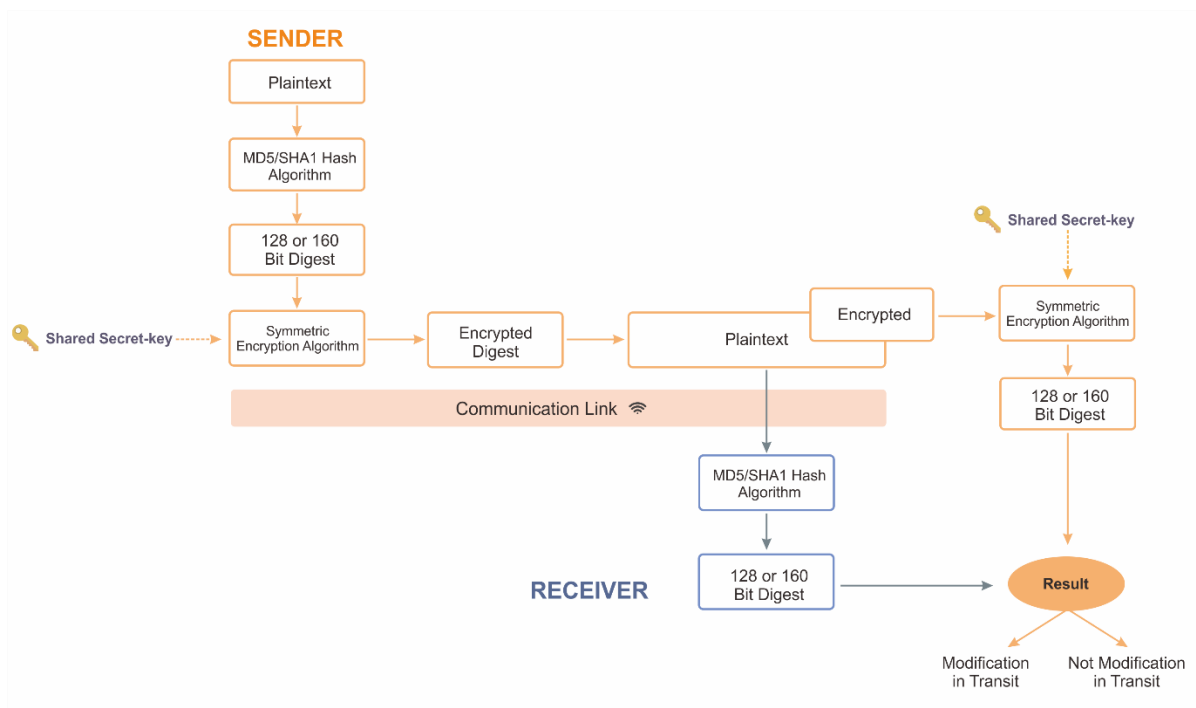
Symmetric Encryption Algorithm

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 transmisi *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.

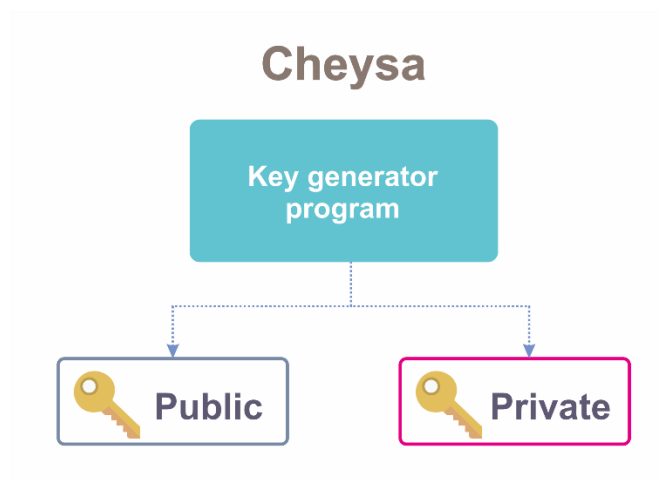


Gambar 64 MAC menggunakan Symmetric Encryption

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.

Assymmetric Cryptography

Assymmetric 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 *Assymmetric Encryption* untuk memproduksi *Public Key & Private Key* disebut dengan **Key Generation Program**.



Gambar 65 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.

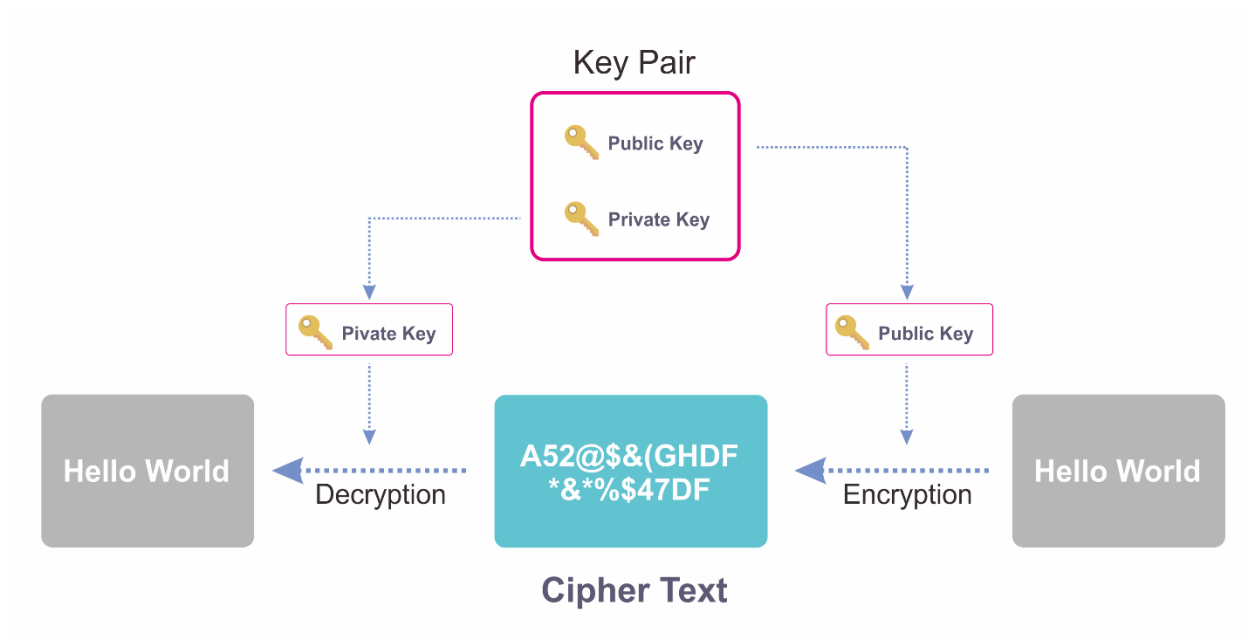
Private Key

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 66 Assymmetric Cryptography

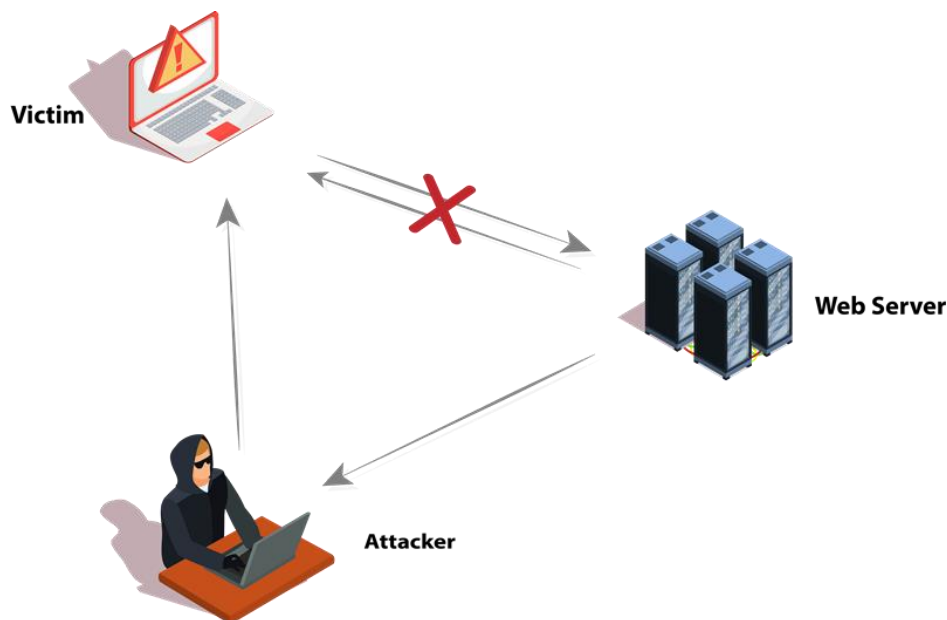
Public-key cryptography pertama kali dipublikasikan oleh Whitfield Diffie, Martin Hellman dan Ralph Merkle pada tahun 1976.

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 access point*. Seseorang di dalam satu jaringan dapat melakukan *packet-sniffing* untuk mendapatkan informasi sensitif, juga terdapat serangan lainnya 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 67 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 dikirimkannya 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

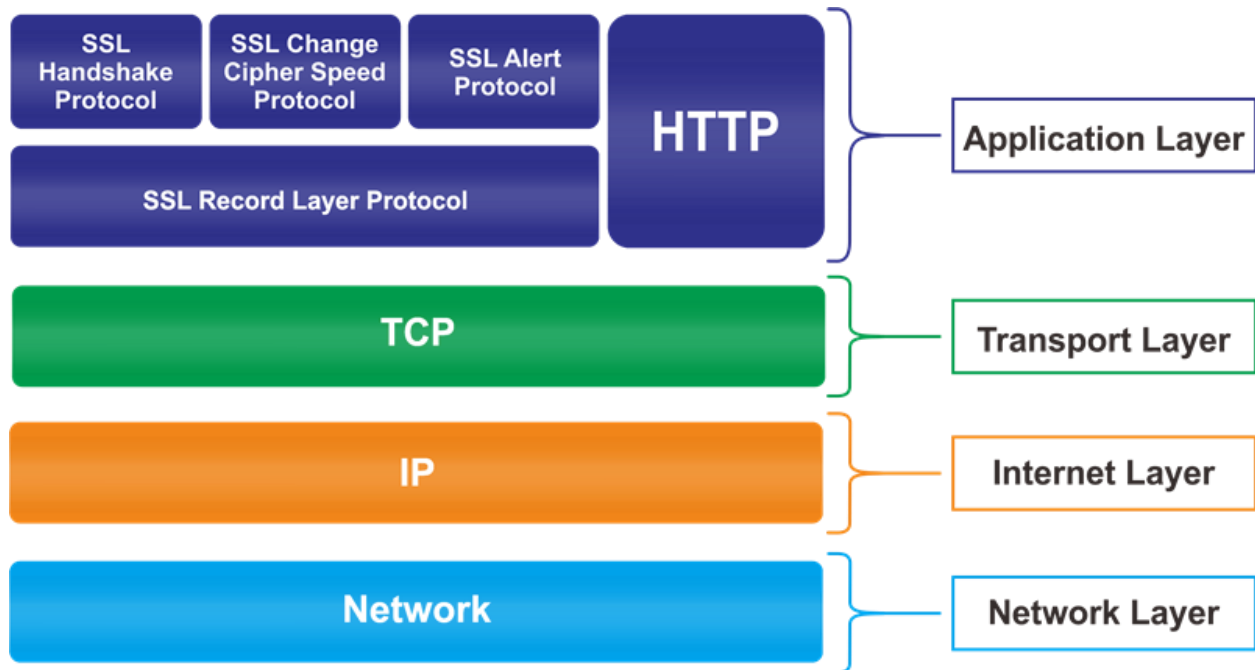
Sang *hacker* mengubah pesan yang dikirimkan oleh Gun Gun Febrianza atau Maudy Ayunda.

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 Transfer Protocol*), digunakan untuk **membangun komunikasi yang aman** antar jaringan komputer dan digunakan dalam internet.



Gambar 68 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* :



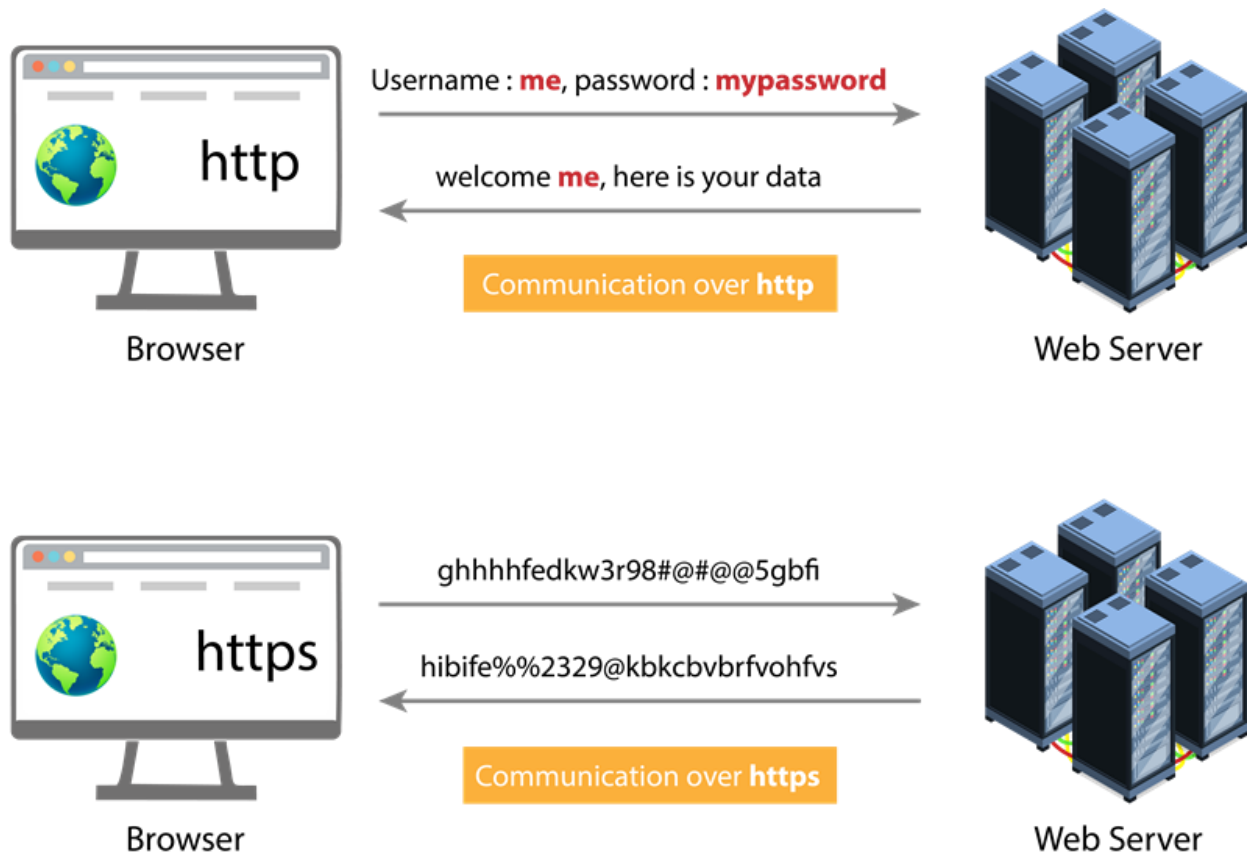
Gambar 69 HTTP



Gambar 70 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 71 Perbedaan Komunikasi HTTP & HTTPS

Tabel Perbedaan *HTTP* & *HTTPS*

HTTP	HTTPS
Data berupa <i>Hypertext format</i>	Data berupa <i>Encrypted format</i>
Secara <i>default</i> menggunakan <i>port</i> 80	Secara <i>default</i> menggunakan <i>port</i> 443
Data tidak diamankan	Diamankan dengan <i>TLS/SSL</i>
<i>Protocol</i> yang digunakan <i>http://</i>	<i>Protocol</i> yang digunakan <i>https://</i>

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 menggunakannya 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].

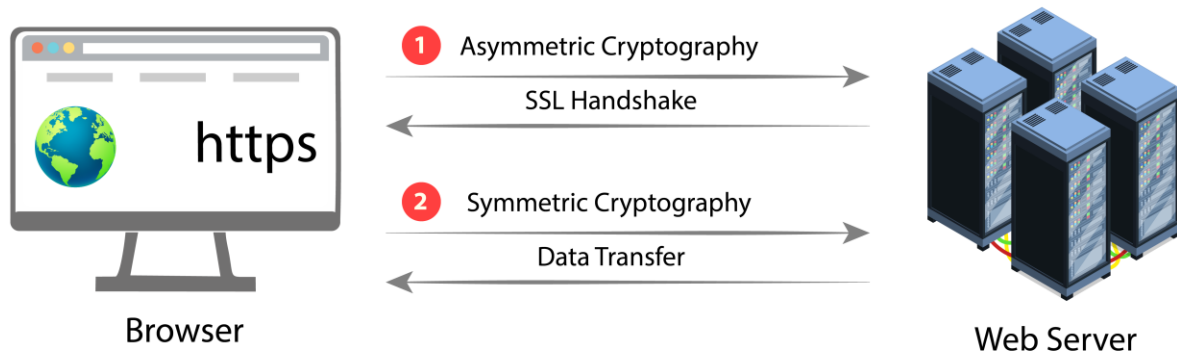
SSL/TLS protokol biasanya digunakan untuk *server* dan *client*, namun dapat juga digunakan untuk *server* ke *server* dan *client* ke *client* jika dibutuhkan. Data pribadi yang sensitif dan data mengenai keuangan dapat dilindungi dengan baik. Maka dari itu migrasi dari

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 :



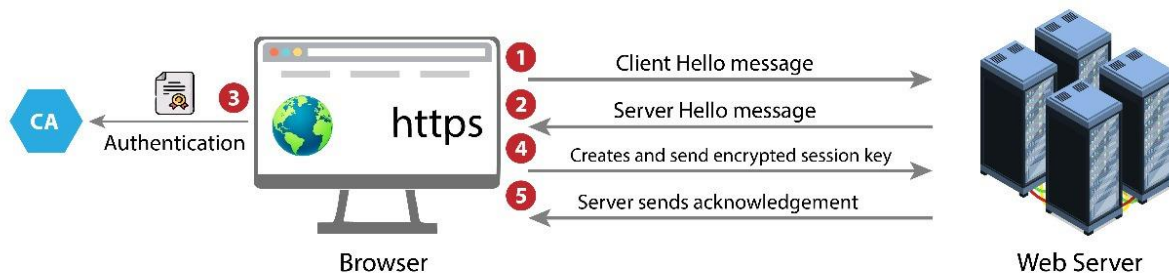
Gambar 72 SSL Communication

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 73 SSL Handshake in-depth

1. *Client* mengirimkan pesan "*client hello*", *SSL version number*, *cipher settings*, *session-specific data* dan informasi penting lainnya 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 lainnya 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 mengirimkannya 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.*

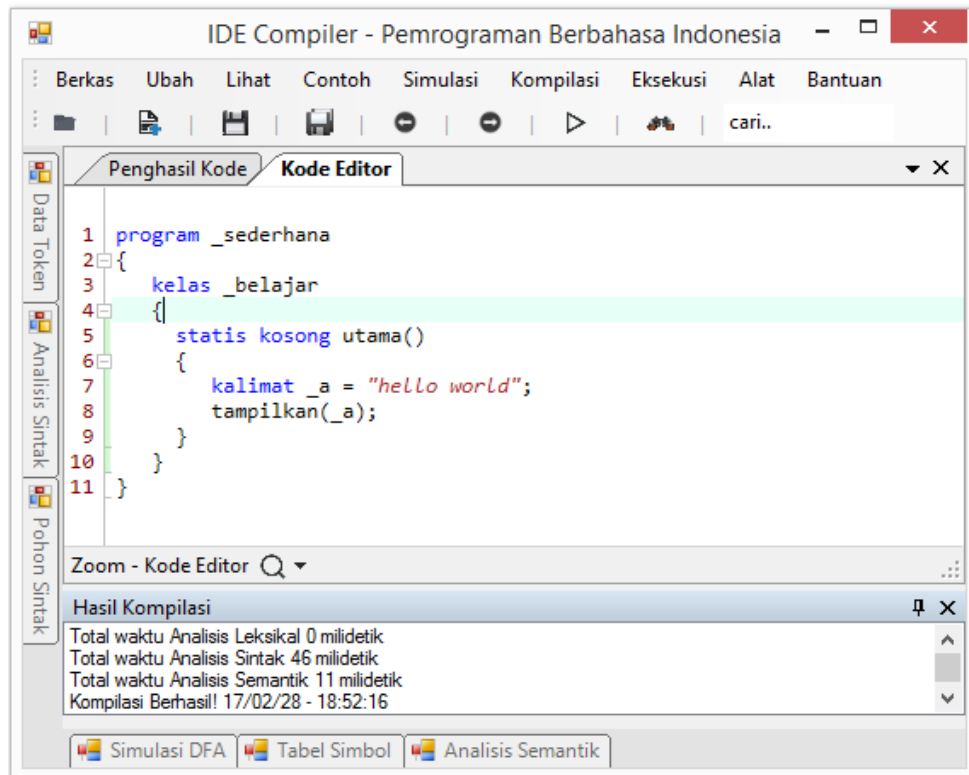
— Martin Fowler

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 74 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 populer 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 *desktop environment*. Sederhananya, *Electron.js* adalah *framework* yang dapat membuat aplikasi *web* menjadi aplikasi *desktop* 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 mengembangkannya atau memodifikasinya untuk keperluan anda sendiri. Anda perlu memahami **Typescript** dan CSS jika ingin memodifikasi *source code* dari *visual studio code*.

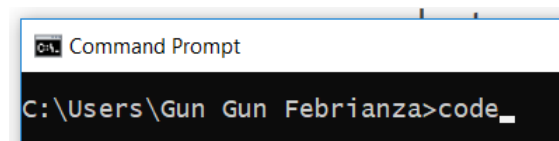
Link Project Visual Studio Code :

<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 :



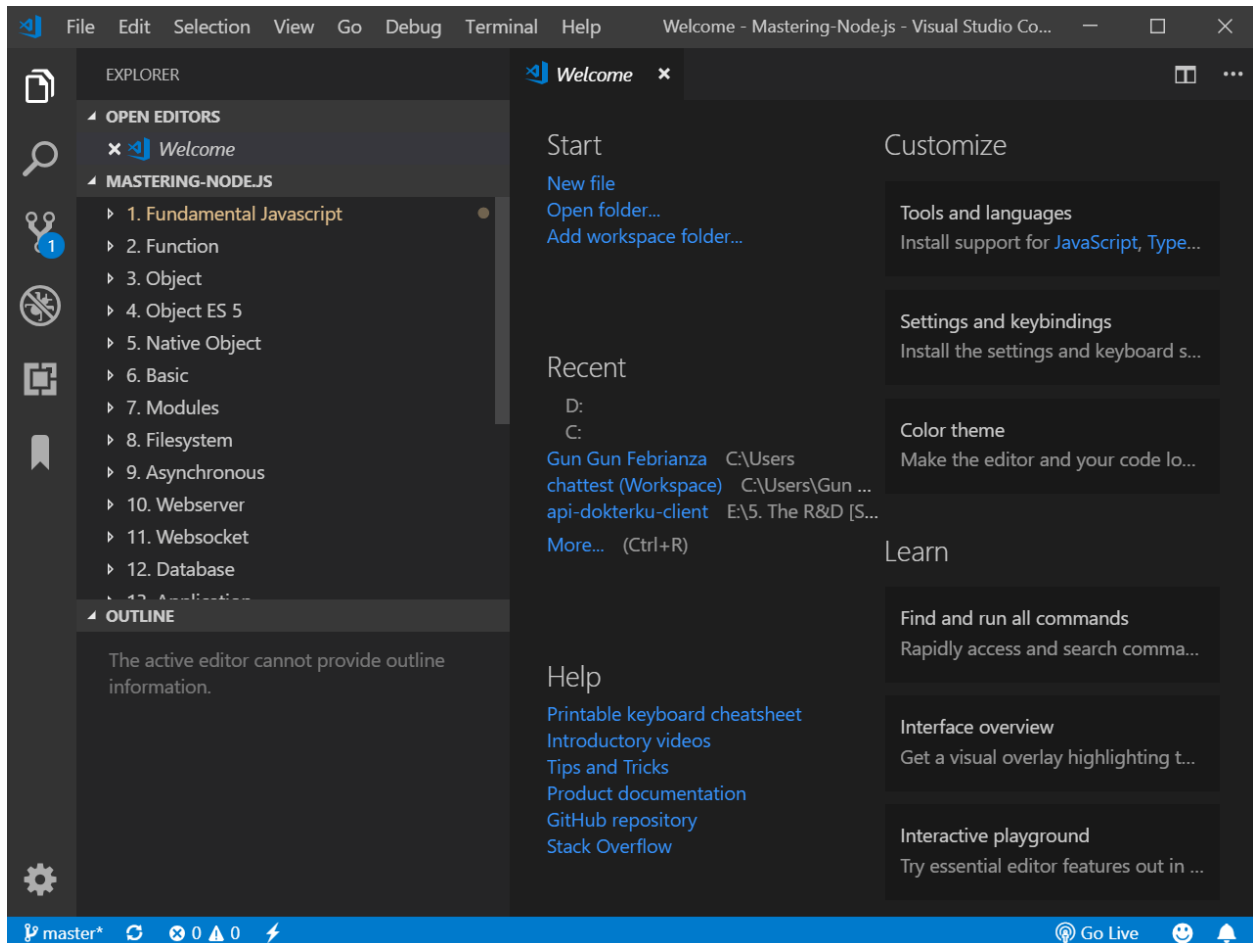
Gambar 75 Command Prompt

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 76 Command Prompt Part II

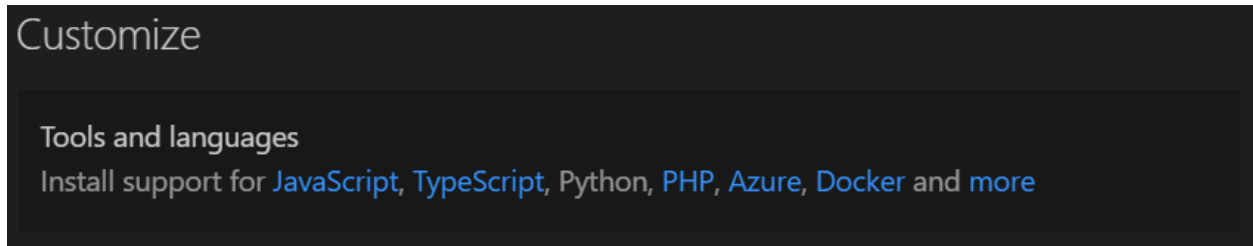
Di bawah ini adalah tampilan *user-interface* dari *viscode* :



Gambar 77 Visual Studio Code Interface

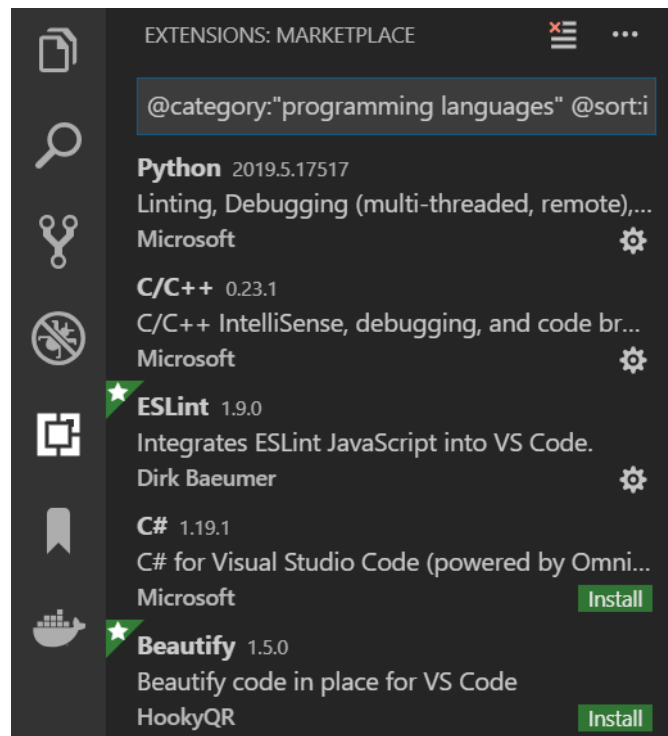
1. Install Programming Language Support

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 78 Install Tools & language

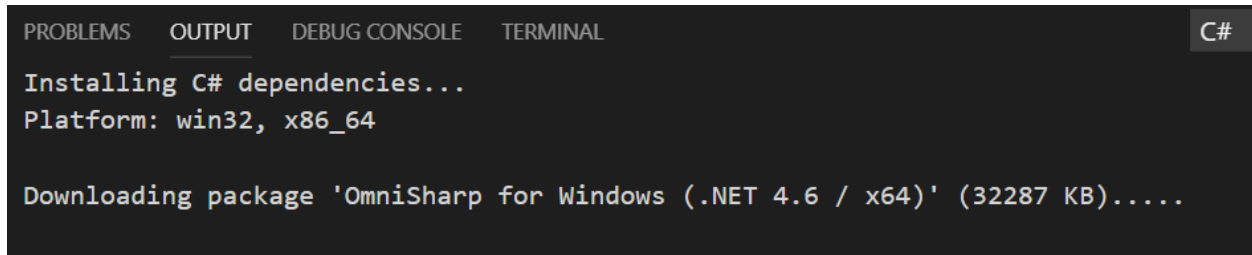
Pada menu sebelah kiri akan muncul kolom pencarian seperti gambar di bawah ini :



Gambar 79 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#* :

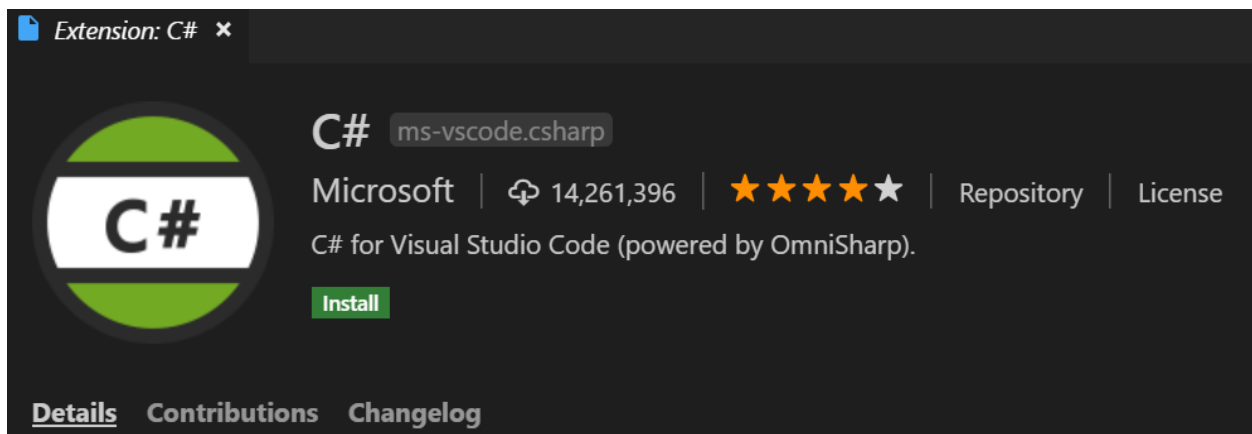


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  C#
Installing C# dependencies...
Platform: win32, x86_64

Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)' (32287 KB).....
```

Gambar 80 Instalasi C# Language

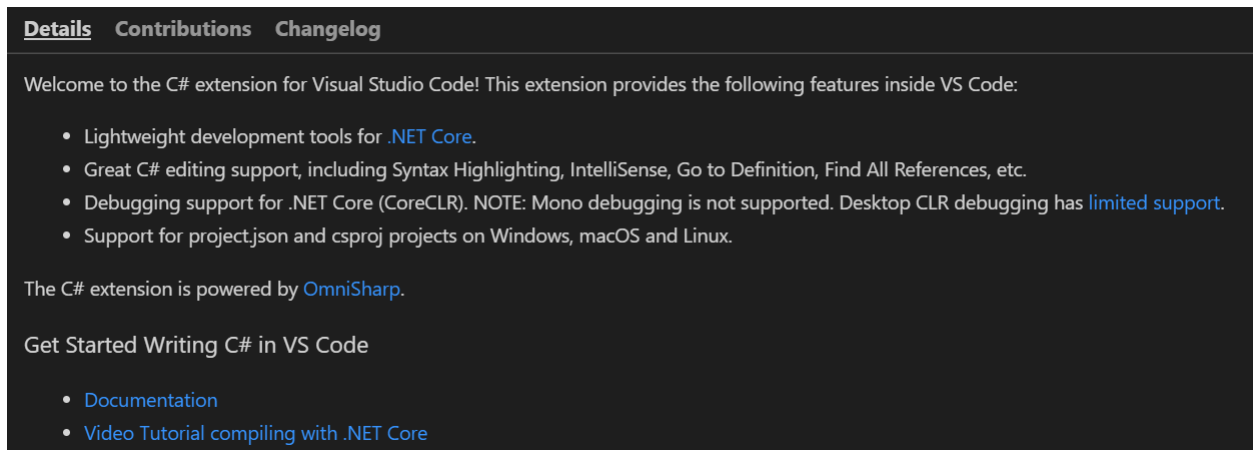
Pada kolom *output* kita akan melihat informasi *download package* yang kita butuhkan untuk dapat menggunakan bahasa pemrograman *C#* dalam *viscode*.



Gambar 81 C# Information

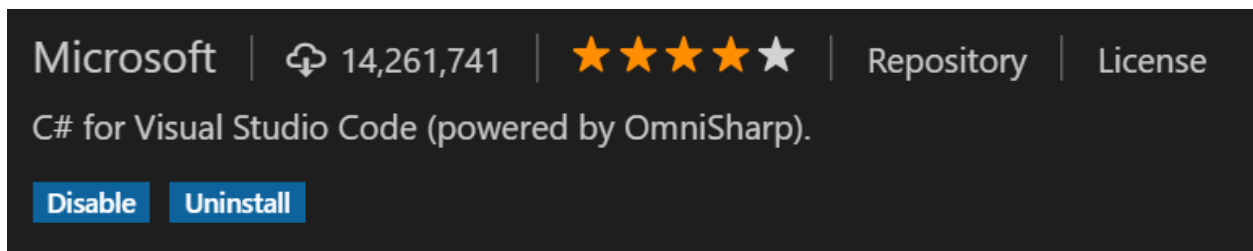
Maka akan muncul informasi mengenai bahasa pemrograman yang akan kita *install*, seperti jumlah *download*, *rating*, *repository* dan *license*. Semakin besar jumlah *rating* dan pengguna yang melakukan instalasi maka dipastikan kualitasnya bagus.

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 82 C# Detail Information

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

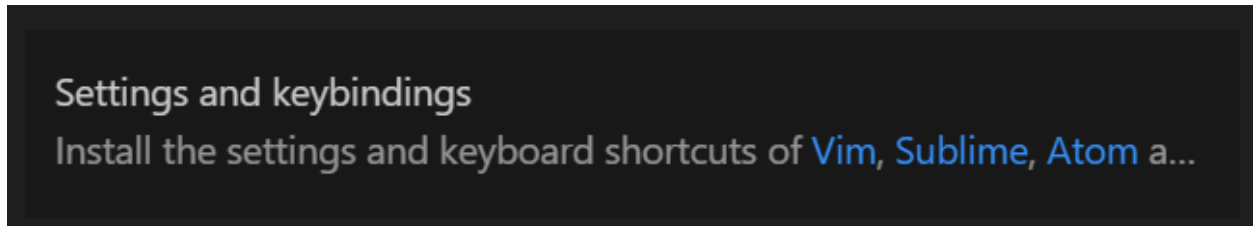


Gambar 83 Success Installed

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

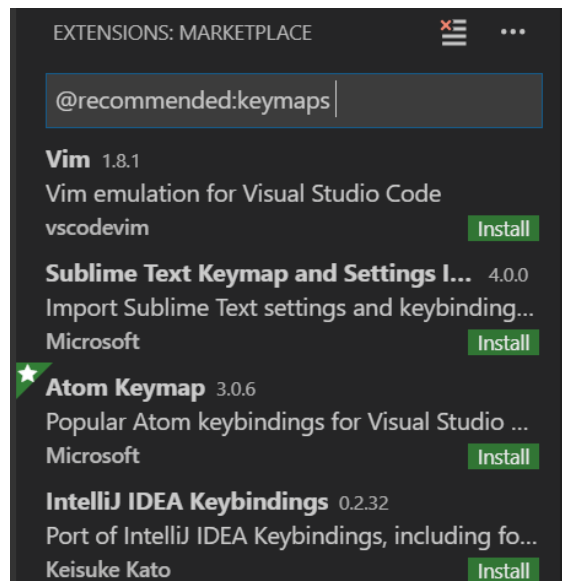
2. Install Keybinding

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 84 Key Bindings

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



Gambar 85 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>



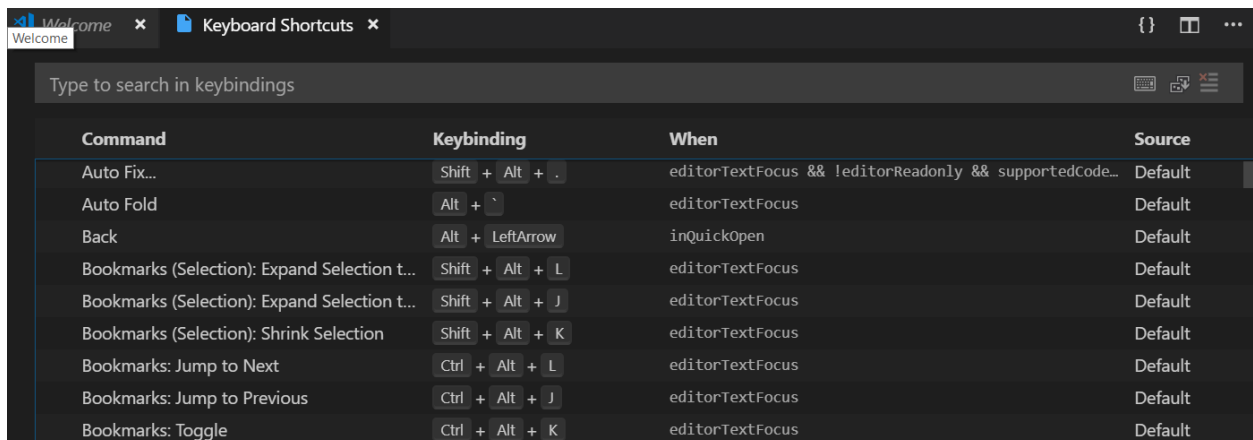
Keyboard shortcuts for Windows

General

Ctrl+Shift+P, F1	Show Command Palette
Ctrl+P	Quick Open, Go to File...
Ctrl+Shift+N	New window/instance
Ctrl+Shift+W	Close window/instance
Ctrl+,	User Settings
Ctrl+K Ctrl+S	Keyboard Shortcuts

Gambar 86 Visual Studio Code Keyboard Shortcut

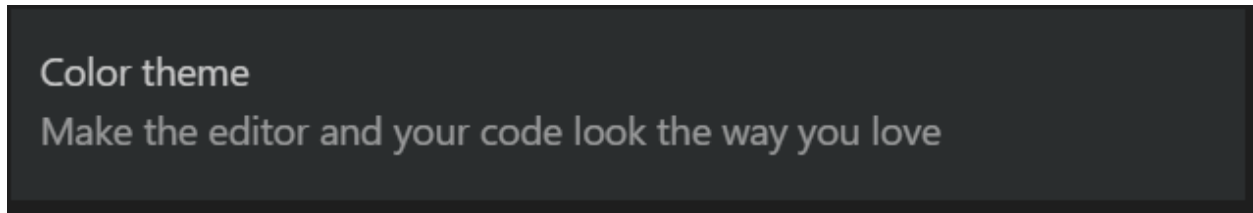
Jika anda ingin mengubah *keyboard shortcut* silahkan memilih menu **File** → **Preferences** → **Keyboard Shortcuts** atau tekan tombol **CTRL+K** kemudian **CTRL+S**.



Gambar 87 Default Keyboard Shortcut

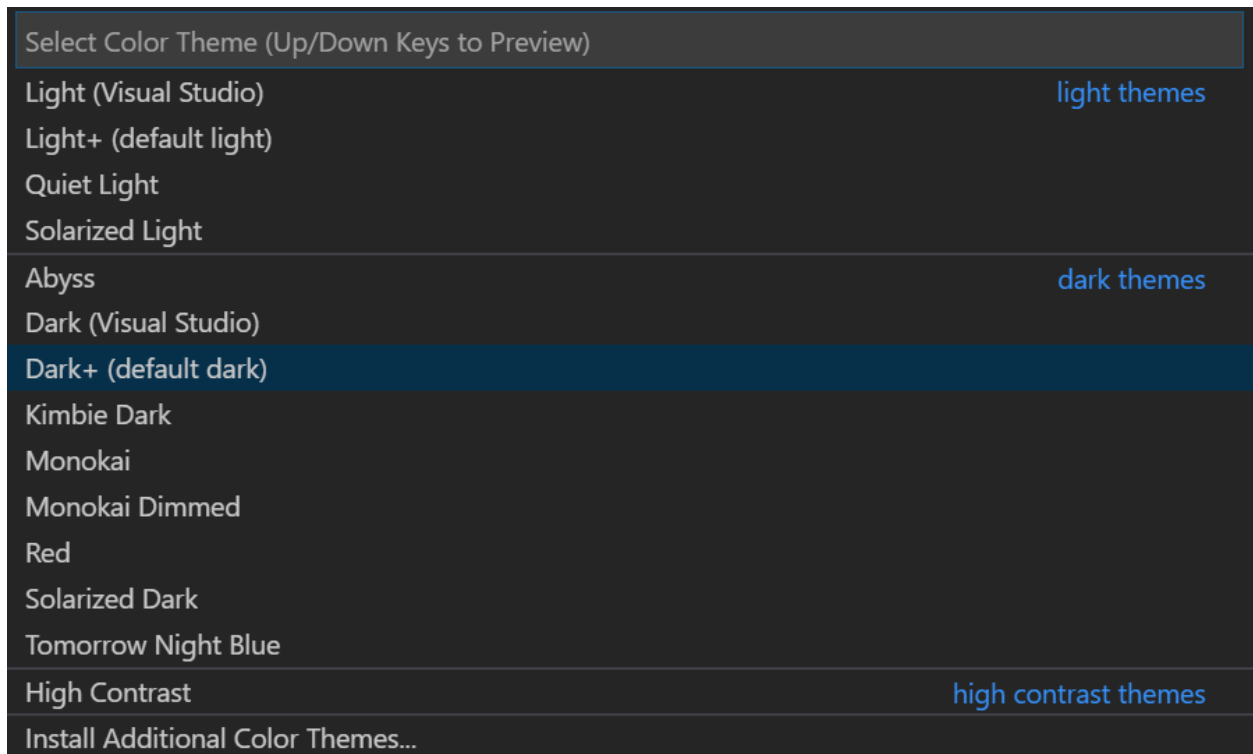
3. Install & Change Theme Editor

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 88 Color Theme Menu

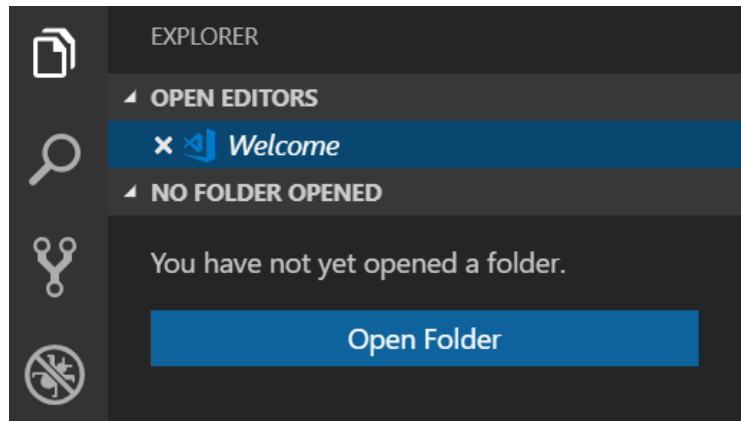
Jika anda kurang puas dengan *theme* yang tersedia anda bisa mencari *theme* lainnya dengan memilih menu paling bawah **Install Additional Color Themes**.



Gambar 89 Install New Themes

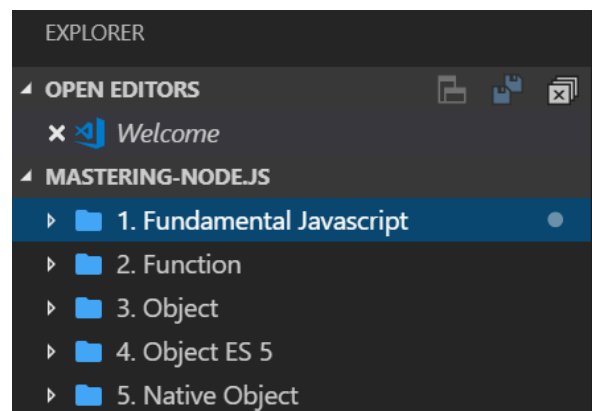
4. The File Explorer

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



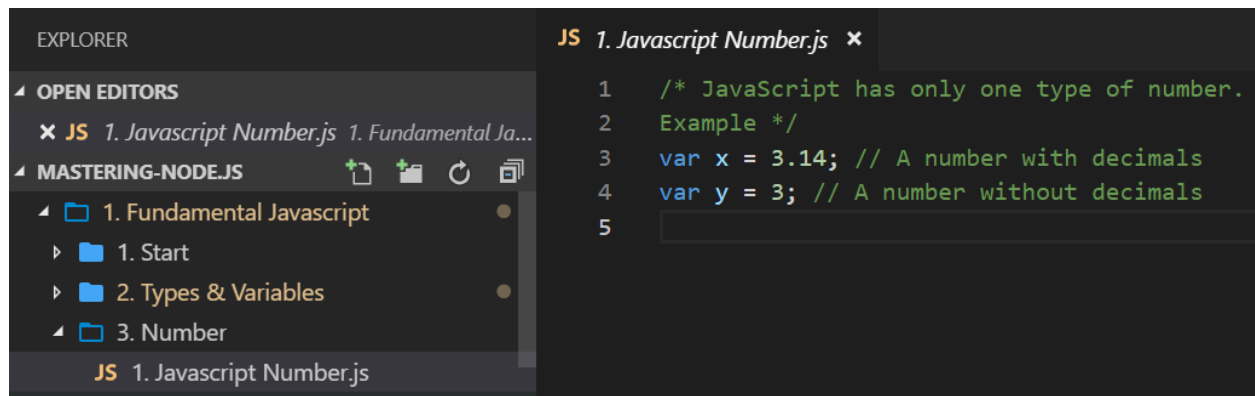
Gambar 90 File Explorer

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



Gambar 91 Display Folder in File Explorer

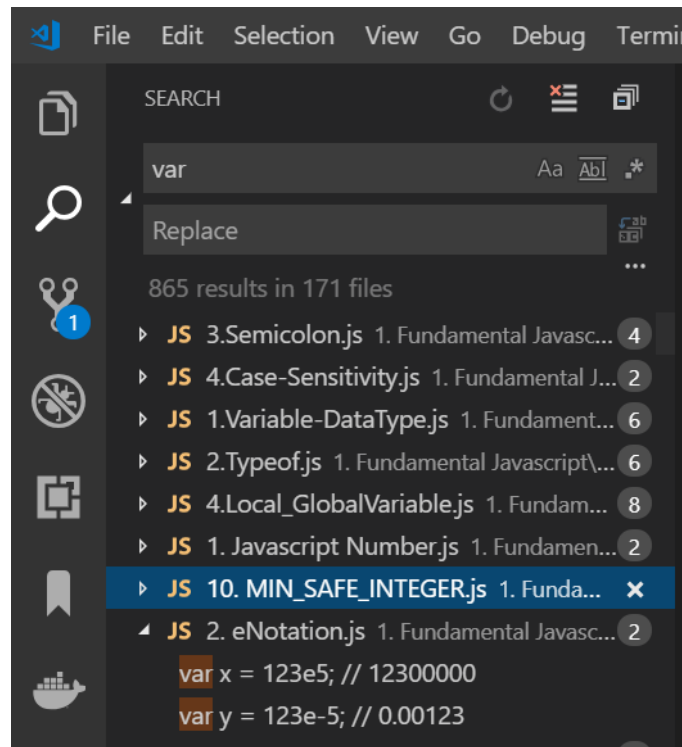
Jika kita ingin membuka salah satu *file* ke dalam *code editor*, klik *file* tersebut :



Gambar 92 Display File in The Code Editor

5. Search Feature

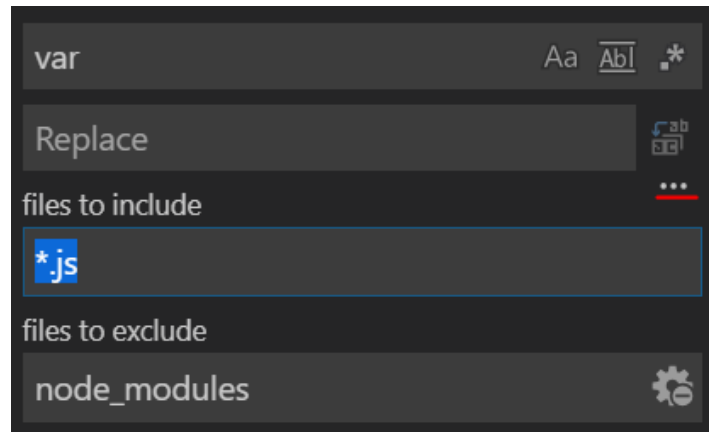
Viscode memiliki fitur yang bisa kita gunakan untuk mencari suatu *string* dalam *files* proyek yang kita muat. Klik gambar kaca pembesar, kemudian masukan *string* yang ingin kita cari. Pada kasus ini penulis memasukan *keyword* **var** :



Gambar 93 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.

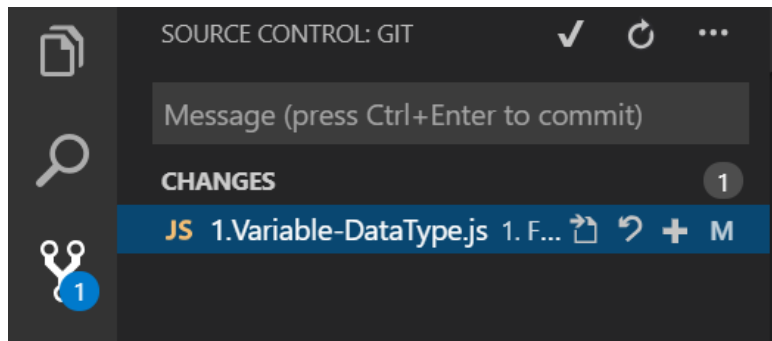


Gambar 94 Include & Exclude on Search String

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**.

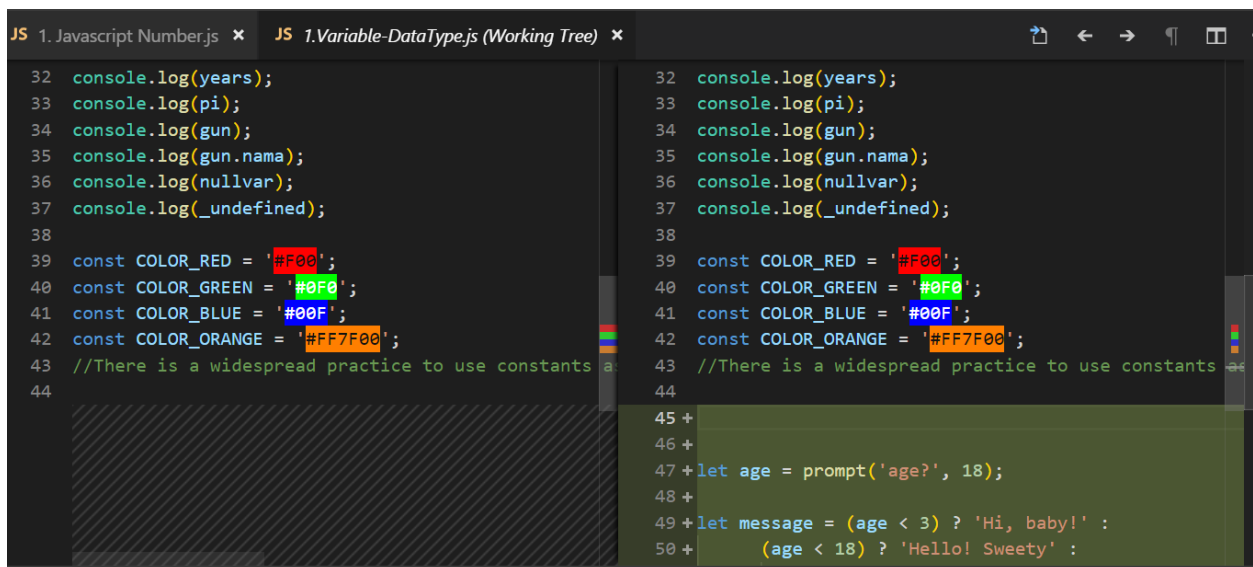
6. Source Control

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 95 Source Control

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



Gambar 96 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*.

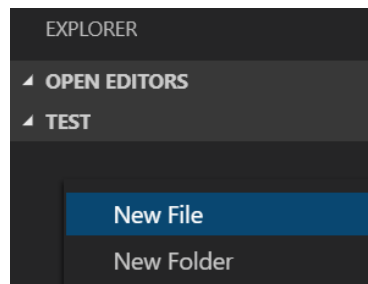
8. Extension

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



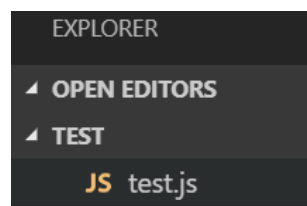
Gambar 97 Create New Folder

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 98 Create New File

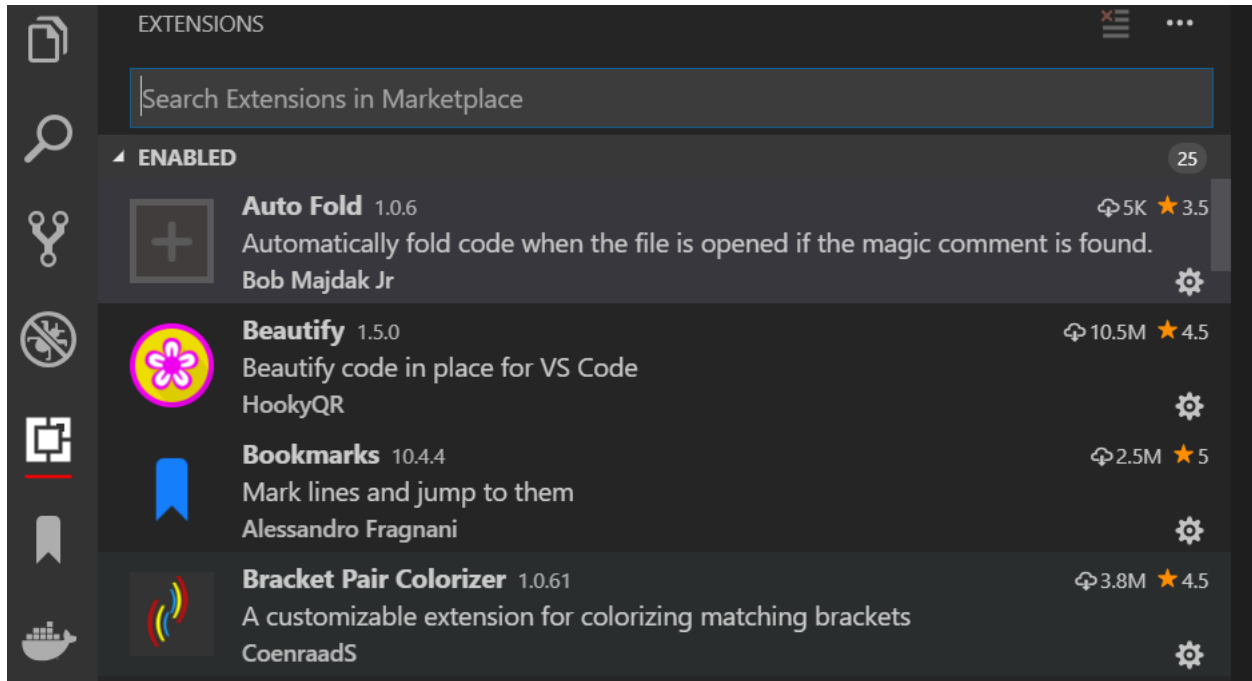
Biarkan **file** tersebut berisi kode kosong :



Gambar 99 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 maksimal dengan *tools* yang telah disediakan dan dikembangkan oleh *expert* di komunitas *viscode*.

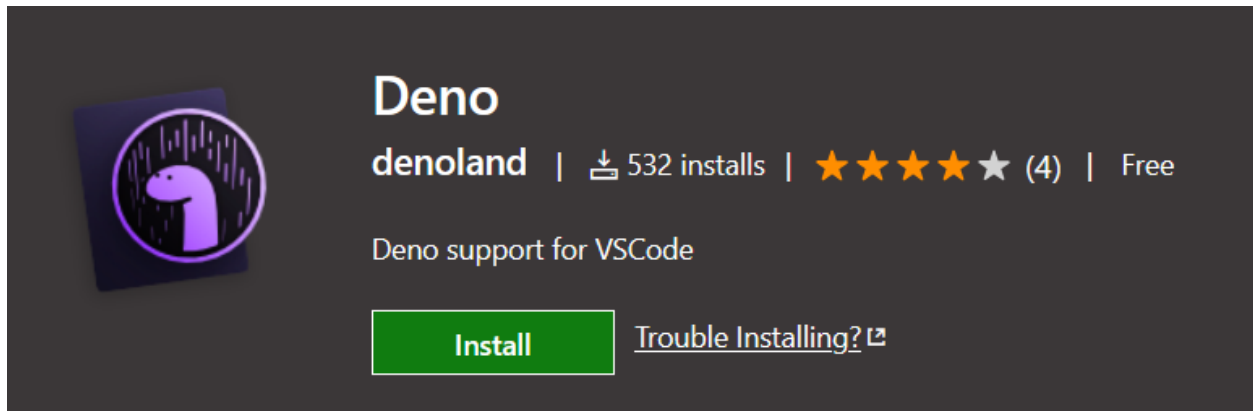


Gambar 100 Extension

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

Deno

Ekstensi ini berguna saat kita akan mengembangkan aplikasi **javascript** dan **typescript** pada **visual studio code**. Silahkan menggunakan **Deno Official Extension** yang telah disediakan.

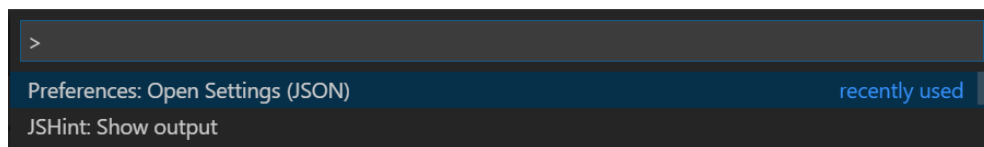


Gambar 101 Add Deno Support

Auto Fold

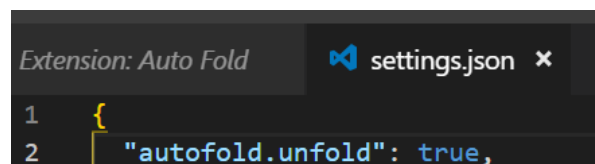
Ekstensi ini berguna agar *viscode* langsung membuka seluruh baris kode yang tertutup oleh *bracket* {...}, sehingga anda tidak perlu membukanya secara manual.

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



Gambar 102 Viscode Settings

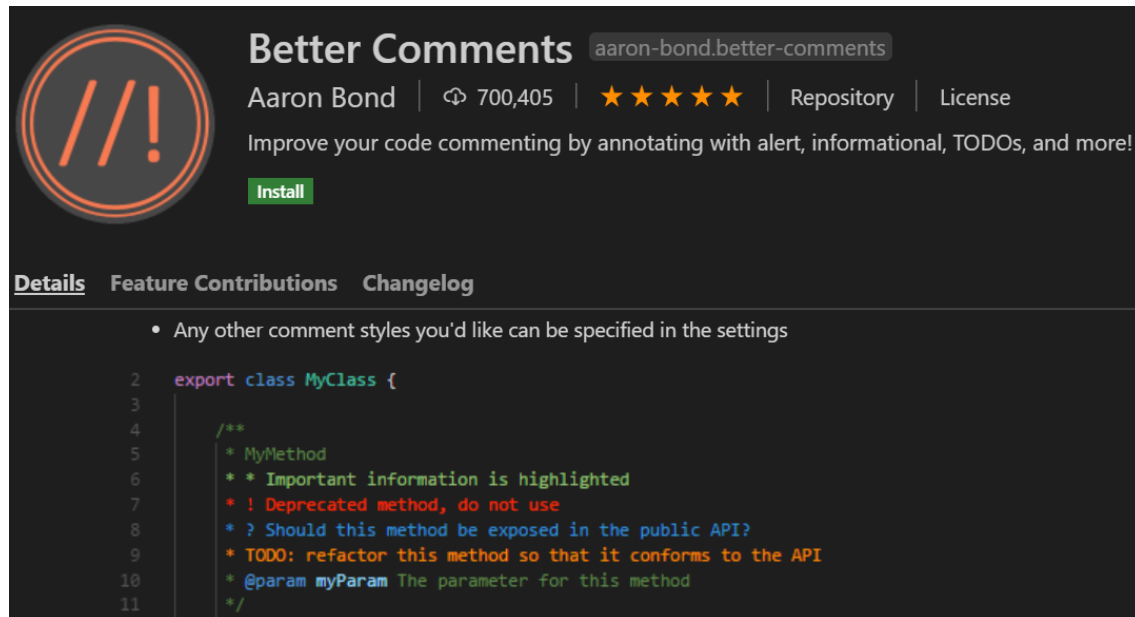
Pada **setting.json** masukan kode **"autofold.unfold": true**, agar ekstensi *auto fold* yang kita gunakan bekerja. Perhatikan gambar di bawah ini :



Gambar 103 Autofold Configuration

Better Comment

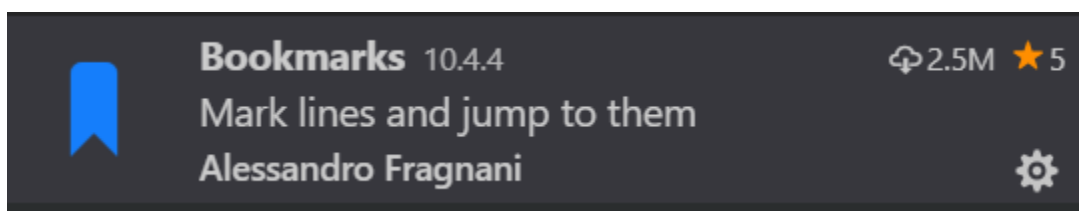
Ekstensi **better comment** membantu kita membedakan jenis komentar dengan warna menarik yang mudah diingat.



Gambar 104 Better Comment Extension

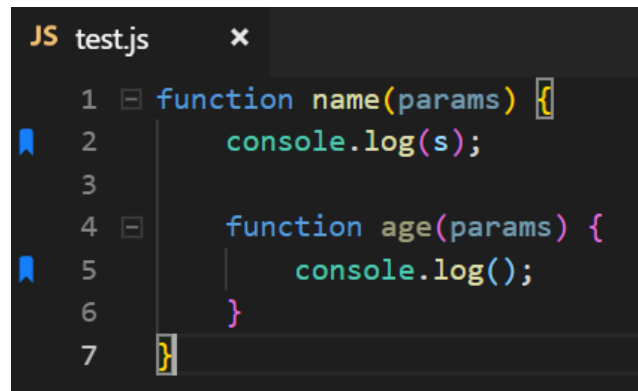
Bookmarks

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.



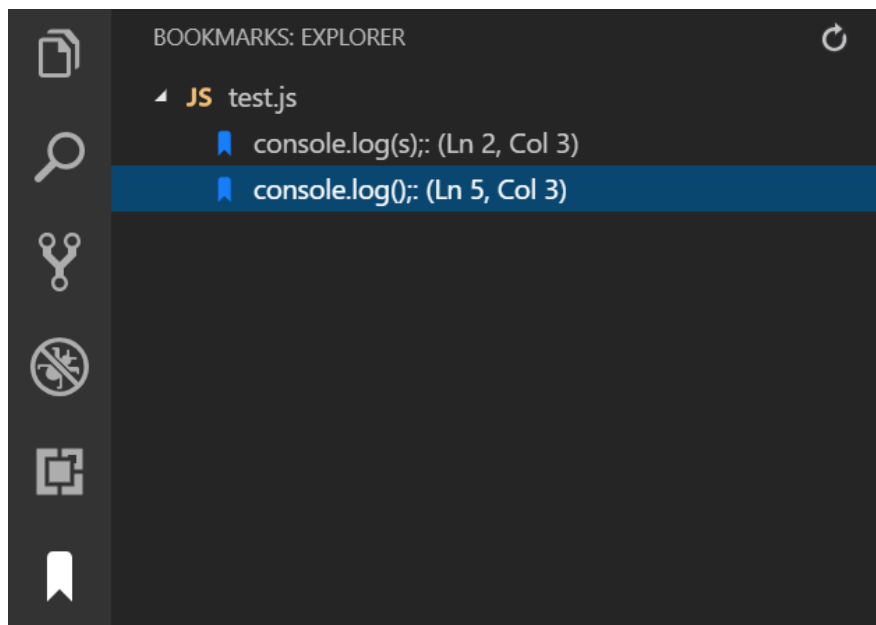
Gambar 105 Bookmarks Extension

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



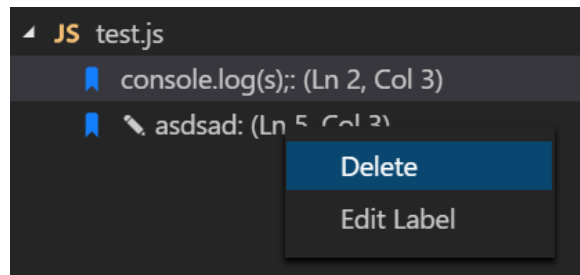
Gambar 106 Bookmarked Codes

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 107 Bookmark Explorer

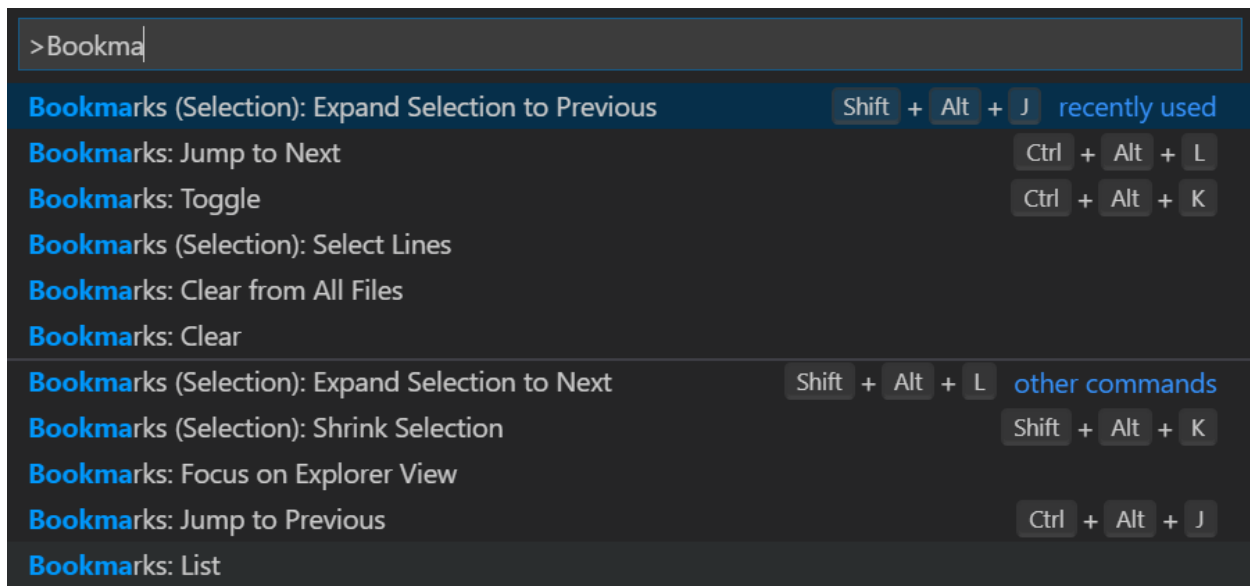
Pada *bookmark explorer* kita bisa melihat *file* apa saja dan alamat kode mana saja yang telah kita *bookmark*. Untuk mempermudah memahami kode yang telah kita *bookmark* kitapun bisa memberikan *label* pada kode yang telah kita *bookmark*.



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*.

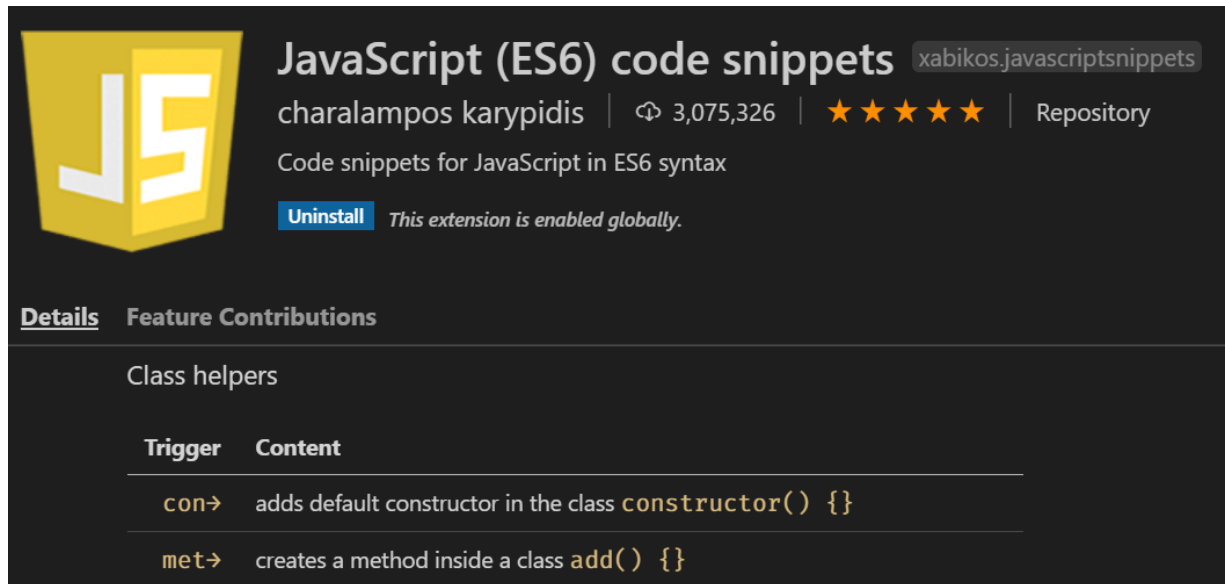
Untuk mengetahui ada fitur *bookmark* apa saja, tekan tombol **F1** kemudian ketikkan **bookmarks** seperti pada gambar di bawah ini :



Gambar 108 *Bookmark – List Commands*

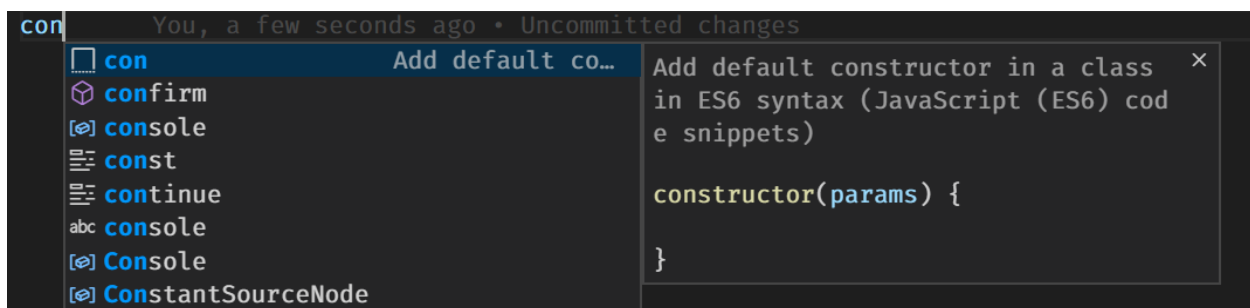
Javascript (ES6) Code Snippets

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



Gambar 109 Javascript ES6 Snippets

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



Gambar 110 Constructor Snippets

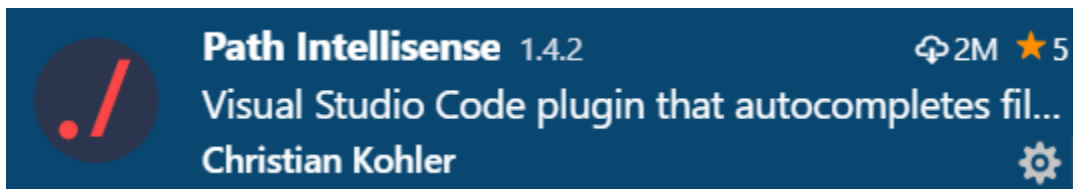
Jika kita enter maka kode akan langsung dibantu dibuat :

```
constructor(params) {  
    
}
```

Gambar 111 Generated Snippet

Path Intellisense

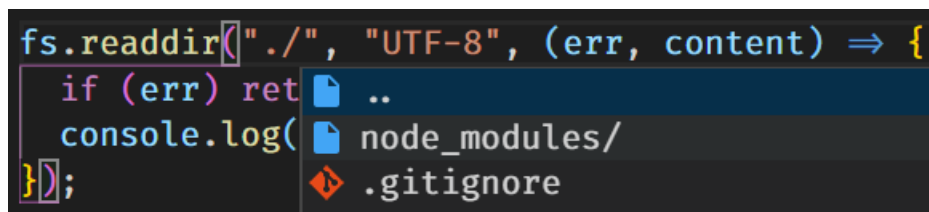
Ekstensi ini membantu kita saat kita berhadapan dengan *path*, ekstensi ini akan memberikan fitur **autocomplete** ketika ingin mengeksplorasi suatu *path* di dalam *code editor*.



Gambar 112 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 :

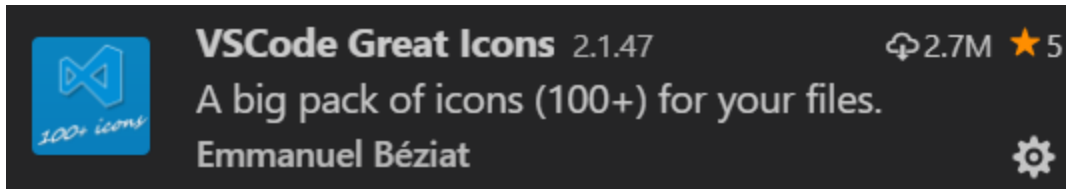
```
fs.readdir("./", "UTF-8", (err, content) => {  
  if (err) ret  
  console.log(  
});  
}
```



Gambar 113 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.



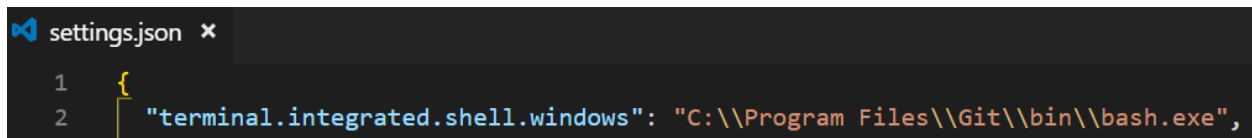
Gambar 114 VSCode Great Icons

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** :



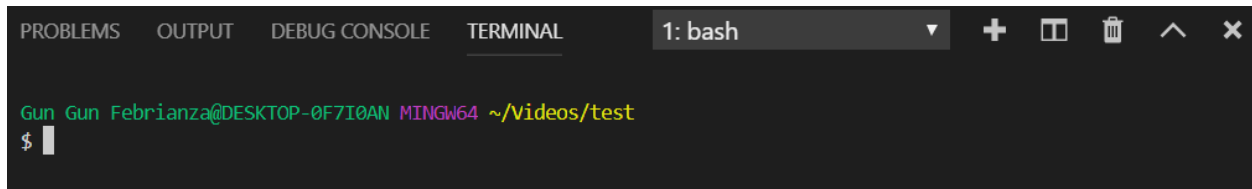
```
settings.json x
1 {
2   "terminal.integrated.shell.windows": "C:\\Program Files\\Git\\bin\\bash.exe",
```

Gambar 115 Shell Integration

Notes

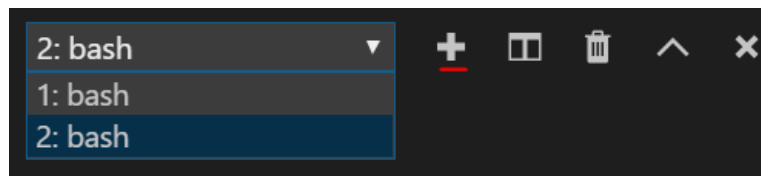
Pastikan anda sudah melakukan instalasi Git !

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



Gambar 116 Terminal Visual Studio Code

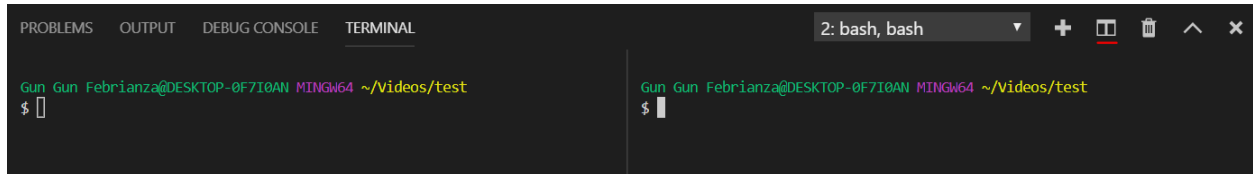
Menambah *Terminal* Baru



Gambar 117 Add New Terminal

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

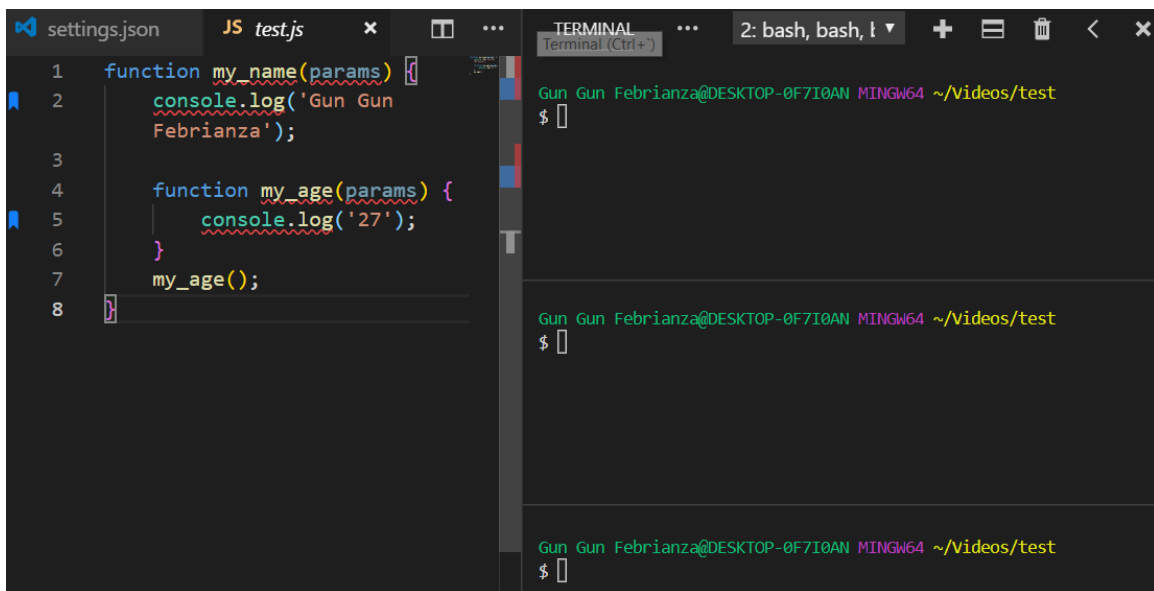
Melakukan *Split Terminal*



Gambar 118 Add New Terminal

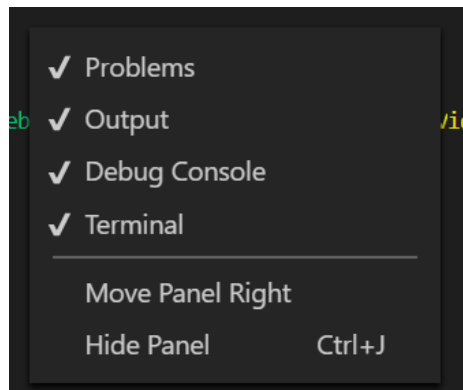
Bagi yang memiliki layar dengan lebar yang sangat panjang fitur ini sangat membantu. Sehingga kita tidak perlu menggunakan **drop down** untuk mengganti *channel terminal*.

Mengubah Posisi Terminal



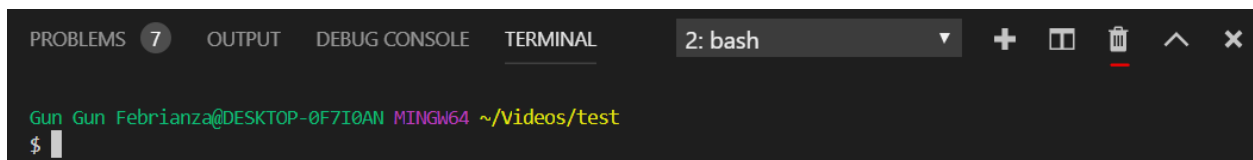
Gambar 119 Change Terminal Position

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 120 Move Panel

Menghapus Terminal

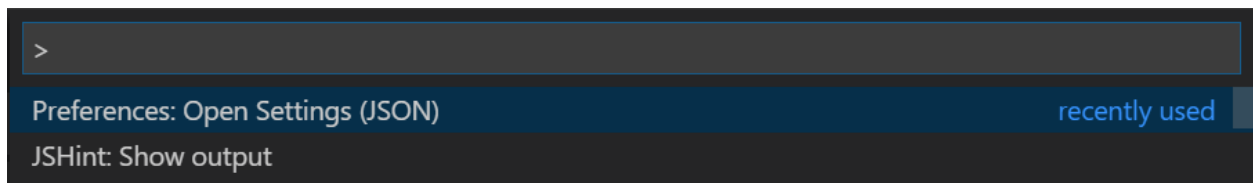


Gambar 121 Delete Terminal

10. Performance Optimization

Jika kita melakukan pengembangan aplikasi *nodejs*, 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 :



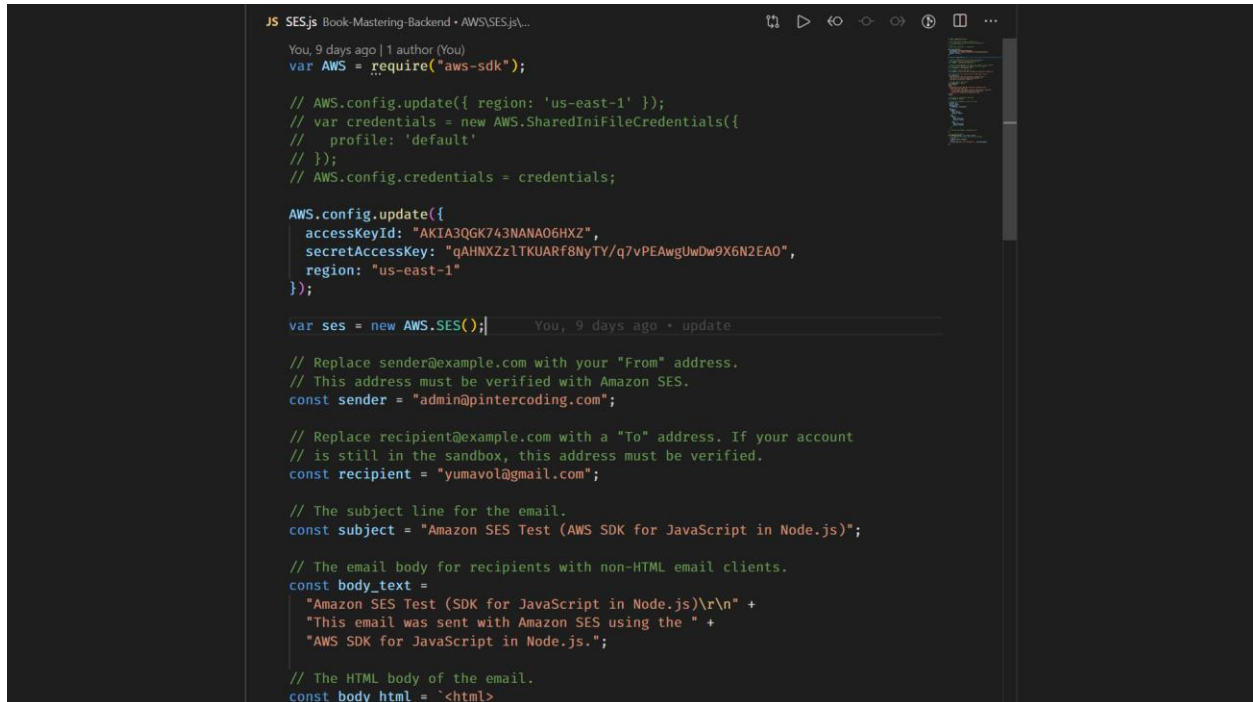
Gambar 122 Viscode Settings

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** :

A screenshot of a code editor in Zen Mode. The editor has a dark background and shows a JavaScript file named 'SES.js'. The code is for sending an email using the AWS SDK for JavaScript. It includes comments and code for setting up AWS credentials, creating an SES client, and sending an email. The code is as follows:

