

BAB III

PROSES DAN HASIL BELAJAR DI DU/DI

A. Instalasi Android Studio IDE

1. Penjelasan Pekerjaan



Gambar 3.1. Logo Android Studio

Android Studio adalah *Integrated Development Environment* (IDE) untuk Sistem Operasi Android, yang dibangun di atas perangkat lunak JetBrains IntelliJ IDEA dan didesain khusus untuk pengembangan Android. IDE ini merupakan pengganti dari Eclipse Android *Development Tools* (ADT) yang sebelumnya merupakan IDE utama untuk pengembangan aplikasi Android.

Android Studio diumumkan pada 16 Mei 2013, di konferensi Google I/O. Itu dalam tahap pratinjau akses awal mulai dari versi 0.1 pada Mei 2013, kemudian memasuki tahap beta mulai dari versi 0.8 yang dirilis pada Juni 2014. Versi stabil pertama dirilis pada Desember 2014, mulai dari versi 1.0.

Pada 7 Mei 2019, Kotlin menggantikan Java sebagai bahasa pilihan Google untuk pengembangan aplikasi Android. Meskipun demikian, Java masih didukung seperti halnya C++.

Selain sebagai editor kode dan fitur developer IntelliJ yang andal, Android Studio menawarkan banyak fitur yang meningkatkan produktivitas dalam membuat aplikasi Android, seperti:

- a. Sistem versi berbasis Gradle yang fleksibel
- b. Emulator yang cepat dan kaya fitur
- c. Lingkungan yang menyatu untuk pengembangan bagi semua perangkat Android
- d. Instant Run untuk mendorong perubahan ke aplikasi yang berjalan tanpa membuat APK baru
- e. Template kode dan integrasi GitHub untuk membuat fitur aplikasi yang sama dan mengimpor kode contoh
- f. Alat pengujian dan kerangka kerja yang ekstensif
- g. Alat Lint untuk meningkatkan kinerja, kegunaan, kompatibilitas versi, dan masalah-masalah lain
- h. Dukungan C++ dan NDK

2. Alat dan Bahan

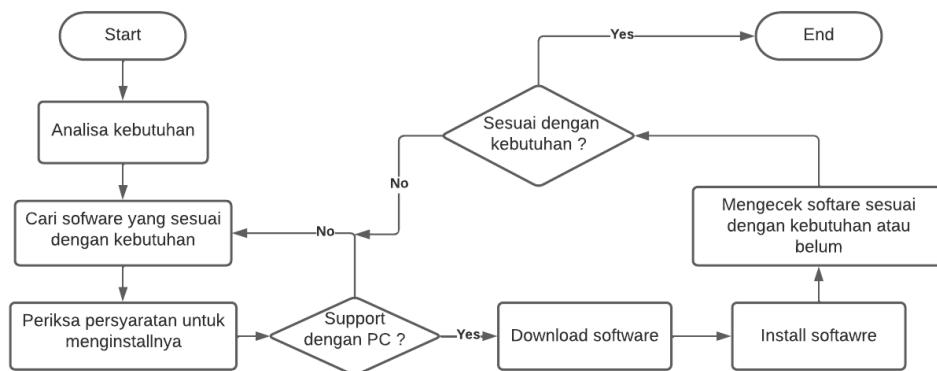
- a. PC
- b. Web Browser
- c. Koneksi Internet

3. Keselamatan Kerja

- a. Berdoa sebelum mengerjakan.
- b. Memastikan alat dan bahan sesuai dengan yang dibutuhkan.

- c. Menggunakan alat sesuai dengan fungsinya.
- d. Memperhatikan posisi duduk dan jarak pandang dengan komputer.
- e. Berhati hati dengan air dan listrik.
- f. Mematikan perangkat jika telah selesai digunakan.
- g. Memastikan alat dan bahan berada di posisi yang tepat.

4. Gambar Kerja

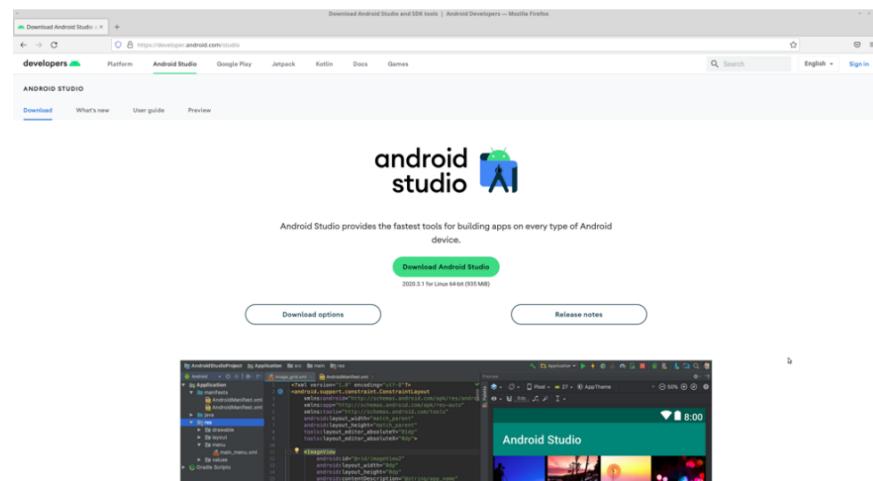


Gambar 3.2. Gambar Kerja Instalasi Android Studio

5. Analisa dan Langkah Kerja

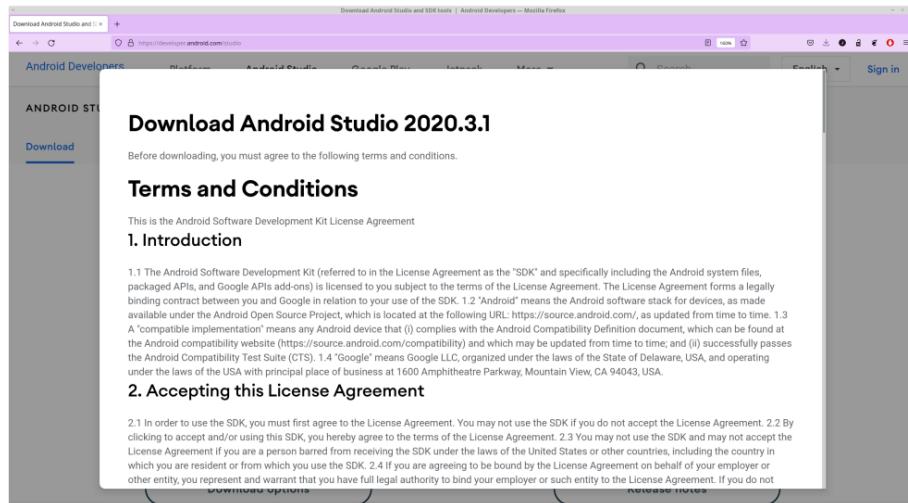
- a. *Download file installer Android Studio* sesuai dengan *operating system*

yang digunakan dari website resminya di
<https://developer.Android.com/studio>.



Gambar 3.3. Halaman Untuk *Download* Android Studio

Setelah website terbuka, tekan tombol *Download Android Studio* untuk mulai men-download, sebelum itu akan muncul dialog *Terms and Conditions*.



Gambar 3.4. Dialog *Terms and Conditions*

Terms and Conditions (T&C) adalah perjanjian hukum antara penyedia layanan dan orang yang menggunakan layanan tersebut, orang tersebut harus menyetujuinya untuk menggunakan layanan yang ditawarkan.

Setelah itu, di bagian bawah ada *checkbox* “*I have read and agree with the above terms and conditions*” kemudian tekan *Download* Android Studio untuk mulai men-download *installer*-nya.

- b. *File Installer* Android Studio berbeda-beda tiap *Operating System*. Untuk Windows, *file installer*-nya ber-extension .exe dan untuk memulai proses instalasinya tinggal *double click* *installer*-nya dan ikuti petunjuk instalasinya. Untuk Mac OS, *file installer*-nya akan ber-extension .dmg dan untuk cara instalasinya adalah *double click* *file* dmg-nya dan seret

Android Studionya ke folder Application. Sedangkan untuk Linux, filenya akan bereksistensi .tar.gz.



Gambar 3.5. *File installer* Android Studio untuk Linux

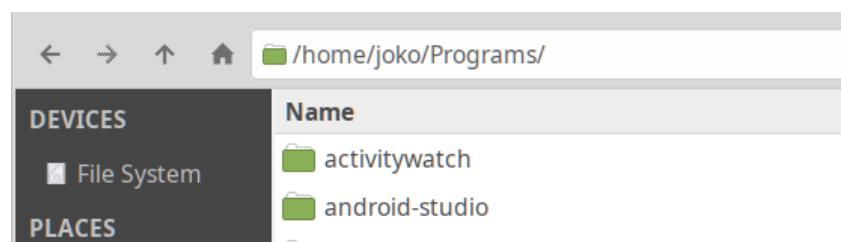
- c. Sebelum menjalankan Android Studio, jika komputer yang digunakan berbasis 64-bit, maka diharuskan untuk meng-*install* beberapa *library* 32-bit dengan menjalankan perintah berikut di terminal.

A screenshot of a terminal window. The title bar says 'jokosupriyanto@Jokos-MBP:~'. The window shows a command line session:

```
Last login: Sat Feb 12 20:41:58 on ttys003
~ via ● v14.15.3
•1% → sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:i386
```

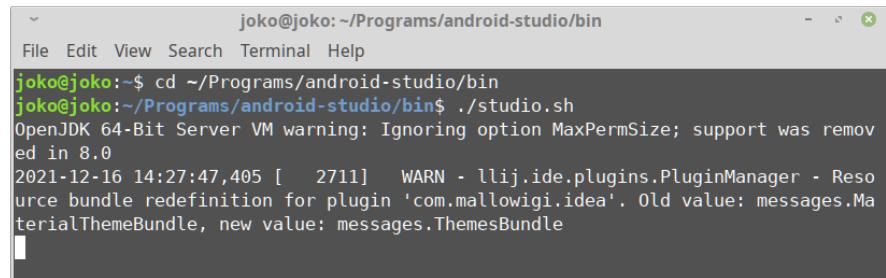
Gambar 3.6. Meng*install* Library Tambahan

- d. Karena di PC penulis menggunakan Linux Mint, maka *file installer*-nya berbentuk tar.gz. Setelah *file installer*-nya selesai ter-download, unpack *file* .tar.gz ke lokasi yang sesuai seperti /usr/local/ untuk spesifik *user* atau /opt/ untuk semua *user*. Tapi di sini penulis akan menaruhnya di /home/{username}/Programs/.



