Modul
01

MSIM4401
Edisi 1

Pengenalan Lingkungan Pengembangan Aplikasi Berbasis Perangkat Bergerak

Dr. Bambang Purnomosidi D.P

Daftar Isi Modul

Pendahuluan

Aplikasi perangkat bergerak merupakan aplikasi yang dimaksudkan untuk di-install dan dijalankan pada perangkat bergerak (mobile phone/gadget). Saat ini, pasar mobile phone terbagi menjadi dua kelompok besar yakni Android dan iOS (iPhone). Pangsa pasar untuk Android saat ini menempati bagian yang lebih besar daripada iOS. Statistik pada https://gs.statcounter.com/os-market-share/mobile/worldwide menunjukkan bahwa pangsa pasar pada bulan November 2020 adalah 71,18% untuk Android, 28,19% untuk iOS, dan sisanya untuk perangkat bergerak lainnya.

Sebagai suatu perangkat bergerak dengan pangsa pasar terbesar, Android mempunyai berbagai perangkat pengembangan yang dapat digunakan. Dari berbagai perangkat pengembangan tersebut, Google sebagai perusahaan yang memproduksi Android OS mendukung dua bahasa pemrograman resmi yang, yakni **Java** dan **Kotlin**. Pengembangan menggunakan Java dan Kotlin dilakukan dengan menggunakan *software* Android Studio yang dapat diperoleh secara bebas. Meskipun demikian, banyak komunitas maupun perusahaan-perusahaan lain yang membuat perangkat pengembangan mereka sendiri. Dari berbagai piranti pengembangan tersebut, kita dapat membaginya ke dalam 2 kategori besar yaitu ***native*** dan ***hybrid***.

Perangkat pengembangan *native* langsung dapat dijalankan oleh *platform* Android - artinya dikembangkan dan dikompilasi ke *bytecode* JVM (*Java Virtual Machine*) karena Android *runtime* dikembangkan menggunakan Java. Java dan Kotlin merupakan bagian dari peranti pengembangan *native* karena hasil yang diperoleh merupakan *bytecode* JVM.

Perangkat pengembangan *hybrid* dijalankan dengan menggunakan fitur ***WebView*** dan dengan pustaka (*library code*) tertentu dibuat hingga dapat mengakses sisi *native* dari sistem Android. Perangkat pengembangan ini menggunakan berbagai sarana spesifikasi *Web* (terutama HTML dan CSS) untuk membentuk antarmuka (*interface*), dan memerlukan pustaka (*library*) lain jika ingin mengakses sisi *native* (seperti mengaktifkan dan menggunakan kamera, dan sejenisnya).

Untuk dapat/mampu membangun aplikasi dengan menggunakan pendekatan *hybrid* ini, seorang pemrogram harus memahami pembuatan aplikasi di sisi *Web* terutama menggunakan HTML, CSS, serta *JavaScript* dan *framework* antarmuka (*interface*) *Web* tertentu. Salah satu *framework* yang dapat digunakan untuk keperluan tersebut adalah *Vue*. *Vue* dikembangkan dengan menggunakan *JavaScript* tetapi akhir-akhir ini lebih dikembangkan dengan menggunakan *TypeScript*. Dengan demikian, pemahaman terhadap *TypeScript* diperlukan untuk dapat membangun antarmuka serta proses bisnis dan akses ke *server remote*. *Framework Ionic* yang akan digunakan pada modul ini merupakan *framework* yang menggunakan 3 *framework* lainnya, yaitu: *React, Angular*, serta *Vue*.

Pembahasan tentang *Ionic* dengan menggunakan *Vue* untuk mengembangkan aplikasi berbasis perangkat bergerak akan menjadi topik dari modul ini.

Setelah mempelajari modul ini, diharapkan Anda mampu:

1. menguraikan gambaran umum dua *platform* pengembangan aplikasi perangkat bergerak: Android dan iOS;
2. menjelaskan perbedaan aplikasi *native* dan *hybrid*;
3. menyebutkan perangkat pengembangan dan *framework* untuk aplikasi perangkat bergerak;
4. menguraikan fungsi seta peran dari perangkat pengembangan dan *framework* untuk aplikasi perangkat bergerak;
5. menjelaskan komponen-komponen aplikasi *hybrid*;
6. menguraikan gambaran umum dari Ionic *Framework*;
7. menguraikan berbagai komponen dari Ionic *Framework*;
8. menjelaskan ekosistem dari Ionic *Framework*;
9. melakukan instalasi lingkungan pengembangan aplikasi perangkat bergerak;
10. melakukan konfigurasi lingkungan pengembangan aplikasi perangkat bergerak;
11. menjelaskan sejarah dan latar belakang pembuatan *TypeScript*;
12. menguraikan perbedaan antara *static typing* dengan *dynamic typing*;
13. menjelaskan keterkaitan antara *TypeScript* dengan *JavaScript/EcmaScript*;
14. melakukan instalasi *TypeScript* dan berbagai perangkat (*tools)* yang diperlukan;
15. menggunakan konstruksi dasar *TypeScript* dalam pembuatan *software*;
16. menggunakan fungsi di *TypeScript* untuk pemrograman modular;
17. menggunakan paradigma *OOP* di *TypeScript*;
18. melakukan pemrograman *asynchronous* pada *TypeScript*;
19. membuat *Web API - RESTful API* dengan menggunakan *TypeScript*.

Lingkungan Pengembangan Aplikasi Perangkat Bergerak, Hybrid App dan Ionic Framework

Kegiatan Belajar 1

Pada umumnya, aplikasi yang berjalan pada perangkat bergerak Android dapat dikembangkan secara *native* maupun *hybrid*. Aplikasi yang dikembangkan secara *native* memungkinkan dapat langsung dijalankan oleh *runtime* dari Android; sedangkan aplikasi *hybrid* merupakan aplikasi yang harus dijalankan di atas *platform Web* – pada sistem operasi Android, aplikasi tersebut dijalankan dengan menggunakan fitur *WebView*.

Salah satu *framework* yang dapat digunakan untuk mengembangkan aplikasi *hybrid* ini adalah **Ionic**. Berbagai peranti pengembangan sekarang sudah memungkinkan untuk membangun aplikasi *cross mobile devices platform*, namun demikian dalam buku materi pokok ini hanya akan dibahas aplikasi berbasis sistem operasi Android.

A. *PLATFORM* PENGEMBANGAN APLIKASI PERANGKAT BERGERAK

Aplikasi *mobile* dikembangkan dengan *platform mobile* tertentu. Saat ini terdapat 2 *platform* utama untuk pengembangan aplikasi *mobile* yaitu iOS (untuk perangkat iPhone) dan Android. Pada sistem Android, sistem operasi yang digunakan di perangkat *mobile* tersebut adalah sistem operasi Linux yang sudah dimodifikasi agar dapat digunakan dengan perangkat layar sentuh *mobile*. Sistem operasi untuk Android tersebut merupakan sistem operasi bebas (*free*) dengan kode terbuka (*open source*), itulah sebabnya kemudian banyak muncul berbagai varian dari Android yang dibuat oleh *vendor mobile phone*, misalnya ColorOS - untuk *mobile phone* Oppo.

1. Gambaran Umum

Android OS dikembangkan berdasar pada kernel Linux yang telah dimodifikasi. Kode sumber untuk Android OS ini dapat diperoleh di https://source.android.com. Android OS merupakan sistem operasi yang tersedia bebas. Secara umum, komponen dari Android OS ini dapat dilihat pada Gambar 1.1.

Gambar 1.1
Arsitektur Android OS

Agar dapat memahami cara membangun suatu aplikasi Android, tidak terlalu diperlukan pengetahuan mendalam sampai di level paling bawah (kernel), melainkan cukup dengan memahami arsitektur Android OS secara umum.

Aplikasi Android berada di level paling atas dan akan dijalankan oleh suatu *runtime*. Pada Android di bawah versi 4.4 (**KitKat**) *runtime* yang digunakan adalah **Dalvik**. Pada versi 4.4 sudah menggunakan versi ART (*Android Run Time*) sebagai pengganti dari Dalvik, namun masih menyertakan Dalvik. Mulai versi 5.0 (**Lolipop**) hingga versi pada saat ini (Android v.11/**Red Velvet Cake**), Android sudah menggunakan ART secara penuh dan menghilangkan Dalvik. *Runtime* tersebut akan menjadi jembatan ke akses rendah/mesin melalui HAL (*Hadware Abstraction Layer*) yang didefinisikan menggunakan HIDL (*HAL Interface Definition Language*). *Kernel* merupakan *software* yang mengelola mesin secara langsung dan menyediakan antarmuka (*interface*) bagi ART.

2. Perbedaan Aplikasi *Native* dan *Hybrid*

Aplikasi *native* merupakan aplikasi yang dijalankan langsung oleh ART (pada Dalvik di versi 4.4/KitKat ke bawah). Aplikasi *native* dibuat dengan menggunakan bahasa pemrograman Java/Kotlin atau semua bahasa yang menghasilkan *bytecode* atau C++ dengan akses langsung ke fungsi *native* dan kemudian menggunakan JNI (*Java Native Interface*) yang dipanggil oleh Java/Kotlin. Beberapa aplikasi dibangun dengan SDK (*Software Development Kit*) lain seperti Flutter yang menggunakan C/C++ di NDK (*Native Development Kit*) untuk membangun *engine*.

Pada teknologi lainnya, suatu aplikasi *hybrid* memerlukan suatu *software* lain yang akan dimasukkan ke dalam suatu aplikasi yaitu teknologi *browser*. *Browser* akan dimasukkan sebagai *engine* dari suatu aplikasi dan memungkinkan aplikasi yang

dibangun dapat menggunakan standar *Web* (HTML, CSS, JavaScript) sebagai sarana antarmukanya. Secara *default*, seperti halnya aplikasi *Web* pada perangkat bukan *mobile*; suatu aplikasi *Web* tidak memungkinkan/mengizinkan aplikasi untuk dapat mengakses fasilitas level rendah/mesin secara langsung. Untuk keperluan tersebut, digunakan perangkat *software* lain yang berfungsi untuk menyediakan jembatan bagi aplikasi *Web* dengan akses level rendah.

Aplikasi *native* mempunyai kemampuan yang baik untuk mengakses semua fasilitas standar dari *mobile phone*. Sementara itu, untuk *hybrid* diperlukan jembatan dan *wrapper* terhadap kemampuan akses level rendah. Hal ini menyebabkan kode sumber (*source code*) untuk aplikasi *native* relatif dapat langsung mengakses level rendah tanpa bantuan dari berbagai perangkat pihak ketiga.

Dari sisi pengembangan, aplikasi *native* bersifat relatif lebih khusus/ spesifik karena kode sumber dibuat sesuai dengan perangkat yang dituju/digunakan. Hal ini berbeda dengan *hybrid* yang menggunakan spesifikasi terbuka – *Web*; maka sekali dibangun untuk satu *platform*, biasanya dapat digunakan/dijalankan pada *platform* yang lain (Android, iOS, maupun *Web desktop*/Internet). Tugas terbesar pada pengembangan versi *hybrid* dilakukan oleh pihak ketiga yang menyediakan akses ke level rendah/mesin/ *native*. Pada pengembangan aplikasi berbasis Ionic, tugas ini dikerjakan oleh *Capacitor* dan/atau *Apache Cordova.* Subyek/topik ini akan disajikan pada saat membahas *API Plugins*.

Dari sisi ukuran dan kecepatan aplikasi, aplikasi *native* mempunyai ukuran yang lebih kecil serta kecepatan yang lebih baik dibandingkan aplikasi *hybrid*. Hal ini disebabkan karena aplikasi *hybrid* menyertakan fasilitas *browser engine* sebagai perangkat untuk menampilkan antarmuka serta menyediakan kemungkinan akses ke level rendah melalui jembatan penghubung; Sedangkan pada aplikasi *native* akan disertakan hasil kompilasi berupa *bytecode* yang siap untuk langsung dijalankan oleh ART.

3. Perangkat Pengembangan dan *Framework* untuk Aplikasi *Mobile*

Pada dasarnya, tersedia banyak perangkat lunak yang dapat digunakan untuk membangun suatu aplikasi *mobile*. Perangkat lunak tersebut terbagi dalam kategori *native* serta *hybrid*. Beberapa perangkat lunak tersebut dapat dilihat pada Tabel 1.1.

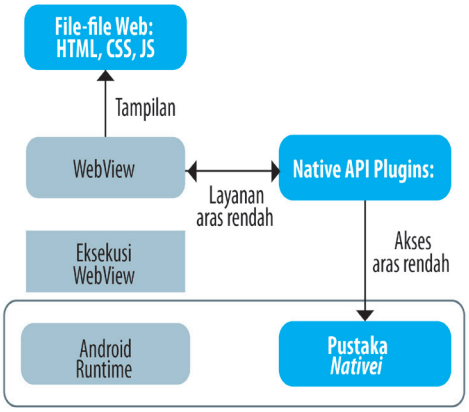
Tabel 1.1
Daftar Framework untuk Mengembangkan Aplikasi Android

Nama	Bahasa Pemrograman	Teknologi	Cross Platform Android - iOS
Java	Java	Native	Tidak
Kotlin	Kotlin	Native	Tidak
KMM (Kotlin Multiplatform Mobile)	Kotlin	Native	Ya
Flutter	Dart	Native	Ya
NativeScript	JavaScript /TypeScript	Native	Ya

Nama	Bahasa Pemrograman	Teknologi	Cross Platform Android - iOS
Quasar Framework	JavaScript /TypeScript	Hybrid	Ya
Framework7	JavaScript /TypeScript	Hybrid	Ya
Xamarin	C#	Native	Ya
ReactNative	JavaScript	Native	Ya
Ionic Framework	JavaScript /TypeScript	Hybrid	Ya

B. APLIKASI *HYBRID*

Aplikasi *hybrid* menyertakan *browser engine* sebagai komponen untuk membangun antarmuka serta akses ke level rendah/mesin. Untuk memahami lebih lanjut tentang Ionic, diperlukan pemahaman tentang arsitektur aplikasi *hybrid* seperti pada Gambar 1.2.



Gambar 1.2
Arsitektur Aplikasi *Hybrid*

Terdapat 2 komponen utama dari suatu aplikasi *hybrid*, yaitu komponen untuk tampilan dan komponen untuk akses pustaka *native*. Jika aplikasi tidak menggunakan fasilitas *native*, maka tidak diperlukan *native API plugins*. Namun apabila aplikasi memerlukan akses ke level rendah - misalnya untuk mengaktifkan kamera, *geolocation*, dan lain-lain, maka diperlukan *native API plugins* untuk fungsi tersebut. *WebView* dijalankan oleh **ART** dan kemudian menghasilkan suatu tampilan. *WebView* juga menyediakan *engine* JavaScript yang memungkinkan penggunaan JavaScript sebagai bahasa pemrograman untuk mengimplementasikan *business logic* dari aplikasi tersebut.