```
JS SES.js Book-Mastering-Backend • AWSSES.js...
You, 9 days ago | 1 author (You)
var AWS = require("aws-sdk");

// AWS.config.update({ region: 'us-east-1' });
// var credentials = new AWS.SharedIniFileCredentials({
//   profile: 'default'
// });
// AWS.config.credentials = credentials;

AWS.config.update({
  accessKeyId: "AKIA3QGK743NANAO6HXZ",
  secretAccessKey: "qAHNXZz1TKUARf8NyTY/q7vPEAwgUwDw9X6N2EA0",
  region: "us-east-1"
});

var ses = new AWS.SES();

// Replace sender@example.com with your "From" address.
// This address must be verified with Amazon SES.
const sender = "admin@pintercoding.com";

// Replace recipient@example.com with a "To" address. If your account
// is still in the sandbox, this address must be verified.
const recipient = "yumavol@gmail.com";

// The subject line for the email.
const subject = "Amazon SES Test (AWS SDK for JavaScript in Node.js)";

// The email body for recipients with non-HTML email clients.
const body_text =
  "Amazon SES Test (SDK for JavaScript in Node.js)\r\n" +
  "This email was sent with Amazon SES using the " +
  "AWS SDK for JavaScript in Node.js.";

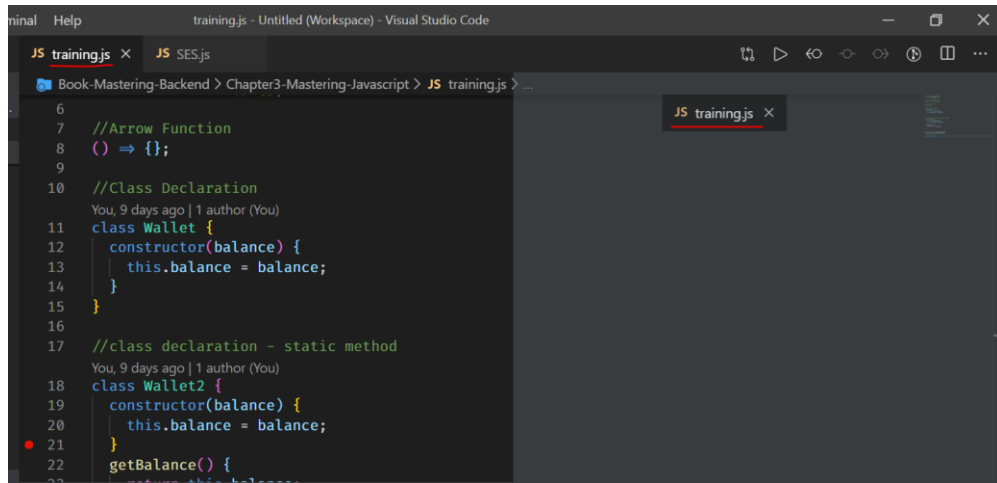
// The HTML body of the email.
const body_html = `<html>
```

Gambar 123 Zen Mode

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

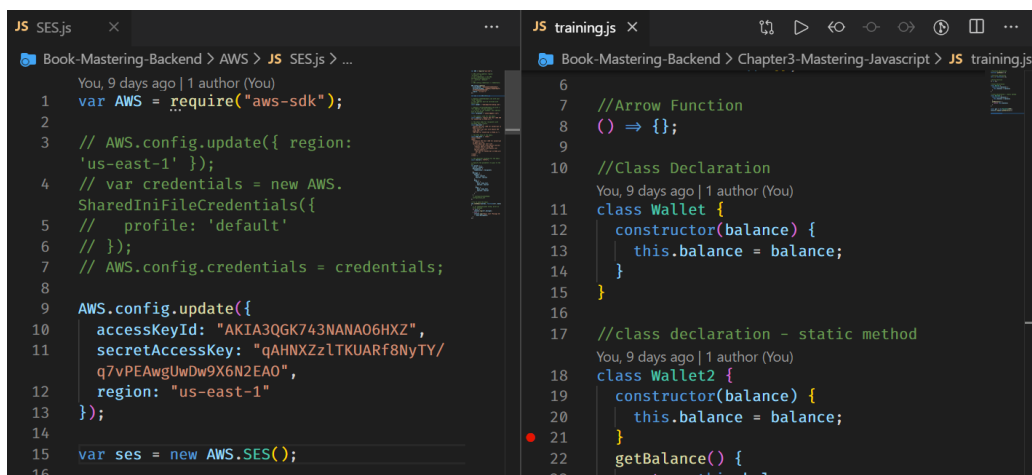
12. Display Multiple File

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 124 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 :



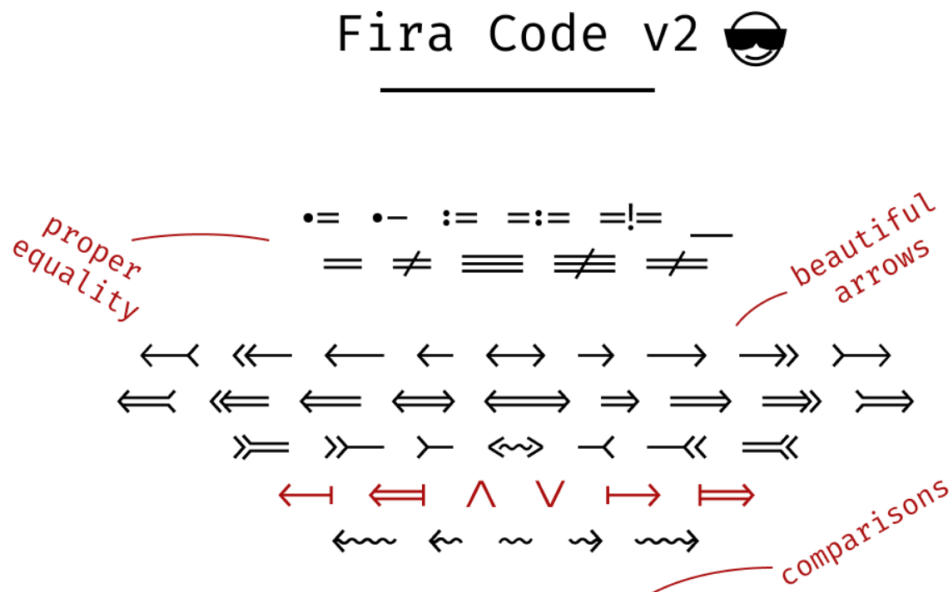
Gambar 125 Display Double File

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 :

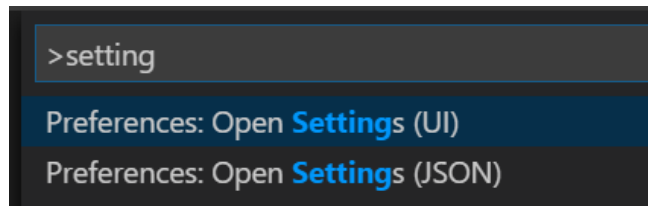


Gambar 126 Font Fira Code

Untuk cara instalasi Font dapat dibaca disini :

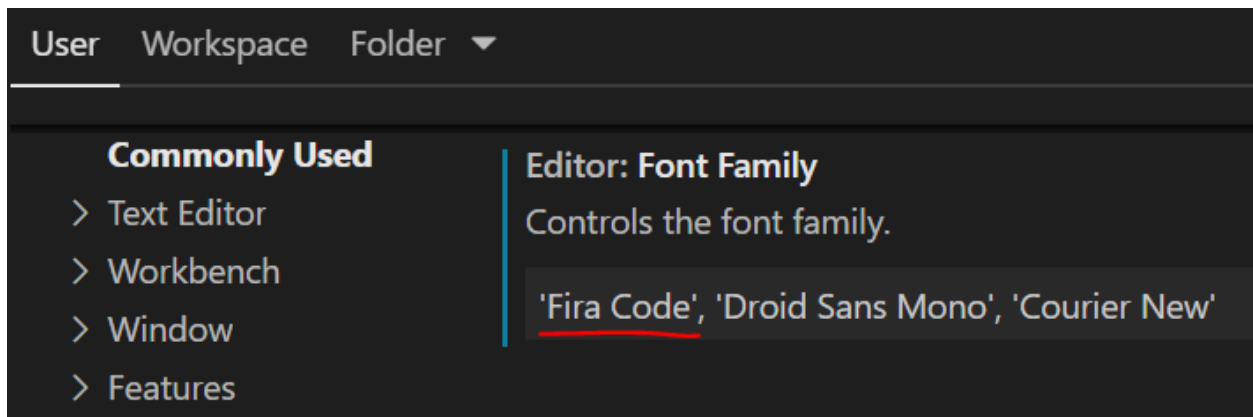
<https://github.com/tonsky/FiraCode/wiki/Installing>

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



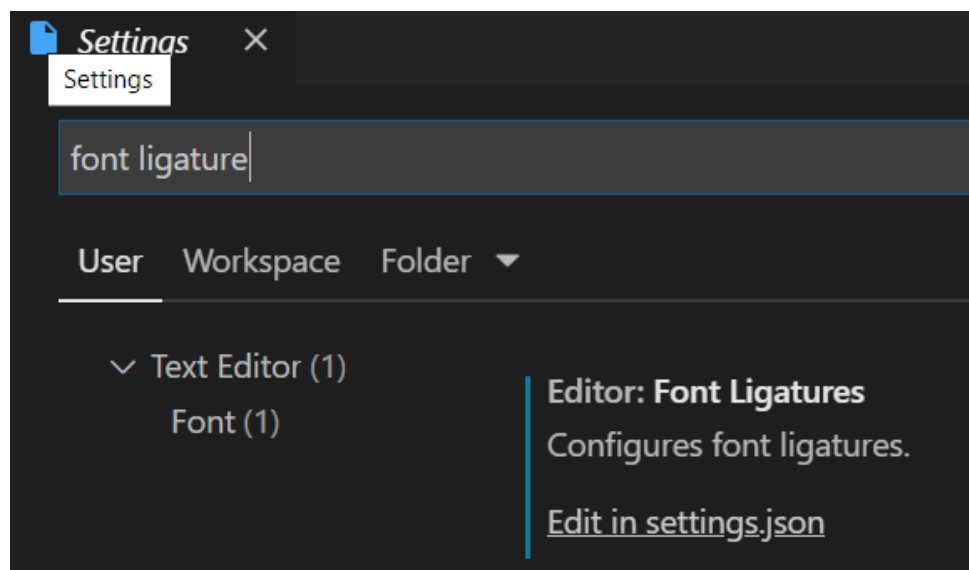
Gambar 127 Open Settings

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



Gambar 128 Add Fira Code

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



Gambar 129 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** :

```
if (true ≡ true) {  
  
} else if (false ≡ false) {  
  
} else if (10 ≥ 10) {  
  
}
```

Gambar 130 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

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

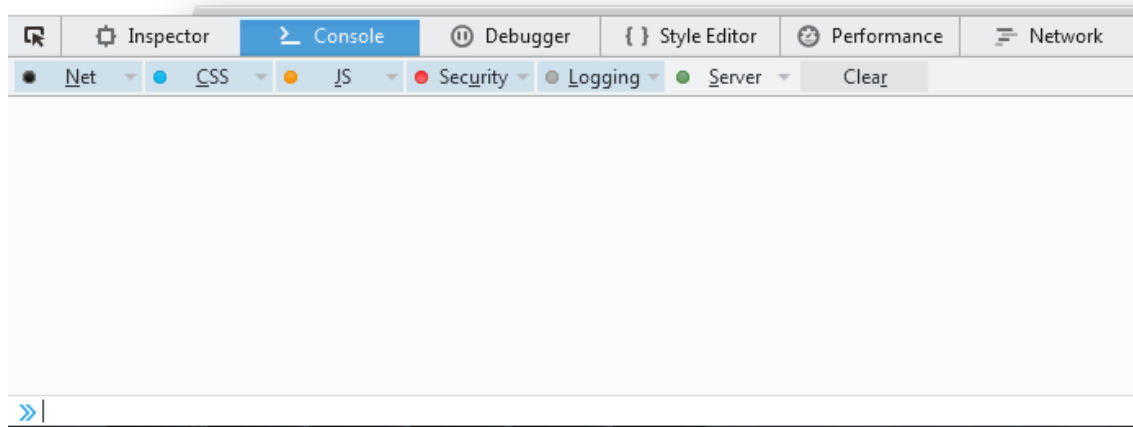
1. Web Browser

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

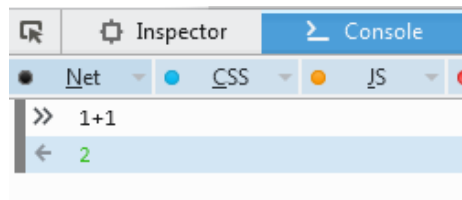
Dengan *web console* kita bisa mengetahui seluruh informasi yang terjadi di dalam suatu halaman *website*. Informasi yang dimaksud adalah *network request, javascript, css, security error* dan pesan peringatan lainnya yang bisa anda pelajari lebih lanjut.

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



Gambar 131 Web Console

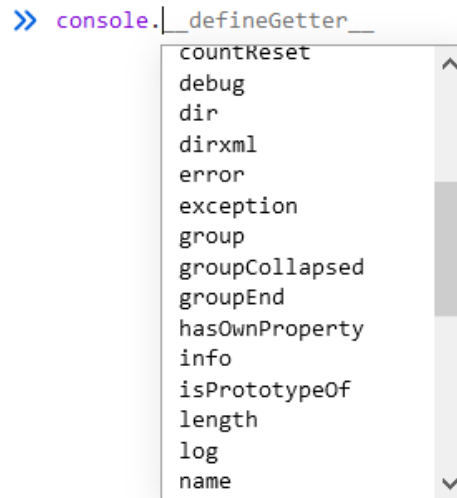
Untuk mengujinya coba ketik : **1 + 1** kemudian tekan *enter*. Hasilnya :



Gambar 132 Addition Operation

Autocomplete

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



Gambar 133 Autocomplete

Syntax Highlighting

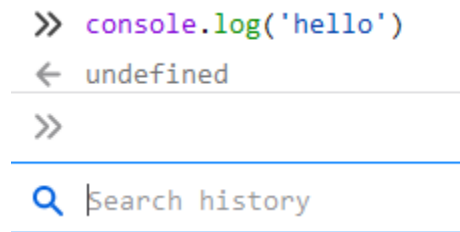
Web Console juga menyediakan fitur *syntax highlighting* untuk membantu kita dalam membaca kode *javascript* yang ditulis :

```
>> 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 134 Syntax Highlight Feature

Execution History

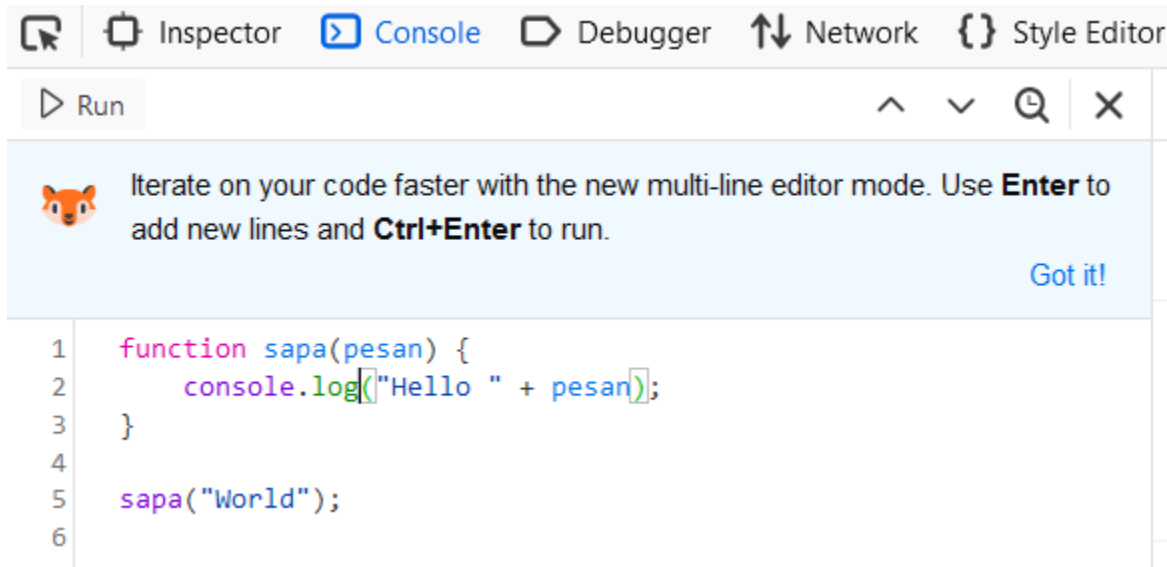
Setiap kali kita mengeksekusi suatu *statement*, *statement* tersebut akan dicatat di dalam *history*. Untuk mendapatkan *list statement* yang telah dieksekusi cukup tekan tombol *scroll* ke atas atau menekan tombol **F9**.



Gambar 135 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 136 Multi-line code editor pada Firefox

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



Gambar 137 Hello Word Pada Javascript

Chapter 3

Mastering Deno

Subchapter 1 – Install Deno

Great execution towards a terrible idea will get you nowhere.

— Sam Altman

Subchapter 1 – Objectives

- Mempelajari instalasi **Deno** pada **Windows**
 - Mempelajari instalasi **Deno** pada **Linux**
 - Mempelajari instalasi **Deno** pada **MacOS**
 - Cek **Deno Version** yang digunakan
-

1. Installation

Deno For Windows

Kita akan melakukan instalasi **Deno** menggunakan **Package Manager** yang disebut dengan **Chocolatey** :

Install Chocolatey Package Manager

Jalankan **Powershell** (**run as administrator**), kemudian eksekusi perintah di bawah ini :

```
Select Administrator: Windows PowerShell

PS C:\WINDOWS\system32> Get-ExecutionPolicy
Restricted
PS C:\WINDOWS\system32> Set-ExecutionPolicy AllSigned
```

Gambar 138 Powershell Initial Setup

Setelah itu eksekusi perintah **powershell** di bawah ini :

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

Proses instalasi sedang di lakukan, kita akan memiliki **directory Chocolatey Package Repository** di :

```
C:\ProgramData\chocolatey\lib
```

Kita juga akan memiki **directory** untuk **Chocolatey Binary** di :

```
C:\ProgramData\chocolatey\bin
```

Pastikan **chocolatey** sudah terpasang dengan cara mengeksekusi perintah di bawah ini dalam **cmd.exe** :

```
choco
```

Maka informasi versi **choco** akan di tampilkan :

```
Chocolatey v0.10.15
```

Install Deno Via Package Manager

Lakukan instalasi **Deno** menggunakan perintah di bawah ini :

```
choco install deno
```

Jika berhasil maka dalam **directory Chocolatey Package Repository**, **package Deno** akan tersimpan :

```
C:\ProgramData\chocolatey\lib\deno
```

Check Deno

Check Eksistensi **Deno** dengan mengeksekusi perintah di bawah ini :

```
deno -V
```

Jika berhasil maka akan muncul informasi versi **deno** yang sedang kita gunakan :

```
C:\Users\Gun Gun Febrianza>deno -V  
deno 1.3.0
```

Gambar 139 Check Deno Version

Install Deno For Linux

Install Deno via Curl

Buka terminal eksekusi perintah di bawah ini :

```
> curl -fsSL https://deno.land/x/install/install.sh | sh
```

Jika berhasil maka proses instalasi akan dilakukan :

```
##### 100,0%
Archive: /home/<username>/deno/bin/deno.zip
  inflating: deno
Deno was installed successfully to /home/<username>/deno/bin/deno
Manually add the directory to your $HOME/.bash_profile (or similar)
  export DENO_INSTALL="/home/<username>/deno"
  export PATH="$DENO_INSTALL/bin:$PATH"
Run '/home/<username>/deno/bin/deno --help' to get started
```

Gambar 140 Deno Installation Process

Jalankan **command** `deno -V`, apabila muncul "**commnad not found: deno**" maka lanjut ke step berikutnya.

Menambahkan **path** ke **bash_profile** atau **bashrc**, ketik **nano .bashrc** atau **nano .zshrc** tergantung dari **shell** yang kalian gunakan :

```
> nano .bashrc
```

Salin perintah di bawah ini dan simpan di baris terakhir di **.bashrc** atau **.zshrc** :

```
export DENO_INSTALL="/home/<username>/deno"
export PATH="$DENO_INSTALL/bin:$PATH"
```

Ganti bagian `<username>` dengan **username** kalian, lalu **save**, terakhir **restart terminal** dan jalankan perintah :

```
> deno -V
```

Install Deno via Package Manager

Untuk instalasi Via **Brew** eksekusi perintah di bawah ini :

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Install Deno via **Brew** :


```
> brew install deno
```

Install Deno For MacOS

Install Deno via Package Manager

Cara yang paling mudah menggunakan **brew** :

Install Deno via **Brew** :

```
> brew install deno
```

Check Deno Version

Untuk memastikan **Deno Runtime** sudah terpasang eksekusi perintah di bawah ini :

```
> deno --version
```

Jika berhasil maka akan muncul informasi :

```
deno 1.3.0  
v8 8.6.334  
typescript 3.9.7
```

Informasi di atas menjelaskan :

- 1. Deno Runtime Version***
- 2. V8 Engine Version***
- 3. Typescript Compiler Version***

Subchapter 2 – Introduction to Deno

Node could have been much nicer

— Ryan Dahl

Subchapter 2 – Objectives

- Mempelajari Konsep **Deno Runtime**
 - Mempelajari Konsep **Deno Infrastructure**
 - Mempelajari Konsep **Deno Program**
 - Mempelajari Konsep **Deno Command**
 - Mempelajari Konsep **Deno Standard Modules**
 - Mempelajari Konsep **Deno Third-party Modules**
-

1. Why Deno?

Ada beberapa penyebab pencipta **node.js**, Ryan Dahl membangun **deno**. Semuanya memiliki latar belakang yang cukup serius untuk dipertimbangkan. Terdapat beberapa kesalahan yang sempat beliau sesali saat berbicara di **JSConf** EU tahun 2018.

Promise Dilemma

Node.js didesain menggunakan konsep **callback** dan **callback** adalah suatu **function** yang akan berjalan ketika suatu pekerjaan telah selesai. Ketika kita menggunakan **standard module** yang disediakan dalam **Node.js** semuanya mengandalkan **callback**.

Ryan menyadari ada yang lebih baik dari konsep **callback** yaitu **Promise**, yang menyediakan kemampuan lebih baik dan **simple** untuk menangani problema **asynchronous processing**.

Selanjutnya pengembangan **module util** dalam **node.js** dilakukan agar proses **promisify** pada **module** yang mengandalkan **callback** dapat dilakukan. Pengembangan terus

berlanjut dengan menambahkan fitur **async** / **await** agar pengembangan kode **asynchronous javascript** seperti **synchronous javascript**.

Fitur **promise** ditambahkan ke dalam **core node.js** pada tahun 2009 namun setahun kemudian dicabut kembali karena Ryan merasa **codebase node.js** perlu dibuat menjadi lebih sederhana.

Namun kemudian Ryan menyadari bahwa apa yang dilakukannya salah karena konsep **promise** pada akhirnya menjadi standard industri.

Sekarang **Deno Runtime** yang dikembangkannya mendukung konsep **promise**, tersedia **Asynchronous API** yang dikembangkan menggunakan konsep **promise**.

Security

Node didesain sejak awal sudah memiliki **privilege** untuk dapat melakukan akses pada area-area sensitif dalam sistem operasi. Berangkat dari sini Ryan ingin membangun **deno** dengan konsep "**secure by design**" dari sejak awal pengembangan.

Anda akan mempelajari konsep **sandbox** di **chapter** selanjutnya, bagaimana **deno runtime** mengamankan sistem operasi agar tidak dapat diakses oleh **runtime** tanpa **permission** secara eksplisit.

The Build System (GYP)

Ada saatnya kita ingin membuat sebuah **node.js module** yang memiliki **performance** lebih baik dengan cara membuat **native node.js module** menggunakan bahasa **low-level** seperti C++.

Jika kita menulis sebuah **node.js module** yang ingin berinteraksi dengan **library** yang ditulis dengan bahasa C++ kita memerlukan **tool** seperti **GYP (Generate Your Project)**.

Terdapat **tool** seperti **node-gyp** yang dapat melakukan kompilasi **native node.js module** yang ditulis dengan bahasa C atau C++ agar bisa digunakan oleh **node.js module** lainnya.

Node menggunakan **GYP** mengikuti pengembangan **Google Chrome** yang bergantung penuh pada **GYP** namun dikemudian hari, **Google Chrome** berpindah menggunakan GN yang dianggap lebih baik karena bisa memberikan peningkatan pada **performance**.

Hal ini berakibat pada nasib pengembangan **node.js** yang semakin tidak jelas sebagai pengguna **GYP**.

Deno runtime menyelesaikan permasalahan ini dengan menggunakan **GN** dan **cargo**, sebuah **build system** untuk **rust**. Keduanya memiliki **performance build** yang lebih baik dan **API** yang lebih mudah difahami untuk melakukan kompilasi pada **native modules**.

Package.json

Ryan tidak menyukai konsep **package.json** karena kita harus mendefinisikan dua kali pertama dalam **package.json** selanjutnya di dalam **node.js application** melalui **keyword require**.

Package.json juga dikatakan memiliki banyak sekali **noise** dengan menyertakan informasi lainnya seperti **name**, **author**, **description** dan sebagainya.

Selain itu ryan menyadari terdapat perkembangan yang pada akhirnya membuat sebuah ekosistem untuk **node.js repository** tersentral, tepatnya adalah **npmjs dot com**.

Node Modules

Pengembangan menggunakan **node module** dianggap usang karena setiap kali **module** tersimpan dalam **directory project**, duplikasi **dependency** diperlukan dalam **project** lainnya agar dapat berjalan.

Hal ini bisa menjadi sasaran inovasi yang lebih baik dalam **deno runtime**, yaitu dengan menyediakan pendekatan yang lebih baik.

Jika kita memerlukan **module** eksternal pada **deno**, **module** akan di **download** dan **dependencies** disimpan dalam spesifik **directory** yang dapat kita atur.

Jadi kita tidak akan melakukan **fetch module** yang sama lagi melainkan menggunakan **dependencies** yang sudah di **cache**.

Module & Extension

Pada **Node** kita dapat menggunakan **keyword require** dengan **path** tanpa **.js extension**, ini menambah kinerja pada **node** karena harus melakukan **querying** pada **filesystem** untuk melakukan pemeriksaan agar dapat digunakan oleh **node module loader**.

Pada **deno** kita perlu menggunakan ekstensi baik itu **ts** ataupun **js** untuk melakukan import sehingga tidak menambah kinerja pada **runtime**.

Index.js

Jika kita memerlukan sebuah **module** melalui **keyword require**, konsep **file index.js** seringkali digunakan sebagai **entrypoint** pada suatu **module** atau **application**. Sama seperti konsep **index.html** yang akan digunakan oleh **web server** sebagai **entrypoint**.

Node harus melakukan pencarian pada **index.js** seperti pada **folder node_modules** yang benar-benar **bloats**. Hal ini menambah beban kerja lagi pada **node** tepatnya pada **module loading system** untuk mencari **file index.js**

2. Deno Runtime

Perlu diluruskan **Deno** bukanlah sebuah bahasa pemrograman. **Deno** adalah sebuah **Runtime Engine** untuk mengeksekusi **javascript** dan **typescript** secara **stand-alone** diluar **browser**. **Deno** adalah **improvement** dari **node.js** untuk versi yang lebih baik.

Typescript

Deno mendukung pengembangan aplikasi menggunakan kode yang ditulis dengan **typescript** sehingga pembuatan **large-scale project** bisa menjadi lebih mudah. Sebagai **superset** dari **javascript**, **typescript** juga mendukung spesifikasi **EcmaScript**.

Typescript menjadi bahasa yang efektif seperti bahasa **highest-level language** lainnya **C#** dan **Java**. **Typescript** akan membawa banyak sekali **javascript developer** ke **battle royale** di level yang sejajar dengan **C#** dan **Java**.

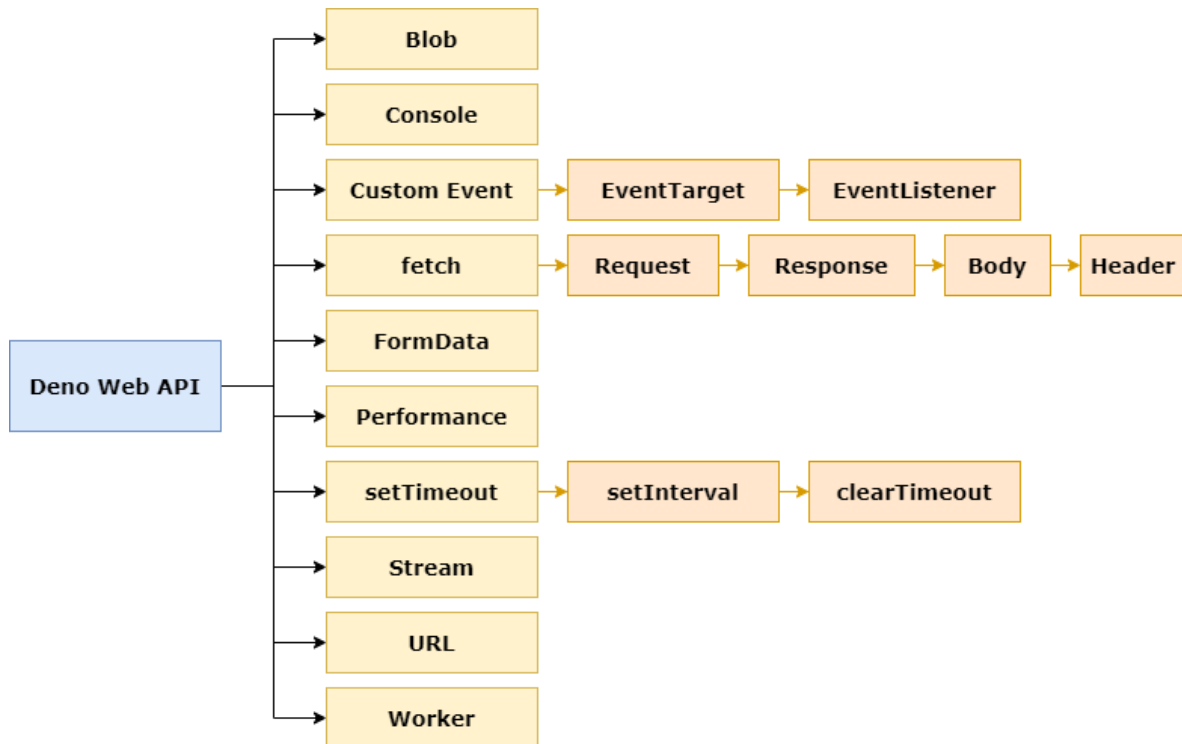
Package Manager

Tidak terdapat dukungan **package manager** seperti pada **Node.js**, **Ryan Dahl** tidak menyukai konsep tempat **package manager** dan **centralized repository** sehingga mengusung **decentralized module**.

Browser Object Support

Meskipun sebagai **runtime** yang mendukung pembuatan **stand-alone application** menggunakan **typescript**, **deno runtime** juga menyediakan **objects** yang ada pada **browser** seperti **object window** dan **fetch**.

Untuk **Web API** yang disediakan dalam **deno runtime** :



Gambar 141 Deno Web API

Buku ini juga membahas kajian tentang :

1. **Performance Web API** di **Chapter 3, Subchapter 19** tentang **Web API**.
2. **fetch Web API** di **Chapter 6, Subchapter 5** tentang **Asynchronous**.

Deno benar-benar menjadi **runtime** yang sangat canggih dengan menerapkan **Web API** secara internal.

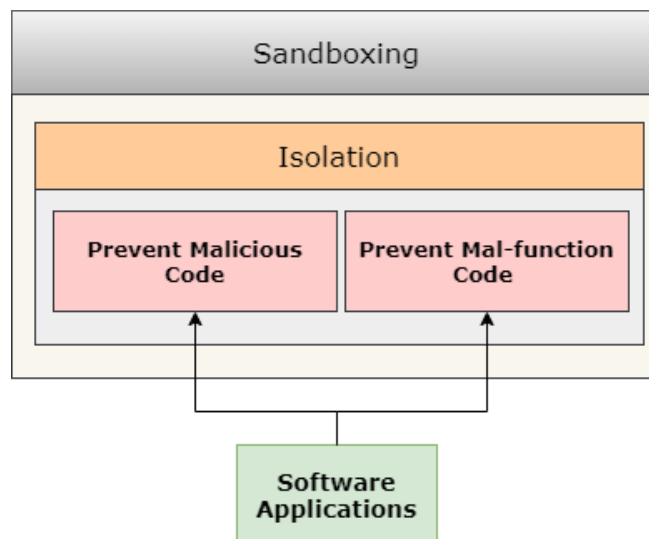
Built-in Tooling

Tersedianya **tools** untuk melakukan **unit-testing**, **code formatting** dan **linting**.

Sandbox

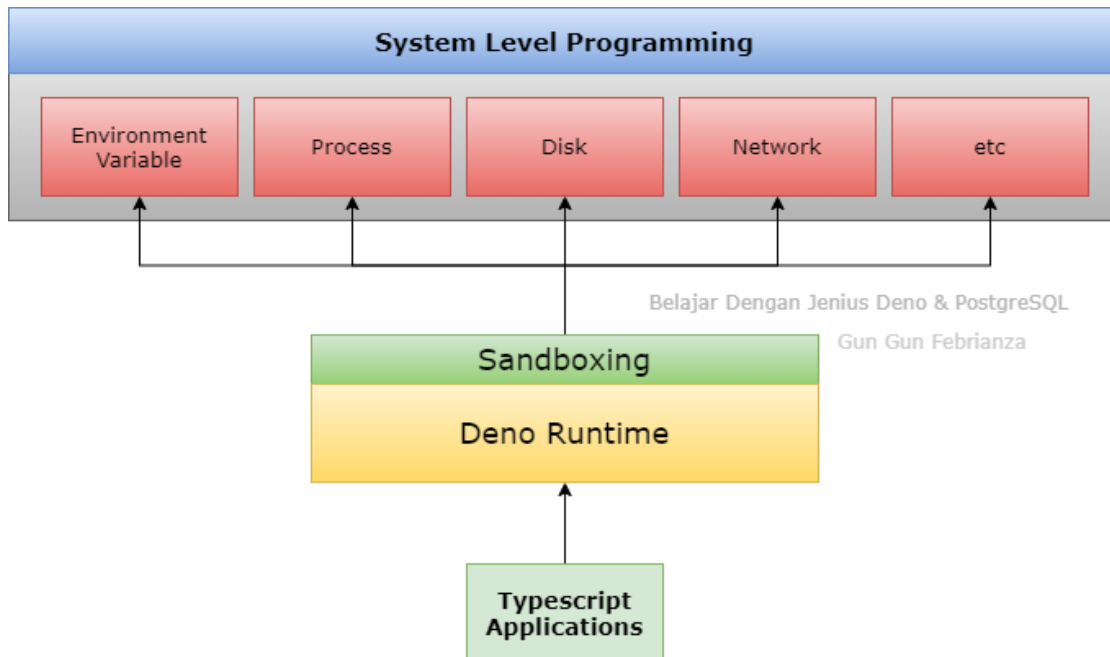
Target utama **Deno** adalah **developer** memberikan sebuah **executable** tunggal yang dapat melakukan apa saja (*arbitrary complex behaviour*), tanpa perlu menggunakan **tooling** tambahan.

Secara **default**, kode **javascript** dan **typescript** akan dieksekusi di dalam **deno runtime** dengan tehnik keamanan terkenal yang disebut dengan **sandboxing**. Sebuah program bernama **sandbox** dibuat dan anda sudah menggunakannya tanpa anda sadari saat menggunakan sebuah **browser**.



Gambar 142 Sandboxing in Deno

Penggunaan **sandboxing** untuk mencegah **malicious code** dan **mal-function** pada **software** yang dapat merusak sistem operasi. Sebuah ruang isolasi disediakan dengan **space memory** tertentu untuk mencegah kerusakan terjadi.



Gambar 143 Deno Sandbox

Pada **deno** terdapat **sandboxing** yang digunakan untuk mengamankan setiap aplikasi yang ditulis menggunakan **javascript** dan **typescript**. Secara teknis keamanan menjadi bersifat **non-repudiation**.

User bertanggung jawab penuh atas keamanan sistemnya sendiri dengan cara mengatur **permission**.

Sebagai contoh jika kita membuat program atau mengeksekusi suatu program yang memerlukan **network** seperti membuka sebuah **socket** maka kita perlu menambahkan **permission**. Pada **deno** untuk akses **network** kita memerlukan **flag** `-allow-net`.

Sebagai pencetus **Deno**, **Ryan Dahl** menginginkan agar **deno** dapat digunakan untuk menyelesaikan permasalahan menggunakan **dynamic language** di berbagai **programming domain**.

Ryan menyadari ketika program yang ditulis semakin kompleks konsep **Type Checking** sangatlah penting agar **deno** bisa menjadi **runtime** untuk menyelesaikan setiap permasalahan dalam berbagai **programming domain**.

Oleh karena itu **typescript** diadopsi oleh **Ryan Dahl**. **Design deno** sebagai **runtime** sekaligus dengan **typescript** di dalamnya, seluruh **standard module** dalam **deno** juga ditulis menggunakan **typescript**.

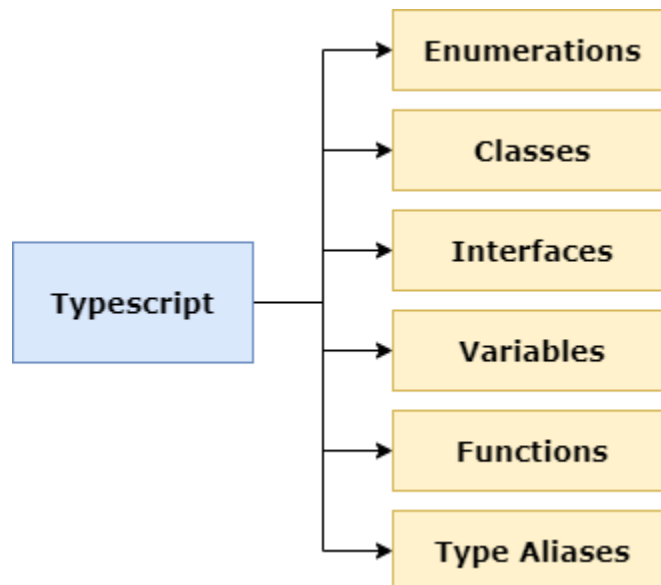
3. Deno Infrastructure

Infrastructure dalam **Deno Runtime** sendiri terbagi menjadi tiga bagian :

1. **Front-end**
2. **Middle-end**
3. **Back-end**

Deno Front-end

Pada bagian **front-end**, infrastruktur **deno** terdapat **typescript**. Disini kita dapat menggunakan **Classes, Interfaces, Variables, Functions, Enumerations** dan **Type Alias** yang ditulis menggunakan **typescript**.



Gambar 144 Front-end Component

Untuk melihat lebih detailnya kunjungi alamat di bawah ini :

<https://deno.land/typedoc>

Di bawah ini adalah sekumpulan **function** yang telah disediakan :

Functions

• chdir	• lstatSync	• realPathSync
• chmod	• makeTempDir	• remove
• chmodSync	• makeTempDirSync	• removeSync
• chown	• makeTempFile	• rename
• chownSync	• makeTempFileSync	• renameSync
• close	• metrics	• resources
• connect	• mkdir	• run
• connectTls	• mkdirSync	• seek
• copy	• open	• seekSync
• copyFile	• openSync	• stat
• copyFileSync	• read	• statSync
• create	• readAll	• test
• createSync	• readAllSync	• truncate
• cwd	• readDir	• truncateSync
• execPath	• readDirSync	• watchFs
• exit	• readFile	• write

Gambar 145 TypescriptFunctions

Jika kita mencoba membuka salah satunya, misal cwd (current working directory) maka kita akan mendapatkan informasi di bawah ini :

cwd

• `cwd(): string`

Defined in [lib.deno.ns.d.ts:173](#)

Return a string representing the current working directory.

If the current directory can be reached via multiple paths (due to symbolic links), `cwd()` may return any one of them.

```
const currentWorkingDirectory = Deno.cwd();
```

Throws `Deno.errors.NotFound` if directory not available.

Requires `--allow-read`

Returns *string*

Pada informasi di atas kita mendapatkan beberapa poin informasi :

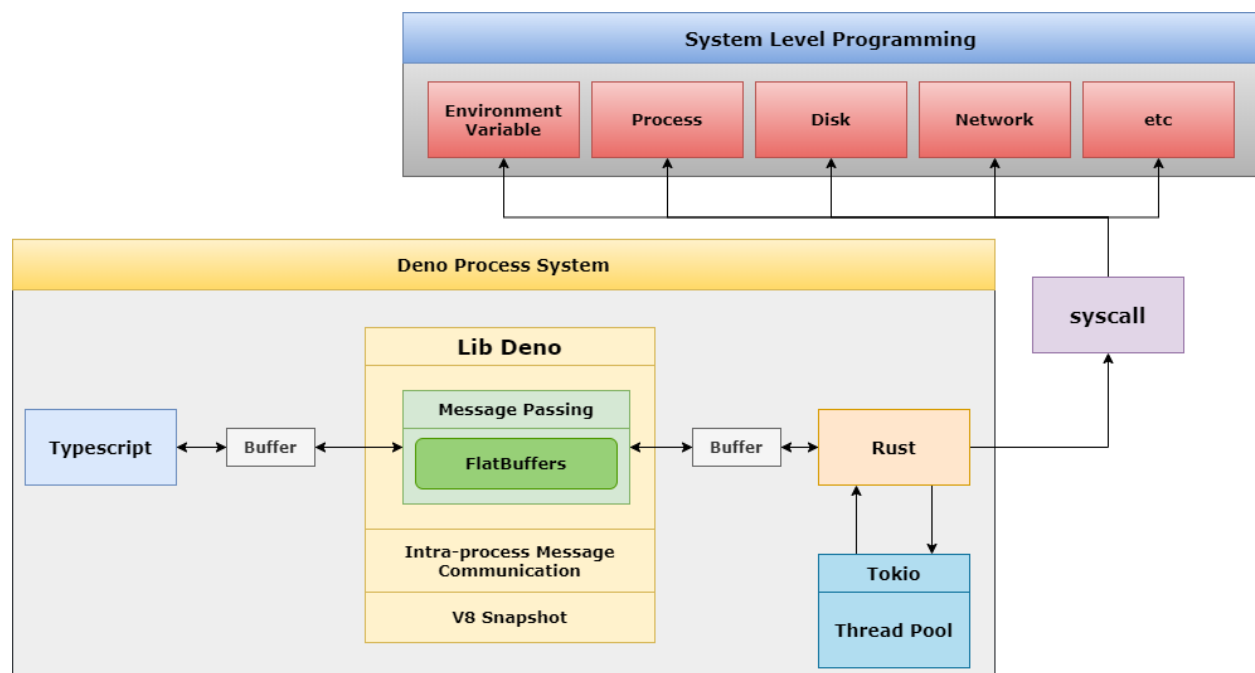
1. Letak `cwd()` dalam **Source Code**
2. Deskripsi dari method `cwd()`
3. **Example code** dalam **typescript**

4. **Error** yang dapat terjadi dalam penggunaan `cwd()`
5. Syarat untuk menggunakan **method** `cwd()` `-allow-read`
6. **Return** dari **method** `cwd()` yaitu **string**.

Bagian **front-end** adalah bagian yang tidak memiliki **privilege** untuk mengelola **syscall** sebagai contoh untuk akses **disk** agar dapat melakukan **file system programming** tidak tersedia karena eksekusi dilakukan dalam **V8 Sandbox**.

Untuk melakukan hal tersebut **passing** informasi harus dilakukan dari **front-end** ke **backend**. Sebagai **backend**, **Rust** memiliki **privilege** untuk pengelolaan **syscall**. **Passing** informasi dengan cara membuat data **buffer**, dikirim melalui **libdeno** sebagai **middle-end** menuju **backend**.

Hasil dapat berupa **output** dari proses yang dilakukan secara **synchronous** atau **asynchronous** di sisi **backend**.



Gambar 146 Deno Infrastructure

Deno Middle-end

Fungsi dari ***deno middle-end*** adalah sebagai layer yang menjadi jembatan antara ***frontend*** dan ***backend***.

Selain menjadi tempat untuk ***message passing API*** antara ***typescript*** dan ***rust***, tugas dari ***libdeno*** adalah berinteraksi dengan ***V8 Engine*** yang di dalamnya sudah tersedia ***typescript compiler***.

Libdeno dapat memuat ***V8 snapshot*** yang tersimpan dalam bentuk ***data blob*** yang di serialisasi (***serialized***) ke dalam bentuk ***heap*** dengan begitu proses inisiasi ***V8*** menjadi lebih cepat.

Untuk ***message passing***, ***library*** yang digunakan adalah ***FlatBuffers*** sebuah ***cross platform serialized library*** yang dikembangkan oleh ***Google***. Kelebihan dari ***FlatBuffers*** adalah pesan yang di ***passing*** dapat dilakukan tanpa proses ***parsing*** dan ***packing*** yang ***overhead***.

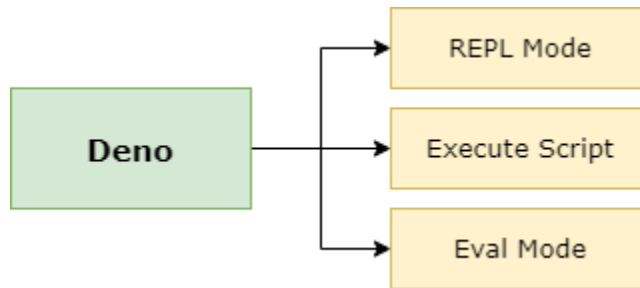
Deno Back-end

Sebagai ***backend*** yang memiliki izin (***privilege***) ***rust*** dijadikan andalan, dikarenakan keunggulannya dalam ***system programming***. Bahasa yang didesain untuk keamanan dalam berinteraksi dengan ***memory*** dan ***concurrency***.

Dari sisi ***backend*** terdapat ***Tokio*** yang menjadi ***asynchronous runtime***, dengan ***tokio*** kita dapat membuat suatu ***event*** dan ***handling event***. Kita dapat melakukan proses ***spawn task*** di dalam ***internal thread pool*** dan mendapatkan notifikasi jika ***process*** sudah selesai.

3. Deno Program

Untuk memulai menggunakan **Deno** ada 3 langkah yang bisa kita gunakan :



Gambar 147 Deno 101

REPL Mode Execution

Untuk menggunakan **Deno** dalam Mode **REPL (Read – Eval – Print – Loop)**, pada **cmd.exe** ketik :

```
> deno
```

Eksekusi **statement** di bawah ini :

```
> 1 + 1
```

Kemudian eksekusi **statement** di bawah ini untuk menampilkan pesan Hello World

```
> console.log("Hello World")
```

Jika berhasil maka akan memproduksi :

```
Hello World
```

Jika ingin keluar dari **REPL Mode**, klik **CTRL+Shift+D**.

Script Execution

Jika kita ingin mengeksekusi sebuah **file javascript** yang telah diubah menjadi **typescript** menggunakan **deno**, eksekusi perintah di bawah ini :

```
> deno run https://deno.land/std/examples/welcome.ts
```

Maka anda akan melihat **output** informasi seperti di bawah ini :

```
Download https://deno.land/std/examples/welcome.ts
Warning Implicitly using master branch
https://deno.land/std/examples/welcome.ts
Compile https://deno.land/std/examples/welcome.ts
Welcome to Deno 🐼
```

Eval Mode

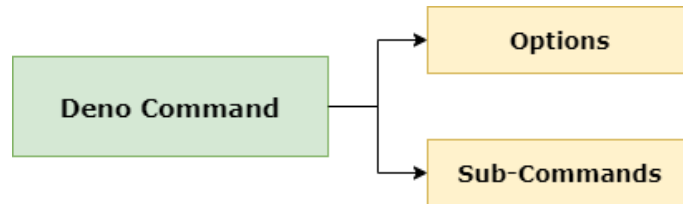
Mengeksekusi dalam mode **eval** jarang sekali dilakukan dan hanya digunakan di kasus-kasus tertentu saja. Di versi **node.js** sebelumnya **eval** mode merupakan model eksekusi yang rentan untuk dieksploitasi, pada **deno** kerentanan tersebut sudah di perbaiki.

Untuk mengetahui cara mengeksekusi dalam **eval** mode, eksekusi contoh kode di bawah ini :

```
> deno eval "console.log(30933 + 404)"
```


4. Deno Command

Deno memiliki **Options & Sub Commands** yang dapat kita gunakan :



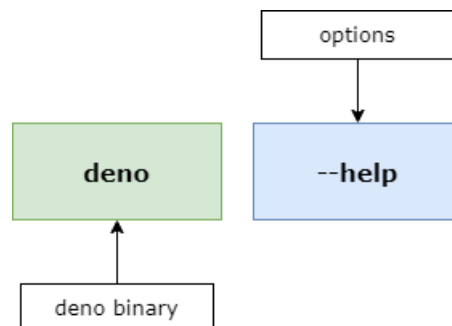
Gambar 148 Deno Command

Deno Options

Saat kita mengeksekusi perintah dengan *options* di bawah ini :

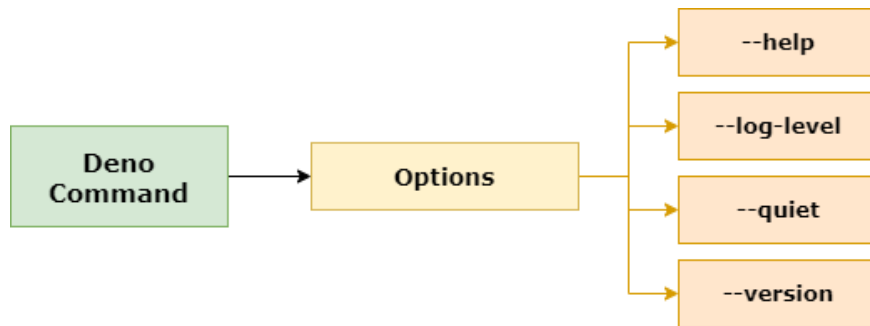
```
> deno --help
```

Terdapat struktur yang bisa kita pelajari :



Gambar 149 Deno Options

Ada beberapa opsi **options** yang dapat kita gunakan :



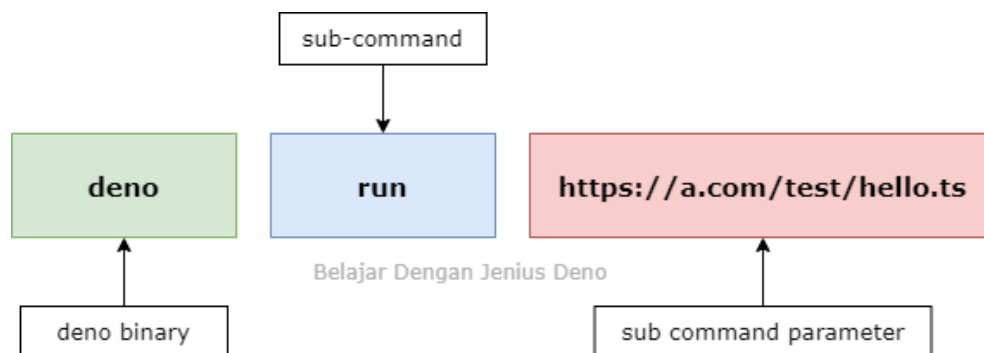
Gambar 150 Deno Options

Deno Sub Command

Saat kita mengeksekusi perintah dengan *sub command* di bawah ini :

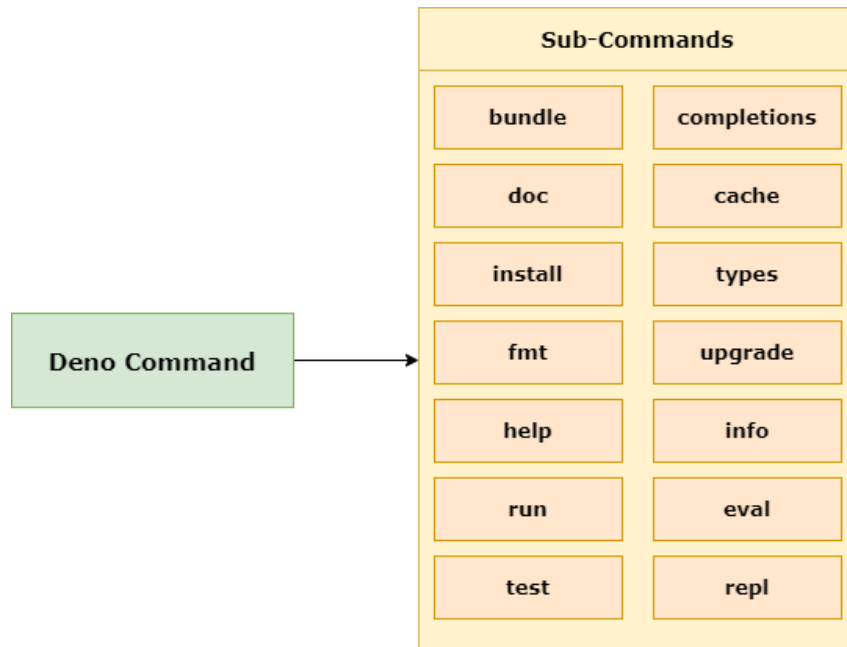
```
> deno run https://a.com/test/hello.ts
```

Terdapat struktur yang bisa kita pelajari :



Gambar 151 Deno Subcommand

So *far* kita telah mengeksekusi satu subcommand yaitu **run**. Jika dieksplorasi lagi ada sekumpulan **sub-commands** yang tersedia dalam **deno**, anda dapat melihatnya pada gambar di bawah ini :



Gambar 152 Deno Sub-commands

Kita akan mempelajari beberapa **subcommand** di atas pada buku ini.

Deno Help

Jika kita ingin mengetahui terdapat apa saja dalam **deno runtime**, eksekusi perintah di bawah ini untuk mendapatkan informasi lebih lanjut :

```
> deno help
```

Jika kita ingin mengetahui informasi salah satu **sub-command** eksekusi perintah di bawah ini :

```
> deno help run
```

Maka anda akan mendapatkan informasi sebagai berikut :

```
Run a program given a filename or url to the module.
```

Run Javascript & Typescript Files

Buatlah dua buah **file** dengan nama **Example.js** dan **Example.ts** dan tulis kode di bawah ini :

```
import { serve } from "https://deno.land/std@0.50.0/http/server.ts";
for await (const req of serve({ port: 8000 })) {
  req.respond({ body: "Hello World\n" });
}
```

Kemudian eksekusi perintah di bawah ini :

```
> deno run Example.js
```

Anda akan mendapatkan **error** sebagai berikut :

```
error: Uncaught PermissionDenied: network access to "0.0.0.0:8000",
run again with the --allow-net flag
```

Ini bukti bahwa **deno** memiliki mekanisme keamanan yang baik dengan menerapkan konsep **Access Control** yang baik, agar kita dapat menggunakannya tambahkan **--allow-net flag** setelah **sub command** :

```
> deno run --allow-net Example.js
```

Buka **browser** anda dan kunjungi 127.0.0.1:8000

Untuk mengeksekusi **typescript file** :

```
> deno run --allow-net Example.ts
```

Deno Permissions

Deno secara **default** tidak memiliki akses untuk mengelola **system programming** secara langsung, izin harus dilakukan secara eksplisit. Di bawah ini adalah beberapa **flag** yang harus diberikan secara eksplisit untuk mengelola **system programming** :

1. **-allow-env**

Flag ini diberikan agar program dapat membaca **environment variables**.

2. **-allow-net=**

Flag ini diberikan agar program dapat berinteraksi dengan **network**.

3. **-allow-plugin**

Flag ini diberikan agar program dapat berinteraksi dengan **plugins**.

4. **-allow-read=**

Flag ini diberikan agar program dapat memiliki izin untuk membaca **file system**.

5. **-allow-run**

Flag ini diberikan agar program dapat memiliki izin untuk menjalankan **subprocesses**.

6. **-allow-write=**

Flag ini diberikan agar program dapat memiliki izin untuk menulis pada **file system**.

7. **-allow-all**

Flag ini diberikan agar program memiliki akses untuk semua **permissions**.

Permission Whitelist

Untuk **granularity control**, **deno** juga memiliki kemampuan untuk melakukan **whitelist permission** sehingga akses dapat dibatasi pada area tertentu saja.

Filesystem Access

Sebagai contoh pada **filesystem programming**, kita dapat membatasi **deno** untuk dapat beroperasi pada **directory** tertentu saja. Perhatikan perintah di bawah ini :

```
> deno run --allow-read=/usr https://deno.land/std/examples/cat.ts /etc/passwd
```

Perintah di atas akan gagal dieksekusi karena **permission whitelist** hanya untuk **directory** `/usr` bukan untuk `/etc/passwd`. Pada perintah di atas kita menggunakan **flag** `--allow-read` untuk membatasi akses dalam membaca, untuk membatasi akses dalam menulis dapat menggunakan **flag** `--allow-write`.

Network Access

Ada saatnya kita mendapatkan atau menulis program yang memiliki akses untuk mengeksplorasi area **network**, seperti :

```
const result = await fetch("https://deno.land/");
```

Di bawah ini adalah perintah untuk memberikan **whitelist** pada program di atas :

```
> deno run --allow-net=github.com,deno.land fetch.ts
```

Jika dalam prosesnya program mencoba melakukan akses selain dari **github.com** dan **deno.land** maka proses akan gagal.

Deno Bundling

Bundling adalah proses untuk menggabungkan aplikasi yang kita buat dan seluruh **dependencies** untuk memproduksi satu buah **file javascript** tunggal.

```
> deno bundle application.ts output.bundle.js
```

Selanjutnya kita dapat mengeksekusinya menggunakan perintah :

```
> deno run output.bundle.js
```

Deno Testing

Deno juga menyediakan **built-in tool** untuk melakukan **test runner** pada kode **javascript** dan **typescript** yang kita gunakan. Kita dapat melakukan **full test suit** menggunakan perintah di bawah ini :

```
> deno test application_test.js
```

Kita akan mempelajari **testing** lebih dalam pada buku ini.

Deno Reload Module

Mekanisme untuk **update module** adalah menggunakan perintah **reload**, sebab **module** yang dimuat dari suatu **URL** di **cache** oleh sistem secara lokal. Di bawah ini perintah untuk melakukan **reload** :

```
> deno run -reload app.ts
```

Deno Install

Deno memiliki **built-in script installer** yang dapat membantu kita mendistribusikan **executable** tunggal agar dapat digunakan dikomputer lainnya.

```
> deno install https://www.resources.com/application-x.ts
```

Jika berhasil script akan tertanam di **folder .deno/bin** dalam **home directory** sistem operasi kita.

Deno Formatting

Kita juga dapat melakukan **formatting** agar kode yang kita tulis menjadi rapih dengan cara mengeksekusi perintah di bawah ini pada **current directory** tempat anda menulis kode :

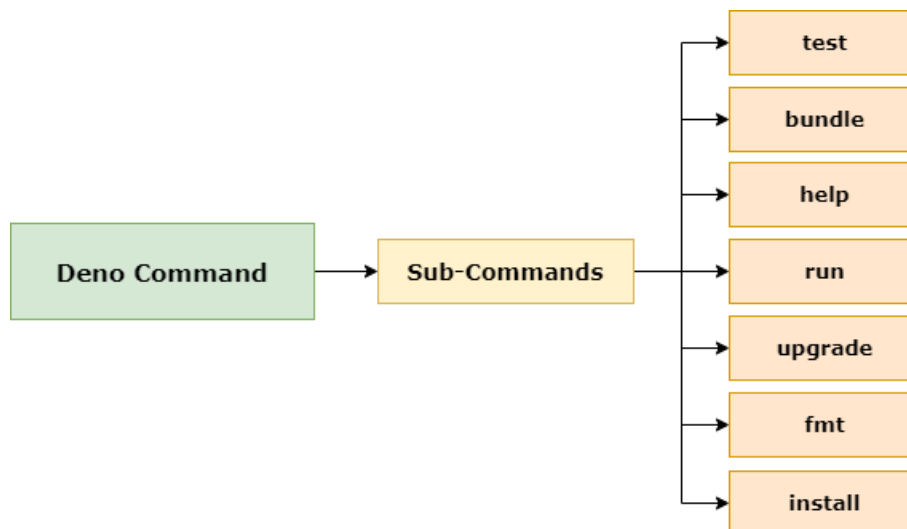
```
> deno fmt
```

Deno Upgrade Command

Instruksi untuk melakukan **upgrade deno runtime**, upgrade ke versi **1.0.2** :

```
> deno upgrade --version 1.0.2
```

Pada perintah di atas jika ada update terbaru lagi cukup ubah 1.0.2 dengan versi terbaru yang anda dapatkan. *So far*, kita telah mencoba mengeksekusi **subcommand** **run** & **upgrade**, *achievement unlocked*.



Gambar 153 Sub-command upgrade

5. Deno Standard Module

Deno telah menyediakan **standard module** yang dapat kita gunakan tanpa memerlukan **external dependencies**. Di bawah ini adalah sekup **standard module** yang disediakan pada **deno** pada versi terbaru :

Deno Standard Module		
archive	hash	signal
async	http	testing
bytes	io	textproto
datetime	log	uuid
encoding	mime	ws
flags	node	
fmt	path	
fs	permissions	

Gambar 154 Deno Standard Module

Beberapa **module** masih **unstable** sehingga untuk mengeksekusinya diperlukan **flag** **unstable**. Untuk mengunjungi halaman resmi dokumentasi **standard module**, dapat di cek disini :

<https://deno.land/std>

Kita dapat membaca cara penggunaan suatu **module**, pada gambar di bawah ini penulis sedang membaca dokumentasi **module fs** :



Gambar 155 Module fs

Module fs

Module ini digunakan jika kita ingin memanipulasi **filesystem**. Untuk memeriksanya silahkan cek disini :

<https://deno.land/std/fs>

Kita akan mempelajari **module** ini pada **Chapter 4**, tentang **Filesystem Programming**.

Module http

Module yang dapat digunakan untuk membuat **application server**. Untuk memeriksanya silahkan cek disini :

<https://deno.land/std/http>

Module datetime

Module yang digunakan untuk melakukan **parsing** pada **date string** untuk dikonversi ke dalam **object date** dengan **function** baru tambahan. Untuk memeriksanya silahkan cek disini :

<https://deno.land/std/datetime>

Module node

Module ini digunakan agar kita bisa berinteraksi dengan **Node.js Standard Library** yang ada disini :

<https://nodejs.org/api/fs.html>

Ada beberapa **module** yang sudah siap, ada juga **module** yang belum selesai.

- | | |
|--|---|
| • <input type="checkbox"/> crypto | • <input checked="" type="checkbox"/> module |
| • <input type="checkbox"/> dgram | • <input type="checkbox"/> net |
| • <input type="checkbox"/> dns | • <input checked="" type="checkbox"/> os <i>partly</i> |
| • <input checked="" type="checkbox"/> events | • <input checked="" type="checkbox"/> path |
| • <input checked="" type="checkbox"/> fs <i>partly</i> | • <input type="checkbox"/> perf_hooks |
| • <input type="checkbox"/> http | • <input checked="" type="checkbox"/> process <i>partly</i> |
| • <input type="checkbox"/> http2 | • <input checked="" type="checkbox"/> querystring |
| • <input type="checkbox"/> https | • <input type="checkbox"/> readline |

Gambar 156 Supported Built-in

Untuk memeriksanya silahkan cek disini :

<https://deno.land/std/node>

Module ws

Module ini digunakan agar kita dapat mengembangkan aplikasi **web socket** berbasis **client** ataupun **server** :

<https://deno.land/std/ws>

6. Deno Third-party Modules

Untuk menggunakan **third-party modules** silahkan kunjungi :

<https://deno.land/x/>

event_kit	Simple module for implementing event subscription APIs	>
evt	💧 A type safe replacement for Node's EventEmitter 💧	>
exec	Run external applications more easily and with numerous options.	>
execute	Execute and get the output of a shell or bash command in Deno.	>
expect	A deno implementation of expect in order to write tests in a more jest like style.	>
fanfou_sdk	Fanfou SDK for Deno	>
fastlist	List all running processes on Windows.	>

Gambar 157 Third-party Modules

Kita dapat melakukan **import third-party module** dari sebuah **web**, seperti **github**, **web server** yang kita buat sendiri, atau melalui **Content Delivery Network** seperti :

1. pika.dev
2. jspm.io

Subchapter 3 – Typescript

JavaScript is the world's most misunderstood programming language.

— Douglas Crockford

Subchapter 3 – Objectives

- Mempelajari sejarah singkat **javascript**
 - Mempelajari sejarah singkat **node.js**
 - Mempelajari sejarah singkat **typescript**
 - Mempelajari Konsep **Compilation** dalam **typescript**
 - Mempelajari Konsep **Static Typing** dalam **typescript**
 - Mempelajari **Compiler & Compilation** dalam **typescript**
 - Mempelajari **Compiler Options** dalam **typescript**
-

1. Javascript

Javascript is an **Interpreted Language**. Secara historis penciptaan javascript digunakan agar sebuah halaman *web* menjadi lebih hidup. Sehingga seringkali disebut dengan **client-side programming**.

Javascript adalah sebuah bahasa pemrograman 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*.

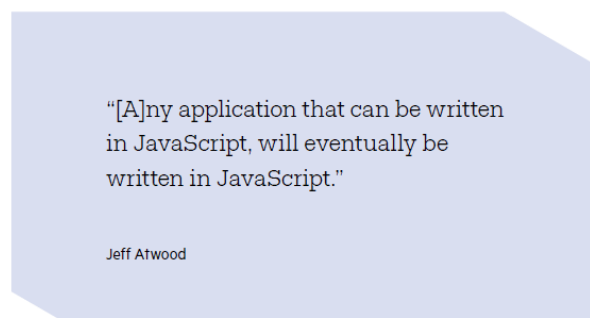
Machine code adalah sebuah bahasa yang dimengerti oleh komputer. *Machine code* adalah sebuah *string* yang terdiri dari bilangan biner 1(s) dan 0(s). Analoginya seperti mengkonversi bahasa inggris kedalam bahasa Indonesia sehingga dapat kita mengerti.

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 populer 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 158 Javascript Quote by Jeff Atwood

2. Node.js

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*.

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 pemrograman *C#, Java, Ruby, Python* dan sebagainya.

Hal ini membuat predikat bahasa *C++* mendapatkan klasifikasi sebagai *middle level programming laguage*, beberapa literatur mengklasifikasinya 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**.

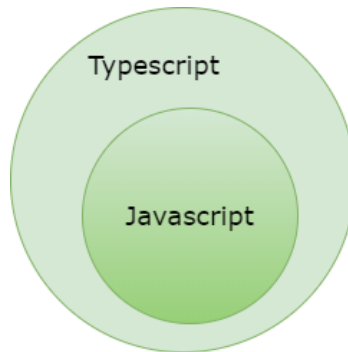
Jadi kira kira aplikasi apa saja yang mampu dibuat dengan **node.js**?

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**.

3. Typescript

Typescript adalah sebuah **superset javascript** yang dikembangkan oleh **Microsoft**, sebuah **open source project** yang menginspirasi **Ryan Dahl** untuk mengembangkan **Deno** sebuah **Runtime Engine** yang dapat digunakan untuk mengeksekusi **Typescript**.



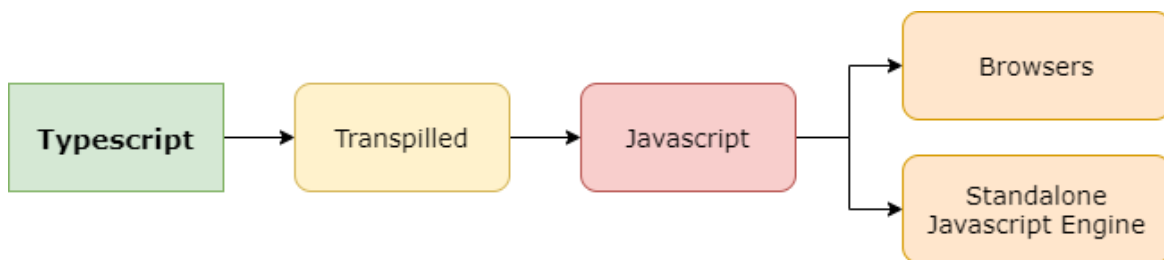
Gambar 159 Superset Javascript

Jika anda melihat gambar di atas pasti faham, setiap **Javascript Code** adalah sebuah **typescript** dan **typescript** memberikan kelebihan baru untuk **javascript** sehingga seringkali disebut dengan **Extended Javascript**.

What if we could strengthen JavaScript with the things that are missing for large scale application development, like static typing, classes [and] modules... that's what TypeScript is about.

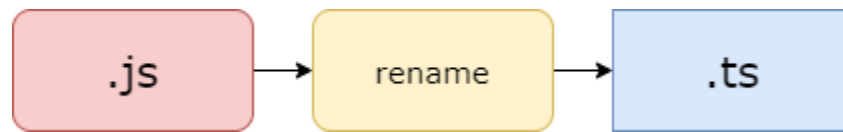
—Anders Hejlsberg

Sebuah program yang ditulis menggunakan **typescript** harus dikompilasi terlebih dahulu kedalam **javascript** agar bisa dieksekusi oleh sebuah **browser** atau **javascript engine**.



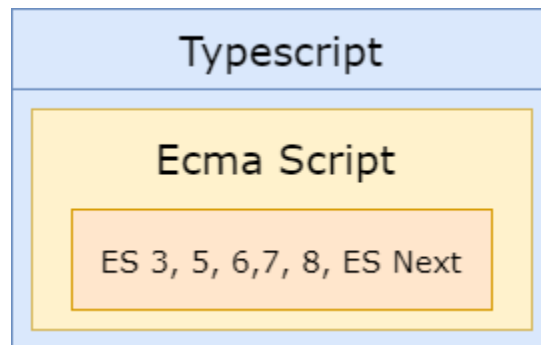
Gambar 160 Transpilation

Sebagai **superset javascript** kita dapat mengubah setiap **javascript file** ke dalam **typescript file** dengan cara mengubah ekstensinya :



Gambar 161 Javascript to Typescript

Typescript juga menjadi sebuah superset atas **EcmaScript** yang menjadi penentu spesifikasi standar penulisan **javascript**. **ES Next** merepresentasikan versi yang akan datang dan versi terakhir dari **EcmaScript**.



Gambar 162 Superset over ES

Sebagai **superset javascript** apa saja keuntungan yang dapat diberikan oleh **typescript**?

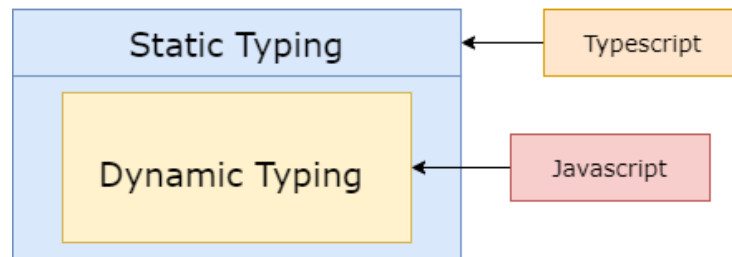
Compilation

Pada **javascript** kita harus mengeksekusi kode **javascript** terlebih dahulu untuk memastikan bahwa program **javascript** benar benar **valid**. Pada **Typescript**, terdapat proses **transpilation**.

Proses **transpilation** dari **typescript** ke dalam **javascript** memberikan kelebihan untuk melakukan **error checking**. **Typescript compiler** akan memeriksa kode dan memproduksi **error** jika terdapat kesalahan. **Error** dapat terdeteksi sebelum kode dieksekusi.

Static Typing

Javascript adalah bahasa yang memiliki karakteristik **dynamic typing**. Sebagai **superset**, **typescript** memberikan kelebihan baru yaitu dukungan karakteristik **static typing**. So, apasih keunggulan dari **static typing** ?



Gambar 163 Typing

Setiap kali kita membuat sebuah variabel kita harus menentukan terlebih dahulu tipe data yang akan digunakan. Hal ini melindungi **developer** dari sekumpulan **bug** umum yang sering muncul dan **security flaw** dari kode yang kita tulis.

Dengan begitu pengembangan **software** menjadi lebih aman, sebagai contoh di bawah ini kita membuat variabel **name** dengan tipe data **number**. Namun kita memberikan literal string sehingga kode **javascript** akan gagal saat dikompilasi ke dalam **javascript** :

```
let fullname: number;
fullname = "Maudy Ayunda"; // compile-time error
```

Typescript Compiler

Typescript Compiler adalah program **compiler** yang akan melakukan **transpiling** yaitu konversi **typescript** ke dalam **javascript** dan memastikan kode tidak memiliki **type error**.

Untuk melakukan instalasi **typescript compiler** eksekusi perintah di bawah ini :

```
npm install -g typescript
```

Untuk memastikan **typescript compiler** sudah terpasang eksekusi perintah di bawah ini :

```
tsc -v
```

Compile Typescript

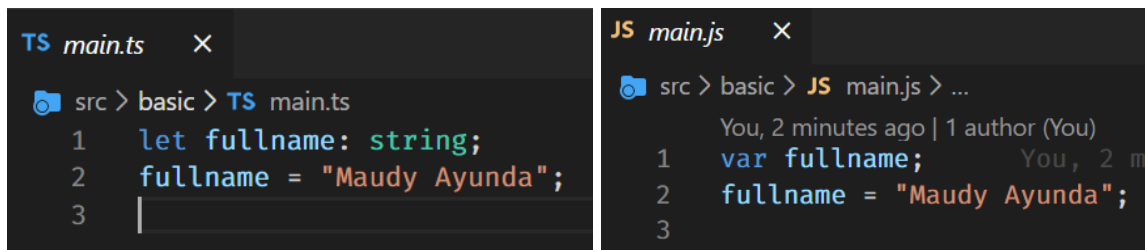
Buatlah sebuah **file** dengan nama **main.ts** kemudian tulis kode dibawah ini :

```
let fullname: string;  
fullname = "Maudy Ayunda";
```

Untuk melakukan kompilasi eksekusi perintah di bawah ini :

```
tsc main.ts
```

Maka sebuah **file javascript** akan diproduksi dengan nama main.js.



Gambar 164 Comparison

Jika kita bandingkan sebelum dan sesudah kompilasi kita dapat melihat perbedaannya pada gambar di atas. Pada kasus yang lebih **advance** struktur penulisan pada **typescript** menjadi lebih rumit namun setiap komponen program yang kita bangun tertata rapih.

Perhatikan gambar di bawah ini :

```

interface person {
  name: string,
  gender: string,
  age: number
}

class Agent implements person {
  constructor(
    public name: string,
    public gender: string,
    public age: number) {
  }
}

const hooman = new Agent("Maudy", "Woman", 27)
alert(hooman.name)

```

```

"use strict";
class Agent {
  constructor(name, gender, age) {
    this.name = name;
    this.gender = gender;
    this.age = age;
  }
}
const hooman = new Agent("Maudy", "Woman", 27);
alert(hooman.name);

```

Gambar 165 Typescript to Javascript

Typescript Compiler Options

Kita dapat membuat sebuah **preconfiguration** sebelum kompiler melakukan kompilasi, dengan cara membuat sebuah **file tsconfig.json**. Di bawah ini adalah contoh **Compiler Options** :

```

{
  "compilerOptions": {
    "baseUrl": "basic",
    "outDir": "./dist",
    "noEmitOnError": true,
    "target": "es5"
  }
}

```

Selain membuat **preconfiguration** sendiri kita juga dapat memproduksinya menggunakan perintah :

```
> tsc --init
```

Option baseUrl

Digunakan agar **compiler** mengetahui lokasi **directory** tempat **compiler** akan melakukan **transpilation**.

Option outDir

Digunakan agar **compiler** memproduksi **output target javascript file** ke dalam lokasi **directory** yang kita tentukan.

Option noEmitOnError

Pada kode di atas kita menggunakan **noEmitOnError** agar kompiler tidak memproduksi **javascript** jika di kode **typescript** yang ditulis masih terdapat **error**.