Gambar 3.7. Lokasi Android Studio

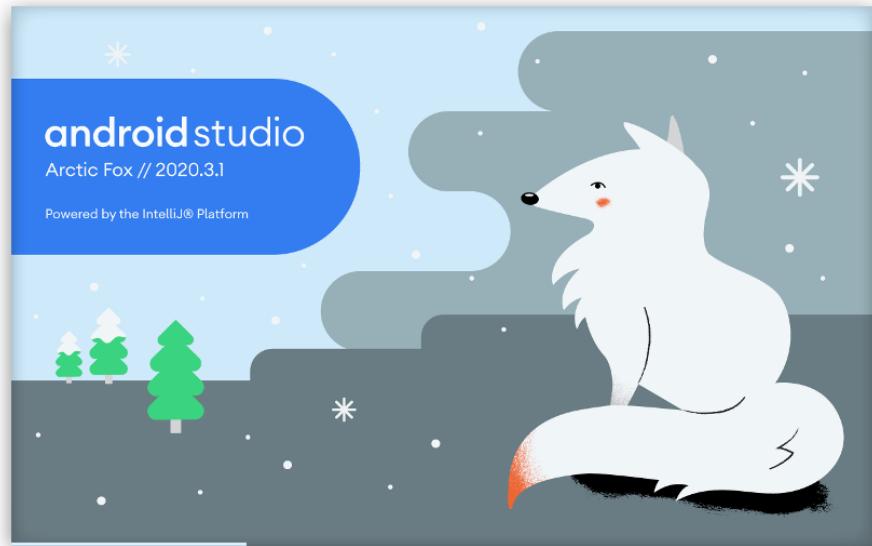
- e. Kemudian untuk menjalankan Android Studio, buka terminal dan pindah ke *directory* tempat instalasinya dan jalankan *file* studio.sh yang berada di folder bin.



```
joko@joko:~/Programs/android-studio/bin
File Edit View Search Terminal Help
joko@joko:~$ cd ~/Programs/android-studio/bin
joko@joko:~/Programs/android-studio/bin$ ./studio.sh
OpenJDK 64-Bit Server VM warning: Ignoring option MaxPermSize; support was removed in 8.0
2021-12-16 14:27:47,405 [ 2711]  WARN - llij.ide.plugins.PluginManager - Resource bundle redefinition for plugin 'com.mallowigi.idea'. Old value: messages.MaterialThemeBundle, new value: messages.ThemesBundle
```

Gambar 3.8. Menjalankan Android Studio Melalui Terminal

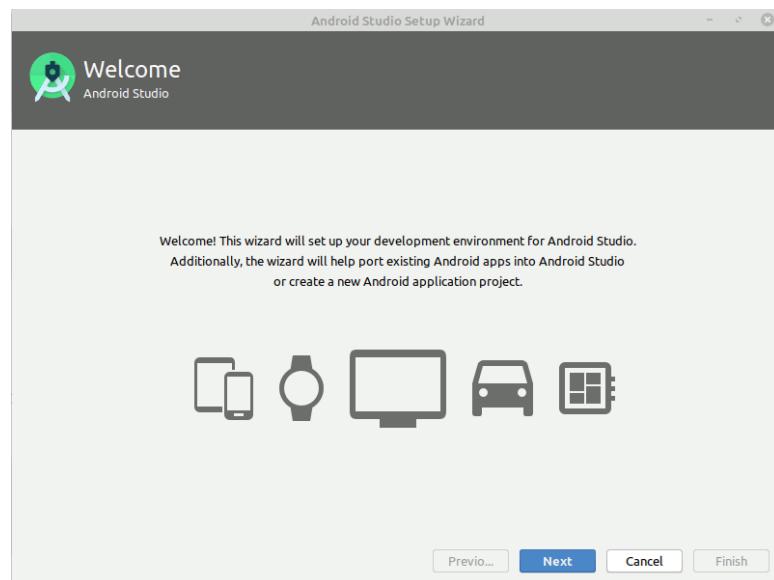
Setelah itu Android studio akan mulai berjalan, tunggu hingga *loading*-nya selesai.



Gambar 3.9. *Loading* Android Studio

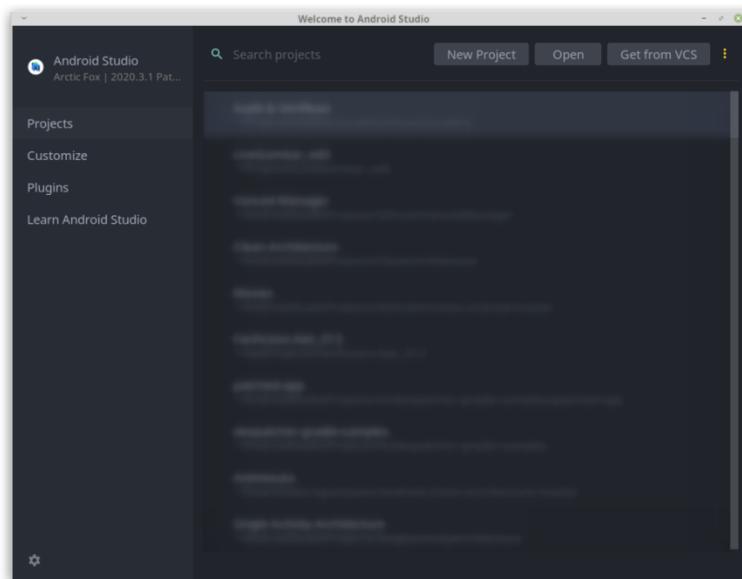
- f. Kemudian pilih apakah ingin *import setting* Android Studio yang sudah ada atau tidak, kemudian tekan OK.
- g. Android Studio akan menampilkan Setup Wizard yang akan memandu melalui proses instalasi yang tersisa, seperti memilih tema, *download*

Android SDK Components dan komponen lain yang dibutuhkan untuk proses *development*.



Gambar 3.10. Android Studio Setup Wizard

- h. Setelah seluruh proses selesai, Android Studio sudah siap digunakan untuk mengembangkan aplikasi Android.



Gambar 3.11. Jendela Utama Android Studio

6. Quality Control

Setelah instalasi Android Studio selesai, perlu dilakukan beberapa setup sebelum Android Studio benar-benar dapat digunakan dengan maksimal. Di antaranya adalah pengecekan fitur yang akan digunakan, atur Android Studio sesuai preferensi seperti ukuran *font* editor dan aplikasi, penggunaan *memory*, pengaturan tema, dan lain-lain. Atau bahkan *install* beberapa *plug-in* yang dibutuhkan untuk mempermudah pekerjaan di Android Studio.

7. Kesimpulan

Android Studio adalah Lingkungan Pengembangan Terpadu *Integrated Development Environment* (IDE) untuk pengembangan aplikasi Android, berdasarkan IntelliJ IDEA . Selain merupakan editor kode IntelliJ dan alat pengembang yang berdaya guna, Android Studio menawarkan lebih banyak fitur. Gunanya untuk meningkatkan produktivitas saat membuat aplikasi Android, dan mendapat dukungan langsung dari pihak Google.

Dalam Android Studio dapat menggunakan Java dan juga Kotlin. Dalam Android Studio, *programmer* dapat menulis, mengedit, menyimpan dan menguji proyek beserta dan *file* lainnya yang ada dalam proyek itu hanya dengan Android Studio. Android Studio juga memberi akses ke Android *Software Development Kit* (SDK). SDK adalah sebuah ekstensi dari kode Java yang memperbolehkannya untuk berjalan dengan mulus di perangkat Android.

B. Instalasi Postman

1. Penjelasan Pekerjaan



Gambar 3.12. Logo Postman

Postman adalah platform kolaborasi untuk pengembangan API (*Application Programming Interface*). Fitur-fitur Postman dapat menyederhanakan pembangunan API dan mempersingkat kolaborasi sehingga dapat membuat API yang lebih baik dan cepat. Selain pengembangan, Postman juga sering digunakan sebagai API Testing, QA, dan Manajemen produk perangkat lunak.

API adalah sebuah *software* yang dapat menghubungkan dua atau lebih aplikasi dalam platform apa pun. Cara kerja API adalah dengan menerima *request* dari sebuah aplikasi kemudian memberikan *response* sesuai dari *request* tersebut. Dengan kata lain, API bisa disebut sebagai server, namun server ini tidak hanya memberikan pelayanan kepada satu platform, melainkan banyak platform.

Postman dapat digunakan secara gratis dan instalasinya sangat mudah, cepat dan dapat berjalan pada sistem operasi Windows, Linux, ataupun MacOS. Di CV. Karya Hidup Sentosa aplikasi ini biasa digunakan oleh *programmer* untuk melakukan *trial* API yang sedang dikembangkan. Postman sendiri juga digunakan untuk melihat hasil atau *result* dari API

ketika dijalankan baik dengan metode GET ,POST , PUT, DELETE, dan lainnya dan menampilkannya dalam bentuk JSON, XML, maupun HTML.

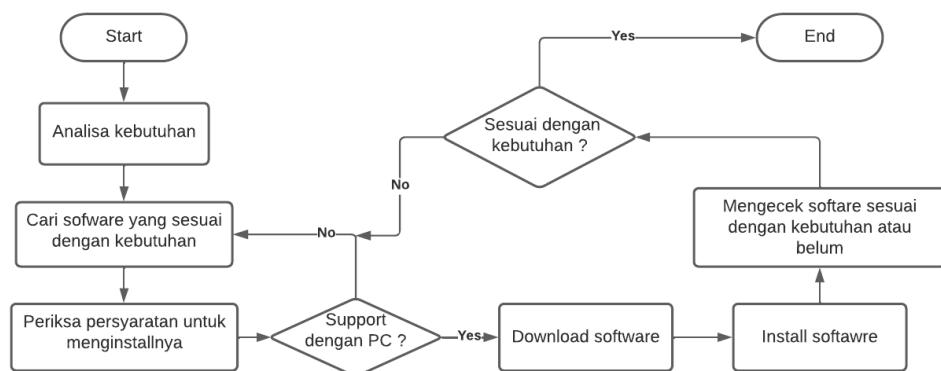
2. Alat dan Bahan

- a. PC
- b. Terminal
- c. Koneksi Internet (rekomendasi >1MB/s)

3. Keselamatan Kerja

- a. Berdoa sebelum mengerjakan.
- b. Memastikan alat dan bahan sesuai dengan yang dibutuhkan.
- c. Memastikan alat dan bahan berada di posisi yang tepat.
- d. Menggunakan alat sesuai dengan fungsinya.
- e. Memperhatikan posisi duduk dan jarak pandang dengan komputer.
- f. Berhati hati dengan air dan listrik.
- g. Mematikan perangkat jika telah selesai digunakan.

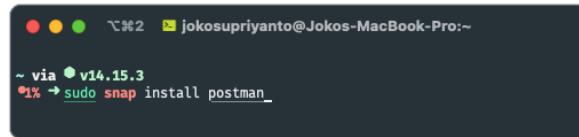
4. Gambar Kerja



Gambar 3.13. Gambar Kerja Instalasi Postman

5. Analisa dan Langkah Kerja

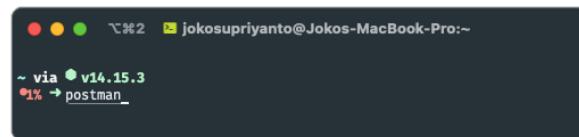
- Untuk meng-*install* Postman di Linux caranya cukup mudah, pertama buka terminal kemudian jalankan perintah berikut untuk meng-*install* Postman menggunakan *snap package manager*.



```
jokosupriyanto@Jokos-MacBook-Pro:~ % via v14.15.3
91% → sudo snap install postman
```

Gambar 3.14. *Install* Postman Dengan Terminal

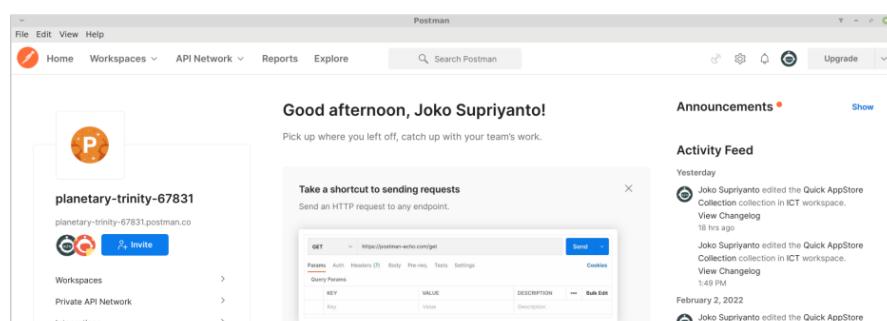
- Setelah proses instalasi selesai, buka *application menu* dan cek apakah Postman sudah berhasil ter-*install* atau belum, jika ada maka tinggal membukanya dengan mengeklik *icon* tersebut. Atau bisa juga dengan menjalankan perintah postman di terminal.



```
jokosupriyanto@Jokos-MacBook-Pro:~ % via v14.15.3
91% → postman
```

Gambar 3.15. Menjalankan Postman Melalui Terminal

- Setelah menjalankan perintah untuk menjalankan Postman di terminal, akan muncul *loading* dialog dari Postman. Tunggu hingga selesai dan jika muncul seperti berikut maka Postman sudah berhasil di-*install* dan siap digunakan.