C. IONIC FRAMEWORK

1. Gambaran Umum

Ionic merupakan salah satu *framework* yang dapat digunakan untuk membangun aplikasi *mobile* dengan menggunakan teknologi *hybrid*. Ionic dibuat oleh perusahaan yang bernama **Drifty Co.**, pertama kali dirilis pada tahun 2013. Pada saat ini, Ionic sudah mencapai versi 5. Ionic memungkinkan pembuatan aplikasi yang bersifat *cross platform* dan dapat dijalankan pada berbagai perangkat *mobile* (Android, iOS), aplikasi *desktop*, serta aplikasi PWA (*Progressive Web Apps*).

2. Komponen Ionic *Framework*

Ionic terdiri atas 2 (dua) komponen, yakni (a) komponen untuk antarmuka dan (b) komponen untuk akses ke pustaka *native* guna memenuhi kebutuhan akses level rendah. Ionic menggunakan Apache Cordova serta Capacitor untuk memfasilitasi akses ke level rendah/mesin - khususnya untuk memberi kemampuan menggunakan sarana yang tersedia pada *mobile phone* seperti kamera, lampu (*flashlight)*, *geolocation*, dan lain sebagainya.

Pada sisi pembuatan antarmuka, Ionic memungkinkan penggunaan berbagai *framework* yang banyak digunakan di industri, yaitu Angular, React, atau Vue.

Bahasa pemrograman yang digunakan secara *default* adalah JavaScript. Meskipun demikian, TypeScript merupakan bahasa pemrograman yang makin banyak digunakan dan dapat digunakan untuk ketiga *framework* antarmuka tersebut. Vue pada versi terakhir (versi 3) dikembangkan dengan menggunakan TypeScript.

3. Ekosistem Ionic *Framework*

Beberapa *software* atau komponen *software* mempunyai keterkaitan dengan Ionic.

1. **Node.js**: JavaScript *runtime* yang dibuat dari *V8* (*JavaScript engine* dari Google Chrome).
2. **TypeScript**: bahasa pemrograman bertipe *static typing* yang merupakan *superset* dari JavaScript. TypeScript ini banyak digunakan di berbagai *framework* untuk tampilan antarmuka. TypeScript dikompilasi menjadi JavaScript dan dijalankan dengan menggunakan Node.js.
3. ***Framework* untuk antarmuka**: Ionic mendukung setidaknya 3 *framework*, yaitu Angular, React, dan Vue.
4. Pustaka / paket JavaScript dan TypeScript di **npm (**suatu *software* yang digunakan untuk mengelola paket serta proyek Node.js)
5. ***Web Components***: komponen siap pakai untuk tampilan antarmuka dengan menggunakan Ionic.
6. **Stencil**: kompilator untuk membuat *web components*.
7. **Apache Cordova**: perangkat penyedia *native API plugins* untuk memungkinkan aplikasi yang dibangun; dengan menggunakan sarana ini, Ionic dapat mengakses perangkat sistem di level rendah.

8. **Capacitor**: mempunyai fungsi yang sama dengan Apache Cordova, tetapi dibuat oleh Ionic.
9. **Android SDK** (*Software Development Kit*): perangkat pengembangan yang digunakan untuk membangun aplikasi Android.
10. **IDE** (*Integrated Development Environment*): perangkat *software* yang menyediakan lingkungan pengembangan terpadu untuk membangun aplikasi. Bagi Ionic, meskipun dapat menggunakan IDE/editor teks apa saja, tetapi disarankan menggunakan Visual Studio Code. Visual Studio Code menyediakan dukungan yang komprehensif untuk pengembangan berbasis JavaScript / TypeScript.
11. Informasi tentang integrasi Ionic dengan berbagai komponen lainnya, dapat diakses di https://ionicframework.com/integrations.

4. Instalasi dan Konfigurasi Lingkungan Pengembangan

Pembahasan pada modul ini akan lebih ditekankan pada penggunaan TypeScript. Dengan demikian, lingkungan pengembangan yang diperlukan di modul ini adalah yang terkait dengan pengembangan aplikasi dengan menggunakan TypeScript saja. Pada modul-modul selanjutnya akan dibahas tentang cara instalasi secara lebih lanjut.

Pada kegiatan belajar ini, instalasi hanya mencakup instalasi untuk:

a. Node.js.

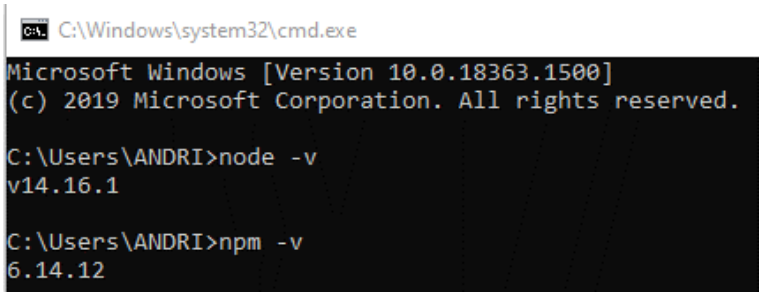
Software Node.js dapat diunduh di https://nodejs.org/en/download/. Pilih menu unduhan sesuai dengan sistem operasi yang digunakan untuk pengembangannya. Apabila menggunakan sistem operasi Windows, maka jalankan langsung file *installer* hasil unduhan (berekstensi **msi**). Untuk sistem operasi lain, gunakan *package manager* (homebrew untuk Mac, apt, rpm, pacman, dan lain-lain untuk Linux).

Node.js mempunyai dua versi yakni LTS dan *current*. LTS relatif lebih stabil, sedangkan *current* digunakan untuk aplikasi yang memerlukan fitur-fitur terkini. Untuk keperluan pembuatan aplikasi yang mengutamakan stabilitas, digunakan versi LTS. Setelah dilakukan proses instalasi, lakukan pemeriksaan dengan menggunakan perintah berikut pada *command prompt* atau *shell prompt*.

```
$ node -v
v14.15.4
$ npm -v
6.14.10
$
```

Langkah pada Windows:

1. Buka cmd : `win+R` → `ketik "cmd"` → `enter`
2. Ketik `node -v` dan `npm -v`
3. Hasilnya seperti berikut:



note : perbedaan hasil v14.16.1 dan 6.14.10 dikarenakan versi node.js yang berbeda

Catatan: Tanda $ merupakan tanda prompt/shell, merupakan pertanda bahwa perintah dijalankan pada shell (jika menggunakan Linux/Mac) atau command prompt/powershell (jika menggunakan Windows).

5. Visual Studio Code

Software Visual Studio Code, dapat diperoleh pada situs https://code.visualstudio.com/Download. Unduh sesuai dengan sistem operasi yang akan digunakan, kemudian eksekusi *installer* tersebut. Visual Studio Code mempunyai siklus rilis yang relatif cepat, oleh karena itu ada baiknya jika selalu melakukan pemeriksaan apakah Visual Studio Code yang kita gunakan merupakan software yang paling *up-to-date*.

Latihan

Untuk memperdalam pemahaman Anda mengenai materi di atas, kerjakanlah latihan berikut!

1) Terdapat beberapa *framework* yang bersifat *hybrid* selain Ionic Framework, yaitu Quasar dan Framework7. Buatlah tabel yang menunjukkan perbedaan maupun persamaan antara ketiga *framework* tersebut!

Petunjuk Jawaban Latihan

Pelajari secara singkat fitur masing-masing framework pada masing-masing dokumentasi:

1. Framework7 (https://framework7.io/docs/).
2. Quasar (https://quasar.dev/start).
3. Ionic Framework (https://ionicframework.com/docs).

Setelah itu, buat tabel untuk merangkum fitur masing-masing.

Rangkuman

Saat ini tersedia 2 platform besar untuk perangkat *mobile* yaitu Android serta iOS. Android menempati pangsa pasar tertinggi di seluruh dunia. Aplikasi untuk Android dikembangkan dengan menggunakan 2 teknik, yaitu *native* dan *hybrid*.

Aplikasi *native* adalah aplikasi yang dikembangkan secara langsung untuk dijalankan secara langsung dengan menggunakan ART (*Android Runtime*).

Aplikasi *hybrid* menyertakan *browser engine* di dalam aplikasi. *Browser engine* tersebut berfungsi untuk menghasilkan tampilan berbasis spesifikasi teknologi *Web* (HTML, CSS, JavaScript). Guna mendapatkan akses aras/level rendah/sistem, aplikasi *hybrid* menggunakan perangkat pembantu yang disebut *Native API plugins*.

Salah satu *framework* yang dapat digunakan untuk mengembangkan aplikasi *hybrid* adalah Ionic. Ionic memungkinkan seseorang pengembang aplikasi untuk dapat membangun aplikasi dengan menggunakan spesifikasi teknologi *Web* serta memakai *framework* antarmuka Angular, React, atau Vue; dengan menggunakan JavaScript dan/atau TypeScript sebagai bahasa pemrograman untuk Ionic.

Tes Formatif 1

Pilihlah satu jawaban yang paling tepat!

1) Berikut ini adalah perangkat *mobile phone*, *kecuali* ….
   A. windows phone
   B. android
   C. flutter
   D. iphone

2) Komponen dari Android OS yang digunakan untuk mengelola sistem pada aras rendah adalah ….
   A. kernel linux
   B. hal
   C. *android runtime*
   D. dalvik

3) Komponen yang selalu disertakan dalam *bundle* aplikasi *hybrid* adalah ….
   A. *art*
   B. *webview*
   C. *native api plugins*
   D. *node.js*

4) Aplikasi *hybrid* dapat mengakses aras rendah / sistem dengan menggunakan ….
   A. cross platform development tools
   B. stencil
   C. *kernel engine*
   D. *native api plugins*

5) Ionic merupakan *framework* yang dibuat dengan menggunakan bahasa pemrograman ….
   A. java
   B. native c/c++
   C. typescript
   D. kotlin

6) Untuk membuat tampilan antarmuka, Ionic menggunakan berbagai *framework* berikut ini, *kecuali* ….
   A. angular
   B. svelte
   C. vue
   D. react

7) Untuk mengakses sistem aras rendah, Ionic menggunakan komponen dari ….
   A. Capacitor
   B. NDK (*Native Development Kit*)
   C. KMM
   D. JVM

8) Salah satu bagian dari Ionic adalah *Web components*. Bagian ini dikembangkan oleh Ionic dengan menggunakan ….
   A. apache cordova
   B. stencil
   C. capacitor
   D. vue

9) Bahasa pemrograman JavaScript dapat dijalankan dengan menggunakan *runtime* ….

A. kompilator typescript
B. node.js
C. kotlin
D. npm

10) Berikut ini adalah *framework* yang menyediakan teknologi pengembangan *native* untuk aplikasi Android, *kecuali* ….

A. react
B. flutter
C. kotlin
D. nativescript

Cocokkanlah jawaban Anda dengan Kunci Jawaban Tes Formatif 1 yang terdapat di bagian akhir modul ini. Hitunglah jawaban yang benar. Kemudian, gunakan rumus berikut untuk mengetahui tingkat penguasaan Anda terhadap materi Kegiatan Belajar 1.

$$\text{Tingkat Penguasaan} = \frac{\text{Jumlah Jawaban yang Benar}}{\text{Jumlah Soal}} \times 100$$

Arti tingkat penguasaan

<70%	70% - 79%	80% - 89%	90% - 100%
kurang	cukup	baik	baik sekali

Apabila mencapai tingkat penguasaan 80% atau lebih, Anda dapat meneruskan dengan Kegiatan Belajar 2. **Bagus!** Jika masih di bawah 80%, Anda harus mengulangi materi Kegiatan Belajar 1, terutama bagian yang belum dikuasai.

Kegiatan Belajar 2

Pemrograman Typescript

Kegiatan belajar ini akan membahas tentang dasar-dasar dari bahasa pemrograman TypeScript. TypeScript merupakan bahasa pemrograman yang dirancang dan dibuat oleh Microsoft dengan bantuan dari komunitas. **Anders Hejlsberg** dari Microsoft memimpin proses pembuatan rancangan spesifikasi serta implementasi dari TypeScript.

A. DASAR-DASAR TYPESCRIPT

TypeScript merupakan *superset* dari JavaScript sehingga semua sintaksis dari JavaScript juga merupakan sintaksis yang *valid* pada TypeScript. TypeScript merupakan bahasa pemrograman dengan fasilitas *static typing*, artinya menggunakan berbagai definisi tipe dalam konstruksi bahasa pemrograman (variabel, fungsi, modul). Penggunaan *static typing* ini mencegah *error* pada saat *runtime* (saat program dijalankan) karena kompilator akan memeriksa kemungkinan terjadinya kesalahan tipe data (misal pengulangan sejumlah "x" - *error* - karena harusnya pengulangan sejumlah angka).

1. Instalasi TypeScript

Implementasi dari spesifikasi TypeScript (kompilator) dibuat dengan menggunakan JavaScript. Untuk instalasi, sebenarnya kita dapat menggunakan dua perangkat lunak:

a. **Kompilator** TypeScript - paket dari **npm**. Versi ini merupakan versi yang selalu diperbaharui sesuai dengan kondisi terbaru dari TypeScript.

b. **Deno**, *runtime* untuk JavaScript serta TypeScript. Dibuat oleh Ryan Dahl (pencipta asli dari Node.js). Deno mirip dengan Node.js, hanya saja Deno sudah sekaligus menyertakan TypeScript. Versi TypeScript yang ada di Deno biasanya bukan versi terbaru. Deno dapat diperoleh di https://deno.land/.

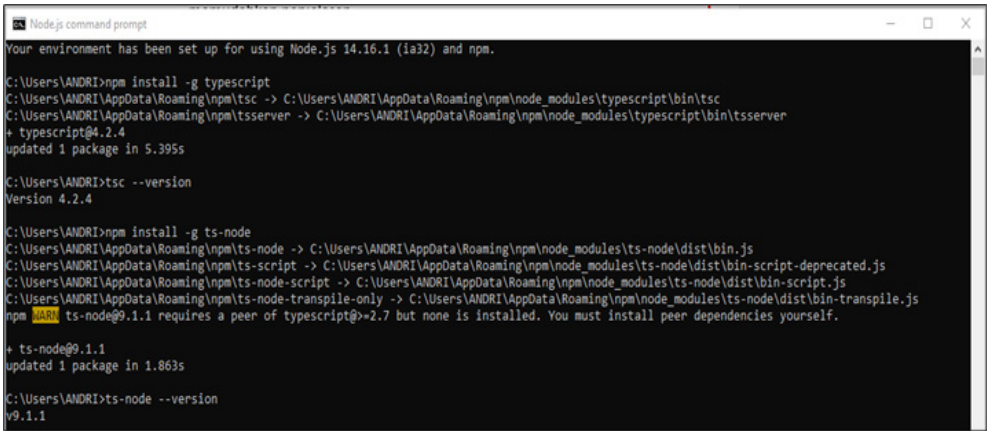
Untuk keperluan modul ini, kita menggunakan versi paket dari npm. Instalasi TypeScript dapat dilakukan dengan perintah npm dari Node.js (buka Node.js command prompt).

```
$ npm install -g typescript                    // <1>
...
... keluaran hasil npm
...
$ tsc --version                                // <2>
Version 4.1.3
$ npm install -g ts-node                       // <3>
...
... keluaran hasil npm
...
$ ts-node --version                            // <4>
v9.1.1
$
```

1) Instalasi kompilator TypeScript (*tsc*);
2) Memeriksa hasil instalasi TypeScript;
3) Instalasi ts-node untuk REPL dari TypeScript. REPL (*Read-Eval-Print-Loop*) digunakan untuk menuliskan kode-kode pendek yang akan langsung dikerjakan;
4) Memeriksa hasil instalasi ts-node.