Option Target

Target spesifikasi kode **javascript** yang ingin diproduksi, terdapat spesifikasi sebagai berikut :

1. **ES3**
2. **ES5**
3. **ES2015**
4. **ES2016**
5. **ES2017**
6. **ES2018**
7. **ES2019**
8. **ES2020**
9. **ESNext**

Jika sudah eksekusi perintah di bawah ini :


```
tsc
```

Maka kompiler akan mengeksekusi script sesuai dengan konfigurasi yang telah diberikan.

Option Watch

Jika kita ingin melakukan watch mode tambahkan kedalam `tsconfig.json` :

```
"watch": true
```

Jika sudah eksekusi perintah di bawah ini :

```
C:\Users\Gun Gun Febrianza\Pictures\Mastering-DenoTheWKWKLand\src\basic>tsc
[10:47:22] Starting compilation in watch mode...

[10:47:24] Found 0 errors. Watching for file changes.
```

Gambar 166 Watch Mode

Option Module

Module format yaitu *syntax* yang digunakan untuk membuat sebuah *module*. Sebagai contoh pada ***Universal Module Definition (UMD)***, ***format*** ini dapat digunakan di dalam *browser* dan *node.js*.

Pada ***Module Format CommonJS (CommonJS)***, format ini digunakan di dalam *Node.js* cirinya penggunaan *keyword* `require` dan `module.exports` untuk menentukan sebuah *dependencies* dan *modules*.

Format Module yang didukung adalah :

1. ***CommonJS***
2. ***AMD***
3. ***UMD***
4. ***System***

5. ES2015

6. ESNext

More Options

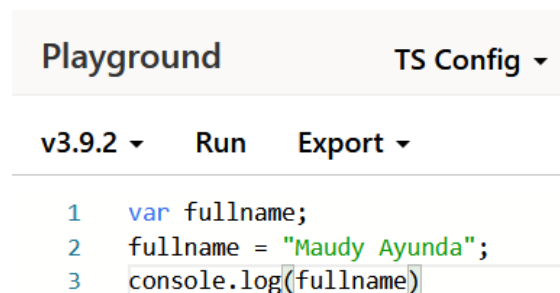
Untuk mengetahui lebih lengkap **options** yang disediakan kunjungi halaman berikut :

<https://www.typescriptlang.org/docs/handbook/compiler-options.html>

Typescript Playground

Selain menggunakan **local typescript compiler** kita juga dapat mempelajari **typescript** di dalam **browser**. Silahkan kunjungi **typescript playground** di sini :

<https://www.typescriptlang.org/play>



Gambar 167 Typescript Playground

Subchapter 4 – *Fundamental Deno*

JavaScript is the world's most misunderstood programming language.

— Douglas Crockford

Subchapter 4 – Objectives

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

1. Deno Task Runner

Sebelum kita belajar menggunakan buku ini agar kita tidak secara manual melakukan kompilasi, silahkan **install** terlebih dahulu **third-party module drun**. Eksekusi perintah di bawah ini untuk melakukan instalasi :

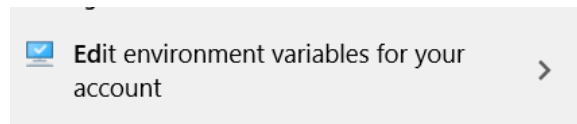
```
> deno install --allow-read --allow-run --unstable -f https://deno.land/x/drun/drun.ts
```

Untuk **repository module** cek disini :

<https://deno.land/x/drun>

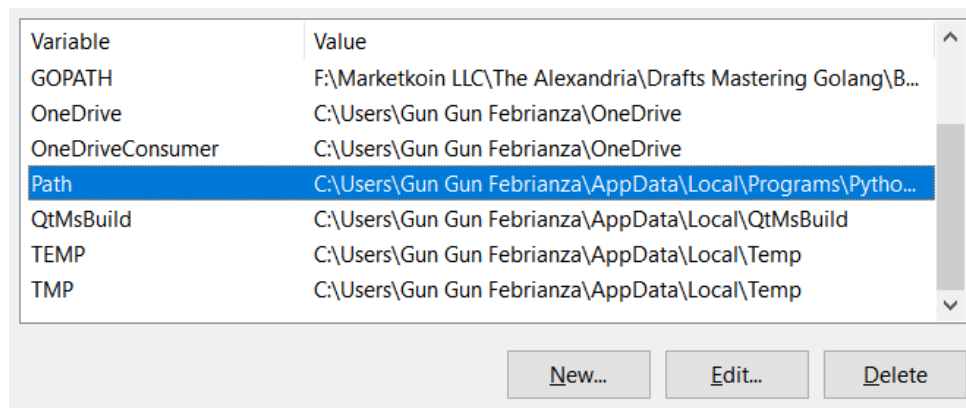
Edit Environment Variable

Pada sistem operasi **windows**, lakukan konfigurasi **Edit Environment Variable**. Pada kolom pencarian cukup ketik **edit environment variable**, kemudian klik ikon aplikasi di bawah ini :



Gambar 168 Edit Environment Variable

Kemudian **edit path variable**, klik tombol **edit** :



Gambar 169 Path Variable

Selanjutnya klik tombol **New**, kemudian tambahkan **path** di bawah ini :

```
C:\Users\Gun Gun Febrianza\.deno\bin
```

Jika sudah **restart** kembali **visual studio code** dan **terminal** yang kita gunakan.

Create Training Directory

Buat sebuah **folder** dengan nama **learning** dan buatlah **file** dengan nama **drun.json**, kemudian tulis kode di bawah ini :

```
{
  "entryPoint": "./index.ts",
  "cwd": "./",
  "excludes": ["./exclude.ts"],
  "runtimeOptions": ["--allow-write", "--allow-read"]
}
```

Pada kode **json** di atas terdapat 4 **pre-configuration** :

Entrypoint

Nama **file** yang akan kita gunakan untuk belajar tempat **drun** membantu kita melakukan **restart** jika terdapat perubahan pada **file** yang sedang kita tulis.

Cwd

Lokasi **root folder** yang menjadi area **drun** untuk melakukan **watching** jika terdapat perubahan pada setiap **files** dalam **directory** tersebut.

Excludes

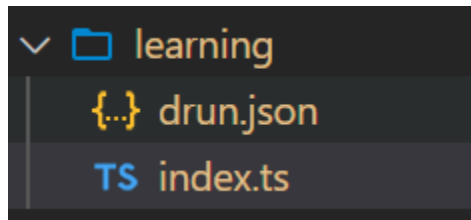
Untuk memberi instruksi pada **drun** agar tidak melakukan **watching** pada file **exclude.ts**, anda bisa menambahkan daftar **file** yang tidak ingin di observasi jika terdapat perubahan.

RuntimeOptions

Tempat kita akan memberikan **options** pada **deno runtime**, pada kode di atas kita memberikan **flag permission** **--allow-write** dan **--allow-read**.

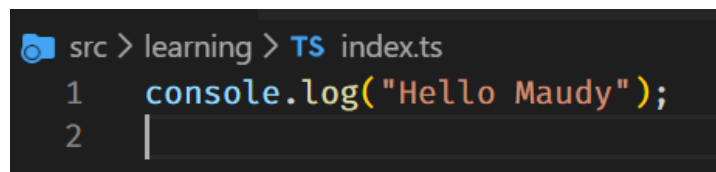
Create *index.ts*

Buatlah **file** yang akan menjadi tempat **playground** kita untuk mempelajari **typescript** dalam buku ini :



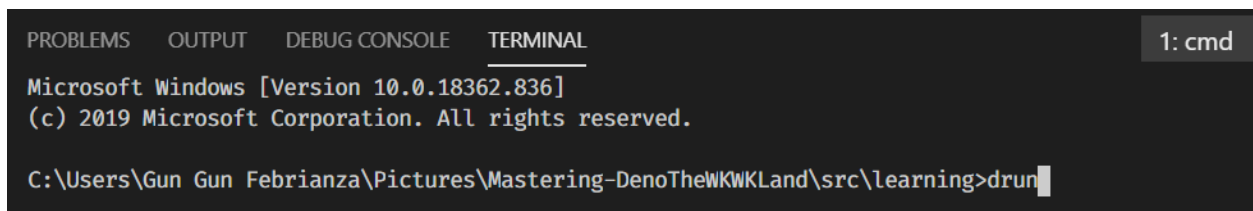
Gambar 170 Create Index.ts

Kemudian tulis kode di bawah ini :



Gambar 171 Hello World

Kemudian pada terminal eksekusi perintah di bawah ini :



Gambar 172 Drun

Kemudian silahkan ubah konten dalam **index.ts** misal menjadi **"Hello Maudy Ayunda"** kemudian tekan tombol **save** (**CTRL+S**) maka anda akan melihat output secara langsung secara otomatis.

Untuk keluar dari **process drun**, cukup tekan **CTRL+D** kemudian ketik yes.

2. Hello World

Sudah menjadi tradisi jika kita berkenalan dengan suatu bahasa pemrograman kita akan membuat program yang paling sederhana yaitu **Hello World**. Buatlah sebuah **file** dengan nama **helloworld.ts** :

```
console.log("Hello World");
```

3. 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 lainnya 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.*

Pada ***typescript*** tidak terdapat perubahan pola dalam gaya penulisan komentar.

4. Variable Declaration

Variable

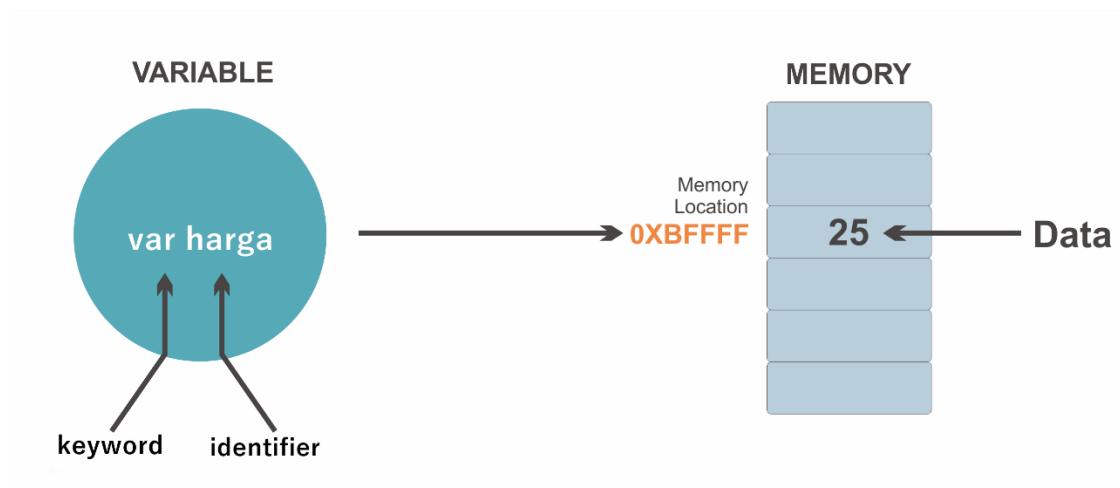
Setiap program pasti membutuhkan tempat untuk menyimpan suatu data. Data tersebut akan disimpan di lokasi memori tertentu. **RAM (Random Access Memory)** komputer terdiri dari jutaan sel memori. Ukuran dari setiap sel tersebut sebesar 1 **byte**.

Sebuah *RAM* komputer dengan ukuran 8 (**GigaBytes**) memiliki $8 \times 1024\text{MB} = 8192 \times 1024\text{KB} = 8.589.934.592$ sel memori.

Saat **deno runtime engine** membaca *statement code* di bawah ini :

```
var harga : number = 25;
```

Maka secara **internal** dapat di representasikan sebagai berikut :



Gambar 173 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*.

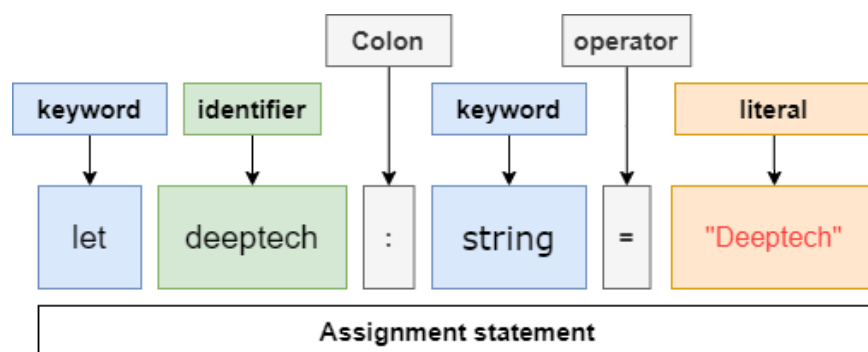
Literal

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 : string = "DeepTech"

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

*Link sumber kode.



Gambar 174 Variable Declaration Syntax Rule

Binding

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

Reserved Words

Dalam *typescript* 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 *typescript compiler*. *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 *Typescript keywords* :

Table 5 Typescript Keyword

as	any	enum	get
number	string	module	public
private	package	implements	interface
static	void	while	yield
break	case	catch	class
const	continue	debugger	default

delete	do	else	export
extends	finally	for	function
if	import	in	instanceof
new	return	super	switch
this	throw	try	typeof
var	void		

Future Reserved Words

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

Table 6 Future Reserved Keyword

abstract	protected*		
----------	------------	--	--

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*, *Deno Runtime Engine* akan memproduksi *error*!

Naming Convention

Seperti yang telah kita fahami sebelumnya, *identifier* adalah nama yang menjadi pengenalan 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 diawali dengan **symbol** \$ misal `$deeptech`.
3. Dapat diawali dengan **symbol** _ misal `_deeptech`.
4. Dapat diawali dengan **character** dalam *unicode* (contoh π atau \ddot{o}). dilanjutkan dengan *symbol* \$, _ angka, atau *character* lagi misal : `deep_tech` atau `deep$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$tech : string = "deep$tech"
console.log(deep$tech);

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

var deeptech2019 : string = "deeptech2019"
console.log(deeptech2019);
```

```
var namingConvention : string = "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 : number = Math.PI;  
console.log(π);  
  
var ၀_၀ : string = "၀_၀";  
console.log(၀_၀);  
  
var ლ_ჲჲლ : string = "ლ_ჲჲლ";  
console.log(ლ_ჲჲლ);  
  
var << : string = "<<";  
console.log(<<);  
  
// berbeda:  
// << << <<;  
  
// Roman numerals  
var IV : number = 4;  
var V : number = 5;  
console.log(IV + V); // 9
```

**Link sumber kode.*

Apa itu Unicode?

Typescript 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

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

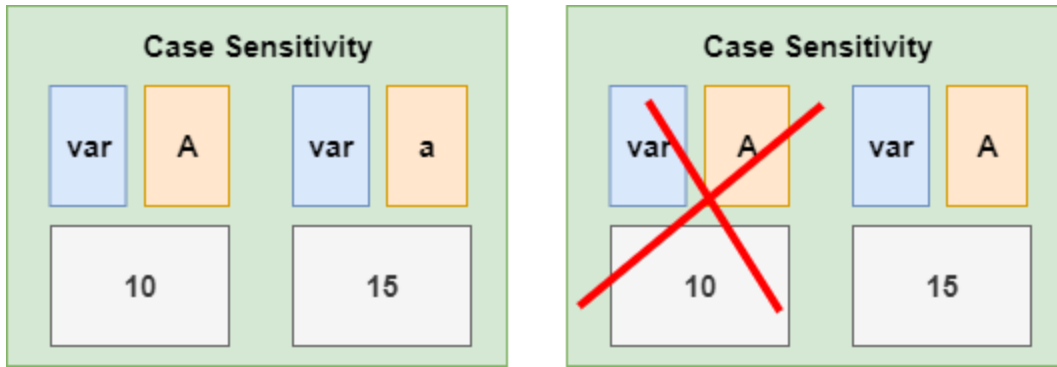
```
var A = 10;
var a = 15;
console.log(A); //A Capital
console.log(a); //a small
```

Output :

10

15

**Link sumber kode.*



Gambar 175 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**.

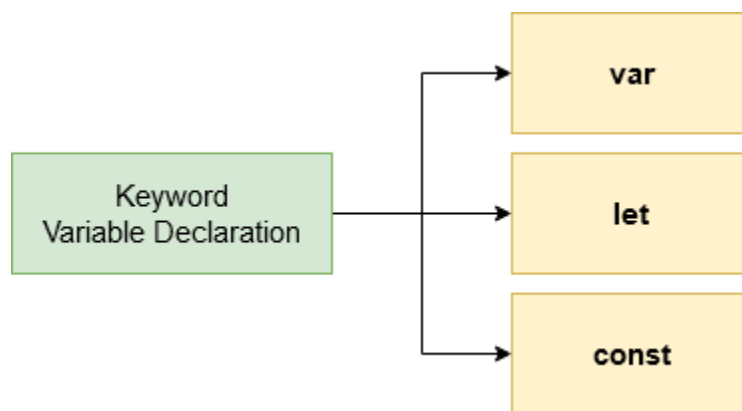
Loosely Typed Language

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**.

Typescript Static Typing

JavaScript sebagai bahasa yang tidak memiliki karakteristik **strong typed** dan **typescript** hadir untuk memberikan opsi **static typing** dan **type inference** sistem melalui **Typescript Language Service (TLS)**.

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



Gambar 176 Variable Declaration Keyword

Var Keyword

Keyword yang digunakan untuk mendeklarasi *variable* semenjak *javascript interpreter* pertama kali dibuat. Di bawah ini adalah contoh *variable declaration* menggunakan **reserved keyword** **var** :

```
var angka: number = 99;  
var logika: boolean = true;  
var teks: string = "Maudy Ayunda";
```

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* tulis kode di bawah ini :

```
console.log(angka); //99  
console.log(logika); //true  
console.log(teks); //Maudy Ayunda
```

Let Keyword

Selain **keyword** **var**, kita juga dapat menggunakan **keyword** **let** yang dapat kita gunakan untuk membuat **variable** dengan **scope local**. Berbeda dengan **var** yang memiliki **scope global**, jadi apa itu **scope**?

Variable Scope

Dalam deklarasi variabel terdapat istilah **Variable Scope**, yang menjadi terminologi untuk menentukan dimana lokasi suatu variabel. Sebuah variabel dapat berada di dalam **scope global** dan **scope local**.

Variabel yang berada dalam **scope local** hanya dapat digunakan di dalam **scope local** itu sendiri, sementara variabel dalam **scope global** dapat digunakan diseluruh **scope** program.



Gambar 177 Variable Scope

Gambar di atas menjelaskan bahwa dalam sebuah deno program kita dapat membuat banyak sekali **variable** dengan **scope global** dan **variable scope local**.

Sebagai contoh jika kita membuat 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 kode di bawah ini *variable x* tetap bisa dibaca diluar *block* karena deklarasinya menggunakan *reserved keyword* **var**.

```
if (true) {  
  var x: number = 5;  
}  
console.log(x); //5
```

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

```
var x: number = 8;
{
  let x: number = 5;
  console.log(x);
}
console.log(x);
```

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

```
if (true) {
  let z: number = 5;
}
console.log(z);
```

Eksekusi kode di atas maka **error** akan diproduksi sebagai berikut :

```
error: TS2304 [ERROR]: Cannot find name 'z'.
  console.log(z);
               ^
```

Gambar 178 Error

Constant Keyword

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

Pada *Typescript* 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 *alphanumeric*, *numeric* dan *underscore character*.

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

```
const z: number = 9  
z = 3
```

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

```
error: TS2588 [ERROR]: Cannot assign to 'z' because it is a constant.  
  z = 3  
  ^
```

Gambar 179 Redeclaration variable x

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

```
var x: number = 8;  
{  
  const x: number = 5;  
}  
console.log(x); //8
```

5. 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 sebuah *statement* :

```
let x: number; //declaration statement
```

Pada *statement* di atas terdapat perintah untuk melakukan deklarasi variabel, dimana x adalah variabel yang dapat digunakan untuk menyimpan nilai ***number***.

```
let x: number; //declaration statement
x = 2 * 10; //expression statement
console.log(x); // 20
```

**Link sumber kode.*

Pada baris kedua 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 *deno runtime* mengetahui mana operasi aritmetika yang harus dilakukan terlebih dahulu, yaitu operasi bagi & perkalian terlebih dahulu kemudian penjumlahan :

```
function arithmeticOperation(params: number): number {  
  params = 2 + 2 * 10 / params; //expression statement  
  return params; //12  
}  
  
console.log(arithmeticOperation(2));
```

**Link sumber kode.*

Fungsi *arithmeticOperation* membutuhkan parameter sebuah argument dengan type *number* dan

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 7 Arithmetic Operator

Operator	Description	Example
+	<i>Addition (String Concat)</i> atau penjumlahan	2+1 //3
-	<i>Subtraction</i> atau pengurangan	2-1 //1
/	<i>Division</i> atau pembagian	6/2 //3
*	<i>Multiplication</i> atau perkalian	3*2 //6
%	<i>Remainder</i>	3%2 //1
-	<i>Unary Negation</i>	-x // negative x
+	<i>Unary Plus</i>	+x // positive x
++	<i>Pre-Increment</i>	++x
++	<i>Post-Increment</i>	X++
--	<i>Pre-Decrement</i>	--x
--	<i>Post-Decrement</i>	X--

Arithmetic Operation

Contoh *arithmetic operation* dalam *typescript* :

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

**Link* sumber kode.

Modulus Operation

Contoh *modulus operation* dalam *typescript* :

```
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 *typescript* :

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

// 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*.

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 *typescript* :

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

**Link sumber kode.*

Pada kode di atas kita melakukan operasi pengurangan, hasilnya dapat berupa *integer* positif atau *integer* negatif.

Division Operation

Contoh *division operation* dalam *typescript* :

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

**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 *typescript* :

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

**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 *typescript* :

```
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 *typescript* :

```
var a: number = 5, b: number = 5;
a = ++a;
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 8 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

<code><=</code>	Less than or Equal (Lebih kecil atau setara)	<code>x <= 1</code>	<i>False</i>
--------------------	--	------------------------	--------------

Equality Operator

Contoh penggunaan *equality operator* dalam *typescript* :

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

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

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

**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 *typescript* :

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

**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 *typescript*:

```
var x: number = 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 *typescript*.

Less Than Operator

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

```
var x: number = 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 *typescript*.

Greater Than or Equal Operator

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

```
var x: number = 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 *typescript*.

Less Than or Equal Operator

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

```
var x: number = 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 *typescript*.

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 9 Logical Operator

Operator	Description	Comparing	Return
&&	And	$(x > 3 \ \&\& \ y = 5)$	<i>True</i>
 	Or	$(x > 3 \ \ y = 5)$	<i>True</i>
!	Not	$!(x!=y)$	<i>False</i>

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: number = 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: number = 04;
let minute: number = 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 *typescript*. Kita dapat menggunakannya untuk menetapkan sebuah nilai dengan berbagai cara :

Table 10 Assignment Operator

Operator	Description	Example
=	Menetapkan sebuah <i>literal</i> pada sebuah variabel. <i>Literal</i> dapat berupa data tunggal atau sebuah <i>expression</i> .	$C = A \rightarrow C = 12$ $C = A * B \rightarrow C = 12 * 2$

+=	Menetapkan sebuah <i>literal</i> pada <i>operand</i> kiri, hasil dari <i>operand</i> kiri adalah penjumlahan <i>operand</i> kiri dengan <i>operand</i> kanan.	$C += A \rightarrow C = C + A$ $12 += 2 \rightarrow 12 = 12 + 2$
-=	Menetapkan sebuah <i>literal</i> pada <i>operand</i> kiri, hasil dari <i>operand</i> kiri adalah selisih dari <i>operand</i> kiri dengan <i>operand</i> kanan.	$C -= A \rightarrow C = C - A$ $12 -= 2 \rightarrow 12 = 12 - 2$
*=	Menetapkan sebuah <i>literal</i> pada <i>operand</i> kiri, hasil dari <i>operand</i> kiri adalah perkalian <i>operand</i> kiri dengan <i>operand</i> kanan.	$C *= A \rightarrow C = C * A$ $12 *= 2 \rightarrow 12 = 12 * 2$
/=	Menetapkan sebuah <i>literal</i> pada <i>operand</i> kiri, hasil dari <i>operand</i> kiri adalah pembagian <i>operand</i> kiri dengan <i>operand</i> kanan.	$C /= A \rightarrow C = C / A$ $12 /= 2 \rightarrow 12 = 12 / 2$
%=	Menetapkan sebuah <i>literal</i> pada <i>operand</i> kiri, hasil dari <i>operand</i> kiri adalah modulus <i>operand</i> kiri dengan <i>operand</i> kanan.	$C \% = A \rightarrow C = C \% A$ $12 \% = 2 \rightarrow 12 = 12 \% 2$

6. Javascript Strict Mode

Sebelum typescript hadir, terdapat model baru dalam mengeksekusi kode *javascript* yang disebut dengan *Strict Mode*. Mode ini muncul semenjak **ECMAScript** ke **5** muncul, gunanya agar kode *javascript* yang dihasilkan lebih bersih dan baik.

Namun hal ini sudah diperbaiki dengan model static typing yang diberikan oleh typescript. Pada **typescript** kita dapat menambahkan **options** pada **compiler** dengan **flag** `--alwaysStrict` agar kode *javascript* yang diproduksi memiliki `use strict`.

Pembuatan kode javascript tanpa strict mode disebut dengan **normal mode**, sebuah mode yang telah lakukan *javascript* developer sebelum **strict mode** muncul disebut dengan **sloppy mode**.

Tujuan dari **strict mode** agar pembuat kode *javascript* dengan **sloppy mode** bisa diminimalisir, dulunya 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 maintained*) 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 180 Sloppy Mode in the browser

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 menggunakan **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 181 Strict Mode

Untuk *sample code* penggunaan *strict* dalam **node.js** adalah sebagai berikut, kesalahan pada kode di bawah ini juga akan dideteksi oleh **Deno** 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.*

7. Automatic Add Semicolon

Dalam **deno** 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: number = 10
var B: number = 20
console.log(A)
console.log(B)
var C: number = 40;
var D: number = 60;
console.log(C);
console.log(D);
```

**Link* sumber kode.

8. Clean Code Variable Declaration

Avoid Global Variable

Penggunaan *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: string;
var price, discount, fullPrice: number;

firstName = "Maudy";
lastName = "Ayunda";

price = 136.36;
discount = 0.10;

console.log(firstName);
console.log(lastName);
console.log(fullPrice = price * 100 / discount);
```

**Link sumber kode.*

Subchapter 5 – Deno Data Types

Without data, you're just another person with an opinion.

— W. Edward Deming

Subchapter 5 – Objectives

- Memahami Apa itu **Data? Types? & Data Types?**
 - Memahami Apa itu **Javascript Data Types?**
 - Memahami Apa itu **Typescript Type Annotation?**
 - 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 **Data Types** secara maksimal kita harus memahami terlebih dahulu konsep data types dalam *javascript*, ada beberapa konsep fundamental yang harus kita pelajari.

Di antaranya memahami konsep *data*, *types*, *generic variable*, dan **type annotation** yang disediakan oleh typescript dan dapat digunakan dalam **Deno Runtime**. Kajian lainnya membahas tentang *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*).

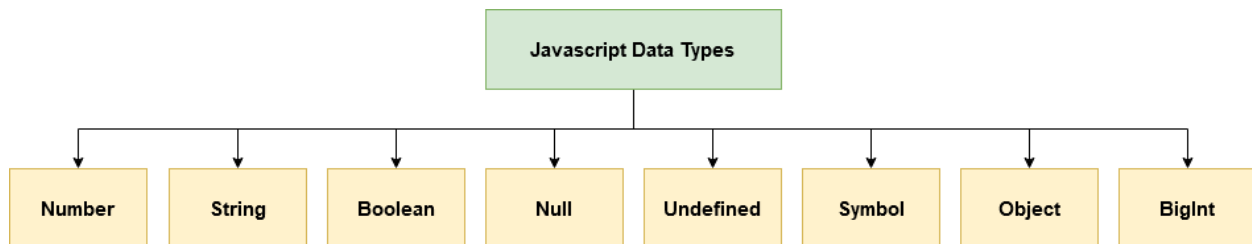
Dalam *javascript* terdapat tiga tipe data dasar yaitu *number*, *string* dan *boolean*. Sebuah variabel dalam *javascript* memiliki *data type* yang spesifik.

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 182 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
*/
```

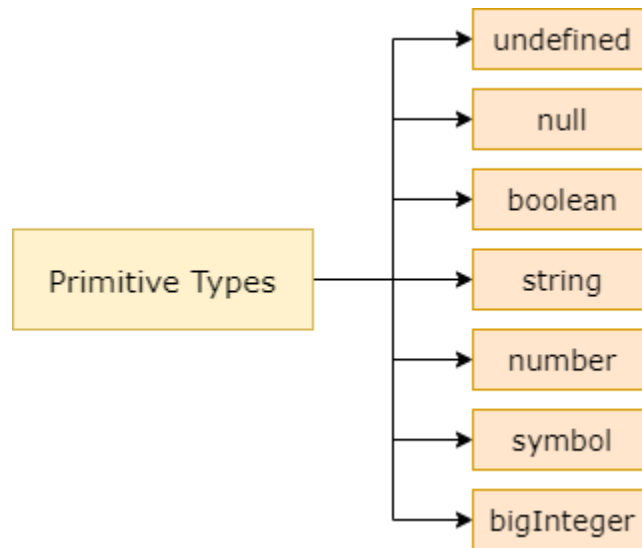
**Link sumber kode.*

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 lambat.

Primitive Types

Seperti yang telah di bahas sebelumnya, **javascript** memiliki 8 **Data Types** dan 7 diantaranya disebut dengan **primitive** atau **primitive value**.



Gambar 183 Primitive Types

Istilah **primitive** digunakan karena hanya menyimpan satu nilai tunggal, data bukan sebuah **object** dan tidak memiliki **method**. Sebelumnya dalam **javascript** kita dapat membuat sebuah **primitive types** tanpa menggunakan **type annotation** :

```
// strings
var name = "Gun Gun Febrianza";

// numbers
var age = 25;
var price = 1.51;

// boolean
var isExist = true;
```

```
// null
var objectx = 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`.

Variable Object	
animal1	Dinosaurus
animal2	Dinosaurus

Gambar 184 Variable Object

Meskipun `animal1` dan `animal2` memiliki nilai yang sama, masing masing menyimpan 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 = "godzilla"
console.log(animal1); //godzilla
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.

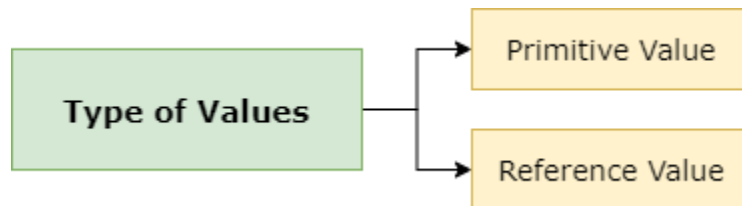
```
var x = "hello"
console.log(x.toUpperCase()); // HELLO
console.log(x); // hello
```

*Link sumber kode.

Meskipun begitu bukan berarti nilai sebuah variabel tidak bisa diubah, kita bisa mengubahnya dengan melakukan **re-binding** melalui menetapkan operasi **assignment** :

```
let str = "Maudy Ayunda";  
str = "Gun Gun Febrianza";  
console.log(str);
```

Apa itu Primitive & Reference Values?



Gambar 185 Type of 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** tujuannya untuk membuat bahasa **javascript** menjadi lebih konsisten.

Saat bahasa pemrograman lainnya 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***.

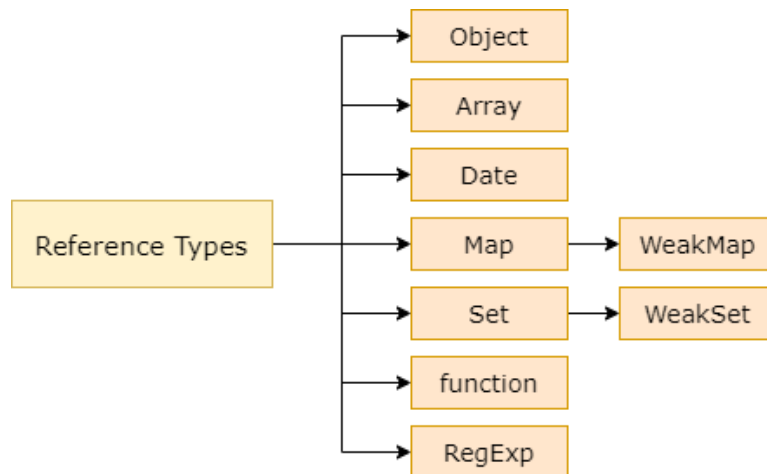
Primitive Values disimpan secara langsung di dalam ***variable object*** & ***Reference Values*** ditempatkan sebagai ***pointer*** di dalam ***Variable Object (VO)***, menjadi referensi lokasi memori tempat ***object*** disimpan.

Reference Types

Dalam *javascript*, ***reference types*** direpresentasikan dengan sebuah ***object***. Sebuah *object* berbeda dengan ***primitive***, sebuah ***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* :



Gambar 186 Reference Types

Built-in Object tersebut juga tersedia dalam *Node.js/Deno*, 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.ucapkanSalam(); // Hello World!
```

**Link* sumber kode.

Pada kode di atas kita membuat *object* bernama **Gun** yang memiliki *properties* **name** dan *method* **ucapkanSalam**, *properties* **name** di isi dengan nilai *string* dan *properties* **ucapkanSalam** 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 kemampuannya 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

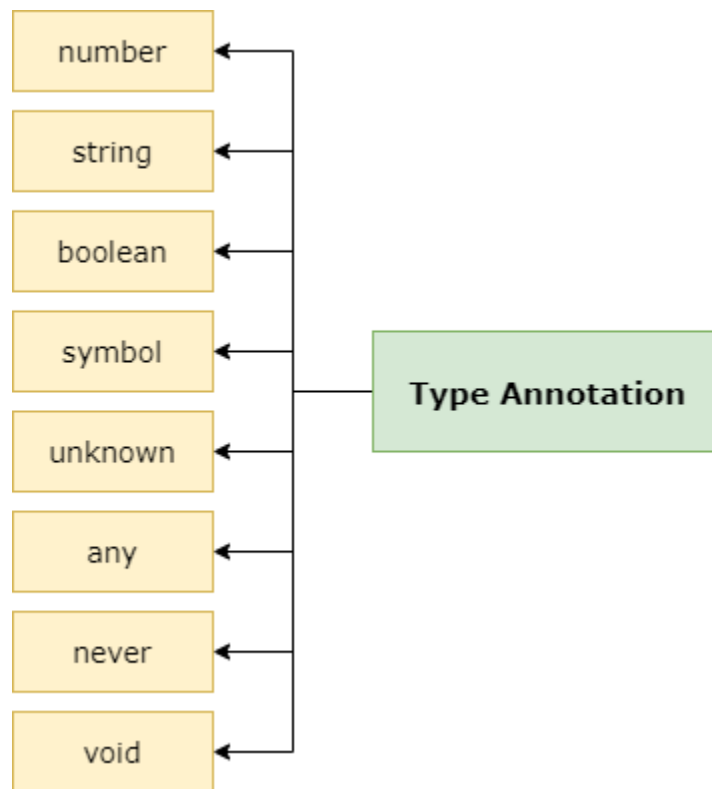
fungsi `console.log()` dipanggil *object wrapper* akan dihapus kembali. Sehingga keasliannya sebagai *primitive type* tetap terjaga.

2. Typescript Data Type

Typescript Type Annotation

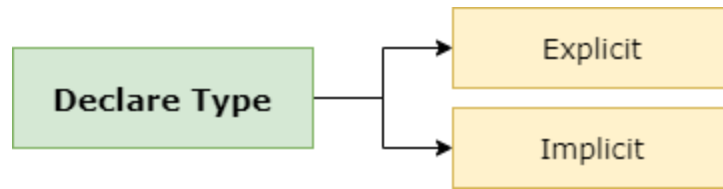
Typescript hadir agar kita melakukan **Static Typing** terlebih dahulu sehingga mencegah kita untuk membuat sebuah **generic variable**. Sekarang kita harus menggunakan **type annotation** agar terbiasa dengan pengembangan aplikasi dalam **Deno runtime**.

Saat kita mendeklarasikan suatu variabel dalam **typescript** kita harus menambahkan sebuah **colon** dan **type annotation**. Di bawah ini adalah **type annotation** yang tersedia di dalam **typescript** :



Gambar 187 Type Annotation

Untuk mendeklarasikan data tipe pada suatu variabel kita dapat melakukannya secara **explicit** atau **implicit** :



Gambar 188 Declare Type

Declare Explicit

Di bawah ini adalah contoh penggunaan ***type annotation*** pada ***primitive data type***, deklarasi dilakukan secara eksplisit dengan menegaskan tipe data yang diberikan :

```
const username: string = "Maudy Ayunda";  
const height: number = 167.13;  
const isCute: boolean = true;
```

Declare Implicit

Di bawah ini adalah deklarasi tipe data secara implisit, ***typescript compiler*** akan memberi tipe data otomatis berdasarkan ***literal*** yang diberikan :

```
const username = "Maudy Ayunda";  
const height = 167.13;  
const isCute = true;
```


3. 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 :

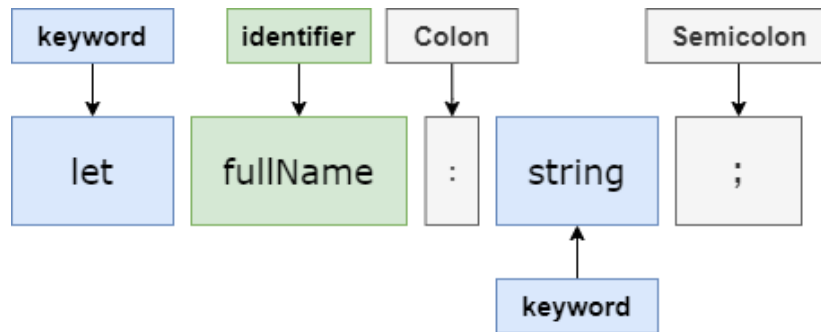
1. **double quotation** mark (" ... "),
2. **single quotation mark** (' ... ') atau
3. *backticks*.

String Type

Contoh **Type Annotation** dalam *typescript* saat deklarasi variabel **string** :

```
let fullName :string;
```

Jika kita perhatikan pada **statement** di atas terdapat struktur diagram sebagai berikut :



Gambar 189 Variable Declaration Structure

Colon diperlukan agar selanjutnya kita bisa menentukan **type annotation** yang akan digunakan. Kita bisa menyimpan nilai **string** ke dalam sebuah variabel menggunakan **double** atau **single quote**. Perhatikan kode di bawah ini :

```
let nama: string = "Gun Gun Febrianza";
nama = 'Gun Gun Febrianza';
```

Template String

Typescript juga mendukung **template string** menggunakan **syntax** `${expression}`. Perhatikan kode di bawah ini :

```
var nama: string = "Gun Gun Febrianza";
let doa: string = `Hallo ${nama}, kami doakan semoga anda selalu sehat.`;
```

Escaping

Typescript juga memiliki *escape sequences* yang bisa kita gunakan pada REPL, pertama kita bisa membuat *newline* :

```
console.log("Maudy \n Ayunda \n Faza");
```

Di bawah ini *escape sequences* untuk membuat *single* atau *double quotation mark character* :

```
console.log("Maudy \" Ayunda \" Faza");  
console.log("Maudy \' Ayunda \' Faza");
```

Di bawah ini *escape sequences* untuk membuat tab :

```
console.log("Maudy \t Ayunda \t Faza");
```

Kita juga bisa menggunakan *special character* dan *unicode **literal*** :

```
console.log('\xA9');
```

String Concatenation

Jika kita menjumlahkan *literal number* dalam bentuk *string*, maka akan memproduksi *string concatenation* :

```
let s1: string = "5";  
let s2: string = "15";  
console.log(s1 + s2); //515
```

String Interpolation

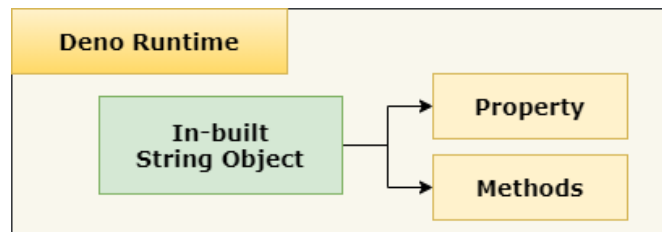
Di dalam *typescript* juga bisa melakukan *string interpolation* seperti dalam *python* menggunakan *syntax* di bawah ini :

```
var x1: number = 3;
var x2: number = 2;

console.log("Jumlah dari x1 dan x2 adalah " + (x1 + x2));
```

String Object

Dalam **javascript** segala sesuatu adalah **object**, semua sudah di design dalam setiap **runtime engine** khusus untuk **javascript**. Begitu juga pada **deno**, terdapat **in-built string object** yang memiliki **property** & **methods**.



Gambar 190 String Object

String Property

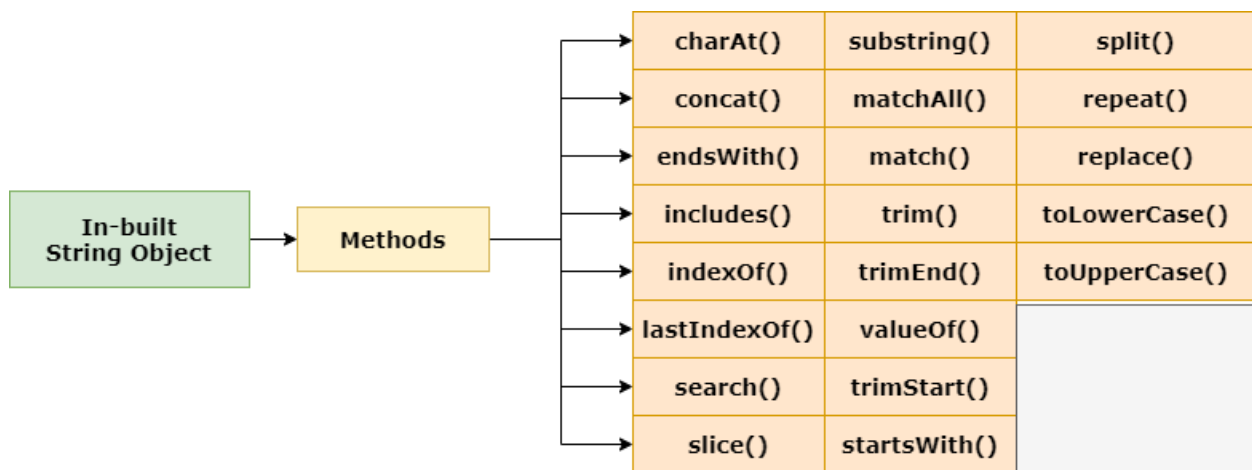
Length Property

Sebelumnya kita telah belajar konsep **object-wrapper**, jadi jangan heran kenapa **variable name** sebagai **primitive type** dapat memiliki sebuah **property** :

```
let name: string = "Maudy Ayunda"
console.log(name.length); //12
```

String Methods

Untuk **object string** sendiri terdapat sekumpulan **methods** yang bisa kita gunakan dan eksplorasi, di bawah ini adalah beberapa **methods** untuk **string** yang bisa digunakan dalam **Deno Runtime** :



Gambar 191 String Methods

Method charAt()

Pada kode di bawah ini kita menggunakan *method charAt* untuk mengetahui suatu karakter berdasarkan *index*.

```
let maudy : string = "Maudy"
console.log(maudy.charAt(0)); //M
console.log(maudy.charAt(1)); //a
console.log(maudy.charAt(2)); //u
```

Method indexOf()

Guna dari *method indexOf* untuk membaca *index* berdasarkan karakter,

Method startWith()

Guna dari *method* **startWith** untuk membaca karakter awal

```
let Maudy: string = "Maudy";  
console.log(Maudy.startWith('M')); //true
```

Method endsWith()

Guna dari *method* **endsWith** membaca karakter akhir dengan *return* berupa *boolean* (*true* or *false*).

```
let Maudy: string = "Maudy";  
console.log(Maudy.endsWith('y')); //true
```

Method includes()

Guna dari *method* **includes** untuk memastikan karakter tersebut terdapat pada *string*.

```
let Maudy: string = "Maudy";  
console.log(Maudy.includes('d')); //true
```

Method split()

Selain itu juga terdapat *method* **split** untuk memisahkan *string* kedalam bentuk *array*, seperti pada kode di bawah ini :

```
const name: string = "Hello Maudy Ayunda";  
  
const arrayString = name.split(" ");  
console.log(arrayString[2]); //ayunda
```

```
const arrayChars = name.split("");  
console.log(arrayChars[4]); //0
```

4. 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 π tidak bisa direpresentasikan secara akurat.

Dalam *algebra*, *symbol* π termasuk kedalam *irrational number*. Berbeda dengan bahasa pemrograman lain *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.

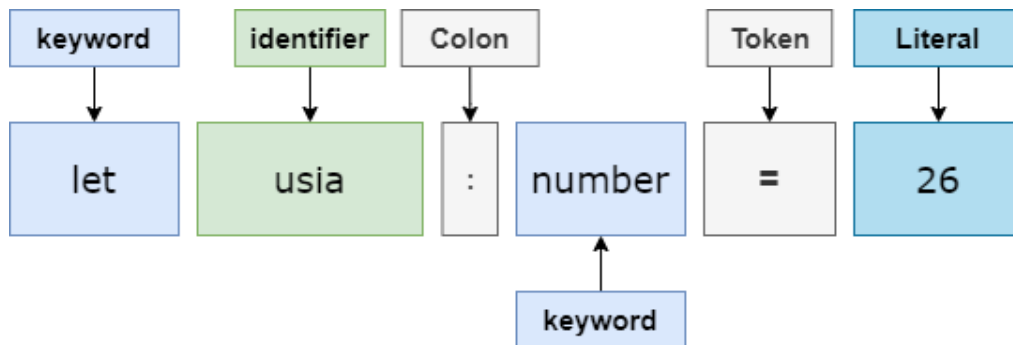
```
var x: number = 99.8
console.log(x); //99.8
var y: number = 99
console.log(y); //99
```


Number Type

Sebuah angka di dalam *typescript* merupakan tipe data *number*.

```
let usia: number = 26;
```

Jika kita perhatikan pada **statement** di atas terdapat struktur diagram sebagai berikut :



Gambar 192 Number Data Type

JavaScript adalah bahasa dengan karakteristik **types** : **Loosely typed language** yaitu tidak mengenal **type** data seperti **integer, short, long** atau **float** dan seterusnya.

Begitu juga di dalam **typescript** seluruh **number** representasinya di dalam sistem diubah menjadi **64-bit floating point**. Namun begitu kita tetap bisa menyimpan sebuah nilai angka desimal dalam variabel **typescript**.

```
let tinggi: number = 900.888;
```

Infinity

Infinity merepresentasikan *mathematical Infinity* dengan notasi ∞ . Dalam *javascript* ini adalah nilai spesial yang nilainya selalu paling besar. Untuk mendapatkannya dalam *javascript* kita bisa mengeksekusi kode sebagai berikut :

```
// Angka yg dibagi 0 (zero) memproduksi Infinity:
var x: number = 2 / 0; // x memproduksi Infinity
var y: number = -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: number = 2;
while (myNumber != Infinity) {
    // Execute until Infinity
    console.log((myNumber = myNumber * myNumber));
}
/* Output :
4
16
256
65536
4294967296
18446744073709552000
3.402823669209385e+38
1.157920892373162e+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: number) {
```

```

    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

Pada **typescript** **NaN** dapat dicegah karena terdapat dukungan **static typing**, namun di level **javascript** yang berbasis **dynamic typing** permasalahan **NaN** dapat terjadi. Di bawah ini adalah contoh kode bagaimana **typescript compiler** mendeteksi **NaN error** :

```

var x: number = 10 / "Maudy";

```

Jika kode di atas di eksekusi menggunakan deno maka akan menghasilkan error sebagai berikut :

```

error: TS2363 [ERROR]: The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.
var x: number = 10 / "Maudy";

```

Gambar 193 Error Example

Untuk penjelasan lebih detail mengenai **NaN** simak kajian di bawah ini.

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*.

```
var x = 100 / "Maudy"  
console.log(x)  
//NaN
```

Pada kode di atas hasilnya **NaN** karena kita mencoba membagi *number* dengan *string* sehingga memproduksi *computation error*.

NaN Computation

Operasi aritmetika pada **NaN** akan selalu memproduksi **NaN**:

```
var a = NaN;  
console.log(a + 50);  
//NaN
```

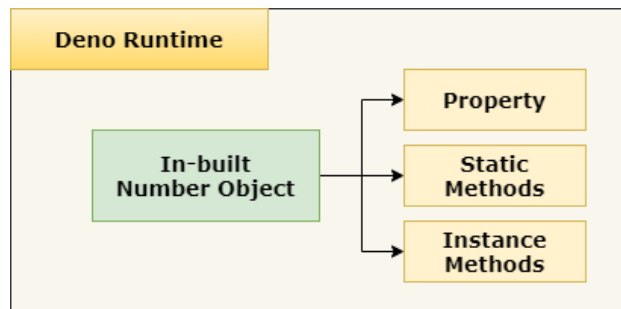
NaN Equality Check

Meskipun begitu *javascript engine* memperlakukan **NaN** (*Not a Number*) sebagai tipe data *number*, dengan karakteristiknya yang aneh.

```
console.log(typeof NaN);  
//"number"  
console.log(NaN === NaN);  
//false
```

Number Object

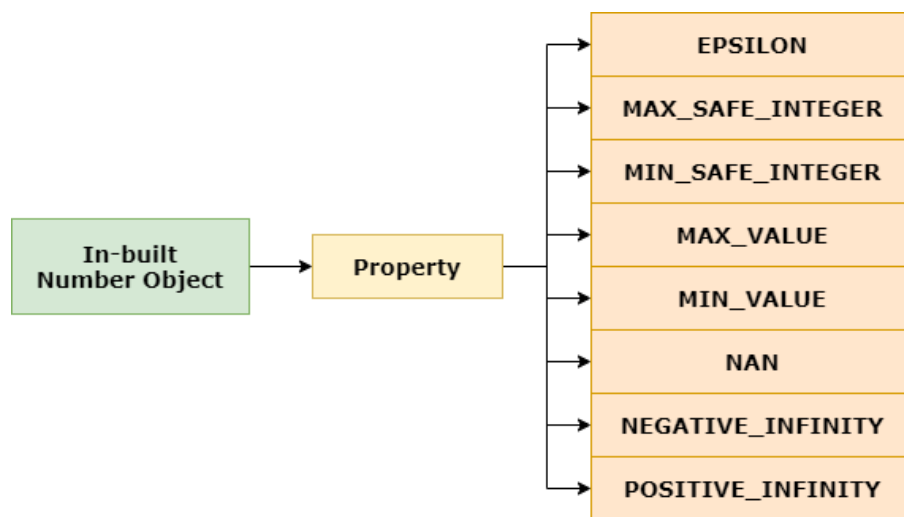
Dalam **javascript** segala sesuatu adalah **object**, semua sudah di design dalam setiap **runtime engine** khusus untuk **javascript**. Begitu juga pada **deno**, terdapat **in-built number object** yang memiliki **property** & **methods**.



Gambar 194 Number Object

Number Property

Sebelumnya kita telah belajar konsep **object-wrapper**, jadi jangan heran kenapa **variable name** sebagai **primitive type** dapat memiliki sebuah **property** :



Gambar 195 Number Property

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: number, y: number) {  
  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 = Number.MAX_SAFE_INTEGER + 1;
var y: number = 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.

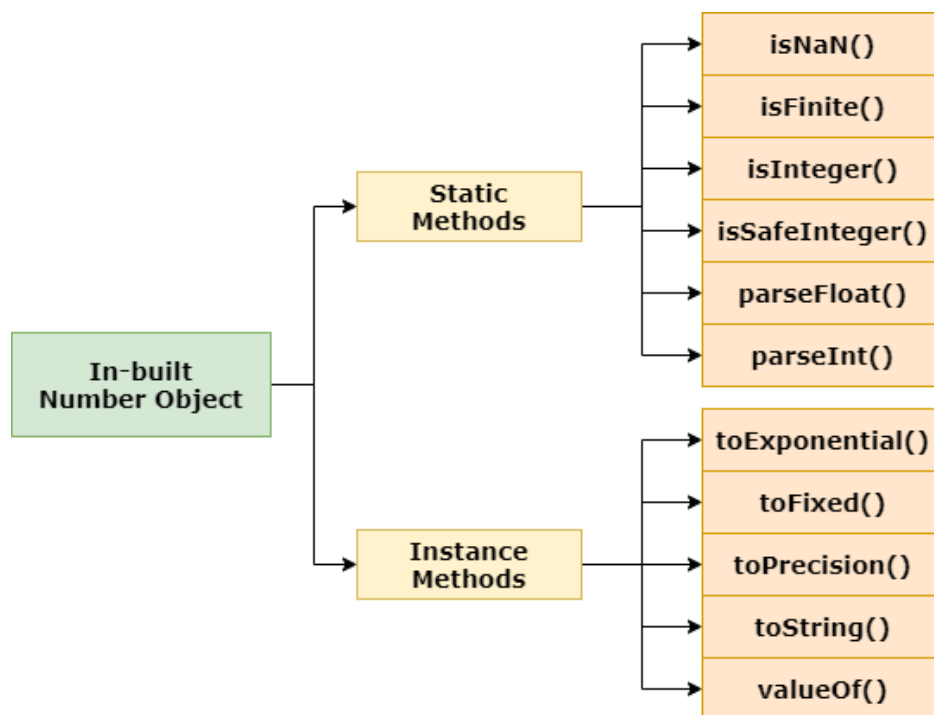
Pada kode di bawah ini, operasi dilakukan didalam ranah *safe integer* sehingga bisa melakukan perbandingan matematis secara akurat :

```
var x: number = Number.MAX_SAFE_INTEGER;
var y: number = 9007199254740990;
console.log(Number.MAX_SAFE_INTEGER);
//9007199254740991
```

```
console.log(y);  
//9007199254740990  
console.log(x === y);  
//false
```

Number Methods

Untuk **object number** sendiri terdapat sekumpulan **methods** yang bisa kita gunakan dan eksplorasi, di bawah ini adalah beberapa **methods** untuk **string** yang bisa digunakan dalam **Deno Runtime** :



Gambar 196 Object Number Methods

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(3.1);         // false
Number.isSafeInteger(3.0);         // true
```

**Link sumber kode.*

Positive e Notation

Untuk menghindari kesalahan penulisan *number*, apalagi jika terdapat angka nol dan angka lainnya yang sangat panjang. Untuk mengatasi hal ini kita bisa mempersingkatnya dengan menggunakan ***e notation*** :

```
let billion: number = 1e9; //1 billion (1 & 9 zero)
//Contoh lainnya :
// 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: number = 0.000001;
//implicit zero
ms = 1e-6;

//1e-3 = 1 / 1000
//1.23e-6 = 1.23 / 1000000
```

*Link sumber kode.

Rounding

Dalam *javascript 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*

<i>Literal</i>	<i>Math.floor</i>	<i>Math.ceil</i>	<i>Math.round</i>	<i>Math.trunc</i>
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*.

```
let num = new Number(27.123456);
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); // "27.1246"
num.toPrecision(8); // "27.123456"
console.log(num.toPrecision(9)); // "27.1234560"
console.log(3e+1); // 30
```

Exponentiation

Jika kita ingin membuat *number* menampilkan notasi eksponensial, kita bisa menggunakan *method* `toExponential()` dan *e notation* seperti pada gambar di bawah ini :

```
let num: number = 1225.30;
console.log(num.toExponential()); // "1.2253+3"
console.log(num.toExponential(1)); // "1.22e+3"
console.log(num.toExponential(2)); // "1.23e+3"
console.log(num.toExponential(4)); // "1.22253e+3"
console.log(num.toExponential(5)); // "1.22530e+3"
console.log(1.2e+3); // 1200
console.log(1.23e+3); // 1230
console.log(1.2253e+3); // 1225.3
console.log(1.22530e+3); // 1225.3
```

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 :

```
console.log(10000000000000); // thousand bn
// Output10000000000000
console.log(100000000000000);
// Output100000000000000
console.log(1000000000000000);
// Output1000000000000000
console.log(10000000000000000);
// Output10000000000000000
console.log(100000000000000000);
```

```
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput10000000000000000000
console.log(10000000000000000000);
// Ouput1e+21
```

Untuk **1e21** dan **1e20** :

```
console.log(1e21);
//1e+21
console.log(1e20);
//1000000000000000000000
```

Jika *number* terlalu besar dieksekusi maka akan terjadi *overflow*, berpotensi mendapatkan *infinity*.

```
console.log(1e500); // infinity
console.log(0.1 + 0.2 == 0.3); //false
```

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: number = 9999999999999999;
var y: number = 9999999999999999;

console.log(x);
//9999999999999999
console.log(y);
//100000000000000000
```

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 197 Number Representation

Maksimum jumlah *digits* ada angka desimal adalah *17 digits*, namun *floating point arithmetic* tidak selalu menghasilkan penjumlahan yang 100% akurat:

>> var x = 0.2 + 0.1; ➡ 0.30000000000000004

Gambar 198 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 sebenarnya 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*)** $0.33333(3)\dots$

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.

```
console.log(0.1.toFixed(20));  
//0.10000000000000000555
```

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](#).

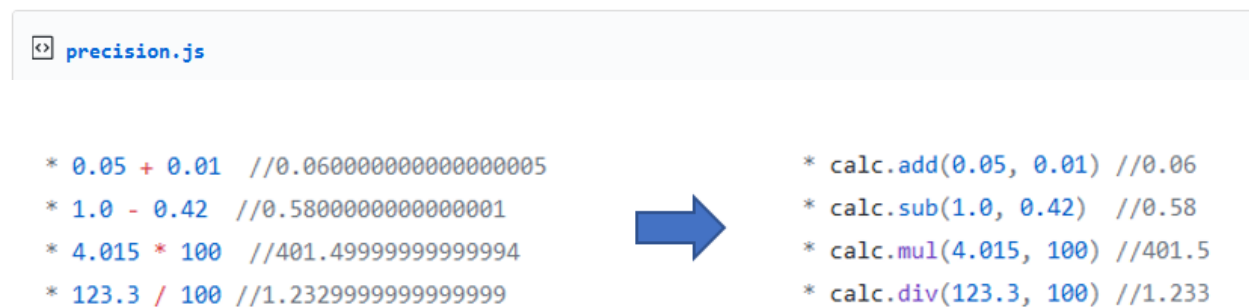
Solution to Imprecise

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 $0.2 \times 10 = 2$, 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.



Gambar 199 Precision.js Result

Fixed Number

Format *number* dengan membuat *digit* yang spesifik. *Method* di bawah ini menghasilkan *return* berupa *string*:




```
console.log(0.1 + 0.2);  
//0.30000000000000004  
console.log((0.1 + 0.2).toFixed(2));  
//0.30  
console.log(typeof (0.1 + 0.2).toFixed(2));  
//string
```

Jika ingin mengubahnya kedalam *number* gunakan *unary plus* :

```
let sum = 0.1 + 0.2;  
console.log(sum.toFixed(2));  
//0.3  
  
console.log(typeof +sum.toFixed(2));  
//"number"  
  
console.log(typeof sum.toFixed(2));  
//"string"
```

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 :

Math.abs(x)

Returns the absolute value of a number.

Math.max([x[, y[, ...]]])

Returns the largest of zero or more numbers.

Math.pow(x, y)

Returns base to the exponent power, that is, base^{exponent}.

Math.random()

Returns a pseudo-random number between 0 and 1.

Math.sign(x)

Returns the sign of the x, indicating whether x is positive, negative or zero.

methods →

$$\text{Math.abs}(x) = |x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

properties →

$$\text{Math.SQRT2} = \sqrt{2} \approx 1.414$$

Gambar 200 Math Objects

Hexadecimal, Binary dan Octadecimal

Typescript juga mendukung **number-literal** dalam bentuk **hex**, **binary** dan **octal**.

Perhatikan kode dibawah ini dan **prefix** yang digunakan :

```
let dec: number = 27;
let hex: number = 0x001b;
let binary: number = 0b11011;
let octal: number = 0o0033;
```

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/**

5. Booleans Data Types

Boolean Type

Di dalam *javascript*, *boolean* adalah sebuah **primitive data type**. Sebuah tipe data yang hanya memiliki dua nilai yaitu *True* dan *False Keywords*. Kita bisa membuat *primitive boolean* dengan memberinya nilai *true* atau *false*.

Untuk membuat sebuah variabel yang dapat merepresentasikan nilai logika kita perlu menggunakan **type annotation boolean** :

```
let isCute: boolean = true;
```

Pada kode di atas kita memberikan tipe data **boolean** secara eksplisit dan di bawah ini kita memberikan tipe data **boolean** secara implisit :

```
var yes = true;  
var no = false;
```

6. Inferred Type

Sebuah **type** yang diberikan pada suatu variabel dapat diatur secara eksplisit oleh seorang **software developer**, atau secara implisit menggunakan **typescript compiler**. Pemberian **type** secara implisit disebut dengan **inferred type** dalam **typescript**.

Pada kode di bawah ini kita tidak memberikan **type** pada variabel harga, namun karena kita memberikan sebuah **literal number** maka **typescript compiler** akan mengetahui bahwa tipe data diberikan secara implisit yaitu sebuah **number**.

```
let harga = 9000;  
console.log(harga); //9000
```

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.

Dynamic Typed

Pada kode di bawah ini, 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*.

Sebagai contoh :

```
let angka = 99;  
angka = "Hi Maudy";
```

```
console.log(angka); //"Hi Maudy"
```

Static Typed

Pada ***typescript*** penulisan di atas dilarang dan tidak bisa dilakukan, karena ***typescript*** mendukung ***static typing***.

```
let number: number = 99;  
let result :string = "Hi Maudy";  
console.log(result); //"Hi Maudy"
```

Ada cara lain yaitu menggunakan ***Any Type*** yang akan anda pelajari dihalaman berikutnya, namun hal ini sangat tidak disarankan.

7. Type Conversion

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: number = parseInt("99");  
console.log(maudy); //99  
console.log(maudy + 1); //100
```

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: number = parseFloat("99.8");  
console.log(maudy); //99.8
```

Number to String

Kita juga dapat melakukan konversi *number integer* ke dalam *string* :

```
var a: number = 10  
var b: string = String(a)  
console.log(b); //"10"
```

Decimal Number to String

Kita juga dapat melakukan konversi *decimal number* ke dalam *string* :

```
var c: number = 22.3
var d: string = String(c)
console.log(d); //"22.3"
```

Boolean to String

Kita juga dapat melakukan konversi *boolean* ke dalam *string* :

```
var bool: boolean = true;
console.log(String(bool)) //true
```


8. Check Data Type

Primitive Types

Untuk memeriksa **primitive type** kita dapat menggunakan **method** `typeof()` seperti pada kode di bawah ini :

```
let stringPrimitive: String = "hi";
let numberPrimitive: Number = 90;
let boolPrimitive: Boolean = true;
let nullPrimitive: null = null;
let undefinedPrimitive: any;
let bigintPrimitive: BigInt = 2323n;

console.log(typeof stringPrimitive);
console.log(typeof numberPrimitive);
console.log(typeof boolPrimitive);
console.log(typeof nullPrimitive);
console.log(typeof undefinedPrimitive);
console.log(typeof bigintPrimitive);
```

Jika kode di atas di eksekusi maka akan memproduksi **output** :

```
/*
string
number
boolean
object
undefined
bigint
*/
```

Pada kode di atas hasil dari pengujian `typeof()` `null` memberikan **ouput object**, namun dalam **javascript** tetap diperlakukan seperti **primitive literal**. Anda akan mempelajari alasan kenapa `null` memproduksi **object** pada **chapter** selanjutnya tentang **type widening**.

Jika kita menggunakan `typeof` pada **reference type** maka akan dianggap sebuah **object**, kecuali **function** dia akan diperlakukan sebagai sebuah **function**. Namun begitu dalam **javascript function** juga adalah sebuah **object** :

```
const map1 = new Map();
const array = ["Hi", "Maudy"];
const object1 = {};
function reflect(param: any): any {
    return param;
}
console.log(typeof map1);
console.log(typeof array);
console.log(typeof object1);
console.log(typeof reflect(() => {}));
```

Jika kode di atas di eksekusi maka akan memproduksi **output** :

```
/*
object
object
object
function
*/
```

Reference Types

Untuk memeriksa apakah klasifikasi suatu **object** kita dapat menggunakan **keyword instanceof** seperti pada kode di bawah ini :

```
const map = new Map();
const items = ["Hi", "Maudy"];
const object = {};
function reflect(value: string) {
  return value;
}
console.log(map instanceof Map);
console.log(items instanceof Array);
console.log(object instanceof Object);
console.log(reflect instanceof Function);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

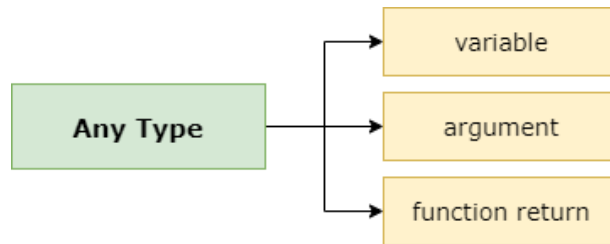
```
/*
true
true
true
true
*/
```

Untuk mendeteksi **type array**, lebih spesifik lagi **ES 6** telah membuat spesifikasi **isArray** yang dapat kita gunakan untuk memeriksa sebuah **array** :

```
const exarray: string[] = [];
console.log(Array.isArray(exarray));
//true
```

9. Any Type

Setiap **variable**, **argument** dan **function return** yang tidak diberikan **type** secara eksplisit akan diperlakukan sebagai **any type**. **Any type** artinya kita dapat memiliki tipe apa saja tergantung dari **literal value** yang diberikan.



Gambar 201 Any Type

Pada **typescript** setiap **variable**, **argument** dan **function return** memerlukan sebuah **type** saat kompilasi dilakukan. Variabel **y** di bawah ini akan diperlakukan sebagai **any type** :

```
var y; // Same as y: any
```

Jika kita memberikan **literal number** pada sebuah **any type** maka tipe datanya adalah **number** :

```
y = 10
```

Jika kita memberikan **literal string** pada sebuah **any type** maka tipe datanya adalah **string** :

```
y = "Maudy Ayunda"
```

Termasuk ketika kita memberikan **literal true** atau **false** pada **any type** maka tipe datanya adalah **boolean** :

```
y = true
```

Sebagai **developer** yang memahami filosofi **typescript** diusahakan kita harus menghindari penggunaan **Any type**, kenapa? Ingat lagi penggunaan **type** untuk apa? Kita membutuhkan suatu **type** untuk tujuan yang jelas.

Any type membuat kode yang kita tulis akhirnya kembali ke level **javascript** yang berbasis **dynamic typing**, karena **typechecker** tidak digunakan untuk memecahkan problema **static typing**. Gunakan **any type** sebagai solusi terakhir jika memaksa.

Penggunaan **any type** dalam **typescript** :

```
var x: any = 4; // Explicitly typed
x = "Bisa sebuah string";
x = false;
```

Terkadang kita membutuhkan sifat **dynamic typing** bukan hanya **static typing** seperti yang telah kita lakukan setiap kali hendak membuat sebuah variabel. Dalam **typescript** kita bisa menggunakan **any-type**.

```
let diaCantik: any = true; // inisialisasi dengan boolean
console.log(typeof diaCantik);

diaCantik = "benar"; // nilai variabel diubah menjadi string
console.log(typeof diaCantik);
```

Pada kode di atas **typeof** adalah **keyword** yang digunakan untuk mengetahui tipe data suatu variabel. **Any-type** yang digunakan akan membuat kompiler untuk tidak memberikan **semantic error** saat memeriksa **type** yang hendak digunakan.

Parameter Any Type

Any-type dapat digunakan sebagai contoh saat kita menghadapi kasus dimana kita membutuhkan **function-parameter** dengan karakteristik **argument** yang bersifat implisit atau tidak diketahui.

Di bawah ini adalah contoh **function** dengan parameter implisit. Dikatakan implisit karena kita tidak akan pernah tau apa saja nilai yang akan dimasukkan kedalam **parameter**, karena **parameter** tidak memiliki **type** :

```
function tulisNama(teman:any) {  
    console.log(teman.namaKepanjangan);  
}
```

Option noImplicitAny

Untuk mencegah penggunaan **any type** dan memastikan setiap variabel memiliki tipe data yang jelas kita dapat menggunakan **options noImplicitAny** dengan nilai **true**.

10. Type Union

Ada saatnya kita menginginkan variabel yang bisa disimpan dengan berbagai **type**, selain menggunakan **type any** kita bisa menggunakan **union**. Sebuah **union type** digunakan untuk mengekspresikan **type** kombinasi dari berbagai **type**.

Kode di bawah ini adalah variabel dengan kemampuan untuk bisa menampung dua **type** data sekaligus yaitu sebagai **boolean** atau **number**.

```
let isVisible : boolean|number = true;
isVisible = 1; // OK
isVisible = "yes"; // akan menghasilkan error
```

Parameter Union Type

Union-type dapat digunakan saat kita menghadapi kasus yang membutuhkan **function-parameter** dengan karakteristik **argument** dengan berbagai tipe data yang telah kita tentukan sebelumnya.

Pada kode di bawah ini kita menerima **function-parameter** dengan **boolean** dan **number** :

```
function tulisAngka(angkaAtauLogika: boolean | number) {
  console.log(angkaAtauLogika);
}
```

11. Symbol Type

Symbol adalah sebuah **primitive data type** baru baru dalam *ECMAScript 6* yang merepresentasikan *token* unik dan memiliki karakteristik *immutable*.

Pada kode di bawah ini kita membuat *variable constant* dengan *identifier* **symbol**, setelah itu melakukan operasi *assignment* memberinya nilai *return* dari *symbol function*. *Return* yang dihasilkan secara ***under the hood*** berupa *id unique*.

```
const symbol = Symbol();
console.log(String(symbol));
// Symbol()
```

Kenapa *unique*? Karena setiap *symbol* menghasilkan identitas berbeda (yang tidak bisa kita lihat). Kenapa *id unik* dalam ***symbol*** tidak terlihat? Karena *id unik* tersebut direpresentasikan secara internal dibelakang layar tanpa kita ketahui.

Namun kita bisa membuktikannya dengan mengeksekusi kode di bawah ini, jika memang setiap ***symbol*** memiliki identitas yang berbeda-beda maka operasi perbandingan harus bernilai ***false*** :

```
var sym1 = Symbol("test");
var sym2 = Symbol("test");
console.log(sym1 === sym2);
// false
```

Untuk memudahkan pembacaan (*Code Readability*) kita bisa menggunakan *parameter* yang dimiliki *symbol function*, kita bisa melakukannya seperti pada kode di bawah ini:

```
const symbol1 = Symbol('symbol1');
console.log(String(symbol1));
```


Biasanya ***symbol*** digunakan untuk membuat sebuah kunci unik sebagai ***properties*** dari sebuah ***object***, sebagai contoh :

```
const userIdentity = Symbol("IDuser");

const client = {
  userIdentity: "888",
};

console.log(client["userIdentity"]); //888
```

12. Type Widening

Dalam **javascript** untuk proses komputasi **undefined** dan **null** adalah **literal value** yang seringkali menjadi penyebab **error**.

Pada **typescript** jika kita mendeklarasikan suatu variabel tanpa memberikan **value**, maka secara **internal typescript** akan memberikannya **type null** atau **undefined** yang selanjutnya dikonversi kedalam **type any**.

Typescript compiler mendukung **options** `strictNullCheck` yang melarang penggunaan **type null** ke dalam variabel. Kode di bawah ini akan mengalami **error** jika **options** tersebut di aktifkan :

```
let maudy = 232;
maudy = null; //compile error
maudy = undefined; //compile error
```

Undefined

Dalam *Javascript* sebuah *variable* yang tidak memiliki nilai secara otomatis memiliki *Data Types Undefined*.

```
var ayunda;
console.log(ayunda); //undefined
```

Perbedaan antara *null* dan *undefined* adalah *null variable* harus dideklarasikan secara eksplisit sementara *undefined* hanya ada pada *variable* yang tidak dideklarasikan :

```
console.log(typeof undefined);
//"undefined"
```

```
console.log(typeof null);  
// "object"  
console.log(null === undefined);  
// false  
console.log(null === undefined);  
// true
```

Null

Dalam *Javascript*, **Null** artinya *nothing* yaitu sesuatu yang tidak ada. Meskipun begitu dalam *javascript* *null* adalah sebuah *object*.

```
var maudy = null;  
console.log(typeof maudy); //object
```

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, membetulkannya 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>

Option strictNullCheck

Pada ***typescript*** jika ***assignment statement*** di bawah ini dianggap valid :

```
let testNull: number = null;
```

Untuk mencegahnya kita dapat memberikan ***option strictNullCheck***.

Jika kita ingin memberikan ***null*** maka kita harus menggunakan ***type union*** :

```
let testNull: number | null = null;  
console.log(testNull);
```

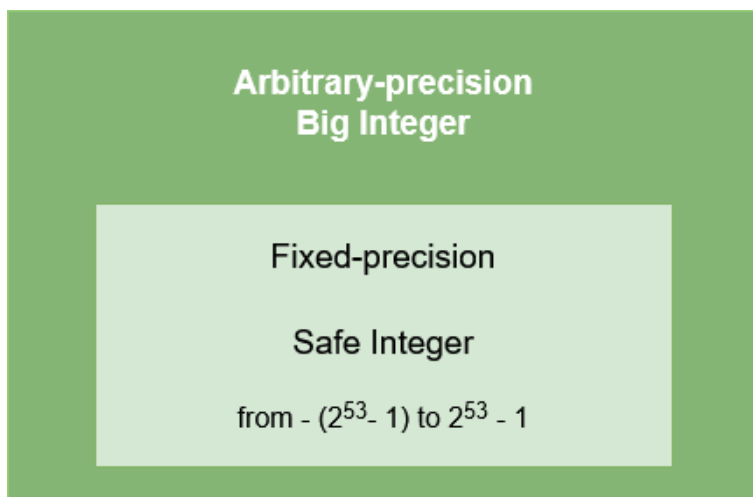
13. BigInt Data Types

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 202 Arbitrary & fixed Precision

Untuk mendeklarasikan variabel dengan tipe data *big integer* perhatikan kode di bawah ini :

```
const bigInt1: bigint = 1234567890123456789012345678901234567890n;
const bigInt2: bigint = BigInt("1234567890123456789012345678901234567890");
const bigInt3: bigint = BigInt(1234567890123456789012345678901234567890);
```

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 primitif *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(1234567890123456789012345678901234567890n);
// output : 1234567890123456789012345678901234567890n
console.log(typeof 1234567890123456789012345678901234567890n);
// 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 melakukannya :

```
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.*

14. Custom Type

Dengan **type keyword** kita dapat membuat **type** baru atau sebuah **type alias**. Di bawah ini kita membuat sebuah **type alias** :

```
type Balance = number;
type Type = string;
```

Selanjutnya kita membuat sebuah **type** baru :

```
type Wallet = {
  name: string;
  amount: Balance;
  symbol: Type;
};
```

Pada **type Wallet** di atas kita menggunakan dua buah **type alias** untuk **properties amount** dan **symbol**. Selanjutnya kita bisa membuat sebuah variabel dengan tipe data **Wallet** :

```
let bitcoinWallet: Wallet = {
  name: "Bitcoin",
  amount: 5.88800007,
  symbol: "BTC",
};
```

Selanjutnya setiap kali kita membuat variabel dengan tipe **Wallet** pastikan kita mengisi setiap **properties** yang tersedia jika tidak maka hasilnya akan **error**. Jika tidak ingin **error** terjadi maka kita harus menggunakan optional **properties** menggunakan **questional mark**.

Perhatikan kode di bawah ini :

```
type CryptoWallet = {  
  name: string;  
  amount: Balance;  
  symbol?: Type;  
};  
  
let ethereumWallet: CryptoWallet = {  
  name: "Ethereum",  
  amount: 23.88800007,  
};
```

Property symbol menggunakan **question mark**, jadi jika kita tidak mengisi **property symbol** saat membuat variabel **error** tidak akan terjadi.