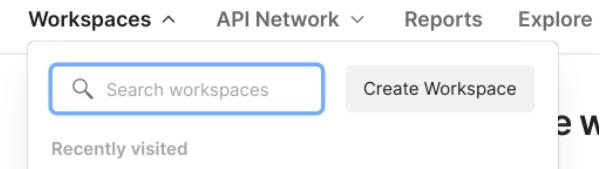


Gambar 3.16. Dashboard Postman

6. Quality Control

Setelah selesai meng-*install* Postman, perlu dilakukan pengecekan apakah fitur-fitur yang dibutuhkan pada aplikasi tersebut dapat berjalan dengan baik. Apabila ada yang belum lengkap atau *corrupt* dapat dilakukan instalasi ulang.

- Klik menu *Workspaces* di bagian atas Postman, kemudian tekan tombol *Create Workspace* untuk membuat *workspace* baru.



Gambar 3.17. Proses Pembuatan Workspace Baru

- Setelah itu akan muncul tampilan untuk membuat *workspace* baru. Isi nama *workspace* dan *summary* sesuai keinginan, dan juga pilih *visibility*-nya. Setelah semua terisi tekan tombol *Create Workspace* untuk membuat *workspace*.

Create workspace

 A screenshot of the 'Create workspace' dialog box. At the top, it says 'Create workspace'. Below that is a 'Name' field containing 'Testing Quality Control'. Underneath is a 'Summary' field with the placeholder 'Add a brief summary about this workspace.' At the bottom, there is a 'Visibility' section with four options: 'Personal' (radio button not selected), 'Private' (radio button not selected), 'Team' (radio button selected), and 'Public' (radio button not selected). The 'Team' option has a sub-note 'All team members can access'. At the very bottom are two buttons: 'Create Workspace' (highlighted in orange) and 'Cancel'.

Gambar 3.18. Membuat *Workspace* Baru di Postman

- c. Kemudian akan muncul tampilan berikut ini yang tandanya *workspace* sudah berhasil dibuat. Kemudian tekan tombol + di bagian atas untuk membuat *request* baru.

Gambar 3.19. *Workspace* Baru Yang Telah Dibuat

- d. Selanjutnya akan muncul tampilan seperti berikut.

Gambar 3.20. Tampilan Request Baru

- e. Isi *request URL* dengan URL API yang bisa kita gunakan untuk testing.

Semisal menggunakan json placeholder dari URL <https://jsonplaceholder.typicode.com/users>. Setelah URL-nya terisi, klik tombol Send untuk mulai mengirim *request*.

Gambar 3.21. Pengisian URL Request

- f. Jika mendapat *response* 200 OK berarti Postman dapat berjalan dengan baik untuk melakukan *network requesting*.



Gambar 3.22. Hasil Request Yang Berhasil

7. Kesimpulan

Postman merupakan sebuah sebuah *software* otomatisasi untuk API testing yang banyak digunakan oleh developer untuk menguji apakah API bekerja dengan baik pada *backend* dan *frontend* pada sebuah sistem. Postman dapat digunakan secara gratis dan instalasinya sangat mudah, cepat dan dapat berjalan pada sistem operasi Windows, Linux, ataupun MacOS.

C. Instalasi dan Konfigurasi DBeaver

1. Penjelasan Pekerjaan



Gambar 3.23. Logo DBeaver

DBeaver adalah sebuah SQL *client software* dan *database administration tool open source* dan multi platform yang dibuat dengan Java dan berdasarkan platform Eclipse. DBeaver menggunakan JDBC *application programming interface* (API) untuk berinteraksi dengan *relational database* melalui JDBC Driver.

DBeaver menyediakan editor yang mendukung *code completion* dan *syntax highlighting*. Aplikasi ini menyediakan sebuah arsitektur *plug-in* yang memungkinkan *user* untuk memodifikasi sebagian besar dari perilaku aplikasi untuk setiap spesifik *database* fungsionalitas atau fitur yang dibutuhkan *database*.

DBeaver Community Edition adalah versi pertama dari DBeaver yang di rilis di tahun 2010 dan menjadi *open source* di 2011. Versi Community Edition *includes extended support* beberapa *database* seperti MySQL and MariaDB, PostgreSQL, Greenplum, Oracle, DB2 (LUW), EXASOL, SQL Server, Sybase, Firebird, Teradata, Vertica, SAP HANA, Apache Phoenix, Netezza, Informix, Apache Derby, H2, SQLite, SnappyData, dan *database* lain yang mempunyai driver JDBC atau ODBC.

Aplikasi ini mempunyai beberapa fitur yang di antaranya :

- a. *SQL queries execution*
- b. *Data browser/editor with a huge number of features*
- c. *Syntax highlighting and SQL auto-completion*
- d. *Database structure (metadata) browse and edit*
- e. *SQL scripts management*
- f. *DDL generation*
- g. *ERD (Entity Relationship Diagrams) rendering*
- h. *SSH tunnelling*
- i. *SSL support (MySQL and PostgreSQL)*
- j. *Data export/migration*
- k. *Import, export and backup of data (MySQL and PostgreSQL)*
- l. *Mock data generation for database testing*

Kami biasanya menggunakan aplikasi ini untuk melakukan operasi ke *database* sehari-hari. Karena operasi *database* adalah pekerjaan yang sering kami lakukan, maka adanya aplikasi ini sangat dibutuhkan, sehingga kami harus meng-*install* dan mengkonfigurasinya dengan benar.

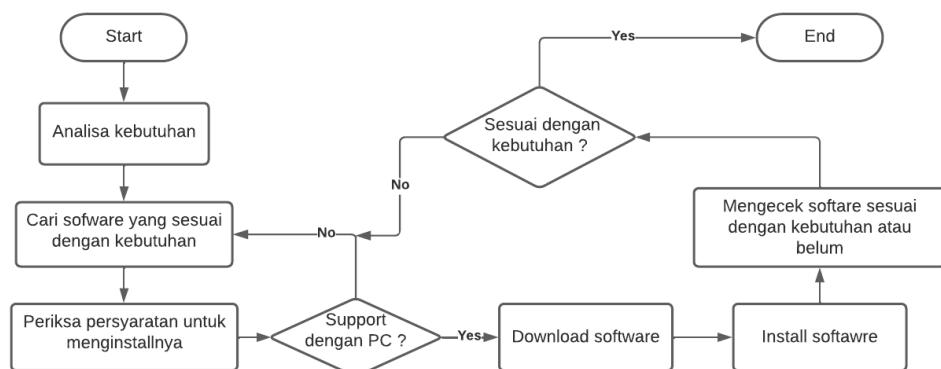
2. Alat dan Bahan

- a. Linux PC
- b. Koneksi Internet
- c. Firefox Web Browser

3. Keselamatan Kerja

- a. Berdoa sebelum mengerjakan.
- b. Memastikan alat dan bahan sesuai dengan yang dibutuhkan.
- c. Memastikan alat dan bahan berada di posisi yang tepat.
- d. Menggunakan alat sesuai dengan fungsinya.
- e. Memperhatikan posisi duduk dan jarak pandang dengan komputer.
- f. Berhati hati dengan air dan listrik.
- g. Mematikan perangkat jika telah selesai digunakan.

4. Gambar Kerja



Gambar 3.24. Gambar Langkah Kerja Instalasi DBeaver

5. Analisa dan Langkah Kerja

- a. Untuk meng-*install* DBeaver di Linux, kita bisa menggunakan beberapa cara yaitu dengan *file installer*, dengan archive aplikasinya langsung, atau dengan melalui *command line*. Di sini kita akan menggunakan *file installer*-nya.
- b. Untuk men-*download file installer*-nya, buka website resmi DBeaver di <https://dbeaver.io/download/> kemudian scroll sedikit ke bawah untuk melihat bagian untuk sistem operasi Linux. Lalu klik *link* Linux Debian Package 64 bit (*installer*) untuk men-*download* DBeaver-nya dan tunggu sampai proses *download*-nya selesai.

Linux

- [Linux Debian package 64 bit \(installer\)](#)
- [Linux RPM package 64 bit \(installer\)](#)
- [Linux 64 bit \(zip\)](#)
- [Linux x86 64 bit \(zip without Java included\)](#)
- [Linux ARM 64 bit \(zip without Java included\)](#)

- [Snap \(sudo snap installdbeaver-ce\)](#)
- [Flatpak \(flatpak install flathub io.dbeaver.DBeaverCommunity\)](#)

Gambar 3.25. *Link Download DBeaver Untuk Linux*

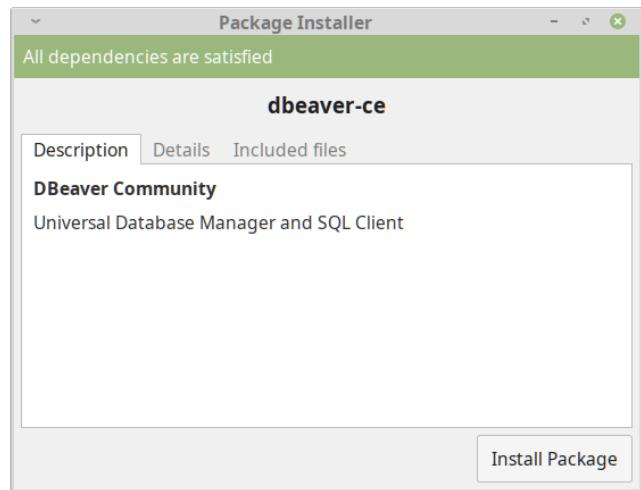
- c. Setelah *installer*-nya ter-*download*, *double click* *file*-nya untuk memulai proses instalasinya.



Gambar 3.26. *File Installer DBeaver*

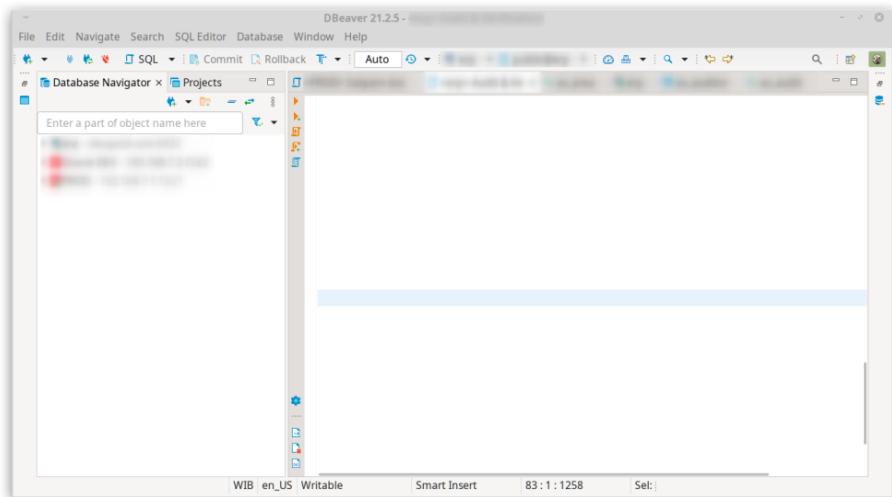
- d. Kemudian akan muncul jendela instalasi seperti berikut, untuk memulai proses instalasi klik tombol *Install Package* dan juga masukan *user*

password jika diminta. Setelah itu maka proses instalasi akan berjalan, tunggu hingga selesai.