Catatan: tanda "*// <angka>*" di atas dan seterusnya pada modul ini bukan merupakan bagian dari perintah/hasil, ditulis hanya untuk memudahkan penjelasan.

Hasil dari langkah 1 sampai dengan 4 adalah sebagai berikut:



2. Menjalankan TypeScript

TypeScript dapat dijalankan dengan menggunakan mode *kode sumber* maupun menggunakan REPL. Menggunakan REPL biasanya hanya untuk keperluan mencoba berbagai konstruksi pemrograman yang pendek atau untuk proses pembelajaran saja

```
$ ts-node
> console.log("Halo dari TypeScript")
Halo dari TypeScript
undefined
>
```

Dengan menggunakan mode *kode sumber*, maka langkah yang dilakukan adalah:

a. Pemrogram membuat program TypeScript (dengan ekstensi **.ts**);
b. Kompilasi program tersebut menggunakan kompilator TypeScript. Jika semua berjalan baik, maka hasil kompilasi berupa file JavaScript;
c. Jalankan file JavaScript tersebut dengan menggunakan Node.js.

Berikut adalah contoh dengan menggunakan Listing 1.1. Pembahasan akan diberikan di bagian-bagian berikutnya. Pada bagian ini, pelajari bagaimana cara membuat kode sumber dan kemudian menjalankan kode sumber tersebut.

```
let message: string = 'Hello, World!';      // <1>
console.log(message);                       // <2>
```

Listing 1.1
Kode sumber sederhana untuk ilustrasi menjalankan TypeScript

1) Mendefinisikan *variable message*.
2) Menampilkan tulisan sesuai isi *variabel message* yaitu 'Hello, World!'.

Urut-urutan kompilasi sampai dengan menjalankan hasil adalah sebagai berikut.

```
$ ls                        // <1>
app.ts
$ tsc app.ts                // <2>
$ ls
app.js  app.ts              // <3>
$ node app.js               // <4>
Hello, World!
$
```

1) Perintah untuk menampilkan isi direktori. Ganti dengan *command*/perintah **dir** jika menggunakan Windows.
2) Perintah untuk mengkompilasi.
3) Menghasilkan file JavaScript sesuai nama, dengan ekstensi **.js**.
4) Menjalankan hasil kompilasi.

3. Dasar-dasar Pemrograman TypeScript

a. Variabel

Variabel merupakan nama di suatu lokasi *memory* tertentu yang digunakan untuk menyimpan suatu nilai. Penyimpanan nilai dalam suatu nama ini diperlukan karena pemrograman berkaitan dengan manipulasi data/nilai. Ada beberapa cara untuk menetapkan variabel di TypeScript.

1) **var**: tidak menentukan tipe data, dapat dideklarasikan ulang kapan saja.
2) **let**: dapat menentukan atau tidak menentukan tipe data, hanya dapat dideklarasikan sekali saja, selebihnya langsung memanipulasi nilai.
3) **const**: dapat menentukan atau tidak menentukan tipe data, sekali dideklarasikan tidak dapat diubah.

Penggunaan masing-masing dapat dilihat pada Listing 1.2.

```
var a = 10;
console.log(a);
var a = 20;                          // <1>
console.log(a);
let b = 30;
console.log(b);
b = b + 5;                           // <2>
// let b = b + 10;                   // <3>
const c = “softwareku”;
console.log(c);
// c = “softwareku - versi 1.0”;     // <4>
```

Listing 1.2
Deklarasi variabel

1) Deklarasi dengan **var** dapat dideklarasikan ulang. Contohnya var a = 20 dideklarasikan ulang dari var a = 10;
2) Deklarasi dengan **let** hanya dapat diteruskan dengan memanipulasi nilai;
3) *Error* karena deklarasi ulang;
4) *Error* karena mengubah nilai.

Meskipun terdapat *error*, jika dikompilasi akan tetap menghasilkan file **.js** dan dapat dijalankan dengan menggunakan Node.js. Meskipun demikian, sebaiknya perbaiki dulu semua *error*.

Pada saat membahas tentang **Pemrograman Modular - Fungsi**, kita akan membahas sisi lain dari ketiga hal tersebut.

b. Tipe Data Dasar

Tipe data merupakan hal yang sangat penting dalam suatu struktur kode sumber. Masalah yang muncul sering kali adalah masalah yang terkait dengan tipe data. Dengan menggunakan tipe data statis (sekali ditetapkan tidak dapat diubah), maka kode sumber

akan dapat diperiksa berbagai kemungkinan terjadinya *bugs*/*error* sebelum program tersebut dijalankan. TypeScript mempunyai beberapa tipe data dasar. Tipe data dasar yang terkait dengan pembahasan lain (*class, object*) akan dibahas di bagian lain.

Beberapa tipe data dasar dari TypeScript adalah sebagai berikut:

1) **Boolean**: menyimpan nilai benar (*true*) atau salah (*false*) saja.
2) **Number**: menyimpan nilai angka (*floating point*: pecahan dan *big integer*: bilangan bulat besar). TypeScript juga mendukung *decimal*, *hexadecimal* (0x), *octal* (0o), biner (0b).
3) **String**: menyimpan nilai tulisan.
4) **Array**: menyimpan nilai variabel dengan indeks.
5) **Tuple**: seperti *array*, tetapi jumlahnya sudah pasti dan dengan tipe data tertentu yang boleh berbeda-beda.
6) **Enum**: menyimpan data enumerasi.
7) **Unknown**: menyimpan data yang kita belum tahu akan bertipe apa.
8) **Any**: menyimpan tipe data apa saja - tidak dilakukan pemeriksaan oleh TypeScript.
9) **Null**: menyimpan nilai "tidak ada nilai", terjadi jika suatu variabel dideklarasikan tetapi tidak mempunyai nilai.
10) **Undefined**: menyimpan nilai yang tidak didefinisikan, terjadi jika suatu variabel tidak dideklarasikan tetapi dirujuk dalam program.
11) **Union**: mempunyai kemungkinan lebih dari 1 tipe data.

Tipe *void, never*, dan *object* akan dibahas di pembahasan tentang fungsi dan OOP. Contoh deklarasi serta tipe penggunaan tipe data dasar di TypeScript dapat dilihat pada Listing 1.3.

```
let isFinished: boolean = false;
console.log(isFinished, typeof isFinished);                    // <1>
let price: number = 150.34;
console.log(price, typeof price);
let numOfEmployees: number = 25;
console.log(numOfEmployees, typeof numOfEmployees);
let progLang: string = "TypeScript";
console.log(progLang, typeof progLang)
let university: string[] = ['UT', 'UGM', 'ITB'];
console.log(university, typeof university)
let employee: [number, string, boolean, number, string];
employee = [1, "Zaky Aditya", true, 20, "Engineer"];
console.log(employee, typeof employee);
enum Color {
 Black = 2,
 Blue,
 Yellow,
 Green = 3,
 Red = 3 * 3
```

```
}
console.log(Color, typeof Color);
let code: string | number;
console.log(code, typeof code);
code = ‘my code’;
console.log(code, typeof code);
code = 21;
console.log(code, typeof code);
let valueNull = null
console.log(valueNull, typeof valueNull)
let valueUndefined = undefined
console.log(valueUndefined, typeof valueUndefined)
let valueAny: any;
console.log(valueAny, typeof valueAny)
valueAny = true;
console.log(valueAny, typeof valueAny)
valueAny = 42;
console.log(valueAny, typeof valueAny)
valueAny = “TypeScript”;
console.log(valueAny, typeof valueAny)
valueAny = [];
console.log(valueAny, typeof valueAny)
valueAny = {};
console.log(valueAny, typeof valueAny)
valueAny = Math.random;
console.log(valueAny, typeof valueAny)
valueAny = null;
console.log(valueAny, typeof valueAny)
valueAny = undefined;
console.log(valueAny, typeof valueAny)
let valueUnknown: unknown;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = true;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = 42;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = “TypeScript”;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = [];
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = {};
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = Math.random;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = null;
console.log(valueUnknown, typeof valueUnknown);
valueUnknown = undefined;
console.log(valueUnknown, typeof valueUnknown);
```

Listing 1.3
Tipe data dasar di TypeScript

typeof digunakan untuk menghasilkan nilai tipe data dari variabel yang bersangkutan.

c. Struktur Kendali

1) **Pencabangan dengan *if***

Digunakan untuk mengevaluasi suatu nilai (atau hasil ekspresi) dan melakukan sesuatu berdasarkan nilai yang dievaluasi tersebut. Contoh penggunaannya dapat dilihat di Listing 1.4.

```
let a: number = 21;
let b: number = 3;

let c:number = a / b;

if (c > 5) {
 console.log(“lebih besar dari 5”);
} else if (c > 3) {
 console.log(“antar 3 - 5”);
} else {
 console.log(“di bawah 3”);
}
```

Listing 1.4
Pencabangan dengan if

2) **Seleksi Kondisi dengan *switch***

Switch digunakan untuk mengevaluasi beberapa nilai dan melakukan sesuatu jika nilai tersebut sesuai. Contoh penggunaan dapat dilihat pada Listing 1.5.

```
let hari: number = 5;
let hariStr: string;

switch (hari) {
 case 0:
   hariStr = “Minggu”;
   break;
 case 1:
   hariStr = “Senin”;
   break;
case 2:
   hariStr = “Selasa”;
   break;
case 3:
   hariStr = “Rabu”;
   break;
case 4:
   hariStr = “Kamis”;
   break;
```

```
case 5:
  hariStr = “Jum’at”;
  break;
case 6:
  hariStr = “Sabtu”;
  break;
default:
  hariStr = “Tidak ada hari tersebut”;
  break;
}
console.log(hariStr);
```

Listing 1.5
Seleksi kondisi dengan switch

3) **Perulangan dengan *for***

Perulangan (*looping*) dengan *for* mempunyai 3 bentuk:

a. Perulangan sejumlah tertentu.
b. Perulangan untuk mengakses nilai dari *array* (*for ... of*).
c. Perulangan untuk mengakses indeks dari *array* (*for ... in*).

Penggunaan masing-masing perulangan tersebut dapat dilihat pada Listing 1.6

```
for (let i = 0; i < 3; i++) {                    // <1>
 console.log(“Looping ke “ + i);
}
let arr = [“nilai 1”, “nilai 2”, “nilai 3”, “nilai 4”];
for (var nilai of arr) {                         // <2>
 console.log(nilai);
}
for (var index1 in arr) {                        // <3>
 console.log(index1);
}
console.log(index1);                             // <4>

for (let index2 in arr) {                        // <5>
 console.log(index2);
}

// console.log(index2)                           // <6>

let str = “Universitas Terbuka”;

for (var huruf of str) {                         // <7>
 console.log(huruf);
}
```

Listing 1.6
Perulangan menggunakan for

1) Perulangan dengan jumlah pasti tertentu.
2) Perulangan untuk mengambil nilai *array*.
3) Perulangan untuk mengakses indeks, var digunakan untuk definisi variabel.
4) Variabel masih tetap tersedia setelah keluar dari perulangan.
5) Perulangan untuk mengakses indeks, let digunakan untuk definisi variabel.
6) Jika komentar dihilangkan akan *error* karena let tidak membuat variabel tetap dapat digunakan di luar perulangan.
7) Penggunaan *for ... of* untuk *string*.

Untuk mengkompilasi kode sumber di atas, gunakan spesifikasi ES6 (ES merupakan spesifikasi bahasa pemrograman Ecmascript yang diimplementasikan oleh JavaScript). Jika menggunakan spesifikasi *default* (`*tsc namafile.ts*`), maka akan terjadi kesalahan karena *default* hasil kompilasi akan menggunakan spesifikasi ES3 dan *for ... of* tidak didukung oleh ES3. Perintah untuk mengkompilasi dengan menggunakan ES6 adalah sebagai berikut:

```
$ tsc --target es6 for.ts
```

4) **Perulangan dengan *while***

Pernyataan *while* digunakan untuk perulangan sejumlah yang belum diketahui, sampai dengan suatu kondisi terpenuhi.

Ada 2 tipe pernyataan *while*:

a. ***while***: perulangan sampai kondisi terpenuhi, dapat tidak dilakukan jika evaluasi nilai *while* pertama sudah bernilai *false*.
b. ***do ... while***: perulangan sampai kondisi terpenuhi, dilakukan minimal sekali sampai dengan evaluasi nilai *while* bernilai *false*.

Penggunaan masing-masing perulangan tersebut dapat dilihat pada Listing 1.7.

```
let nilai1: number = 5;

while (nilai1 < 10) {                // <1>
  console.log( nilai1 )
  nilai1++;
}

let nilai2: number = 5;

while (nilai2 < 5) {                 // <2>
  console.log( nilai2 )
  nilai2++;
}

let nilai3: number = 5;
```

```
do {                                    // <3>
   console.log( nilai3 )
   nilai3++;
} while ( nilai3 < 10)

let nilai4: number = 5;

do {                                    // <4>
   console.log( nilai4 )
   nilai4++;
} while ( nilai4 < 5)
```

Listing 1.7
Perulangan menggunakan *while*

1) Selama *nilai1* lebih kecil dari 10 (*nilai1* awal adalah 5), akan ditampilkan. Hasil: 5 6 7 8 9.
2) Perulangan ini tidak dikerjakan karena saat evaluasi pertama sudah bernilai *false* (*nilai2* berisi 5, dibandingkan apakah lebih kecil dari 5).
3) Selama *nilai3* lebih kecil dari 10 (*nilai3* awal adalah 5), akan ditampilkan. Hasil: 5 6 7 8 9.
4) Perulangan ini hanya akan dikerjakan sekali saja, sebelum evaluasi *nilai4* kurang dari 5. Begitu menemui evaluasi tersebut, hasil evaluasi adalah *false*. Hasil: 5.

B. PEMROGRAMAN MODULAR DI TYPESCRIPT

Pemrograman modular merupakan suatu teknik perancangan *software* yang membagi program menjadi sekumpulan unit yang independen. Unit independen tersebut menyediakan fungsionalitas tertentu.