15. 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 :

```
let x: string = "Gun Gun Febrianza";  
let 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)*.

Subchapter 6 – Control Flow

*The most important property of a program is
Whether it accomplishes the intention of its user.*

— C.A.R Hoare

Subchapter 6 – 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* atau **Standalone Application** yang dibuat dengan *typescript* menjadi lebih interaktif.

Ada beberapa hal yang akan kita bahas disini di antaranya adalah :

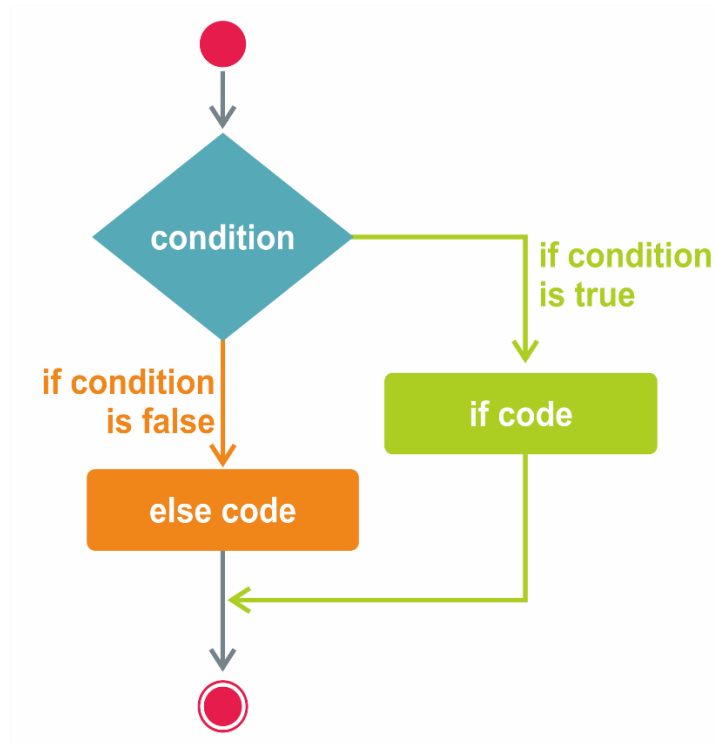
1. Block Statements

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 cara agar program yang kita tulis dapat menentukan pilihan berdasarkan ekspresi logika yang diberikan, sebuah **block statement** di dalam **if** akan dieksekusi jika kondisi bernilai *true*.



Gambar 203 Conditional Statement

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 :

```
let time: number = 20;
let statement: string;
if (time < 20) {
  statement = "statement_1";
} else {
  statement = "statement_2";
}
```

Keluaran dari kode di atas adalah :

```
//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 204 Syntax Ternary Operator

Variabel **result** akan mendapatkan nilai berdasarkan kondisi yang diberikan, jika hasil kondisi bernilai **true** maka kode yang akan dieksekusi adalah **value1** dan jika kondisi bernilai **false** maka kode yang akan dieksekusi adalah **value2**.

```
let usia: number = 19;  
let izin: boolean = (usia > 19) ? true : false;
```

Bagaimana jika kita ingin menggunakan lebih dari satu kondisi menggunakan *ternary operator*? Kita bisa melakukannya perhatikan kode di bawah ini :

```
let message = (age < 3) ? 'Hi, baby!' :  
  (age < 18) ? 'Hello! Sweetie' :  
  (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 :

```
let time: number = 44;  
let x: string;  
if (time < 20) {  
    x = "Statement_1";  
} else if (time == 20) {  
    x = "Statement_2";  
} else {  
    x = "Statement_3";  
}
```



```
}
```

Keluaran dari kode di atas adalah :

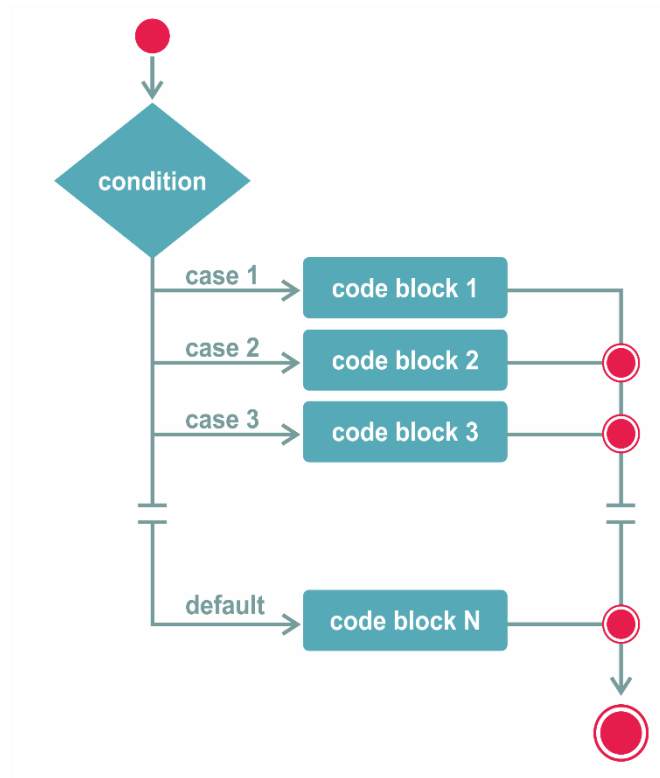
```
//output  
//"Statement_3"
```

**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.



Gambar 205 Switch Flowchart

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 kode di bawah ini :

```
let day: number = 2;
let x: string;
switch (day) {
    case 0:
        x = "Hari ini minggu";
        break;
    case 1:
        x = "Hari ini senin";
        break;
    case 2:
        x = "Hari ini selasa";
        break;
}
```

Keluaran dari kode di atas adalah :

```
//ouput
//"Hari ini selasa"
```

**Link sumber kode.*

Label yang digunakan sebagai kondisi adalah **number**, karena nilai *number* adalah 2 maka *statement* di dalam *case 2* yang akan di eksekusi. Keyword **break** digunakan untuk menghentikan *statement* agar *case* berikutnya tidak dieksekusi.

Subchapter 7 – Loop & Iteration

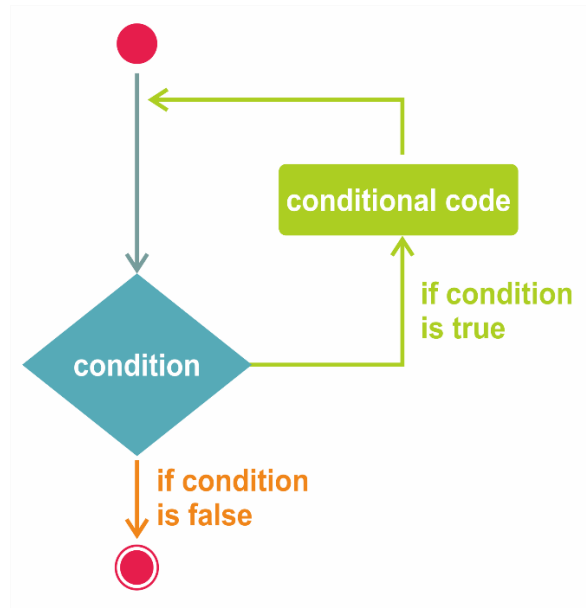
*Programmers are not to be measured by their ingenuity and their logic
but by the completeness of their case analysis.*

— Alan J. Perlis

Subchapter 7 – 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 jika sebuah **expression** hasil komputasinya bernilai **true**.



Gambar 206 Looping Example

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"  
}
```

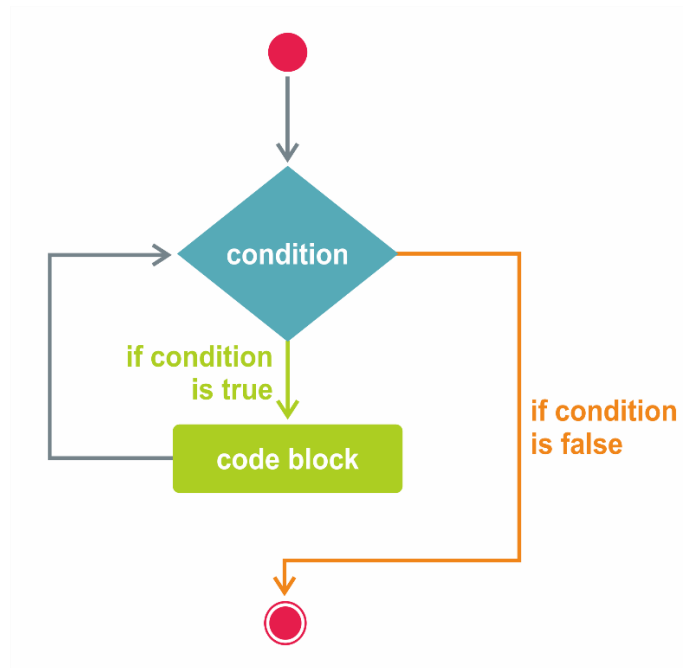
Contoh penggunaan *While Statement* dengan tiga *iteration* bisa kita lihat pada kode di bawah ini :

```
let n: number = 0;  
while (n < 3) {  
  console.log(n++);  
}
```

Keluaran dari kode di atas :

```
/* output :  
0  
1  
3 */
```

**Link sumber kode.*



Gambar 207 While Statement

Selain cara di atas juga terdapat *shorthand-while*, perulangan menggunakan *while* dalam satu *statement* :

```
let i: number = 3;
while (i) console.log((i--));
```

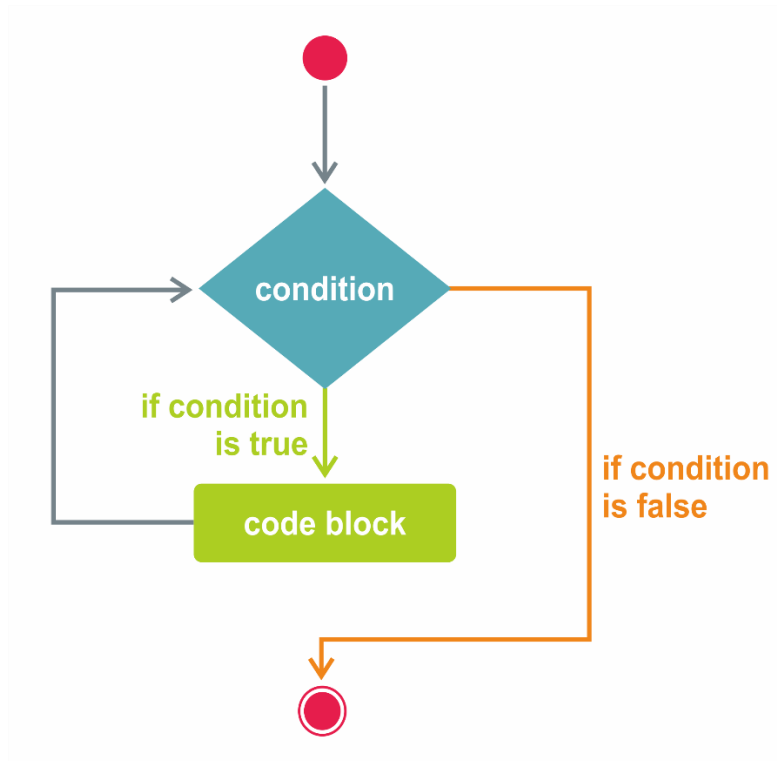
Keluaran dari kode di atas :

```
/* 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.



Gambar 208 Do...While Statement

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: number = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5)
```

Keluaran dari kode di atas :

```
/* Output
1
2
3
4
5 */
```

**Link sumber kode.*

3. For Statement

Di bawah ini adalah contoh *syntax for statement*.

```
for ([initialExpression]; [condition]; [incrementExpression];)  
    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. Setiap kali *statement* di dalam *block for statement* dieksekusi ***incrementExpression*** akan terus meningkat.

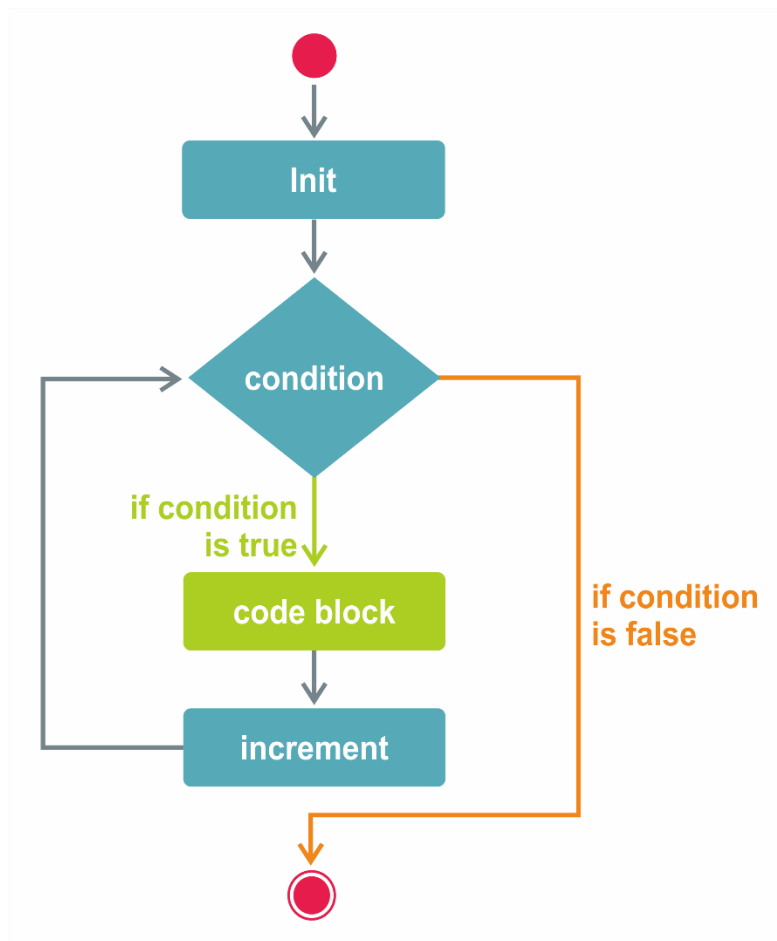
Contoh penggunaan *For Statements* bisa kita lihat pada kode di bawah ini :

```
let index: number = 0;  
for (index; index < 5; index++) {  
    console.log("Perulangan ke " + index);  
}
```

Keluaran dari kode di atas :

```
/* Output :  
Perulangan ke 0  
Perulangan ke 1  
Perulangan ke 2  
Perulangan ke 3  
Perulangan ke 4  
*/
```

**Link sumber kode.*



Gambar 209 For Statement

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.

```
let index: number = 10;
for (index; index > 5; index--) {
  console.log("Perulangan ke " + index);
}
```

Keluaran dari kode di atas :

```
/* Output :  
Perulangan ke 10  
Perulangan ke 9  
Perulangan ke 8  
Perulangan ke 7  
Perulangan ke 6  
*/
```

**Link sumber kode.*

4. For ... Of

Typescript juga mendukung perulangan menggunakan **for...of statement** untuk melakukan **iteration** pada **object** seperti **array**, **list** atau **tuple**. Untuk melakukan iterasi agar bisa mendapatkan element pada sebuah array perhatikan kode di bawah ini :

```
let arr: number[] = [1, 2, 3, 4];

for (let arrValue of arr) {
  console.log(arrValue);
}
```

Kode di atas jika dieksekusi akan menghasilkan keluaran sebagai berikut :

```
// output : 1, 2, 3, 4
```

**Link* sumber kode.

5. For..in

Typescript juga mendukung perulangan menggunakan **for...in statement** untuk melakukan **iteration** pada **object** seperti **array**, **list** atau **tuple**. Untuk melakukan iterasi agar bisa mendapatkan **element** pada sebuah **array** perhatikan kode di bawah ini :

```
let arr: number[] = [10, 20, 30, 40];

for (let index in arr) {
  console.log(index);
  console.log(arr[index]);
}
```

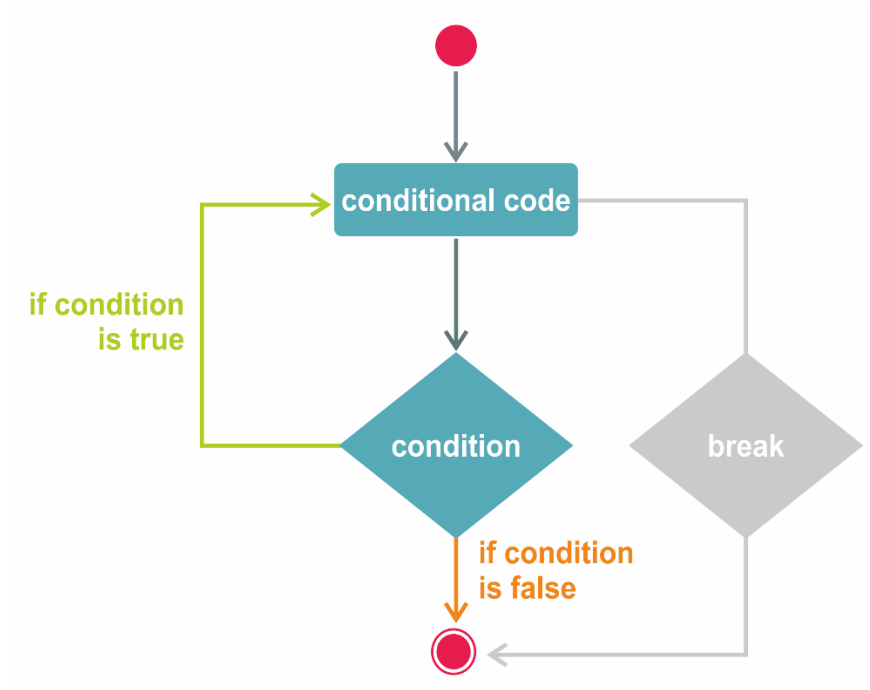
Kode di atas jika dieksekusi akan menghasilkan keluaran sebagai berikut :

```
/*
0
10
1
20
2
30
3
40
*/
```

*[Link](#) sumber kode.

6. 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.



Gambar 210 Break Statement

Untuk menggunakan **break statement** tulis kode di bawah ini :

```
let i: number = 0;
for (i; i < 10; i++) {
  if (i === 3) break;
  console.log("Perulangan ke " + i);
}
```

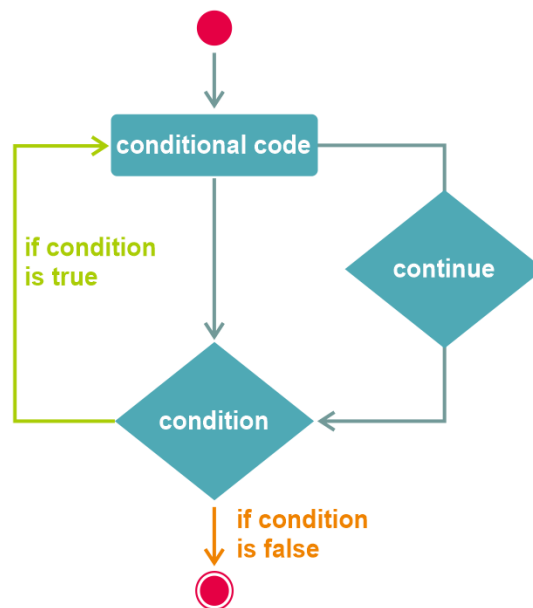
Keluaran dari kode di atas :

```
/* Output :  
Perulangan ke 0  
Perulangan ke 1  
Perulangan ke 2  
*/
```

**Link sumber kode.*

7. Continue Statement

Penggunaan **continue** dilakukan jika kita ingin menghentikan eksekusi suatu **statement** dalam suatu **iteration** dan melanjutkan kembali perulangan dengan **iteration** selanjutnya sampai selesai.



Gambar 211 Continue Statement

Untuk menggunakan **continue statement** tulis kode di bawah ini :

```
let index: number = 0;

for (index = 0; index < 6; index++) {
  if (index === 3) continue;
  console.log("Perulangan Ke " + index);
}
```

Keluaran dari kode di atas :


```
// Output
// Perulangan Ke 0
// Perulangan Ke 1
// Perulangan Ke 2
// Perulangan Ke 4
// Perulangan Ke 5
```

**Link sumber kode.*

8. *Labeled Statement*

Dengan *label statement* kita dapat membuat sebuah *identifier* sebagai sebuah *statement* yang menjadi acuan saat menggunakan *break* atau *continue statement*.

```
let str: string = "";
let idx: number = 0;

loop1:
for (idx; idx < 5; idx++) {
  if (idx === 1) {
    continue loop1;
  }
  str = str + idx;
  console.log(str);
}
```

Keluaran dari kode di atas :

```
//output :
// 0
// 02
// 023
// 0234
```

**Link sumber kode.*

Subchapter 8 – Function

*Any application that can be written in JavaScript
will eventually be written in JavaScript.*

—Atwood's Law, by Jeff Atwood

Subchapter 8 – 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 pemrograman 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 execution context*. Di dalam satu program *javascript* 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 memasukannya 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*.

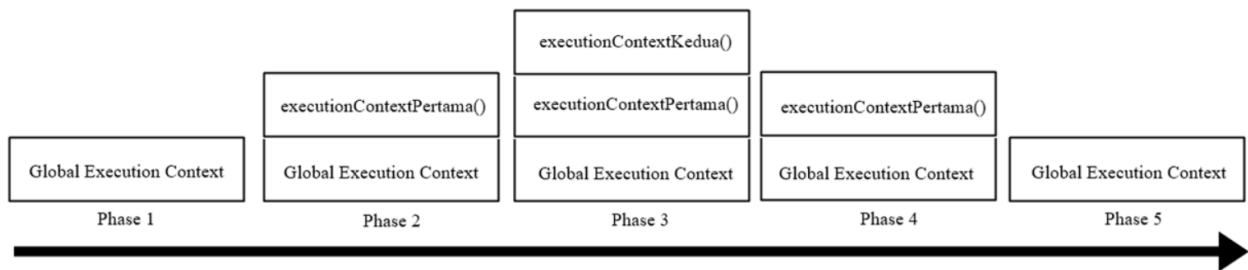
Ketika fungsi `executionContextPertama()` dipanggil maka *execution context* baru akan dibuat dan menyimpannya (*push*) ke dalam *exection stack* di posisi paling atas.

Ketika `executionContextKedua()` dipanggil maka di dalam

`executionContextPertama()` *javascript engine* akan membuat *execution context* baru dan menyimpannya 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 212 Execution Stack Simulation

3. Simple Function

Di bawah ini adalah bentuk paling sederhana untuk membuat fungsi :

```
function displayName() {  
  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()** :

```
displayName()  
/*  
Output  
Hello!  
Gun Gun Febrianza! */
```

Kita dapat menggunakan fungsi tersebut lagi dan lagi :

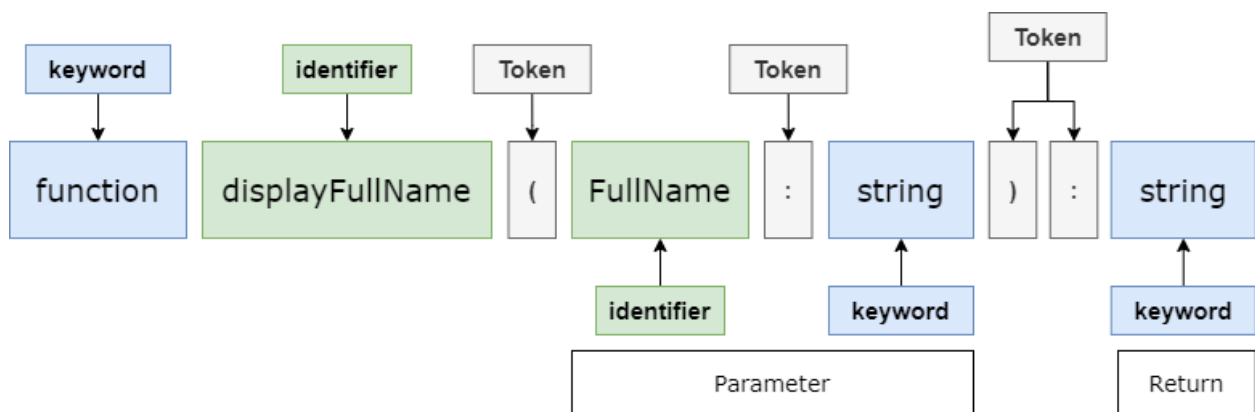
```
displayName()  
displayName()  
/*  
Output  
Hello!  
Gun Gun Febrianza!  
Hello!  
Gun Gun Febrianza! */
```

Typescript Version

Pada **typescript** ada perubahan dalam membuat **function** yaitu wajibnya penggunaan tipe data pada **parameter** dan **function return** :

```
function displayName(FullName: string): string {  
    return `Hello ${FullName}`;  
}
```

Jika kode di atas kita ubah ke dalam diagram maka terdapat beberapa struktur :



Gambar 213 Function Structure

Jika return berupa **string** maka **body function** wajib menyertakan **keyword return** di akhir baris **statement** sesuai dengan tipe data yang digunakan untuk melakukan **return**.

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!");
```

Kode di atas jika dieksekusi maka akan memproduksi keluaran :

```
/*  
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!");
```

Jika dieksekusi maka akan memproduksi :

```
/*  
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.

Parameter Type

Oleh karena itu penggunaan ***type annotation*** pada ***typescript*** dapat mencegah hal seperti di atas terjadi apabila ternyata ***input*** yang diberikan bukan berupa ***string***. Untuk versi ***typescript*** perhatikan kode di bawah ini :

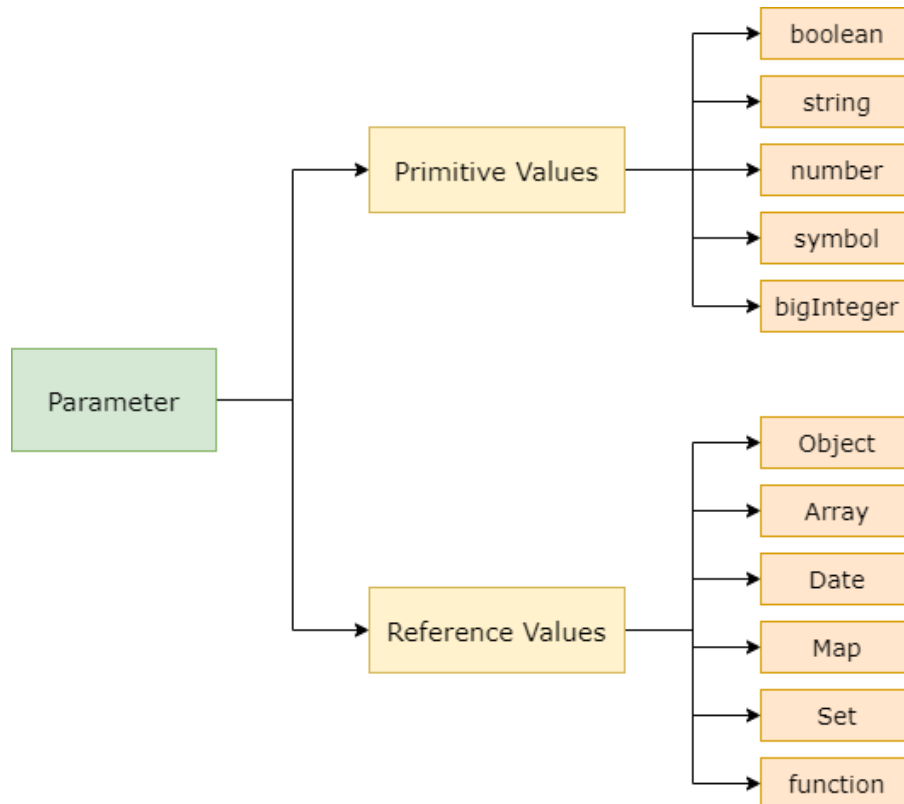
```
function displayMessage(from: string, text: string) {  
  // arguments: from, text  
  console.log(from + ": " + text);  
}  
  
displayMessage("Kaiz", "Hello! Maudy Ayunda!");  
displayMessage("Maudy Ayunda", "What's up? kaiz!");
```

Kode di atas jika dieksekusi maka akan memproduksi keluaran :

```
/*  
Output  
Kaiz: Hello! Maudy Ayunda!  
Maudy Ayunda: What's up? kaiz! */
```

Pada **function** di atas kita menggunakan **return void** menegaskan bahwa tidak terdapat **return** untuk fungsi tersebut, dengan begitu **body function** tidak perlu melakukan **return**.

Untuk **parameter type** sendiri kita dapat menggunakan **primitive data type** atau **references** :



Gambar 214 Parameter Type

Optional Parameter

Pada **typescript** kita dapat membuat sebuah **optional parameter**, dengan begitu kita dapat membuat **parameter** yang dapat diisi atau tidak diisi sama sekali. Syaratnya adalah **optional parameter** harus ditaruh diakhir parameter posisinya.

Di bawah ini adalah contoh **Optional Parameter**, dimana **parameter** ketiga tidak digunakan sama sekali :

```
function getAverage(a: number, b: number, c?: number): string {  
  let total = a + b;  
  let counter = 1;  
  counter++;  
  if (typeof c !== "undefined") {  
    total += c;  
    counter++;  
  }  
  const average = total / counter;  
  return "The average is " + average;  
}
```

Untuk menggunakan fungsi di atas eksekusi kode di bawah ini :

```
const result = getAverage(2, 6);  
console.log(result);
```

Hasilnya :

```
// 'The average is 5'
```

Default Parameter

Selain menggunakan ***optional parameter*** kita juga dapat menggunakan ***default parameter*** yang dapat memberikan nilai ***default*** jika ***argument*** tidak disediakan, saat ***function*** akan dieksekusi.

```
function concatName(firstName: string = "Gun Gun"): string {  
  return `${firstName}`;  
}
```

```
console.log(concatName("Maudy"));  
console.log(concatName());
```

Jika kode di atas dieksekusi maka akan memproduksi hasil :

```
//Maudy  
//Gun Gun
```

Rest Parameter

Ada saatnya kita berhadapan dengan situasi dimana kita membutuhkan suatu ***function*** yang dapat menerima ***argument*** tanpa batas, solusinya adalah ***rest parameter***. Di bawah ini kita membuat sebuah ***function*** dengan ***rest parameter number*** :

```
function getAverage(...param: number[]): string {  
  let total = 0, count = 0, index = 0;  
  for (index; index < param.length; index++) {  
    total += param[index];  
    count++;  
  }  
  const average = total / count;  
  return "The average is " + average;  
}
```

Untuk menggunakan ***function*** di atas :

```
const res = getAverage(4, 4, 4, 2, 2, 2);  
console.log(res);
```

Jika kode di atas di eksekusi maka akan memproduksi :

```
// 'The average is 3'
```


5. Function Return

Di bawah ini adalah sebuah *function* dalam **javascript** 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
```

Jika kode di atas dieksekusi maka akan memproduksi :

```
/*  
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);`

Untuk versi **typescript** sendiri kita hanya perlu menambahkan **static typing** pada **parameter** dan *return* yang jelas seperti pada kode di bawah ini :

```
function summary(a: number, b: number): number {  
    return a + b;  
}  
  
let resultSum = summary(1, 2);  
console.log(resultSum); // 3
```

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()`.

Untuk versi typescript sendiri perhatikan kode di bawah ini :

```
function displayMessage(from: string, text = test()) {  
  console.log(from + ": " + text);  
}  
  
function test(): string {  
  return "Hello";  
}  
  
displayMessage("Maudy Ayunda");
```

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(): void {  
    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 **typescript** di mana *statement* di dalam *function* mencoba mengakses variabel diluar sekup *function*. Operasi ini tetap dapat dilakukan di dalam **typescript** :

```
let userName = "Gun Gun Febrianza";

function tampilkanPesan(): void {
  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.*

Untuk ***callback*** versi ***typescript*** sendiri perhatikan kode di bawah ini :

```
function greeting(name: string): void {  
    console.log("Hello " + name);  
}  
  
function processUserInput(callback: (name: string) => void) {  
    var name = "Gun Gun Febrianza";  
    callback(name);  
}  
  
processUserInput(greeting);
```

```
//Hello Gun Gun Febrianza
```

10. Arrow Function

Arrow function pertama kali diperkenalkan dalam ES6, dengan *arrow function* kita dapat membuat sebuah *function* dengan *syntax* yang lebih singkat. Selain itu kita juga dapat membuat ***anonymous function*** menggunakan ***arrow function***.

Di bawah ini adalah contoh kode menggunakan *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.

Untuk versi ***typescript*** sendiri :

```
let sum = (a: number, b: number): number => a + b;
console.log(sum(1, 2)); // 3
```


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 function  
  let result = a + b;  
  return result; // gunakan return untuk memproduksi hasil  
};  
  
console.log(sum(1, 2)); // 3
```

**Link* sumber kode.

Untuk versi ***typescript*** sendiri perhatikan kode di bawah ini :

```
let summary = (a: number, b: number): number => {  
  // curly brace untuk membuat multiline function  
  let result = a + b;  
  return result; // gunakan return untuk memproduksi hasil  
};  
  
console.log(summary(1, 2)); // 3
```

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.

Untuk versi **typescript** sendiri :

```
let addition = function (a: number, b: number): number {  
    return a * b;  
};  
console.log(addition(4, 3)); //12
```

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.

Untuk versi **typescript** sendiri :

```
function additionFunc(a: number, b: number): number {  
    return a * b;  
}  
var x: number = additionFunc(4, 3) * 2; //expression  
console.log(x); //24
```

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.

Untuk versi **typescript** sendiri :

```
function addition(): number {  
  var counter: number = 0;  
  
  function plus(): void {  
    counter += 1;  
  }  
  plus();  
  return counter;  
}  
console.log(addition()); //1
```

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 di dalam **browser firefox** jika kita mengesekusi kode di bawah ini :

```
>> // define a function
var myFunction = function () {
  console.log(this);
};

// call it
myFunction();
← undefined
  ► Window about:newtab
```

Gambar 215 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 216 *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.*

Untuk versi **typescript** sendiri :

```
(function (): void {  
    console.log("hello");  
})();
```

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.*

Pada **typescript** permasalahan **javascript** ini dapat dicegah karena sudah mendukung **static typing**.

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.*

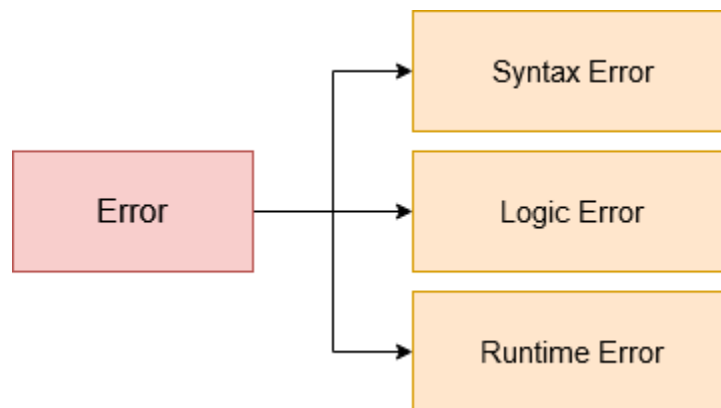
Subchapter 9 – Error Handling

The best error message is the one that never shows up.

—Thomas Fuchs

Subchapter 9 – Objectives

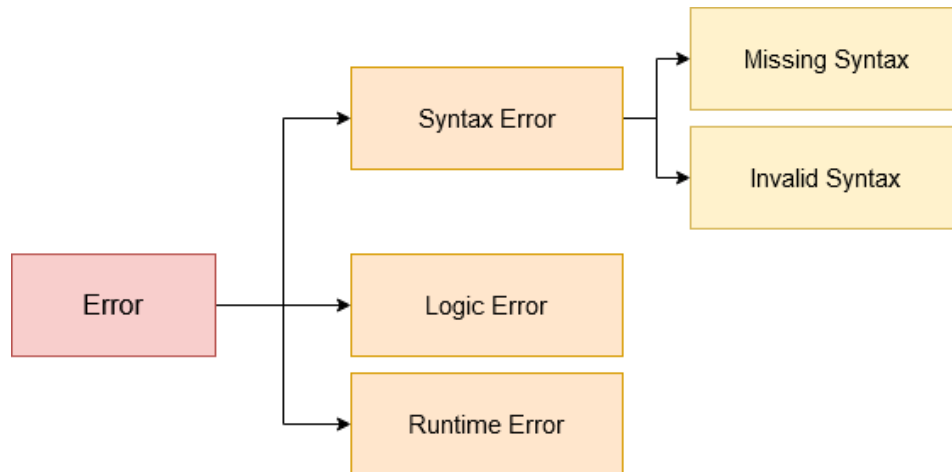
- Memahami Apa itu **Syntax Error**?
 - Memahami Apa itu **Logical Error**?
 - Memahami Apa itu **Runtime Error**?
 - Memahami Apa itu **Reference Error**?
 - Memahami Apa itu **Range Error**?
 - Memahami Apa itu **Type Error**?
 - Memahami Apa itu **Try & Catch Statement**?
 - Memahami Apa itu **Finally Statement**?
 - Memahami Apa itu **Error Object Properties**?
 - Memahami Apa itu **Stack Trace**?
 - Memahami Apa itu **Custom Error**?
-



Gambar 217 Error Type

Sebuah program memiliki tiga kemungkinan errors yaitu *syntax error*, *logical error* dan *runtime error*. [59]

1. Syntax Error



Gambar 218 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 219 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*.

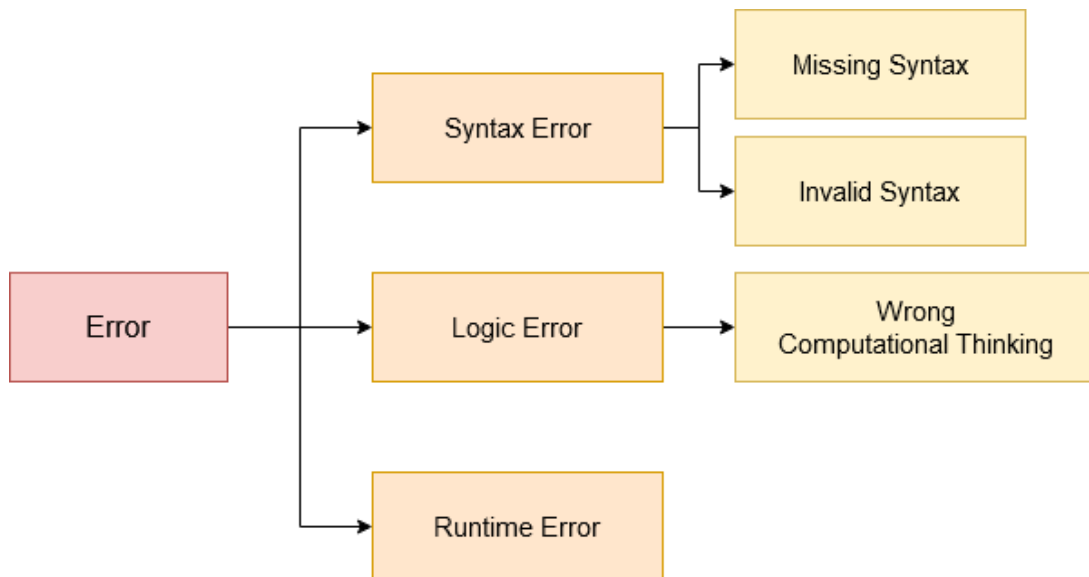
```
>> var x = 10var
```

❗ **SyntaxError: identifier starts immediately after numeric literal** [\[Learn More\]](#)

Gambar 220 Invalid Syntax Example

Pada *syntax error* di atas *interpreter javascript* memperlakukan **10var** sebagai sebuah *identifier*, menolak *syntax rule* tersebut karena *identifier* tidak boleh diawali dengan angka. Pesan *error* ini telah diatur oleh pembuat *javascript engine*.

2. Logical Error



Gambar 221 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);
```

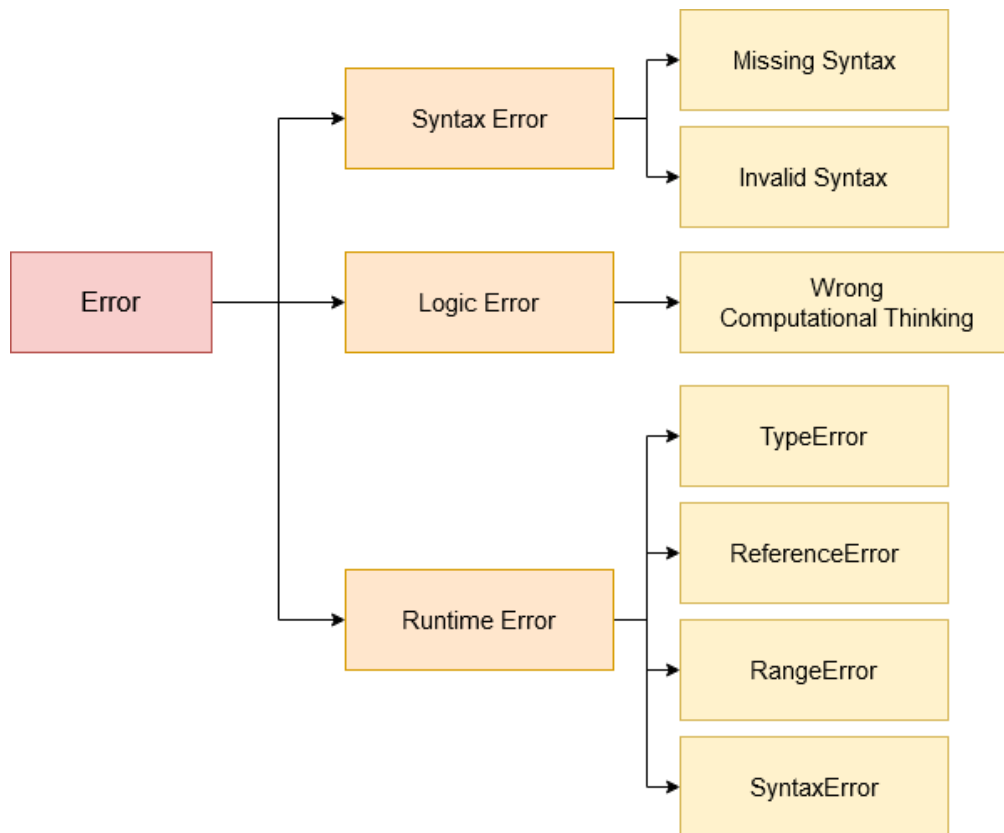
Kita asumsikan anda telah menulis kode hingga ratusan atau ribuan baris, kemudian anda berpikir bahwa seharusnya nilai dari variabel **result** adalah **200**. Karena anda yakin nilai **x** adalah **20**, padahal nilai **x** bukanlah **20** melainkan **10**, namun anda tidak menyadarinya sama sekali.

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 222 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.

Reference Error

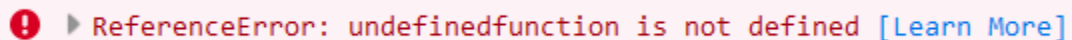
Javascript memiliki salah satu *type error* yang disebut dengan *reference error*.

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 :

A screenshot of a JavaScript error message displayed in a browser's developer console. The message is "ReferenceError: undefinedfunction is not defined" followed by a blue link "[Learn More]". The message is preceded by a red exclamation mark icon and a right-pointing triangle.

Gambar 223 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()` :

[illegible]


Gambar 224 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* :

```
>>> var num = 1;
      num.toUpperCase();
```


 ▶ **TypeError: num.toUpperCase is not a function** [\[Learn More\]](#)

```
>>> |
```

Gambar 225 Type Error

Syntax Error

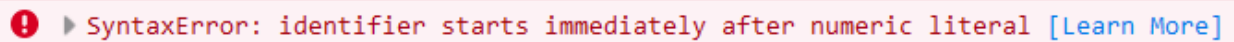
Syntax Error juga dapat terjadi saat *runtime*, sebagai contoh jika kita menggunakan *method* `eval()` yang dapat membaca *string javascript* untuk di eksekusi :

```
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 :



! ▶ SyntaxError: identifier starts immediately after numeric literal [\[Learn More\]](#)

Gambar 226 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.

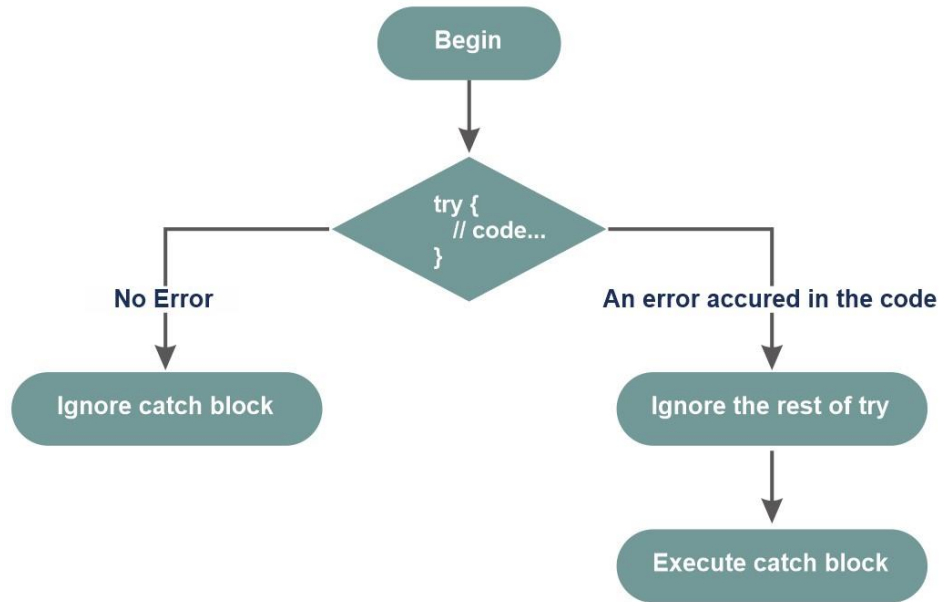
Meskipun begitu *error* tetaplah terjadi namun kita berhasil mengatasinya, untuk mengetahui tipe *error* dan pesan *error* kita dapat memanggil *error object*. Lalu melihat *properties name & message* yang dimiliki oleh *error object*.

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.

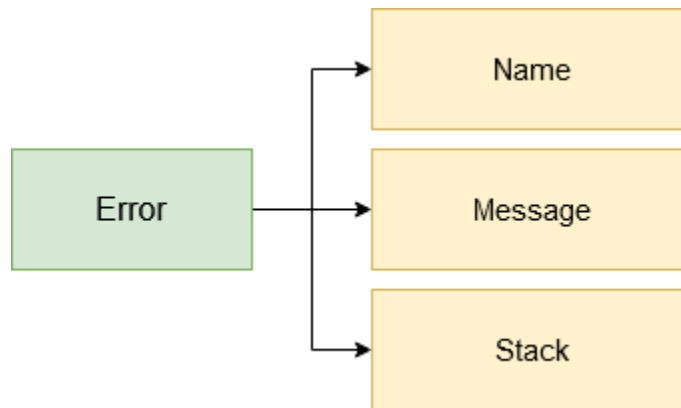


Gambar 227 Try & Catch Statement

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 228 Error Object Properties

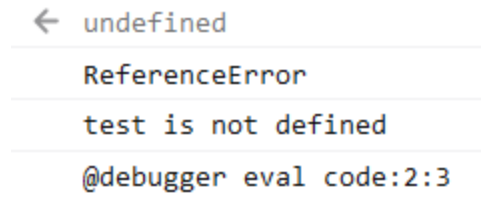
Property `name` akan memberikan informasi tipe *error* yang terjadi dan property `message` akan memberikan informasi pesan *error* yang terjadi.

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 :



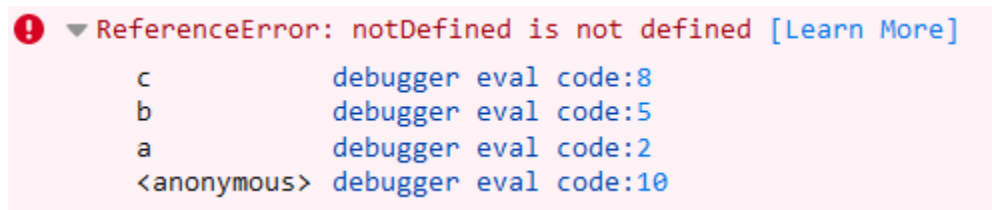
Gambar 229 property stack result

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 230 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*.

Finally

Apapun hasil dari `try` & `catch` *statement*, `finally` *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 `finally` 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 melakukannya?

```
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 Object 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 number !");
```

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 231 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 be 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 232 Custom Validation Error 2

Subchapter 10 – Object

Good code is its own best documentation.

—Steve McConnell

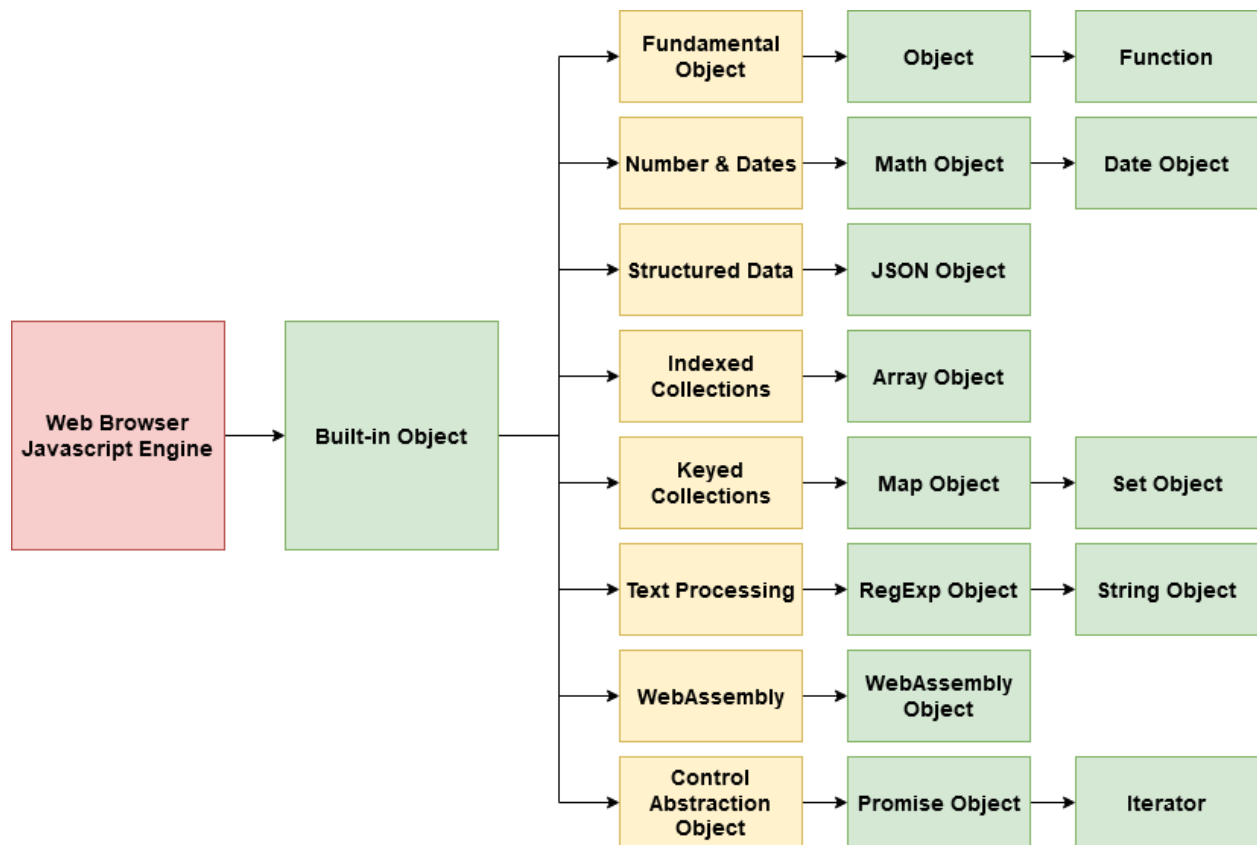
Subchapter 10 – Objectives

- 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 233 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*.

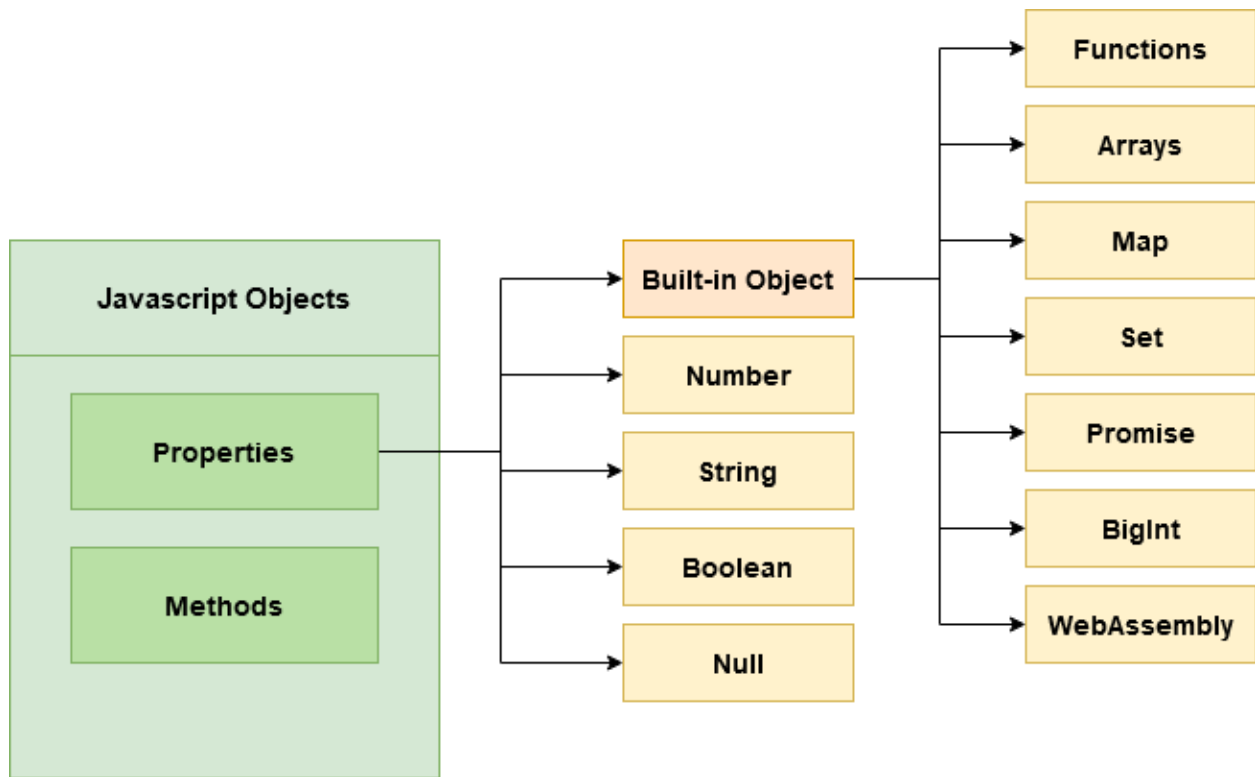
Di antaranya adalah :

- | | |
|---------------------|--------------------------|
| 1. <i>Object</i> | 7. <i>InternalError</i> |
| 2. <i>Function</i> | 8. <i>RangeError</i> |
| 3. <i>Boolean</i> | 9. <i>ReferenceError</i> |
| 4. <i>Symbol</i> | 10. <i>SyntaxError</i> |
| 5. <i>Error</i> | 11. <i>TypeError</i> |
| 6. <i>EvalError</i> | 12. <i>URIError</i> |

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 234 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 235 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".**

Object Property

Pada gambar di bawah ini kita membuat sebuah *object* bernama **mod** dengan *properties* **firstname**, **lastname**, **age** dan **haircolor**.

```
>> var mod = { firstname: 'Maudy', lastname: 'Ayunda Faza', age: 23, haircolor: 'black' }
← undefined
>> mod
← ▶ Object { firstname: "Maudy", lastname: "Ayunda Faza", age: 23, haircolor: "black" }
```

Gambar 236 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 237 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 238 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 melakukannya 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 239 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 240 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 241 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* :

Function.prototype.apply()

Calls a function and sets its *this* to the provided value, arguments can be passed as an *Array* object.

Function.prototype.bind()

Creates a new function which, when called, has its *this* set to the provided value, with a given sequence of arguments preceding any provided when the new function was called.

Function.prototype.call()

Calls (executes) a function and sets its *this* to the provided value, arguments can be passed as they are.

Gambar 242 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
```

Typescript Type Template

Pada ***typescript*** untuk membuat ***object*** kita dapat memanfaatkan ***type template*** :

```
const mod: {  
  firstname: string;  
  lastname: string;  
  age: number;  
  haircolor: string;  
} = {  
  firstname: "Maudy",  
  lastname: "Ayunda Faza",  
  age: 23,  
  haircolor: "black",  
};
```


Pada kode di atas kita akan memberikan terlebih dahulu tipe data yang akan digunakan untuk setiap **properties** dalam **object** `mod`.

Complex Object Type

```
type TypeAlias = number[];

let objComplex: {
  a: string[];
  b: (param: string[]) => string[];
  c: { d: boolean; e: TypeAlias };
} = {
  a: ["Hi", "Maudy"],
  b: function (param: string[]): string[] {
    return this.a;
  },
  c: { d: true, e: [22, 33] },
};

console.log(objComplex);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*
{
  a: [ "Hi", "Maudy" ],
  b: [Function: b],
  c: {
    d: true,
    e: [ 22, 33 ]
  }
}
```

```
}  
*/
```

Kode di atas dapat disederhanakan menjadi :

```
type TypeAlias = number[];  
type ComplexObj = {  
  a: string[];  
  b: (param: string[]) => string[];  
  c: { d: boolean; e: TypeAlias };  
};  
  
let objComplex: ComplexObj = {  
  a: ["Hi", "Maudy"],  
  b: function (param: string[]): string[] {  
    return this.a;  
  },  
  c: { d: true, e: [22, 33] },  
};  
  
console.log(objComplex);
```

Kita dapat membuat **custom data type** menggunakan **type alias** seperti yang telah anda pelajari sebelumnya dalam **subchapter custom data type**.

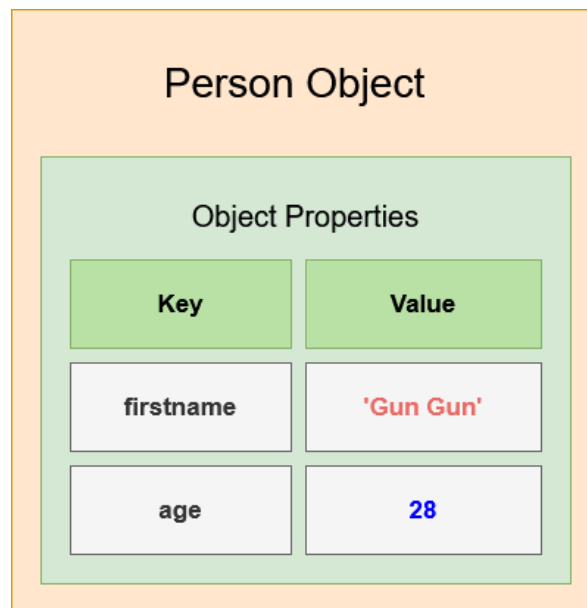
3. Custom Object Property

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 243 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 244 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?

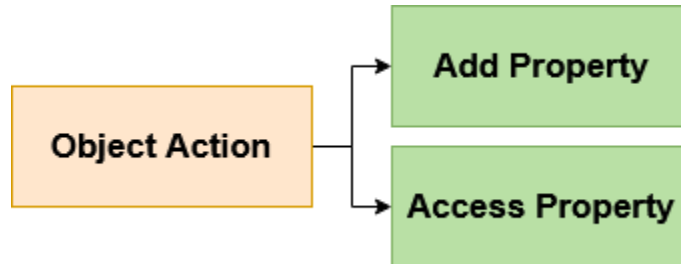
```
person.nationality = 'Indonesia';  
console.log(person);
```

*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 245 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.');
```

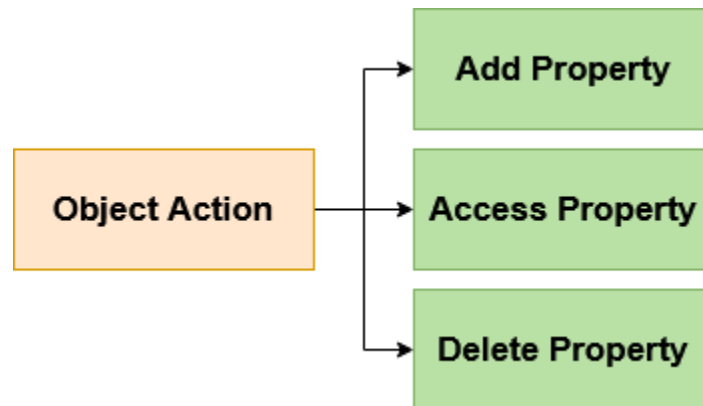
// Gun Gun is 28 years old.

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 246 Delete Property

Pada kode di bawah ini kita membuat sebuah *object* bernama *person*.

```
var person = {  
  firstName: 'Gun Gun',  
  lastName: 'Febrianza',  
  age: 28,  
};
```

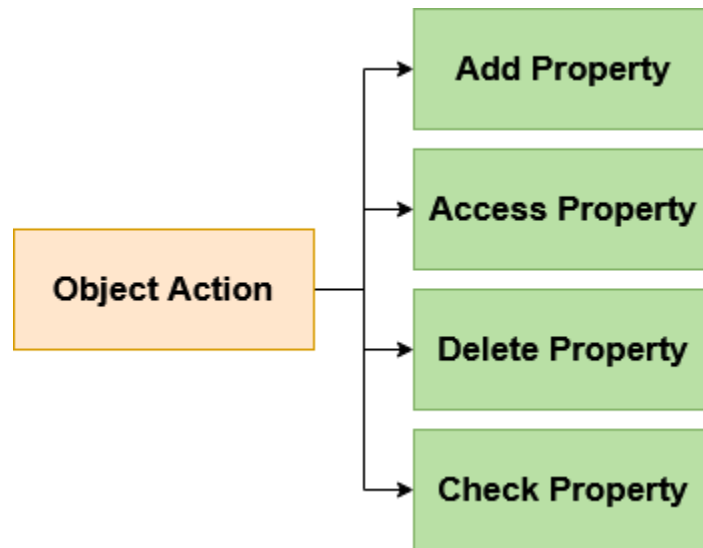
Jika kita ingin menghapus *property* *age*, maka kita perlu mengeksekusi perintah di bawah ini :

```
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 247 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 248 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

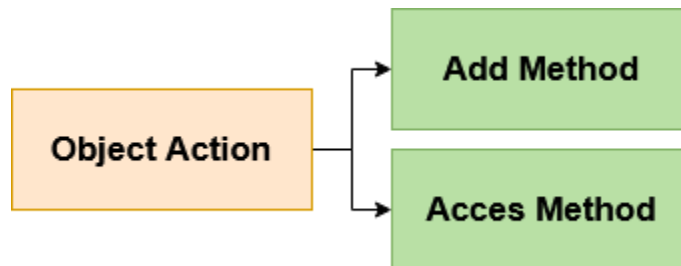
Jika kita ingin memanggil *method* `fullName()` yang dimiliki oleh *object* `person` maka perhatikan kode di bawah ini :

```
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 249 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 Enumeration

Ada kalanya kita ingin melakukan operasi **looping** pada suatu **object literal**, baik itu untuk mendapatkan seluruh **property** ataupun **value** yang dimilikinya.

Secara **default** ketika kita menambahkan suatu **property** pada **object** terdapat pengaturan internal yang akan memberikan nilai **true** pada **attribute** `[[Enumerable]]`.

Kita dapat menggunakan **looping** menggunakan **for...in statement** untuk melakukan **enumeration** pada seluruh **enurable property** dalam suatu **object**. Untuk melakukannya 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, perbedaannya 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* lainnya.

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 250 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)
```

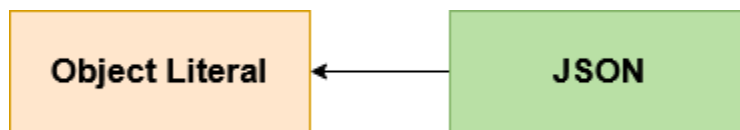


```
// {"name":"Gun Gun Febrianza","age":28,"city":"Bandung"}
```

*[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 251 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":"Bandung"}'
```

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 11 – Classes

Learning to code is learning to create and innovate.

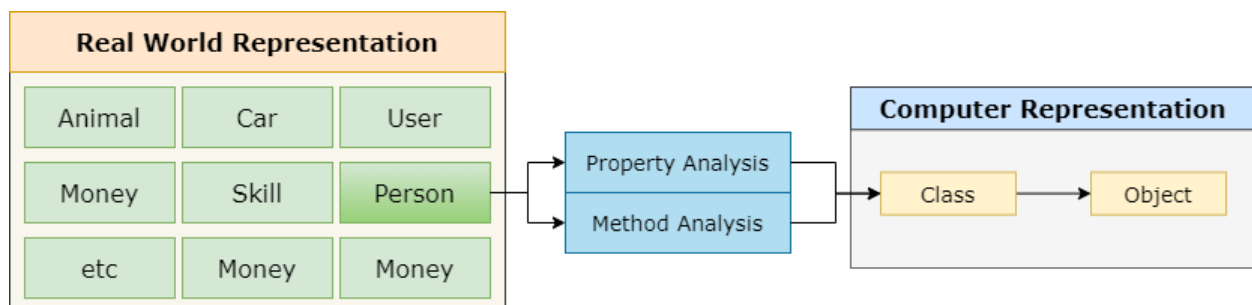
—Enda Kenny

Subchapter 11 – Objectives

- Memahami Apa itu **Class-based Language?**
 - Memahami Apa itu **Class Inheritance?**
 - Memahami Apa itu **Class Access Modifier?**
 - Memahami Apa itu **Class Constructor?**
 - Memahami Apa itu **Static Keyword?**
 - Memahami Apa itu **Super Method?**
 - Memahami Apa itu **Method Override?**
 - Memahami Apa itu **Accessor Getter & Setter?**
 - Memahami Apa itu **Abstract Class?**
-

Ada saatnya dalam pengembangan aplikasi kita ingin merepresentasikan sesuatu yang ada di dunia nyata (**Real World Representation**) ke dalam komputer (**Computer Representation**).

Ada banyak sekali representasi di dunia nyata yang dapat diubah ke dalam bahasa pemrograman berorientasi **object** (**OOP**). Pada gambar di bawah ini kita mengetahui bahwa **person** adalah representasi suatu object di dunia nyata.



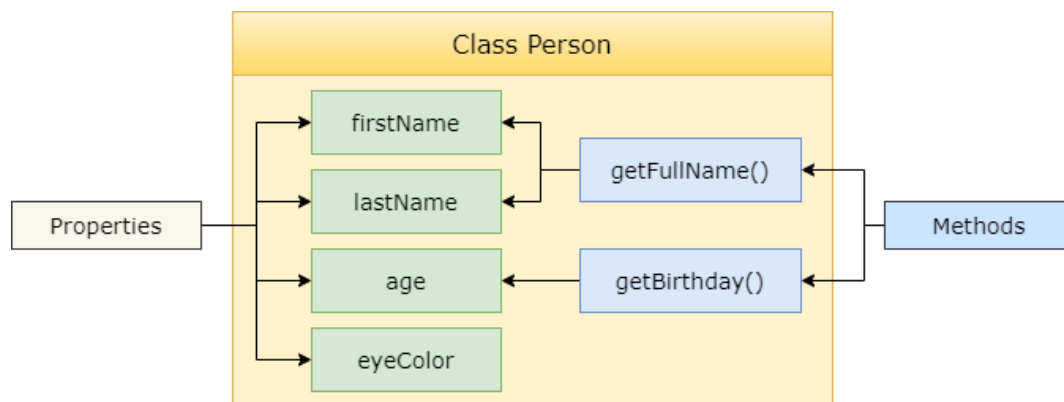
Gambar 252 Class Representation

Untuk bisa merepresentasikannya kedalam komputer kita harus memahami **property** dan **method analysis** terlebih dahulu, agar dapat membuat sebuah **class** di dalam pemrograman.

Jika sebelumnya anda telah mempelajari konsep **function** dan **variable** yang dapat menampung data **primitive & reference**, maka **property** pada dasarnya bekerja seperti variabel untuk menampung data dan **method** bekerja seperti **function**.

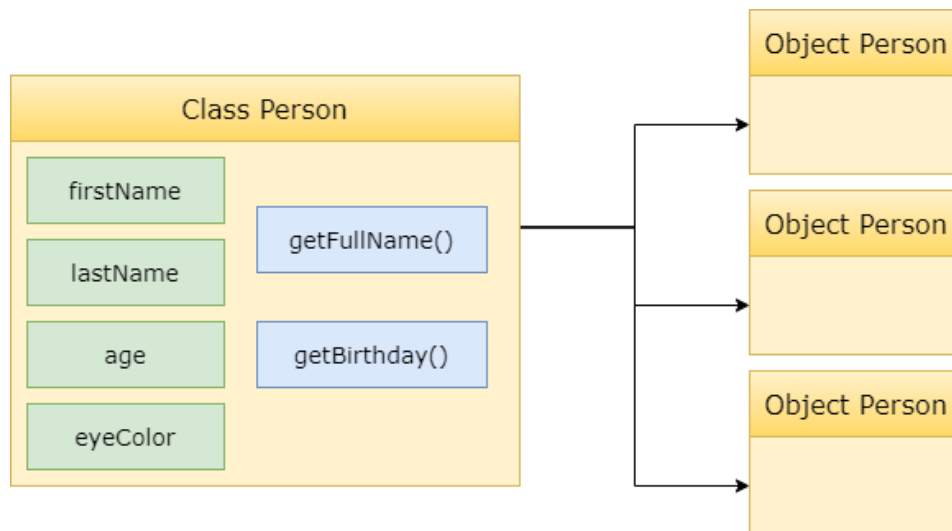
Biasanya **property** dan **method** yang dibutuhkan muncul atas dorongan untuk memecahkan permasalahan dalam setiap **programming domain**.

Misal kita ingin merepresentasikan sebuah data **person** dalam aplikasi, **person** tersebut cukup hanya memiliki 4 **properties** saja dan 2 **methods** saja seperti gambar di bawah ini :



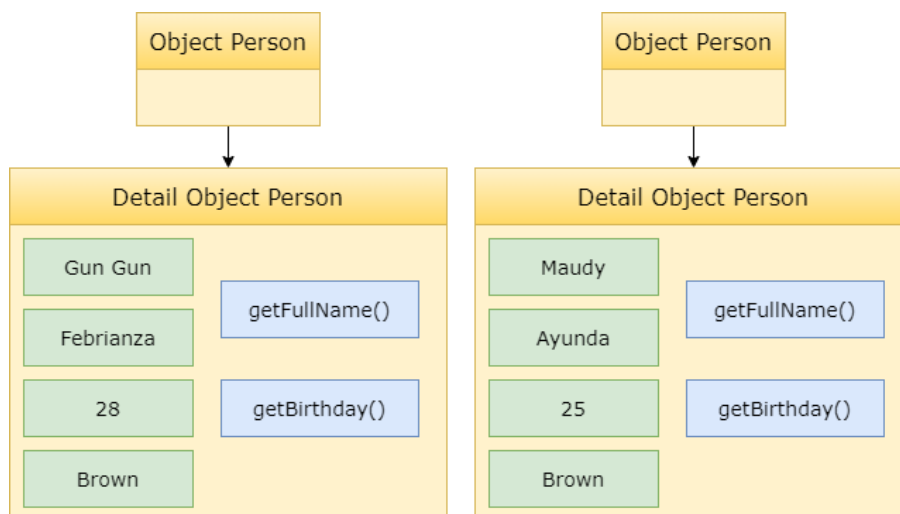
Gambar 253 Class Person

Class adalah sebuah *template* atau *blueprint* dari *object* yang akan dibuat. Sebuah kelas menjelaskan seluruh *attribute* sebuah *objects*, termasuk *methods* yang akan menentukan sifat dari *object* yang akan diciptakan.



Gambar 254 Class to Objects

Dari sebuah **class** kita dapat menciptakan banyak **object** yang memiliki **property** dan **method** sama seperti **blueprint class** yang di tiru. Di bawah ini adalah contoh **object person** secara detail yang memiliki karakteristik seperti **class person** :



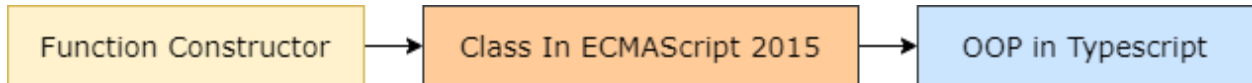
Gambar 255 Object In Detail

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

Buku ini didesain untuk pengembangan menggunakan **Typescript** dan **Deno Runtime** jadi kita akan mempelajari konsep **Function Costructor** pada **javascript** terlebih dahulu, kemudian konsep **class** dalam **ECMAScript 2015**, setelah itu kita mempelajari konsep **OOP** dalam **typescript**.



Gambar 256 Learn Roadmap

Manfaatnya adalah anda mengetahui **language abstraction** konsep OOP dari **high level** yang ditulis menggunakan **typescript** hingga ke representasi **low level** dalam bahasa **javascript**.

Function Constructor

Kode **javascript** di bawah ini digunakan untuk membuat sebuah *function constructor* **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.getFullName = function() {
  return this.firstname + " " + this.lastname;
};
```

Class ECMAScript2015

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*.

Untuk mentransformasikan *function constructor* sebelumnya ke dalam *class* menggunakan **javascript** perhatikan kode di bawah ini :

```
class person {  
  constructor(firstname, lastname, age, eyecolor) {  
    this.firstname = firstname;  
    this.lastname = lastname;  
    this.age = age;  
    this.eyecolor = eyecolor;  
  }  
  
  getFullName() {  
    return this.firstname + " " + this.lastname;  
  }  
}
```

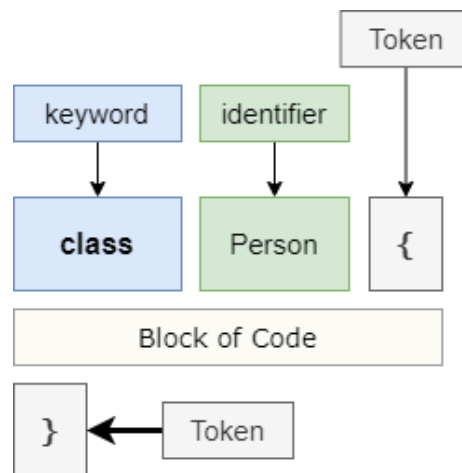
Untuk membuat **object** dari **class** di atas perhatikan kode di bawah ini :

```
const hooman = new Person();  
hooman.firstName = "Maudy";  
hooman.lastName = "Ayunda";  
hooman.age = 25;  
hooman.eyecolor = "Brown";  
console.log(hooman.getFullName());
```


Class Typescript

Pada **typescript** tipe data untuk **properties** tidak dideklarasikan secara eksplisit, **typescript compiler** akan melakukan **type inferring** dengan membaca tipe data berdasarkan **literal** yang diberikan.

Struktur **syntax** untuk membuat **class** :



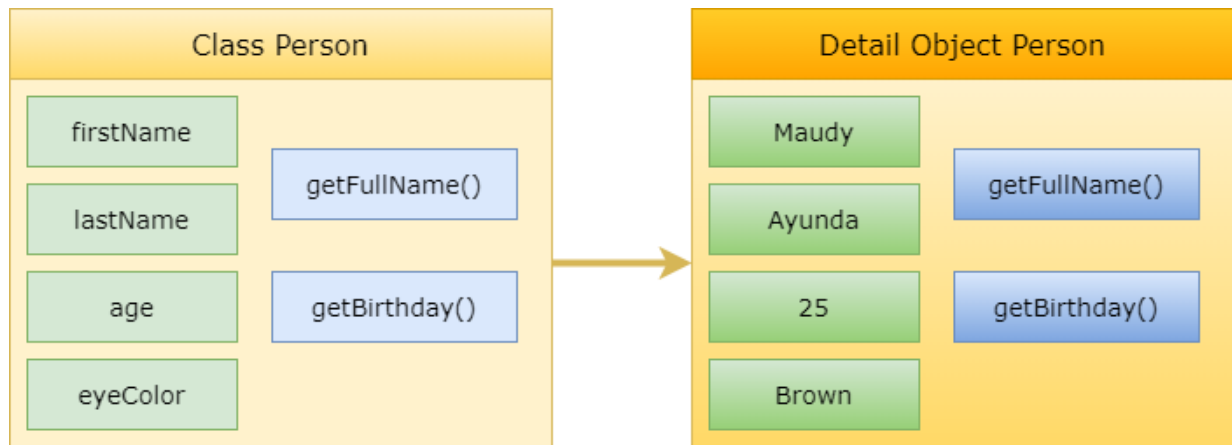
Gambar 257 Create Class Diagram

Di bawah ini adalah contoh **class** dalam **typescript** :

```
class Person {
  firstName = "";
  lastName = "";
  age = 0;
  eyecolor = "";
}

getName(): string {
  return this.firstName + " " + this.lastName;
}
```

Jika kode di atas direpresentasikan ke dalam visual perhatikan diagram di bawah ini :



Gambar 258 Object Example

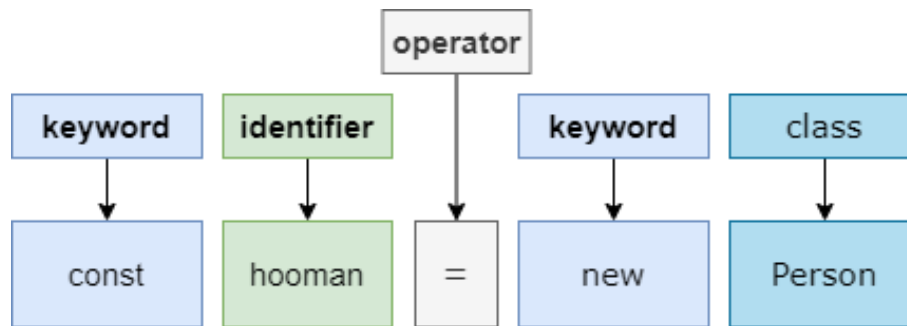
Untuk membuat **object** dari **class Person** berdasarkan diagram di atas perhatikan kode di bawah ini :

```
const hooman = new Person();
hooman.firstName = "Maudy";
hooman.lastName = "Ayunda";
hooman.age = 25;
hooman.eyecolor = "Brown";
console.log(hooman.getFullName());
```

Penjelasan dari kode di atas adalah sebagai berikut, pertama kita akan membuat **object** terlebih dahulu menggunakan **class Person** :

```
const hooman = new Person();
```

Jika **statement** di atas di bedah ke dalam diagram maka strukturnya adalah :



Gambar 259 Create Object Person

Setelah **object hooman** dibuat kita perlu mengisi seluruh **properties** sesuai dengan **blueprint class Person** :

```
hooman.firstName = "Maudy";  
hooman.lastName = "Ayunda";  
hooman.age = 25;  
hooman.eyecolor = "Brown";
```

Setelah itu kita dapat memanggil **method** yang dimiliki oleh **object hooman** :

```
console.log(hooman.getFullName());
```

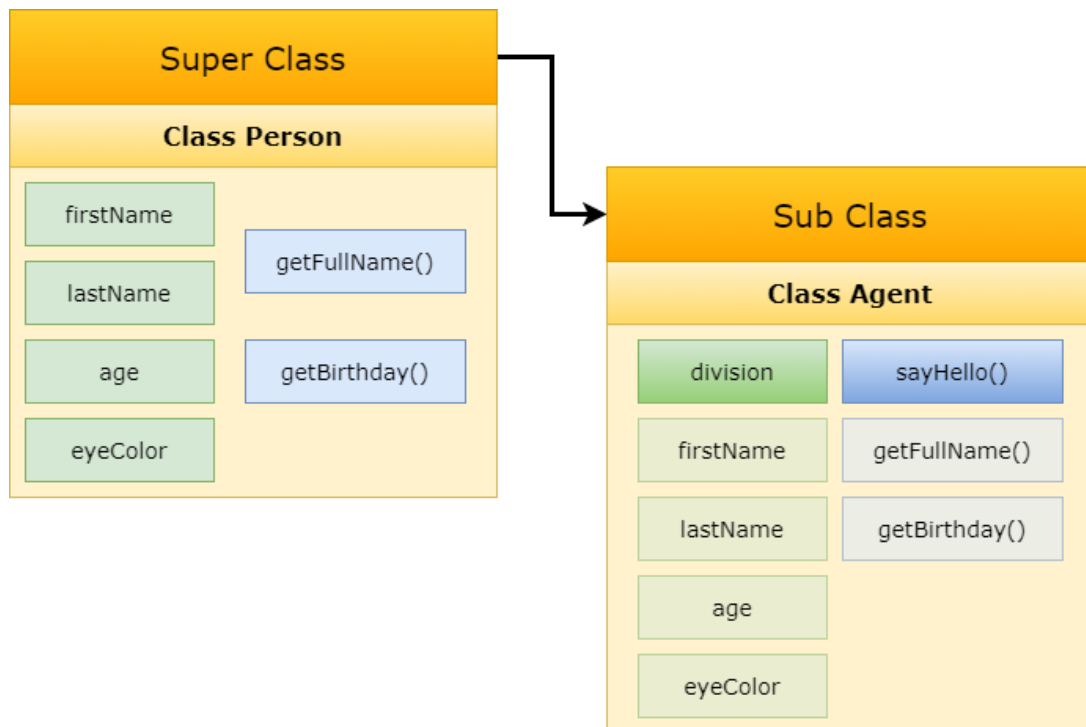
Ada Catatan kecil :

Notes

Gunakan **Compiler Options** `strictPropertyInitialization` agar mencegah kita untuk membuat **class** tanpa **property**.

2. Class Inheritance

Secara teori *Class Inheritance* adalah kemampuan suatu *class* untuk memberikan karakteristik yang dimilikinya pada *class* turunannya. Pada diagram di bawah ini terdapat contoh **inheritance** :

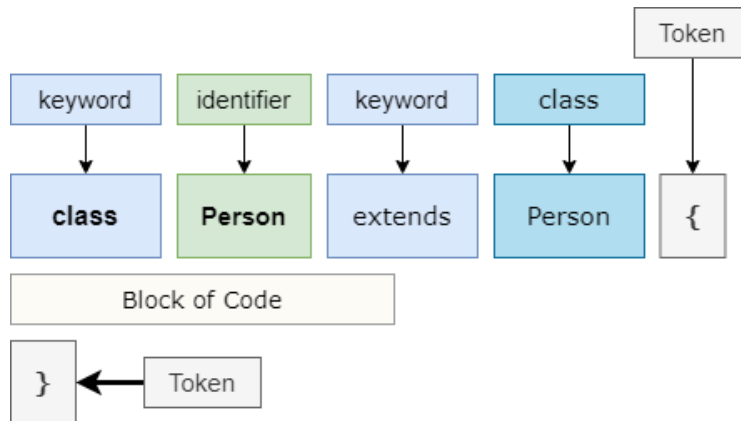


Gambar 260 Class Inheritance

Class Person menjadi **Super Class** atas **Class Agent**, dan **Class Agent** menjadi **Sub Class Agent** atau **Class Person**. Jika dikonversi ke dalam bahasa pemrograman perhatikan kode di bawah ini :

```
class Agent extends Person {
    division = "";
}
```

Strukturnya sebagai berikut :

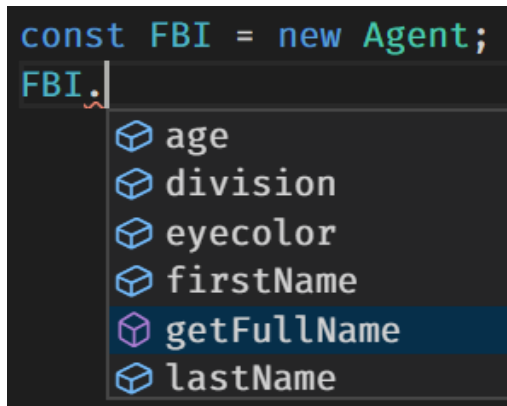


Gambar 261 Extends Class

Untuk membuat sebuah **object** dari **class Agent** tulis kode di bawah ini :

```
const FBI = new Agent;
```

Untuk membuktikan bahwa **class** turunan mendapatkan karakteristik dari **super class** perhatikan gambar di bawah ini, **object** dari **class Agent** memiliki **property** yang dimiliki oleh **class Person** :

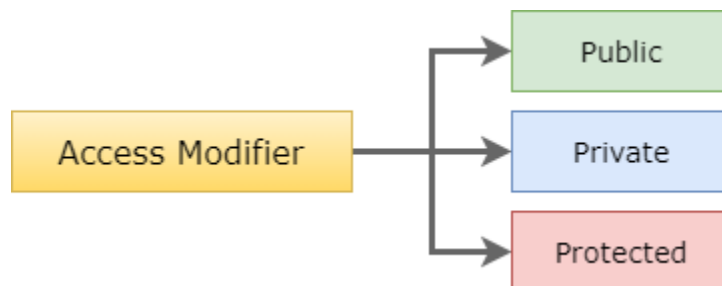


Gambar 262 Super Class Inheritance

Jika kita membuat **Sub Class** yang baru adakah cara untuk membatasi agar **Sub Class** tidak dapat memiliki akses pada salah satu **property** atau **method** dalam **Super Class**? Jawabannya : Ada kita dapat memanfaatkan konsep **Access Modifier**.

3. Class Access Modifier

Typescript memiliki fitur **Access Modifier** yaitu **public**, **private**, **protected** keywords sebagai jalur untuk akses kontrol terhadap suatu **class**.



Gambar 263 Access Modifier

Public

Class member (property & method) yang ditandai sebagai **public** dapat diakses secara internal dan eksternal oleh **class** turunannya.

Private

Class member (property & method) yang diberi tanda **private** hanya dapat di akses di dalam **class** itu sendiri. Tidak dapat di akses oleh **class** turunannya.

Protected

Class member (property & method) yang diberi tanda **protected** hanya dapat diakses dari dalam **class** itu sendiri atau **class** turunannya (**descendants**).

Kita akan mencoba memahami cara kerja ketiga **access modifier** tersebut melalui kode di bawah ini :

```
class Person {
    public firstName = "";
    public lastName = "";
    private age = 0;
    public eyecolor = "";

    protected getFullName(): string {
        return `Fullname : ${this.firstName} ${this.lastName}`;
    }
}
```

Kita akan memberikan **access modifier** :

1. **Public** untuk *properties* `firstName`, `lastName` dan `eyecolor`.
2. **Private** untuk property `age`
3. **Protected** untuk method `getFullName()`

Selanjutnya kita membuat sebuah **class** turunan (**descendant**) bernama **Agent** :

```
class Agent extends Person {
    division = "";
}
```

Public Behaviour

Seluruh **properties public** yang ada pada **super class** `Person` dapat diakses oleh **class** turunannya, pada kode di bawah ini kita memiliki akses pada **property** `firstName` :

```
agentName = `Agent ${this.firstName}`
```

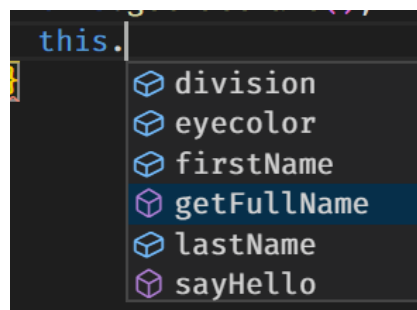
Protected Behaviour

Jika kita membuat **method** baru dengan nama **sayHello()** di dalamnya kita dapat memanggil **method** **getFullName()** yang dimiliki oleh **super class**. Maka access modifier mengijinkan hal tersebut seperti yang ada pada kode di bawah ini :

```
class Agent extends Person {  
    division = "";  
  
    sayHello(): void {  
        this.getFullName();  
    }  
}
```

Private Behaviour

Jika kita mencoba melakukan akses pada **property** **age** dalam **superclass** maka informasinya tidak akan ditampilkan.



Gambar 264 Private Modifier

Readonly Property

Readonly property digunakan jika kita ingin membuat sebuah **property** yang hanya bisa dibaca dan tidak dapat diubah lagi. Perhatikan kode di bawah ini :

```
class Person {  
  readonly firstName: string;  
  
  constructor(  
    firstName: string,  
    public lastname: string,  
    public age: number  
  ) {  
    this.firstName = firstName;  
  }  
}
```

Pada kode di bawah ini jika kita mencoba mengubah nilai pada **readonly property** maka **typescript compiler** akan memberikan pesan **error** :

```
const hooman = new Person("Maudy", "Ayunda", 25);  
hooman.firstName = "Change it!!"; //error  
console.log(hooman.lastname); //Ayunda
```

4. Class 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.

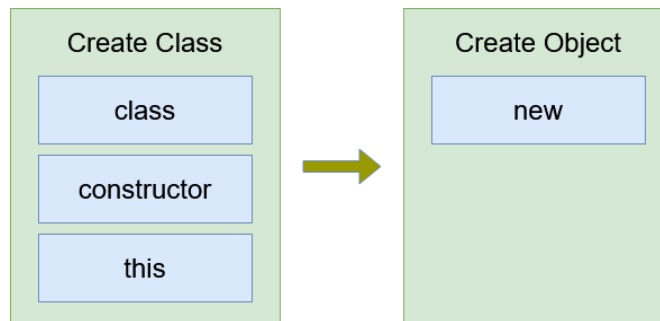
Sebelumnya kita membuat **property** pada **class** secara eksplisit, selanjutnya jika kita ingin membuat **class** dengan versi yang lebih **verbose** dan fleksibel kita dapat menggunakan **constructor** seperti pada kode di bawah ini :

```
class Person {  
  firstName = "";  
  lastName = "";  
  age = 0;  
  
  constructor(firstName: string, lastName: string, age: number) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
  }  
}
```

Untuk membuat **object** cukup dengan satu **statement** :

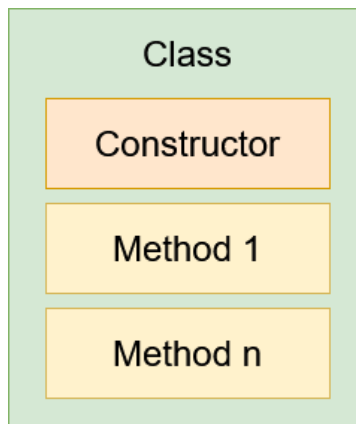
```
const human = new Person("Maudya", "Ayunda Faza", 25)
```

Terdapat 3 *keywords* utama yang digunakan yaitu **class**, **constructor** & **this** *keyword*.



Gambar 265 Class Keywords

Selain membuat **class** member **property** kita juga dapat membuat **class member method**. Di dalam sebuah **class** kita dapat membuat sebuah **constructor** dan sekumpulan **methods** :



Gambar 266 Class Structure

Spread Argument

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 : string[]) {
    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.

5. Static Keyword

Sejak ES6 kita dapat memiliki **property** yang nilainya dapat digunakan oleh banyak **object** dari suatu **class**. Sebagai contoh pada kode di bawah ini kita membuat sebuah **class Archer** dengan **static property** di dalamnya :

```
class Archer {  
  static totalArrow = 20;  
  
  shoot(): void {  
    Archer.totalArrow--;  
    console.log(`Total Arrow left : ${Archer.totalArrow}`);  
  }  
}
```

Kita akan membuat dua buah **objects** yang akan menggunakan **static property** :

```
const Maudy = new Archer();  
Maudy.shoot();  
  
const Gun = new Archer();  
Gun.shoot();
```

Keluaran yang akan diproduksi adalah :

```
/* output :  
Total Arrow left : 19  
Total Arrow left : 18 */
```

Jika kita amati kedua **objects** tersebut berbagi total **arrow** yang tersedia di dalam property totalArrow

6. Super Method

Ada saatnya kita ingin memanggil **costructor** yang ada pada **super class** menggunakan **keyword super**. Hal ini terjadi karena terdapat property yang sama antara super class dan sub class.

Manfaat penggunaan **method super** adalah kita dapat mengisi **property** pada **super class** melalui sebuah **sub class** :

```
class Agent extends Person {  
  constructor(  
    firstName: string,  
    lastName: string,  
    age: number,  
    public division: string,  
  ) {  
    super(firstName, lastName, age);  
  }  
}
```

Pada **constructor class Agent** didalamnya terdapat **super method** yang memanggil **constructor** dari parent **class Person**. Perhatikan pada kode di bawah ini :

```
const CIA = new Agent("Gun Gun", "Febrianza", 28, "IT");  
console.log(CIA.division); //IT  
console.log(CIA.firstName); //Gun Gun
```

Pada **argument** pertama, kedua dan ketiga akan terkirim untuk **constructor class** Person sementara argument keempat yaitu **"IT"** akan terkirim untuk **constructor class Agent**.

7. Method Override

Jika kita memiliki **method** dengan nama yang sama antara **super class** dan **sub class** maka diperlukan manajemen penamaan **method**. Jika kita ingin menimpa **method** yang dimiliki oleh **parent class** agar memiliki **behaviour** baru pada **sub class** maka kita dapat menggunakan *method overriding*.

Pada kode di bawah ini **parent class** memiliki **method** `getFullName()` :

```
class Person {
  constructor(
    public firstName: string,
    public lastName: string,
    private age: number,
  ) {}
  protected getFullName(): string {
    return `Fullname : ${this.firstName} ${this.lastName}`;
  }
}
```

Jika pada **sub class** kita ingin memiliki **method** dengan nama yang sama namun dengan **behaviour method** yang berbeda, maka kita dapat mendeklarasikan ulang **method** `getFullName()` pada **sub class** :

```
class Agent extends Person {
  constructor(
    firstName: string,
    lastName: string,
    age: number,
    public division: string,
  ) {
```

```

    super(firstName, lastName, age);
  }
  getFullName(): string {
    return `Agent ${this.firstName} ${this.lastName}`;
  }
}

```

Pada kode di atas *method* `getFullName()` pada **sub class** akan menerima **method** milik **parent class**, mari kita buktikan dengan menulis dan mengeksekusi kode di bawah ini :

```

const CIA = new Agent("Gun Gun", "Febrianza", 28, "IT");
console.log(CIA.getFullName()); //Agent Gun Gun Febrianza

```

Lalu bagaimana jika kita ingin tetap memiliki fungsi *original* dari *parent class*?