Gambar 3.27. Jendela Instalasi DBeaver

- e. Setelah selesai, jalankan aplikasi DBeaver dari launcher. Jika sudah muncul tampilan seperti berikut ini maka DBeaver sudah berhasil ter-*install*.

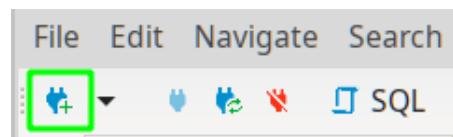


Gambar 3.28. Jendela Utama Aplikasi DBeaver Yang Telah Berhasil Ter-*install*

6. Quality Control

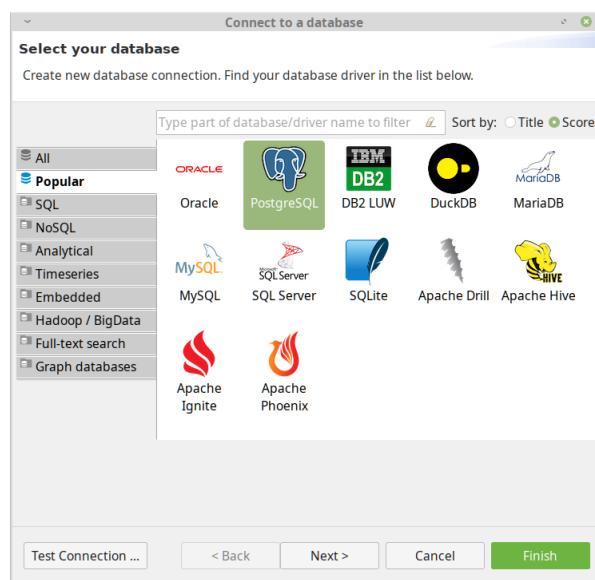
Setelah DBeaver ter-install, langkah selanjutnya adalah menguji apakah aplikasi ini dapat berjalan dengan baik atau tidak dengan cara membuat koneksi *database* baru.

- Untuk menambah koneksi *database* server baru, tekan icon *Create new connection* di bagian kiri atas DBeaver.



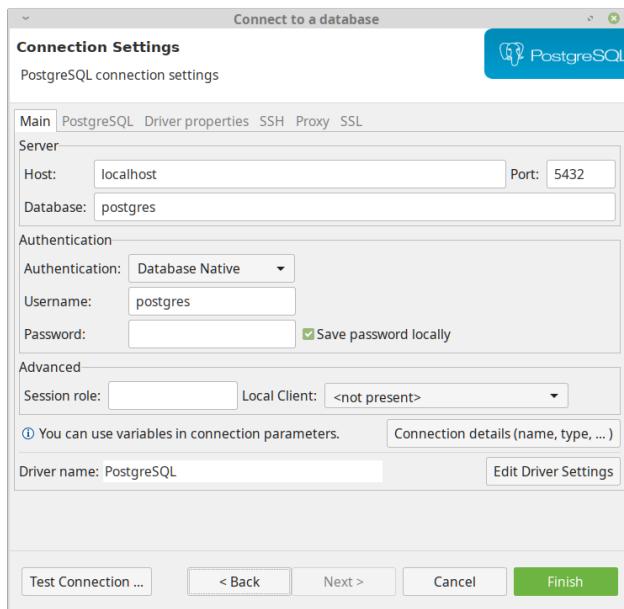
Gambar 3.29. Membuat Koneksi Baru

- Kemudian akan muncul jendela baru seperti berikut ini. DBeaver dapat menambah berbagai jenis koneksi *database* yang digunakan. Di sini pilih jenis *database* yang ingin ditambahkan, sebagai contoh kita akan memilih PostgreSQL. Setelah itu tekan tombol Next.



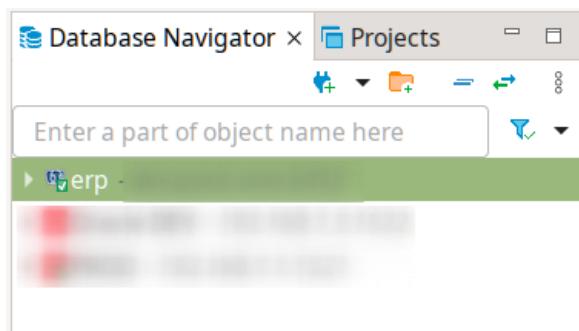
Gambar 3.30. Jendela Pemilihan Jenis Koneksi Baru

- c. Kemudian akan muncul jendela konfigurasi server *database*-nya seperti berikut ini. Di sini isi semua pengaturan *database*-nya sesuai dengan benar. Setelah selesai tekan tombol *Finish*.



Gambar 3.31. Jendela Pembuatan Koneksi Baru

- d. Setelah selesai, koneksi yang baru dibuat akan muncul di *Database Navigator* DBeaver.



Gambar 3.32. Koneksi Baru Yang Berhasil Dibuat

7. Kesimpulan

DBeaver Community Edition adalah sebuah SQL *client software* dan *database administration tool open source* dan multi platform yang

dibuat dengan Java dan berdasarkan platform Eclipse. DBeaver menyediakan editor yang mendukung *code completion* dan *syntax highlighting*. Dbever juga mendukung banyak *database* seperti MySQL and MariaDB, PostgreSQL, Greenplum, Oracle, DB2 (LUW), EXASOL, SQL Server, Sybase, Firebird, Teradata, Vertica, SAP HANA, Apache Phoenix, Netezza, Informix, Apache Derby, H2, SQLite, SnappyData, dan *database* lain yang mempunyai driver JDBC atau ODBC.

D. Pembuatan *Database* Aplikasi Audit & Verification

1. Penjelasan Pekerjaan



Gambar 3.33. Ilustrasi *Database*

Database (Pangkalan Data) adalah kumpulan informasi yang terorganisir, atau data yang biasanya disimpan secara elektronik di dalam sistem komputer. Sebuah *database* biasanya dikontrol dengan sebuah *Database Management System* (DBMS).

Data dalam tipe *database* yang paling umum saat ini biasanya dimodelkan dalam bentuk baris dan kolom dalam serangkaian tabel untuk membuat pemrosesan dan *query* data menjadi efisien. Data kemudian dapat dengan mudah diakses, dikelola, dimodifikasi, diperbarui, dikendalikan dan diatur. Sebagian besar *database* menggunakan bahasa *Structured Query Language* (SQL) untuk menulis dan mendapatkan data.



Gambar 3.34. Logo Bahasa SQL

SQL adalah bahasa pemrograman yang digunakan hampir semua *relational database* untuk *query*, memanipulasi, dan mendefinisikan data, dan untuk menyediakan akses kontrol. SQL pertama kali dikembangkan di IBM pada tahun 1960-an dengan Oracle sebagai kontributor utama, yang menyebabkan penerapan standar SQL ANSI, SQL telah memacu banyak ekstensi dari perusahaan seperti IBM, Oracle, dan Microsoft. Meskipun SQL masih banyak digunakan saat ini, bahasa pemrograman lain juga mulai muncul.

Hampir setiap aplikasi Android yang kami buat saat di CV. KHS pasti membutuhkan sebuah *database* atau mungkin beberapa *database* untuk menyimpan data yang diproses oleh aplikasi tersebut. *Database* yang kami gunakan di aplikasi Audit & Verification ini adalah PostgreSQL.

2. Alat dan Bahan

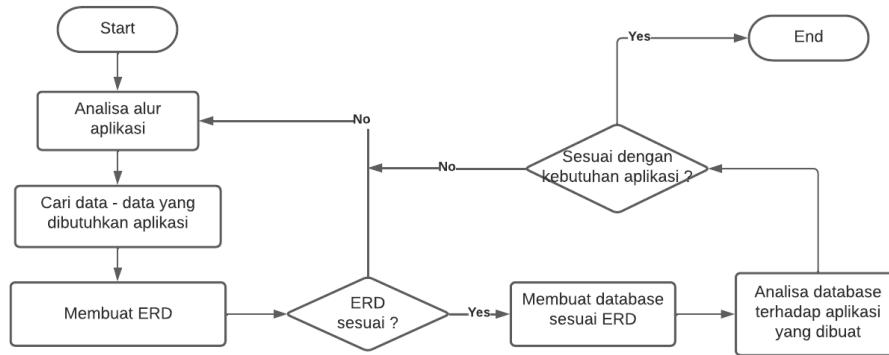
- a. Komputer
- b. DBeaver
- c. <https://dbdiagram.io/>
- d. Koneksi Internet

3. Keselamatan Kerja

- a. Berdoa sebelum mengerjakan.
- b. Memastikan alat dan bahan sesuai dengan yang dibutuhkan.
- c. Memastikan alat dan bahan berada di posisi yang tepat.
- d. Menggunakan alat sesuai dengan fungsinya.

- e. Memperhatikan posisi duduk dan jarak pandang dengan komputer.
- f. Berhati hati dengan air dan listrik.
- g. Mematikan perangkat jika telah selesai digunakan.

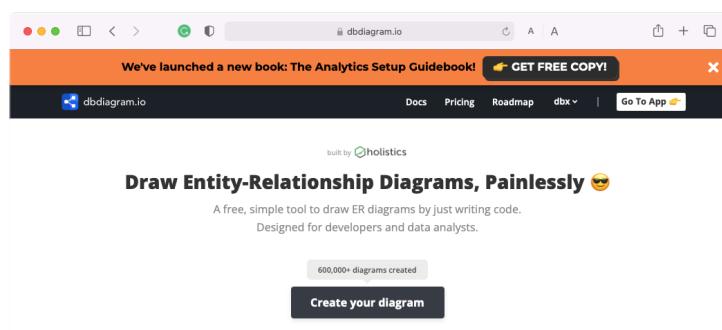
4. Gambar Kerja



Gambar 3.35. Gambar Kerja Pembuatan *Database Audit & Verification*

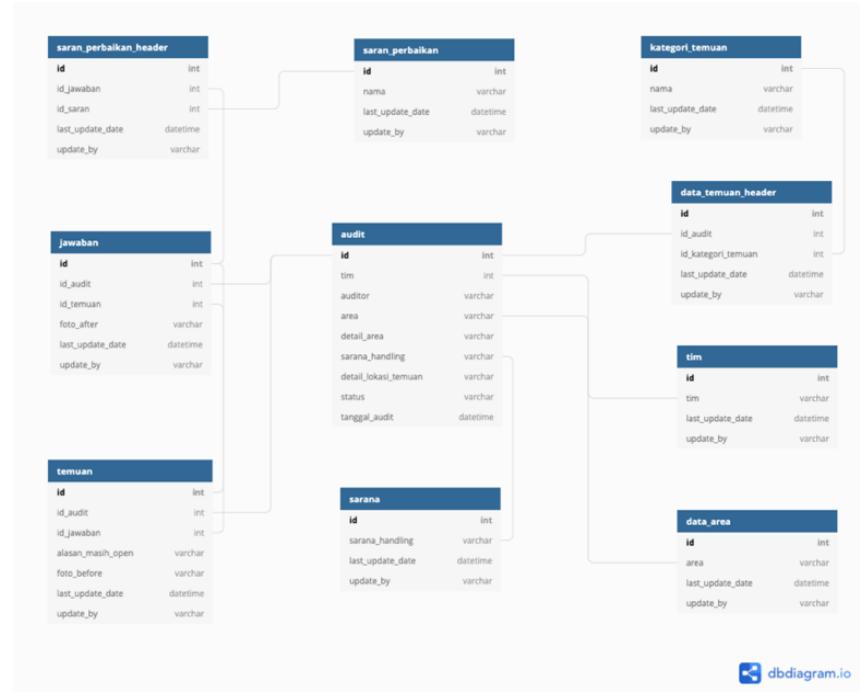
5. Analisa dan Langkah Kerja

- a. Langkah pertama adalah menganalisa alur aplikasi yang dibutuhkan user. analisa data-data apa yang dibutuhkan, yang diolah dan yang digunakan.
- b. Setelah itu, buat ERD sesuai hasil analisa yang kami lakukan. Untuk membuatnya bisa menggunakan website dbdiagram.io.