Di dalam TypeScript, unit tersebut disebut dengan *function* (fungsi). Kumpulan *fungsi* ini biasanya dimasukkan dalam suatu *modul* yang berisi berbagai *fungsi* dalam suatu domain permasalahan tertentu.

1. Mengenal Fungsi

Fungsi di dalam TypeScript merupakan unit terkecil yang menjadi bangunan dasar dari suatu aplikasi.

Ada 2 kategori *fungsi*:

a. ***Fungsi* pustaka standar**: disediakan oleh Node.js dan siap digunakan oleh JavaScript / TypeScript.
b. **UDF (*User-Defined Function*):** fungsi yang dibuat sendiri oleh pemrogram. Jika disediakan untuk pemrogram lainnya, maka kumpulan fungsi ini sering juga disebut pustaka pihak ketiga atau *third party library*.

2. Fungsi Pustaka Standar

Untuk keperluan ini, harus diinstal modul untuk membuat supaya pustaka standar Node.js dikenali dan dapat digunakan oleh TypeScript. Langkah-langkah untuk menggunakan pustaka standar adalah sebagai berikut:

```
$ npm init -y                                                    // <1>
...
...
  “author”: “”,
  “license”: “ISC”
}

$ ls                                                             // <2>
package.json
$ npm install @types/node --save-dev                             // <3>
...
...
+ @types/node@14.14.17
added 1 package from 44 contributors and audited 1 package in 5.597s
found 0 vulnerabilities

$ ts-node                                                        // <4>
> import * as os from ‘os’;                                      // <5>
{}
> of.userInfo()                                                  // <6>
{
  uid: 1000,
  gid: 1000,
  username: ‘bpdp’,
  homedir: ‘/home/bpdp’,
  shell: ‘/usr/bin/fish’
}
>
```

Listing 1.8
Fungsi Pustaka Standar

1) Inisialisasi proyek.
2) Hasil dari inisialisasi adalah file *package.json*.
3) Instalasi paket yang diperlukan: **@types/ode**.
4) Masuk ke REPL.
5) *Import* digunakan untuk mengaktifkan modul yang ingin digunakan (**os**).
6) Menggunakan fungsi **userInfo()** dari modul **os**.

Semua modul yang tersedia di repositori npm (https://www.npmjs.com/) dapat diakses dan diaktifkan dengan menggunakan cara yang telah disebutkan di atas.

3. Fungsi Buatan Sendiri

Fungsi buatan sendiri didefinisikan oleh pemrogram jika dirasakan tidak ada modul yang diinginkan sehingga harus membuat sendiri berbagai fungsionalitas tersebut. Untuk selanjutnya, pembahasan akan mengacu pada fungsi yang dibuat sendiri ini.

a. Membuat Fungsi

Fungsi dapat dibuat dengan menggunakan *function definition* dan *function expression*. Dari sisi penamaan, suatu fungsi juga dapat mempunyai nama (*named function*), dapat juga tidak mempunyai nama (*anonymous function*).

b. Function Definition

Dengan menggunakan kata kunci **function** di awal, maka berarti *fungsi* didefinisikan. Definisi serta penggunaan *fungsi* dapat dilihat pada Listing 1.9.

```
console.log(add(32,12));                          // <1>

function add(x: number, y: number): number {      // <2>
 return x + y;                                    // <3>
}

console.log(add(21,12));                          // <4>
```

Listing 1.9
Mendefinisikan Fungsi

1) Fungsi dapat dipanggil dari mana saja.
2) Definisi fungsi. **add** adalah nama fungsi, 2 argumen fungsi (x dan y) dengan tipe data *number*, hasilnya adalah *number* (:number).
3) Badan dari fungsi, tempat mendefinisikan fungsi.
4) Fungsi dipanggil atau digunakan.

Fungsi dapat dipanggil dari bagian mana saja karena fasilitas *hoisting*, yaitu kemampuan kompilator JavaScript/TypeScript untuk memindahkan semua definisi di bagian atas sebelum program dijalankan. Dengan demikian, tidak masalah jika definisi fungsi dilakukan di bagian akhir.

c. Function Expression

Dengan menggunakan *function expression*, definisi dari fungsi dapat diletakkan pada bagian ekspresi. Sebagai suatu ekspresi, fungsi akan dijalankan saat kompilator menemui *function expression* sehingga tidak dapat digunakan sebelum dibuat. Listing 1.10 adalah contoh *function expression*.

```
//console.log(2, 3);                                    // <1>

let add = function (x: number, y: number): number {     // <2>
 return x + y;                                          // <3>
};

console.log(add(2, 3));                                 // <4>
```

Listing 1.10
Membuat fungsi menggunakan *function expression*

1) Jika dihilangkan komentar, akan terjadi *error* karena pada baris ini belum ditemui nama fungsi **add**.
2) Membuat fungsi **add** sebagai ekspresi dari variabel **add**.
3) Badan dari fungsi, tempat mendefinisikan fungsi.
4) Baru dapat dipanggil jika sudah didefinisikan di bagian atas.

d. Nilai Kembalian Fungsi

Pada penjelasan pembuatan program di atas, bagian **:number** setelah definisi argumen ***x*** dan ***y*** merupakan nilai kembalian yang diharapkan. Bagian tersebut dapat berisi tipe data yang telah dibahas sebelumnya. Selain itu, terdapat dua tipe nilai kembalian yang digunakan pada kondisi tertentu:

1) **void**: nilai kembalian dari fungsi tersebut tidak ada atau suatu fungsi yang tidak menghasilkan nilai. Contoh dapat dilihat pada Listing 1.11.
2) **never**: tipe data untuk nilai kembalian suatu fungsi yang tidak pernah dicapai. Contoh dapat dilihat pada Listing 1.12.

```
function tampilkan(arg1: any): void {
  console.log(arg1)
}

tampilkan("Halo!");
```

Listing 1.11
Nilai kembalian *void*

```
nonStop();

function nonStop(): never {
 while (true) {
  console.log('tulisan non-stop');
 }
}
```

Listing 1.12
Nilai kembalian *never*

e. Arrow Function

Tanda panah => digunakan sebagai *arrow function*, yaitu sintaks untuk membuat suatu fungsi menjadi terlihat lebih ringkas. Penggunaannya dapat dilihat pada Listing 1.13.

```
let add1 = (a: number, b: number): number => { return a + b };      // <1>
let add2 = (a: number, b: number): number  => a + b;                // <2>
let lenStr1 = (s: string): number => s.length;                      // <3>
let lenStr2 = s => s.length;                                        // <4>

// Penggunaan arrow func                                            // <5>
console.log(add1(2,3));
console.log(add2(2,3));
console.log(lenStr1('abcdefg'));
console.log(lenStr2('abcdefg'));
```

Listing 1.13
Penggunaan _arrow function_ (=>)

1) Ekspresi lengkap dengan nama **add1**, argumen *a* dan *b* bertipe *number*, nilai kembalian *number* (**:number**), serta definisi fungsi di dalam *{...}*.
2) Penulisan lebih singkat.
3) Penulisan singkat.
4) Jika tanpa tipe, akan lebih singkat lagi.
5) Berbagai penggunaan dari *arrow function*.

4. Fungsi *Anonymous*

Fungsi anonymous adalah *fungsi* yang tidak mempunyai nama. Biasanya digunakan sebagai ekspresi yang dapat langsung dijalankan, tidak dimaksudkan untuk penggunaan berikut-berikutnya, serta memungkinkan untuk digunakan sebagai parameter fungsi (*higher order function*). Penggunaan dari fungsi *anonymous* ini dapat dilihat pada Listing 1.14.

```
((str: string, idx: number) => {                                    // <1>
 console.log(str[idx])                                              // <2>
})('Universitas Terbuka', 4);                                       // <3>

let idxStr = function(str: string, idx: number): void {             // <4>
 console.log(str[idx]);                                            // <5>
}

console.log(idxStr('Universitas Terbuka', 4));                      // <6>
```

Listing 1.14
Fungsi Anonymous

1) Membuat fungsi dengan argumen ***str*** dan ***idx***. Kegunaannya untuk mengambil karakter ke ***idx*** dari suatu string ***str***.
2) Badan dari fungsi.
3) Sekaligus memanggil fungsi dengan argumen tertentu.
4) Membuat ekspresi fungsi yang diberikan ke variabel ***idxStr*** dengan argumen string ***str*** dan indeks ke berapa yang akan diambil (***idx*** berupa *number*).
5) Badan dari fungsi.
6) Penggunaan fungsi.

5. let, var, dan Ruang Lingkupnya

Setelah memahami fungsi, kita dapat lebih memahami penggunaan *let* dan *var*. Menggunakan *let* lebih baik karena *let* dikenali di ruang lingkup tempat variabel tersebut didefinisikan. Jika berada pada suatu fungsi maka variabel tersebut hanya dikenal di dalam fungsi tersebut serta ruang lingkup yang lebih kecil. Berbeda dengan *var* yang sekali didefinisikan akan dapat digunakan di mana saja dan dapat mengakibatkan kita memanipulasi nilai yang seharusnya tidak dimanipulasi; khususnya pada suatu proyek yang mempunyai ruang lingkup yang besar dengan kode sumber besar. Tentu saja *let* dan *var* ini akan mempunyai efek yang sama jika *let* dilakukan pada ruang lingkup global *script* kode sumber, bukan di level *fungsi*. Contoh dapat dilihat pada Listing 1.15.

```
let scoping = function(input: any) {              // <1>

 let angka1 = 100;

 if (typeof input == 'number') {
   let angka2 = angka1 + input;                   // <2>
   var angka3 = angka1 - input;                   // <3>
 }

 // Error: Cannot find name 'angka2'
 // return angka2;                                // <4>
 // karena definisi dengan var, maka
 // dapat diakses
 return angka3;                                   // <5>

}

console.log(scoping(1));
```

Listing 1.15
Ruang lingkup *let* dan *var*

1) Mendefinisikan ekspresi fungsi.
2) Mendefinisikan variabel dalam blok *if* menggunakan *let*.
3) Mendefinisikan variabel dalam blok *if* menggunakan *var*.

d. *Error* karena variabel didefinisikan menggunakan *let*, hanya dikenali di blok tempat variabel tersebut didefinisikan.

e. Tidak *error* karena didefinisikan menggunakan *var*, dapat dikenali di seluruh blok ekspresi *fungsi*.

C. INTERFACE

Interface adalah struktur yang digunakan sebagai kontrak untuk definisi (data, fungsi, maupun *class* di OOP). Penggunaannya dapat dilihat pada Listing 1.16.

```
interface IPerson {                                          // <1>
 nik: string;
 nama: string;
 alamat: string;
 menikah: boolean;
}

interface IPegawai extends IPerson {                         // <2>
 readonly npp: string;
 jabatan: string;
 gaji: number;
 email?: string;
}

let peg01: IPegawai = {                                      // <3>
 nik: '012345',
 nama: 'Donal',
 alamat: 'Jl. Awan Biru 21',
 menikah: true,
 npp: '98123',
 jabatan: 'Manager SDM',
 gaji: 15000000
}

console.log(peg01.nama, peg01.jabatan);      // <4>
// error: Cannot assign to 'npp' because it is a read-only property
// peg01.npp = '981234';

interface IKamusList {                                       // <5>
   [index:string]:string
}

let strKamus: IKamusList = {};                               // <6>
strKamus["university"] = "universitas";
strKamus["freedom"] = "merdeka";

console.log(strKamus["university"]);

interface IPemrosesNilai {                                   // <7>
 (kunci: number,
 nilai: string ): void
}
```

```
function tambahNilai (kunci: number, nilai: string): void {          // <8>
 console.log('Menambah ', kunci, nilai);
}

function perbaruiNilai (kunci: number, nilaiBaru: string): void {   // <9>
 console.log('Memperbarui ', kunci, nilaiBaru);
}

let pemrosesTambah: IPemrosesNilai = tambahNilai;                   // <10>
pemrosesTambah(123, 'Nilai 123');

let pemrosesPerbarui: IPemrosesNilai = perbaruiNilai;
pemrosesPerbarui(123, 'Nilai baru 123');
```

Listing 1.16
Penggunaan *Interface*

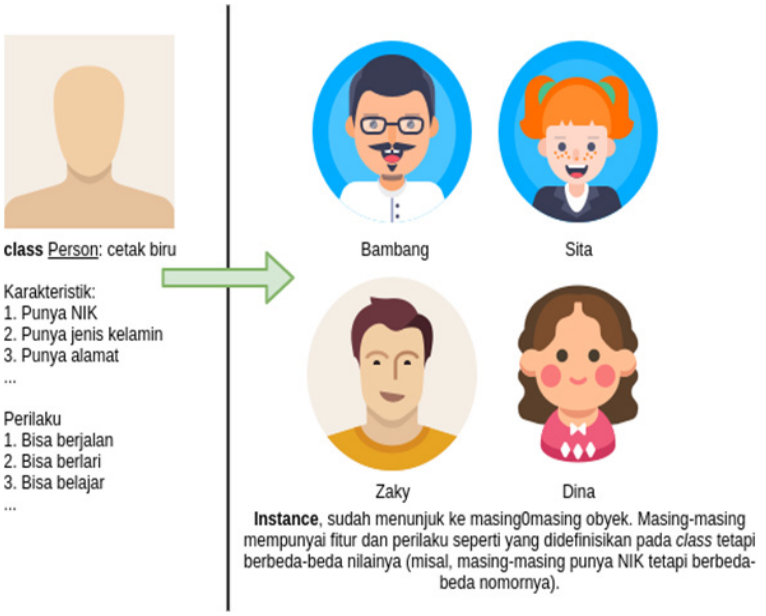
1) Mendefinisikan *interface* untuk kontrak **person**, pada umumnya pola penamaannya "I" + nama (dengan huruf pertama besar).
2) Mendefinisikan *interface* yang merupakan turunan dari definisi sebelumnya.
3) Membuat data dengan kontrak *interface* tertentu.
4) Mengakses data.
5) Mendefinisikan *interface* untuk *array* dengan indeks string.
6) Membuat data sesuai kontrak *interface*.
7) Mendefinisikan *interface* untuk fungsi.
8) Mendefinisikan fungsi yang akan dipastikan kontraknya dengan *interface*.
9) Mendefinisikan fungsi yang akan dipastikan kontraknya dengan *interface*.
10) Membuat ekspresi fungsi sesuai *interface*.

D. OOP DI TYPESCRIPT

1. Pengertian OOP

OOP (*Object-Oriented Programming* atau Pemrograman Berorientasi Obyek) adalah suatu paradigma pemrograman yang menyelesaikan masalah dengan melakukan abstraksi terhadap berbagai entitas (yang mempunyai *properties*/fitur dan *behaviour*/perilaku dalam suatu sistem serta aliran kerja dari sistem tersebut.