```

getFullName(): string {
  return `Agent ${super.getFullName()}`;
}

```

**Link sumber kode.*

Perhatikan pada *method* `getFullName()` di atas kita menggunakan *keyword* **super** untuk mengakses *method* `getFullName()` yang dimiliki oleh *parent class*.

8. Accessor Getter & Setter

Typescript juga mendukung konsep **accessor** menggunakan **getter** & **setter** agar kita memiliki **finer-gained control** untuk setiap **property** yang akan diakses. Di dalam sebuah **class** kita dapat membangun **getter & setter** untuk melakukan komputasi pada **properties** dari **object** yang akan diciptakan.

Perhatikan kode di bawah ini :

```
class Person {
  constructor(
    public firstName: string,
    public lastName: string,
    private _age: number,
  ) {}

  get age(): number {
    return this._age;
  }
  set age(newAge: number) {
    this._age = newAge;
  }
}
```

Pada kode di atas kita membuat sebuah **getter** untuk **property** `_age` :

```
get age(): number {
  return this._age;
}
```

Juga kita membuat sebuah **setter** :

```
set age(newAge: number) {  
  this._age = newAge;  
}
```

Kemudian kita membuat **object** dari **class Person** :

```
const Maudy = new Person("Maudy", "Ayunda", 25);
```

Untuk menggunakan **setter** gunakan gaya **assignment statement** seperti pada kode di bawah ini :

```
Maudy.age = 26;
```

Untuk menggunakan **getter** perhatikan kode di bawah ini :

```
if (Maudy.age) {  
  console.log(Maudy.age);  
}
```

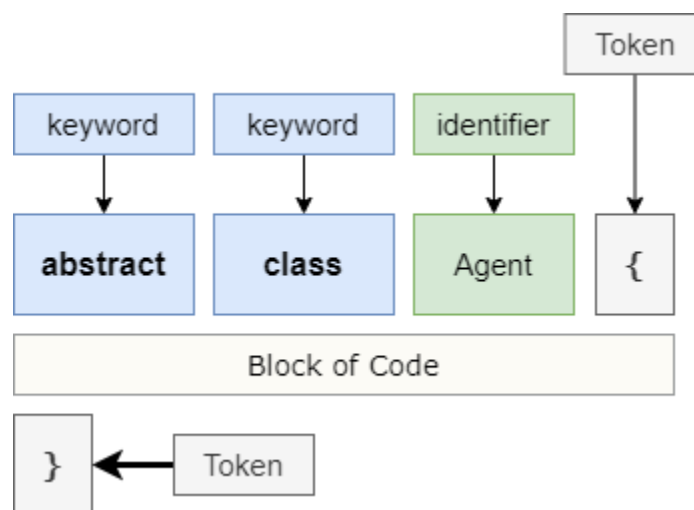
9. Abstract Class

Sebuah **Abstract Class** adalah **class** yang hanya dijadikan acuan untuk **class** turunannya, **abstract class** tidak didesain agar bisa ditransformasikan ke dalam sebuah **object**. Sebuah **abstract class** menjelaskan **implementation detail** yang wajib ditiru oleh **sub class**.

Kita akan belajar cara membuat sebuah **Abstract Class**, perhatikan kode **Abstract Class** di bawah ini :

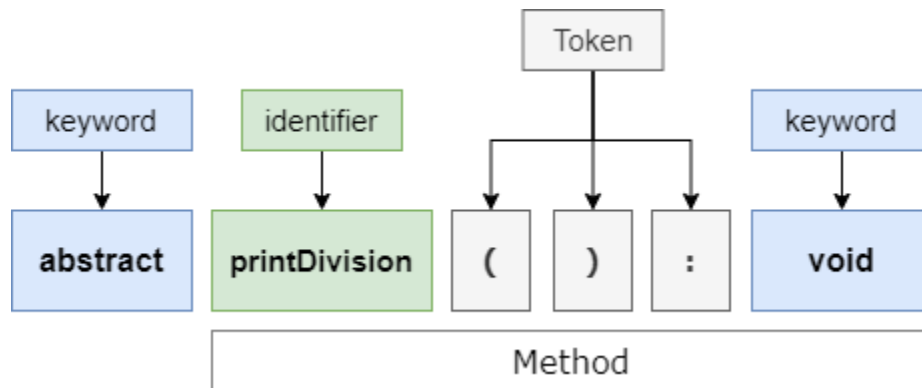
```
abstract class Agent {  
    constructor(public fullName: string) {  
    }  
    printName(): void {  
        console.log(this.fullName);  
    }  
    abstract printDivision(): void;  
}
```

Jika kita perhatikan cara membuat membuat **abstract class** diagramnya adalah sebagai berikut :



Gambar 267 Create Abstract Class

Selain itu pada **abstract class** di atas juga terdapat **abstract method** dengan struktur **syntax** sebagai berikut :



Gambar 268 Abstract Method Diagram

Selanjutnya kita akan membuat sebuah **class** turunan dari **abstract class Agent** bernama **class AgentCIA**, maka kita wajib memenuhi member **class** yang ada pada **abstract class**. Salah satunya adalah membuat **method** dengan nama **printDivision** sesuai dengan **member class** yang ada pada **abstract class** :

```
class AgentCIA extends Agent {
  constructor(public fullName: string, public division: string) {
    super(fullName);
  }
  public printDivision(): void {
    console.log(`Agent ${this.fullName} from ${this.division}`);
  }
}
```

Jika **method printDivision()** tidak dibuat pada **sub class** maka **compilation** akan mengalami **error**. Selanjutnya kita melakukan deklarasi variabel dengan tipe **Agent** :

```
let agentCIA: Agent;
```

Variabel **agentCIA** digunakan untuk menyimpan **Class AgentCIA** :

```
agentCIA = new AgentCIA("Gun Gun Febrianza", "Counter-terorrism Division");  
agentCIA.printDivision();
```

10. Class Declaration

Kita **refresh** kembali dan evaluasi lagi, setelah mempelajari konsep **class**, jika class yang dibuat memiliki **property** selalu gunakan **constructor** dan **access modifier** saat kita membuat sebuah **class**. Contoh *class declaration* :

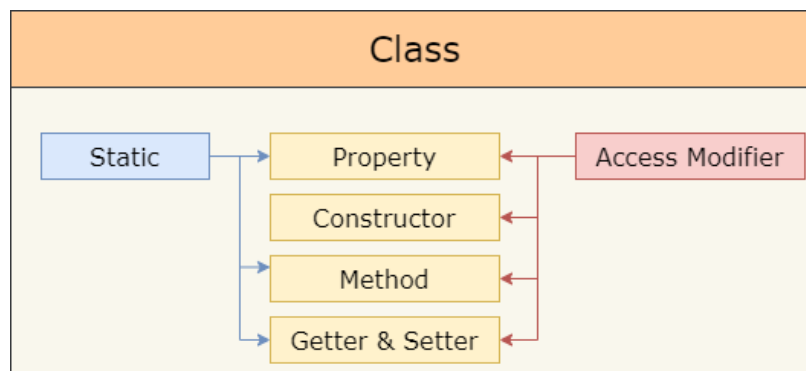
```
class Wallet {  
  constructor(public id: number, public balance: number) {  
    this.id = id;  
    this.balance = balance;  
  }  
}
```

Untuk menggunakan *class* tersebut perhatikan kode di bawah ini :

```
let myWallet = new Wallet(1,2000)  
console.log(myWallet);  
//wallet { id: 1, balance: 2000 }
```

**Link sumber kode.*

Setelah mempelajari sampai halaman ini kita sampai pada sebuah kesimpulan bahwa sebuah **class** terdiri dari beberapa komponen yang dapat digunakan sesuai kebutuhan.



Gambar 269 Class Components

Subchapter 12 – Interface

Don't reinvent the wheel, just realign it.

— Anthony J. D'Angelo

Subchapter 12 – Objectives

- Memahami Apa itu **Interface**?
 - Memahami Bagaimana suatu **Class** melakukan **implement Interface**
 - Memahami Bagaimana suatu **Interface** melakukan **extend Interface**
 - Memahami Perbedaan **implement** dan **extend**
 - Memahami Salah satu Aplikasi dari **Interface**
-

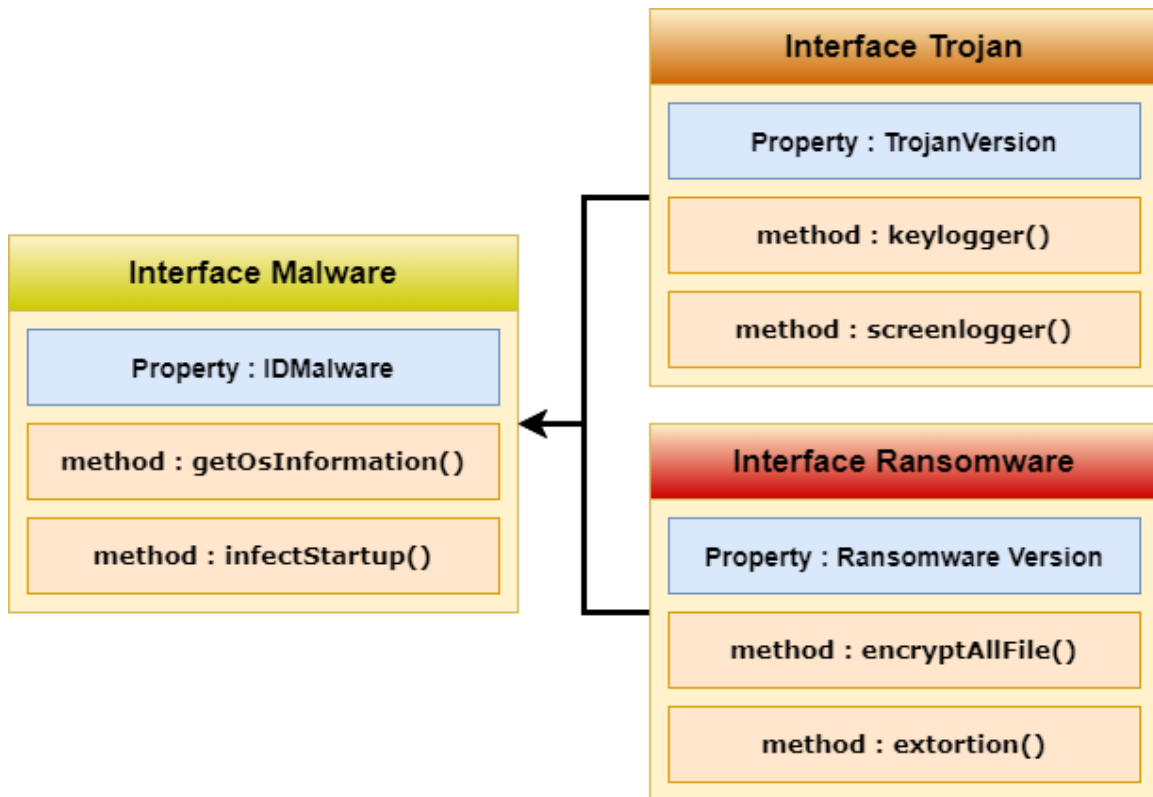
Sebuah **interface** menyediakan kita mekanisme **property** dan **method** apa yang harus disediakan oleh suatu **object**. Dalam paradigma pemrograman **object oriented (OOP)** aturan itu disebut dengan **contract**, aturan yang harus dilakukan oleh sebuah **class** jika ingin menggunakan suatu **interface**.

Kita dapat menggunakan **interface** untuk :

1. Membuat sebuah **Object**,
2. Sebagai **parameter** suatu **function**
3. Sebagai **return** dari suatu **function**

1. Design Interface

Kita akan belajar membuat sebuah **interface** dengan **study** kasus dari pengembangan sebuah aplikasi **virus computer**. Perhatikan diagram gambar di bawah ini :



Gambar 270 Example Interfaces

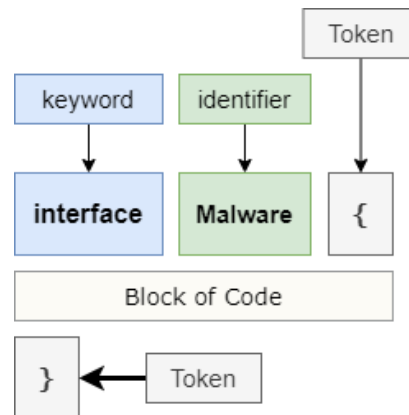
Create Malware Interface

Pada diagram di atas kita memiliki 3 buah **interface** dimana **malware interface** adalah **malware** yang menjadi dasar untuk pengembangan **malware** lainnya. Selanjutnya **interface Ransomware** dan **interface Trojan** adalah turunan dari **interface Malware**.

Jika diagram di atas kita ubah ke dalam sebuah **interface** :

```
interface Malware {
    IDMalware: string;
    getOsInformation(): void;
    infectStartup(): void;
}
```


Jika kita perhatikan struktur ***syntax interface*** jika dikonversi ke dalam ***diagram*** :

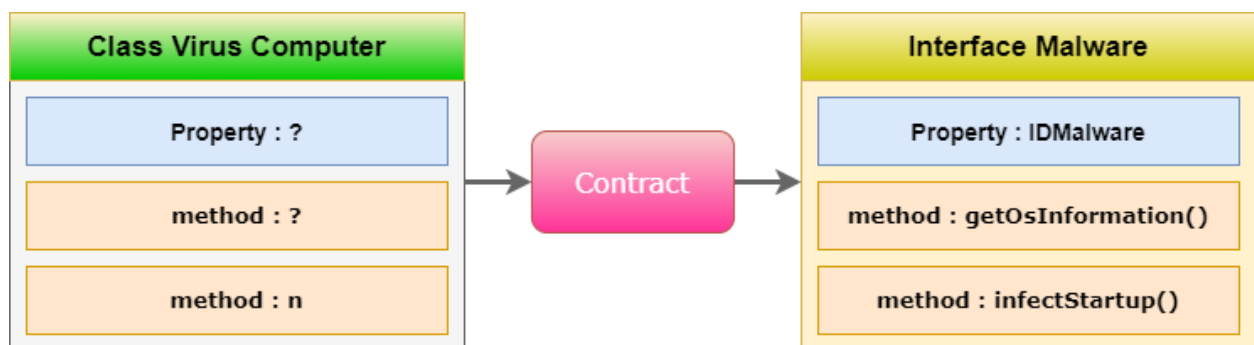


Gambar 271 Interface Syntax Structure

Selanjutnya bagaimana cara membuat ***class*** yang dapat menggunakan ***interface*** **Malware**? Kita ingat bahwa arti dari ***contract*** adalah seluruh ***class member*** (***property & methods***) dari suatu ***interface*** harus dibuat sebagai syarat minimum.

Implement Malware Interface

Untuk bisa menggunakan ***Interface*** **Malware**, ***class*** yang dibuat harus memenuhi ***contract*** terlebih dahulu dengan cara memenuhi syarat ***member class*** pada ***interface*** **Malware**. Ilustrasinya dapat anda lihat pada gambar di bawah ini :



Gambar 272 Implement Malware Interface

Pada kode dibawah ini kita membuat sebuah **class** bernama **Virus** yang selanjutnya memanfaatkan **interface Malware** :

```
class Virus implements Malware {
  public IDMalware: string;
  constructor(IDmalware: string) {
    this.IDMalware = IDmalware;
  }
  getOSInformation(): void {
    console.log("OS Windows : User Maudy");
  }
  infectStartup(): void {
    console.log("Startup OS Infected");
  }
}
```

Jika kita perhatikan secara seksama pada kode di atas, **class member (property & methods)** dari **interface Malware** wajib kita buat terlebih dahulu sebagai syarat minimum di dalam **class**. Sebab jika tidak artinya kontrak tidak dipenuhi dan **typescript compiler** akan memberikan **error**.

Untuk membuat **object** dari **class Virus** eksekusi kode di bawah ini :

```
const exampleVirus : Virus = new Virus("101");
```

Selanjutnya kita dapat memanggil **methods** yang dimiliki oleh **object exampleVirus** :

```
exampleVirus.getOSInformation();
exampleVirus.infectStartup();
```

Selanjutnya kita dapat menambahkan **method** baru pada **class Virus** agar bisa menampilkan identitas **malware** :

```
...
getIDMalware(): void {
    console.log(this.IDMalware);
}
...
```

Untuk menggunakannya cukup panggil **method** yang sudah dibuat :

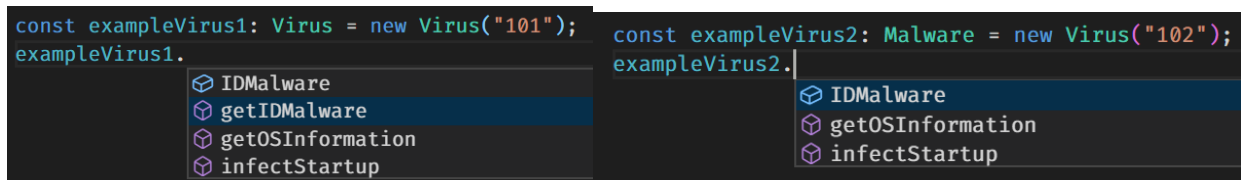
```
exampleVirus1.getIDMalware();
```

Class Type & Interface Type

Okay kita sudah cukup faham bagaimana implementasi suatu **interface**, selanjutnya apa perbedaan dari dua **statement code** di bawah ini :

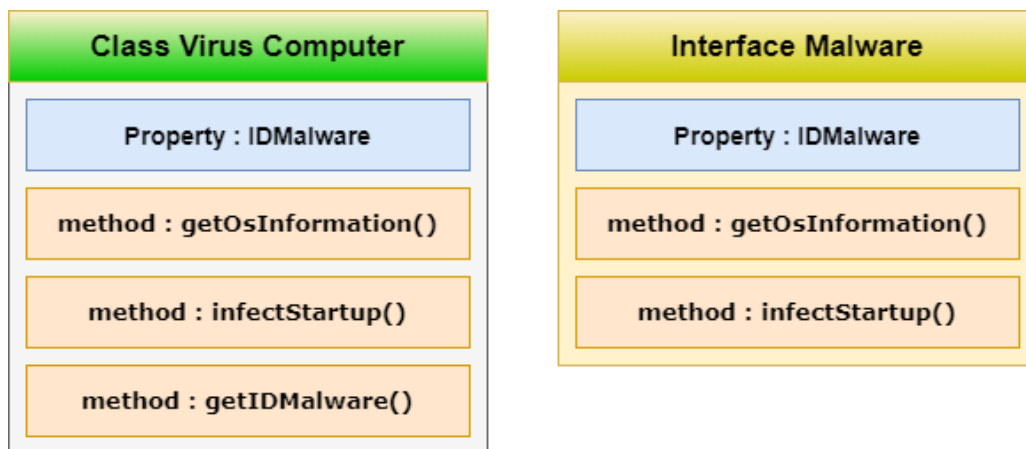
```
const exampleVirus1: Virus = new Virus("101");  
const exampleVirus2: Malware = new Virus("102");
```

Jika kita amati perbedaannya terletak pada **member class**, silahkan buktikan dengan mencoba memanfaatkan fitur **intellisense** pada **visual studio code** :



Gambar 273 Difference Class & Interface Type

Jika ditransformasikan ke dalam visual perbedaannya terlihat jelas :



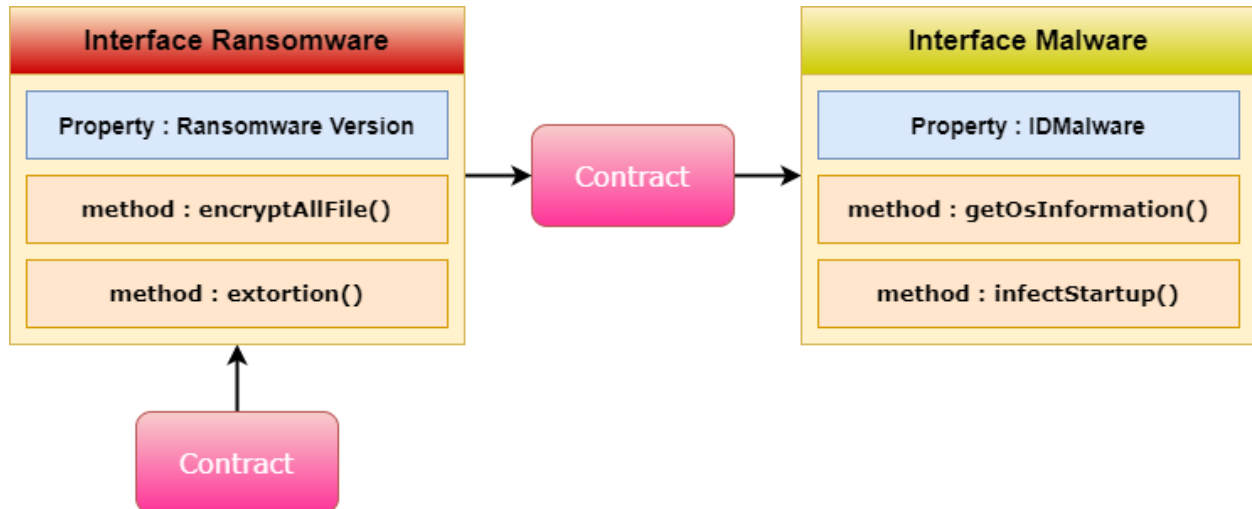
Gambar 274 Class & Interface Type

Create Ransomware Interface

Pada studi kasus kali ini kita akan membuat sebuah **interface** yang merupakan turunan dari suatu **interface**. Kita akan membuat **interface Ransomware** yang menjadi turunan dari **interface Malware** :

```
interface Ransomware extends Malware {  
    RansomwareVersion: string;  
    encrypt(): void;  
    extortion(): void;  
}
```

Jika kita ingin menggunakan **interface Ransomware** maka kita harus memenuhi dua **contract** yaitu **interface Ransomware** & **Malware**. Jika di ilustrasikan secara visual maka perhatikan gambar di bawah ini :

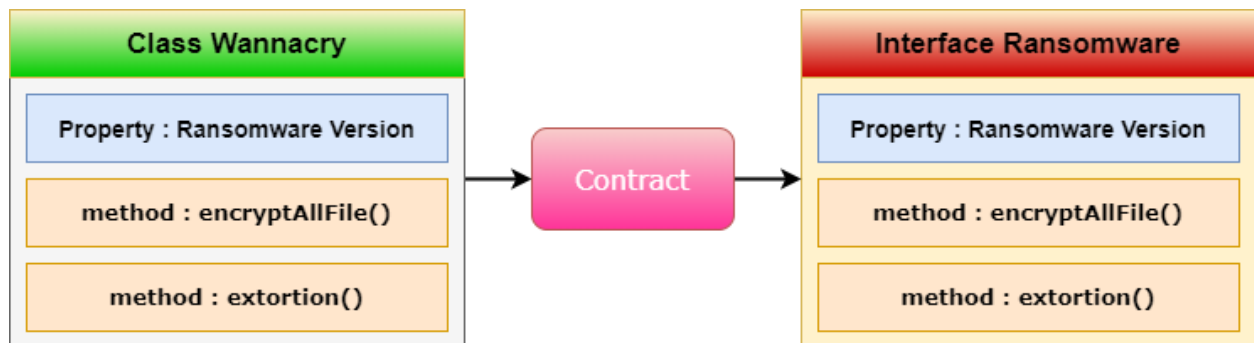


Gambar 275 Interface Ransomware

Ok selanjut kita tulis sebuah **class** yang dapat memenuhi **contract** untuk menggunakan **interface Malware** dan **Ransomware**.

Implement Ransomware Interface

Pada kasus kali ini kita akan membuat sebuah **class** dengan nama **Wannacry**, **class** tersebut akan menggunakan **interface** dari **Ransomware**. Maka kita harus memenuhi kontrak pada **interface Ransomware** terlebih dahulu :

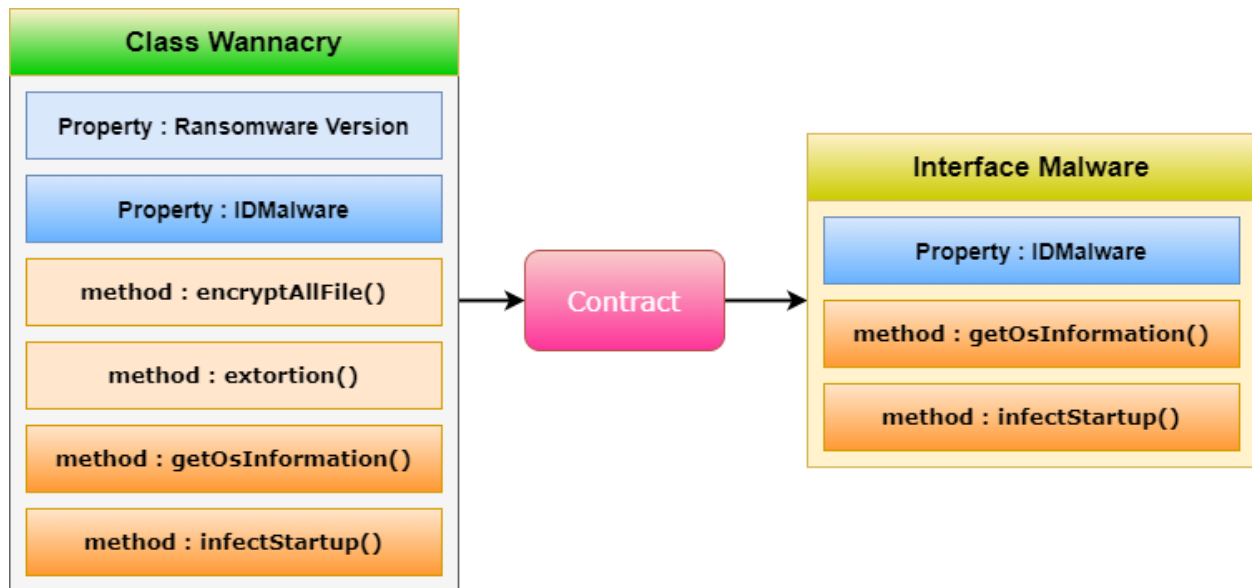


Gambar 276 Contract Ransomware

Jika diagram di atas kita transformasikan kedalam kode maka :

```
class WannaCry implements Ransomware {
    public RansomwareVersion: string;
    encrypt(): void {
        console.log("Encrypt All Files, Success");
    }
    extortion(): void {
        console.log("Send Bitcoin to : 8sdhshj6sgjhd&");
    }
}
```

Selanjutnya kita harus memenuhi kontrak untuk **interface Malware** karena **Ransomware** adalah turunan dari **Interface Malware**. Perhatikan diagram di bawah ini agar kita mengetahui kontrak apa saja yang harus dipenuhi :



Gambar 277 Implement Interface Malware

Jika diagram di atas ditransformasikan kedalam kode, perhatikan kode dibawah ini :

```
class WannaCry implements Ransomware {
    public RansomwareVersion: string;
    public IDMalware: string;
    getOsInformation(): void {
        console.log("OS Windows : User Maudy");
    }
    infectStartup(): void {
        console.log("Startup OS Infected");
    }
    encryptAllFiles(): void {
        console.log("Encrypt All Files, Success");
    }
    extortion(): void {
        console.log("Send Bitcoin to : 8sdhshj6sgjhd&");
    }
}
```

Untuk membuat **object** dari **class** `WannaCry` :

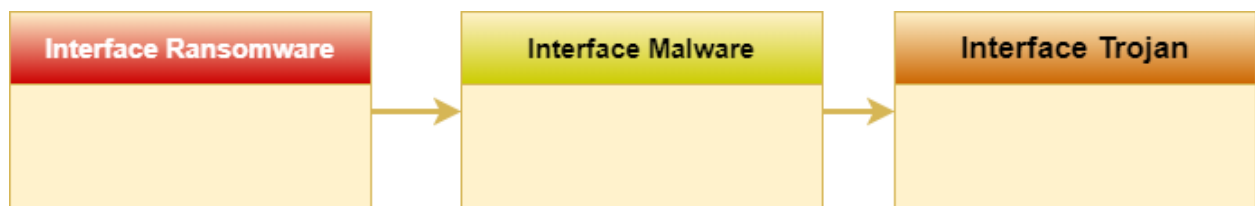
```
const exampleVirus: WannaCry = new WannaCry();
```

Panggil **method** yang dimiliki oleh **object** `exampleVirus` dari **class** `WannaCry`:

```
exampleVirus.encryptAllFiles()  
exampleVirus.extortion()
```

Interface Extend Multi-Interface

Jika kita ingin agar implementasi `Ransomware` memerlukan **interface** lainnya, kita dapat melakukan **extend** pada multi **interface** sekaligus. Misalkan kita harus menambahkan **interface** `Trojan` sebagai syarat untuk implementasi **interface** `Ransomware`.



Gambar 278 Multi-interface

Create Trojan Interface

Sekarang kita akan membuat **interface** untuk `Trojan`, tulis kode di bawah ini :

```
interface Trojan {  
  IDTrojan: string;  
  keylogger(): void;  
  screenlogger(): void;  
}
```


Implement Multi-interface

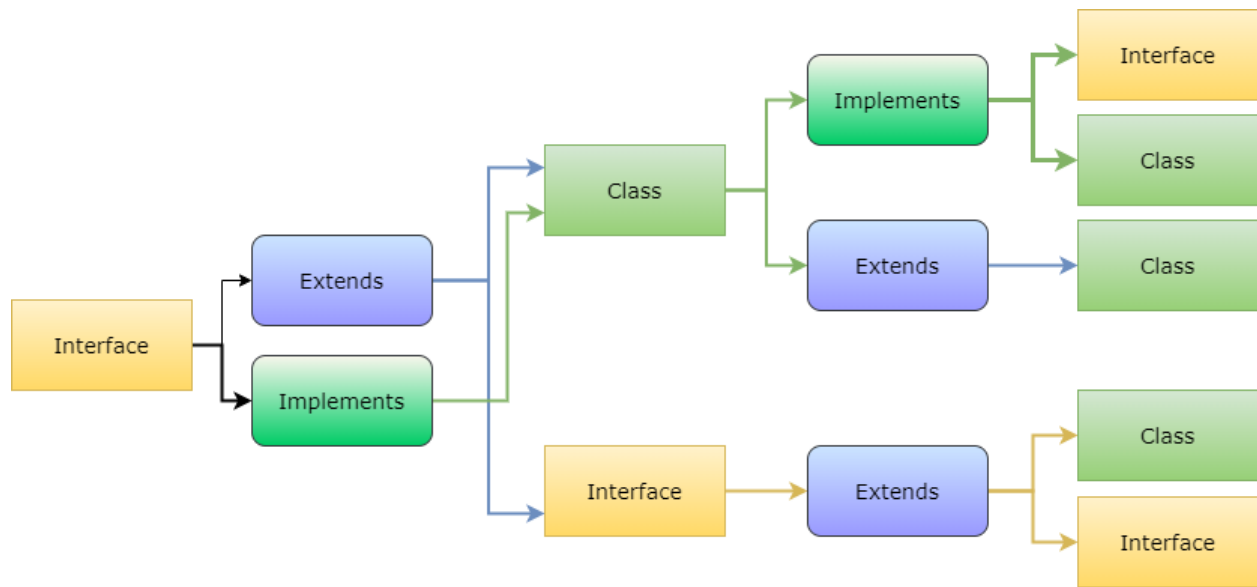
Untuk memastikan bahwa syarat untuk menggunakan ***interface*** **Ransomware** harus memenuhi ***contract*** pada ***interface*** **Trojan**, cukup tambahkan koma dan nama ***interface*** **Trojan** setelah ***interface*** **Malware**. Seperti pada kode di bawah ini :

```
interface Ransomware extends Malware, Trojan {  
    RansomwareVersion: string;  
    encryptAllFiles(): void;  
    extortion(): void;  
}
```

Selanjutnya ketika kita membuat sebuah ***Class*** yang akan menggunakan ***interface*** **Ransomware**, 3 ***contract*** perlu dipenuhi terlebih dahulu.

2. Interface & Class

Setelah kita mempelajari **class**, **inheritance** dan **interface** kita dapat membuat kesimpulan dalam bentuk **diagram** agar lebih mudah difahami. Perhatikan diagram di bawah ini :



Gambar 279 Interface Characteristic

Perbedaan yang signifikan dari **Interface** & **Class** pada **typescript** adalah :

Interface Extends Interface

Sebuah **interface** dapat melakukan **extends** pada sebuah **interface**.

Interface Extends Class

Sebuah **interface** dapat melakukan **extends** pada sebuah **class**.

Interface Cant Implements Interface

Sebuah **interface** tidak dapat melakukan **implements** pada sebuah **interface**.

Interface Implements Class

Sebuah ***interface*** dapat melakukan ***implements*** pada sebuah ***class***.

Class Cant Extends Interface

Sebuah ***class*** tidak dapat melakukan ***extends*** pada sebuah ***interface***.

Class Extends Class

Sebuah ***class*** dapat melakukan ***extends*** pada sebuah ***class***.

Class Implements Interface

Sebuah ***class*** dapat melakukan ***implements*** pada sebuah ***interface***.

Class Implements Class

Sebuah ***class*** dapat melakukan ***implements*** pada sebuah ***class***.

Subchapter 13 – 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

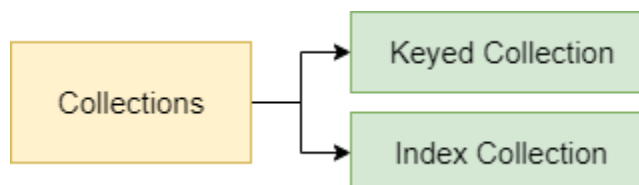
Subchapter 13 – Objectives

- 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?

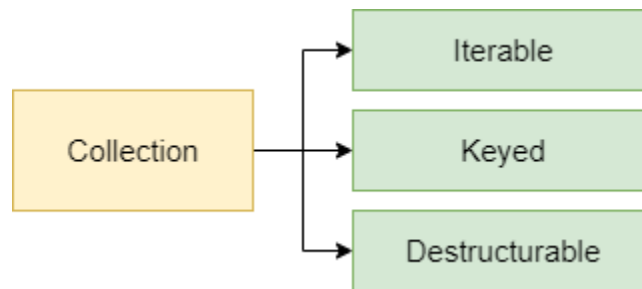
Collection adalah **data** dalam kesatuan tunggal yang merepresentasikan kumpulan kumpulan **object references**. **Collections** juga dapat disebut sebagai *containers*^[26].

Dalam **javascript** sendiri terdapat dua **collections**, yaitu **Keyed Collections** dan **Indexed Collections**.



Gambar 280 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 281 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?

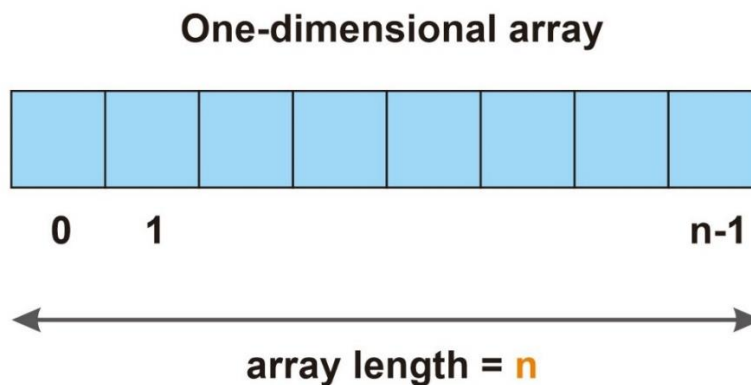
2. Apa itu *Indexed Collections*?

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*, *deconstructable* dan tidak memiliki karakteristik *keyed*.

Dalam buku berjudul *Javascript – The Definitive Guide* karya **David Flanagan** yang diterbitkan pada tahun 2002, dikatakan 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 282 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, untuk cara yang ketiga kita akan menggunakan **typescript** :

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** :

```
var mod = new Array()  
mod[0] = "Maudy"  
mod[1] = "Ayunda"  
mod[2] = "Faza"
```

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 kode di bawah ini :

```
var mod = new Array("Maudy", "Ayunda", "Faza")
console.log(mod);
```

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 kode di bawah ini :

```
var mod = ["Maudy", "Ayunda", "Faza"];
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi :

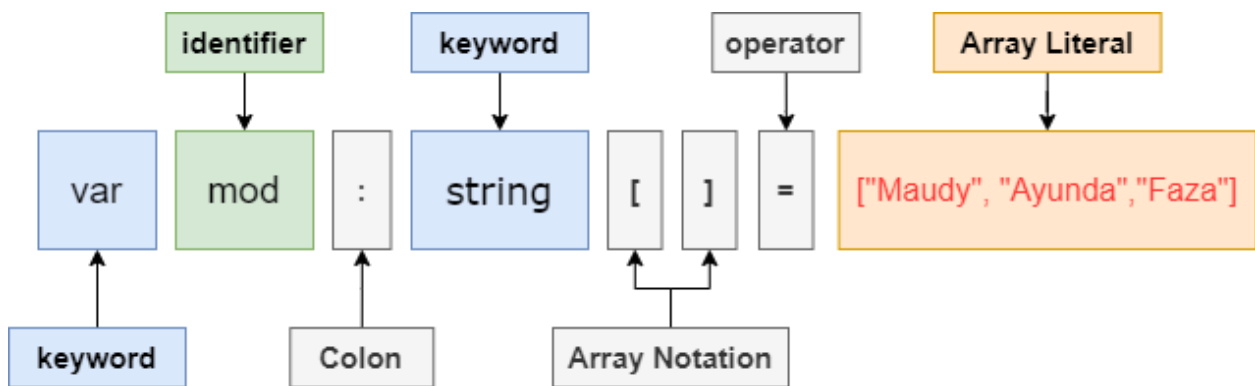
```
[ "Maudy", "Ayunda", "Faza" ]
```


Array Type

Pada **typescript** sebuah **array** memerlukan data tipe kita dapat menggunakan **primitive type** atau **reference type**, pada kasus kali ini kita akan membuat **array** menggunakan **primitive type string**.

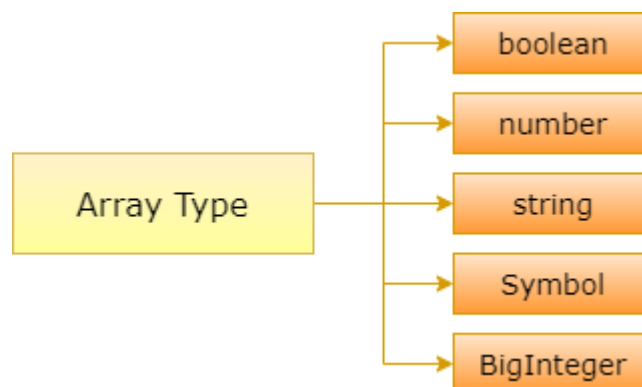
```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];
```

Jika kita ubah **statement** di atas ke dalam diagram maka kita dapat melihat strukturnya secara detail :



Gambar 283 Array Declaration

Kita dapat membangun **array** dengan berbagai tipe data primitif :



Gambar 284 Array Type

Di bawah ini adalah contoh **array** lengkap dengan berbagai tipe data primitif :

```
const arrSym: Symbol[] = [Symbol("maudy"), Symbol("ayunda")];
const arrBig: BigInt[] = [1n, 2n, 3n, 3289327323543264324n];
const arrBol: boolean[] = [false, true, false, true, false];
const arrStr: string[] = ["Gun Gun", "Febrianza", "Maudy"];
const arrInt: number[] = [1, 2, 3, 4, 5, 6, 7, 8, 55, 33];
```

Akses Array Element

Selain membuat sebuah **array** kita juga bisa melakukan akses pada salah satu **element** yang ada di dalam sebuah **array**. Sebagai contoh pada kode di bawah ini kita bisa mengakses **array** yang ada pada indeks pertama (dimulai dari nol).

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];
console.log(mod);
console.log(mod[0]);
console.log(mod[1]);
console.log(mod[2]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*
[ "Maudy", "Ayunda", "Faza" ]
Maudy
Ayunda
Faza
*/
```

Modify Array Element

Selain itu kita bisa mengubah salah satu **element** yang ada di dalam **array**, caranya bisa kita lihat pada kode di bawah ini :

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];  
mod[2] = "cantik";  
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Maudy", "Ayunda", "cantik" ]
```

Array mod pada indeks kedua nilainya diubah dari **faza** menjadi **cantik**, untuk membuktikanya kita bisa mengakses lagi **element** tersebut seperti pada gambar di atas.

Iterate Array

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** :

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];  
var i: number = 0;  
for (i; i < mod.length; i++) {  
    console.log(i + " " + mod[i]);  
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*  
0 Maudy  
1 Ayunda  
2 Faza  
*/
```

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 kode di bawah ini kita membuat **callback** untuk menampilkan **index** dan **element** yang dimiliki oleh **array** `mod` :

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];  
mod.forEach(function (item: string, index: number) {  
    console.log(item, index);  
});
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*  
Maudy 0  
Ayunda 1  
Faza 2  
*/
```

Array Looping for...of

Selain menggunakan **ES 6** kita juga telah mempelajari sebelumnya melakukan perulangan menggunakan **for...of statement** pada bab **looping**.

Array Looping for...in

Selain menggunakan **ES 6** kita juga telah mempelajari sebelumnya melakukan perulangan menggunakan **for...in statement** pada bab **looping**.

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: string[] = ["Maudy", "Ayunda Faza", "Gun Gun", "Febrianza"];
var x = soul.entries();
var n: [number, string];
for (n of x) {
    console.log(n = n);
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*
[ 0, "Maudy" ]
[ 1, "Ayunda Faza" ]
[ 2, "Gun Gun" ]
[ 3, "Febrianza" ]
*/
```

Untuk membuktikan bahwa **x** adalah **iterator object**, panggil **variable** tersebut :

```
console.log(x);  
//Array Iterator {}
```

Array Destructuring (ES6)

Destructuring mempermudah pekerjaan kita jika ingin melakukan ekstraksi 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: string[] = ["Maudy", "Ayunda", "Faza"];  
var [satu, dua] = mod;  
console.log(satu);  
console.log(dua);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*  
Maudy  
Ayunda  
*/
```

Array Map

Method **map()** digunakan jika kita ingin mendapatkan **array** baru setelah kita mengeksekusi sebuah **callback** untuk setiap **element** yang ada di dalamnya.

Pada kode 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: string[] = ["Maudy", "Ayunda", "Faza"];
var gun = mod.map(function (item) {
    return item.toUpperCase();
});

console.log(gun);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "MAUDY", "AYUNDA", "FAZA" ]
```

Array Filter

Method `filter()` digunakan jika kita ingin mendapatkan **array** baru yang dihasilkan dari **callback** dengan **return** bernilai **true**.

```
const array1: (string | number)[] = ["maudy", 100, "ayunda", 200,
"faza", 300];
const array2 = array1.filter(function (item) {
    return typeof item === "string";
});
console.log(array2);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "maudy", "ayunda", "faza" ]
```

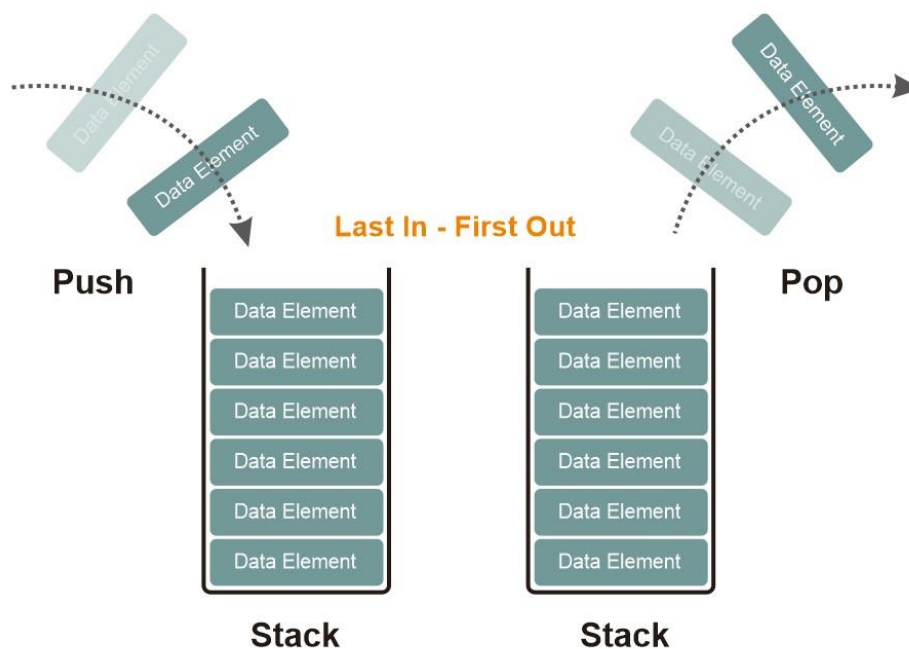
Array Property & Method

Array Properties

Sebuah **array** memiliki **property** `length` untuk mengetahui jumlah **element** yang dimilikinya. Pada kode di bawah ini didapatkan jumlah **element** dari **array** `mod`.

```
const mod: string[] = ["Maudy", "Ayunda", "Faza"];  
console.log(mod.length);
```

Push Array



Gambar 285 Push & Pop Method Visualization

Kita bisa menggunakan **method** `push()` yang dimiliki **array** untuk menambahkan **element** baru diakhir elemen sebuah **array**, caranya bisa kita lihat pada kode di bawah ini :

Pop Array

Method `pop()` digunakan untuk menghapus **element** terakhir di dalam sebuah **array**.

Bisa kita lihat pada kode di bawah ini :

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];
mod.pop();
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Maudy", "Ayunda" ]
```

Shift Array

Dalam **array** kita bisa menggunakan **method** `shift` untuk menghapus **element** yang posisinya di awal dalam sebuah **array**. Pada kode di bawah ini **element** awal atau indeks pertama dari **array** `mod` di hapus.

```
var mod: string[] = ["Maudy", "Ayunda", "Faza"];
mod.shift();
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Ayunda", "Faza" ]
```

Unshift Array

Untuk menambahkan sebuah **element** di awal elemen sebuah **array** kita bisa menggunakan **method** `unshift`, cara penggunaanya bisa kita lihat pada kode di bawah ini :

```
const mod: string[] = ["Maudy", "Ayunda", "Faza"];  
mod.unshift("Maudy");  
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Maudy", "Maudy", "Ayunda", "Faza" ]
```

Find Index Array

Untuk mengetahui **index** sebuah **element** kita bisa menggunakan **method** `indexOf` yang dimiliki sebuah **array**. Bisa kita lihat pada kode di bawah ini :

```
const mod: string[] = ["Maudy", "Ayunda", "Faza"];  
console.log(mod.indexOf("Ayunda"));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
//1
```

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 kode di bawah ini :

```
const mod: string[] = ["Maudy", "Ayunda", "Faza"];
mod.splice(2);
console.log(mod);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Maudy", "Ayunda" ]
```

Copy an Array

Dalam **Javascript** untuk mendapatkan salinan sebuah **array** kita bisa menggunakan **slice function**, bisa kita lihat pada kode di bawah ini :

```
const mod: string[] = ["Maudy", "Ayunda", "Faza"];
var firstname = mod.slice(1);
console.log(firstname);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
//[ "Ayunda", "Faza" ]
```

Merge Array

Untuk menggabungkan dua buah **array** atau lebih menjadi satu kita bisa menggunakan **method** `concat()`. Di bawah ini adalah contoh dua *hati** yang di gabungkan :

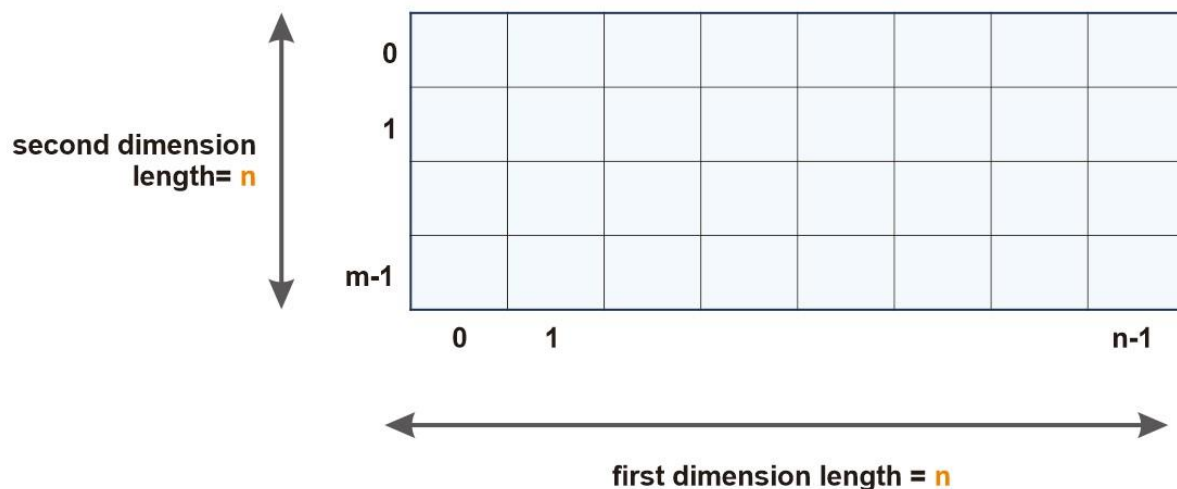
```
const mod: string[] = ["Maudy", "Ayunda Faza"];
const gun: string[] = ["Gun Gun", "Febrianza"];
const together: string[] = mod.concat(gun);
console.log(together);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
// [ "Maudy", "Ayunda Faza", "Gun Gun", "Febrianza" ]
```

Multidimensional Array

Two-dimensional array

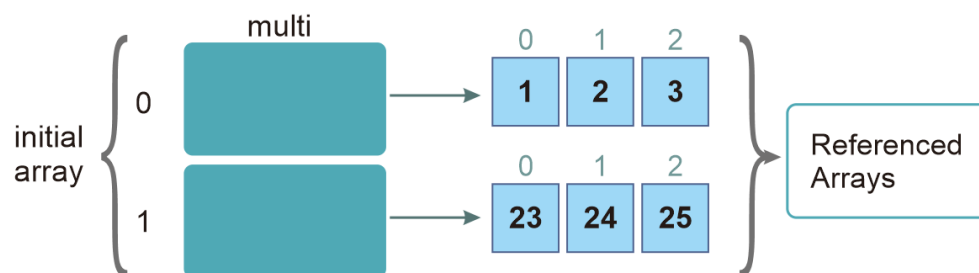


Gambar 286 Multidimensional Array

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** :

```
const multi: number[][] = [[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]);
/*
1
2
3
23
24
25 */
```

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 dari **array multi-dimension** :



Gambar 287 Visualized Multi-dimensional Array

Looping Multidimensional-Array

Untuk melakukan **looping** pada **multidimensional-array** kita perlu melakukan perulangan dua kali, seperti pada kode di bawah ini :

```
const multi: number[][] = [[1, 2, 3], [23, 24, 25]];
var i: number = 0;
for (i; i < multi.length; i++) {
  for (let j: number = 0; j < multi[i].length; j++) {
    console.log(multi[i][j]);
  }
}
```

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** :

```
const matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

console.log(matrix[1][1]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
//5
```

Looping Matrix

Untuk melakukan looping pada matrix perhatikan kode di bawah ini :

```
var i : number = 0
for (i; i < matrix.length; i++) {
    for(let j : number = 0; j < matrix[i].length; j++) {
        console.log("Matrix[" + i + "][" + j + "] = " + matrix[j]);
    }
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
/*
Matrix[0][0] = 1,2,3
Matrix[0][1] = 4,5,6
Matrix[0][2] = 7,8,9
Matrix[1][0] = 1,2,3
Matrix[1][1] = 4,5,6
Matrix[1][2] = 7,8,9
Matrix[2][0] = 1,2,3
Matrix[2][1] = 4,5,6
Matrix[2][2] = 7,8,9
*/
```

Multi-type Array

Kita juga dapat membuat sebuah **array** dengan berbagai tipe data sekaligus memanfaatkan **type union** :

```
let multi: (string | number)[] = [2, 2, "ayunda", 4, "Maudy"];
```

```
console.log(multi);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** sebagai berikut :

```
//[ 2, 2, "ayunda", 4, "Maudy" ]
```

Array of Interface

Kita juga dapat membuat sebuah **array** dengan tipe data **reference** seperti **interface**, untuk melakukannya kita buat dahulu sebuah **interface**. Perhatikan kode di bawah ini :

```
interface IBook {  
  isbn: string;  
  author: string;  
  title: string;  
}
```

Selanjutnya gunakan **interface** sebagai **data type** dilengkapi dengan notasi **array** seperti pada kode di bawah ini :

```
const ebooks: IBook[] = [];
```

Selanjutnya gunakan **method** **push()** untuk menambah data :

```
ebooks.push({  
  isbn: "111111",  
  author: "Gun Gun Febrianza",  
  title: "Belajar Dengan Jenius Deno",  
}, {  
  isbn: "22222",
```



```
author: "Gun Gun Febrianza",  
title: "Belajar Dengan Jenius Blockchain",  
});
```

Untuk melakukan **looping** pada **array of interface** eksekusi kode di bawah ini :

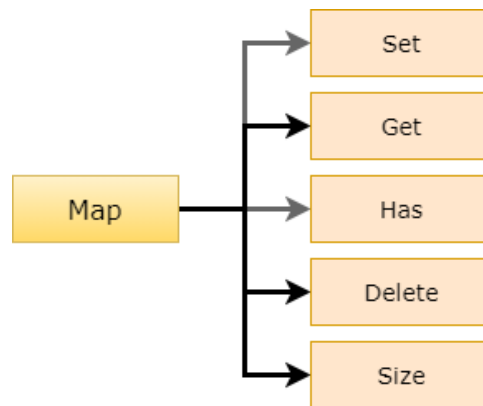
```
for (let i = 0; i < ebooks.length; i++) {  
  console.log(ebooks[i].author);  
  console.log(ebooks[i].title);  
}
```

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** melalui spesifikasi ES6. Di desain untuk menyimpan **key & value pair** menggantikan **object**. Di bawah ini adalah beberapa **method** dalam **map** yang wajib anda pelajari.



Gambar 288 Map Methods

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 kode di bawah ini :

```
let exampleMap = new 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 kode di bawah ini :

```
const exampleMap = new Map();
console.log(exampleMap.set("1", "Maudy"));
// Map { "1" => "Maudy" }
console.log(exampleMap.set(1, "Ayunda"));
// Map { "1" => "Maudy", 1 => "Ayunda" }
console.log(exampleMap.set(true, "Faza"));
// Map { "1" => "Maudy", 1 => "Ayunda", true => "Faza" }
```

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(exampleMap.get("1"));
console.log(exampleMap.get(1));
console.log(exampleMap.get(true));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*
```

```
Maudy  
Ayunda  
Faza  
*/
```

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 :

```
exampleMap.delete(true);  
console.log(exampleMap);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//Map { "1" => "Maudy", 1 => "Ayunda" }
```

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 :

```
exampleMap.clear();  
console.log(exampleMap);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//Map {}
```

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 :

```
console.log(exampleMap.has("1"));  
console.log(exampleMap.has("3"));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
true  
false  
*/
```

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 :

```
console.log(exampleMap.size);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
3  
*/
```

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 :

```
for(const k of exampleMap.keys()) {  
  console.log(k);  
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
1  
1  
true  
*/
```

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 exampleMap.values()) {  
  console.log(v);  
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
Maudy  
Ayunda  
Faza  
*/
```

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 :

```
for (const [k,v] of exampleMap.entries()) {  
  console.log(k,v);  
}
```

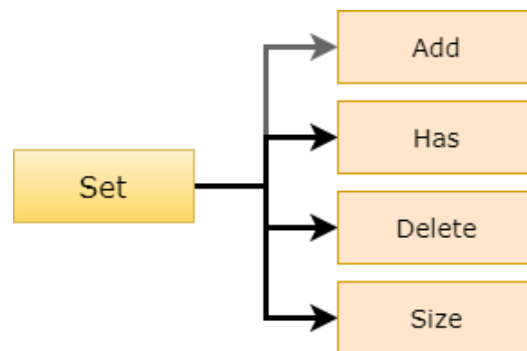
Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
1 Maudy  
1 Ayunda  
true Faza  
*/
```

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*.

Di bawah ini adalah beberapa **method** dalam **set** yang wajib anda pelajari.



Gambar 289 Set Methods

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 kode di bawah ini :

```
const s = new Set();
```


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* :

```
var s = new Set();  
s.add("Maudy");  
s.add("Ayunda");  
console.log(s);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// Set { "Maudy", "Ayunda" }
```

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 :

```
console.log(s.has("Maudy"));  
console.log(s.has("Husband"));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//true  
//false
```

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 :

```
console.log(s.delete("Maudy"));  
console.log(s);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//true  
//Set { "Ayunda" }
```

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 :

```
console.log(s.size);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//2
```

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 :

```
console.log(s.clear());
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//undefined
```

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 :

```
for (const k of s.values()) {  
  console.log(k);  
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*  
Maudy  
Ayunda  
*/
```

Subchapter 14 – Tuples

Software solves business problems.

—Patrick Mckenzie

Subchapter 14 – Objectives

- Memahami Apa itu **Tuple**?
 - Memahami **Property & Method** pada **Tuple**
 - Memahami Salah Satu Aplikasi dari **Tuple**
-

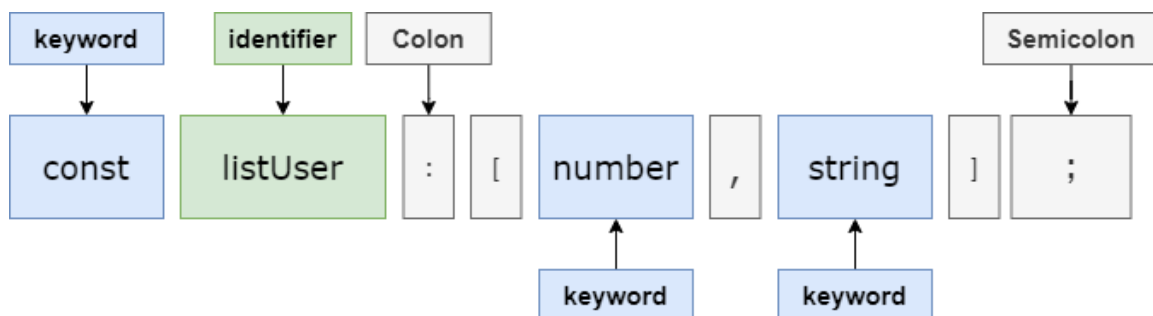
Typescript memperkenalkan tipe data baru yang disebut dengan **Tuple**. Struktur data **Tuples** adalah **array** yang dapat memiliki berbagai tipe data.

Create Tuple

Di bawah ini adalah contoh kode **Tuple** yang dapat menampung tipe data **number** dan **string** :

```
const listUser: [number, string];
```

Jika kita konversi ke dalam diagram maka terdapat struktur sebagai berikut :



Gambar 290 Tuple Structure

Add Elements

Untuk menambahkan item ke dalam tuple tulis statement di bawah ini :

```
const listUser: [number, string];  
listUser = [1, "Maudy"];  
console.log(listUser);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//[ 1, "Maudy" ]
```

Kita juga dapat menambahkan **element** berdasarkan **index** :

```
listUser[0] = 1;  
listUser[1] = "Maudy";
```

Read Elements

Untuk mengubah **elements** yang ada di dalam tuple perhatikan kode di bawah ini :

```
const listUser: [number, string] = [1, "Maudy"];  
console.log(listUser[0]);  
console.log(listUser[1]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//1  
//Maudy
```

Modify Elements

Untuk mengubah **elements** yang ada di dalam tuple perhatikan kode di bawah ini :

```
const listUser: [number, string] = [1, "Maudy"];
listUser[0] = 888;
listUser[1] = "Ayunda";
console.log(listUser[0]);
console.log(listUser[1]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//888
//Ayunda
```

Extend Tuple

Kita juga dapat membuat salah satu tipe data dalam **tuple** sebuah **references type**, pada kode di bawah ini kita membuat **tuple** dengan tipe data **number**, **string** dan **array of string** :

```
var listUsers: [number, string, string[]] = [1, "Maudy", ["Artist", "Singer"]];
listUsers[0] = 888;
listUsers[1] = "Ayunda";
console.log(listUsers[0]);
console.log(listUsers[1]);
console.log(listUsers[2]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*
888
```

```
Ayunda
[ "Artist", "Singer" ]
*/
```

Modify Extended Tuple

Kita juga dapat memodifikasi **extended tuple** yang kita buat sebelumnya, perhatikan kode di bawah ini :

```
var listUsers: [number, string, string[]] = [1, "Maudy", ["Artist", "Singer"]];
listUsers[0] = 888;
listUsers[1] = "Ayunda";
listUsers[2] = ["Young", "Single"];
console.log(listUsers[0]);
console.log(listUsers[1]);
console.log(listUsers[2]);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
/*
888
Ayunda
[ "Artist", "Singer" ]
*/
```

Array of Tuple

Kita juga dapat membuat sebuah **array of tuple** perhatikan kode di bawah ini :

```
const listUser1: [number, string][] = [[1, "Maudy"], [2, "Gun Gun "]];
console.log(listUser1);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//[ [ 1, "Maudy" ], [ 2, "Gun Gun " ] ]
```

Tuple Property & Method

Tuple Property

Sebuah ***tuple*** juga dapat memiliki sebuah ***property*** :

```
var listUser: [number, string] = [1, "Maudy"];  
console.log(listUser.length);
```

Jika kode di atas dieksekusi maka akan memproduksi ***output*** :

```
//2
```

Tuple Method

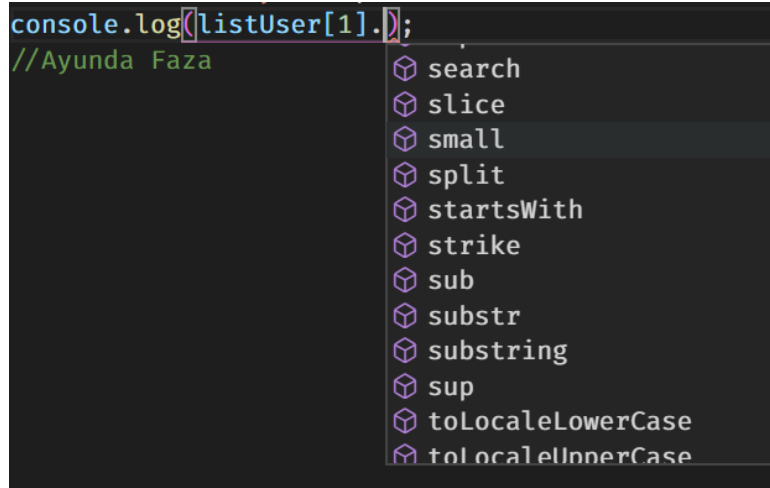
Setiap ***element*** di dalam ***tuple*** memiliki ***method*** yang sama seperti pada ***array***, kita dapat membuktikannya dengan mencoba menulis kode di bawah ini :

```
const listUser: [number, string] = [1, "Maudy"];  
listUser[0] = 888;  
listUser[1] = "Ayunda";  
console.log(listUser[1].concat(" Faza"));
```

Jika kode di atas dieksekusi maka akan memproduksi ***output*** :

```
//Ayunda Faza
```

Jika kita perhatikan pada ***intellisense*** yang disediakan ***visual studio code*** :



Gambar 291 Tuple Element Methods

Push Method

Kita juga dapat menambahkan **element** baru ke dalam **tuple** menggunakan **method** `push()`, perhatikan kode dibawah ini :

```
const listUser: [number, string] = [1, "Maudy"];
listUser[0] = 888;
listUser[1] = "Ayunda";
listUser.push(2, "Gun Gun Febrianza");
console.log(listUser);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//[888, "Ayunda", 2, "Gun Gun Febrianza"];
```

Pop Method

Kita juga dapat mencabut **element** terakhir di dalam **tuple** menggunakan **method** `pop()`, perhatikan kode dibawah ini :

```
const listUser: [number, string] = [1, "Maudy"];
listUser[0] = 888;
listUser[1] = "Ayunda";
listUser.push(2, "Gun Gun Febrianza");
listUser.pop();
console.log(listUser);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

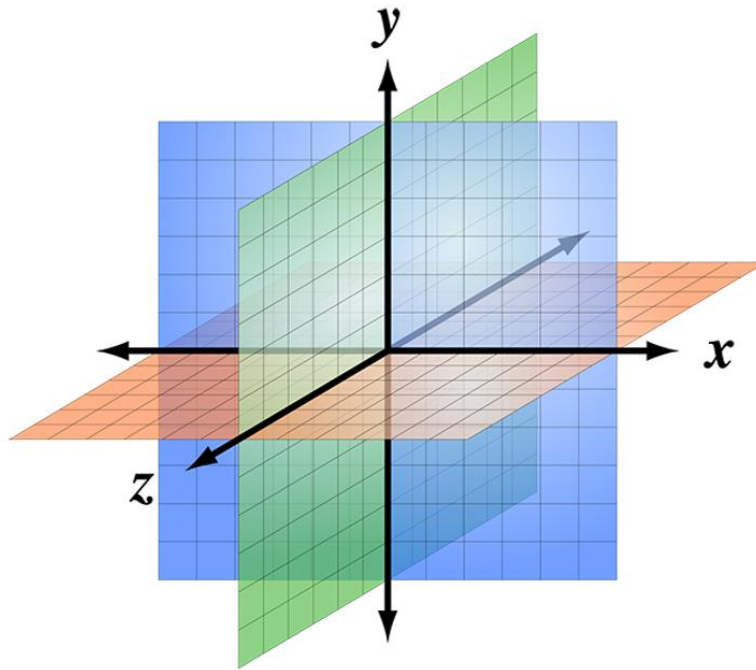
```
//[ 888, "Ayunda", 2 ]
```

Application

Pada kasus didunia nyata kita dapat menggunakan tuple sebagai sebuah data structure untuk mengeksplorasi dunia **3D Modelling**. Dalam dunia nyata pemanfaatannya dapat digunakan untuk membangun **Modelling 3D World, 3D Plotting, 3D Chart** hingga membentuk model wajah kita sendiri.

Wajah kita akan di scan melalui **3D Scanner**, rekonstruksi wajah kita dalam 3 Dimensi virtual adalah sebuah **encoding** dengan **format Stereolithography (STL)**. Dari sana kita dapat menerapkan wajah kita sendiri ke dalam pemetaan 3 dimensi.

Ada banyak **framework** seperti **OpenGL** dan **WebGL** yang dapat menampilkan format 3D **Object**. Dukungan **WebAssembly** untuk performa yang lebih baik membawa para **javascript developer** dengan **talent** seniman 3 Dimensi menjajal dunia yang menarik ini.