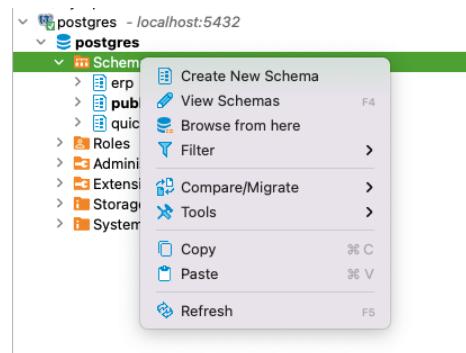
Gambar 3.36. Website dbdiagram.io

Berikut adalah hasil ERD yang dibuat.



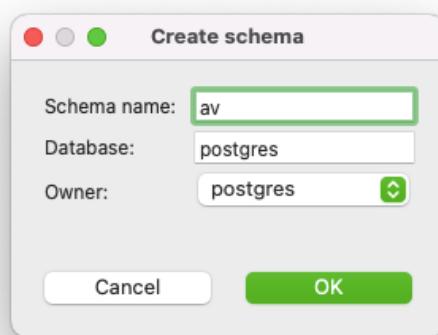
Gambar 3.37. ERD Database

- Setelah ERD sudah dibuat, kemudian buat *database*-nya di PostgreSQL dengan menggunakan Sql Tools DBeaver.
- Untuk membuat *database*/skema di DBeaver, buka DBeaver kemudian hubungkan dengan server. Setelah terhubung dengan server, buat skema baru dengan nama av. Caranya dengan klik kanan di *Schemas* pada salah satu *database* dan pilih *Create New Schema*.



Gambar 3.38. Membuat Skema Baru

Setelah itu akan muncul dialog *Create Schema*, isi *Schema name* dan tekan tombol *OK* untuk membuat *schema*-nya.



Gambar 3.39. Jendela Untuk Membuat Skema Baru

- e. Kemudian buat *script file* baru dan tulis *query* untuk membuat tabel tabelnya. Berikut *query* untuk membuat tabel-tabel yang dibutuhkan aplikasi Audit & Verification :

```

CREATE TABLE "audit" (
    "id" SERIAL PRIMARY KEY,
    "tim" int,
    "auditor" varchar,
    "area" varchar,
    "detail_area" varchar,
    "sarana_handling" varchar,
    "detail_lokasi_temuan" varchar,
    "status" varchar,
    "tanggal_audit" datetime
);
CREATE TABLE "data_temuan_header" (

```

```
    "id" SERIAL PRIMARY KEY,  
    "id_audit" int,  
    "id_kategori_temuan" int,  
    "last_update_date" datetime,  
    "update_by" varchar  
);
```

```
CREATE TABLE "data_area" (  
    "id" SERIAL PRIMARY KEY,  
    "area" varchar,  
    "last_update_date" datetime,  
    "update_by" varchar  
);
```

```
CREATE TABLE "tim" (  
    "id" SERIAL PRIMARY KEY,  
    "tim" varchar,  
    "last_update_date" datetime,  
    "update_by" varchar  
);
```

```
CREATE TABLE "kategori_temuan" (  
    "id" SERIAL PRIMARY KEY,  
    "nama" varchar,  
    "last_update_date" datetime,  
    "update_by" varchar
```

```
);
```

```
CREATE TABLE "saran_perbaikan_header" (
    "id" SERIAL PRIMARY KEY,
    "id_jawaban" int,
    "id_saran" int,
    "last_update_date" datetime,
    "update_by" varchar
);
```

```
CREATE TABLE "saran_perbaikan" (
    "id" SERIAL PRIMARY KEY,
    "nama" varchar,
    "last_update_date" datetime,
    "update_by" varchar
);
```

```
CREATE TABLE "sarana" (
    "id" SERIAL PRIMARY KEY,
    "sarana_handling" varchar,
    "last_update_date" datetime,
    "update_by" varchar
);
```

```
CREATE TABLE "jawaban" (
```

```

"id" SERIAL PRIMARY KEY,
"id_audit" int,
"id_temuan" int,
"foto_after" varchar,
"last_update_date" datetime,
"update_by" varchar
);

```

```

CREATE TABLE "temuan" (
"id" SERIAL PRIMARY KEY,
"id_audit" int,
"id_jawaban" int,
"alasan_masih_open" varchar,
"foto_before" varchar,
"last_update_date" datetime,
"update_by" varchar
);

```

- f. Setelah ditulis semua, klik tombol execute untuk menjalankan query-nya dan membuat tabel-tabelnya.

```

1 CREATE TABLE "audit" (
2   "id" SERIAL PRIMARY KEY,
3   "tim" int,
4   "auditor" varchar,
5   "area" varchar,
6   "detail_lokasi" "varchar",
7   "detail_lokasi_ling" varchar,
8   "detail_lokasi_temuan" varchar,
9   "status" varchar,
10  "tanggal_audit" datetime
11 );
12

```

Gambar 3.40. Menjalankan Query Di DBeaver

6. Quality Control

Setelah *query* untuk membuat semua tabelnya dijalankan, periksa tabel-tabelnya di DBeaver. Caranya dengan mengklik skema av yang sudah dibuat dan periksa apakah tabel-tabelnya sudah benar-benar terbuat atau belum.

		Name:	Object ID	Owner	Tablespace	Row Count	Estimate	Partitions	Partition by	Extra Options	Comment
Tables		audit	65,542	postgres	pg_default	0	[]	0			
Views		data_area	65,564	postgres	pg_default	0	[]	0			
Materialized Views		data_temu...	65,553	postgres	pg_default	0	[]	0			
Indexes		jawaban	65,630	postgres	pg_default	0	[]	0			
Functions		kategori_tem...	65,586	postgres	pg_default	0	[]	0			
Sequences		saran_perbai...	65,608	postgres	pg_default	0	[]	0			
Data types		saran_perbai...	65,597	postgres	pg_default	0	[]	0			
Aggregate functions		sarana	65,619	postgres	pg_default	0	[]	0			
Permissions		temuan	65,641	postgres	pg_default	0	[]	0			
Source		tim	65,575	postgres	pg_default	0	[]	0			

Gambar 3.41. Daftar Tabel Yang Berhasil Dibuat

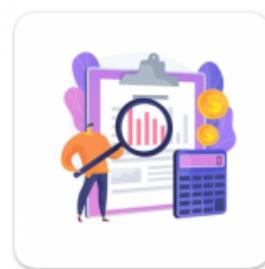
7. Kesimpulan

Database adalah kumpulan informasi yang terorganisir, atau data yang biasanya disimpan secara elektronik di dalam sistem komputer. Untuk mengelola *database* biasanya menggunakan bahasa pemrograman SQL.

SQL adalah bahasa pemrograman yang digunakan hampir semua *relational database* untuk *query*, memanipulasi, dan mendefinisikan data, dan untuk menyediakan akses kontrol.

E. Pembuatan Aplikasi Audit & Verification

1. Penjelasan Pekerjaan



Gambar 3.42. Logo Aplikasi Audit & Verification

Aplikasi Audit & Verification adalah aplikasi yang digunakan untuk melakukan audit di kawasan CV. Karya Hidup Sentosa Tusono. Aplikasi ini digunakan untuk menginput audit yang dilakukan oleh auditor jika ada audit yang ditemukan saat patroli. Jika ditemukan audit di suatu seksi, maka dengan menggunakan aplikasi ini, membuat audit baru sesuai keadaan di lokasi.

Kemudian *auditee* menggunakan aplikasi ini untuk melihat audit yang dibuat auditor di seksinya. Dan menggunakan data yang telah di inputkan oleh auditor untuk melakukan tindak lanjut/*follow up* terhadap audit tersebut dan kemudian menginputkan *follow up*-nya di aplikasi ini.

Setelah di-*follow up*, maka auditor akan memverifikasi *follow up* yang telah dilakukan oleh *auditee* dan memutuskan apakah audit tersebut telah selesai (*close*) atau perlu dilakukan tindakan lebih lanjut lagi (*open*). Dan kemudian jika di *open* lagi maka *auditee* harus melakukan *follow up* lagi sesuai dengan saran dari auditor. Dan proses ini berulang terus sampai auditor menutup (*close*) audit tersebut.

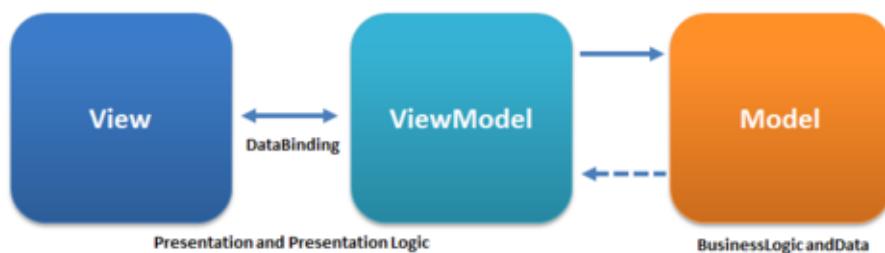
Di aplikasi ini juga ada menu *monitoring* untuk me-*monitoring* semua data audit yang ada di aplikasi ini.

Dalam *project* ini penulis menggunakan Android Studio sebagai IDE-nya, Kotlin sebagai bahasa pemrogramannya, dan MVVM sebagai *software architecture pattern*-nya.



Gambar 3.43. Logo Kotlin

Kotlin adalah sebuah bahasa pemrograman dengan pengetikan statis yang berjalan pada Mesin Virtual Java (JVM) ataupun menggunakan *compiler* LLVM. Pengembang utama Kotlin adalah tim *programmer* dari JetBrains yang bermarkas di Rusia yang pertama dirilis tahun 2011. Setelah Google mengumumkan bahwa Kotlin menjadi bahasa utama atau yang direkomendasikan ketika membuat aplikasi Android bersama dengan Java dan C++.



Gambar 3.44. Diagram Arsiteksur MVVM

MVVM (Model View ViewModel) adalah salah satu *Architectural Pattern* yang membagi tanggung jawab dalam pembuatan *Graphical User Interface* (View) kepada tiga komponen utama, yaitu Model, View dan ViewModel. View bertanggung jawab untuk semua hal yang berhubungan

dengan UI (*User Interface*). Model merupakan komponen yang bertanggung jawab untuk menyediakan data yang dibutuhkan. ViewModel adalah komponen inti dari *Architectural Pattern* ini, yang tugasnya menyimpan dan mengambil data dari Model untuk nantinya ditampilkan oleh View.