Ada banyak cara untuk mengimplementasikan OOP. Salah satu cara adalah dengan menggunakan *class* dan kemudian membuat *instance*/obyek dari *class* tersebut. *Class* merupakan cetak biru dari sekumpulan obyek. Proses untuk membentuk *class* dari sekian banyak obyek dengan mencari fitur serta perilaku yang sama/sejenis disebut dengan **meng-abstraksi**. Keterkaitan antara *class* dengan *instance* dapat dilihat pada Gambar 1.3.



Gambar 1.3
Class dan *Instance*

2. Definisi *Class* dan *Instance*

TypeScript mendefinisikan *class* dengan menggunakan kata kunci **class**. Untuk mendefinisikan *instance* digunakan kata kunci **new**. Setiap *class* mempunyai berbagai fitur (dengan sifat *public*, *private*, serta *protected*). Perilaku yang dimiliki oleh kelas didefinisikan menggunakan *method* (mirip dengan fungsi tetapi khusus untuk *class*). *Method* khusus yang dijalankan pada saat *instance* dibuat disebut *constructor*. *Class* dapat diturunkan ke *subclass* (disebut dengan *inheritance*). Pembuatan atas definisi *class* serta *instance* dapat dilihat pada Listing 1.17.

```
class Person {                                          // <1>
  perNik: string;                                       // <2>
  protected perNama: string;
  perAlamat: string;

  constructor(nik: string, nama: string) {              // <3>
   this.perNik = nik;
   this.perNama = nama;
  }

}

class Pegawai extends Person {                          // <4>
  pegNpp: string;                                       // <5>
  private _pegJmlTanggungan: number;
  readonly dept: string;
  gaji: number;
```

```
static potongPajak = 10;                                    // <6>

  constructor(nik: string, npp: string, // <7>
              nama: string, dept: string) {
    super(nik, nama);
    this.pegNpp = npp;
    this.dept = dept;
  }

  getGaji(): number {                                       // <8>
    return this.gaji;
  }

  setGaji(gajiBaru: number): void {                         // <9>
    this.gaji = gajiBaru;
  }

  getPotonganPajak(): number {                              // <10>
    return this.gaji * (Pegawai.potongPajak / 100);
  }

  presensi(): void {                                        // <11>
    let dateTime = new Date();
    console.log("Presensi pada " +
                dateTime.toLocaleTimeString() +
                ' - ' + dateTime.toDateString());
  }

}

let pakBambang = new Pegawai('nik1122', 'npp123',           // <12>
                             'Bambang Purnomosidi', 'IT');

console.log(pakBambang.setGaji(15750500));                  // <13>
console.log(pakBambang.getGaji());
console.log(pakBambang.presensi());
console.log(pakBambang.getPotonganPajak());
// error:
// Property '_pegJmlTanggungan' is private and only
//    accessible within class 'Pegawai'
// console.log(pakBambang._pegJmlTanggungan);
```

Listing 1.17
Definisi *Class* dan *Instance*

1) Mendefinisikan *class* Person.
2) Mendefinisikan *properties* dari *class* Person, semuanya *public* kecuali *perNama* yang mempunyai *access modifier protected* - dapat diakses dari *class* yang mendefinisikan dan turunannya.

3) Konstruktor dari Person, akan dikerjakan pertama kali saat membuat *instance* Person.
4) Mendefinisikan *class* Pegawai yang merupakan turunan dari *class* Person.
5) Mendefinisikan *properies* dari *class* Pegawai. *Private* berarti hanya dapat diakses di *class* tersebut, *readonly* berarti hanya dapat dibaca - dapat ditetapkan hanya saat deklarasi dan definisi di konstruktor.
6) Property *static* berarti ada di level *class* bukan di level *instance*.
7) Konstruktor untuk Pegawai. Penggunaan *super* berarti mengacu pada konstruktor induknya.
8) Kelas menyediakan *getter* untuk mengambil nilai pada suatu *class*.
9) Kelas menyediakan *setter* untuk menetapkan suatu nilai.
10) Mendefinisikan *method*, *property static* dimana *property* yang berada pada sisi *class* dapat diakses dengan *namaClass.property*.
11) Mendefinisikan *method*.
12) Pembuatan *instance* baru. Argumen sesuai konstruktor.
13) Menjalankan berbagai *method* serta *setter* dan *getter*.

E. GENERICS

Generics merupakan suatu teknik pemrograman yang digunakan untuk membuat tipe data yang ada pada kode fungsi kita se-generik/se-umum mungkin sehingga kemudian dapat digunakan untuk berbagai macam tipe data. *Generics* diperlukan untuk membuat kode kita *reusable* dengan cara tidak menetapkan secara kaku untuk suatu jenis tipe data yang akan diproses di depan (di langkah berikutnya). Sebagai gantinya, tipe data yang akan diproses baru akan didefinisikan pada saat akan menggunakan *fungsi* tersebut. Bagian tipe data yang menandai *generics* dibuat dengan menggunakan satu huruf besar.

Contoh penggunaan *generics* dapat dilihat pada Listing 1.18 berikut ini :

```
// tanpa generics                                          // <1>
function funN(argN: number): number {
 return argN;
}

function funS(argS: string): string {
 return argS;
}

console.log(funN(23));
console.log(funS('TypeScript'));

// tanpa generics - menggunakan any
// tidak type safe, karena menerima tipe
// data apa saja.
function funA(argA: any): any{                             // <2>
 return argA;
}
```

```
console.log(funA(true));
console.log(funA([1,2,3]));

// menggunakan generics
function funG1<T>(argG1:T):T {                                    // <3>
 return argG1;
}

function funGn<T, U>(argGn1:T, argGn2: U): U {                    // <4>
 return argGn2;
}

console.log(funG1<string>('TypeScript'));
console.log(funG1<number>(23));

console.log(funGn<string, number>('TypeScript', 23));

interface argGenConstraint {                                      // <5>
 length: number;
}

function panjang<T extends argGenConstraint>(argGC:T): number {   // <6>
 return argGC.length;
}

let hasil1 = panjang({ length: 23, name: 'TypeScript'});          // <7>
let hasil2 = panjang('TypeScript');

console.log(hasil1);
console.log(hasil2);
```

Listing 1.18
Penggunaan *Generics*

1) Terdapat 2 (dua) fungsi tanpa *generics* yang boleh digunakan untuk memproses argumen *string* serta *number*. Jika tanpa *generics* maka harus didefinisikan secara langsung dengan tipe data yang dikehendaki agar dapat diproses; sementara itu bisa saja terjadi adanya kemiripan pada proses badan fungsi; sehingga keadaan ini dapat menyebabkan program yang kita buat menjadi bertele-tele.
2) Tanpa *generics* hanya menggunakan 1 (satu) fungsi saja. Penggunaan argumen *any* memang memungkinkan, namun tidak *type safe* (aman) karena tipe data yang diproses dapat saja tidak sesuai dengan tipe data dari pemroses pada badan fungsi; sehingga dengan kata lain dapat dinyatakan bahwa masih terdapat unsur ketidakpastian.
3) Menggunakan *generics*, cukup mendefinisikan 1 (satu) fungsi. Huruf **T** digunakan untuk menandai tipe data yang digunakan pada argumen nantinya. Contoh pemakaiannya ada di bagian bawah. **T** merupakan tanda *generics*, dapat diganti dengan huruf lainnya.

4) *Generics* dapat lebih dari satu, dengan nilai kembalian sesuai dengan yang kita inginkan.
5) Seringkali kita harus memastikan suatu *constraint* tertentu yang dapat digunakan untuk memeriksa apakah penggunaan suatu fungsi sudah sesuai atau belum. Dalam kasus ini, kita memastikan bahwa setiap argumen mempunyai *property* ***length***. Definisi dari *constraint* tersebut dibuat dengan menggunakan *interface*.
6) Untuk menerapkan *contraint* pada suatu *fungsi*, digunakan perintah **extends** sesuai dengan *interface* yang kita inginkan untuk diterapkan.
7) Dua baris berikut digunakan untuk memberi contoh bahwa setiap argumen harus mempunyai *property* ***length***. Untuk contoh kedua, tidak perlu didefinisikan karena suatu *string* mempunyai *property* ***length***.

Latihan

Untuk memperdalam pemahaman Anda mengenai materi di atas, kerjakanlah latihan berikut!

Selain menggunakan *tsc*, cobalah semua kode sumber yang ada pada kegiatan belajar ini dengan menggunakan *Deno*.

Perhatikan baik-baik, adakah kode sumber mana yang tidak dapat dijalankan dengan menggunakan Deno?

Petunjuk Jawaban Latihan

Kunjungi web Deno di https://deno.land/. Install Deno sesuai petunjuk pada URL tersebut. Setelah itu, jalankan berbagai kode sumber TypeScript yang ada pada kegiatan belajar ini dengan menggunakan petunjuk di manual (https://deno.land/manual), khususnya pada **Getting Started - Command line interface**.

Rangkuman

TypeScript adalah bahasa pemrograman yang merupakan *superset* dari JavaScript. TypeScript mengenali dan dapat memproses sintaksis JavaScript. TypeScript melakukan berbagai penambahan pada sintaksis JavaScript sehingga berbagai kelemahan JavaScript yang selama ini menjadi keluhan dari berbagai pengembangan perangkat lunak dapat diperbaiki.

Kode sumber TypeScript akan dikompilasi menjadi JavaScript yang dapat dijalankan dengan menggunakan Node.js. Kompilator TypeScript dapat digunakan untuk mengkompilasi kode sumber JavaScript maupun TypeScript.

Sebagai suatu bahasa pemrograman, TypeScript juga mempunyai konstruksi tersendiri yang sebenarnya merupakan perluasan dari JavaScript. Beberapa hal yang diubah dan diperbaiki dari JavaScript antara lain adalah penggunaan *static typing* dalam kode sumber, berbagai tipe data, penggunaan *let* dan *variable scoping*, OOP, *interface*, serta fasilitas modul. Kegiatan belajar ini mempelajari berbagai sintaks dan konstruksi dasar dari TypeScript, terutama bagian-bagian yang membedakan dengan JavaScript.

Tes Formatif 2

Pilihlah satu jawaban yang paling tepat!

1) TypeScript dapat dijalankan dengan menggunakan, *kecuali* ….
   A. deno
   B. node.js
   C. tsc
   D. tsc-node

2) Hasil dari **let a = 21; let a = 22; console.log(a);** adalah ….
   A. 21
   B. 22
   C. *error*
   D. 2122

3) Jika kita mempunyai data lebih dari satu, jumlahnya pasti, dan antara satu data dengan data lainnya mempunyai tipe yang berbeda, maka sebaiknya variabel yang kita gunakan adalah variabel dengan tipe ….
   A. array
   B. string
   C. set
   D. tuple

4) Jika kita masih ragu-ragu dengan kemungkinan tipe data, maka kita dapat menggunakan ….
   A. *generics*
   B. *array*
   C. *tuple*
   D. *union*

5) Untuk mengakses nilai dari suatu *array* dalam suatu perulangan, digunakan ….
   A. for ... in
   B. for ... of

C. do ... while
D. while ... do

6) Kemampuan untuk meletakkan definisi dari fungsi di bagian manapun dari kode sumber yang dibuat dalam TypeScript adalah ….
A. *hoisting*
B. *function expression*
C. *function declaration*
D. *higher Order Function*

7) Jika suatu fungsi tidak mempunyai nilai kembalian, maka tipe data yang digunakan untuk nilai kembalian adalah ….
A. *null*
B. *void*
C. *never*
D. *any*

8) Pada kode sumber **let lenStr2 = s => s.length;**, bagian yang merupakan argumen adalah ….
A. lenStr2
B. =>
C. s.length
D. s

9) Bagian dari *class* yang akan langsung dijalankan saat membuat *instance* baru adalah ….
A. *constructor*
B. *property*
C. *setter* dan *getter*
D. tidak ada, semua *method* baru akan dijalankan saat di-*invoke*.

10) Suatu *generics* dapat di-*extend* untuk keperluan *filtering* kondisi tertentu yang dianggap valid untuk memanggil suatu *fungsi* dengan cara ….
A. memfilter melalui *constructor*
B. menggunakan *extends*
C. menggunakan *interface* dan *extends*
D. tidak dapat dilakukan dengan menggunakan TypeScript

Cocokkanlah jawaban Anda dengan Kunci Jawaban Tes Formatif 2 yang terdapat di bagian akhir modul ini. Hitunglah jawaban yang benar. Kemudian, gunakan rumus berikut untuk mengetahui tingkat penguasaan Anda terhadap materi Kegiatan Belajar 2.

$$\text{Tingkat Penguasaan} = \frac{\text{Jumlah Jawaban yang Benar}}{\text{Jumlah Soal}} \times 100$$

Arti tingkat penguasaan

<70%	70% - 79%	80% - 89%	90% - 100%
kurang	cukup	baik	baik sekali

Apabila mencapai tingkat penguasaan 80% atau lebih, Anda dapat meneruskan dengan Kegiatan Belajar 3. Bagus! Jika masih di bawah 80%, Anda harus mengulangi materi Kegiatan Belajar 2, terutama bagian yang belum dikuasai.

Kegiatan Belajar 3

Pemrograman *Typescript* (Lanjut)

A. ASYNCHRONOUS PROGRAMMING

Asynchronous programming banyak digunakan di berbagai bahasa pemrograman termasuk pada JavaScript dan TypeScript. *Asynchronous programming* sering disebut juga dengan *non-blocking I/O*. Di TypeScript, *asynchronous programming* ini didukung mulai dari versi 1.7. Sub kegiatan belajar ini akan mempelajari konsep dari *asynchronous programming* serta implementasi dari konsep tersebut dengan menggunakan TypeScript.

1. Memahami *Asynchronous Programming*

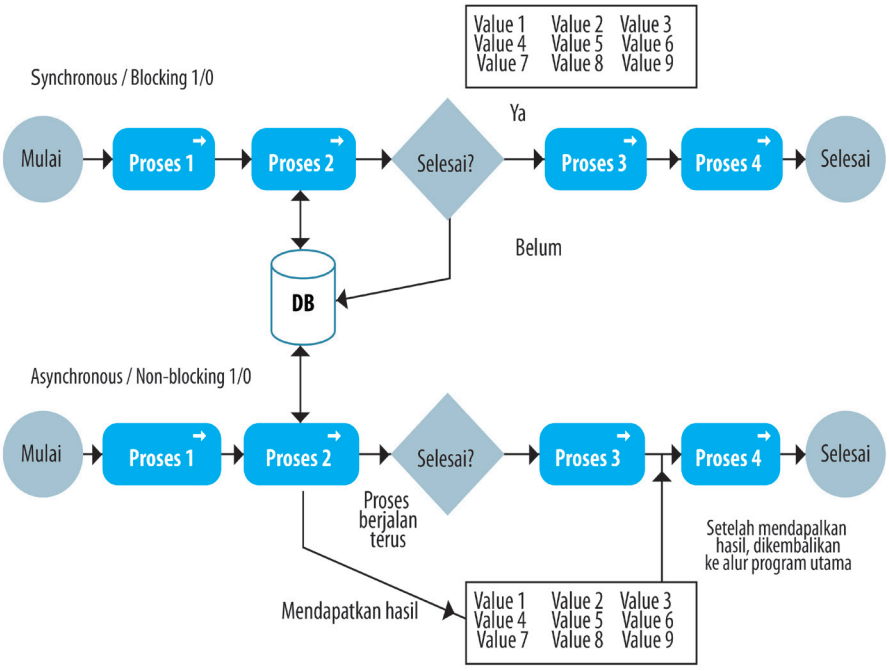
Asynchronous programming adalah suatu teknik pemrograman konkuren yang digunakan untuk menangani kondisi independen dari aliran utama program. Istilah pemrograman konkuren digunakan untuk mengacu pada suatu kondisi pengerjaan lebih dari satu tugas pada suatu saat bersamaan; khususnya jika terdapat bagian yang memerlukan akses / sumber daya yang besar dan kemungkinan memerlukan waktu yang cukup signifikan untuk menyelesaikan tugas tersebut.