Gambar 292 3D Modelling

Salah satu contoh **function** untuk melakukan **plotting** 3 Dimensi :

```
function draw(...point3D: [number, number, number]): void {  
  
}
```

Kita akan mencoba membuat representasi dasar untuk melakukan **plotting** pada dunia 3 dimensi, silahkan tulis kode di bawah ini :

```
const xCoordinate: number = 0;  
const yCoordinate: number = 3;  
const zCoordinate: number = 5;  
function draw(...point3D: [number, number, number]): void {  
    console.log(point3D);  
}  
draw(xCoordinate, yCoordinate, zCoordinate);
```

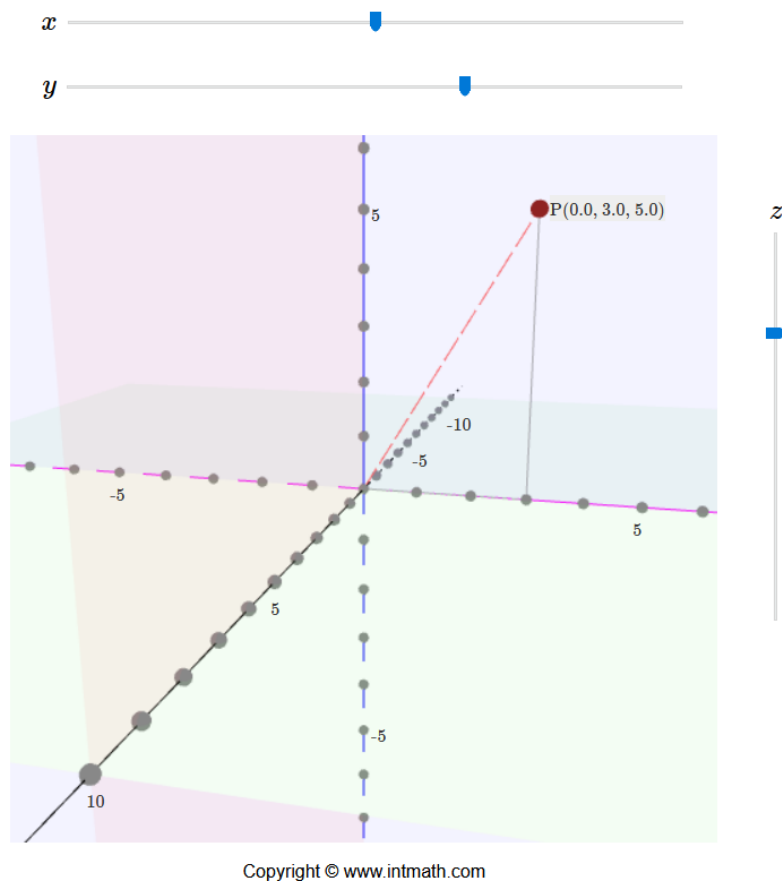
Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// [ 0, 3, 5 ]
```

Jika kita buka web di bawah ini :

<https://www.intmath.com/vectors/3d-space-interactive-applet.php>

Kita dapat melihat representasi dari **plotting** `[0, 3, 5]` yang telah kita lakukan :



Jika anda ingin melakukan **plotting object** 3 dimensi di dalam **web browser**, penulis menyarankan anda untuk mempelajari D3.js dasar kemudian bereksplorasi menggunakan **library** di bawah ini :

<https://github.com/Niekes/d3-3d>

Selamat bereksplorasi.

Subchapter 15 – Enum

Enum adalah sekumpulan **constant** yang menjadi satu unit kesatuan tunggal. Jika anda memiliki sekumpulan **variable constant** yang memiliki karakteristik sama maka lebih baik disimpan di dalam **enum**.

1. Create Enum

Di bawah ini adalah contoh kode enum :

```
enum AppStatus {  
    ACTIVE = "ACTIVE",  
    INACTIVE = "INACTIVE",  
    ONHOLD = "ONHOLD",  
}
```

Untuk membaca seluruh **constants** dalam **enum** tulis kode di bawah ini :

```
console.log(AppStatus);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// { ACTIVE: "ACTIVE", INACTIVE: "INACTIVE", ONHOLD: "ONHOLD" }
```

2. Access Enum

Untuk membaca salah satu **constant** di dalam **enum**, cukup panggil **enum member** yang ada di dalamnya. Perhatikan kode di bawah ini :

```
enum AppStatus {  
    ACTIVE = "ACTIVE",  
    INACTIVE = "INACTIVE",  
    ONHOLD = "ONHOLD",  
}
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
console.log(AppStatus.ACTIVE);  
// ACTIVE
```

3. Enum Function Argument

Pada contoh kode di bawah ini kita menggunakan **enum** sebagai **type** dalam **function argument** :

```
enum AppStatus {  
  ACTIVE = "ACTIVE",  
  INACTIVE = "INACTIVE",  
  ONHOLD = "HOLD",  
}  
  
function checkStatus(status: AppStatus): void {  
  console.log(status);  
}  
  
checkStatus(AppStatus.ACTIVE);
```

Jika kode di atas di eksekusi maka akan memproduksi :

```
//ACTIVE
```


Subchapter 16 – Generic

Dalam **software engineering**, komponen adalah sekumpulan kode dalam sebuah unit tunggal yang digunakan untuk menyelesaikan suatu pekerjaan.

Pada bahasa pemrograman tingkat tinggi seperti C# dan Java salah satu komponen yang membantu kita untuk membuat *reusable component* adalah **Generic**. Terdapat dua kelebihan **generic** yaitu :

1. **Generic programming** membuat kita dapat menulis algoritma yang dapat menerima input semua jenis tipe data tanpa melanggar konsep **type safety**.
2. **Generic programming** membuat kita dapat menulis algoritma yang dapat menerima **input** semua jenis tipe data tanpa perlu membuat algoritma khusus untuk setiap tipe data.

1. Generic Function

Kita akan mempelajari **generic**, di awali dengan studi kasus bagaimana memproses dua tipe data yang berbeda menggunakan dua buah fungsi :

```
function displayString(item: string): string {  
    console.log(item);  
    return item;  
}  
  
function displayNumber(item: number): number {  
    console.log(item);  
    return item;  
}
```

Type Union Way

Jika kita perhatikan kode di atas dapat disederhanakan menggunakan **type union** :

```
function displayItem(item: string | number): string | number {  
  console.log(item);  
  return item;  
}
```

Jika kita menggunakan fungsi di atas maka kita dapat menampilkan **input string** dan **number** namun ketika kita memberikan **input boolean** maka kompilasi akan **error** :

```
displayItem("Hello Maudy")  
displayItem(888)  
//displayItem(true) //error
```

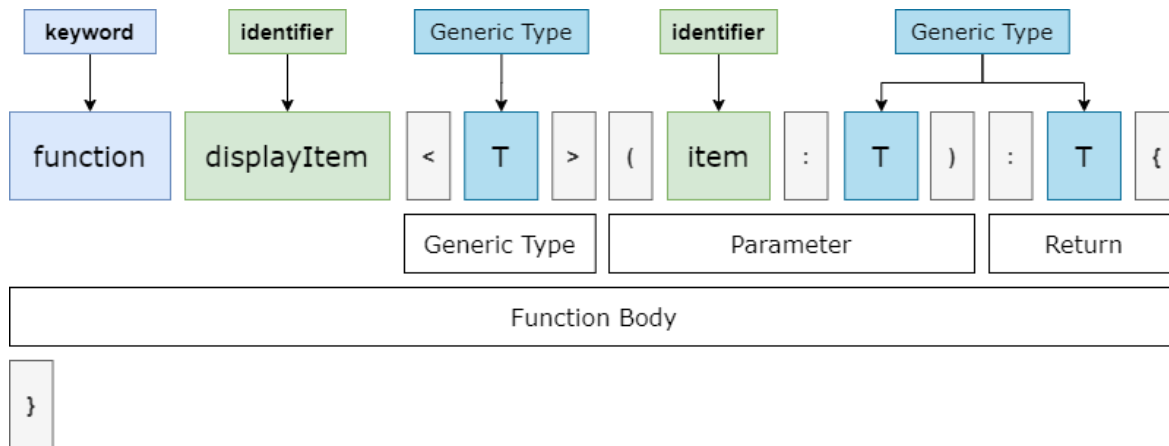
Ada cara yang lebih efektif yaitu memanfaatkan **generic** pada **function** yang ingin kita buat. Dengan begitu kita dapat menyimpan **primitive** dan **reference values** tanpa perlu menggunakan **type union**.

Generic Type

Di bawah ini adalah sebuah **Generic Function** :

```
function displayItem<T>(item: T): T {  
  console.log(item);  
  return item;  
}
```

Jika kita transformasikan **Generic Function** di atas ke dalam diagram maka :



Gambar 293 Generic Syntax Structure

Untuk menambahkan **generic** pada **function** kita memerlukan notasi `<T>`, **symbol T** disebut dengan **Generic Type Parameter**. Jika kita ingin mencoba menggunakan **generic function** yang telah kita buat sebelumnya tulis dan eksekusi kode di bawah ini :

```
displayItem("Hello Maudy");
displayItem(2);
displayItem(true);
```

Maka akan memproduksi **output** :

```
/*
Hello Maudy
2
true
*/
```

Kita juga dapat memberikan **input** berupa **reference value** seperti **array** :

```
displayItem([2, 3, 4]);
```

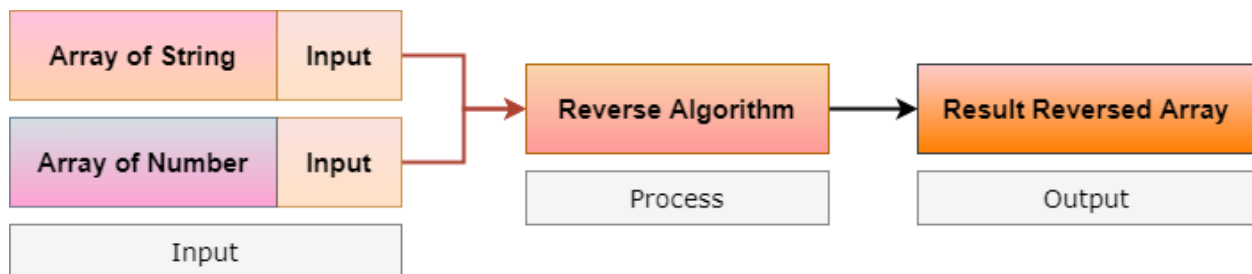
Jika kode di atas dieksekusi maka akan memproduksi keluaran :

```
// [ 2, 3, 4 ]
```

2. Reverse Array Element

Sebelumnya anda sudah mendapatkan gambaran bagaimana **generic function** bekerja. Sekarang kita akan membuat studi kasus menggunakan **generic**, yaitu bagaimana membuat sebuah algoritma yang dapat melakukan **reverse** pada sebuah **array**.

Pada gambar di bawah ini terdapat dua buah **input** yaitu **array of string** dan **array of number**, selanjutnya dibutuhkan sebuah **reverse algorithm** yang dapat melakukan **reverse** pada setiap **array** dan menampilkan hasil **reverse** sebagai **output**.



Gambar 294 Reverse Algorithm

Ada beberapa step yang harus kita lalui :

1. Membuat **Generic Function**.
2. Membuat **Parameter** untuk **Generic Function** yang dapat menerima **Input Array**.
3. Membuat **Return** untuk **Generic Function** yang dapat memproduksi **Output Array**.
4. Membuat sebuah **Array** untuk menyimpan hasil yang akan di **reverse**.
5. Menghitung panjang **Array** yang menjadi **input** untuk di **reverse**.
6. Melakukan perulangan dengan **index** terbalik pada **array** yang akan di **reverse**.
7. Simpan **array** yang telah di **reverse** dengan **method push**.
8. **Return** seluruh **array elements** yang telah di **reverse**.

Create Generic Function

Buat **function** menggunakan notasi **generic type** :

```
function reverseList<T>() {  
  
}
```

Create Parameter For Generic Function

Selanjutnya kita akan membuat **parameter** berupa **generic array type** :

```
function reverseList<T>(list: T[]) {  
  
}
```

Create Return For Generic Function

Tambahkan **return generic array** :

```
function reverseList<T>(list: T[]): T[] {  
  
}
```

Create Internal Array Data Structure

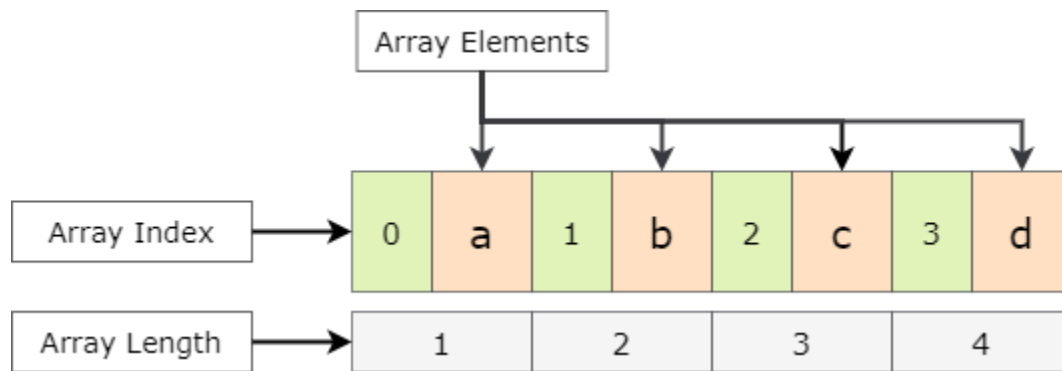
Selanjutnya kita akan membuat variabel dengan tipe data **generic array** untuk menyimpan hasil yang akan di **reverse** :

```
function reverseList<T>(list: T[]): T[] {
```

```
const reversedList: T[] = [];  
}
```

Create Internal Algorithm

Di bawah ini adalah salah satu visualisasi dari input array string yang akan kita olah dalam algoritma yang kita buat.

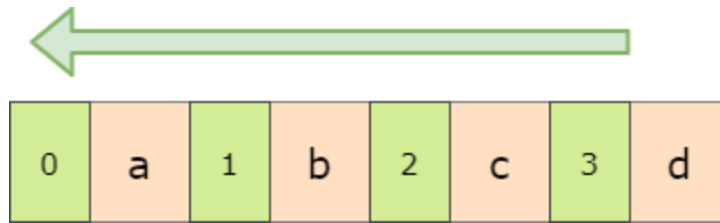


Gambar 295 Array Structure

Pada kode di bawah ini kita membuat variabel `i` yang merupakan hasil komputasi dari `list.length` dikurangi 1. Pada gambar di atas **array length** dari input yang diberikan adalah 4, jika dikurangi 1 maka hasilnya adalah 3.

```
function reverseList<T>(list: T[]): T[] {  
  const reversedList: T[] = [];  
  let i = (list.length - 1);  
}
```

Selanjutnya kita akan melakukan perulangan pada array **index** ke 3, sehingga **looping** harus menggunakan notasi `--` agar perulangannya mundur kebelakang.



Gambar 296 Decrement Looping Illustration

Proses **looping** menggunakan notasi **decrement** dilakukan dimulai dari **index** ke 3 sampai **index** ke 0 sehingga kita bisa mendapatkan setiap **Array Element** secara **reverse** yaitu **d, c, b, a**.

Setiap kali kita melakukan **looping** pada **index array**, kita simpan **array element** ke dalam variabel **array** baru yaitu **reversedList**. Perhatikan kode di bawah ini :

```
function reverseList<T>(list: T[]): T[] {  
  const reversedList: T[] = [];  
  let i = (list.length - 1);  
  for (i; i >= 0; i--) {  
    reversedList.push(list[i]);  
  }  
}
```

Create Return Inside Generic Function

Selanjutnya **return** hasil yang di proses :

```
function reverseList<T>(list: T[]): T[] {  
  const reversedList: T[] = [];  
  let i = (list.length - 1);  
  for (i; i >= 0; i--) {  
    reversedList.push(list[i]);  
  }  
}
```



```
    return reversedList;
}
```

Untuk menggunakannya kita akan mengujinya dengan ***array of string*** :

```
const letters = ["a", "b", "c", "d"];
const reversedLetters = reverseList<string>(letters);
console.log(reversedLetters);
```

Jika di eksekusi kode di atas maka akan memproduksi :

```
/*
d
c
b
a
*/
```

Selanjutnya kita akan mengujinya menggunakan ***array of number*** :

```
const numbers = [1, 2, 3, 4];
const reversedNumbers = reverseList<number>(numbers);
console.log(reversedNumbers);
```

Jika di eksekusi kode di atas maka akan memproduksi :

```
/*
4
3
2
1
*/
```



Subchapter 17 – Module

Module adalah istilah yang muncul dalam paradigma **modular programming**. Tujuan dari **modular programming** agar kita bisa membangun sebuah program skala besar yang tersusun dari sekumpulan **module**.

Bahasa pemrograman memiliki **modularity** yang baik jika setiap program yang ingin dibuat dalam skala yang besar dapat dibangun dengan cepat, melalui sekumpulan module yang tersedia.

1. Module Concept

Javascript adalah bahasa yang sejak dari awal tidak mendukung konsep modul, sebelumnya hal ini tidak menjadi masalah karena **javacript** digunakan untuk membangun **script** sederhana untuk tugas-tugas kecil.

Namun meningkatnya penggunaan **javascript** diberbagai lini membuat **javascript** menjadi semakin kompleks dan **bloats**. Sehingga konsep **module** diperlukan untuk mempermudah pengembangan **application** menggunakan **javascript**.

Ketika **javascript application** yang kita buat semakin besar kita perlu memecahnya menjadi beberapa bagian berupa sekumpulan **file** yang akan kita sebut dengan **module**.

Module

Module adalah sebuah *file javascript* atau *typescript* tunggal yang merepresentasikan suatu fungsionalitas. *Modules is a reusable piece of code that encapsulates implementation details.*

Module sederhananya adalah sekumpulan kode yang dapat kita gunakan lagi dan lagi untuk mempercepat pengembangan aplikasi. Sebuah *Deno/Node.js application*

dibangun oleh sekumpulan *module*, sehingga *module* adalah *basic block* dari sebuah *Deno/Node.js application*.

Sebuah *modules* ditulis oleh :

1. *Deno/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 *Deno/Node.js application* agar dapat digunakan. Sebuah *module* harus memberikan kita :

Abstraction

Kita dapat mendelegasikan sebuah fungsionalitas pada sekumpulan *module* yang diciptakan khusus untuk permasalahan tertentu, tanpa harus memahami kompleksitas cara untuk implementasinya secara aktual.

Encapsulation

Untuk membungkus dan mengklasifikasikan sekumpulan kode (*block of code*) atau sekumpulan *statements* tunggal dalam sebuah *module*.

Reuse Code

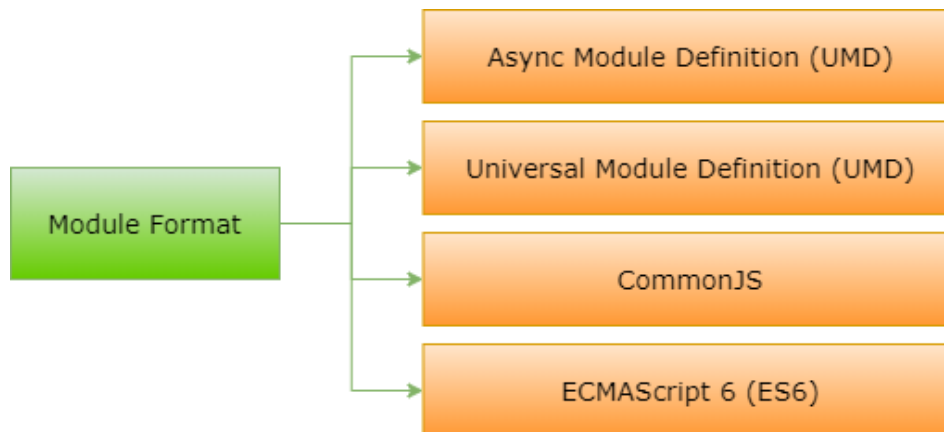
Untuk mencegah penulisan kode yang sama lagi dan lagi.

Manage Dependencies

Memudahkan kita untuk manajemen *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 :



Gambar 297 Module Format

CommonJS (CommonJS)

Format ini digunakan di dalam *Node.js* untuk penggunaan **Server Side**, cirinya penggunaan *keyword* `require` untuk melakukan **import module** dan **export** suatu **module** menggunakan `module.exports` untuk menentukan sebuah **dependencies** dan **modules**.

CommonJS Tree Shaking

Pada **CommonJS** tidak tersedia **Tree Shaking**, artinya saat kita melakukan **import** beberapa **code** yang tidak digunakan juga ikut termuat. **Code** yang tidak digunakan di dalam **module** disebut dengan **dead code**.

Tree Shaking adalah istilah untuk mencabut kode-kode yang tidak digunakan di dalam **module**.

CommonJS Static Analysis

Pada **CommonJS** tidak tersedia **static analysis** disebabkan **object** didapatkan saat **runtime**.

Async Module Definition (AMD)

Format ini digunakan di dalam **browser** untuk keperluan **client-side**, format ini digunakan oleh sebuah **module loader** bernama **RequireJS**.

Universal Module Definition (UMD)

Format ini adalah kombinasi dari **Async Module Definition (AMD)** dan **CommonJS** sehingga dapat digunakan di dalam **browser** untuk **client-side** dan **node.js** untuk **server-side**.

ECMAScript (ES6)

Format ini dapat digunakan untuk keperluan server-side atau client-side. Cirinya penggunaan keyword **import** untuk melakukan **import module** dan keyword **export** melakukan **export module**.

ES6 Static Analysis

Kita dapat melakukan **static analysis** jika format menggunakan ES6.

ES 6 Tree Shaking

Terdapat dukungan **tree shakable** karena tersedianya **static analysis**.

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)*.

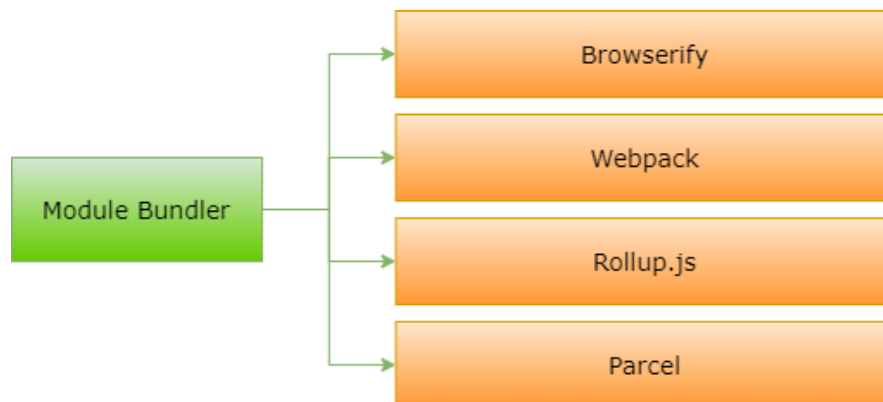
SystemJs

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* agar dapat dieksekusi di dalam **browser**.



Gambar 298 Module Bundler

Terdapat dua *module bundlers* yang populer :

Browserify

Module bundler untuk *CommonJS Modules*

Webpack

Module bundler untuk *AMD, CommonJS, & ES 6 Module*.

Parcel

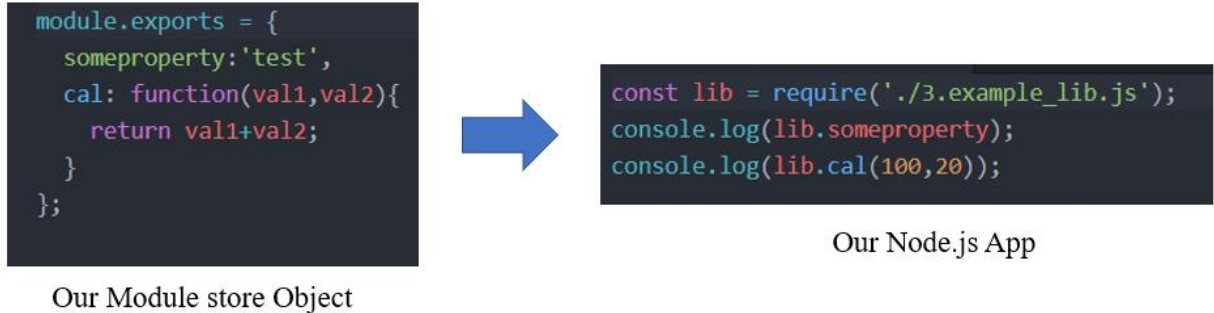
Module bundler untuk *CommonJS, & ES 6 Module*.

Rollup.js

Module bundler untuk *ES 6 Module*.

2. Node.js Module

Perhatikan pada gambar di bawah ini :



Gambar 299 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* yang dapat kita lihat disini :

<https://nodejs.org/api/>

Seluruh informasi seperti dokumentasi kode untuk setiap *modules* telah disediakan disana.



Gambar 300 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_writing.js',
  '3.ensureDir.js',
  '4.remove.js',
  '5.move.js',
  'global.html',
  'node_modules',
]
```

Gambar 301 Output readdir method

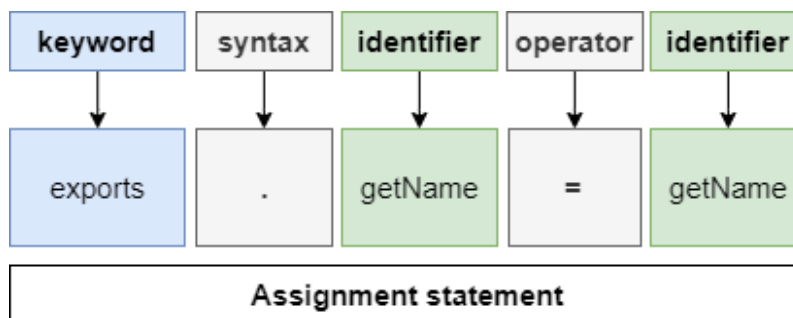
Create & Export Module

Kita akan membuat sebuah *module* dengan nama ***simple-module.js*** :

```
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.



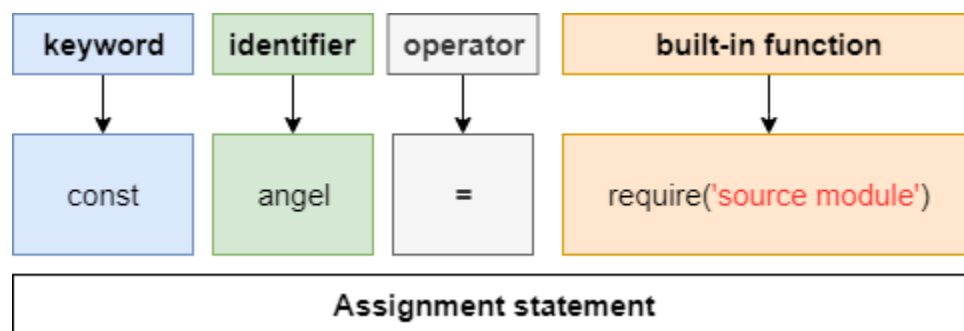
Gambar 302 Syntax Rule Export Module

Use Module

Kita akan membuat aplikasi *node.js* yang akan menggunakan **simple-module**, buatlah *file main.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 303 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* *class-module.js*. Selanjutnya kita bisa memanggil *method* **topup()** dan menggunakan *property* **balance**.

3. Deno Module

Sebelumnya kita telah belajar bagaimana cara membuat **module commonJS** menggunakan **javascript** untuk **Node.js**, sekarang kita akan membuat **module CommonJS** menggunakan **typescript**.

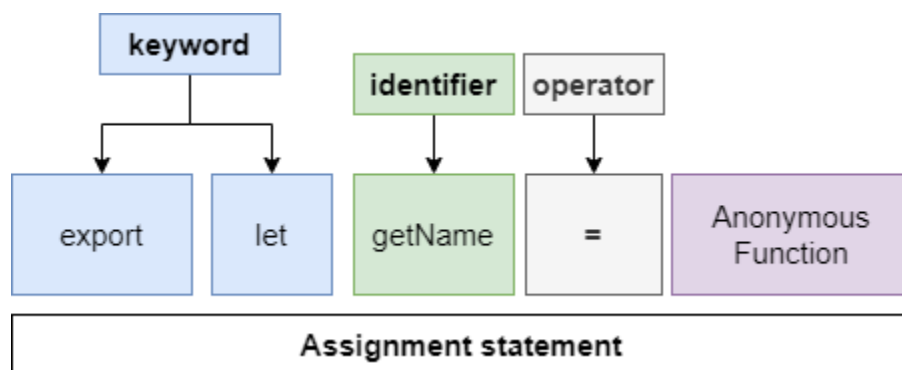
Create & Export Module

Buatlah sebuah **folder** dengan nama **module-typescript** kemudian kita buat sebuah **module** dengan nama **simple-module.ts** :

```
export let getName = (): string => {  
    return "My Name is Maudy";  
};
```

Dalam **module** di atas kita hanya memiliki satu **variable** yang menampung sebuah **object** berupa **anonymous function**.

Kita akan menggunakan **keyword** **export** untuk menegaskan bahwa **getName** adalah **object** yang akan di **export** agar bisa digunakan oleh aplikasi **Node.js** lainnya.



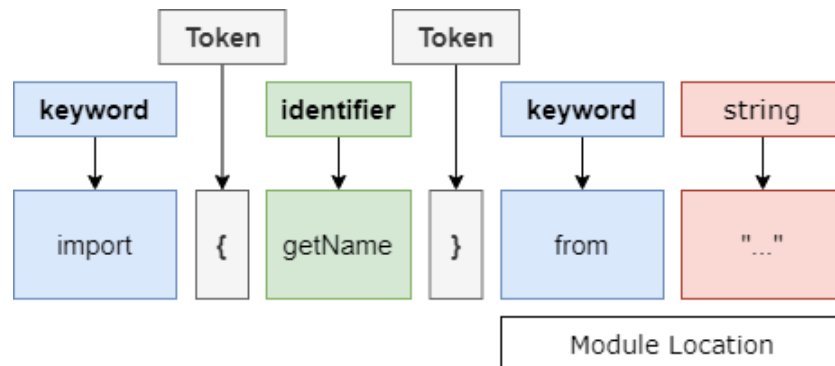
Gambar 304 Module Export Deno

Use Module

Kita akan membuat aplikasi *deno* yang akan menggunakan **simple-module**, buatlah file **main.ts** dan tulis kode di bawah ini :

```
import { getName } from "./simple-module";  
console.log(getName());
```

Pada kode di atas *destructure* digunakan agar *module* dapat digunakan dan object langsung dapat diakses. Kita akan menggunakan keyword **import** untuk menentukan dimana lokasi *module* disimpan.



Gambar 305 Module Import Deno

Selanjutnya kita dapat menggunakan object **getName** :

```
console.log(getName());  
// My Name is Maudy Ayunda
```

*Link sumber kode.

Compile to Javascript

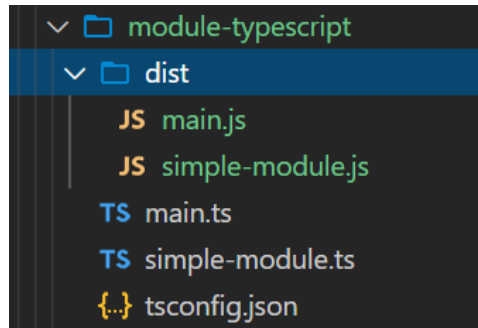
Buatlah sebuah **tsconfig.ts** dengan **preconfiguration** sebagai berikut :

```
{
  "compilerOptions": {
    "target": "ES2019",
    "module": "commonjs",
    "strict": true,
    "outDir": "dist",
    "noEmitOnError": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Pada pengaturan di atas kita menggunakan **option module commonjs** agar dapat digunakan oleh **Node.js Runtime**. Selanjutnya eksekusi perintah dibawah ini untuk melakukan kompilasi :

```
> tsc
```

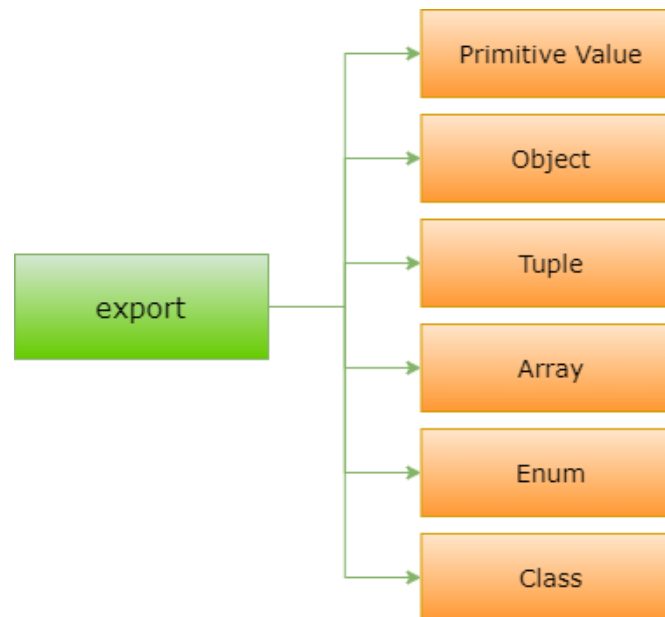
Jika berhasil akan muncul sebuah **folder** baru dengan nama **dist**, di dalamnya terdapat dua **file javascript** :



Gambar 306 Compiled Javascript

Export Visualization

Kita akan membuat sebuah **example**, membuat **module** yang terdiri dari **primitive value** dan **reference value** untuk di **export** dengan **format CommonJS**.



Gambar 307 Export Visualization Example

Buatlah sebuah **folder** dengan nama **module-deno** kemudian di dalamnya buat sebuah **file** dengan nama **mod.ts**.

Selanjutnya buatlah sebuah **variable** yang menyimpan sebuah **primitive value** :

```
export let bookTitle: string = "Belajar Dengan Jenius Typescript";
```

Selanjutnya kita akan membuat sebuah **object** yang akan kita **export** :

```
export let bookDetail: {  
  author: string;  
  price: number;  
} = { author: "Gun Gun Febrianza", price: 90000 };
```

Selanjutnya kita akan membuat sebuah **object tuple** yang akan kita **export** :

```
export const publication: [string, number] = ["PinterCoding", 2020];  
//Tuple
```

Selanjutnya kita akan membuat sebuah **object array** yang akan kita **export** :

```
export const keywords: string[] = ["javascript", "typescript", "deno"];  
//Array
```

Selanjutnya kita akan membuat sebuah **enum** yang akan kita **export** :

```
export enum Contributor {  
  M = "Maudy",  
  N = "Nikolaj",  
  Y = "Yuma",  
}
```

Selanjutnya kita akan membuat sebuah **object function** yang akan kita **export** :

```
export function readBook(): void {  
    console.log(`${bookTitle} karya ${bookDetail.author}`);  
}
```

Selanjutnya kita akan membuat sebuah **class** yang akan kita **export** :

```
export class Wallet {  
    constructor(public name: string, public balance: number) {  
        this.name = name;  
        this.balance = balance;  
    }  
    topUp(newbalance: number) {  
        return (this.balance += newbalance);  
    }  
}
```

Import Module

Untuk menggunakan **module** di atas :

```
import { readBook, bookDetail, publication } from "./mod";  
readBook();  
// Belajar Dengan Jenius Typescript karya Gun Gun Febrianza  
console.log(bookDetail);  
// { author: 'Gun Gun Febrianza', price: 90000 }  
console.log(publication);  
// [ 'PinterCoding', 2020 ]
```

Subchapter 18 – Namespace

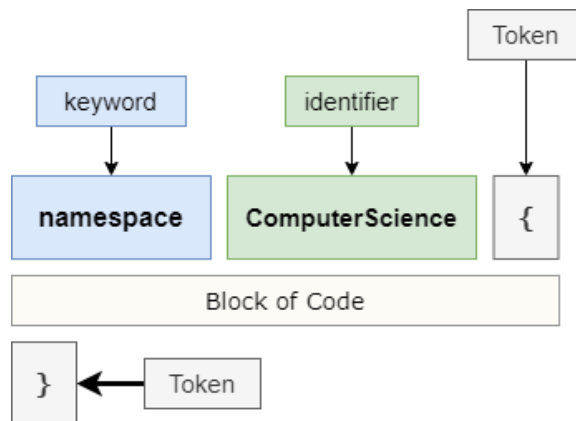
Pada **javascript** kita dapat melakukan manajemen kode agar memiliki **modularity** yang baik menggunakan konsep **module**, pada **typescript** diperkenalkan konsep **namespace** yang memiliki fungsi sama persis dan gaya penulisan **modular programming** terbaru.

1. Create Namespace

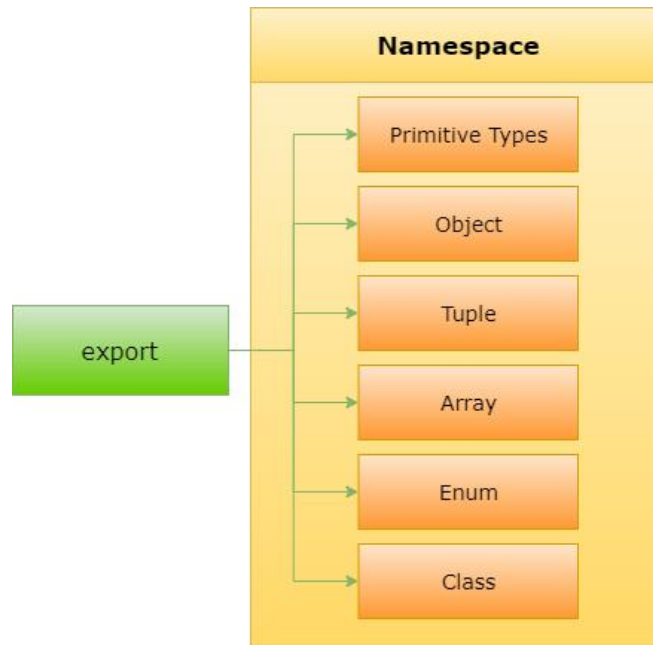
Di bawah ini adalah contoh **namespace** dalam **typescript** :

```
namespace ComputerScience {  
    export let course: string = "Computer Science";  
    export interface DataStructure {  
    }  
    export class ArtificialIntelligence {  
    }  
}
```

Jika pada kode di atas kita transformasikan ke dalam diagram maka strukturnya :



Gambar 308 Namespace Structure



Gambar 309 Namespace Illustration

Kita dapat membungkus **primitive types** dan **reference types** ke dalam sebuah **namespace**, gambar di atas adalah sebuah ilustrasi dimana kita dapat memanajemen sekumpulan **object**, **array**, **class** dan sebagainya di dalam sebuah **namespace**.

Nested Namespace

Selain membungkus **primitive types** dan **reference types** kita juga dapat membungkus sebuah **namespace** di dalam **namespace**. Sehingga seringkali disebut dengan **nested namespace** atau **namespace** bersarang.

```
namespace ComputerScience {  
    namespace DataStructure {  
        namespace Array {  
  
        }  
    }  
}
```

```
}
```

Namespace Reference

Ada saatnya ***namespace*** yang kita tulis jumlahnya mencapai ribuan baris, anda ingin membaginya ke dalam beberapa ***file*** berbeda namun masih dalam satu ***namespace*** yang sama maka kita dapat menggunakan ***reference path***.

Pada kode di bawah ini, terdapat ***statement reference*** beserta ***path file*** pada baris pertama, menegaskan bahwa ***namespace*** dalam ***file*** yang sedang kita tulis adalah bagian dari ***namespace*** yang ada di dalam suatu ***file*** :

```
///
```


2. Application – Stack Data Structure

Pada kasus kali ini kita akan mempelajari cara penggunaan **namespace** untuk membuat sebuah **code base** yang menjadi **framework** tempat sekumpulan **data structure**. Kita akan memulai dengan membuat sebuah **namespace** yang di dalamnya terdapat **interface Stack Data Structure**.

Stack Data Structure

Stack adalah sebuah **data structure** yang mengimplementasikan sifat **LIFO (Last In First Out)**, pada sebuah **stack** wajib terdapat dua operasi yang bisa dilakukan yaitu **push** and **pop**.

Push digunakan untuk menyimpan suatu elemen ke dalam **stack** dan **pop** digunakan untuk membuang elemen yang ada di dalam **stack**.

Create Interface Data Structure

Pertama kita akan membuat sebuah **Interface Data Structure** di dalam sebuah **namespace**, tulis kode dibawah ini ke dalam suatu **file** dengan nama **LibDataStructure.ts** :

```
namespace DataStructure {  
  export let library: string = "Data Structure Library";  
  export interface StackDataStructure {  
    push(input: string): void;  
    pop(): void;  
    peek(): void;  
    isEmpty():boolean;  
  }  
}
```

Pada kode di atas kita membuat sebuah **interface** untuk merepresentasikan data **structure** yang akan kita buat. Terdapat sekumpulan **methods** yang wajib diterapkan oleh **data structure**.

Create Class Stack Data Structure

Selanjutnya buat lagi **file** dengan nama **ClassDataStructure.ts**, tulis kode di bawah ini :

```
///<reference path = "LibDataStructure.ts" />
namespace DataStructure {
    export class Stack implements StackDataStructure {
        private _stack: string[];
        constructor(s: string[] = [""]) {
            this._stack = s;
        }
        push(input: string): void {
            this._stack.push(input);
        }
        pop(): void {
            this._stack.pop();
        }
        isEmpty(): boolean {
            return this._stack.length == 0;
        }
        peek(): string {
            return !this.isEmpty()
                ? this._stack[this._stack.length - 1]
                : "Empty Stack";
        }
    }
}
```

Pada baris pertama kode di atas kita membangun **reference** bahwa **file** yang kita tulis masih satu **namespace** dengan **namespace** yang ada pada **file LibDataStructure.ts**. Selanjutnya kita membuat sebuah **class** yang menerapkan **interface** dari **StackDataStructure**.

Create Stack Object

Kita memanfaatkan dua **file** sebelumnya yaitu :

1. **Interface Data Structure** pada **file LibDataStructure.ts**
2. **Class Stack Data Structure** pada **file ClassDataStructure.ts**

Buat **file** baru dengan nama **AppExample.ts**, tulis **references path** kemudian buat **object** seperti pada kode di bawah ini :

```
///<reference path = "LibDataStructure.ts" />
///<reference path = "ClassDataStructure.ts" />
const ObjStack = new DataStructure.Stack(["Maudy", "Ayunda"]);
```

Pada kode di atas kita membuat **object** sekaligus melakukan inisialisasi dua buah **elements** ke dalam **stack**.

Access Peek Method

Akses **peek()** **method** untuk melihat **element** terakhir teratas dalam **stack** :

```
console.log(ObjStack.peek());
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//Ayunda
```

Access Push Method

Selanjutnya kita akan mengeksekusi **method** `push()` tulis kode di bawah ini :

```
ObjStack.push("Gun Gun");  
console.log(ObjStack);
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// StackDataStructure { _stack: [ "Maudy", "Ayunda", "Gun Gun" ] }
```

Access Pop Method

Selanjutnya kita akan mengeksekusi **method** `pop()` tulis kode di bawah ini :

```
ObjStack.pop();  
console.log(ObjStack);
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// StackDataStructure { _stack: [ "Maudy", "Ayunda" ] }
```

3. Application – Stack Data Structure

Pada kasus kali ini kita akan mempelajari cara penggunaan **namespace** untuk membuat sebuah **code base** yang menjadi **framework** tempat sekumpulan **data structure**. Kita akan memulai dengan membuat sebuah **namespace** yang di dalamnya terdapat **interface Queue Data Structure**.

Queue Data Structure

Queue adalah sebuah **data structure** yang mengimplementasikan sifat **FIFO (First In First Out)**, terdapat operasi **enqueue** untuk mengisi nilai diposisi akhir **queue** dan **dequeue** untuk mendapatkan nilai yang berada di awal **queue** dan mencabutnya.

Create Interface Data Structure

Masih dengan **file LibDataStructure.ts**, kita akan membuat sebuah **Interface Queue Data Structure** :

```
namespace DataStructure {  
  export let library: string = "Data Structure Library";  
  export interface QueueDataStructure {  
    enqueue(input: string): void;  
    dequeue(): void;  
    peek(): void;  
    isEmpty(): boolean;  
  }  
}
```

Pada kode di atas kita membuat sebuah **interface** untuk merepresentasikan data **structure** yang akan kita buat. Terdapat sekumpulan **methods** yang wajib diterapkan oleh **data structure**.

Create Class Queue Data Structure

Selanjutnya pada **file ClassDataStructure.ts**, tambahkan kode di bawah ini :

```
///<reference path = "LibDataStructure.ts" />
namespace DataStructure {
    ...
    export class Queue implements QueueDataStructure {
        private _queue: string[];
        constructor(s: string[] = [""]) {
            this._queue = s;
        }
        enqueue(input: string): void {
            this._queue.push(input);
        }
        dequeue(): void {
            this._queue.shift();
        }
        isEmpty(): boolean {
            return this._queue.length == 0;
        }
        peek(): string {
            return !this.isEmpty() ? this._queue[0] : "Empty Queue";
        }
    }
}
```

Pada kode di atas kita membuat sebuah **class** yang menerapkan **interface** dari **QueueDataStructure**.

Create Queue Object

Kita memanfaatkan dua **file** sebelumnya yaitu :

1. **Interface Data Structure** pada **file LibDataStructure.ts**
2. **Class Stack Data Structure** pada **file ClassDataStructure.ts**

Pada **file AppExample.ts**, tulis buat **object** seperti pada kode di bawah ini :

```
const ObjQueue = new DataStructure.Stack(["Gun Gun", "Febrianza"]);
```

Pada kode di atas kita membuat **object** sekaligus melakukan inisialisasi dua buah **elements** ke dalam **stack**.

Access Peek Method

Akses **peek()** **method** untuk melihat **element** terakhir teratas dalam **queue** :

```
console.log(ObjQueue.peek());
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//Ayunda
```

Access Enqueue Method

Selanjutnya kita akan mengeksekusi **method enqueue()** tulis kode di bawah ini :

```
ObjQueue.enqueue("Maudy");  
console.log(ObjQueue);
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// StackDataStructure { _queue: [ "Gun Gun", "Febrianza", "Maudy" ] }
```

Access Dequeue Method

Selanjutnya kita akan mengeksekusi **method dequeue()** tulis kode di bawah ini :

```
ObjQueue.dequeue();  
console.log(ObjQueue);
```

Selanjutnya jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// StackDataStructure { _queue: [ "Febrianza", "Maudy" ] }
```


Subchapter 19 – Web API

Deno mendukung **Web API** yang ada di **browser** juga salah satunya adalah **Performance API**. **Web API** tersebut mendukung **High Precision Timing**. Analisis **performance metric** menjadi lebih akurat sampai ke **millisecond**.

Contoh kode di bawah ini pengujian **insert node** pada **Binary Search Tree (BST) Data Structure**.

```
const binarySearchTree: NodeObject = new NodeObject();
const t0 = performance.now(); // <-- Performance API Snapshot
binarySearchTree.insert(30);
binarySearchTree.insert(37);
binarySearchTree.insert(27);
binarySearchTree.insert(37);
binarySearchTree.insert(26);
binarySearchTree.insert(67);
binarySearchTree.insert(47);
binarySearchTree.insert(23);
binarySearchTree.insert(10);
const t1 = performance.now(); // <-- Performance API Snapshot
console.log(binarySearchTree);
console.log(`Took ${t1 - t0} milliseconds.`); // <-- Calculate
```

Jika kode di atas dieksekusi hasilnya selalu 0 **Millisecond**, kenapa? Artinya eksekusi selesai di level **ticks**.

- 1 **Tick** = 100 **Nanoseconds**
- 10,000 **ticks** = 1 **Millisecond**

Chapter 4

Deno Standard Module

Subchapter 1 – Filesystem Programming

Software solves business problems.

—Patrick Mckenzie

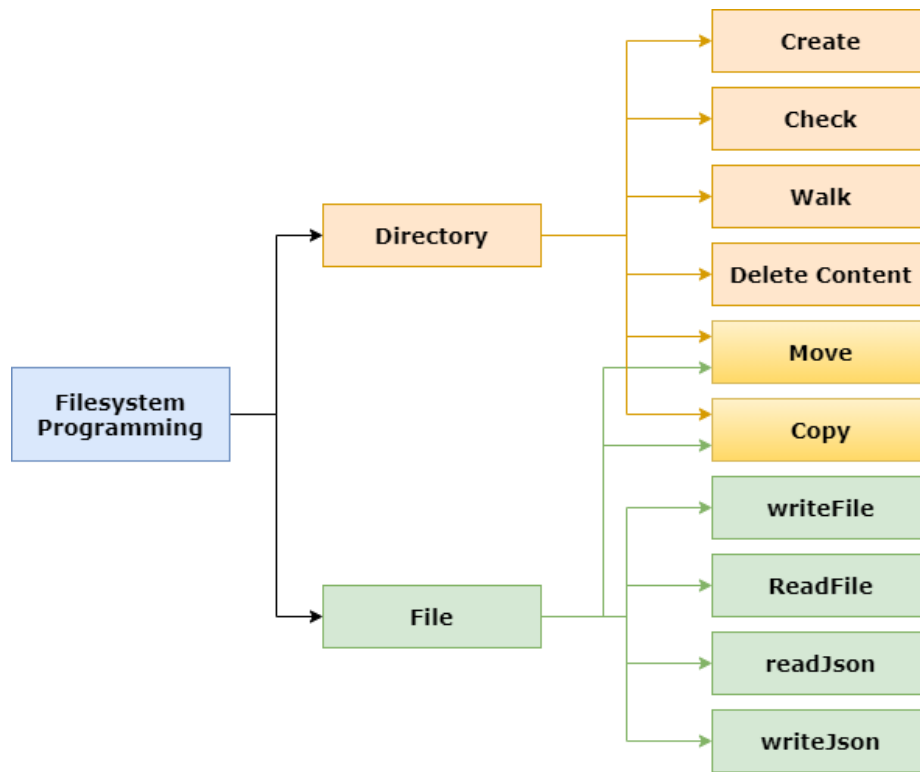
Subchapter 1 – Objectives

- Memahami Konsep **File System Programming**
 - Memahami Cara membuat **Directory**
 - Memahami Cara memeriksa **Content Directory**
 - Memahami Cara menghapus **Content Directory**
 - Memahami Cara memindahkan & menyalin **Directory**
 - Memahami Cara membuat **File**
 - Memahami Cara membaca **File**
 - Memahami Cara memindahkan & menyalin **File**
-

Sekarang kita akan mulai melakukan eksplorasi menggunakan **standard module fs**. Pada **filesystem programming** kita akan berinteraksi dengan **file & directory**, ada beberapa kunci yang wajib kita fahami.

Diagram di bawah ini area kunci untuk eksplorasi jika ingin memahami **filesystem programming** dalam **deno** :

1. File & Directory



Gambar 310 Filesystem Programming

Create Directory

Untuk membuat sebuah **directory** baru, **deno** menyediakan dua metode secara **asynchronous** atau **synchronous**, pada buku ini kita akan mempelajari menggunakan metode **asynchronous**.

Buatlah sebuah file dengan nama **ensureDir.ts** kemudian tulis kode bawah ini, kita akan melakukan **import object ensureDir** pada **module fs** :

```
import { ensureDir } from "https://deno.land/std/fs/mod.ts";
```

ensureDir adalah sebuah **function** yang membutuhkan **parameter string** untuk membuat **directory**, di bawah ini terdapat sebuah **statement** untuk membuat **directory** dengan nama **logs** :

```
ensureDir("./logs");
```

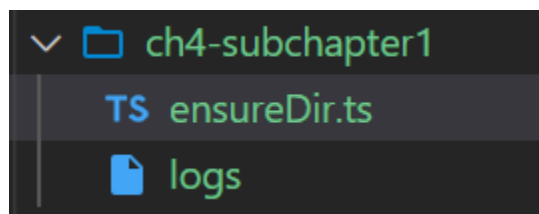
Jika kita ingin membuat **nested directories** tambahkan slash **/** seperti berikut :

```
ensureDir("./logs/security");
```

Dikarenakan kita menggunakan metode **async** maka kita memiliki **return** berupa **promise** yang dapat kita program jika hasilnya berhasil atau gagal :

```
ensureDir("./logs").then(  
  () => console.log("Success Created"),  
).catch((err) => console.log(`Error ${err}`));
```

Jika kita membuat sebuah **file** tanpa ekstensi dengan nama **logs**, kemudian mengeksekusi kode di atas maka hasilnya akan **error** :



Gambar 311 LogsFile

Di bawah ini adalah **error** yang di produksi :

```
Error: Ensure path exists, expected 'dir', got 'file'  
  
at ensureDir (https://deno.land/std/fs/ensure_dir.ts:14:13)
```

Jika tidak terdapat **file** dengan nama yang sama dengan **directory** maka eksekusi akan berhasil :

```
// Success Created
```

Untuk mengeksekusi kode gunakan perintah sebagai berikut :

```
> deno run --unstable --allow-read --allow-write ensureDir.ts
```

Module ini masih **unstable** dan memerlukan dua **flag permissions**.

Check Directory

Untuk memeriksa eksistensi suatu **directory** perhatikan kode di bawah ini :

```
import { exists } from "https://deno.land/std/fs/mod.ts";

exists("./logs").then((result) => console.log(result));
exists("./foo").then((result) => console.log(result));
```

Pada kode di atas kita menggunakan **exists object** yang merepresentasikan sebuah **function**. Jika kode di atas di eksekusi maka akan memproduksi **output** :

```
/*
true
false
*/
```

Delete Directory Content

Jika kita ingin menghapus **content** dari suatu **directory** maka kita dapat menggunakan fungsi **emptyDir**. Fungsi **emptyDir** juga memiliki karakteristik untuk membuat **directory** secara otomatis jika **directory** yang ingin dihapus **content** di dalamnya tidak tersedia.

Untuk menggunakannya silahkan eksekusi kode di bawah ini :

```
import { emptyDir } from "https://deno.land/std/fs/mod.ts";

emptyDir("./logs");
```

Move File & Directory

Jika kita ingin memindahkan suatu **file** atau **directory** kita perlu menggunakan fungsi **move**, di bawah ini adalah contoh kode untuk memindahkan **file a.txt** dalam **current directory** ke dalam **directory** bernama **folder** dengan nama file baru **b.txt** :

```
import { move } from "https://deno.land/std/fs/mod.ts";

const res = move("./a.txt", "./folder/b.txt");
res.then(() => {
  console.log("Moved");
}).catch((err) => {
  console.log(err);
});
```

Jika di dalam **folder** tersebut sudah ada **file b.txt** maka kita dapat menggunakan konfigurasi **overwrite: true** seperti pada **statement** kode di bawah ini :

```
move("./a.txt", "./folder/b.txt", { overwrite: true });
```

Jika kita ingin memindahkan suatu **directory** ke dalam **directory** lainnya, **parameter** yang digunakan tidak memiliki ekstensi. Pada contoh kode di bawah ini kita memindahkan **directory x** ke dalam **directory folder** :

```
const res = move("./x", "./folder/x");
res.then(() => {
  console.log("Moved");
}).catch((err) => {
  console.log(err);
});
```

Copy File & Directory

Jika kita ingin menyalin suatu **file** atau **directory** kita perlu menggunakan fungsi **copy**, di bawah ini adalah contoh kode untuk menyalin **file a.txt** dalam **current directory** ke dalam **directory** bernama **folder** dengan nama file baru **b.txt** :

```
import { copy } from "https://deno.land/std/fs/mod.ts";

const res = copy("./a.txt", "./folder/b.txt");
res.then(() => {
  console.log("Copied");
}).catch((err) => {
  console.log(err);
});
```

Jika di dalam **folder** tersebut sudah ada **file b.txt** maka kita dapat menggunakan konfigurasi **overwrite: true** seperti pada **statement** kode di bawah ini :

```
copy("./a.txt", "./folder/b.txt", { overwrite: true });
```

Jika kita ingin menyalin suatu **directory** ke dalam **directory** lainnya, **parameter** yang digunakan tidak memiliki ekstensi. Pada contoh kode di bawah ini kita menyalin **directory** **x** ke dalam **directory** **folder** :

```
const res = copy("./x", "./folder/x", { overwrite: true });
res.then(() => {
  console.log("Copied");
}).catch((err) => {
  console.log(err);
});
```

WriteJson

Jika kita ingin membuat **file json**, **deno** sudah menyediakan fungsi yang dapat kita gunakan yaitu **writeJson**. Pada kode di bawah ini kita membuat sebuah **file** bernama **user.json** :

```
import { writeJson } from "https://deno.land/std/fs/mod.ts";

const user = writeJson("./user.json", { name: "Maudy" }, { spaces: 2 });
user.then(() => console.log("Done"));
```

Pada **parameter** **writeJson** fungsi terdapat lokasi kita akan menulis **file json**, **parameter** kedua **content** dari **json**, dan **parameter** ketiga **space** yang akan digunakan untuk **file json**.

ReadJson

Jika kita ingin membaca sebuah **file json**, **deno** sudah menyediakan fungsi bernama **readJson**. Pada kode di bawah ini kita membaca sebuah **file** bernama **user.json** yang telah kita buat sebelumnya :

```
import { readJson } from "https://deno.land/std/fs/mod.ts";

const res = await readJson("./user.json");
console.log(res);
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// { name: "Maudy" }
```

writeFileStr

Jika kita ingin menulis sebuah **file**, **deno** sudah menyediakan fungsi bernama **writeFileStr**. Pada kode di bawah ini kita mencoba menulis sebuah **file** teks :

```
import {
  writeFileStr,
} from "https://deno.land/std/fs/mod.ts";

writeFileStr("./surat.txt", "Hello Maudy");
```

Pada **parameter** pertama terdapat lokasi yang akan kita gunakan untuk menulis dan **parameter** kedua adalah **string** yang akan kita tulis sebagai **file**.

ReadFileStr

Jika kita ingin membaca sebuah **file**, **deno** sudah menyediakan fungsi bernama **readFileStr**. Pada kode di bawah ini kita mencoba menulis sebuah **file** teks :

```
import { readFileStr } from "https://deno.land/std/fs/mod.ts";

const res = readFileStr("./surat.txt", { encoding: "utf8" });
res.then((result)=> {console.log(result)})
```

Pada **parameter** pertama kita menentukan lokasi file yang ingin kita baca dan yang kedua konfigurasi **encoding** yang kita gunakan yaitu **UTF8**. Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// Hello Maudy
```

Walk

Untuk mengetahui terdapat **file** dan **directory** apa saja di dalam suatu **directory**, **deno** menyediakan fungsi **walk**. Pada kode di bawah ini kita mencoba melakukan iterasi terdapat **file** apa saja di dalam **current directory** :

```
import { walk } from "https://deno.land/std/fs/mod.ts";

// Async
async function printFilesNames() {
  for await (const entry of walk(".")) {
    console.log(entry.path);
  }
}
```

```
printFileNames().then(() => console.log("Done!"));
```

Jika kode di atas dieksekusi maka ***list file*** dan ***directory*** akan ditampilkan.

Subchapter 2 – HTTP Server

*If you don't know where you are going,
you'll end up someplace else.*

—Yogi Berra

Subchapter 2 – Objectives

- Menggunakan **Standard Module HTTP**
 - Menggunakan **Function serve** dalam **Module HTTP**
 - Memahami cara kerja **request** dalam **Module HTTP**
 - Memahami cara kerja **response** dalam **Module HTTP**
-

Pada **Deno** untuk membuat sebuah **HTTP server** cukup mudah, kita dapat menggunakan **Standard Module HTTP**.

Di bawah ini adalah kerangka paling dasar bagaimana sebuah **web server** sederhana dapat dibangun di dalam **Deno** :

```
import { serve } from "https://deno.land/std/http/server.ts";
const server = serve({ port: 8000 });

console.log("http://localhost:8000/");

for await (const req of server) {
  req.respond({ body: "Hello World\n" });
}
```

Fungsi **serve** digunakan untuk membangun sebuah **HTTP Server**, kita membutuhkan parameter berupa **object** dengan **property port** dengan nilai **8000** atau anda dapat menentukan sendiri **port** yang ingin digunakan.

Selanjutnya untuk membaca **request** yang dilakukan oleh **user** kita menggunakan **for...of statement**, variabel **req** akan menyimpan seluruh informasi **HTTP Request** yang datang menuju **HTTP Server** :

```
for await (const req of server) {  
  req.respond({ body: "Hello World\n" });  
}
```

Pada kode di atas untuk memberikan **HTTP Response** kepada **client**, kita menggunakan **method respond()** pada **object req**. **Parameter** yang kita gunakan adalah **object** dengan **property body** sebagai variabel untuk menyisipkan **body message** pada **HTTP Response**.

Subchapter 3 – Testing

*Quality is never an accident,
It is always result of intelligent effort.*

—John Ruskin

Subchapter 3 – Objectives

- Memahami Kenapa Wajib Melakukan **Software Testing**
 - Mengetahui Tipe **Software Testing** Secara Intensif
 - Mengetahui Konsep **Bug**
 - Mengetahui Konsep **Functional Testing**
 - Mengetahui Konsep **Unit Testing**
 - Mengetahui Konsep **Core Refactoring**
 - Mengetahui Konsep **Code Coverage**
 - Mengetahui Konsep **Integration Testing**
 - Mengetahui Konsep **System Testing**
-

Bab ini menjelaskan kajian **software testing** secara intensif, untuk yang lebih komprehensif silahkan baca buku-buku dibidang **software testing**.

Lalu kenapa sih kita harus melakukan **software testing**?

1. Why Software Testing

Aktivitas **software testing** dalam pengembangan **software** adalah hal wajib untuk memastikan sistem yang kita bangun terbebas dari kecacatan (**Software Defect**).

Jika melihat kembali dalam sejarah ada beberapa catatan menarik kerugian yang ditimbulkan karena **software testing** tidak berjalan dengan baik :

1. Tahun 2015 **Bloomberg terminal** di Inggris mengalami kegagalan **software** yang mengakibatkan 300 ribu trader mengalami kerugian trading.

2. Perusahaan mobil **Nissan** harus menarik kembali 1 juta mobilnya yang sudah memasuki pasar karena terdapat kegagalan **software** pada **airbag sensor**.
3. **Starbuck** sempat harus menutup 60% tokonya di US dan Kanada karena terdapat kecacatan dalam **Point-of-Sale (POS)** sistemnya.
4. Pada bulan April tahun 1999, terdapat **software bug** yang mengakibatkan satelit militer seharga 1.2 milyar **US Dollar** gagal mengudara.

2. Type of Software Testing

Bug

Bug adalah sebuah kecacatan dalam program yang memproduksi kerugian. **Bug** adalah konsekuensi dari kesalahan **coding**.

Ada fakta menarik terkait Sistem Operasi **Windows** 2000 yang diketahui tetap menjual produknya meskipun di dalamnya terdapat 63 ribu **bug** lebih.

Bug selalu hadir dalam setiap pengembangan **software**, semakin rumit dan besar sistem yang dibangun semakin besar juga **software bug** yang dapat ditimbulkan. Hal ini yang mendorong bahwa peran **software testing** sangatlah vital.

Defect Level

Setiap **bug** memiliki **defect level** yang harus segera diprioritaskan :



Gambar 312 Defect Level

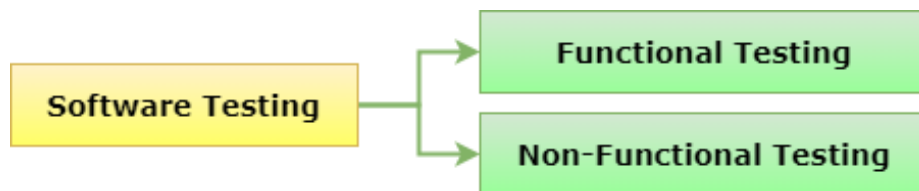
1. Sebuah **bug** dengan **level critical** harus segera diprioritaskan untuk menyelamatkan potensi **software failure** yang dapat merusak produk.
2. Sebuah **bug** dengan **level high** diprioritaskan setelah **bug level critical** diselesaikan, **bug** ini cenderung merusak fitur utama atau fitur unggulan dari produk.
3. Sebuah **bug** dengan **level medium** diprioritaskan setelah **bug level high** diselesaikan, **bug** ini cenderung terjadi **secondary feature** dalam produk. Namun

dengan **magnitude** kerugian yang lebih rendah dibandingkan dengan **bug level high**.

4. Sebuah **bug** dengan **level low** diprioritaskan setelah **bug level medium** diselesaikan, **bug** ini cenderung terjadi memberikan **minor impact** dalam produk.

Functional Testing

Untuk melakukan **software testing** sendiri terbagi menjadi beberapa area, terdapat dua tipe **software testing** yaitu :

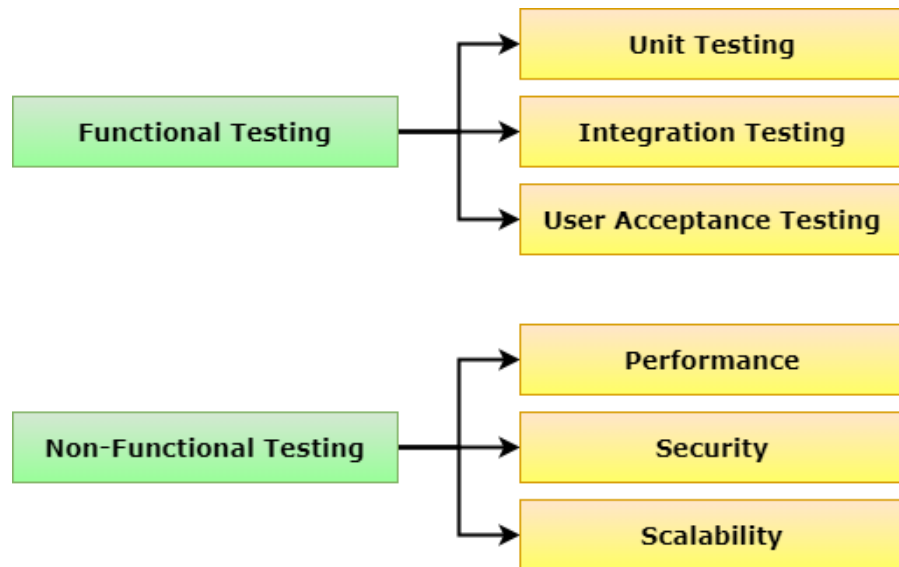


Gambar 313 Type of Software Testing

Functional testing adalah proses untuk memastikan sistem atau komponen dalam sistem sudah sesuai dengan **requirement** pengembangan **software**.

Functional testing dapat meningkatkan komunikasi antara **analyst**, **tester** dan **developer** dengan begitu pengembangan **software** bisa menjadi lebih cepat.

Baik **functional** atau **non-functional testing** terbagi menjadi beberapa kajian :



Gambar 314 Functional & Non-functional Testing

Unit Testing

Fungsi dari **unit testing** untuk memverifikasi **correctness** dari sekumpulan kode (**component**) yang kita tulis. Sehingga kepastian **component** bekerja dengan benar sesuai spesifikasi dapat diketahui di level terkecil.

"If you don't like unit testing your product, most likely your customers won't like to test it either."

Semakin banyak kode yang kita tulis tanpa melalui **unit testing** semakin banyak potensi **error** dan **technical debt** yang akan ditimbulkan kedepannya.

Technical debt akan merenggut waktu anda untuk dimasa depan untuk memperbaiki **component** yang seharusnya sudah selesai sebelumnya. Biasakanlah bekerja sampai tuntas, bekerja dengan tanggung jawab yaitu menulis kode lengkap dengan **unit testing**.

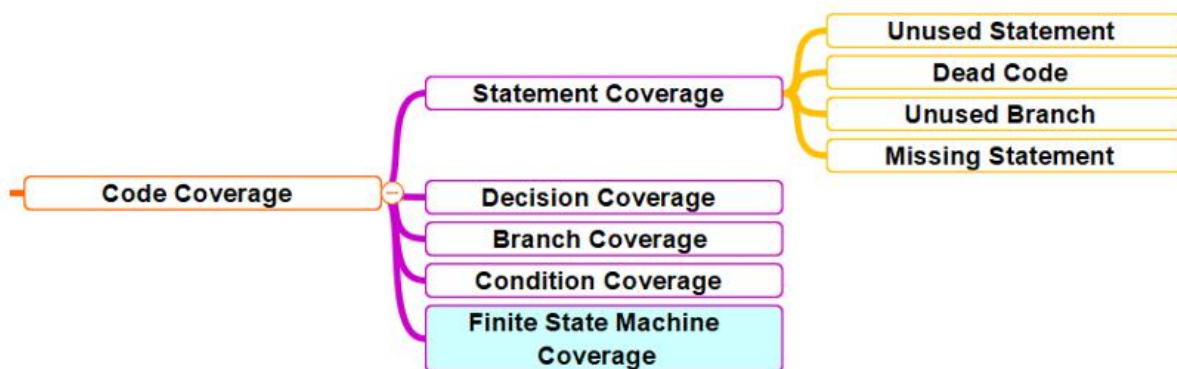
Code Refactoring

Unit testing sangat bergantung pada **mock object** yang kita buat untuk menguji kode yang ditulis, **mock object** adalah data buatan yang digunakan untuk menguji bahwa kode yang kita tulis dapat berjalan dengan benar.

Code refactoring adalah **process** dimana kita melakukan restrukturisasi pada sekumpulan kode (**component**) yang kita tulis, tanpa mengubah sifat dan tujuannya.

Code Coverage

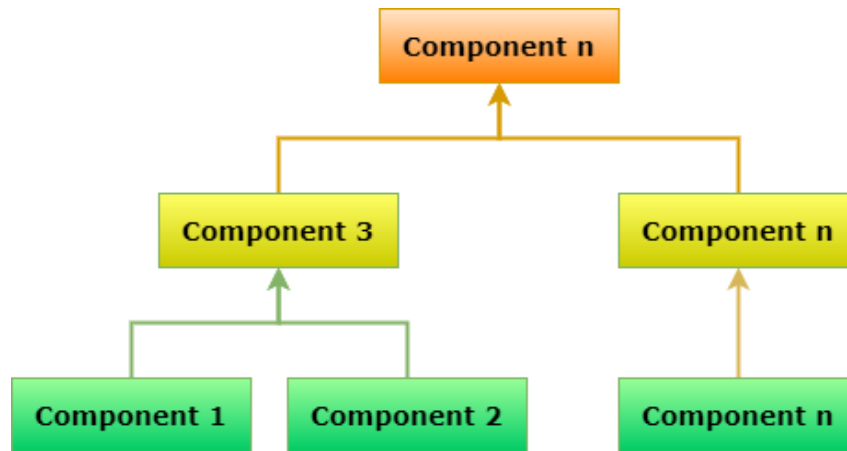
Penggunaan **code coverage** untuk mengukur sudah seberapa jauh **source code** dalam program diuji (**tested**). Pengukuran juga dilakukan secara quantitaf.



Gambar 315 Code Coverage

Integration Testing

Setelah kita menulis sekumpulan kode (**components**) di fase **unit testing** selanjutnya kita harus melakukan integrasi komponen tersebut kedalam **code base** atau sesama komponen lainnya.



Gambar 316 Integration Testing

Kita harus dapat memastikan integrasi antara komponen atau integrasi pada **code base** yang sudah ada berjalan dengan baik. **Integration test** menegaskan bahwa komponen yang dibuat dapat berjalan dengan baik atau tidak.

System Testing

System Testing adalah pengujian **software** yang sudah diklaim menjadi produk yang selesai berdasarkan **requirement** dan spesifikasi yang dibutuhkan. **System testing** memiliki kajian yang sangat luas.

Terdapat 50 lebih tipe untuk melakukan **system testing**, namun ada beberapa faktor yang dapat kita tentukan :

1. Siapa **tester** yang akan melakukan **testing**
2. Waktu yang tersedia untuk melakukan **testing**
3. **Resources** yang tersedia untuk melakukan **testing**
4. **Budget** untuk melakukan **testing**

Use Acceptance Testing

User Acceptance Testing adalah ***testing*** yang akan dilakukan oleh ***client*** yang membeli ***product*** untuk memastikan bahwa ***client*** dapat menerimanya, sebelum ***product*** di bawah ke lingkungan ***production***.

Alpha & Beta Testing

User Acceptance Testing juga terbagi menjadi dua fase yaitu ***alpha testing*** dan ***beta testing***, pada ***alpha testing*** pengujian dilakukan sebelum ***product*** diuji oleh ***real user*** dan ***beta testing*** adalah fase dimana produk yang akan diuji oleh ***real user*** dalam ***real environment*** dengan sekup yang masih terbatas.

3. Deno Testing

Deno telah menyediakan **built-in tool test runner** untuk membantu kita melakukan **testing** dengan dukungan untuk dapat melakukan **full-suite testing**. Jika kita mengeksekusi perintah di bawah ini dalam suatu **directory** :

```
> deno test
```

Maka deno akan secara otomatis melakukan pencarian secara rekursif untuk mencari **file** yang akan kita **testing** baik itu ekstensi **.js** ataupun **.ts**.

Assertion

Konsep **assertion** digunakan untuk menguji suatu kebenaran, pada **deno** terdapat dua **function** dasar untuk mempelajari **testing** yaitu **object assert** dan **equal** :

```
import { equal, assert } from "https://deno.land/std/testing/asserts.ts";

Deno.test("Test Assertion", () => {
  assert(1 === 1);
});

Deno.test("Test Equal", () => {
  equal(1, 1);
});
```

assert Function

Parameter yang harus digunakan dalam **assert function** adalah sebuah **boolean expression** yang harus memberikan nilai **true** jika ingin dianggap berhasil, jika tidak maka dapat disimpulkan terdapat **bug** di dalamnya.

```
> deno test testing.ts
```

Jika perintah di atas dieksekusi maka akan memproduksi :

```
test Test Assertion ... ok (3ms)
test Test Equal ... ok (0ms)
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out (3ms)
```

Pada informasi di atas kita dapat menyimpulkan bahwa :

1. **Unit Test** pada **Test Assertion** berhasil
2. **Unit Test** pada **Test Equal** berhasil
3. Terdapat dua **testing** yang berhasil

Lalu apa yang terjadi jika kita menguji **testing** dengan kode sebagai berikut :

```
Deno.test("Test Assertion", () => {
  assert(1 === 1 + 1 / 2);
});
```

Jika kode di atas kita **test** maka akan memproduksi :

```
running 2 tests
test Test Assertion ... FAILED (1ms)
test Test Equal ... ok (2ms)
```

```
failures:
Test Assertion
failures:
      Test Assertion
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out (9ms)
```

Pada informasi di atas kita dapat menyimpulkan bahwa :

1. **Unit Test** pada **Test Assertion** Gagal
2. **Unit Test** pada **Test Equal** berhasil
3. Terdapat 1 **testing** yang berhasil dan 1 **testing** yang gagal

Pada pengembangan **software** untuk skala **industry** kode yang di **test** tidak akan sesederhana ini. Lalu apa perbedaan dari **method assert()** dan **method equal()**

equal Function

Untuk melakukan pengujian dengan mode *deep-equal*, **deno** menyediakan **function equal**, perhatikan pada kode di bawah ini kita melakukan pengujian **boolean expression** menggunakan **method assert()** dan **equal()** :

```
Deno.test("Test Assertion", () => {
  assert({ hello: "world" } === { hello: "world" });
});

Deno.test("Test Equal", () => {
  equal({ hello: "world" }, { hello: "world" });
});
```

Jika kode di atas kita **test** maka akan memproduksi :


```
running 2 tests
test Test Assertion ... FAILED (0ms)
test Test Equal ... ok (2ms)
failures:
Test Assertion
AssertionError:
failures:
    Test Assertion
test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out (6ms)
```

Perbedaanya adalah **method** `equal()` dapat melakukan pemeriksaan pada struktur **operand** pada **boolean expression** lebih baik dari pada **method** `assert()`. Kedua **operand** yang diberikan adalah sebuah **object** yang notabene adalah **reference type**.

4. Deno Benchmarking

Deno juga menyediakan fitur agar kita dapat melakukan **benchmarking**, untuk mengukur **performance** dari kode yang kita tulis. Di bawah ini adalah contoh kerangka dasar untuk melakukan **benchmarking** :

```
import { runBenchmarks, bench } from "https://deno.land/std/testing/bench.ts";

bench(function forIncrementX1e9(b: any): void {
  b.start();
  for (let i = 0; i < 1000000000; i++);
  b.stop();
});

runBenchmarks();
```

bench Function

Fungsi ini digunakan untuk mendaftarkan kode yang akan kita **benchmark**, pada kode di atas ingin melakukan **benchmark** pada sebuah **function** yang akan melakukan **looping**.

Parameter **b** digunakan sebagai **object** yang dapat kita gunakan untuk mengeksekusi **method** **start()** dan **stop()**. Kode yang akan kita benchmark harus dibungkus di antara kedua **method** tersebut.

runBenchmarks Function

Fungsi ini digunakan untuk mengeksekusi proses **benchmarking** yang telah kita buat. Jika kita melakukan **invoke** **runBenchmark()** pada kode di atas maka akan memproduksi **output** sebagai berikut :

```
running 1 benchmark ...  
benchmark forIncrementX1e9 ...  
    56ms  
benchmark result: DONE. 1 measured; 0 filtered  
running 0 tests  
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered  
out (0ms)
```

Informasi di atas menjelaskan kepada kita bawa :

1. Terdapat 1 **Benchmark**
2. Nama **Identifier** yang kita benchmark yaitu **forIncrementX1e9**
3. Waktu untuk mengesekusi **forIncrementX1e9**
4. Hasil akhir dari proses **benchmark**

Subchapter 4 – Debugging

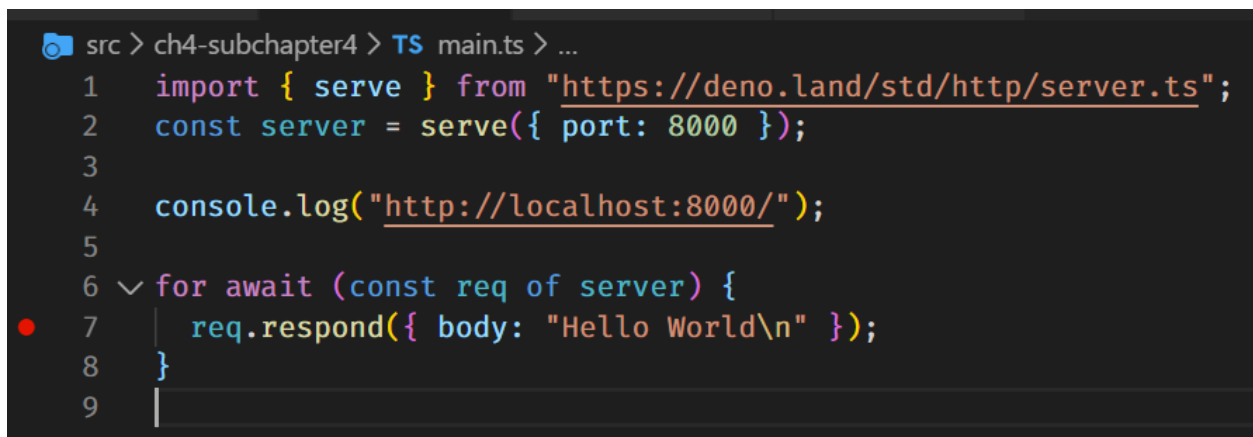
*Teach a man how to debug,
and you teach them for a lifetime.*

— Gun Gun Febrianza

Subchapter 4 – Objectives

- Memahami cara **debugging** dalam **Visual Studio Code**
 - Memahami cara **debugging** pada **Typescript**
 - Studi Kasus **Debugging Live Server**
-

Kita akan melakukan **debugging** pada **HTTP Server** yang telah kita pelajari sebelumnya :

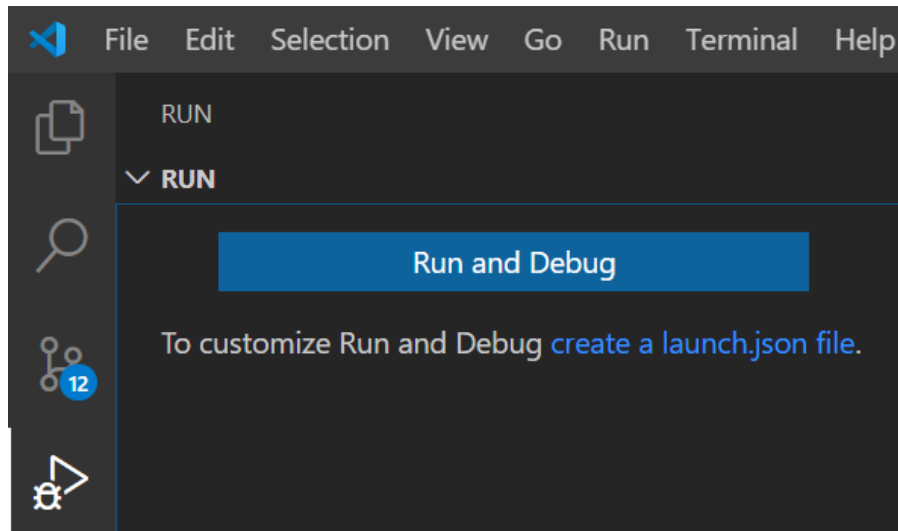


```
src > ch4-subchapter4 > TS main.ts > ...
1  import { serve } from "https://deno.land/std/http/server.ts";
2  const server = serve({ port: 8000 });
3
4  console.log("http://localhost:8000/");
5
6  for await (const req of server) {
7    req.respond({ body: "Hello World\n" });
8  }
9  |
```

Gambar 317 Debug HTTP Server

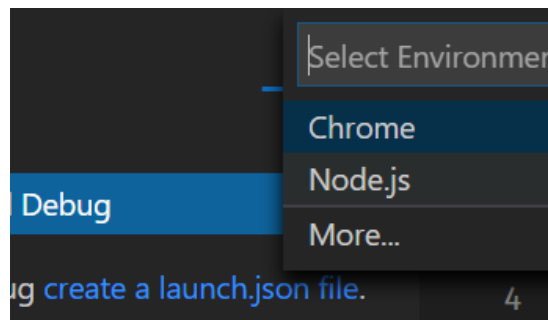
Arahkan **mouse** anda pada baris kode angka 7 sampai muncul simbol dot warna merah, kemudian klik sampai dot tersebut menempel. Proses ini disebut dengan **adding breakpoint**, agar kita dapat melakukan **debugging** pada baris kode tersebut.