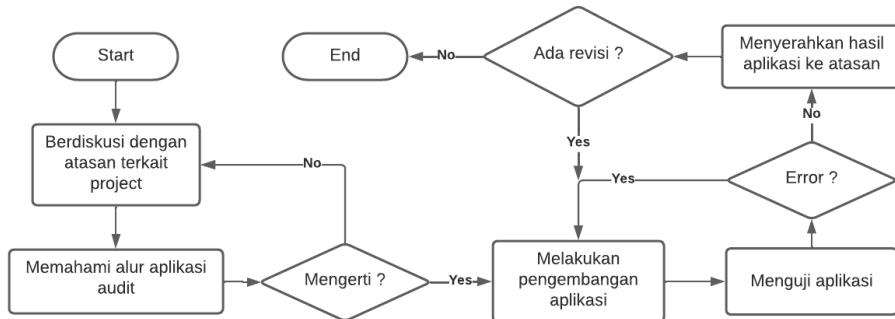
2. Alat dan Bahan

- a. PC
- b. Android Studio
- c. Visual Studio Code
- d. Postman
- e. DBeaver
- f. Web Browser
- g. Internet

3. Keselamatan Kerja

- a. Berdoa sebelum mengerjakan.
- b. Memastikan alat dan bahan sesuai dengan yang dibutuhkan.
- c. Memastikan alat dan bahan berada di posisi yang tepat.
- d. Menggunakan alat sesuai dengan fungsinya.
- e. Memperhatikan posisi duduk dan jarak pandang dengan komputer.
- f. Berhati hati dengan air dan listrik.
- g. Mematikan perangkat jika telah selesai digunakan.

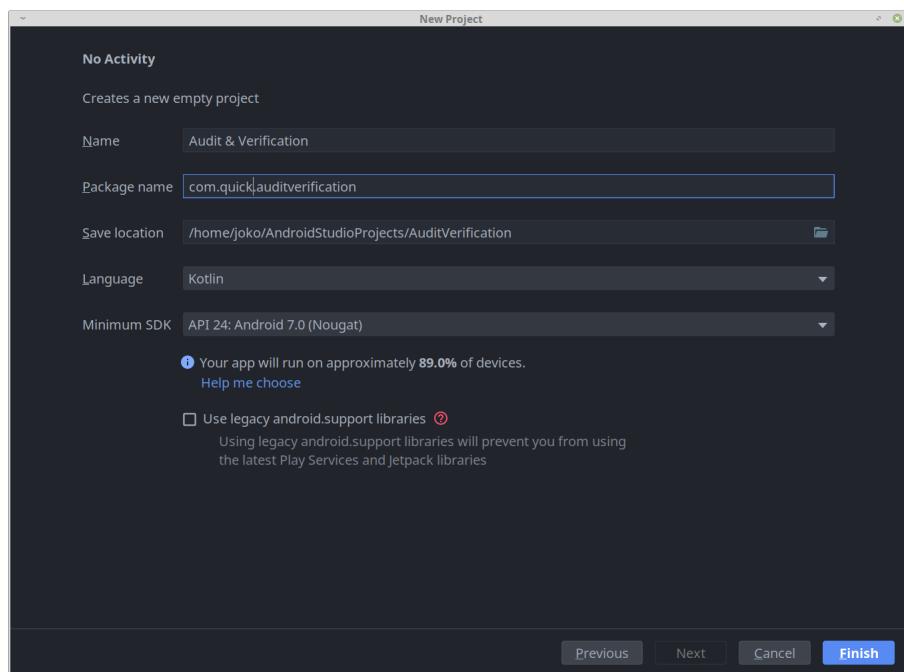
4. Gambar Kerja



Gambar 3.45. Gambar Kerja Pembuatan Aplikasi Audit & Verification

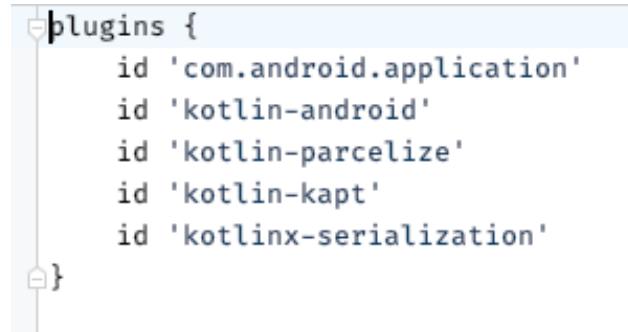
5. Analisa dan Langkah Kerja

- Langkah pertama adalah membuka Android Studio IDE dan membuat *project* baru dengan nama *Audit & Verification*, nama *package* `com.quick.auditverification`, bahasa pemrograman *Kotlin*, dan minimum versi *SDK 24* (*Android versi Nougat*).



Gambar 3.46. Jendela Pembuatan *Project* Baru

- b. Setelah *project* selesai di-build, kemudian buka file build.gradle dan tambahkan beberapa *dependencies* yang dibutuhkan.



Gambar 3.47. *Plug-in* Di Gradle

```

implementation 'org.postgresql:postgresql:42.2.5.jre7'
implementation fileTree(dir: 'libs', include: ['*.jar'])
implementation 'androidx.appcompat:appcompat:1.4.0'
implementation 'com.google.android.material:material:1.4.0'
implementation 'androidx.annotation:annotation:1.3.0'
implementation 'androidx.constraintlayout:constraintlayout:2.1.2'
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.0'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0'
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.1.0'
implementation 'androidx.recyclerview:recyclerview:1.2.1'
implementation 'androidx.multidex:multidex:2.0.1'

```

Gambar 3.48. Daftar *Dependencies*

```

// Material Spinner
implementation 'com.jaredrummler:material-spinner:1.3.1'
implementation 'com.github.arcadefire:nice-spinner:1.4.4'

// Image Picker
implementation 'com.github.dhaval2404:imagepicker:1.8'
implementation 'com.github.florent37:inline-activity-result-kotlin:1.0.4'

implementation 'com.github.bumptech.glide:glide:4.12.0'
// implementation 'com.github.recruit-lifestyle:WaveSwipeRefreshLayout:1.6'
implementation 'com.mohamedabulgasem:loadingoverlay:1.0.0'
implementation 'com.github.chivorns:smartmaterialsspinner:1.5.0'
implementation 'com.hudomju:swipe-to-dismiss-undo:1.0'

```

Gambar 3.49. Daftar *Dependencies*

```

// Ktor
def ktor_version = "1.6.6"
implementation "io.ktor:ktor-client-android:$ktor_version"
implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.0'
implementation "io.ktor:ktor-client-serialization:$ktor_version"
implementation "io.ktor:ktor-client-logging-jvm:$ktor_version"
implementation "io.ktor:ktor-client-gson:$ktor_version"
implementation "io.ktor:ktor-client-core:$ktor_version"

// Koin
// implementation "io.insert-koin:koin-android-viewmodel:2.2.3"
def koin_version = "3.1.0"
implementation "io.insert-koin:koin-android:$koin_version"
testImplementation "io.insert-koin:koin-test:$koin_version"
testImplementation "io.insert-koin:koin-test-junit4:$koin_version"

testImplementation 'org.powermock:powermock-module-junit4-rule:2.0.0-beta.5'
testImplementation 'org.powermock:powermock-core:2.0.0-beta.5'
testImplementation 'org.powermock:powermock-module-junit4:2.0.0-beta.5'
testImplementation 'org.powermock:powermock-api-mockito2:2.0.0-beta.5'

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.1'

```

Gambar 3.50. Daftar *Dependencies*

- c. Setelah semua *dependencies* terdownload, kemudian buat kelas App yang mewarisi kelas Application.

```

package com.quick.auditverifikasi

import ...

@KtorExperimentalAPI
class App : Application() {

    override fun onCreate() { ... }

    companion object { ... }
}

```

Gambar 3.51. Class App Yang Digunakan Untuk Aplikasi Audit & Verification

Di *function onCreate* yang di-*override*, tambah baris berikut.

```
appContext = this

MainScope().launch {
    withContext(Dispatchers.IO) {
        prefs = KsPrefs(appContext as App) {
            encryptionType = EncryptionType.PlainText()
            autoSave = AutoSavePolicy.AUTOMATIC
            commitStrategy = CommitStrategy.COMMIT
        }
    }
}
```

Gambar 3.52. *Code Function onCreate() App*

Di kelas App ini juga ada *companion object* yang berisi appContext dan prefs yang dapat diakses di seluruh aplikasi.

```
companion object {
    lateinit var appContext: Context
    lateinit var prefs: KsPrefs

    fun isPrefsInitialized() = ::prefs.isInitialized
}
```

Gambar 3.53. Companion Object di Class App

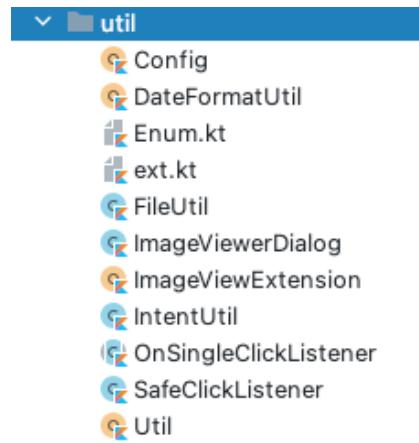
Kemudian agar kelas App ini dijalankan Ketika aplikasi ini dibuka, tambahkan kelas App yang sudah dibuat tadi di manifest pada bagian android:name.

```
<application
    android:name=".App"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
```

Gambar 3.54. Penggunaan Class App Di Manifest Aplikasi Audit &

Verification

- d. Kemudian buat *package* baru dengan nama util. Dalam *package* ini akan berisi kelas-kelas dan fungsi-fungsi yang berguna untuk aplikasinya nanti.



Gambar 3.55. *Package Util*

- e. Kemudian di dalam *package* util ini, buat beberapa *file* berikut:

File Config.kt berisi kode untuk menyimpan *base URL* dari API yang digunakan aplikasi ini.

```
package com.quick.auditverifikasi.util

object Config {
    const val BASE_API = "http://api_url/api"
}
```

Gambar 3.56. *Code Di File Config.kt*

File AppMode.kt yang berfungsi untuk mendefinisikan mode yang ada di aplikasi ini.

```
package com.quick.auditverifikasi.util

enum class AppMode {
    Production,
    Development
}
```

Gambar 3.57. *Code Di File AppModel.kt*

Class yang berfungsi mencegah *user* meneklik tombol berulang kali.

```
package com.quick.auditverifikasi.util

import ...

class SafeClickListener(
    private var defaultInterval: Int = 2000,
    private val onSafeClickListener: (View) -> Unit
): View.OnClickListener {

    private var lastTimeClicked: Long = 0

    override fun onClick(v: View) {
        if (SystemClock.elapsedRealtime() - lastTimeClicked < defaultInterval) return
        lastTimeClicked = SystemClock.elapsedRealtime()
        onSafeClickListener(v)
    }
}
```

Gambar 3.58. Code Safe Button Click Listener Untuk Semua Buton

Click Listener

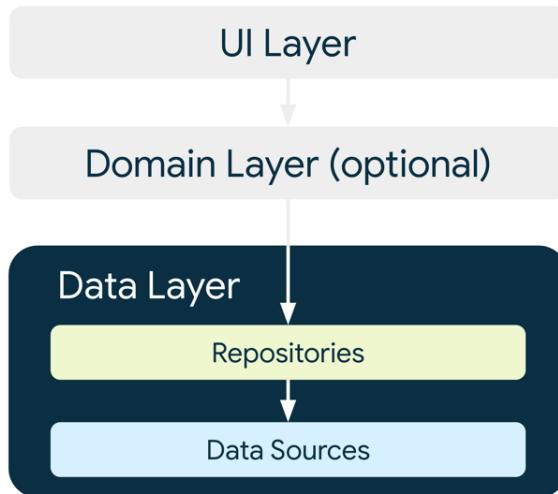
- Kemudian buat *package* data. *Package* data ini berisi kelas-kelas yang bertanggung jawab sebagai layer data untuk aplikasi ini.