Pada dasarnya terdapat dua kemungkinan untuk menangani hal tersebut:

a. ***Blocking I/O*** atau sering disebut juga *synchronous programming*, yaitu menunggu saat suatu proses telah selesai dikerjakan, baru setelah itu melanjutkan ke proses berikutnya. Jika belum selesai, maka proses berikutnya akan menunggu. Hal ini sebenarnya hanya cocok untuk situasi bahwa jika antar proses tersebut terdapat ketergantungan, artinya proses 2 tidak dapat dikerjakan kalau proses 1 belum selesai, proses 3 tidak dapat dikerjakan jika proses 2 belum selesai, dan seterusnya.

b. ***Non-blocking I/O*** atau sering disebut juga dengan *asynchronous programming*, yaitu proses akan tetap berjalan di alur aplikasi utama meskipun terdapat suatu proses yang mungkin memerlukan sumber daya besar dan waktu yang cukup signifikan untuk mengakses/ memprosesnya. Pada saat sampai di proses tersebut, proses akan tetap diteruskan; sementara itu proses yang memerlukan sumber daya besar tersebut juga tetap mengerjakan sehingga proses-proses tersebut berjalan secara konkuren meskipun CPU mengerjakan secara bergantian tetapi perpindahan pengerjaan antar proses tersebut tidak terasa. Setelah selesai mengerjakan, hasil akan dikembalikan ke alur utama.

Gambaran dari kedua hal tersebut dapat dilihat pada Gambar 1.4.



Gambar 1.4
Perbandingan Synchronous Programming Dengan
Asynchronous Programming

2. Implementasi *Asynchronous Programming* pada TypeScript

Pada umumnya, TypeScript menggunakan teknik *Promise* serta *Async/Await* untuk mengelola *asynchronous programming*. Sebenarnya dapat juga menggunakan *callback* tetapi *callback* ini sama sekali tidak disarankan karena kompleksitas dari *callback* pada saat proses *debugging* sehingga tidak dibahas di buku materi pokok ini. Pada BMP ini proses *Async/Await* menggunakan teknik *Promise* dalam proses implementasinya meskipun dapat juga tidak secara eksplisit menggunakan *Promise*.

a. Promise

Sama seperti halnya dalam kehidupan sehari-hari, *promise* atau *janji* merupakan suatu kejadian yang diharapkan akan terjadi di masa yang akan datang. Normalnya, janji akan dipenuhi tetapi belum tentu akan terjadi seperti yang dijanjikan. Kondisi semacam ini sering kali terjadi saat kita membuat *software*. Contoh:

1) Mengakses suatu *endpoint* di Internet (URL tertentu). Normalnya kita dapat mengakses *endpoint* tersebut, tetapi dapat saja Internet mati, atau alamat URL tersebut berpindah, dan kemungkinan-kemungkinan lain.
2) Membaca suatu *file* di lokasi tertentu. Normalnya kita dapat membaca *file* tersebut, namun bisa terjadi *file* tersebut rusak, atau sudah dihapus, atau tidak memiliki hak akses untuk membaca, dan lain-lain.

3) Melakukan *query* ke suatu DBMS. Normalnya kita dapat melakukan *query* dan berharap mendapatkan hasilnya, tetapi dapat saja *query* kita salah, atau hak akses kita dibatasi, dan lain-lain.

Dalam kondisi-kondisi seperti tersebut di atas, seorang pemrogram harus mengantisipasi dengan menggunakan konstruksi bahasa pemrograman tertentu.

Ada 2 hal yang harus diantisipasi terkait *asynchronous programming* ini, yaitu:

1) Kemungkinan latensi atau waktu akses yang relatif lama.
2) Kemungkinan kegagalan operasi, baik di awal maupun setelah terjadi proses dan kegagalan ini tidak dapat kita prediksi di awal.

Dengan menggunakan TypeScript, kita dapat menggunakan *Promise* maupun *async/await* untuk mengantisipasi kedua hal tersebut. Contoh dari *Promise* dapat dilihat pada Listing 1.19.

```
function getAngkaAcak(max: number): number {                    // <1>
 return Math.floor(Math.random() * Math.floor(max));
}

let p = new Promise<unknown>((resolve, reject) => {             // <2>

 let acak = getAngkaAcak(1000);

 if (acak > 500) {
   resolve(true);                                               // <3>
   return;
 }

 reject("Hasil < = 500");                                       // <4>

});

console.log(p);                                                 // <5>
p.catch(err => console.log("ERROR - ", err));                   // <6>
// Berbagai kemungkinan saat run:
// Promise { true }
// -> jika angka random lebih besar dari 500
// Promise { <rejected> 'Hasil < = 500' }
// ERROR -  Hasil < = 500
// -> jika yang terjadi adalah reject
```

Listing 1.19
Penggunaan *Promise* Untuk *Asynchronous Programming*.

1) Mendefinisikan *fungsi* **getAngkaAcak** yang akan digunakan untuk mendapatkan angka acak dengan nilai maksimal tertentu. *Fungsi* ini akan digunakan pada baris-baris berikutnya.

2) Membuat obyek *Promise* dengan *generics* kita isi *unknown* karena kita belum mengetahui hasil dari proses.
3) Disini terdapat 2 argumen yakni *resolve* dan *reject*.
4) *Resolve* akan dikerjakan jika *promise is fulfilled* atau janji ditepati atau proses yang kita inginkan terjadi di masa yang akan datang selesai dengan baik. Jika terjadi sesuatu *error*, maka yang akan dikerjakan adalah *reject*.
5) Jika hasil sesuai dengan yang diinginkan (acak > 500), maka yang akan dikerjakan adalah *resolve*, dalam hal ini hanya mengembalikan nilai *true*.
6) Jika hasil tidak sesuai dengan yang diinginkan (acak <= 500), maka yang akan dikerjakan adalah *reject*.
7) Menampilkan hasil.
8) Jika terjadi *error* / *reject*, maka bagian tersebut dapat kita antisipasi dengan *catch*.

b. Promise Chaining

Namun ada kalanya, pengerjaan satu tindakan dalam *Promise* saja tidak cukup, tetapi diperlukan suatu rangkaian yang akan menyusul proses yang telah berhasil dikerjakan tersebut. Sebagai contoh, kita dapat membaca suatu *file* CSV kemudian mengambil kolom pertama baris pertama kemudian mentransformasikan isinya sesuai dengan rumus tertentu. Dalam kasus seperti ini, diperlukan *chaining* atau merangkai satu proses dengan proses lainnya. Hasil satu proses merupakan *input* bagi proses berikutnya, demikian seterusnya. TypeScript memungkinkan melakukan hal tersebut dengan menggunakan *promise chaining*, seperti yang tertera pada Listing 1.20

```
function getAngkaAcak(max: number): number {
 return Math.floor(Math.random() * Math.floor(max));
}

let p = new Promise<number>((resolve, reject) => {

 let acak = getAngkaAcak(20);
 let hasil: number;                                    // <1>

 if (acak > 10) {
   resolve(acak);
   return;
 }

 reject("Hasil < = 10");

}).then(function(hasil) {                              // <2>
 console.log(hasil);                                   // <3>
 return hasil*2;                                       // <4>
}).then(function(hasil) {                              // <5>
 console.log(hasil);                                   // <6>
 return hasil*2;
```

```
}).then(function(hasil) {
 console.log(hasil);
 return hasil*2;
});

console.log(p);
p.catch(err => console.log("ERROR - ", err));

// run (dapat berbeda-beda):
//  pertama (jika resolve):
//    Promise { <pending> }
//    12
//    24
//    48
//  kedua (jika reject):
//    Promise { <pending> }
//    ERROR -  Hasil < = 10
```

Listing 1.20
Promise Chaining

1) Menyiapkan variabel *hasil* yang nantinya akan digunakan untuk menampung hasil dari bilangan acak jika sesuai dengan kondisi (> 10).
2) *Chain* pertama, jika > 10 maka hasil bilangan acak tersebut akan diletakkan ke variabel hasil.
3) Setelah itu isi variabel hasil akan ditampilkan.
4) Variabel hasil kemudian dikalikan 2. Hasil perkalian tersebut dijadikan nilai kembalian yang akan digunakan oleh *chain* berikutnya. Misal acak berisi 12, maka hasil akan diisi 12 dan kemudian ditampilkan serta dikalikan 2. Hasil perkalian tersebut (24) akan dijadikan nilai kembalian dan menjadi masukan bagi *chain* berikutnya.
5) Mengambil nilai dari nilai kembalian *chain* sebelumnya, yaitu 24 dan kemudian dimasukkan ke variabel hasil.
6) Menampilkan nilai variabel hasil (24).
7) Sama dengan penjelasan nomor 4, tetapi untuk *chain* berikutnya.

c. Async/Await

Konstruksi *async/await* pada dasarnya digunakan untuk pemanis sintaksis dari *asynchronous programming* yang dirasakan relatif membuat kode sumber yang dibuat susah dibaca.

Pola pikir untuk menggunakan konstruksi ini adalah:

1) Buat *fungsi* yang akan di dalam badan fungsi tersebut terdapat operasi *asynchronous*.
2) Letakkan *async* di depan *function*.
3) Di dalam badan *fungsi* tersebut, gunakan *await* untuk operasi yang memerlukan penanganan *asynchronous*.

4) Letakkan *await* dalam kerangka *try ... catch*. Jika berhasil maka bagian *try* akan dikerjakan. Jika *error*, maka bagian *catch* akan menangkap *error* tersebut. Contoh dari penggunaan *async/await* dapat dilihat pada Listing 1.21.

```
function getAngkaAcak(max: number): number {                          // <1>
 return Math.floor(Math.random() * Math.floor(max));
}

function lebihDari(max: number, angka: number): boolean | number {    // <2>
 if (angka > max) {
   throw “ERR: arg 1 harus lebih besar daripada arg 2”
 }
 let acak = getAngkaAcak(max);
 if (acak > angka) {
   return true;
 } else {
   return acak;
 }
}

let p = async function (): Promise<boolean | number> {                // <3>

 try {

   //let hasilOK: boolean | number = await lebihDari(100, 500);       // <4>
   let hasilOK: boolean | number = await lebihDari(1000, 500);       // <5>
   return hasilOK;

 } catch(error) {                                                     // <6>

   return error;

 }

};

(async () => {                                                        // <7>
 console.log(await p())
})()

// jika <4> di uncomment, maka hasil:
//     ERR: arg 1 harus lebih besar daripada arg 2
// jika menggunakan <5>, maka hasil kemungkinan:
//     true
//     134 (atau angka lain)
```

Listing 1.21
Penggunaan *Async/Await*

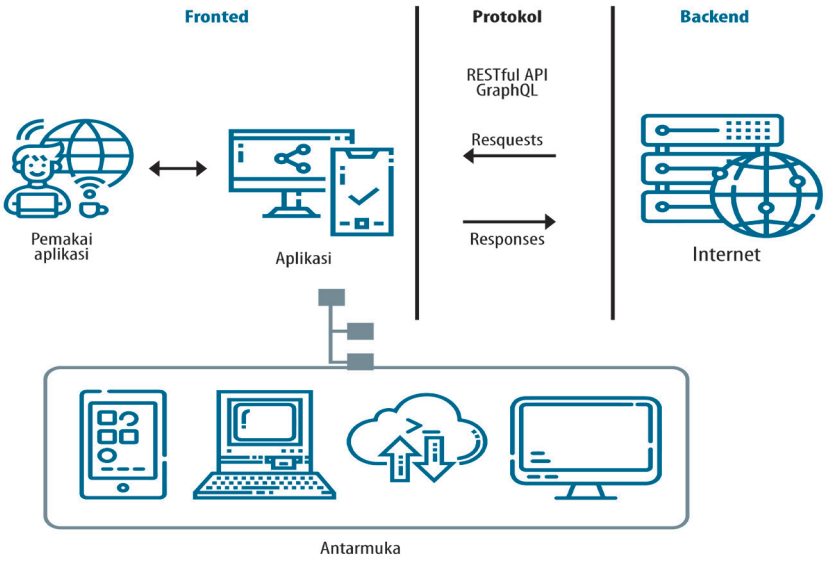
1) Mendefinisikan *fungsi* untuk mencari nilai acak dengan nilai maksimal tertentu.
2) Mendefinisikan *fungsi* untuk membandingkan apakah suatu nilai acak dengan nilai maksimal tertentu (argumen 1) lebih besar daripada nilai tertentu (argumen 2). Jika penggunaan *fungsi* ini salah maka akan dihasilkan *error* (*throw*).
3) *Fungsi* p akan menghasilkan *Promise* dengan nilai kembalian berupa union (*boolean* atau angka / *number*).
4) *Await* akan diletakkan pada konstruksi *try ... catch*. Jika baris ini digunakan maka akan menghasilkan *error*, sehingga akan ditangkap oleh *catch*.
5) *Await* yang ada pada bagian ini digunakan untuk mengerjakan **lebihDari** dan kemudian menempatkan hasilnya ke **hasilOK.** Perlu diketahui, saat proses dijalankan, bagian ini akan dijadikan *Promise* dan baru akan dikembalikan lagi setelah ada hasil.
6) Jika terjadi *error*, maka bagian ini akan dikerjakan.
7) Bagian ini digunakan untuk mengambil hasil dari proses *async/await* yang sudah didefinisikan di **p**.

B. TYPESCRIPT UNTUK *BACKEND*

Seperti halnya JavaScript dengan Node.js, TypeScript memungkinkan digunakan di kedua sisi, yaitu di sisi *client* (*frontend*) maupun *server* (*backend*). Hal ini disebut dengan *isomorphic,* artinya dapat digunakan pada sisi *client* maupun *server* dengan menggunakan satu bahasa pemrograman saja. Meskipun ada beberapa *framework* yang dapat digunakan untuk membuat aplikasi lengkap (sisi *client* maupun *server*) atau *fullstack*, pada modul ini akan dibahas tentang *backend* karena diperlukan untuk pembuatan *endpoint* yang dapat digunakan untuk memberikan akses data dengan operasi tertentu bagi aplikasi *mobile*.