Selanjutnya masuk ke menu **debug** anda dapat menggunakan **CTRL+SHIFT+D** :



Gambar 318 Run & Debug

Klik tombol **Run & Debug**, selanjutnya pilih **node.js** :



Gambar 319 Create Launch.json

Selanjutnya anda akan memiliki sebuah **file launch.json**, ganti seluruh kode yang ada di dalamnya dengan kode di bawah ini :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Deno",
```

```

    "type": "node",
    "request": "launch",
    "cwd": "${workspaceFolder}\\ch4-subchapter4",
    "runtimeExecutable": "deno",
    "runtimeArgs": ["run", "--inspect-brk", "-A", "main.ts"],
    "port": 9229
  }
]
}

```

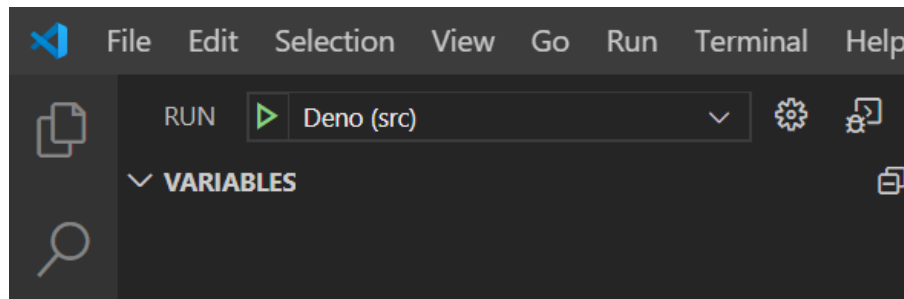
Pada konfigurasi di atas terdapat alamat **directory** dari **file** yang ingin kita **debug** :

```
"cwd": "${workspaceFolder}\\ch4-subchapter4"
```

Selain itu terdapat konfigurasi untuk menentukan nama **file** dan **flag** yang ingin kita gunakan :

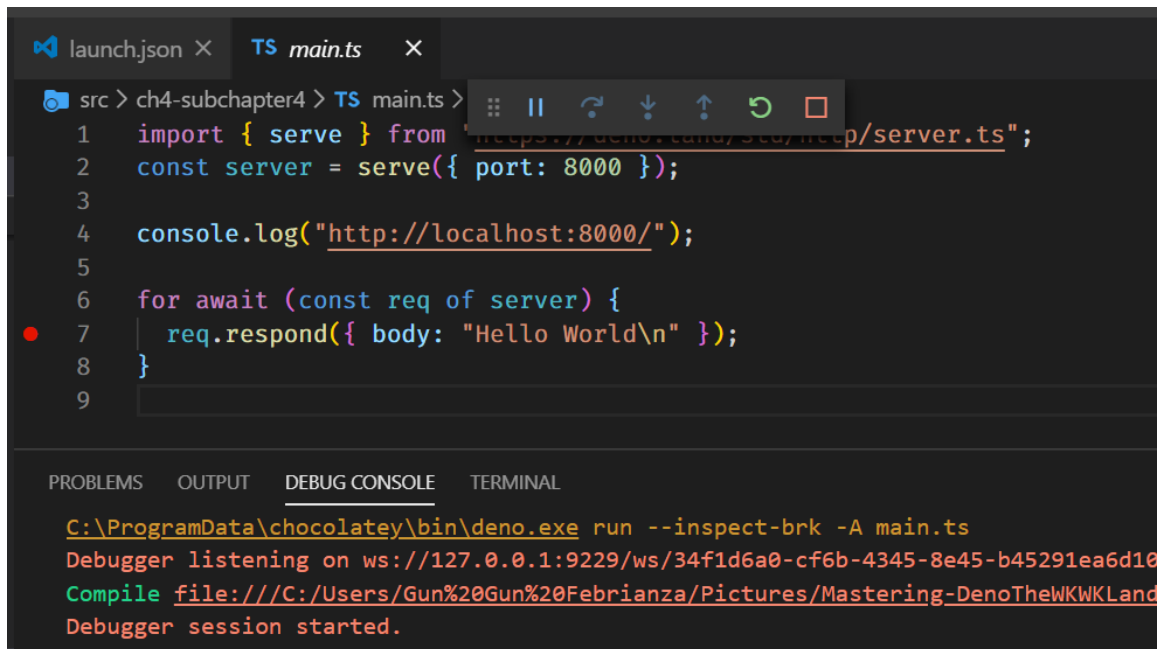
```
"runtimeArgs": ["run", "--inspect-brk", "-A", "main.ts"],
```

File yang akan penulis **debug** bernama **main.ts**, jika sudah tombol untuk melakukan **debug** akan muncul :



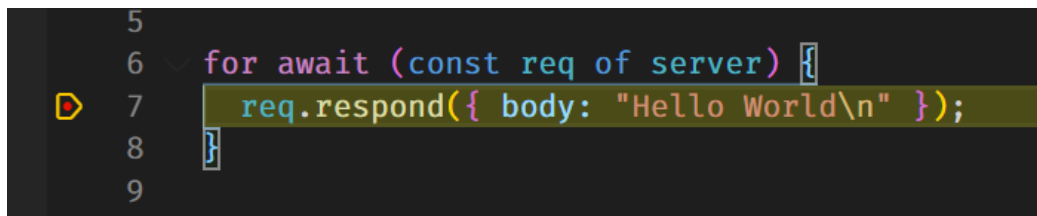
Gambar 320 Debug Deno Application

Klik tombol tersebut, jika berhasil maka anda sudah memasuki **mode debug** ditandai dengan munculnya **debug console** seperti pada gambar di bawah ini :



Gambar 321 Debug Console

Sekarang jika kita mencoba melakukan **request** menuju **localhost:8000** maka proses akan berjalan **stuck** karena kode berhenti pada baris ke 7. Jika gambar di bawah ini muncul maka anda berhasil melakukan **debugging application server** :



Gambar 322 Debugging Application Server

Jika kita arahkan **mouse** kita pada **object req**, maka kita dapat melihat struktur yang ada di dalamnya :

```
for await (const req of server) {  
  req.respond({ body: "Hello World\n" });  
}  
ServerRequest {done: Promise, _contentLength:...  
  done: Promise { pending }  
  finalized: false  
  headers: HeadersImpl {Symbol(headers data): ...  
    Symbol(headers data): Array(8) [Array(2), A  
    length: 8  
    > __proto__: Array(0) [, ...]  
    > 0: Array(2) ["host", "localhost:8000"]  
    > 1: Array(2) ["user-agent", "Mozilla/5.0 (W  
    > 2: Array(2) ["accept", "text/html,applicat  
    > 3: Array(2) ["accept-language", "en-GB,en;  
    > 4: Array(2) ["accept-encoding", "gzip, def  
    > 5: Array(2) ["connection", "keep-alive"]  
    > 6: Array(2) ["cookie", "_ga=GA1.1.63251224  
    > 7: Array(2) ["upgrade-insecure-requests",  
    Symbol(Symbol.toStringTag): undefined  
    > __proto__: DomIterable {constructor: }  
  method: "GET"  
  proto: "HTTP/1.1"
```

Gambar 323 ReqObject

Pada gambar di atas penulis mencoba melihat struktur **HTTP Header** yang terkirim pada **application server**. Dengan proses **debugging** ini anda dapat mengetahui bagaimana cara suatu kode bekerja dan seperti apa keadaan yang terjadi jika program ternyata memiliki **output** yang tidak sesuai ekspektasi.

Silahkan mempelajari **debugging** lebih dalam lagi di **chapter** selanjutnya tentang **debugging node.js**, baik **deno** atau **node.js** sama sama menggunakan **V8 Engine**.

Chapter 5

REST API

Subchapter 1 – Oak Framework

*When I'm working on a problem,
I never think about beauty.
I think only how to solve the problem.
But when I have finished,
if the solution is not beautiful,
I know it is wrong*
— William Mougayar

Subchapter 1 – Objectives

- Membuat **Basic Server** Menggunakan **Oak Framework**
 - Memahami Konsep **Middleware** pada **Oak Framework**
 - Memahami Konsep **Routing** pada **Oak Framework**
 - Memahami Konsep **Error Handling** pada **Oak Framework**
 - Memahami Konsep **Static Content** Pada **Oak Framework**
-

Oak adalah sebuah **web framework** untuk membantu kita membuat sebuah **web application**. Pengembangan **Oak** terinspirasi dari pengembangan **koa** sebuah **library** dalam **node.js** untuk membuat sebuah **web application**.

Kita akan melakukan praktek membuat **base application** sebuah **web server** menggunakan **oak**, melakukan **register middleware** dan melakukan **listening**.



Gambar 324 Learn Roadmap

1. Create Basic Server

Buatlah sebuah **file** dengan nama `server.ts`, di bawah ini adalah contoh kode yang menjadi kerangka dasar untuk membuat **basic web server** menggunakan **oak** :

```
import { Application } from "https://deno.land/x/oak/mod.ts";

const app = new Application();

app.use((ctx) => {
  ctx.response.body = "Hello World!";
});

await app.listen({ port: 8000 });
```

Eksekusi perintah di bawah ini :

```
> deno run --allow-net server.ts
```

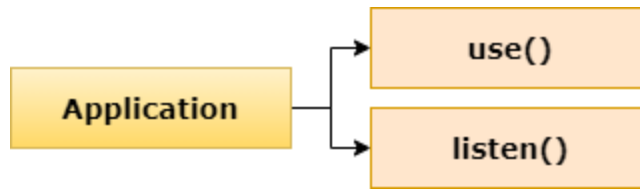
Buka **browser** anda kemudian akses `localhost:8000`

Class Application

Jika kita perhatikan pada kode di atas, kita melakukan **destructure** untuk **import class Application** agar bisa membuat sebuah **object** bernama `app` dari **class Application** :

```
const app = new Application();
```

Selanjutnya kita menggunakan dua **method** yaitu `use()` dan `listen()` :



Gambar 325 Oak

Register Middleware

Middleware adalah suatu **function** yang akan dieksekusi setiap kali terdapat suatu **request** pada **application server**. Pada kode di bawah ini untuk memasang sebuah **middleware** cukup menggunakan **method use()** :

```
app.use();
```

Selanjutnya kita memasang sebuah **middleware** berupa **anonymous function** sederhana untuk memberikan **HTTP Response** :

```
(ctx) => {  
  ctx.response.body = "Hello World!"  
}
```

Ketika dipasang menjadi :

```
app.use((ctx) => {  
  ctx.response.body = "Hello World!";  
});
```

Start Listening

Selanjutnya kita mengeksekusi **method** `listen()` yang membutuhkan **parameter object** yang memiliki **property port**, pada kode di bawah ini kita menggunakan **port 8000** :

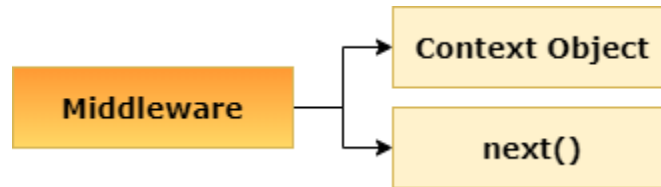
```
await app.listen({ port: 8000 });
```

Jika kita amati pada **statement** di atas, kita menggunakan **top-level await** yang bisa anda pelajari pada **Chapter 6, Subchapter 5** tentang **asynchronous**. Untuk yang sudah mempelajari konsep **async/await** intinya adalah bagaimana membuat **await statement** tanpa dibungkus dengan **async function**.

Listening artinya **application server** siap untuk memproses **HTTP Request** dan memproses setiap **middleware** yang terpasang di dalamnya.

2. Middleware

Middleware adalah **function** yang memiliki dua **parameter** yaitu **context object** dan **method next()** :



Gambar 326 Middleware Parameter

Middleware akan mengeksekusi beberapa pekerjaan di bawah ini :

1. Mengeksekusi **business logic**
2. Memanipulasi **request & response**
3. Menghentikan **request & response cyle**
4. Melanjutkan **flow** menuju **middleware** selanjutnya

Control Execution

Pada **middleware**, **control execution** dapat diatur menggunakan **method next()**.

Pada kode di bawah ini kita melakukan **signalling** untuk mengeksekusi **middleware** selanjutnya, jika sudah selesai kembali lagi ke **middleware function** disini dan mengeksekusi **statement** sisanya :

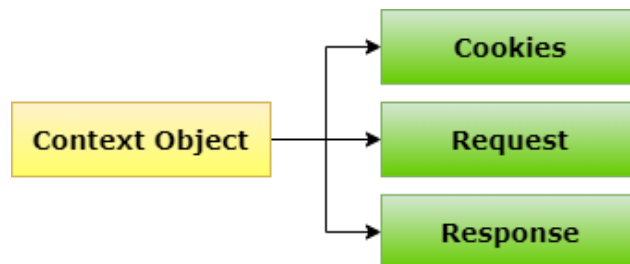
```
app.use(async (ctx, next) => {  
  await next();  
  // Do some things here  
});
```

Ada saatnya kita memiliki skenario harus melakukan **process** komputasi terlebih dahulu, kemudian membiarkan **middleware** selanjutnya untuk dieksekusi lalu kembali lagi ke sini untuk melakukan komputasi sebelum **HTTP Response** di kirim :

```
app.use(async (ctx, next) => {  
  // Process something  
  await next();  
  // do something before sending response  
});
```

Context Object

Middleware di proses secara **asynchronous by design** sehingga dengan **middleware function** kita dapat memberikan **return** sebuah **promise** ketika proses sudah selesai. Pada **context object** terdapat **properties** penting yang perlu kita ketahui :



Gambar 327 Context Object

Cookies

Context object memiliki **property cookie object**.

HTTP memiliki karakteristik **stateless**, **server** tidak dapat mengetahui jika setiap **request** yang datang menuju **server** bisa dilakukan oleh **client** yang sama. Untuk mengatasi permasalahan ini **cookies** diciptakan.

Cookie adalah data yang dikirim oleh **server** melalui **HTTP Response**, **client** dapat menyimpan informasi **cookie** dan menyimpannya kedalam **HTTP Request** agar **state** informasi dapat dibaca oleh **server**.



Gambar 328 Cookies

Server dapat membuat **cookies** dan membaca **cookies** yang dikirim oleh **client**. Dalam **oak** terdapat 3 **methods** yang disediakan saat kita ingin beroperasi dengan **cookies** dapat anda lihat pada diagram di bawah ini :



Gambar 329 Cookies Method

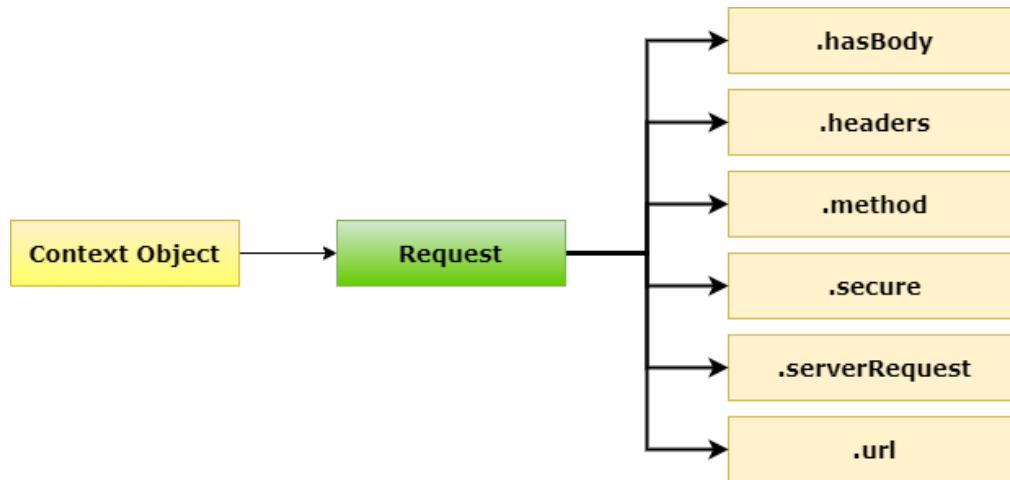
Request

Context object memiliki **property request object**.

Seluruh informasi **HTTP Request** pada **oak** akan tersimpan dalam **property request**. Di dalam **request object** terdapat **properties** dan **methods** yang dapat kita gunakan untuk mengelola **HTTP Request**.

Request Properties

Di bawah ini adalah **properties** yang ada di dalam **request object** :



Gambar 330 Object Request Properties

hasBody Property

Akan memberikan nilai **true** jika **HTTP Request** mengandung **body** dan memberikan nilai **false** jika **body** tidak tersedia dalam **HTTP Request**.

method Property

Informasi **HTTP Method** yang digunakan dalam **HTTP Request**.

secure Property

Akan memberikan nilai **true** jika protokol yang digunakan adalah **HTTPS** dan memberikan nilai **false** jika protokol yang digunakan adalah **HTTP**.

serverRequest Property

Tempat informasi **local address** dan **remote address** tersimpan.

url Property

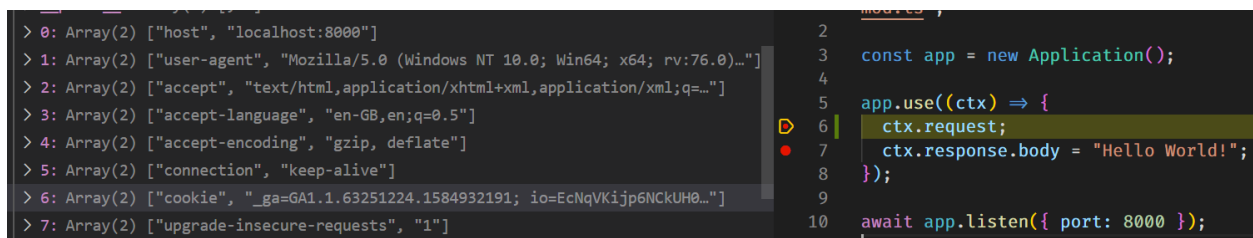
Tempat informasi **URL** secara lengkap tersedia seperti **host**, **hostname**, **href**, **origin**, **pathname**, **port**, dan **protocol** tersedia.

Request Headers

Disini kita dapat membaca **HTTP Headers** seperti informasi :

1. **Host**
2. **User Agent**
3. **Accept**
4. **Accept Language**
5. **Accept Encoding**
6. **Connection**
7. **Cookie**
8. **Upgrade Insecure Request**

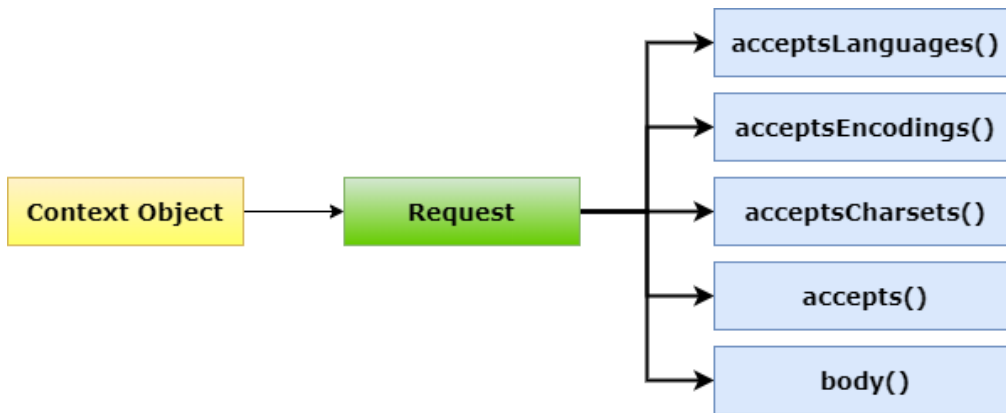
Kita dapat melihatnya saat melakukan **debugging** pada **object request** :



Gambar 331 Debug Headers

Request Methods

Di bawah ini adalah **methods** yang ada di dalam **request object** :



Gambar 332 Object Request Methods

accepts Method

Permintaan untuk melakukan negosiasi dari **client** terkait **content-type** yang ingin diterima **client** melalui **HTTP Response**.

acceptsCharsets Method

Permintaan untuk melakukan negosiasi dari **client** terkait **character encoding** yang ingin diterima **client** melalui **HTTP Response**.

acceptsEncodings Method

Permintaan untuk melakukan negosiasi dari **client** terkait **content encoding** yang ingin diterima **client** melalui **HTTP Response**.

acceptsLanguage Method

Permintaan untuk melakukan negosiasi dari **client** terkait **language** yang ingin diterima **client** melalui **HTTP Response**.

body Method

Saat ini **Oak** mendukung **request body** dalam bentuk :

1. **JSON**
2. **Text**
3. **URL Encoded Form Data**

Jika **content type** tidak ada **request** akan di **reject** dengan memberikan **HTTP Response 415 HTTP Error**.

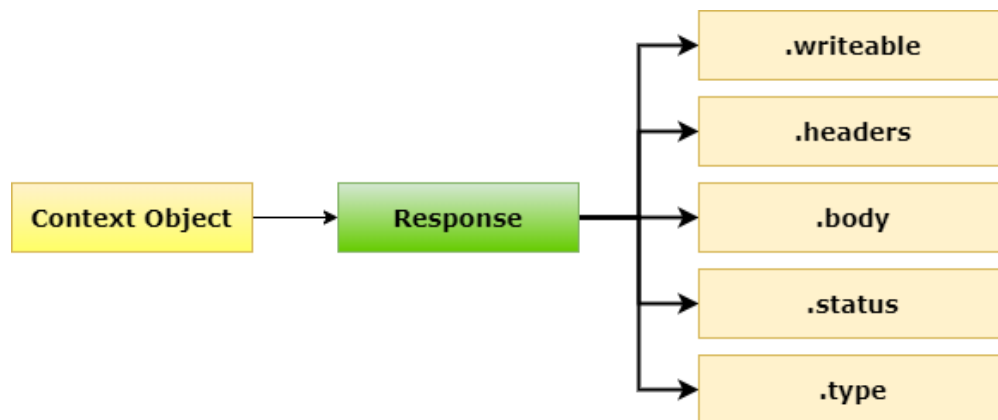
Response

Context object memiliki **property response object**.

Informasi yang ingin kita berikan kepada **client** akan kita atur melalui **response object**.

Response Properties

Di bawah ini adalah **properties** yang ada di dalam **response object** :



Gambar 333 Response Properties

type Property

Tempat kita dapat menentukan **media type** untuk mengatur **content-type header** pada **HTTP Response** yang akan dikirimkan pada **client**. Sebagai contoh kita dapat memberikan **txt** atau **text/plain**.

status Property

Tempat kita dapat menentukan **status code** yang akan dikirimkan pada **client**. **Oak** akan memberikan **status code 200 ok** secara **default** jika kita tidak membuat pengaturan **status code**.

Di bawah ini adalah contoh **statement** saat kita memberikan sebuah **status code** :

```
ctx.response.status = 200;
```

body Property

Tempat kita dapat menentukan **body** pada **HTTP Response** yang akan dikirimkan pada **client**. Di bawah ini adalah contoh **statement** saat kita memberikan **body** berupa **string** pada **HTTP Response** :

```
ctx.response.body = "Hello World!";
```

headers Property

Tempat kita dapat menentukan **headers** pada **HTTP Response** yang akan dikirimkan pada **client**.

Response Method

Di bawah ini adalah **methods** yang ada di dalam **response object** :



Gambar 334 Response Methods

Redirect Methods

Sebuah ***method*** yang dapat kita gunakan untuk melakukan ***redirection*** ke ***URL*** yang lain. Di bawah ini adalah contoh kode untuk melakukan ***redirection*** :

```
import { Application } from "https://deno.land/x/oak/mod.ts";

const app = new Application();

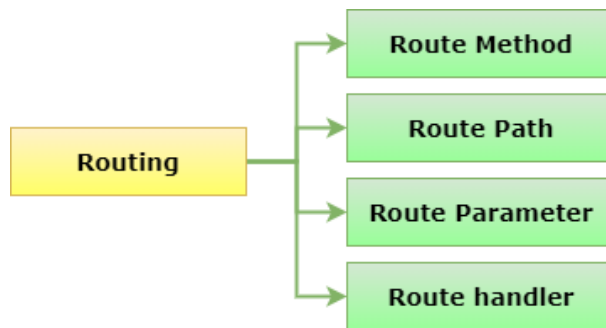
app.use((ctx) => {
  ctx.response.redirect("https://deno.land/");
});

await app.listen({ port: 8000 });
```

3. Routing

Routing adalah konsep bagaimana aplikasi akan merespon ketika **client** melakukan **HTTP Request** pada **endpoint** tertentu. **Routing** menentukan bagaimana aplikasi akan merespon berdasarkan **request** pada **URI** dan **HTTP Method** yang diberikan.

Untuk memahami konsep **routing** adalah beberapa kunci yang harus anda fahami, diantaranya adalah :



Gambar 335 Routing Concept

Di bawah ini adalah kerangka sederhana implementasi **routing** pada **oak** :

```
import { Application, Router } from "https://deno.land/x/oak/mod.ts";

const router = new Router();
router
  .get("/", (context) => {
    context.response.body = "Hello world!";
  });

const app = new Application();
app.use(router.routes());
app.use(router.allowedMethods());
```

```
await app.listen({ port: 8000 });
```

Class Router

Jika kita perhatikan pada kode di atas, kita melakukan **destructure** untuk **import class Router** agar bisa membuat sebuah **object** bernama **router** dari **class Router** :

```
const router = new Router();
```

Route Method

Pada kode di bawah ini kita menggunakan **route get()** :

```
router
  .get("/", (context) => {
    context.response.body = "Hello world!";
  });
```

Terdapat **route method** lainnya yang tersedia dalam **oak** :

```
router.get()
router.post()
router.delete()
router.put()
router.all()
router.options()
router.patch()
router.put()
```

Khusus untuk **router** `all()`, **method** ini dapat digunakan untuk menerima **request** menuju **router** *get*, *post*, *delete* atau *put* sekaligus pada **route** *path* tertentu.

Route Path

Route path adalah alamat **path** yang akan digunakan untuk melakukan **routing** :

```
.get("/", (context) => {  
  context.response.body = "Hello world!";  
})  
.get("/book", (context) => {  
  context.response.body = "Book!";  
});
```

Pada kode di atas kita membuat **route path** menuju dua alamat yaitu :

1. **"/"** atau **index**
2. **"/book"**

Kita dapat membangun **path** tertentu untuk membangun **business logic** yang berbeda-beda pada aplikasi yang kita buat.

Route Parameter

Route parameter adalah sebuah **parameter** dengan nilai dinamis yang tersedia dalam sebuah alamat **path**. Perhatikan contoh kode di bawah ini :

```
.get("/book", (context) => {  
  context.response.body = "Book!";  
})  
.get("/book/:id", (context) => {  
  context.response.body = `Book ID : ${context.params.id}`;  
});
```

Pada kode di atas kita menggunakan **parameter** dengan cara memberikan notasi :

:id

Artinya kita dapat melakukan **request** pada **path** :

- **localhost:8000/book/1**
- **localhost:8000/book/2**
- **localhost:8000/book/n**
- **localhost:8000/book/a**
- **localhost:8000/book/bbb1**
- **localhost:8000/book/n**

Untuk mendapatkan nilai dari **route parameter** yang kita buat kita dapat menggunakan **property params** pada **object context** seperti pada kode di bawah ini :

```
context.params.id
```

Route Handler

Jika kita perhatikan pada kode di bawah ini, **route handler** adalah **input** pada **parameter** kedua dari **route method** :

```
router
  .get("/", (context) => {
    context.response.body = "Hello world!";
  });
```

Biasanya direpresentasikan dalam sebuah **arrow function**.

Register Router Middleware

Selanjutnya anda akan melihat kode di bawah ini :

```
app.use(router.routes());
```

Kode di atas digunakan untuk meregistrasikan **router middleware** yang telah kita buat. Selanjutnya **object router** memanggil **allowedMethods()** untuk mengizinkan penggunaan **route method** pada **router middleware** yang telah ditanam :

```
app.use(router.allowedMethods());
```

4. Error Handling

Ada saatnya **middleware function** yang sedang kita buat memiliki potensi **error**, untuk mengatasinya perhatikan kerangka dasar untuk **error handling** di bawah ini :

```
import {
  Application,
  isHttpError,
  Status,
} from "https://deno.land/x/oak/mod.ts";

const app = new Application();

app.use(async (ctx, next) => {
  try {
    await next();
  } catch (err) {
    if (isHttpError(err)) {
      switch (err.status) {
        case Status.NotFound:
          // handle NotFound
          break;
        default:
          // handle other statuses
      }
    } else {
      // rethrow if you can't handle the error
      throw err;
    }
  }
});
```

Funtion isHttpError

Jika kita perhatikan pada kode di atas, kita melakukan ***destructure*** untuk ***import function*** **isHttpError** agar kita bisa mengetahui jika **error** merupakan **HTTP Error**. Dengan begitu kita dapat menentukan respon saat menghadapi **HTTP Error** :

```
if (isHttpError(err)) {  
    // process HTTP Error here  
} else {  
    // rethrow if you can't handle the error  
    throw err;  
}
```

Enum Status

Selanjutnya kita melakukan ***destructure*** untuk ***import enum*** **status** yang dapat kita gunakan untuk mengenali **HTTP Error** lebih spesifik lagi berdasarkan **status code**, karena terdapat banyak sekali varian **error** yang dapat terjadi seperti misal :

- ***Internal Server Error***
- ***Bad Gateway***
- ***Bad Request***
- ***Conflict***

Penempatan **enum status** diletakan di dalam **if statement** setelah diperiksa oleh **function isHttpError()** seperti pada kode di bawah ini :

```
if (isHttpError(err)) {  
    switch (err.status) {  
        case Status.InternalServerError
```

```
        // handle NotFound
        break;
    default:
        // handle other statuses
    }
}
```

Try & Catch

Pada kode di atas seluruh kode dibungkus menggunakan **try...catch statement** sebagai kunci untuk mendeteksi **error** sebelum dapat digunakan oleh **function isHttpError()** dan **enum status** seperti pada kode di atas.

5. Static Content

Oak telah membuat **function send()** agar kita bisa membuat **middleware function** yang dapat menyediakan **static content**. Kerangka kode sederhananya dapat di lihat pada kode di bawah ini :

```
import { Application, send } from "https://deno.land/x/oak/mod.ts";

const app = new Application();

app.use(async (context) => {
  await send(context, context.request.url.pathname, {
    root: `${Deno.cwd()}/examples/static`,
    index: "index.html",
  });
});

await app.listen({ port: 8000 });
```

Simpan **file HTML, CSS** dan **Javascript** yang kita buat ke dalam **folder examples/static** dan pastikan memiliki nama **index.html**.

Subchapter 2 – Introduction to Database

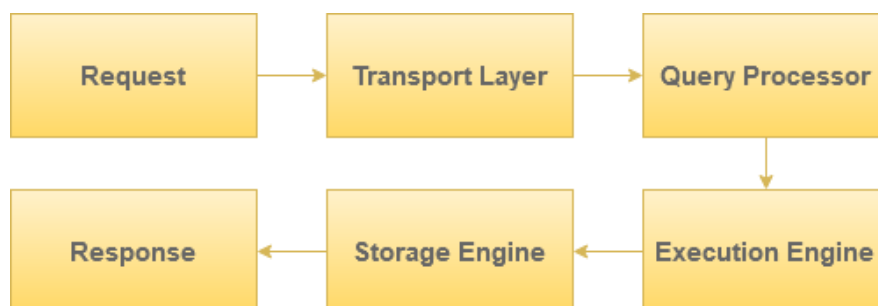
*The old question 'Is it in the database?',
will be replaced by 'Is it on the blockchain?.'*

— William Mougayar

Subchapter 2 – 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 manajemen 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 336 Database Under The Hood

Database terdiri dari beberapa sistem *modular* seperti **Transport Layer** 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 manajemen data dalam memori & *disk*.

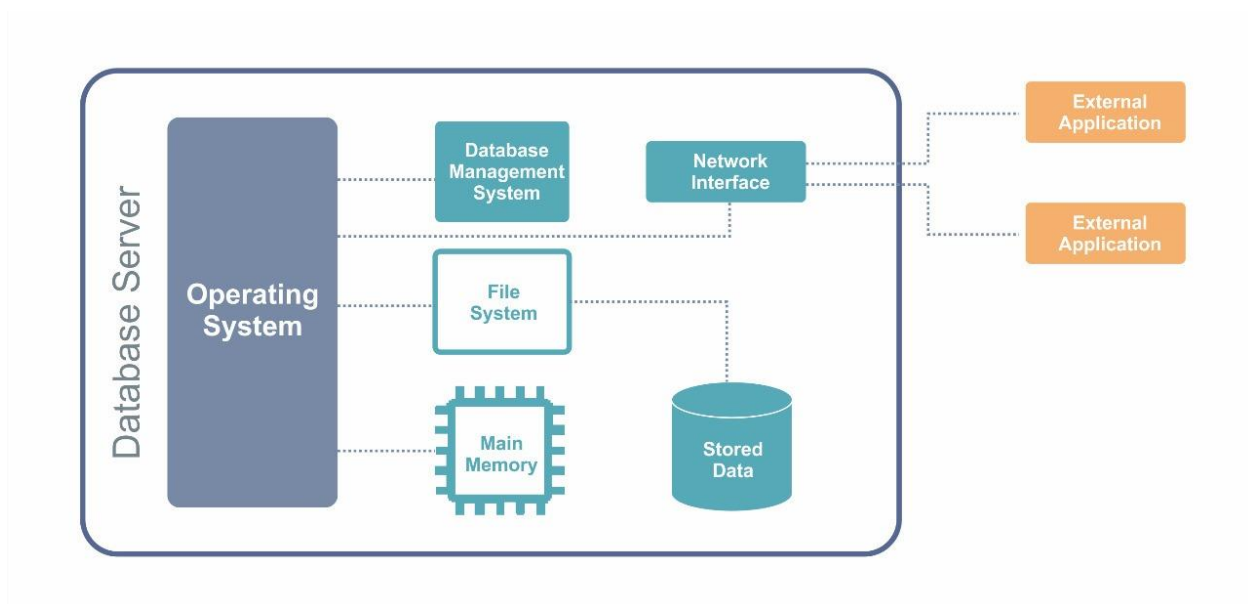
Salah satu *DBMS* yang populer seperti **MySQL** memiliki beberapa *storage engines* sekaligus, diantaranya adalah **InnoDB**, **MyISAM** dan **RockDB**. Pada *MongoDB*, kita dapat memilih menggunakan **WiredTiger**, **In-memory** dan **MMAPv1** yang kini telah usang.

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 337 Ekosistem Database Management System

Untuk membangun **interoperability** dengan aplikasi eksternal, sebuah sistem *database* harus menyediakan sebuah **interface** 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.

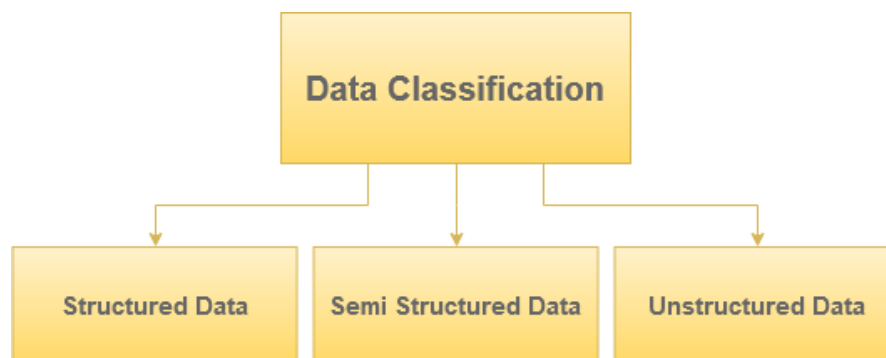
Jika salah satu operasi tidak terlaksana atau serangkaian operasi tidak sampai selesai, maka *transaction* gagal.

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 Heterogeneity

Terdapat 3 klasifikasi data yang dapat disimpan di dalam *database* :



Gambar 338 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*)

Efficiency

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 terjadi 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 lainnya.

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 3 – PostgreSQL

*The old question 'Is it in the database?',
will be replaced by 'Is it on the blockchain?.'*

— William Mougayar

Subchapter 3 – Objectives

- Memahami Cara Untuk berinteraksi dengan **PostgreSQL**
 - Mempelajari Program **psql**
 - Mempelajari Program **pgAdmin**
-

1. Intro PostgreSQL

Agar bisa berinteraksi dengan **PostgreSQL** kita harus terhubung dulu pada **PostgreSQL Server** untuk mendapatkan sebuah **session**. **Session** diperlukan agar kita dapat melakukan **request** pada **database server** dan menutup koneksi (**disconnect**).

Untuk terhubung pada **PostgreSQL Server** kita membutuhkan informasi :

1. **Host**
2. **Port**
3. **Database Name**
4. **User**
5. **Password**

Sebuah **host** baik itu **local** ataupun **remote** harus sudah terpasang **PostgreSQL Server** dengan kondisi **port** terbuka (**listening**), selanjutnya nama **database** dan **user credentials** juga harus sudah terdaftar.

Host juga harus memberikan izin secara eksplisit agar **client** dapat terhubung pada **PostgreSQL Server**, tersedia banyak tipe **authentication** agar **client** dapat terhubung.

Enable External Access

Untuk memberikan izin agar **PostgreSQL Server** dapat diakses oleh pihak eksternal kita harus melakukan konfigurasi pada **file postgresql.conf**. Berikan pengaturan seperti pada konfigurasi di bawah ini :

```
listen_addresses = '*'
```

Selanjutnya pada **file pg_hba.conf**, pada baris pertama konfigurasi berikan pengaturan seperti pada gambar di bawah ini :

#	TYPE	DATABASE	USER	ADDRESS	METHOD
host		all	all	0.0.0.0/0	md5
# IPv4 local connections:					
host		all	all	127.0.0.1/32	md5

Gambar 339 Configure Access

psql Program

psql adalah sebuah program CLI (**Command-line Interface**) yang dapat kita gunakan untuk mengeksekusi **query**.

Eksekusi program **psql** selanjutnya tekan **enter** sampai program meminta **password**, berikan **password** sesuai dengan **password** yang kita atur saat melakukan instalasi **postgreSQL** :

```
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
```

Jika berhasil maka anda akan memasuki **PostgreSQL** :

```
postgres=#
```

Untuk mengetahui informasi **session connection** yang kita lakukan eksekusi perintah di bawah ini :

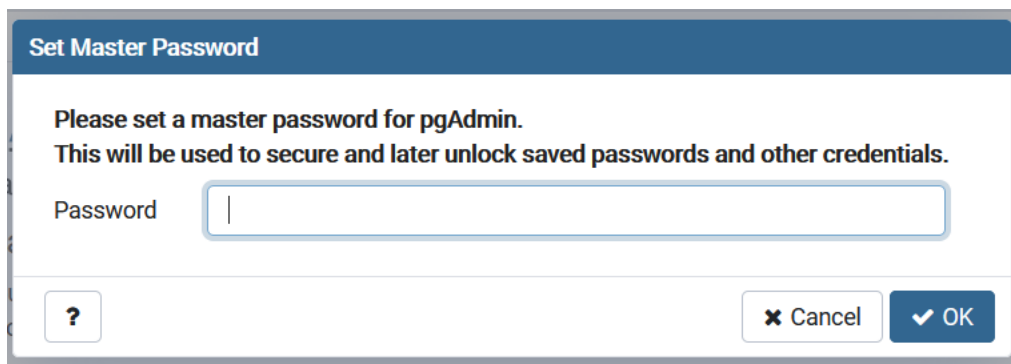
```
postgres-# \conninfo
```

```
You are connected to database "postgres" as user "postgres" on host  
"localhost" (address ":::1") at port "5432".
```

pgAdmin Program

PostgreSQL juga menyediakan program berbasis **Graphic User Interface (GUI)** yang dapat membantu **system administrator**. Saat pertama kali kita menggunakan **pgAdmin** kita akan diminta untuk membuat **master password**.

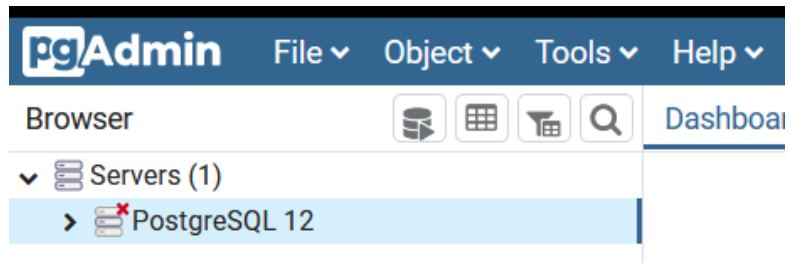
Set Master Password



Gambar 340 Set Master Password

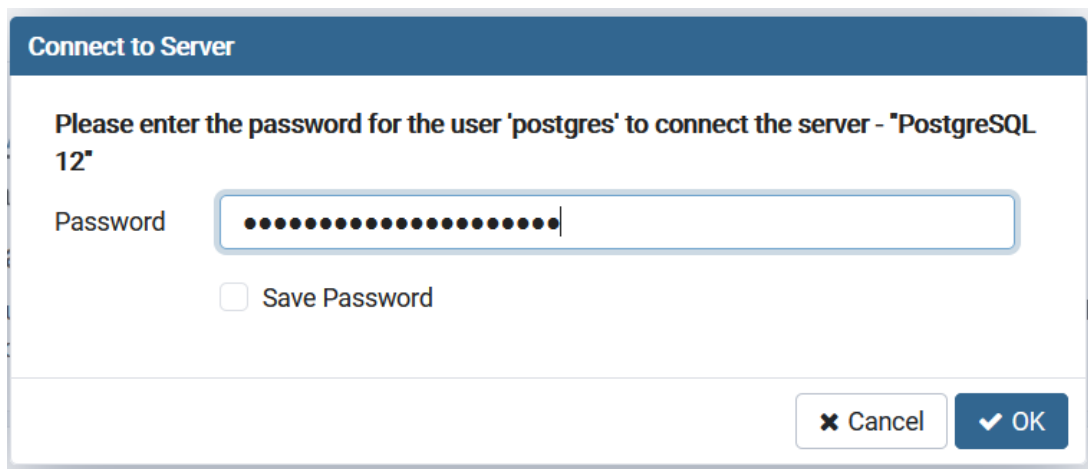
Connect to Server

Selanjutnya pada kolom **server** di pojok kiri atas kita dapat melihat terdapat satu buah **server** yang dapat kita gunakan untuk terhubung, klik ikon tersebut :



Gambar 341 Browser Servers

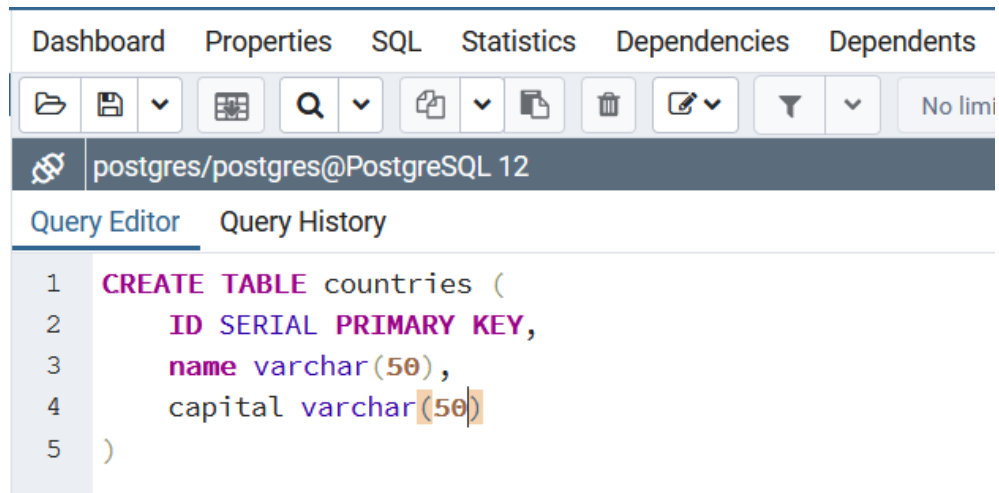
Selanjutnya kita akan diminta untuk memberikan **password** agar terhubung kedalam **server**, seperti pada gambar di bawah ini :



Gambar 342 Connect To Server

Query Tool

Selanjutnya pilih menu **Tools** → **Query Tools**, pada gambar di bawah ini kita dapat melihat ilustrasi **Query Editor** yan dapat kita gunakan untuk mengeksekusi **SQL Command** :



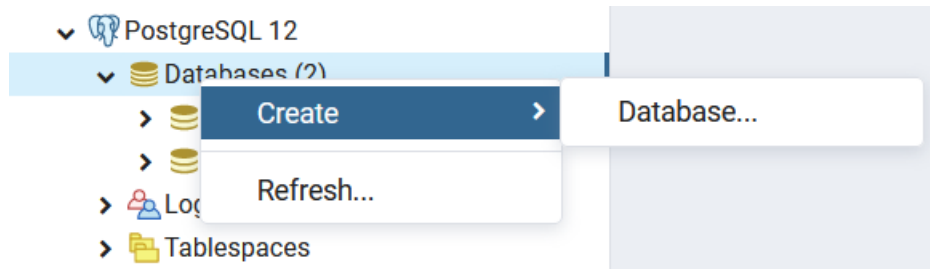
Gambar 343 Query Tool

Fitur ini akan kita gunakan untuk berinteraksi dengan **SQL Query**.

pgAdmin

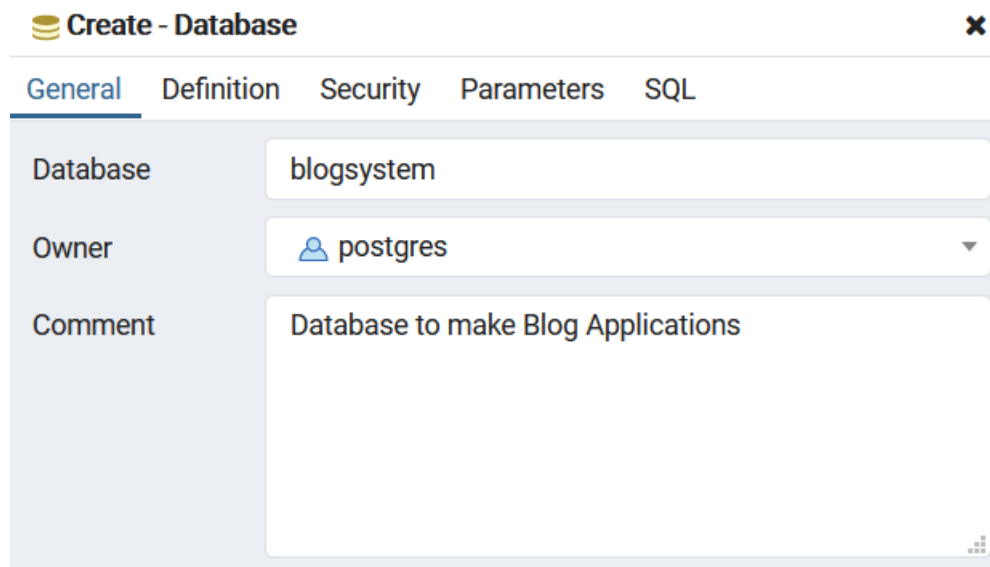
Create Database

Untuk membuat Database baru, pada menu database klik kanan pilih Create → Database... seperti pada gambar di bawah ini :



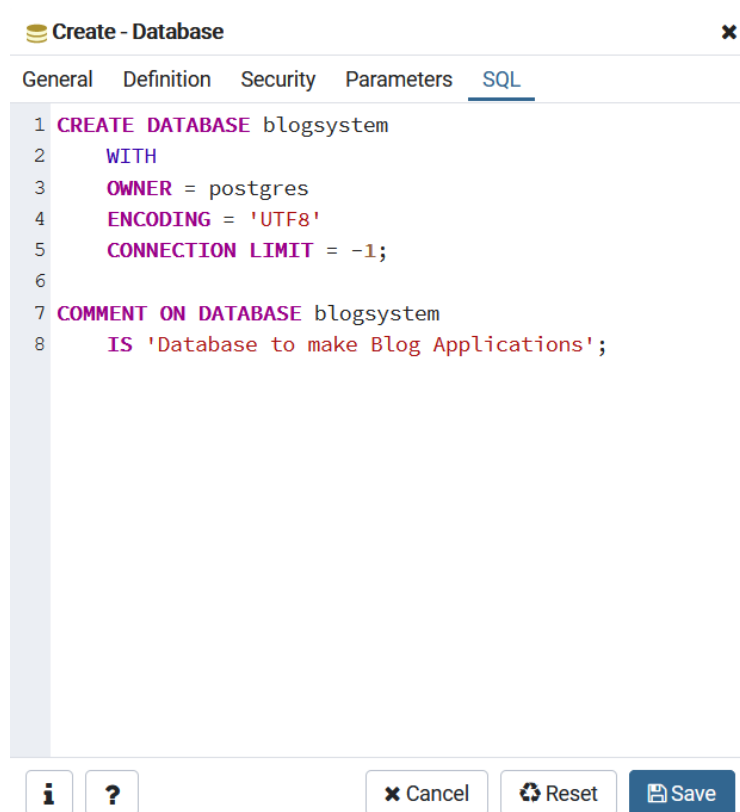
Gambar 344 Create Database

Isi kolom Database untuk memberikan nama pada database, kolom owner digunakan untuk menentukan siapa pemilik database dan kolom comment untuk memberikan komentar pada database. Anda dapat mengisinya seperti pada gambar di bawah ini :



Gambar 345 Database General Information

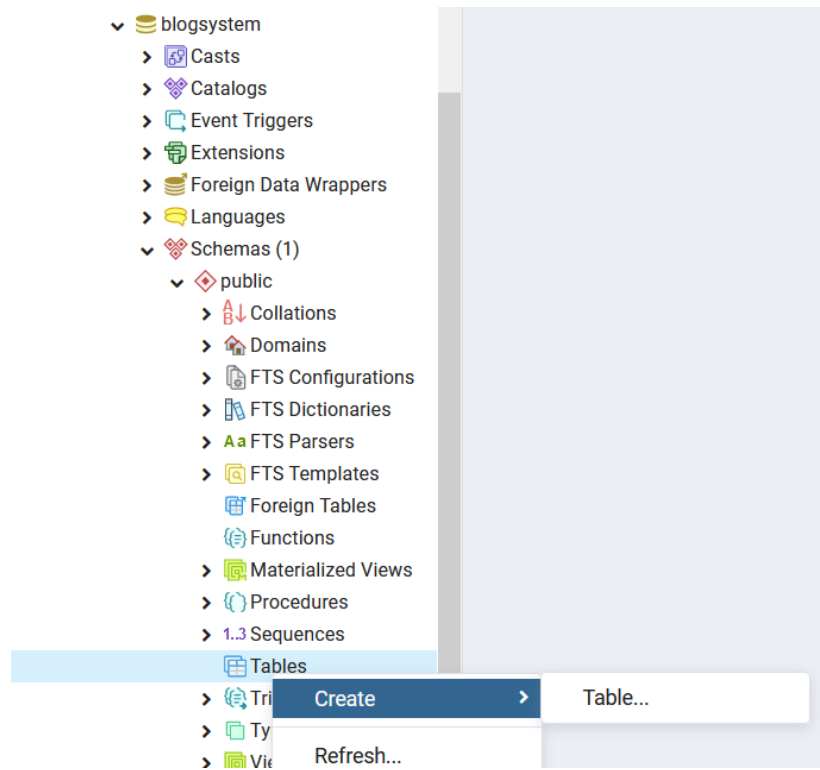
Jika sudah klik tombol **Save** :



Gambar 346 Save Database

Create Table

Untuk membuat **table** pada **database blogsystem**, klik ikon **Schemas** → **Public** kemudian pada **icon Tables** klik kanan lagi pilih **Create** → **Table...** seperti pada gambar di bawah ini :



Gambar 347 Create Table

Create - Table



General Columns Constraints Advanced Partitions Parameters Security SQL


Name

Articles

Owner


 postgres

Schema

 public



Tablespace

 pg_default




Partitioned table?


☐ No

Comment



 Cancel

 Reset

 Save

2. Database Administration

Create Role

Buka program **psql** kemudian eksekusi perintah di bawah ini untuk membuat **user** dengan nama **maudy** lengkap dengan **password** yang diberikan yaitu **ayunda** :

```
postgres=# CREATE ROLE maudy WITH LOGIN PASSWORD 'ayunda';  
CREATE ROLE
```

Selanjutnya kita harus memberikan **permission** pada **role maudy** dengan cara memberikan **privilege CREATEDB (Create Database)**, eksekusi perintah di bawah ini :

```
postgres=# ALTER ROLE maudy CREATEDB;  
ALTER ROLE
```

Selanjutnya untuk membuktikan apakah **role maudy** sudah memiliki **privilege** eksekusi perintah di bawah ini :

```
postgres=# \du  
  
List of roles  
  
Role name | Attributes | Member of  
-----+-----+-----  
maudy      | Create DB  | {}  
postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
```

Pada **output** informasi di atas kita dapat mengetahui bahwa **role maudy** sudah memiliki **privilege CREATEDB**. Selain **role maudy** juga terdapat **role postgres** sebuah **role** bawaan yang telah disediakan oleh **PostgreSQL**. Statusnya adalah **superuser**.

Create Database

Untuk membuat **database** kita perlu menggunakan **SQL Command – Data Definition Language (DDL) CREATE**. Kita akan membuat sebuah **database** dengan nama **BLOGAPP**, silahkan eksekusi perintah di bawah ini :

```
postgres=# CREATE DATABASE BLOGAPP;  
  
CREATE DATABASE
```

Grant Database

Untuk memberikan hak akses pada **database** kita perlu menggunakan **SQL Command – Data Control Language (DCL) GRANT**. Kita akan memberikan izin penuh pada **role maudy**, eksekusi perintah di bawah ini :

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE BLOGAPP TO maudy;  
  
GRANT
```

List Database

Untuk memastikan **database** berhasil dibuat eksekusi perintah di bawah ini :

```
postgres=# \list  
  
List of databases  
  
Name      | Owner  |  
-----+-----+  
blogapp   | postgres |  
postgres  | postgres |  
template0 | postgres |
```

```
template1 | postgres |
```

```
(4 rows)
```

Test Role Access Database

Selanjutnya buka lagi ***psql*** dan ***log in*** kedalam ***postgreSQL Server*** :

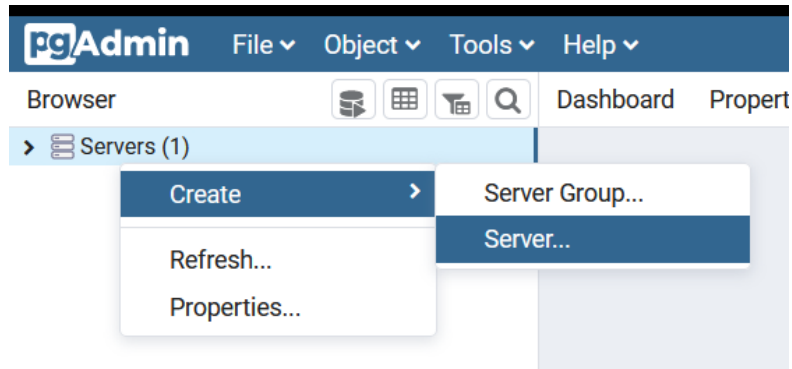
```
Server [localhost]:  
Database [postgres]: blogapp  
Port [5432]:  
Username [postgres]: maudy  
Password for user maudy:
```

Kita juga dapat memastikan kembali dengan cara mengeksekusi perintah di bawah ini :

```
earth=> \conninfo  
  
You are connected to database "blogapp" as user "maudy" on host  
"localhost" (address "::1") at port "5432".
```

Setup Server

Untuk membuat ***table*** agar lebih interaktif kita akan menggunakan program ***pgAdmin***, pada ***browser tree*** klik kanan pilih **Create → Server** seperti pada gambar di bawah ini :



Gambar 348 Create Server

Silahkan isi kolom Name :

 The image shows the 'Create - Server' dialog box with the 'General' tab selected. The 'Name' field contains the text 'LearnPostgreSQL'. The 'Server group' dropdown menu is open, showing 'Servers' as the selected option.

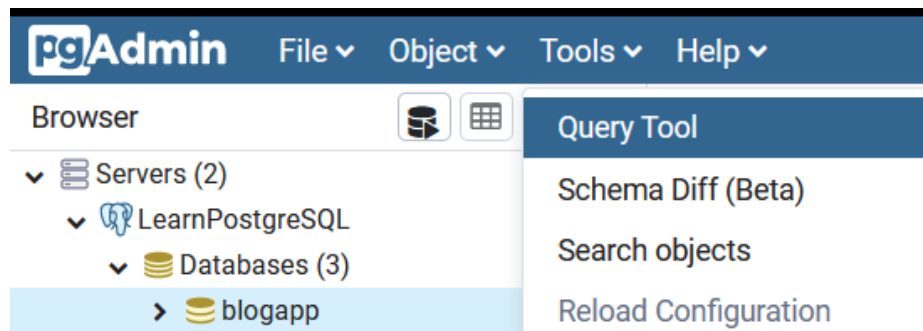
Gambar 349 Create Server Name

Isi seperti pada gambar di bawah ini :

 The image shows the 'Create - Server' dialog box with the 'Connection' tab selected. The 'Host name/address' field is 'localhost'. The 'Port' field is '5432'. The 'Maintenance database' field is 'blogapp'. The 'Username' field is 'maudy'. The 'Password' field is masked with dots.

Gambar 350 Create Server Connection

Selanjutnya klik ikon **blogapp** dan buka menu **Tools → Query Tool** :



Gambar 351 Query Tool

Create Table

Untuk membuat **table** lebih nyaman menggunakan **Query Tools** dalam **pgAdmin**, tulis **query** di bawah ini :

```
CREATE TABLE account(  
    user_id serial PRIMARY KEY,  
    username VARCHAR (50) UNIQUE NOT NULL,  
    password VARCHAR (50) NOT NULL,  
    email VARCHAR (355) UNIQUE NOT NULL,  
    created_on TIMESTAMP NOT NULL,  
    last_login TIMESTAMP  
);
```

Tekan tombol **run** untuk mengeksekusi **query**, jika berhasil maka **table** akan muncul :

Insert Data

Untuk melakukan operasi **insert** data, kita akan menggunakan **query editor** seperti pada gambar di bawah ini :

```
INSERT INTO account (username, password , email, created_on, last_login )
```

```
VALUES
```

```
('maudy','ayundafaZA2020', 'maudy@gmail.com',NOW(),NOW()),
```

```
('gungun','febrianZA2020', 'gungun@gmail.com',NOW(),NOW()),
```

```
('yuma','yusufu', 'yumadol@gmail.com',NOW(),NOW())
```

Eksekusi kode di atas dengan menekan tombol run, selanjutnya view data dari table jika berhasil maka kita dapat melihat data yang telah tersimpan :

Data Output		Explain	Messages	Notifications		
	<div><div>user_id</div><div>[PK] integer</div></div>	<div><div>username</div><div>character varying (50)</div></div>	<div><div>password</div><div>character varying (50)</div></div>	<div><div>email</div><div>character varying (355)</div></div>	<div><div>created_on</div><div>timestamp without time zone</div></div>	<div><div>last_login</div><div>timestamp without time zone</div></div>
1	10	maudy	ayundafaZA2020	maudy@gmail.com	2020-06-01 20:58:29.779886	2020-06-01 20:58:29.779886
2	11	gungun	febrianZA2020	gungun@gmail.com	2020-06-01 20:58:29.779886	2020-06-01 20:58:29.779886
3	12	yuma	yusufu	yumadol@gmail.com	2020-06-01 20:58:29.779886	2020-06-01 20:58:29.779886

Gambar 352 Inserted Data

Subchapter 4 – Blog App

//Todo

CRUD Operation

- ☒ Sign-up Feature
- ☒ Log-In Feature
- ☒ Create Post Feature
- ☒ Update Post Feature
- ☒ Delete Post Feature
- ☒ Get Post By ID Feature
- ☒ Get All Post

Experience to Learn

- ☒ Learn How to Use Drun (Deno Runner)
- ☒ Learn How to Make Router
- ☒ Learn How to Protect Router
- ☒ Learn How to Use JSON Web Authentication
- ☒ Learn How to Hash using BCrypt
- ☒ Learn How to Interact with PostgreSQL
- ☒ Learn How to Handling Error
- ☒ Learn How to Use Deno-nessie for Database Migration

Cek Here :

<https://github.com/gungunfebrianza/Deno-Oak-JWT-CRUD>

Chapter 6

Mastering Node.js

Subchapter 1 – Node.js

*You can never understand everything.
But, you should push yourself to understand the system*

— Ryan Dahl

Subchapter 1 – Objectives

- Memahami Apa itu **Node.js System**
 - Memahami Apa itu **I/O Scaling Problem**
 - Memahami Apa itu **Process & Thread**
 - Memahami Apa itu **Multithread**
 - Memahami Apa itu **Core Modules & libuv**
-

1. 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** :

< (C:) > Program Files > nodejs			
Name	Date modified	Type	Size
node_modules	19/11/2019 16:15	File folder	
install_tools	14/02/2020 19:03	Windows Batch File	3 KB
node	18/02/2020 4:49	Application	28.588 KB
node_etw_provider.man	14/02/2020 19:03	MAN File	9 KB
nodevars	14/02/2020 19:03	Windows Batch File	1 KB
npm	14/02/2020 19:03	File	1 KB
npm	14/02/2020 19:03	Windows Comma...	1 KB
npx	14/02/2020 19:03	File	1 KB
npx	14/02/2020 19:03	Windows Comma...	1 KB

Gambar 353 Node.js System

Terdapat beberapa komponen yaitu :

1. **node.exe**

Program untuk memulai **Javascript V8 Engine**, kita bisa menggunakannya 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* lainnya 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 :

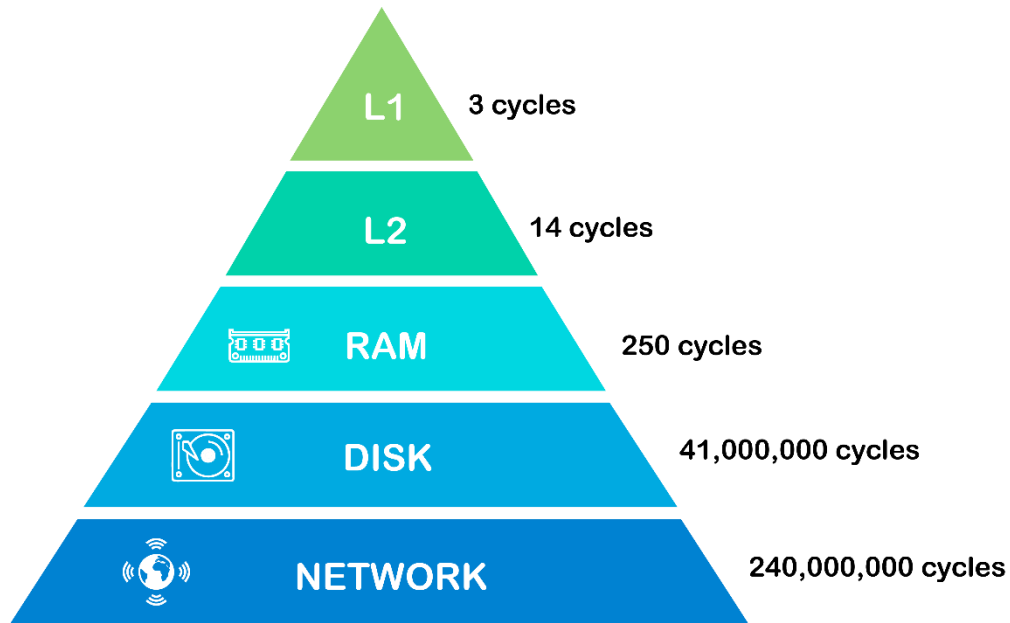
```
C:\project_x>node
Welcome to Node.js v12.16.1.
Type ".help" for more information.
> console.log("Maudy Ayunda")
Maudy Ayunda
undefined
>
```

Gambar 354 Node Virtual Machine

Jika berhasil maka anda melakukan instalasi dengan benar.

2. 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 355 I/O Scaling Problem

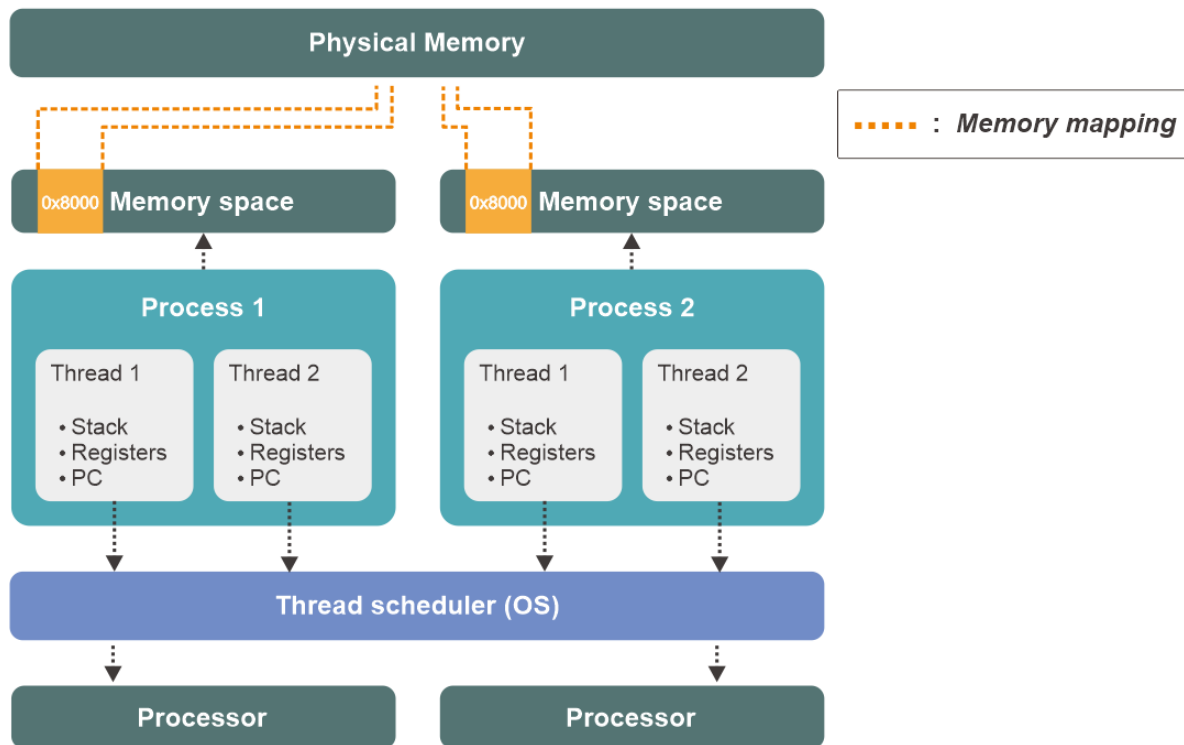
Input/Output Operation pada *disk* dan *network* memerlukan **CPU Cycles** yang sangat besar. Terminologi *CPU Cyle* mengacu pada **Clock Cycle**, kecepatan sebuah *processor* komputer. Akses pada *disk* dan *network* memerlukan waktu yang lebih lama jika dibandingkan dengan **RAM** dan *cache* dalam *CPU* (L1 & L2).

Sebagian besar *web application* akan membaca data di dalam *disk* atau data di dalam jaringan (*network*) komputer lainnya, 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*.

3. 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 356 Process & Thread

Sebuah program dalam sistem operasi yang dieksekusi akan disebut sebagai **Process**. Setiap *process* menyediakan *resources* yang dibutuhkan agar program bisa dieksekusi.

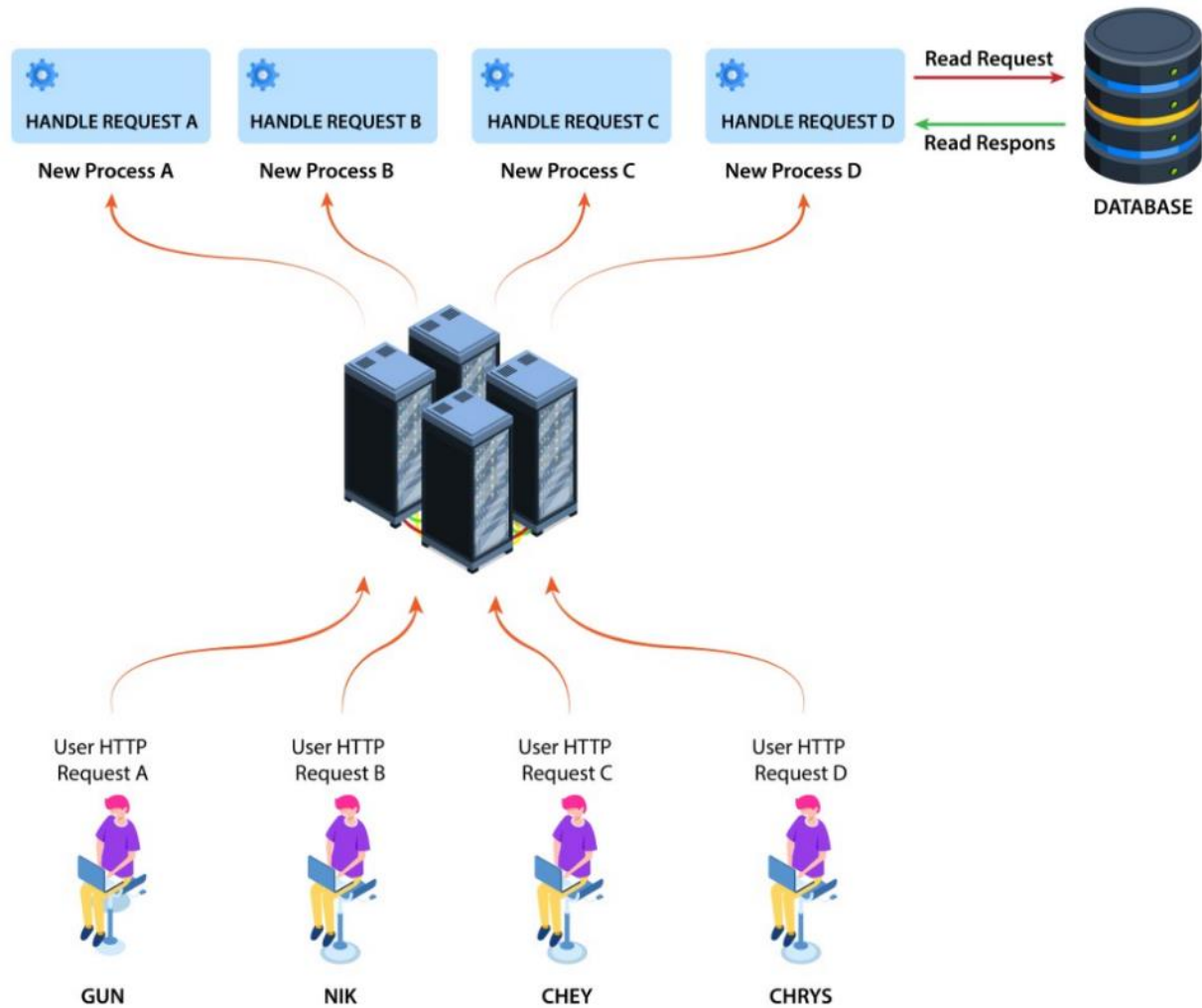
Process selalu disimpan di dalam *main memory/primary memory (RAM)*, sebuah program bisa dibangun dari beberapa *process*, masing-masing *process* dimulai dengan satu *thread tunggal* disebut dengan **primary thread**, sebuah *process* dapat memiliki beberapa *thread*.

Multithread

Sebuah *thread* dapat disebut dengan **Lightweight Process** (LWP), untuk mencapai *parallelism* sebuah *process* akan dibagi menjadi sekumpulan *thread*. Sebagai contoh :

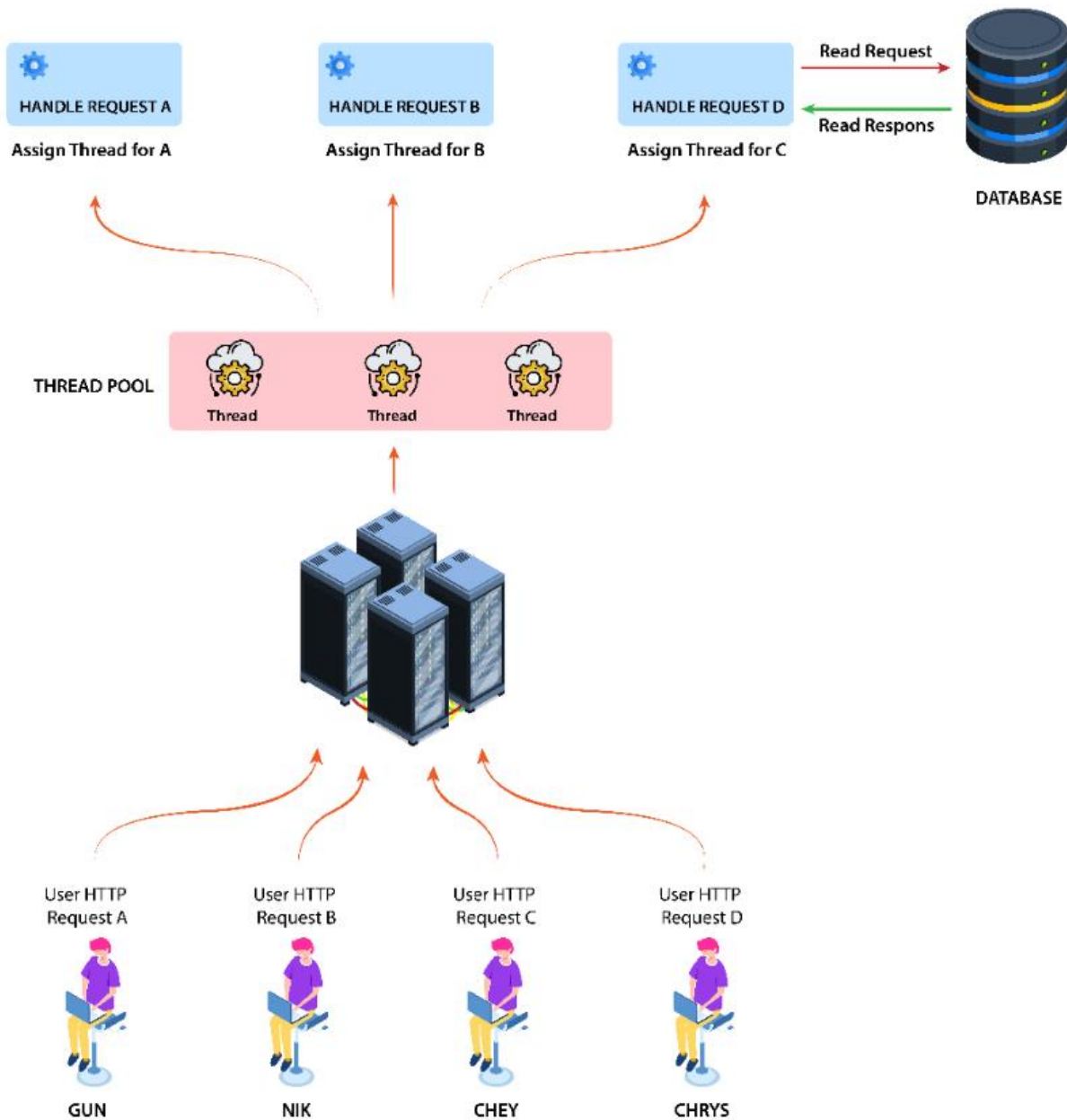
1. Pada aplikasi *browser* masing-masing *tab* yang kita gunakan memiliki *thread* masing-masing.
2. Pada aplikasi *microsoft word* terdapat *multiple thread*, 1 *thread* untuk format teks, 1 *thread* untuk memproses *input* dan 1 *thread* untuk melakukan *autosave* dengan interval waktu tertentu.

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 357 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 358 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.

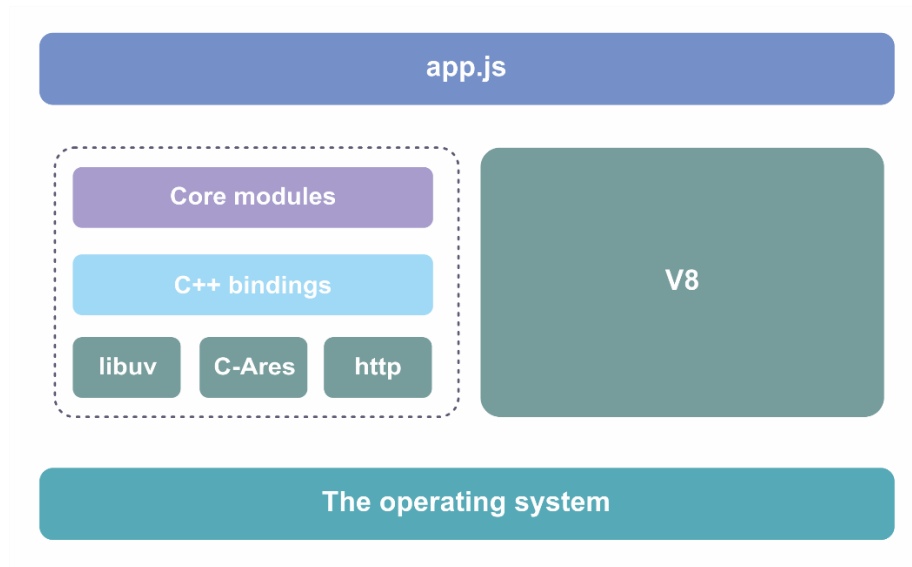
Penambahan *server* memang solusi agar tetap bisa mempertahankan *scalability* dari aplikasi namun ini justru menambah beban biaya dan tanggung jawab.

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*.

4. 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 359 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 *Bottleneck*. 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

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 pemrograman *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 pemrograman) dan
3. *Pragmatic* (implementasi dari bahasa pemrograman) .

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 lainnya 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 :

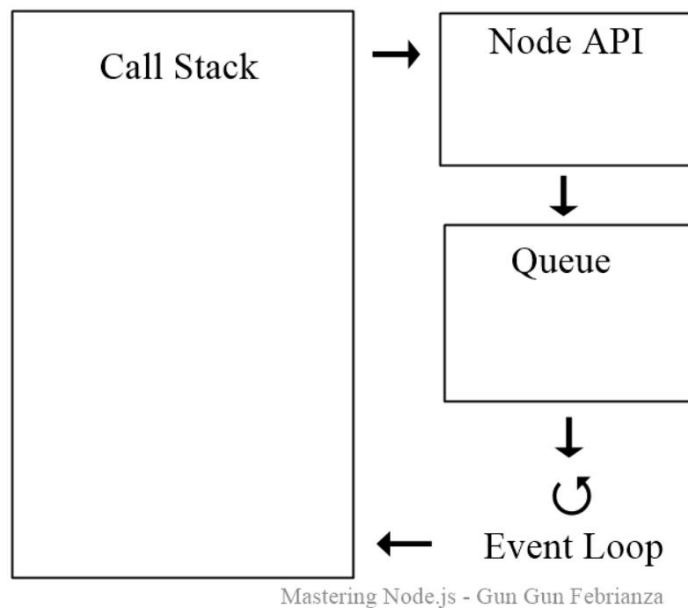
1. *Interpreter*
2. *Compiler*
3. *Memory Management*
4. *Garbage Collection*

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 dieksekusi. Cara kerjanya adalah *Last In, First Out* (LIFO), yang terakhir masuk ke dalam *stack* akan keluar lebih awal. Sehingga *Call Stack* memberikan dua layanan :

1. Kita bisa memasukkan *javascript* ke dalam *stack*.
2. Kita hanya bisa mencabut *javascript* paling atas di dalam *stack*.



Gambar 360 Single Thread & The Call Stack

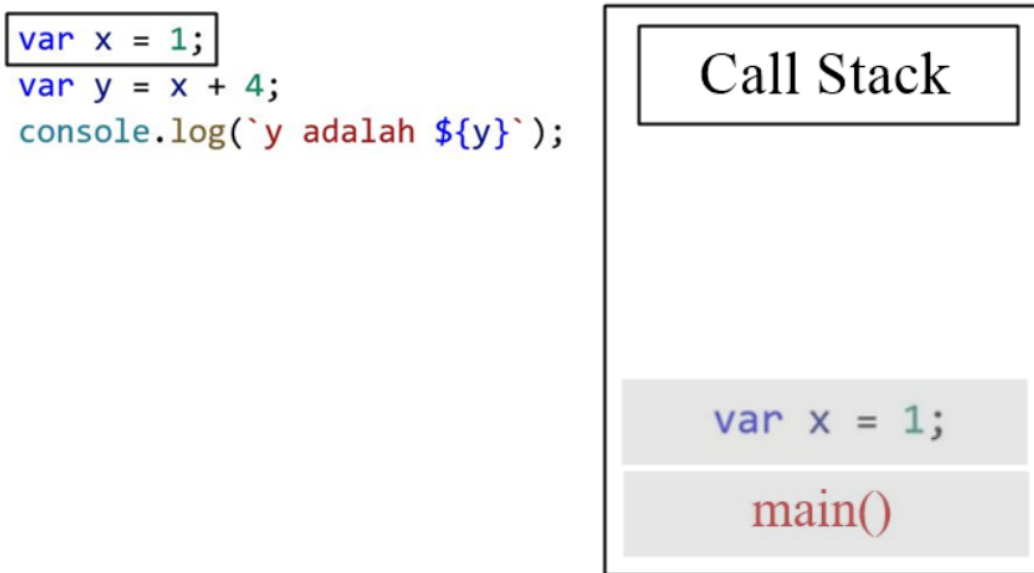
Kita akan mempelajari *call stack* dari 2 bentuk program dalam *javascript*, **Synchronous** dan **Asynchronous program**. Pada *synchronous* eksekusi kode harus dilakukan secara berurutan dan pada *asynchronous* eksekusi kode dapat dilakukan tanpa harus berurutan.

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.

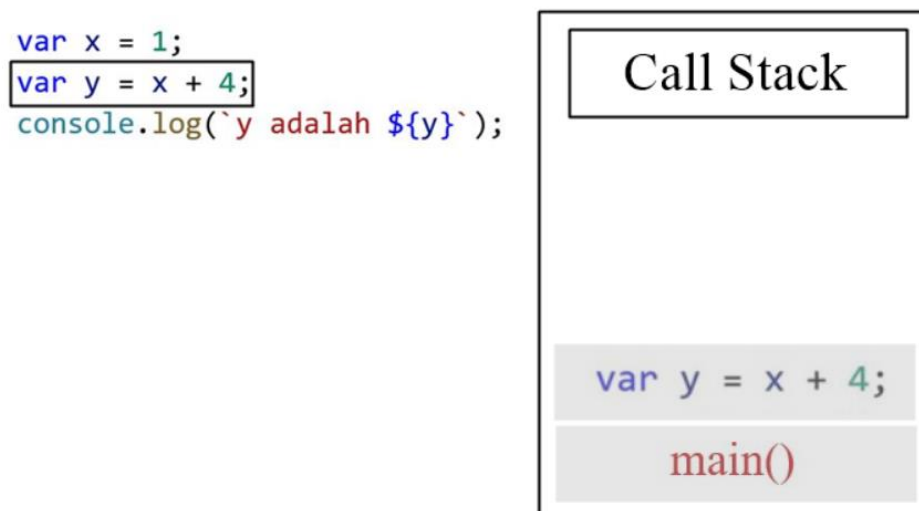
Main Function adalah perumpamaan untuk memberikan ilustrasi, jika kita perhatikan dari kode *javascript* di bawah ini. Kita membuat variabel `x` dan memberikan nilai berupa *literal* 1, ini adalah *statement code* pertama yang berjalan.



Mastering Node.js - Gun Gun Febrianza

Gambar 361 Statement Code 1 On The Call Stack

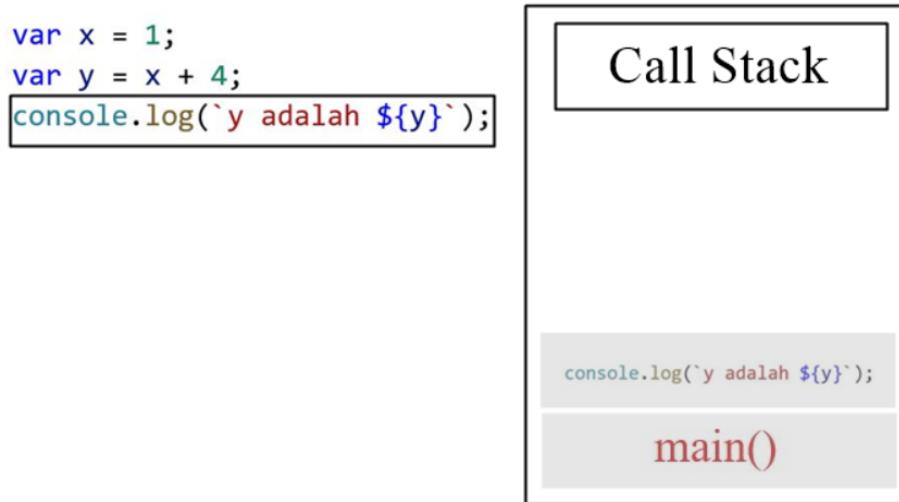
Statement tersebut masuk ke dalam *stack* setelah *main function* karena *statement* `var x = 1` dibungkus atau berada di dalam *main function*. Setelah *statement* `var x = 1` dieksekusi, *statement* tersebut berada di posisi paling atas dan akan dicabut digantikan oleh *statement code* selanjutnya.



Mastering Node.js - Gun Gun Febrianza

Gambar 362 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.