Gambar 3.59. Package Data Untuk Aplikasi Audit & Verification

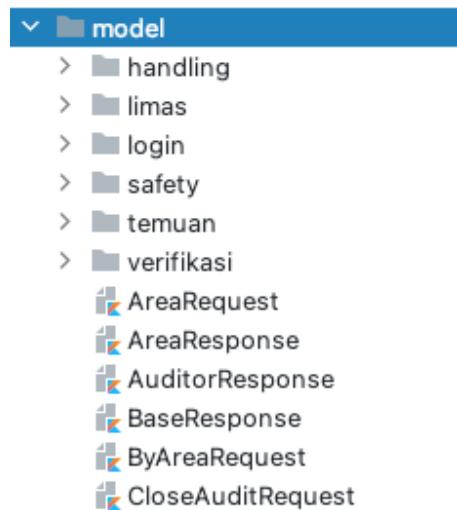
Data Layer sendiri adalah bagian yang berisi *business logic*. *Business logic* adalah bagian yang memberi nilai pada aplikasi, yang terbuat dari aturan yang menentukan cara bagaimana aplikasi membuat, menyimpan, dan mengubah data.

Data Layer terbuat dari *repository* yang masing-masing dapat berisi nol hingga banyak sumber data.



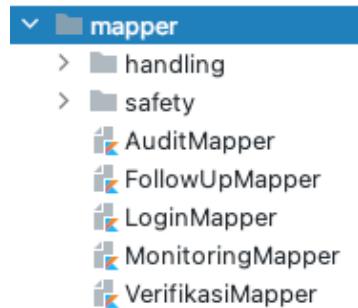
Gambar 3.60. Data Layer Didalam Architecture MVVM

- g. Kemudian di dalam *package* data, buat *package* model. *Package* model ini berisi data-data *class* yang berfungsi sebagai *data transfer object* untuk semua yang berhubungan dengan *network*.



Gambar 3.61. *Package Model*

- h. Setelah itu, buat package lain dengan nama mapper. Dalam package ini berisi *object-object* yang berfungsi untuk mengubah data transfer *object* menjadi domain *object*.



Gambar 3.62. Package Mapper

Semua mapper dalam package mapper memiliki struktur kode yang sama. Yaitu berbentuk *object* dengan satu *function* toDomain() untuk mengubah data dari data *transfer object* menjadi data domain. Berikut adalah contoh salah satu mapper. Mapper ini berfungsi untuk mengubah dto login menjadi domain *object* login.

```
package com.quick.auditverifikasi.data.mapper

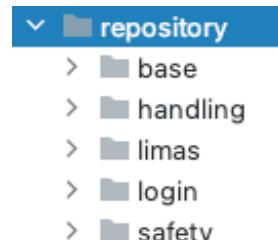
import ...

object LoginMapper {
    map LoginRemote to Login domain
    Params: loginRemote - LoginRemote
    Returns: Login domain

    fun toDomain(loginRemote: LoginRemote): Login {
        return Login(
            loginRemote.error,
            loginRemote.userId,
            loginRemote.user,
            loginRemote.employee,
            loginRemote.kodeSie,
            loginRemote.lokasi
        )
    }
}
```

Gambar 3.63. Code Object Mapper

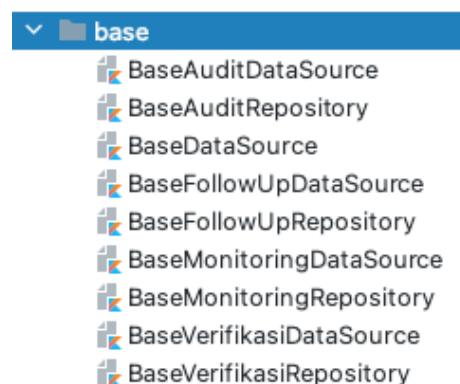
- i. Kemudian buat package lain dengan nama *repository*. Package ini berisi kelas-kelas *data source* dan *repository* untuk aplikasi ini.



Gambar 3.64. *Package Repository*

Dalam *package* ini ada *package* utama dengan nama *base*.

Jadi semua *data source* dan *repository* mewarisi *interface* *base* ini.



Gambar 3.65. *Package Base Repository*

Kemudian buat semua *base data source* seperti berikut ini.

```

package com.quick.auditverifikasi.data.repository.base

import com.quick.auditverifikasi.data.model.Resource

interface BaseAuditDataSource : BaseDataSource {

    suspend fun getArea(): Resource<List<String>>

    suspend fun getKategoriTemuan(): Resource<List<String>>

    suspend fun getAuditor(): Resource<Map<String, String>>

    suspend fun getTim(): Resource<List<String>>
}
  
```

Gambar 3.66. *Interface Base Audit Data Source*

Pada *interface base* audit tersebut, *interface* ini mewarisi *BaseDataSource*, sehingga *BaseAuditDataSource* ini menjadi memiliki semua *behavior* yang ada di *interface BaseDataSource*. Kemudian *function-function* yang ada di dalamnya menggunakan operator *suspend fun* yang berarti *function* ini dapat ditunda pemanggilannya dan dapat dijalankan kembali kapan pun secara *asynchronous* melalui *suspend function* yang lain atau melalui *coroutine scope*.

Kemudian *function-function* ini juga mengembalikan *object* yang bertipe *Resource<T>*, di mana *Resource* ini adalah sebuah *generic wrapper class* yang berfungsi untuk wrapping data yang di dalam tipe *generic*-nya dan layer UI kita untuk mengetahui semua kemungkinan *network request* yang dibuat, seperti ketika sukses, *error* ataupun ketika *loading*. Sehingga prosesnya dapat dengan mudah dideteksi oleh layer UI.

```
package com.quick.auditverifikasi.data.model

data class Resource<out T>(val status: Status, val data: T?, val message: ResourceMessage?) {

    enum class Status {
        SUCCESS,
        ERROR,
        LOADING
    }

    companion object {
        fun <T> success(data: T, message: ResourceMessage? = null): Resource<T> {
            return Resource(Status.SUCCESS, data, message)
        }

        fun <T> error(message: ResourceMessage, data: T? = null): Resource<T> {
            return Resource(Status.ERROR, data, message)
        }

        fun <T> loading(data: T? = null): Resource<T> {
            return Resource(Status.LOADING, data, null)
        }
    }
}

data class ResourceMessage(val title: String, val message: String)
```

Gambar 3.67. Data Class Resource

Ini adalah contoh *base repository* audit.

```
package com.quick.auditverifikasi.data.repository.base

import ...

interface BaseAuditRepository {

    fun getKategoriTemuan(): Flow<Resource<List<String>>>

    fun getArea(): Flow<Resource<List<String>>>

    fun getTim(): Flow<Resource<List<String>>>

    fun getAuditor(): Flow<Resource<Map<String, String>>>

}
```

Gambar 3.68. *Interface Base Audit Repository*

Interface ini digunakan sebagai kontrak yang harus dipenuhi oleh implementasi sesungguhnya dari *repository* ini. *Interface* ini berisi beberapa *function* biasa yang mengembalikan tipe data Kotlin Flow (*Flow<Resource<T>>*). Kotlin Flow sendiri adalah tipe yang dapat memancarkan beberapa nilai secara berurutan, berlawanan dengan *suspend function* yang mengembalikan hanya satu nilai.

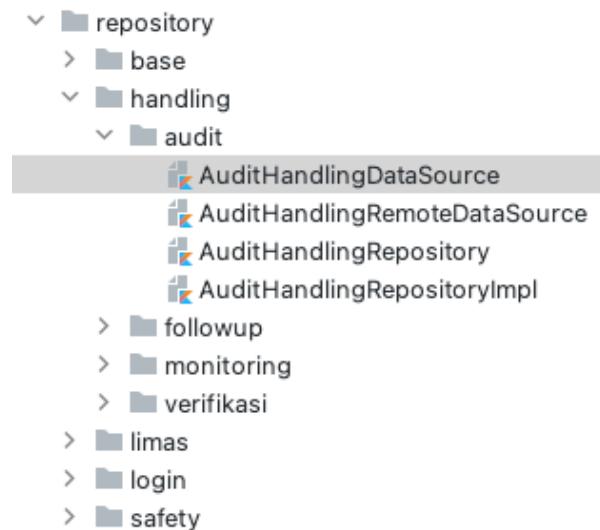
Repository ini adalah kelas yang dipanggil secara langsung oleh viewmodel untuk melakukan *network request*, sehingga *function – function*nya menggunakan *return type* Flow, karena *repository* dapat memancarkan/*emit* banyak nilai secara berurutan. Hal ini dilakukan karena *network request* adalah sebuah operasi panjang yang tidak pasti berhasil, sehingga dengan menggunakan flow ini, viewmodel dapat mengetahui state *network request* yang sedang terjadi, seperti sedang *loading*, *error*, atau sukses.

- j. Kemudian buat package lain yang merupakan implementasi dari *base repository* dan data source yang dibuat di package *base*, seperti *handling*, *limas*, *login* dan *safety*.



Gambar 3.69. *Package Repository Handling*

Setiap *package repository handling*, *limas*, dan *safety* memiliki struktur berikut di dalamnya.



Gambar 3.70. *Package Repository Handling*

Package audit digunakan untuk membuat audit baru. *Package follow up* berisi kode yang digunakan untuk *follow up* audit yang telah dibuat. *Package verifikasi* digunakan untuk memverifikasi *follow up* yang telah dibuat terhadap audit tersebut. Dan yang terakhir *package monitoring* berfungsi untuk me-*monitoring* semua audit yang ada di aplikasi ini.

Berikut adalah salah satu *data source audit handling* yang mewarisi *interface base audit data source* yang ada di *package base*.

```
package com.quick.auditverifikasi.data.repository.handling.audit

import ...

interface AuditHandlingDataSource : BaseAuditDataSource {
    suspend fun saveAudit(auditHandling: AuditHandlingRequest): Resource<SavedAudit>
    suspend fun getSaranaHandling(): Resource<List<String>>
}
```

Gambar 3.71. *Interface Audit Handling Data Source*

Berikut adalah salah satu *repository audit handling* yang mewarisi *interface base audit repository* yang ada di *package base*.

```
package com.quick.auditverifikasi.data.repository.handling.audit

import ...

interface AuditHandlingRepository : BaseAuditRepository {
    fun saveAudit(auditRequest: AuditHandlingRequest): Flow<Resource<SavedAudit>>
    fun getSaranaHandling(): Flow<Resource<List<String>>>
}
```

Gambar 3.72. *Interface Audit Handling Repository*

Kemudian berikut adalah salah satu *data source audit* di bagian *handling*.

```
package com.quick.auditverifikasi.data.repository.handling.audit

import ...

class AuditHandlingRemoteDataSource(private val httpClient: HttpClient)
    : AuditHandlingDataSource {
    override suspend fun getSaranaHandling(): Resource<List<String>> { ... }

    @InternalAPI
    @KtorExperimentalAPI
    override suspend fun saveAudit(auditHandling: AuditHandlingRequest)
        : Resource<SavedAudit> { ... }

    override suspend fun getArea(): Resource<List<String>> { ... }

    override suspend fun getKategoriTemuan(): Resource<List<String>> { ... }

    override suspend fun getAuditor(): Resource<Map<String, String>> { ... }

    override suspend fun getTim(): Resource<List<String>> { ... }
}
```

Gambar 3.73. *Class Audit Handling Data Source*

Class data source ini adalah bentuk implementasi dari *interface audit data source*. *Class* ini memiliki *constructor* yang berisi *http client*, yang nantinya akan di *inject* oleh *dependency injector*. *http client* ini digunakan sebagai *client* yang digunakan untuk melakukan *network request*.