1. Mekanisme *Frontend - Backend*

Dalam situasi seperti saat ini, membangun aplikasi *mobile* dengan data yang tersimpan dalam aplikasi/tempat penyimpanan *mobile phone* bukan merupakan hal yang ideal. Ketersediaan media penyimpan dalam suatu *mobile phone* meskipun makin lama makin besar, tetapi sama sekali tidak cukup untuk aplikasi yang relatif besar. Dalam kondisi demikian, sering kali aplikasi dibuat untuk dua sisi yaitu *frontend* (bagian yang menyediakan tampilan antarmuka pemakai/*user interface*) serta bagian yang menyediakan akses data di sisi *backend* (terletak pada suatu *server* tertentu di Internet). *Backend* akan menyediakan suatu *endpoint* yang dapat diakses oleh *frontend* dengan menggunakan protokol tertentu. Gambaran dari mekanisme *frontend - backend* dapat dilihat pada Gambar 1.5.



Gambar 1.5
Mekanisme *Client - Server/Frontend - Backend*

Bagan di atas memberi gambaran konseptual yang sampai saat ini masih relevan untuk membangun aplikasi. Pemakai akan berhadapan langsung dengan aplikasi yang menyediakan antarmuka dan kemudian setiap berinteraksi dengan aplikasi tersebut, antarmuka akan mengirim permintaan akses data ke *backend*. Supaya dapat terjadi komunikasi antara *frontend* dengan *backend*, maka diperlukan suatu protokol komunikasi. Contoh protokol untuk komunikasi tersebut antara lain adalah *RESTful API* dan *GraphQL*. Dengan demikian, untuk dapat menyediakan *endpoint* pada sisi *backend*; oleh karena itu perlu disediakan *server* dan di dalam *server* tersebut dibuat serta dijalankan program untuk mengekspos *endpoint* sesuai dengan protokol yang disepakati.

2. Menggunakan TypeScript untuk *Backend*

Pada pembahasan kali ini, protokol yang digunakan untuk komunikasi adalah *RESTful API*. Dengan demikian, diperlukan suatu pustaka atau *framework* yang dapat digunakan untuk mengekspos *endpoint* dengan menggunakan protokol *RESTful API*. Bagi sisi *frontend*, tidak diperlukan suatu *software* khusus, cukup hanya menggunakan *http client* saja (*browser*, *curl*, *wget*, Postman, dan lain-lain).

Beberapa *framework* yang dapat digunakan untuk membangun *RESTful API* untuk TypeScript antara lain adalah:

a. Loopback
b. Nest.js
c. Feathers
d. Express
e. Typetron
f. FoalTS
g. Ts.ED

Pada pembahasan ini, akan digunakan **Express** untuk mengekspos suatu *endpoint* data *users*. Jika *endpoint* http://server:port/users diakses, maka seluruh data *users* dalam format JSON akan dikirim sebagai *response* ke *client*.

a. Tentang Express

Express merupakan *framework* yang dapat digunakan untuk membuat aplikasi *Web* secara penuh ataupun hanya untuk membangun *RESTful API endpoint*. *Express* menggunakan pola MVC (*Model-View-Controller*):

- *User* akan mengirimkan *request* melalui *http client* dengan *method* tertentu.
- *Express* akan menerima *request* tersebut kemudian mencocokkan dengan *route*
- Jika ada *route* yang cocok, maka akan diteruskan ke *controller* yang menangani *request* tersebut sesuai dengan *route*.
- *Controller* akan melakukan proses bisnis dan logika tertentu dan kemudian akan mengirimkan hasilnya dalam bentuk *response* JSON atau serialisasi lainnya (jika *RESTful API*) atau ke *view* untuk dibuat file HTML *on-the-fly* dan kemudian dikirimkan ke *client* yang me-*request*. Jika *controller* melakukan proses akses data (baik dari DBMS maupun file JSON atau *resources* lainnya), maka dikatakan *controller* tersebut mengakses *model*.

Dengan demikian, untuk mengekspos *RESTful API*, setidaknya harus didefinisikan:

- *Server* yang akan mengekspos *RESTful API* tersebut.
- Routes.
- Controllers.
- Model.

b. Persiapan

Sebagai langkah awal, ada beberapa paket yang harus di-install serta beberapa konfigurasi dan pengaturan direktori. Buat direktori kosong di tempat yang anda sukai, beri nama **restful-api** dan kemudian masuk ke direktori tersebut. Semua file akan diletakkan di direktori tersebut.

Catatan: materi diambil dari https://morioh.com/p/2063e05353d4 dengan penyesuaian serta penambahan guna mengekspos *file* JSON untuk *endpoint*.

```
$ mkdir restful-api                                          // <1>
$ cd restful-api                                             // <2>
$ npm init -y                                                // <3>
$ npm install -g nodemon concurrently                        // <4>
$ npm install @types/node @types/express express  --save-dev // <5>
$ mkdir src dist                                             // <6>
$ tsc --init                                                 // <7>
```

Listing 1.22
Penggunaan *Resful-API*

1) Membuat direktori kosong untuk proyek yang akan dibangun.
2) Masuk ke direktori yang dibuat sebelumnya.
3) Inisialisasi direktori tersebut untuk proyek Node.js. Argumen -y digunakan supaya tidak perlu menjawab satu persatu.
4) Menginstall paket yang diperlukan di level global yaitu *nodemon* dan *concurrently*, keduanya dipakai pada saat menjalankan aplikasi.
5) Instalasi paket yang diperlukan serta menyimpan nama serta versi paket tersebut pada *package.json*.
6) Membuat direktori *src* (digunakan untuk menyimpan semua kode sumber TypeScript) dan direktori *dist* (digunakan untuk menyimpan hasil kompilasi kode sumber TypeScript ke JavaScript) supaya dapat dijalankan oleh Node.js.
7) Membuat file *tsconfig.json*.

Setelah perintah-perintah tersebut, struktur direktori dan file akan terlihat seperti yang ada pada Gambar 1.6.

Gambar 1.6
Struktur Direktori Dan File Di Awal Pembuatan Proyek

Ubah bagian "*scripts*" pada file *package.json* menjadi seperti berikut ini:

```
{
 "name": "restful-api",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "start:dev": "nodemon dist/index.js",
   "build:dev": "tsc --watch --preserveWatchOutput",
   "dev": "concurrently \"npm:build:dev\" \"npm:start:dev\""
 },
"keywords": [],
 "author": "",
```

```
 “license”: “ISC”,
 “devDependencies”: {
   “@types/express”: “^4.17.10”,
   “express”: “^4.17.1”
 }
}
```

Listing 1.23
Penggunaan *Paket JSON*

Bagian “*scripts”* digunakan untuk memberitahu pada *npm* tentang cara menjalankan aplikasi tersebut. Pada perintah *build*, akan dikompilasi semua kode sumber yang berada pada direktori tertentu (*src*) dan kemudian hasil kompilasi akan diletakkan pada direktori tertentu (*dist*, definisi keduanya terdapat pada file konfigurasi *tsconfig.json*). Hasil kompilasi akan mempunyai struktur direktori dan *file* yang sama dengan letak kode sumber TypeScript. Setelah itu, akan dijalankan *file* JavaScript yang berada pada direktori *dist*.

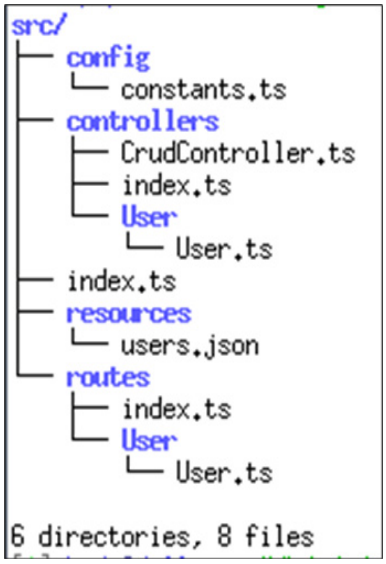
Konfigurasi TypeScript diletakkan pada file *tsconfig.json*. Gantilah isinya dengan isi berikut, perhatikan definisi *outDir* dan *rootDir*:

```
{
 “compilerOptions”: {
   “target”: “es5”,
   “module”: “commonjs”,
   “outDir”: “./dist”,
   “rootDir”: “./src”,
   “strict”: true,
   “esModuleInterop”: true,
   “skipLibCheck”: true,
   “forceConsistentCasingInFileNames”: true,
   “resolveJsonModule”: true
 }
}
```

Listing 1.24
Konfigurasi pada *file* tsconfig.json

c. Kode Sumber

Setelah itu, *file-file* kode sumber diletakkan pada direktori *src* (Silahkan Anda buat struktur file tersebut seperti Gambar 1.7). Struktur dari direktori dan *file-file* yang diperlukan dapat dilihat pada Gambar 1.7.



Gambar 1.7
Struktur Direktori Dan file di *src/*

File-file yang diperlukan untuk kode sumber akan diuraikan berikut ini.

```
export const PORT = process.env.PORT || 4000;
```

Listing 1.25
File *config/constants.ts*

File config/constants.ts digunakan untuk menampung konfigurasi yang diperlukan, dalam hal ini adalah konfigurasi untuk *port* yaitu 4000.

```
[
 {
  “balance”: “$3,946.45”,
  “picture”: “http://placehold.it/32x32”,
  “age”: 23,
  “name”: “Bird Ramsey”,
  “gender”: “male”,
  “company”: “NIMON”,
  “email”: “birdramsey@nimon.com”
 },
 {
  “balance”: “$2,499.49”,
  “picture”: “http://placehold.it/32x32”,
  “age”: 31,
  “name”: “Lillian Burgess”,
  “gender”: “female”,
  “company”: “LUXURIA”,
  “email”: “lillianburgess@luxuria.com”
 },
```

```
 {
  “balance”: “$2,820.18”,
  “picture”: “http://placehold.it/32x32”,
  “age”: 34,
  “name”: “Kristie Cole”,
  “gender”: “female”,
  “company”: “QUADEEBO”,
  “email”: “kristiecole@quadeebo.com”
 },
 {
  “balance”: “$3,277.32”,
  “picture”: “http://placehold.it/32x32”,
  “age”: 30,
  “name”: “Leonor Cross”,
  “gender”: “female”,
  “company”: “GRONK”,
  “email”: “leonorcross@gronk.com”
 },
 {
  “balance”: “$1,972.47”,
  “picture”: “http://placehold.it/32x32”,
  “age”: 28,
  “name”: “Marsh Mccall”,
  “gender”: “male”,
  “company”: “ULTRIMAX”,
  “email”: “marshmccall@ultrimax.com”
 }
]
```

Listing 1.26
resources/users.json

File resources/users.json akan dibaca saat *endpoint* diakses dan kemudian akan dikirimkan ke pemanggil (*http client*) dalam format JSON.

File JSON tersebut diambil dari : https://gist.github.com/rubenCodeforges/ef1f0ce6a055bbb985c0848d8b0c04d5#file-users-json.

```
import { Request, Response } from ‘express’;

export abstract class CrudController {
   public abstract create(req: Request, res: Response): void;
   public abstract read(req: Request, res: Response): void;
   public abstract update(req: Request, res: Response): void;
   public abstract delete(req: Request, res: Response): void;
}
```

Listing 1.27
controllers/CrudController.ts

File controllers/CrudController.ts digunakan untuk mendeklarasikan *abstract class (class* yang dimaksudkan untuk diimplementasikan, bukan untuk dibuat *instance*-nya). Setiap *endpoint* akan didefinisikan CRUD-nya (*create, read, update, delete*). Implementasinya akan dikerjakan oleh *class* turunannya.

```
import { Request, Response } from 'express';
import { CrudController } from '../CrudController';
import usersjson from '../../resources/users.json';

export class UserController extends CrudController {
Public create(req: Request<import("express-serve-static-core").ParamsDictionary>,
res: Response): void {
throw new Error("Belum diimplementasikan");
    }
public read(req: Request<import("express-serve-static-core").ParamsDictionary>,
res: Response): void {
                    res.json(usersjson);
    }
    public update(req: Request<import("express-serve-static-core").ParamsDictionary>,
res: Response): void {
        throw new Error("Belum diimplementasikan");
    }
public delete(req: Request<import("express-serve-static-core").ParamsDictionary>,
res: Response): void {
        throw new Error("Belum diimplementasikan");
    }
}
```

Listing 1.28
controllers/User/User.ts

File controllers/User/User.ts adalah *file* yang akan menangani jika ada *endpoint* yang dipanggil. *File* ini juga mengimplementasikan *CrudController*. Pada implementasi *read* akan dibaca *file* JSON dari *resources/users.json* dan kemudian akan dikirimkan ke pengakses *endpoint*.

```
import { UserController } from './User/User';

const userController = new UserController();

export {
  userController
};
```

Listing 1.29
controllers/index.ts

File controllers/index.ts digunakan untuk mengorganisasikan *export* supaya lebih mudah di-*import* dari program utama..

```
import express, { Request, Response } from 'express';
import { userController } from '../../controllers';

export const router = express.Router({
  strict: true
});

router.post('/', (req: Request, res: Response) => {
  userController.create(req, res);
});

router.get('/', (req: Request, res: Response) => {
  userController.read(req, res);
});

router.patch('/', (req: Request, res: Response) => {
  userController.update(req, res);
});

router.delete('/', (req: Request, res: Response) => {
  userController.delete(req, res);
});
```

Listing 1.30
routes/User/User.ts

File routes/User/User.ts digunakan untuk mendefinisikan pola *request HTTP method* serta permintaan *resources*. Sebagai contoh, jika akses *endpoint* **users/** (didefinisikan pada *src/index.ts*) pada *server* serta *port* yang ditentukan menggunakan *http method* GET, maka akan dikerjakan *userController.read*.

```
import { router as userRouter } from './User/User';

export {
  userRouter
};
```

Listing 1.31
routes/index.ts

File routes/index.ts digunakan untuk mengorganisasikan *export* supaya lebih mudah di-*import* dari program utama.

```
import express from 'express';
import { PORT } from './config/constants';
import { userRouter } from './routes';

const app = express();
app.use(express.json());

app.get('/', (req, res) => {
 res.send('Selamat datang di RESTful API gateway');
});
app.use('/users', userRouter);

app.listen(PORT, () => {
  console.log(`Endpoint sudah siap dan dapat diakses di port ${PORT}`);
});
```

Listing 1.32
File Utama: *index.ts*

File index.ts yang terletak di direktori *src* merupakan file utama. *File* ini akan dikompilasi menjadi *dist/index.js* dan akan dijalankan dengan menggunakan *nodemon* seperti yang terdapat pada *scripts* di *package.json*.