Mastering Node.js - Gun Gun Febrianza

Gambar 363 Statement Code 3 On The Call Stack

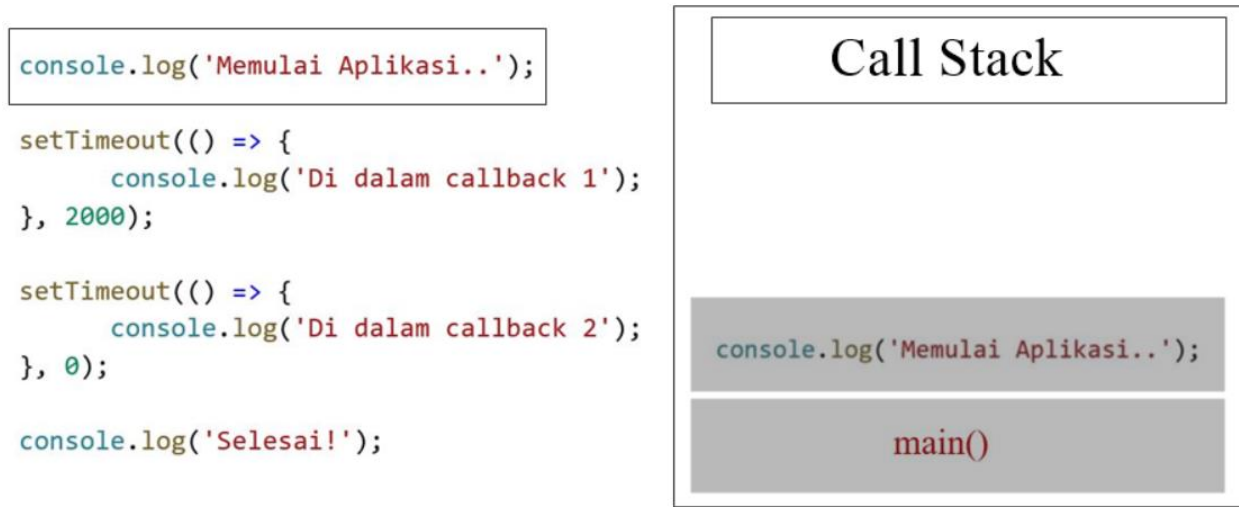
Pada fase ini *statement* yang terakhir akan menampilkan nilai dari **y**, setelah itu *statement* ini akan dicabut dari *call stack* diikuti dengan *main function* sampai *stack* menjadi kosong kembali.

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.



Mastering Node.js - Gun Gun Febrianza

Gambar 364 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**.

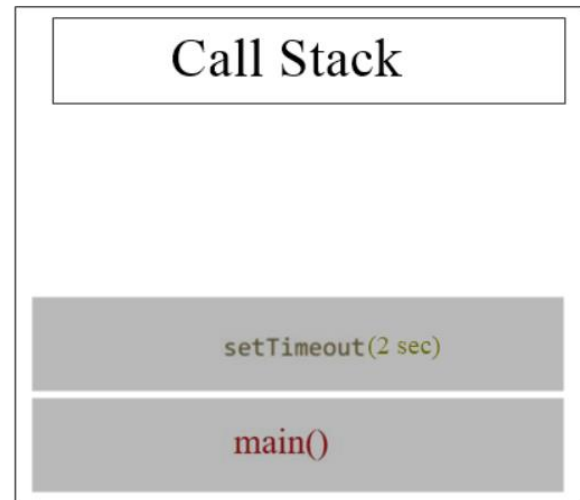
Jika diperjelas lagi *javascript Engine* dalam *Node.js* memiliki *single threaded asynchronous processing model* memiliki manfaat untuk mengatasi **concurrent request** dengan penggunaan *resources* yang lebih hemat.

```
console.log('Memulai Aplikasi..');
```

```
setTimeout(() => {  
  console.log('Di dalam callback 1');  
}, 2000);
```

```
setTimeout(() => {  
  console.log('Di dalam callback 2');  
}, 0);
```

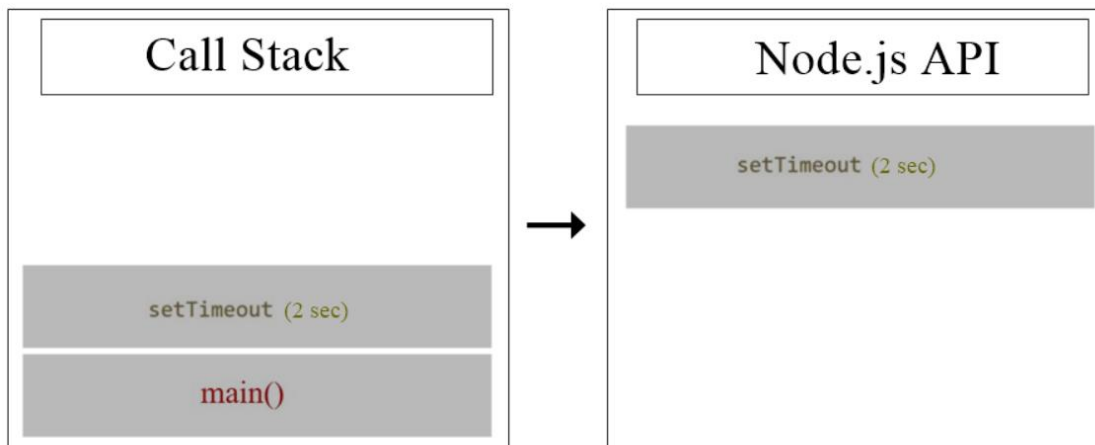
```
console.log('Selesai!');
```



Mastering Node.js - Gun Gun Febrianza

Gambar 365 First Asynchronous Code

Saat kita memanggil **setTimeout (2 sec) function** ke dalam *call stack*, maka kita juga memerintahkan untuk mendaftarkan ke dalam **node.js API**. Pada contoh kode di atas **event** adalah sebuah *event* sederhana untuk menunggu 2 detik sebelum mengeksekusi kode di dalam **callback**.

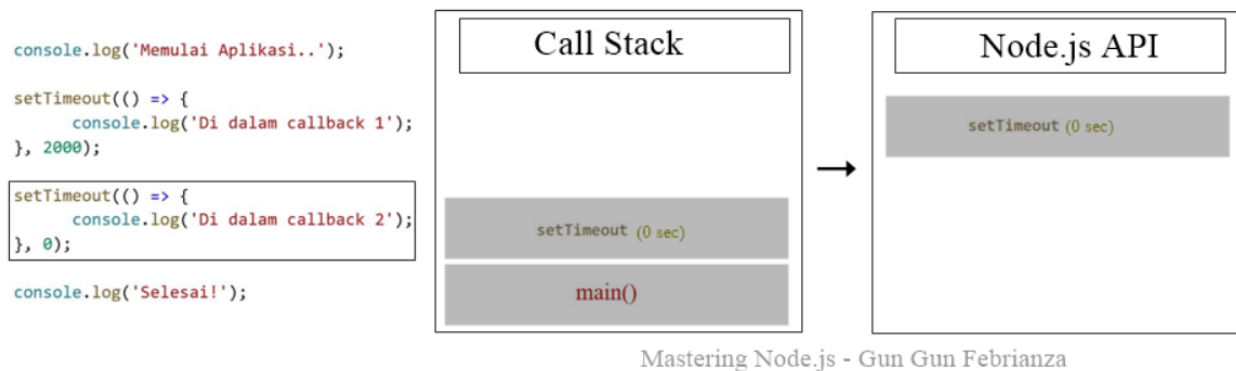


Mastering Node.js - Gun Gun Febrianza

Gambar 366 Register Event

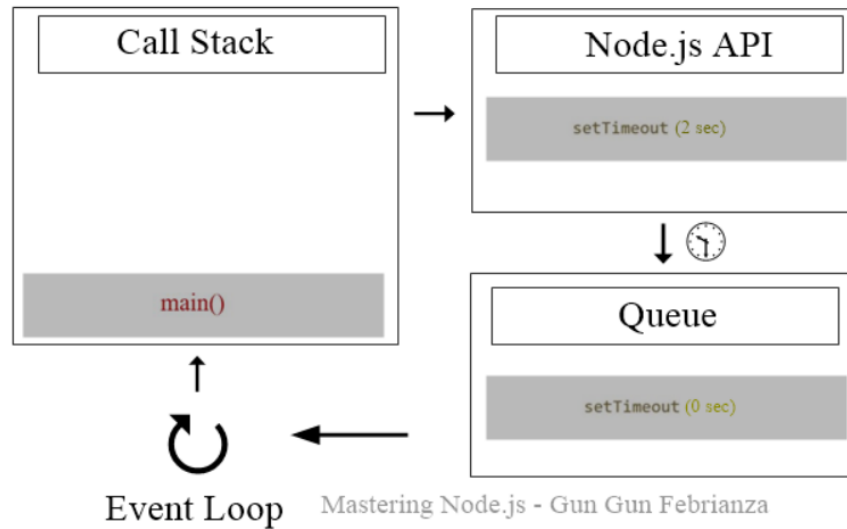
Setelah mendaftarkan *event*, *Call Stack* terus berlanjut, dan fungsi **setTimeout** terus berjalan menghitung waktu mundur (*counting down*). Mungkin anda berpikir kita akan menunggu selama dua detik di dalam *call stack*, namun ternyata tidak.

Call stack memang bisa melaksanakan satu tugas dalam satu waktu namun bukan berarti *call stack* tidak bisa melanjutkan pekerjaannya 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**.



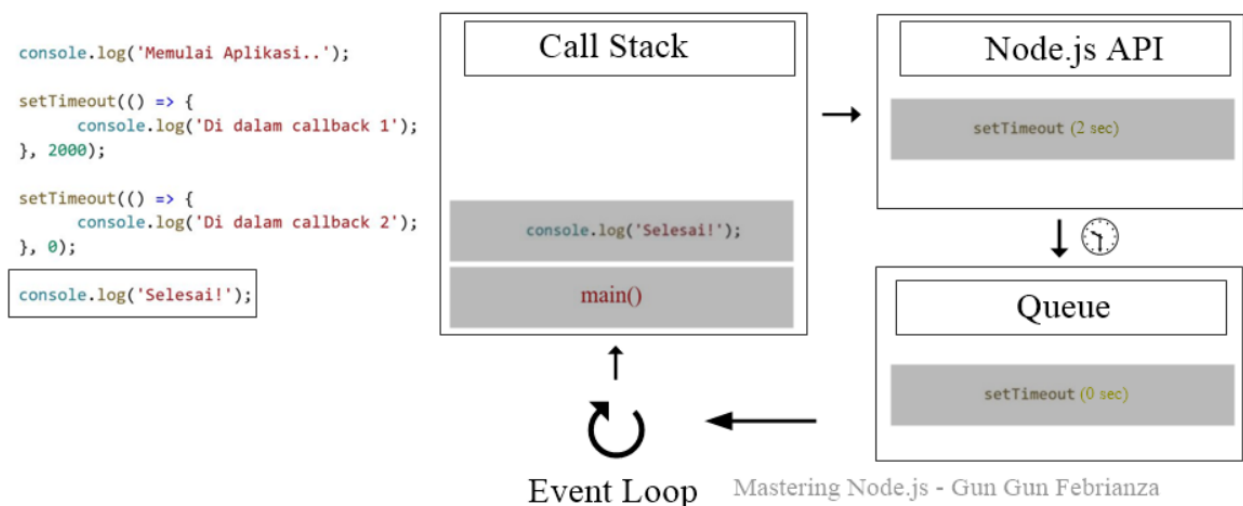
Gambar 367 Register Last Event

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 368 Queue Activity

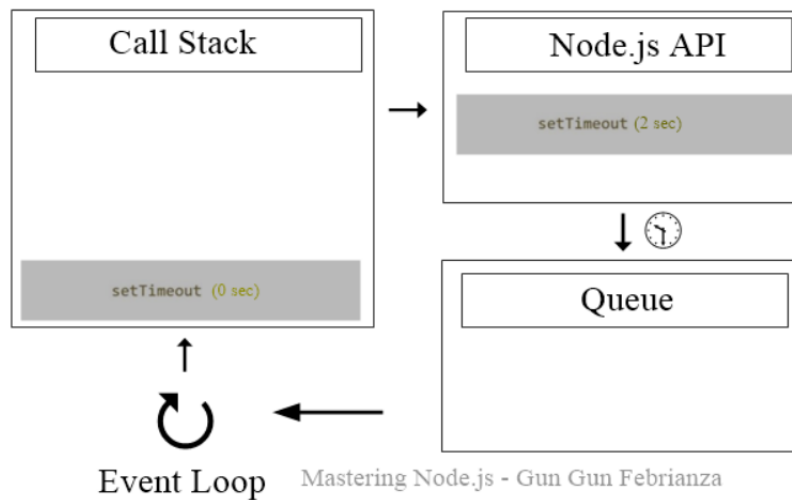
Queue adalah tempat dimana *callback function* yang kita buat menunggu sampai **call stack** menjadi kosong. **Event Loop** akan melihat *Call Stack*, jika *call stack* masih belum kosong kode maka *callback function* dalam *queue* belum bisa dieksekusi. Masih tersisa `console.log("selesai")`, *statement* tersebut akan dicabut dalam *stack* diikuti `main function()`.



Gambar 369 Last Asynchronous Statement

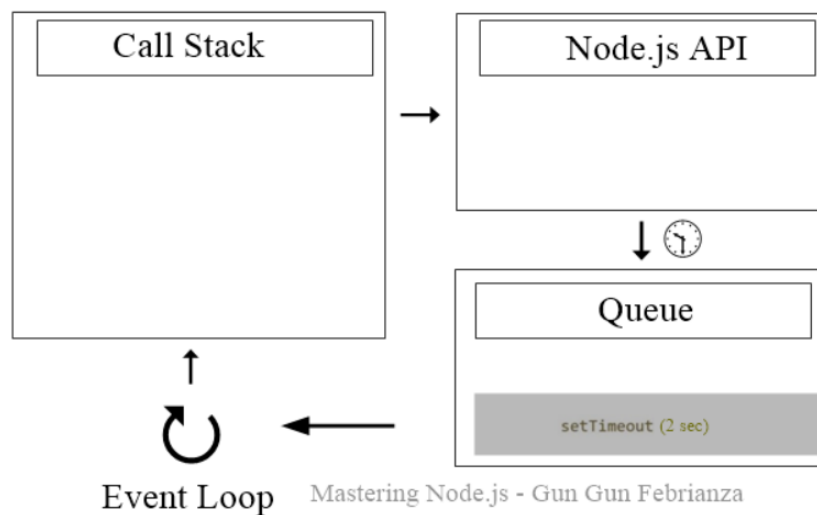
Event Loops

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* :



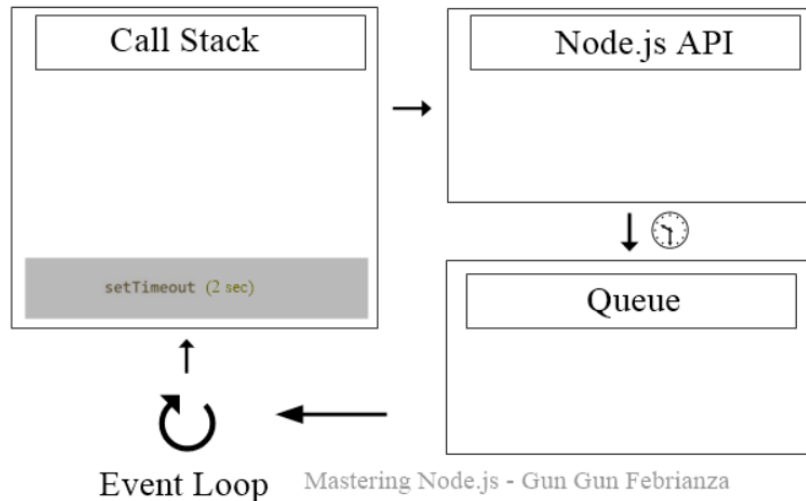
Gambar 370 Extract Callback Function

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 371 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 372 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.



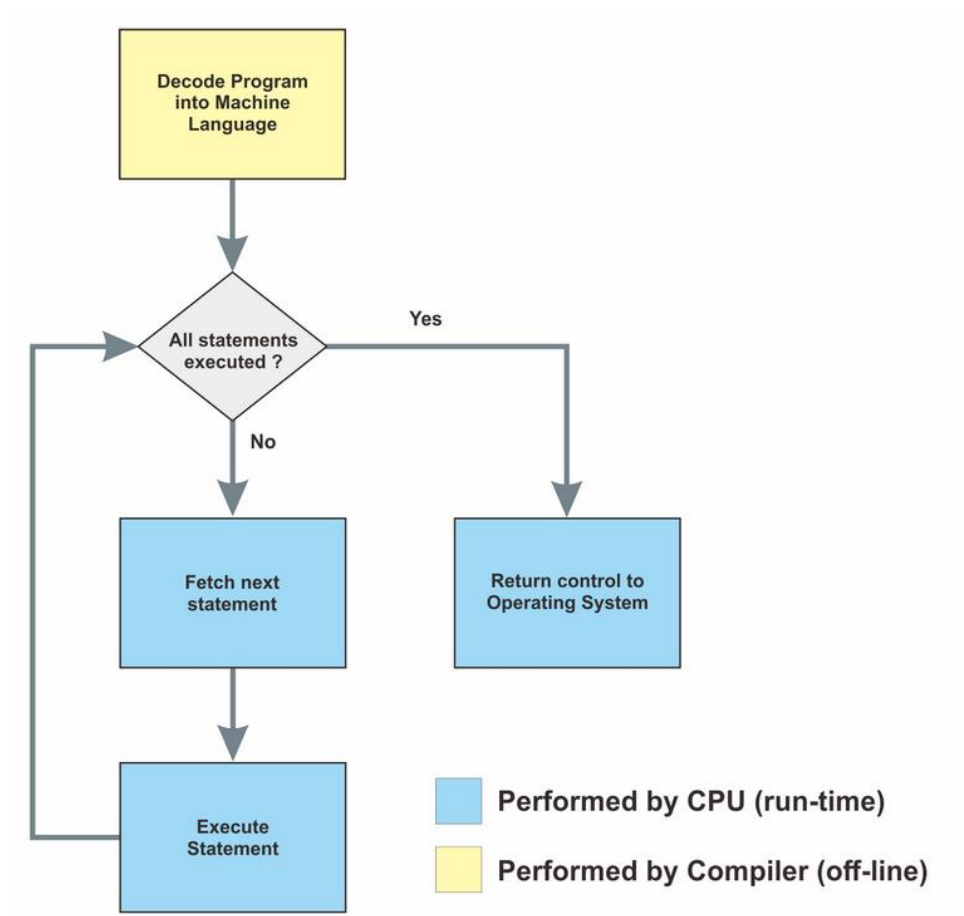
Gambar 373 Interpreter Illustration

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.

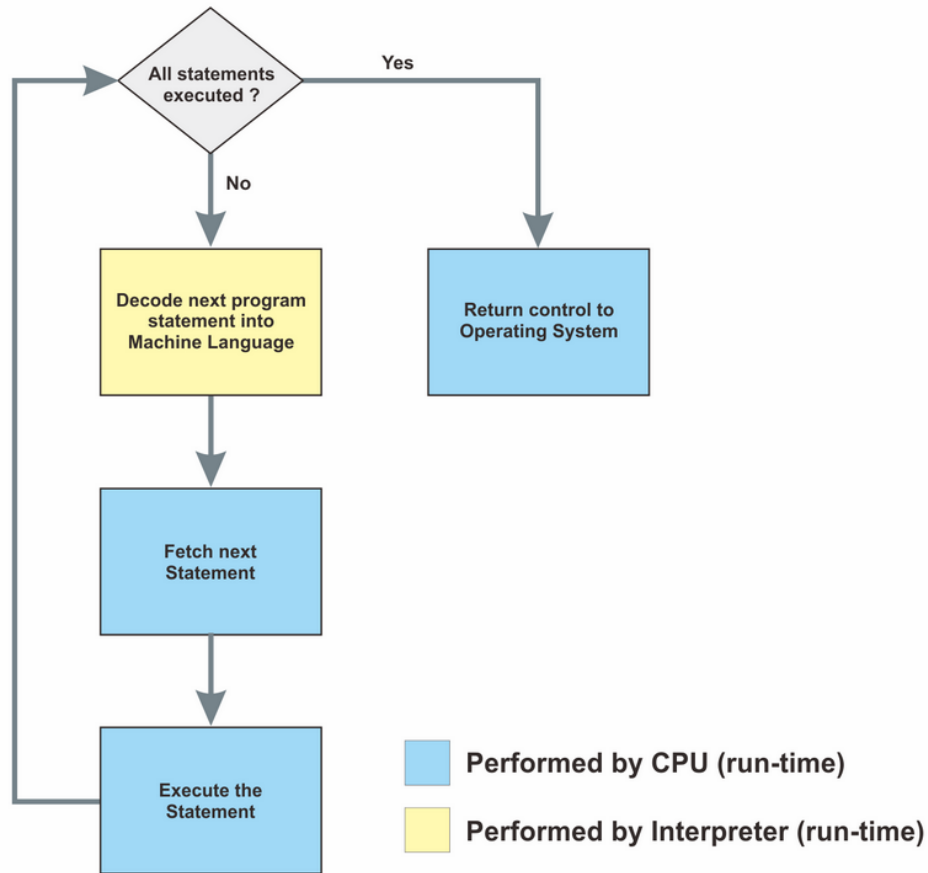
Kompiler menterjemahkan seluruh sumber kode sekaligus dalam satu fase kompilasi, hasil program yang telah dikompilasi dieksekusi dengan mode *fetch-execute cycle*.

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 374 Proses eksekusi pada kompiler



Gambar 375 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 11 Perbedaan Kompiler dan Interpreter

<i>Compiler</i>	<i>Interpreter</i>
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 <i>intermediate object</i> .	Memori lebih efisien karena tidak memproduksi <i>intermediate object</i> .
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 :

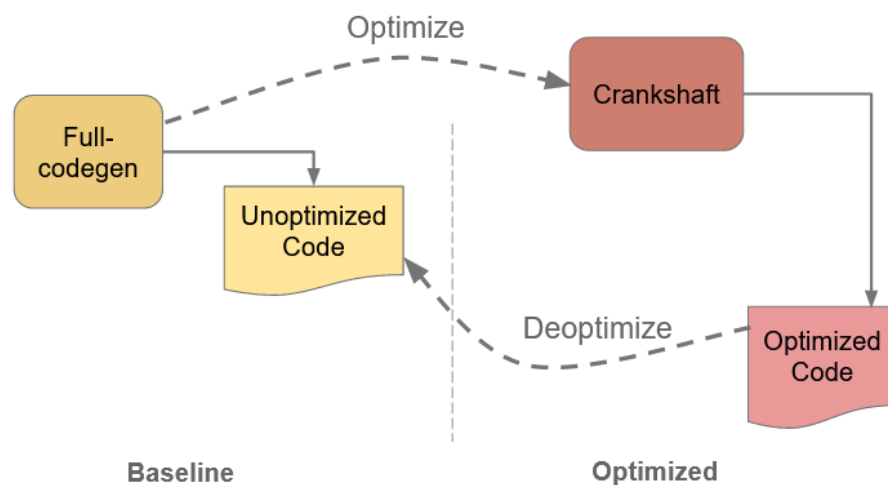
1. **Full-codegen** (*Baseline Compiler*)— Kompiler yang dapat memproduksi kode mesin yang belum dioptimasi (*non-optimized machine code*).
2. **Crankshaft** (*Optimized Compiler*)— Kompiler yang dapat memproduksi kode mesin yang telah dioptimasi (*optimized machine code*).

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 menjalankan **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-compile* secara dinamis menggunakan kompiler yang didesain untuk melakukan optimasi (*optimized compiler*) agar bisa memproduksi *machine code* yang mampu memberikan *performance* sampai puncak (*peak performance*).



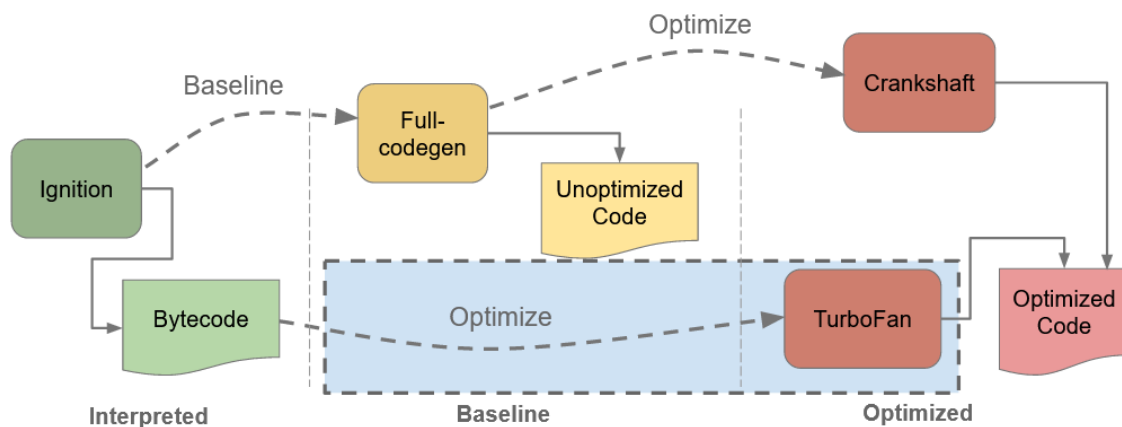
Gambar 376 V8 Compilation Pipeline

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.

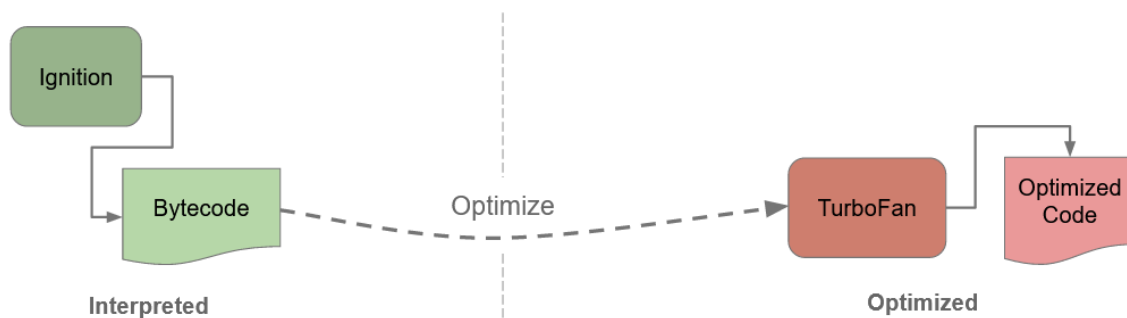
Ignition & Turbofan

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**.



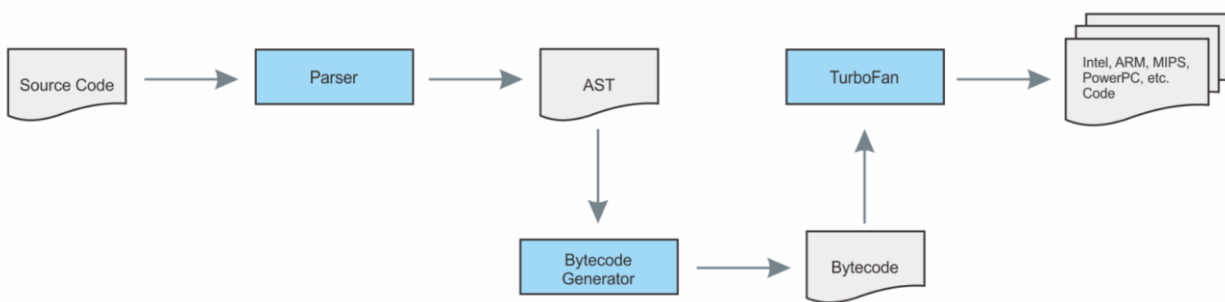
Gambar 377 Baseline Compiler Removed

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 *TurboFan Compiler*.



Gambar 378 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 379 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**.

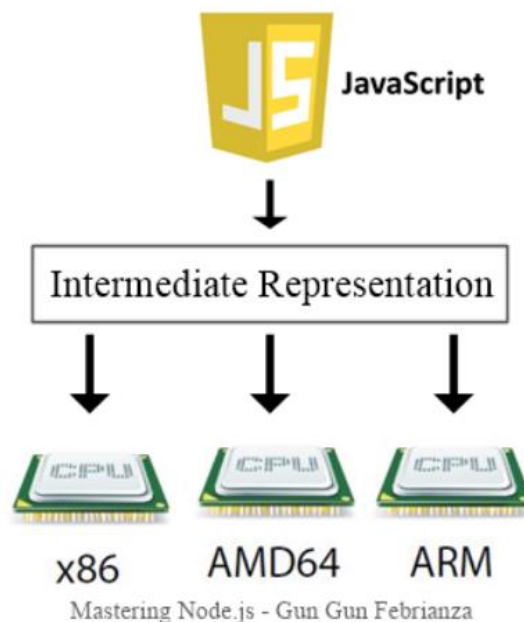
Ignition sebagai program *interpreter*, akan menggunakan *syntax tree* tersebut untuk memproduksi *bytecode*. **TurboFan** sebagai *optimized compiler*, akan mengambil *bytecode* dan memproduksi kode mesin yang telah di optimasi (*optimized machine code*).

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 380 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 lainnya yang anda bisa lihat di dokumentasi *node.js*.

Tim pengembang V8 juga harus memelihara lebih dari 10 ribu baris kode untuk masing-masing *chip architecture*, setiap kali pengembangan baru dibuat semuanya harus di *port*

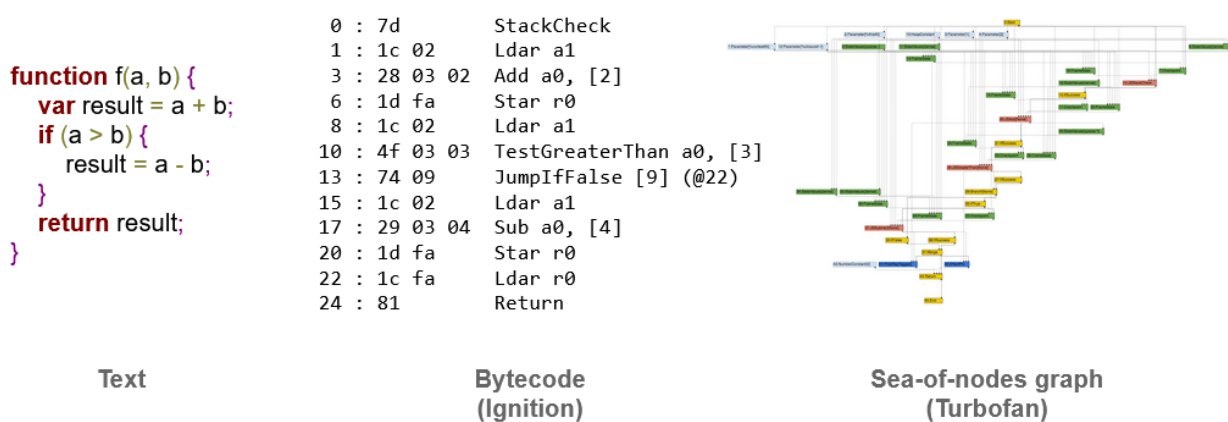
ke *architecture* yang didukung. Artinya setiap **bahasa assembly** untuk setiap *architecture* harus disediakan ketika pengembangan baru dibuat. Benar benar merepotkan ya?

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 381 Javascript Code, Bytecode, & TurboFan

Tertarik untuk mengembangkan bahasa pemrograman 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*). Eksekusi 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 pemrogramannya 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 mengidentifikasi 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* :



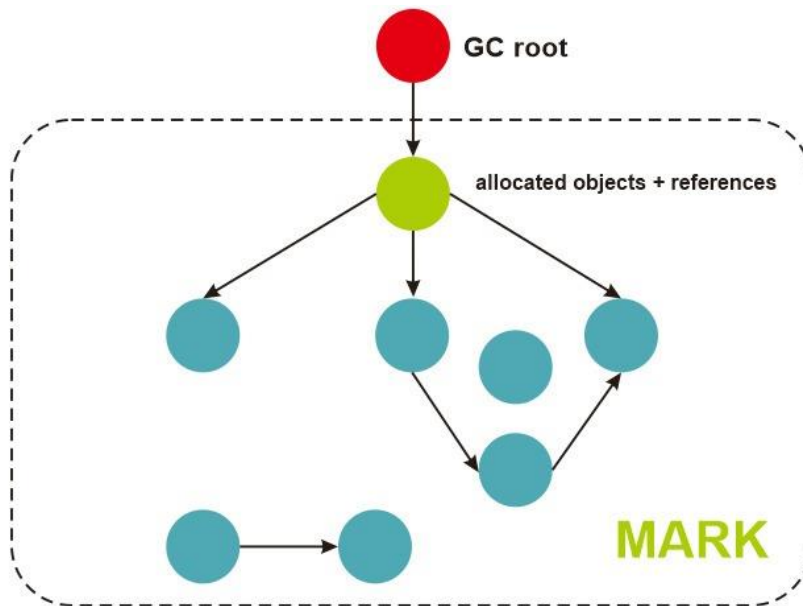
Gambar 382 Window Object

Pada *node.js* melalui *command prompt* ketik saja langsung *global* :

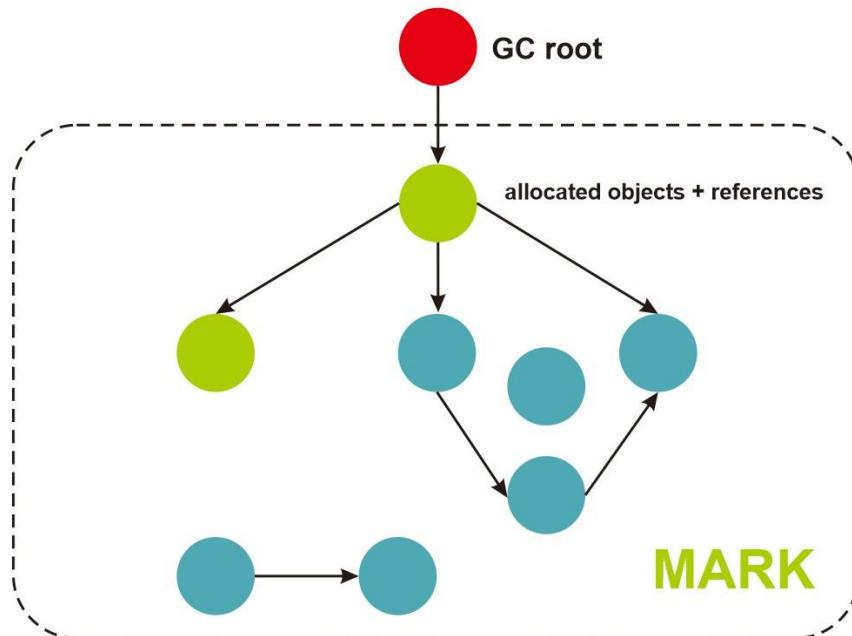

```
C:\Users\Gun Gun Febrianza>node
> console.log('hello')
hello
undefined
> global
Object [global] {
  DTRACE_NET_SERVER_CONNECTION: [Function],
  DTRACE_NET_STREAM_END: [Function],
  DTRACE_HTTP_SERVER_REQUEST: [Function],
  DTRACE_HTTP_SERVER_RESPONSE: [Function],
  DTRACE_HTTP_CLIENT_REQUEST: [Function],
  DTRACE_HTTP_CLIENT_RESPONSE: [Function],
  global: [Circular],
  process:
    process {
      title: 'Command Prompt - node',
      version: 'v11.13.0',
      versions:
        { node: '11.13.0',
          v8: '7.0.276.38-node.18',
          uv: '1.27.0',
          zlib: '1.2.11',
          ...
        }
    }
}
```

Gambar 383 Global Object

Pada gambar di bawah ini terdapat visualisasi bagaimana cara kerja *garbage collector* dalam melacak *object* yang *unreachable*.



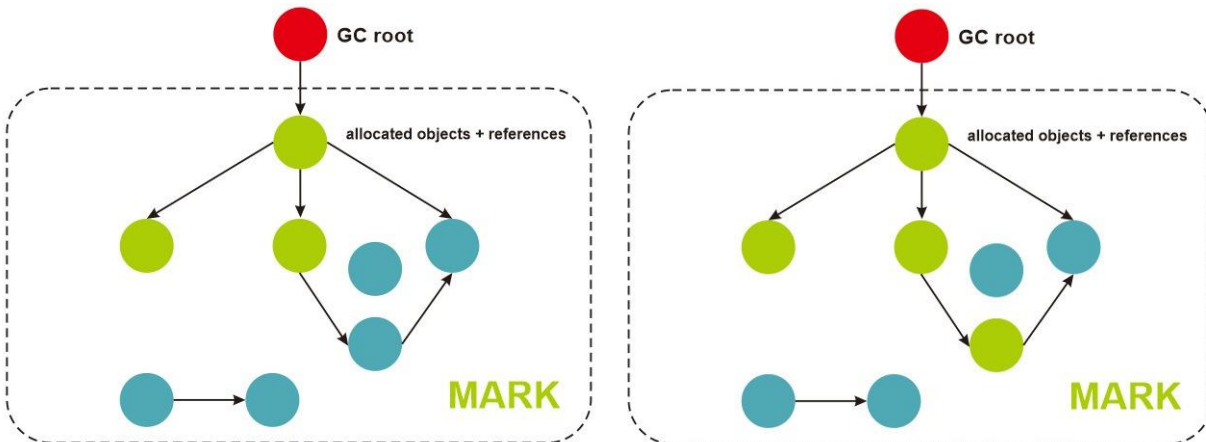
Gambar 384 Mark Phase 1



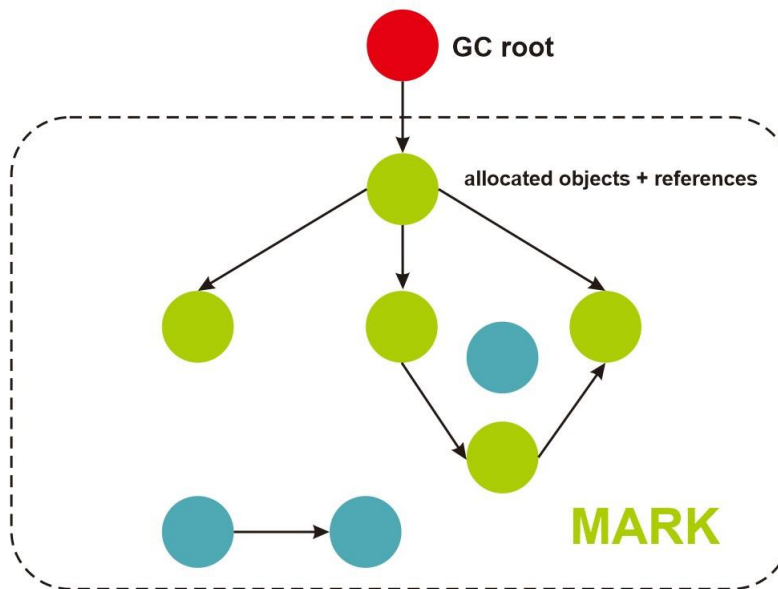
Gambar 385 Mark Phase 2

Setiap *root* akan diinspeksi sampai ke *children* dan menandainya sebagai *object active*.

Seluruh *children* akan di tandai sebagai *object active*.

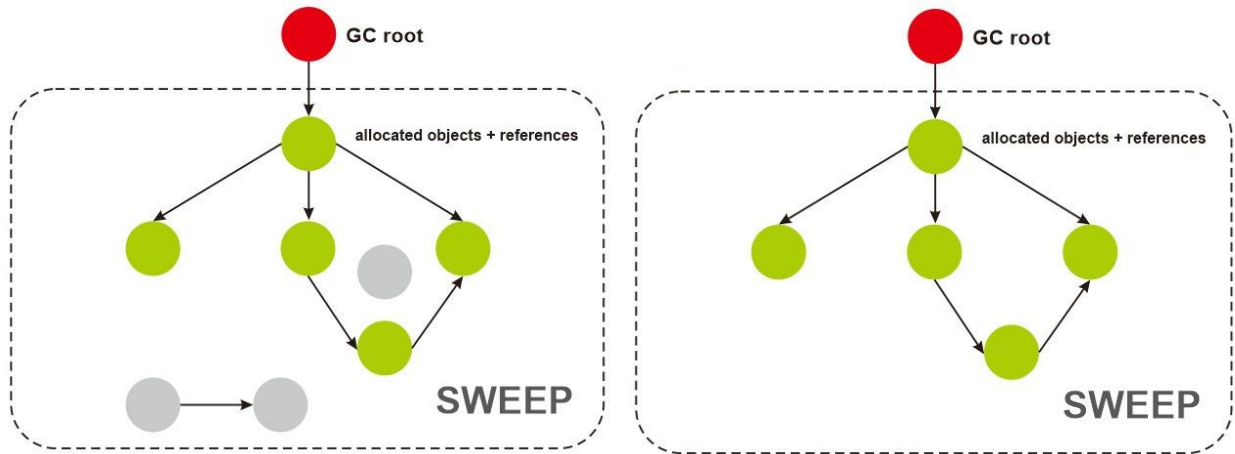


Gambar 386 Mark Phase 3



Gambar 387 Mark Phase 2

Seluruh *object* yang tidak terjangkau (*unreachable*) akan dibersihkan, memori dibebaskan dan dikembalikan lagi pada sistem operasi.



Gambar 388 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 menjadi 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 Publish **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 :

```
-Mastering-Node.js\S4.Node.jsApplication>node 1.hello  
Hello World
```

Gambar 389 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", ".clear", ".editor"];
```

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 :

```
Press ^C to abort current expression, ^D to exit the repl
> .load 2.list-command.js
  let node_command = [".help", ".load", ".save", ".break", ".exit"];
undefined
```

Gambar 390 .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 :

```
> node_command.forEach(command => console.log(command))
.help
.load
.save
.break
.exit
undefined
> []
```

Gambar 391 forEach Method Result

3. *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* lainnya bisa ikut menggunakan dan mengembangkannya.

Saat kita telah bergantung pada kode yang dibuat oleh seorang *developer*, *package manager* akan mempermudah *developer* lainnya 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/>

4. 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 12 npm commands

<i>npm command</i>	<i>Description</i>	<i>Example</i>
npm init	Inisialisasi <i>node.js application</i> dan membuat <i>file package.json</i> .	npm init
npm install	Memasang sebuah <i>module</i> dalam sekup lokal	npm install typescript
npm install -g	Memasang sebuah <i>module</i> dalam sekup global	npm install typescript -g
npm search	Mencari sebuah <i>modules node.js</i>	npm search typescript
npm remove	Menghapus <i>module</i> yang telah dipasang	npm remove typescript
npm update	Memperbaharui <i>module</i> yang telah dipasang	npm update typescript -g
npm list	Menampilkan daftar <i>modules</i> yang telah dipasang	npm list
npm view	Menampilkan informasi <i>module</i> secara detail.	npm view typescript
npm start	Menjalankan aplikasi <i>node.js</i> yang telah dikonfigurasi di dalam <i>package.json</i>	npm start
npm stop	Menghentikan aplikasi <i>node.js</i> yang sedang berjalan	npm stop
npm publish	Publikasikan <i>module</i> ke <i>Node Package Registry</i> .	npm publish mymodule

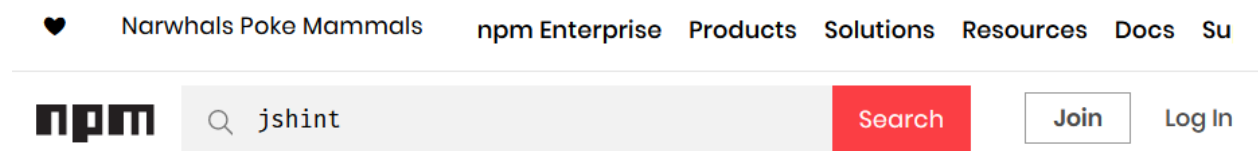
npm unpublish	Batalkan <i>module</i> yang telah dipublikasikan.	npm unpublish mymodule
npm docs	Membaca dokumentasi dari sebuah <i>module</i> .	npm docs typescript

5. Node Package Registry

Jika kita ingin mengetahui *package* apa saja yang tersedia di dalam *node.js* kita bisa mengunjungi *Node Package Registry* di :

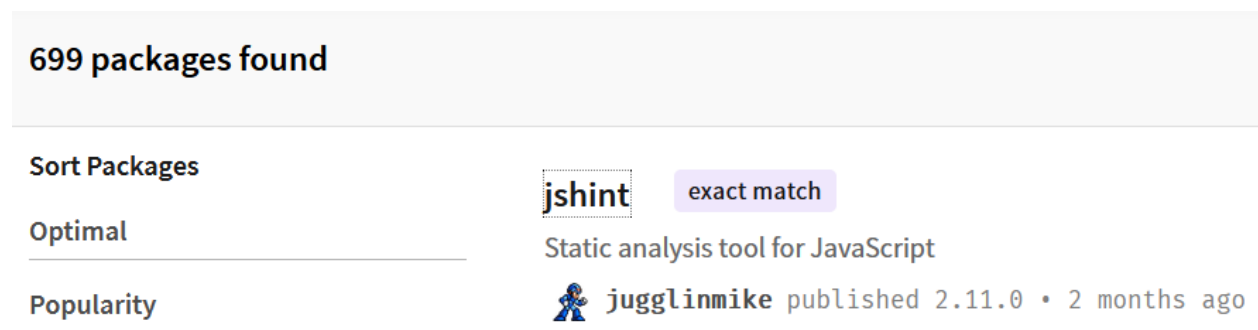
<https://www.npmjs.com/>

Jika kita ingin mencari sebuah *node.js modules*, cukup ketik nama *node.js modules* yang ingin diketahui seperti pada gambar di bawah ini :



Gambar 392 Npmjs.com

Anda akan menemukan *module* tersebut dalam urutan pertama :



Gambar 393 jshint module

Setelah anda mendapatkan *node.js module jshint* selanjut anda bisa melihat lebih detail informasi dari *node.js module* tersebut :

jshint
2.10.2 • Public • Published 6 months ago

Readme
8 Dependencies
771 Dependents
86 Versions

JSHint, A Static Code Analysis Tool for JavaScript

[Use it online • Docs • FAQ • Install • Contribute • Blog • Twitter]

npm v2.10.2
Linux build passing
Windows build passing
dependencies out of date
dev dependencies out of date
coverage 100%

JSHint is a community-driven tool that detects errors and potential problems in JavaScript code. Since JSHint is so flexible, you can easily adjust it in the environment you expect your code to execute. JSHint is open source and will always stay this way.

install

```
> npm i jshint
```

↓ weekly downloads

501,627

version	license
2.10.2	(MIT AND JSON)
open issues	pull requests
361	56

Gambar 394 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 populer dan klasifikasinya :

Popular libraries

lodash
request
async
chalk
express
bluebird
commander
debug
underscore
react
moment
mkdirp

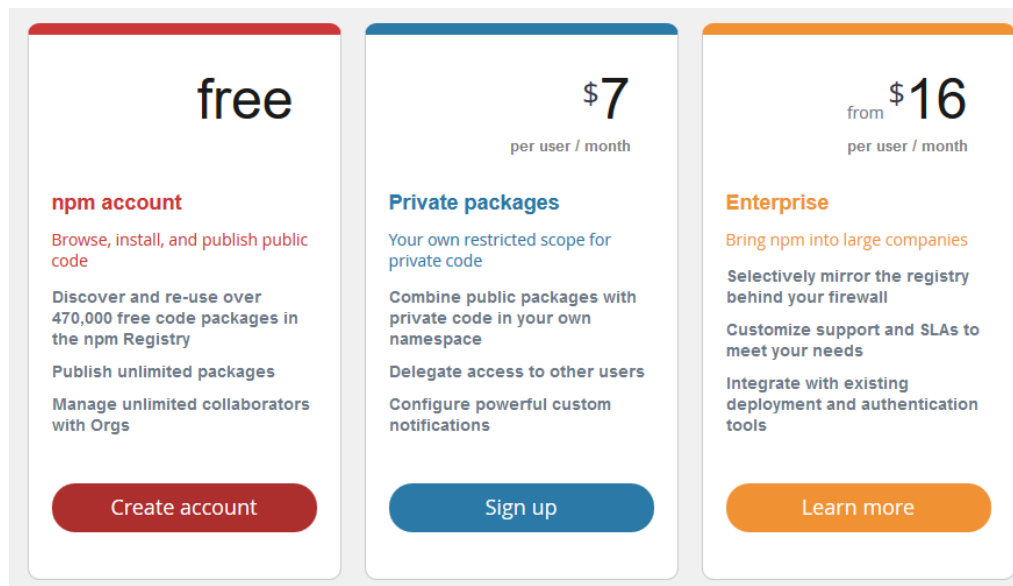
Discover packages

IoT
mobile
front end
backend
robotics
css
testing
cli
documentation
math
coverage
frameworks

Gambar 395 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 396 Buat Akun di Node Package Registry

Isi formulir pendaftaran dan konfirmasi pendaftaran akan dilakukan melalui *email*.

Name

Public Email

Username

username

https://www.npmjs.com/~username

Password

In order to protect your account, make sure your password:

- Is longer than 7 characters
- Does not match or significantly contain your username, e.g. do not use "username123".
- Is not a member of this [list of common passwords](#)

☒ Sign up for the **npm Weekly**

☐ I agree to the **End User License Agreement** and the **Privacy Policy**.

Your email address will show on your profile page, but npm will never share or sell it.

Create an Account

Gambar 397 Form Register npmjs

6. Create Node.js package

Setiap kali kita membuat aplikasi dengan *node.js*, file *package.json* akan selalu dibutuhkan untuk manajemen *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 398 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 399 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 :

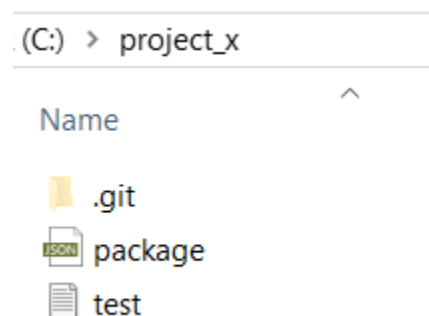
```
About to write to C:\project_x\package.json:

{
  "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"
}

Is this OK? (yes)
```

Gambar 400 Package.json file

Ketik *yes* kemudian tekan tombol *enter*, maka di dalam *directory repository* **project_x** akan muncul *file package.json* :



Gambar 401 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 13 Directive on Package.json

Directive	Description	Example
<i>name</i>	Nama untuk <i>package</i> yang dibuat, wajib diisi dengan huruf kecil semua tanpa spasi dan maksimal tidak lebih dari 214 karakter.	"name": "name-must-unique"
<i>version</i>	Versi dari <i>package</i> 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"

<i>description</i>	Penjelasan dari <i>package</i> yang dibuat, masukan beberapa <i>keyword</i> yang dapat mempermudah orang lain mencari <i>package</i> anda.	<code>"description": "new package"</code>
<i>main</i>	Sebuah <i>entrypoint</i> untuk aplikasi yang dibuat, bisa berupa program biner atau <i>javascript file</i> .	
<i>author</i>	Nama dari pembuat <i>package</i> , dibuat untuk satu orang saja.	<code>"author": "Gun"</code>
<i>contributors</i>	Nama untuk para <i>contributor</i> yang ikut mengembangkan <i>package</i> , dibuat untuk lebih dari satu orang.	<code>{ "name" : "nikolaj" , "email" : "nikolaj@marketkoin.com" , "url" : "https:// marketkoin.com/" }</code>
<i>dependencies</i>	<i>Modules</i> dan versi dari <i>module</i> yang bergantung pada <i>package</i> yang dibuat. Kita bisa menggunakan notasi <i>wilcard</i> dengan * dan x	<code>{ "dependencies" : { "jshint" : "1.0.0" , "cli" : "~1.2.3" , "openssl" : "2.x" } }</code>
<i>license</i>	Lisensi yang akan digunakan untuk <i>package</i> .	<code>"license": "MIT"</code>
<i>bin</i>	Program biner yang ingin ikut tertanam bersama <i>package</i> yang dibuat.	
<i>repository</i>	Tipe <i>repository</i> yang digunakan dan informasi <i>package location</i> .	

<i>homepage</i>	Alamat pengenalan untuk <i>package</i> 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 lainnya yang dibutuhkan oleh *node package manager (NPM)* untuk melakukan instalasi dan publikasi *node.js module* agar bisa digunakan oleh pengguna lainnya.

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* lainnya 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* :

(C:) > project_x

Name	Date modified	Type	Size
.git	09/03/2020 5:34	File folder	
node_modules	09/03/2020 5:33	File folder	
package	09/03/2020 5:33	JSON File	1 KB
package-lock	09/03/2020 5:33	JSON File	1 KB
test	09/03/2020 4:40	Text Document	1 KB

Gambar 402 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 di halaman 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;
```

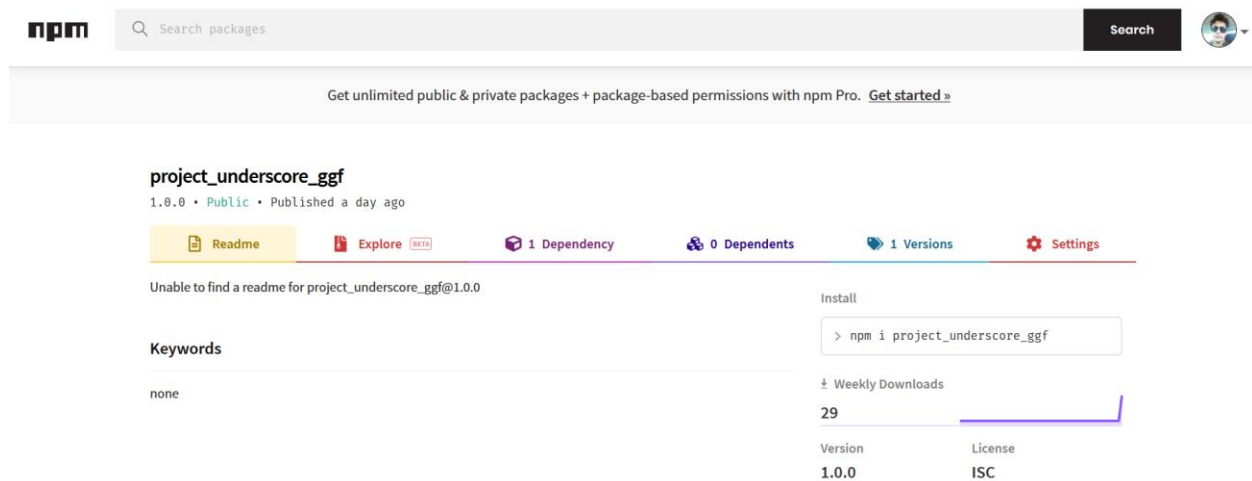
Pada kode di atas kita memanfaatkan *method first* yang dimiliki oleh *library underscore* untuk mendapatkan *index* pertama dari suatu *array* dan *method last* untuk mendapatkan *index* terakhir dari suatu *array*.

Kemudian kita membuat *function* **getFirstIndex** & **getLastIndex** yang akan di *export* sebagai *node.js module*.

7. 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 403 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 404 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 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
```

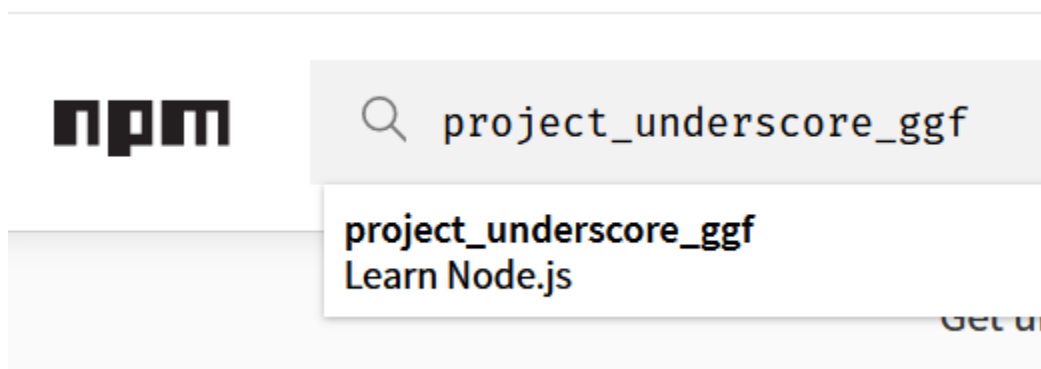
Gambar 405 Publish Module Failed

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 406 Publish Module Success

Anda bisa mencarinya di kolom pencarian yang disediakan oleh **npmjs** :



Gambar 407 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 instalasi *package* yang telah kita buat.

project_underscore_ggf

1.0.0 • Public • Published a minute ago

[Readme](#)[Explore](#) BETA[1 Dependency](#)[0 Dependents](#)[1 Versions](#)[Settings](#)

Unable to find a readme for project_underscore_ggf@1.0.0

Keywords

none

Install

```
> npm i project_underscore_ggf
```

Version

1.0.0

License

ISC

Gambar 408 Package Page Details

Untuk membatalkan *module* yang telah dipublikasikan anda bisa menggunakan perintah di bawah ini :

```
npm unpublish
```


8. 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 melakukannya 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
```

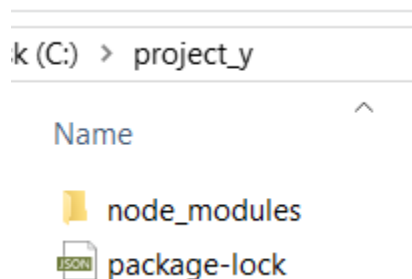
Selanjutnya masuk kedalam *directory* tersebut :

```
cd project_y
```

Selanjutnya *install module* yang telah kita *publish* :

```
npm i project_underscore_ggf
```

Pastikan setelah melakukan instalasi terdapat *folder node_modules* :



Gambar 409 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 :

```
c:\project_y>node index  
5  
1
```

Gambar 410 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.*

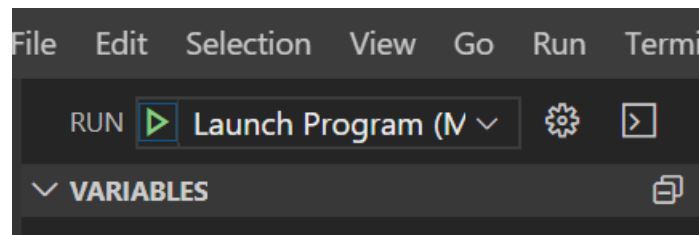
Arahkan *cursor mouse* anda pada baris kode nomor 6 sampai muncul ikon bulat berwarna merah, jika sudah klik baris kode tersebut sampai ikon bulat berwarna merah muncul seperti pada gambar di bawah ini :

```
JS 8.debugging.js X
Book-Mastering-Backend > Chapter5-Mastering-Node.js > S4.Node.jsApplication > JS 8.deb
You, a few seconds ago | 1 author (You)
1  const book = {
2    title: "Belajar dengan Jenius AWS & Node.js",
3    price: 80000
4  };
5  const discount = 0.2;
6  const discountPrice = book.price * discount;
7  console.log(`Harga buku : ${book.price - discountPrice}`);
8
```

Gambar 411 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 412 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 :

```
5  const discount = 0.2;
6  const discountPrice = book.price * discount;
7  console.log(`Harga buku : ${book.price - discountPrice}`);
8
```

Gambar 413 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 :

```
price: 80000
};
const discount = 0.2;
const discountPrice = book.price * discount;
console.log(`Harga buku : ${book.price - discountPrice}`);
```

Gambar 414 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 :

```
6  const discountPrice = book.price * discount;
7  console.log(`Harga buku : ${book.price - discountPrice}`);
8
```

Gambar 415 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.

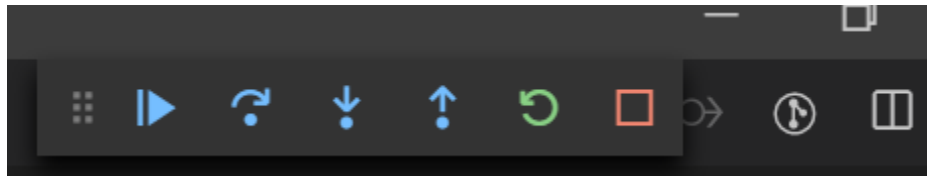
```

4   };
5   const discount = 160000.2;
6   const discountPrice = book.price * discount;
7   console.log(`Harga buku : ${book.price - discountPrice}`);
8

```

Gambar 416 Next Breakpoint

Selain itu ada dapat melihat terdapat *Debugger Panel* pada *visual studio code* :



Gambar 417 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 418 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 :

```
debug> n
break in debugging.js:5
  3 };
  4 person.age = 25;
> 5 person.name = 'Vestorp';
  6 console.log(person);
  7
debug> repl
Press Ctrl + C to leave debug repl
> |
```

Gambar 419 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* :

```
Press Ctrl + C to leave debug repl
> var a = 1 + 3
undefined
> |
```

Gambar 420 Test Expression

Kita juga dapat membaca variabel **person** di dalam REPL :

```
> var a = 1 + 3
undefined
> person
{ name: 'Nikolaj', age: 25 }
> 
```

Gambar 421 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 422 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** :

```
debug> repl
Press Ctrl + C to leave debug repl
> person.name
'Nikolaj'
debug> n
break in debugging.js:7
   5 debugger;
   6 person.name = 'Vestorp';
> 7 console.log(person);
   8
   9 });
```

Gambar 423 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 :

```
debug> repl
Press Ctrl + C to leave debug repl
> person.name
'Vestorp'
>
```

Gambar 424 After Value Changed

Lebih mudah melakukan *debugging* menggunakan *visual studio code* yah?

Subchapter 5 – Asynchronous

*The most beautiful code,
the most beautiful functions,
and the most beautiful programs
are sometimes not there at all.*

— Jon Bentley

Subchapter 5 – Objectives

- Memahami konsep **Callback**
 - Memahami konsep **Promise**
 - Memahami konsep **Fetch**
 - Memahami konsep **Async-await**
-

Pada **subchapter** 2 anda telah mempelajari konsep **Synchronous** & **Asynchronous** dalam **The Call Stack** sehingga anda memiliki gambaran dasar. Sekarang kita akan mengeksplorasi lebih dalam lagi secara teknis secara sistematis.



Gambar 425 Asynchronous Programming

1. Callback

Callback adalah suatu **function** yang akan dieksekusi setelah fungsi tertentu selesai dieksekusi. Itulah alasan kenapa kita memberi nama **call back**.

Dalam **javascript**, kita dapat membangun **higher-order function** yaitu sebuah **function** dapat menggunakan **function** lainnya sebagai **argument** dan memberikan **return function** lagi.

```
const getUser = (id: any, callback: any) => {
```

```

const user = {
  id: id,
  name: "Gun Gun Febrianza",
};

console.log("First Function Done!");
setTimeout(() => {
  callback(user);
}, 3000);

getUser(31, (userObject: any) => {
  console.log("Then Executen Call back");
  console.log(userObject);
});

```

Jika kode di atas dieksekusi maka akan memproduksi **ouput** :

```

/*
First Function Done!
Then Executen Call back
{ id: 31, name: "Gun Gun Febrianza" }
*/

```

Pada kode di atas kita dapat memahami bahwa **callback** akan dieksekusi setelah suatu fungsi selesai beroperasi.

Problem dari penggunaan **callback** adalah fenomena **callback hell**.

2. Promise

Promise sebelumnya adalah sebuah **third-party library** diluar **core javascript**, namun pada akhirnya diterapkan menjadi salah satu **core** penting dalam bahasa pemrograman **javascript**.

Pada akhirnya **promise** diperkenalkan dalam **EcmaScript** 6 sebagai **native object**. **Promise** di desain untuk memecahkan permasalahan terkait **asynchronous programming** dalam aplikasi yang akan kita buat.

Jika dibandingkan dengan **callback**, **promise** mempermudah kita untuk melakukan **asynchronous computation**.

Promise Constructor

Di bawah ini adalah sebuah contoh kode **promise constructor** di dalam **javascript** :

```
const promise = new Promise(function(resolve, reject) {  
  if (condition) {  
    resolve(value); //successfully resolve the Promise  
  } else {  
    reject(reason); //reject the Promise and specify the reason  
  }  
})
```

Parameter dari **object promise** adalah sebuah **function** yang disebut dengan **executor**, fungsi **resolve** dieksekusi jika **promise** berhasil dan fungsi **reject** dieksekusi jika **promise** gagal.

Di dalam sebuah **object promise** terdapat dua **properties** yaitu :

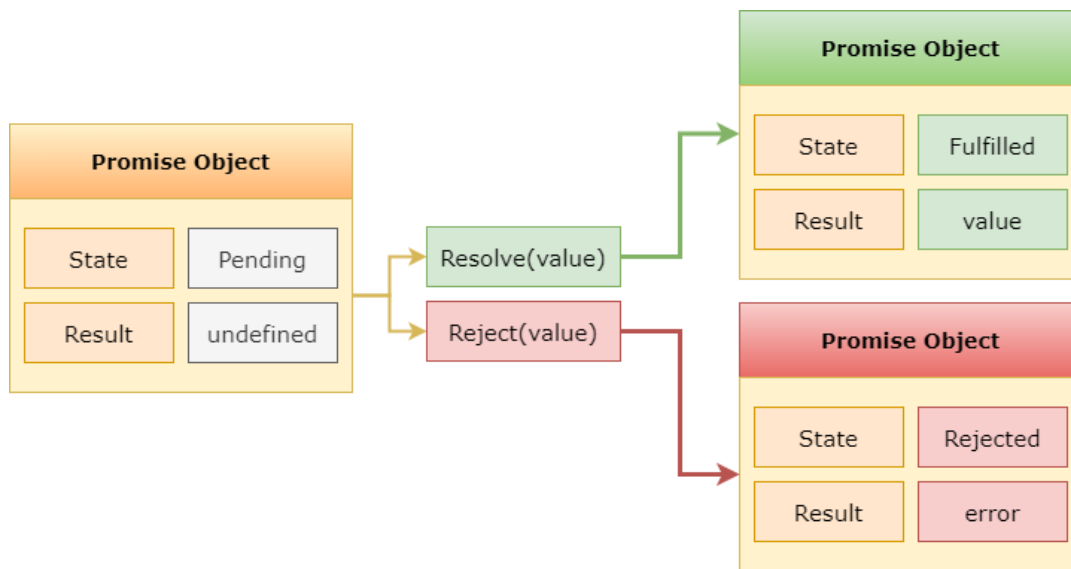
1. State

Saat pertama kali kita membuat sebuah **object promise** nilai **state** adalah **pending**, jika **promise** mulai digunakan **state** dapat berubah menjadi **fulfilled** atau **rejected**.

2. Result

Saat pertama kali kita membuat **object promise** nilainya adalah **undefined**.

Saat kita membuat sebuah **object promise** secara **internal** terdapat direpresentasikan sebagai berikut :

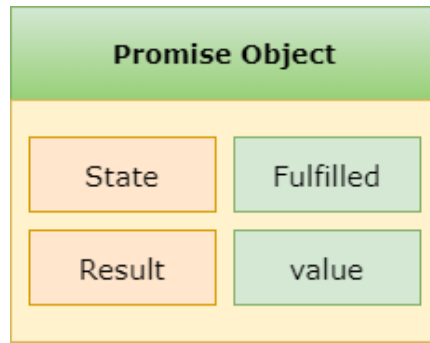


Gambar 426 Promise Object

Saat **executor** berhasil menyelesaikan pekerjaannya terdapat satu respon yang akan dieksekusi yaitu :

Resolve

Resolve adalah **pre-defined function** yang telah disediakan oleh **javascript engine** untuk kita gunakan. Jika respon adalah eksekusi **function resolve** maka **state** akan berubah menjadi **fulfilled** dan **result** adalah hasil dari komputasi.



Gambar 427 Resolve Result

Di bawah ini adalah contoh kode **Promise Resolve** :

```
const promise = new Promise(function (resolve, reject) {
  setTimeout(() => {
    resolve("Success!");
  }, 1000);
});

promise.then(
  (result) => console.log(result),
  (error) => console.log(error)
);
```

Jika kode di atas dieksekusi maka **output** yang diproduksi adalah :

Jika kode di atas dieksekusi maka **output** yang diproduksi adalah :

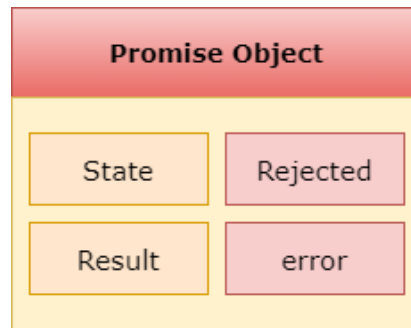
```
// Success!
```

Menggunakan **function** `setTimeout` kita mencoba merepresentasikan **asynchronous processing** sederhana, jika **function** `setTimeout` berhasil dieksekusi setelah satu detik **function** `resolve` harus dieksekusi.

Function Resolve akan mengubah **properties state** dan **result** dari **object promise**.

Reject

Reject juga adalah **pre-defined function** yang telah disediakan oleh **javascript engine** untuk kita gunakan. Jika respon adalah eksekusi **function reject** maka **state** akan berubah menjadi **rejected** dan **result** adalah hasil dari komputasi yang berakhir **error**.



Gambar 428 Rejected Result

Function Resolve & Reject akan mengubah **properties state** dan **result** dari **object promise**. Di bawah ini adalah contoh kode **Promise Reject** :

```
const promise = new Promise(function (resolve, reject) {
  setTimeout(() => {
    reject(new Error("Error!"));
  }, 1000);
});

promise.then(
  (result) => console.log(result),
  (error) => console.log(error)
);
```

Jika kode di atas dieksekusi maka **output** yang diproduksi adalah :

```
// Error: Error!
```


Example Promise

Kita akan membuat sebuah **study** kasus operasi penjumlahan antar **integer**, jika **operand** adalah sebuah **integer** maka **promise** harus mengeksekusi **function resolve** jika **operand** adalah sebuah **decimal** maka **promise** harus mengeksekusi **function reject** :

```
const integerAddition = (a: number, b: number) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Number.isInteger(a) && Number.isInteger(b)) {
        resolve(a + b);
      } else {
        reject("Arguments must be numbers");
      }
    }, 1500);
  });
};
```

Untuk menggunakan **object promise** di atas tulis kode di bawah ini :

```
integerAddition(5, 7).then(
  (res) => {
    console.log("Result:", res);
  },
  (errorMessage) => {
    console.log(errorMessage);
  }
);
```

Pada kode di atas kita melakukan operasi antar **integer**, ada dua **operand** yaitu **5** dan **7** jika kode di eksekusi maka akan memproduksi **output** :

```
// Result: 12
```

Jika kita mengubah salah satu **operand** atau keduanya menjadi **decimal** misal **5.5** dan **7.1** jika kode di eksekusi maka akan memproduksi **output** :

```
// Arguments must be integer
```

3. Fetch

Untuk pada **front-end developer** yang biasa mengembangkan **web application** pasti akan menggunakan **promise** untuk melakukan **network requests**. Kita tahu bahwa **deno** memiliki kelebihan dengan menyediakan **object** dalam **browser** untuk bisa digunakan **stand-alone** diluar **browser**, salah satunya adalah **object fetch**.

Return yang dihasilkan dari **fetch** adalah **promise**.

Fetch Kawal Korona

Pada kode di bawah penulis membuat **study** kasus untuk menggunakan **object fetch** :

```
fetch("https://api.kawalcorona.com/indonesia/")
  .then((res) => res.json())
  .then((data) => {
    data.map((obj: Object) => {
      console.log(obj);
    });
  })
  .catch((err) => console.log(err));
```

Jika kode di atas dieksekusi maka akan memproduksi :

```
/*
{
  name: "Indonesia",
  positif: "23,851",
  sembuh: "6,057",
  meninggal: "1,473",
  dirawat: "16,321"
}
*/
```

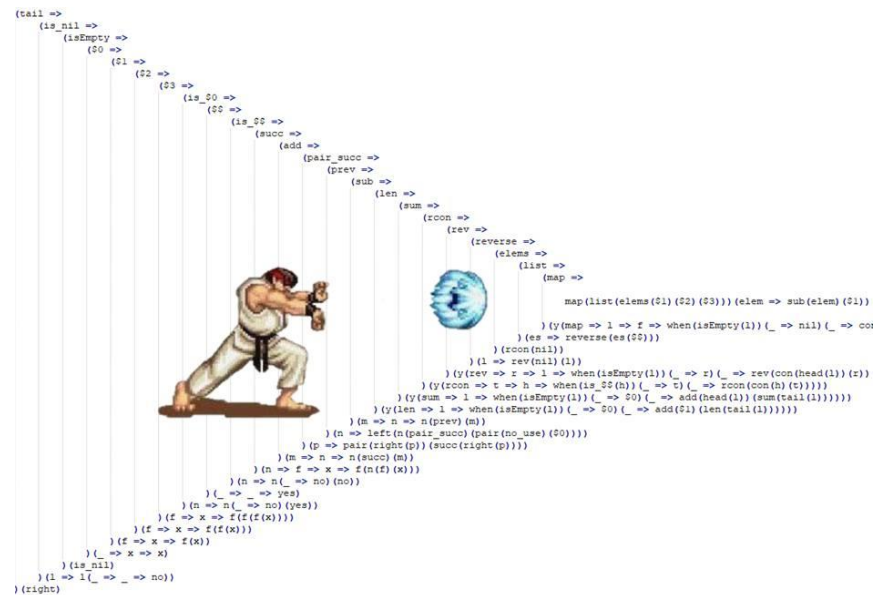
Promise Chaining

Jika kita perhatikan pada kode sebelumnya terdapat **method then** yang dipanggil dua kali, inilah yang disebut dengan **promise chaining**. Kita dapat memproses hasil dari **promise** sebelumnya untuk diproses :

```
.then((res) => res.json())
.then((data) => {
  data.map((obj: Object) => {
    console.log(obj);
  });
})
```

4. Async Await

Ada **syntax** yang lebih menarik untuk digunakan jika ingin menggunakan **promise** yaitu menggunakan konsep **async/await**. Penggunaan **callback** yang berlebihan menimbulkan fenomena **callback hell**, kode menjadi berantakan :



Gambar 429 Callback Hell

Untuk mengatasi permasalahan ini diperkenalkanlah konsep **promise** dan **function chaining** pengembangan terus berlanjut hingga akhirnya diperkenalkan konsep **async/await** dalam **V8 Engine** untuk membenahi **promise** dan **function chaining**.

Kita dapat menggunakan **await function** yang dapat memberikan **return** berupa **promise** dengan syarat harus dibungkus dengan **async function** terlebih dahulu. Seperti pada kode di bawah ini :

```
async function get(req,res) {
  const response = await request.get('https://zzz:3000')
  if (response.err) { console.log('Error!');}
  else { console.log('fetched response');
```

```
}  
}
```

Pada kode di atas kita memberikan instruksi pada **javascript engine** untuk menunggu fungsi **request.get()** selesai dieksekusi sebelum mengeksekusi baris kode berikutnya. Nilai **return** dari **request.get()** adalah **promise**.

Dengan **async/await** kita dapat mengeksekusi kode **asynchronous** secara berurutan

Async Function

Untuk membuat **async function** kita perlu menempatkan **async keyword**, di bawah ini adalah contoh kode **async function** :

```
async function f() {  
    return "Hello Maudy";  
}  
  
console.log(f());
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
//Promise { 'Hello Maudy' }
```

Jika kita menggunakan **async function** maka **return** yang diberikan akan menjadi **promise**.

Pada kode di atas **return** adalah sebuah **string** namun akan otomatis dikonversi kedalam **promise** secara **internal** oleh **javascript engine**, dikarenakan kita menggunakan **async function**.

Untuk mendapatkan **return string** dari **async function**, kita perlu menggunakan **method then** untuk mendapatkan **resolve** :

```
f().then((result) => console.log(result));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// Hello Maudy
```

Kode sebelumnya sama dengan kode di bawah ini :

```
async function f() {  
  //return "Hello Maudy";  
  return Promise.resolve("Hello Maudy");  
}  
  
f().then((result) => console.log(result));
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// Hello Maudy
```

Await

Penggunaan **keyword await** memberikan instruksi pada **javascript engine** untuk menunggu sampai **promise** selesai beroperasi memberikan **return**, sebelum mengeksekusi baris kode selanjutnya :

```
async function f() {
```

```
const objpromise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done"), 1000);
});
let result = await objpromise;
console.log(result);
}

f();
```

Jika kode di atas dieksekusi maka akan memproduksi **output** :

```
// Done
```

Error Handling

Jika terjadi **error** pada **promise** dikarenakan mengalami **rejection**, maka kita dapat menggunakan dua cara seperti pada kode di bawah ini :

```
async function f() {
  await Promise.reject(new Error("error!"))
}

async function f() {
  throw new Error("error!")
}
```

Selain menggunakan cara di atas kita juga dapat memanfaatkan **try...catch** seperti pada kode di bawah ini :

```
async function f() {
  try {
```



```
    let response = await fetch("http://wrong-url");  
  } catch (error) {  
    console.log(error);  
  }  
}  
  
f();
```

5. Top Level Await

Konsep dari **top level await** adalah usulan untuk dapat mengeksekusi **await statement** diluar **async function**. Penggunaan **top level await** agar penggunaan **await statement** diperlakukan seperti **async function**.

Sebelumnya jika kita menggunakan **await statement** di luar **async function** maka **syntax error** akan terjadi, seperti pada kode di bawah ini :

```
await Promise.resolve(console.log('Hi Maudy'));
```

Sehingga kebanyakan **developer** menggunakan **immediately-invoked async function expressions** untuk dapat menggunakan **await statement**. Seperti pada kode di bawah ini :

```
(async function() {  
  await Promise.resolve(console.log('Hi Maudy'));  
  // → Hi Maudy  
})();
```

Sekarang dalam **V8 Engine** yang digunakan di dalam **node.js** dan **deno** kita dapat menggunakan **top level await**. Sebagai catatan, penggunaan hanya dapat dilakukan pada **top level module**.

Proposal pengajuan implementasi pada **top level await** dapat di baca disini :

<https://github.com/tc39/proposal-top-level-await>

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-opec/> [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] Mueller, 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]

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.*

My Books

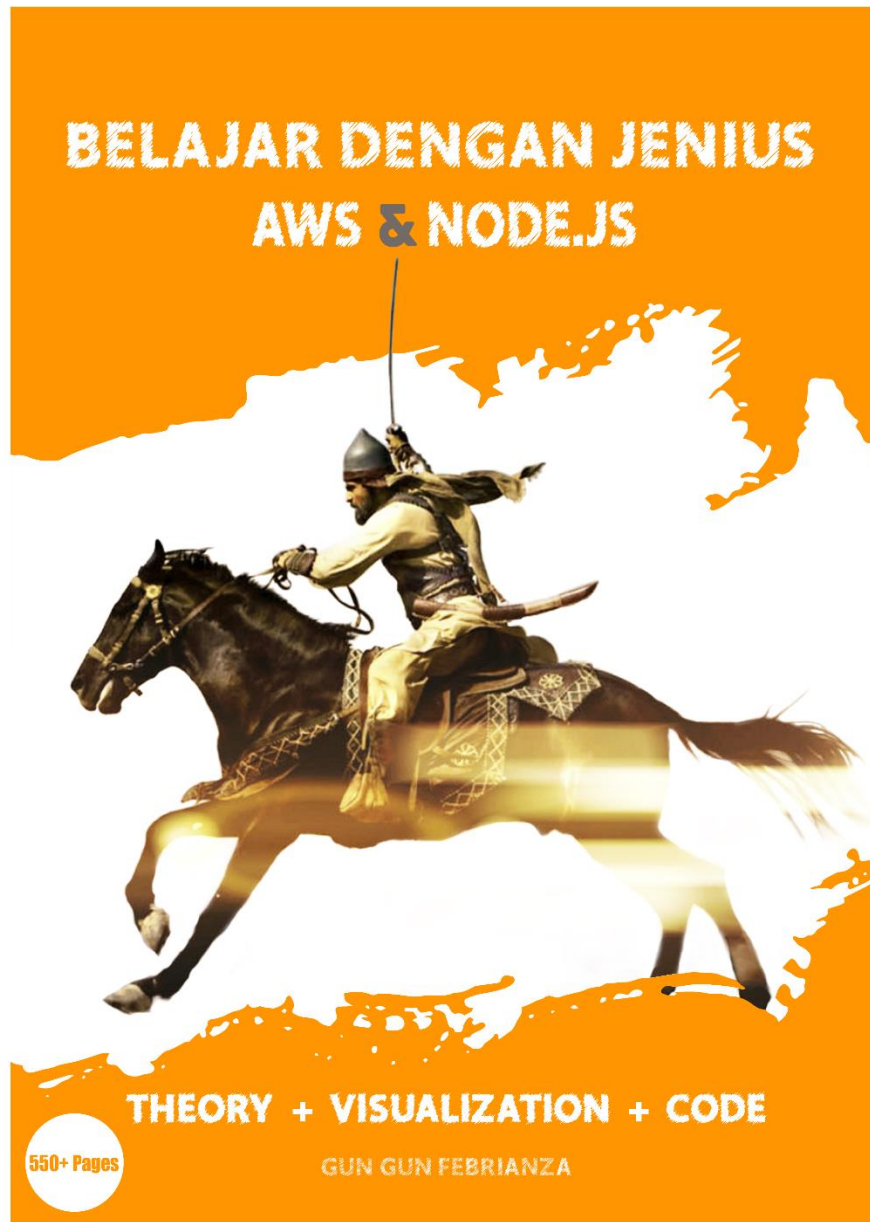
Belajar dengan Jenius Golang

<https://github.com/gungunfebrianza/Belajar-Dengan-Jenius-Golang>



Belajar Dengan Jenius AWS & Node.js

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



Belajar Dengan Jenius Amazon IAM

<https://github.com/gungunfebrianza/Belajar-Dengan-Jenius-AWS-IAM>



Develop Security Software With C#

<https://github.com/gungunfebrianza/Develop-Security-Software-With-CSharp>