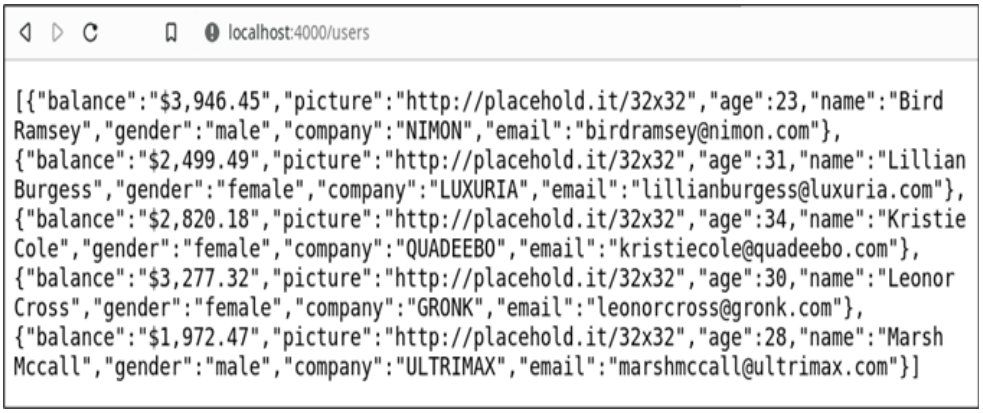
d. Menjalankan dan Mengekspos RESTful API Endpoint

Untuk menjalankan, gunakan perintah **npm run dev** di direktori proyek. Saat pertama kali dijalankan, akan *error* karena kompilasi awal. Tekan *Ctrl-C*, dan setelah itu kerjakan lagi **npm run dev** di direktori proyek.

Tampilan saat *run* adalah sebagai berikut:

```
...
...
[start:dev] [nodemon] 2.0.7
[start:dev] [nodemon] to restart at any time, enter `rs`
[start:dev] [nodemon] watching path(s): *.*
[start:dev] [nodemon] watching extensions: js,mjs,json
[start:dev] [nodemon] starting `node dist/index.js`
[build:dev]
[build:dev] 22.19.00 - Starting compilation in watch mode...
[build:dev]
[start:dev] Endpoint sudah siap dan dapat diakses di port 4000
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[build:dev]
[build:dev] 22.19.01 - Found 0 errors. Watching for file changes.
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] restarting due to changes...
[start:dev] [nodemon] starting `node dist/index.js`
[start:dev] Endpoint sudah siap dan dapat diakses di port 4000
```

Untuk mengakses *endpoint* tersebut, dapat digunakan *browser*. Hasil jika diakses menggunakan *browser* dapat dilihat pada Gambar 1.8.



Gambar 1.8
Hasil akses ke *endpoint*

Latihan

Untuk memperdalam pemahaman Anda mengenai materi di atas, kerjakanlah latihan berikut!

1) Buat program sederhana menggunakan TypeScript untuk membaca dan menampilkan isi suatu *file* teks. Pembacaan dilakukan secara asynchronous.
2) Buat contoh RESTful API seperti yang terdapat pada kegiatan belajar ini, tetapi menggunakan data yang berbeda

Petunjuk Jawaban Latihan

1) Baca manual untuk mengakses file secara asynchronous di https://nodejs.org/docs/latest-v14.x/api/fs.html#fs_fs_readfile_path_options_callback. Konversikan dengan menggunakan async-await di TypeScript. Siapkan satu file teks sebagai contoh.
2) Carilah file JSON di Internet (contoh: https://www.sitepoint.com/10-example-json-files/). Ganti *file* JSON yang terdapat pada *src/resources*, kemudian sesuaikan data di *controllers*.

Rangkuman

Asynchronous programming atau *non-blocking I/O* adalah teknik pemrograman untuk mengantisipasi kemungkinan akan terjadi atau tidak terjadi sesuatu hal pada saat yang akan datang serta untuk mengerjakan proses secara konkuren jika dirasakan latensi untuk bagian tertentu akan signifikan (antara lain karena akses I/O). TypeScript mengantisipasi kedua hal tersebut dengan menggunakan konstruksi *Promise* serta *async/await*.

Suatu aplikasi pada umumnya terdiri atas sisi *client/frontend* maupun *server/backend*. Selain *fullstack*, TypeScript dapat juga digunakan hanya pada salah satu sisi saja. Jika digunakan sebagai *backend,* maka TypeScript akan digunakan untuk mengekspos suatu data dengan menggunakan protokol tertentu, yaitu *RESTful API* dan/atau *GraphQL*.

Dengan menggunakan *RESTful API* sebagai *backend*, maka TypeScript hanya memerlukan *http server* saja. Salah satu *framework* yang dapat digunakan untuk penyediaan *http server* untuk mengekspos suatu *endpoint* tertentu adalah *Express*. *Express* dapat diakses menggunakan TypeScript dengan mengatur paket-paket tertentu (menggunakan *package.json*) serta mengatur konfigurasi TypeScript dengan menggunakan *tsconfig.json*.

Tes Formatif 3

Pilihlah satu jawaban yang paling tepat!

1) Pernyataan berikut terkait *asynchronous programming* di TypeScript adalah benar, *kecuali* ….
   A. dapat diimplementasikan dengan *callback*, *Promise*, dan *async/await*
   B. *promise* dan *async/await* adalah dua hal yang berbeda dan harus digunakan secara terpisah
   C. ada dua argumen dari *Promise*, yaitu *resolve* dan *reject*
   D. *await* harus diletakkan pada badan fungsi yang didefinisikan dengan *async*

2) Jika masih ada beberapa proses yang harus dikerjakan secara berurutan setelah *Promise* selesai dikerjakan, digunakan ….
   A. *nodemon*
   B. *generics*
   C. *union type*
   D. *promise chaining*

3) Berikut adalah *framework* yang dapat digunakan untuk membuat *backend - endpoint* menggunakan RESTful API, *kecuali* ….
   A. vue.js
   B. feathers
   C. express
   D. loopback

4) Pada pola MVC, bagian yang digunakan untuk data adalah ….
   A. view
   B. *model*
   C. *controller*
   D. *model* sekaligus *controller*

5) Bagian dari Express yang digunakan untuk mendefinisikan pola URL yang diakses oleh *user* adalah ….
   A. *model*
   B. *controller*
   C. *route*
   D. *view*

6) Supaya kompilasi dapat berjalan dengan baik, maka pada *tsconfig.json* harus didefinsikan ….
   A. outDir
   B. rootDir
   C. outDir dan rootDir
   D. ourFile

7) Untuk mengakses *RESTful API endpoint*, dapat digunakan ….
   A. semua yang dapat berfungsi sebagai *http-client*
   B. postman
   C. curl
   D. wget

8) Paket yang diperlukan untuk membangun *RESTful API endpoint* menggunakan TypeScript adalah ….
   A. nodemon
   B. express
   C. concurrently
   D. @types/express dan express

9) Supaya dapat membaca file JSON secara langsung dalam *import*, maka TypeScript memerlukan konfigurasi di *tsconfig.json* ….

A. tidak perlu konfigurasi apapun, secara *default* sudah dapat langsung import *json*
B. “jsonImport”: true
C. “jsonModuleResolver”: true
D. “resolveJsonModule”: true

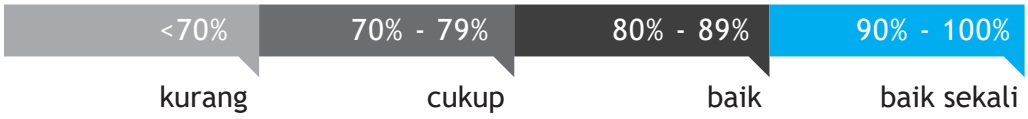
10) Bagian kode sumber *route* di *express* untuk server: ***app.get(‘/’, (req, res)*** => *{* digunakan untuk mengekspos ….

A. http://server:port/resources
B. http://server:port/api
C. http://server:port/
D. *root directory filesystem* (/)

Cocokkanlah jawaban Anda dengan Kunci Jawaban Tes Formatif 3 yang terdapat di bagian akhir modul ini. Hitunglah jawaban yang benar. Kemudian, gunakan rumus berikut untuk mengetahui tingkat penguasaan Anda terhadap materi Kegiatan Belajar 3.

$$\text{Tingkat Penguasaan} = \frac{\text{Jumlah Jawaban yang Benar}}{\text{Jumlah Soal}} \times 100$$

Arti tingkat penguasaan



Apabila mencapai tingkat penguasaan 80% atau lebih, Anda dapat meneruskan dengan modul selanjutnya. Bagus! Jika masih di bawah 80%, Anda harus mengulangi materi Kegiatan Belajar 3, terutama bagian yang belum dikuasai.

Kunci Jawaban Tes Formatif

Tes Formatif 1		*Tes Formatif 2*		*Tes Formatif 3*	
1)	C	1)	B	1)	B
2)	A	2)	C	2)	D
3)	B	3)	D	3)	A
4)	D	4)	D	4)	B
5)	C	5)	B	5)	C
6)	B	6)	A	6)	C
7)	A	7)	B	7)	A
8)	B	8)	D	8)	D
9)	B	9)	A	9)	D
10)	A	10)	C	10)	C

Glosarium

Android	:	Suatu sistem operasi yang digunakan pada perangkat *mobile phone* yang diproduksi oleh Google.
Anonymous Function	:	*Fungsi* yang tidak punya nama.
Arrow Function	:	Suatu *fungsi* yang didefinisikan menggunakan tanda panah (=>) dengan tujuan untuk *fungsi* yang kecil dan membuat kode sumber lebih ringkas
ART	:	Singkatan dari *Android Run Time*, yaitu mesin yang digunakan oleh Android untuk menjalankan berbagai aplikasi *mobile phone* Android.
Aras	:	Level atau tingkat
Asynchronous Programming	:	Teknik pemrograman untuk menangani proses yang konkuren karena latensi dari suatu proses yang dirasa cukup signifikan sehingga jangan sampai terjadi *blocking* terhadap berbagai proses selanjutnya.
Async/Await	:	Teknik pemrograman *asynchronous di TypeScript*
Class	:	Cetak biru dari hasil abstraksi berbagai obyek yang sama/sejenis.
Dalvik	:	Pendahulu ART, digunakan sebagai mesin untuk menjalankan berbagai aplikasi di Android.
Deno	:	Salah satu jenis kompilator TypeScript
EcmaScript	:	Spesifikasi bahasa pemrograman yang merupakan induk dari JavaScript.

Express	:	*Framework* yang digunakan untuk membangun aplikasi Web.
Function Expression	:	Pembuatan *fungsi* dengan cara ditempatkan sebagai suatu ekspresi sehingga dapat langsung dikerjakan.
Function Definition	:	Pembuatan *fungsi* dengan cara mendefinisikan *fungsi* menggunakan kata kunci *function* di bagian awal.
Generics	:	Konstruksi dari TypeScript untuk membuat fungsi yang se-*generik* dan se-umum mungkin dengan tidak memastikan tipe data baik di argumen maupun nilai kembalian. Tipe data baru diberikan saat *fungsi* digunakan.
HAL	:	*Hardware Abstraction Layer*, suatu bagian dari kernel yang digunakan untuk mengabstraksi *hardware* dan menyediakan API untuk memungkinkan aplikasi mengakses *hardware*.
HIDL	:	*HAL Interface Definition Language*, suatu bahasa untuk mendeskripsikan antarmuka antara HAL dan pemakai.
Hybrid Mobile Application	:	Aplikasi *mobile* yang dijalankan dengan menggunakan WebView.
Hoisting	:	Hoisting merupakan aturan dalam javascript yang seolah- olah memindahkan semua deklarasi ke bagian paling atas scope (bagian paling atas dari kode atau fungsi). Hoisting adalah istilah yang digunakan untuk menggunakan sebuah **variabel** yang sudah di deklarasikan sebelumnya tapi belum mempunyai nilai, hoisting lebih baik untuk tidak digunakan karena jika pengembang tidak begitu mengerti maka besar kemungkinan akan terdapat *bug* pada program sehingga ada baiknya untuk semua variabel dideklarasikan sejak awal.

Instance	:	Bentuk nyata dari *class*.
Interface	:	Suatu kontrak yang didefinisikan untuk diikuti oleh *fungsi, class*, maupun data.
Ionic Framework	:	Suatu *framework* untuk membuat aplikasi *hybrid*.
iPhone	:	Suatu perangkat *mobile phone* yang dibuat oleh Apple. iPhone menggunakan sistem operasi iOS.
Kernel	:	Bagian paling inti dari suatu sistem operasi yang bertugas untuk mengelola berbagai sumber daya yang ada pada perangkat keras serta menyediakan abstraksi ke pemakai.
Native Mobile Application	:	Aplikasi *mobile phone* yang dibuat dan dijalankan secara *native* - artinya sesuai dengan *run time* asli dari sistem operasi *mobile phone*.
Node.js	:	*Interpreter JavaScript*, mesin yang digunakan untuk menjalankan kode sumber yang dibuat untuk bahasa pemrograman JavaScript.
npm	:	*Software* yang digunakan untuk mengelola paket serta proyek Node.js
Object-Oriented Programming	:	Suatu paradigma pemrograman yang menyelesaikan masalah pemrogram dengan mendefinisikan berbagai obyek yang ada pada suatu sistem, interaksi antar obyek, serta aliran kerja dari sistem yang melibatkan berbagai obyek tersebut.
Promise	:	Suatu konstruksi bahasa pemrograman TypeScript yang digunakan untuk menangani *asynchronous programming*.
REPL		Tempat untuk mengeksekusi perintah-perintah seperti prompt/shell

RESTful API	:	Suatu API yang disediakan untuk diakses menggunakan protokol HTTP.
TypeScript	:	Bahasa pemrograman yang merupakan superset dari JavaScript.
User-Defined Function	:	Suatu *fungsi* yang didefinisikan sendiri oleh pemrogram.
Visual Studio Code	:	IDE yang mendukung pengembangan software menggunakan JavaScript serta TypeScript secara default.

Daftar Pustaka

Android Developers and Contributors, Android Reference, https://source.android.com/reference, diakses 2 Januari 2021.

Basarat Ali Syed, TypeScript Deep Dive, https://basarat.gitbook.io/typescript/, diakses 28 Desember 2020.

Express Developers and Contributors, Express Documentation, https://expressjs.com/, diakses 29 Desember 2020.

Marijn Haverbeke, Eloquent JavaScript, 3rd edition, 2018, https://eloquentjavascript.net/index.html, diakses pada 20 Desember 2020.

Node.js Developers and Contributors, Learn Node.js, https://nodejs.dev/learn, diakses 29 Desember 2020.

Node.js Developers and Contributors, Node.js Documentation, https://nodejs.org/en/docs/, diakses 28 Desember 2020.

TypeScript Team and Contributors. TypeScript Documentation, https://www.typescriptlang.org/docs, diakses 29 Desember 2020.