Kemudian buat semua *function* dengan struktur kode seperti berikut. *Function* ini digunakan untuk mendapatkan semua sarana *handling* dari api menggunakan *http request*.

```
override suspend fun getSaranaHandling(): Resource<List<String>> {
    return dataStoreRunCatching {
        val response =
            httpClient.get<BaseResponse<SaranaHandlingResponse>>
                ("${Config.BASE_API}handling/sarana_handling") {
                    addDefaultHeader()
                }

        val result = response.data?.let { data →
            val dataResult = data.map {
                it.saranaHandling
            }
            Resource.success(dataResult)
        } ?: noDataFromServer(response)

        result
    }
}
```

Gambar 3.74. *Function* Di *Class Data Source*

Semua *network request* yang dilakukan di *data source*, dilakukan di dalam *extension function* *dataStoreRunCatching*. *Extension function* ini berbentuk *suspend inline function* yang ditambahkan ke *interface* *BaseDataSource*, sehingga yang dapat menggunakan *function* ini hanya kode yang mewarisi *interface* *BaseDataSource*. *Function* ini berfungsi untuk *error handling* dari

network request yang dilakukan, sehingga kode ini tidak perlu ditulis di semua *network request* yang dilakukan di semua *data source*.

```

suspend inline fun <R> BaseDataSource.dataStoreRunCatching(block: () -> Resource<R>):
    Resource<R> {
    return try {
        block()
    } catch (ex: ClientRequestException) {
        ex.printStackTrace()
        val result = ex.response.tryReceive<BaseResponse<FollowUp5SResponse>>()
        result.message?.let {
            return Resource.error(ResourceMessage(ex.response.status.description, it))
        }
        Resource.error(
            ResourceMessage(
                ex.response.status.description,
                "Client Error (no error message from server) \n${ex.message}"
            )
        )
    } catch (ex: ServerResponseException) {
        ex.printStackTrace()
        val result = ex.response.tryReceive<BaseResponse<FollowUp5SResponse>>()
        result.error?.let {
            return Resource.error(
                ResourceMessage(
                    ex.response.status.description,
                    "Server Error \n$it"
                )
            )
        }
        Resource.error(
            ResourceMessage(
                ex.response.status.description,
                "Server Error (no error message from server) \n${ex.message}"
            )
        )
    } catch (ex: Exception) {
        ex.printStackTrace()
        Resource.error(ResourceMessage("App Exception", ex.message))
    }
}

```

Gambar 3.75. Extension Function Untuk Semua Data Source

Kemudian untuk melakukan *network request*, gunakan *object httpClient* dari *library Ktor*. Caranya adalah dengan memanggil method *get<T>(URL: String)*, dengan T adalah *generic type* jenis tipe data yang diminta dari *network request* yang dilakukan, dan URL sebagai endpoint dari API-nya. Kemudian *addDefaultHeader()* digunakan untuk menambah *default header* untuk setiap *request* yang dibuat. *Default header* ini berisi jenis *connection* yang digunakan, apakah *development*

atau *production* berdasarkan pilihan mode yang dipilih *user* ketika login di aplikasi ini.

```
val response =
    httpClient.get<BaseResponse<SaranaHandlingResponse>>
    ("${Config.BASE_API}handling/sarana_handling") {
        addDefaultHeader()
}
```

Gambar 3.76. *Code Untuk Mendapatkan generic Response Dari*

Server

Kemudian setelah data diperoleh dari API, ubah data dari API tersebut sesuai kebutuhan dan kembalikan hasil yang didapatkan. Dalam kasus ini, jika hasil dari *network request* tidak *null* maka akan di map kemudian diubah menjadi *list of sarana handling* dan kemudian disimpan dalam *object Resource.success()* yang kemudian nanti akan dikembalikan. Tetapi jika *response-nya null*, maka akan mengembalikan *error* tidak ada data dari server.

```
val result = response.data?.let { data →
    val dataResult = data.map {
        it.saranaHandling
    }
    Resource.success(dataResult)
} ?: noDataFromServer(response)

result
```

Gambar 3.77. *Code Untuk Mengubah generic Response Dari Server*

Ke Bentuk Yang Dibutuhkan

Kemudian buat kode untuk *repository*. *Repository* adalah bagian yang menangani operasi data. Mereka menyediakan API yang bersih sehingga aplikasi lainnya dapat mengambil data ini dengan mudah. Mereka tahu dari mana mendapatkan data dan panggilan API

apa yang harus dilakukan saat data diperbarui. *Repository* dapat dianggap sebagai mediator antara berbagai *data source* yang berbeda, seperti model persisten, layanan web, dan *cache*. Berikut adalah salah satu contoh *repository* dari audit *handling*.

```
package com.quick.auditverifikasi.data.repository.handling.audit

import ...

class AuditHandlingRepositoryImpl(val dataSource: AuditHandlingDataSource)
    : AuditHandlingRepository {

    override fun getSaranaHandling(): Flow<Resource<List<String>>> { ... }

    override fun saveAudit(auditRequest: AuditHandlingRequest)
        : Flow<Resource<SavedAudit>> { ... }

    override fun getKategoriTemuan(): Flow<Resource<List<String>>> { ... }

    override fun getArea(): Flow<Resource<List<String>>> { ... }

    override fun getTim(): Flow<Resource<List<String>>> { ... }

    override fun getAuditor(): Flow<Resource<Map<String, String>>> { ... }
}
```

Gambar 3.78. Class Implementasi *Interface Repository*

Class ini adalah implementasi dari *interface audit handling repository*. Kelas ini berfungsi untuk menghubungkan viewmodel dan *data source*. Di bagian *constructor class* ini membutuhkan *audit handling data source* yang nantinya akan di *inject* oleh *dependency injector*.

Berikut ini adalah salah satu *function* yang ada di *class audit handling repository*. *Function* ini adalah *function* yang mengembalikan flow yang berisi *resource* dari sebuah list of string. Karena ini adalah flow, jadi *function* ini dapat memancarkan beberapa nilai secara berurutan. Sehingga ketika *function* ini dipanggil, pertama dia akan memancarkan *state loading*, kemudian *delay* selama 400 untuk

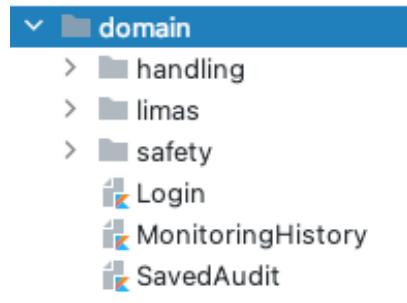
menunjukkan proses *loading*, kemudian memancarkan lagi nilai yang didapatkan dari *data source*.

```
override fun getSaranaHandling(): Flow<Resource<List<String>>> {
    return flow {
        emit(Resource.loading())
        delay(400)
        emit (dataSource.getSaranaHandling())
    }
}
```

Gambar 3.79. Function Di Class Implementasi *Repository*

Semua *data source* dan *repository* di aplikasi ini memiliki dasar kode seperti yang sudah penulis jelaskan diatas.

- k. Kemudian buat *package domain*, *package* ini adalah *package* yang berisi *bussines model* dari aplikasi ini. Dalam *package* ini berisi model-model *class* yang dipakai di seluruh aplikasi ini.



Gambar 3.80. Package Domain

Berikut adalah contoh salah satu domain model audit *handling*.

```
package com.quick.auditverifikasi.domain.handling

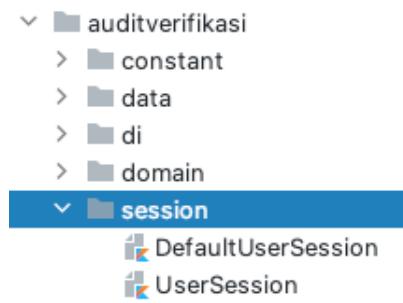
import java.util.*

data class AuditHandling(
    private var id: Int,
    private var status: String?,
    private var auditor: String?,
    private var tanggalAudit: Date?,
    private var tim: String?,
    private var area: String?,
    private var saranaHandling: String?,
    private var detailLokasiTemuan: String?
)
```

Gambar 3.81. Data Class Audit *Handling*

Semua domain model adalah sebuah data *class* yang berisi berbagai *property* dari sebuah entity tersebut.

- I. Kemudian buat *package* baru lagi dengan nama *session*. *Package* ini berisi kode yang berfungsi untuk mengatur *user session* di dalam aplikasi ini. Dalam *package* ini berisi *interface UserSession* dan implementasi dari *UserSession* yang bernama *DefaultUserSession*.



Gambar 3.82. *Package Session*

Berikut adalah kode dari *interface UserSession*. *Function startUserSession* berfungsi untuk memulai *user session*, *function* ini dipanggil setelah *user* berhasil login dengan *username* dan *password* yang benar. Kemudian *endUserSession* berfungsi untuk mengakhiri *user session*, *function* ini dipanggil ketika *user* melakukan logout dari keadaan sudah login.

```

package com.quick.auditverifikasi.session

interface UserSession {
    fun startUserSession(username: String, password: String, mode: String)
    fun endUserSession()
    fun getUser(): String
    fun isUserLogged(): Boolean
    fun getTheme(): String
    fun setTheme(theme: String)
    fun getMode(): String
}
  
```

Gambar 3.83. *Interface Session*

Berikut adalah isi dari *class DefaultUserSession*. *Class* ini adalah implementasi dari *interface UserSession*.

```
package com.quick.auditverifikasi.session

import com.quick.auditverifikasi.App.Companion.prefs

class DefaultUserSession : UserSession {

    override fun startUserSession(username: String, password: String, mode: String) { ... }

    override fun endUserSession() { ... }

    override fun getMode(): String { ... }

    fun setMode(mode: String) { ... }

    override fun getTheme(): String { ... }

    override fun setTheme(theme: String) { ... }

    override fun getUser(): String { ... }

    override fun isUserLogged(): Boolean { ... }

}
```

Gambar 3.84. Class Implemtasi Dari *Interface Session*

Dalam *function startUserSession*, berisi menyimpan pengaturan ketika login berhasil login, seperti *username*, *password*, dan mode aplikasinya, yang kemudian disimpan di dalam shared preferences.

```
override fun startUserSession(username: String, password: String,
                             mode: String) {
    prefs.clear()
    prefs.push("username", username)
    prefs.push("password", password)
    prefs.push("mode", mode)
    prefs.push("theme", mode)
}
```

Gambar 3.85. Function Untuk Memulai Session Baru

Object *prefs* adalah *object* yang dibuat di *class Application* sehingga dapat dipanggil di seluruh *project*. *Object* ini digunakan untuk melakukan operasi dengan shared preferences. Shared preferences sendiri adalah salah satu teknologi yang digunakan untuk menyimpan

data *primitive* berbentuk pasangan *key value* yang kemudian disimpan di *storage* Android.

```
@KtorExperimentalAPI
class App : Application() {

    override fun onCreate() { ... }

    companion object {
        lateinit var applicationContext: Context
        lateinit var prefs: KsPrefs

        fun isPrefsInitialized() = ::prefs.isInitialized
    }
}
```

Gambar 3.86. Variabel prefs Di Class App

Kemudian *function endUserSession* berisi menghapus semua data yang tersimpan di *shared preferences* untuk aplikasi ini.

```
override fun endUserSession() {
    prefs.clear()
}
```

Gambar 3.87. Function Untuk Menghentikan Session

Kemudian untuk *function-function* lainnya hanya mengambil dan menyimpan data dari *shared preferences* sesuai dengan namanya, dan untuk yang mengambil data jika data tersebut tidak ada maka diberi *default value* di parameter keduanya..

```
override fun getMode(): String {
    return prefs.pull("mode", "DEV")
}

fun setMode(mode: String) {
    prefs.push("mode", mode)
}

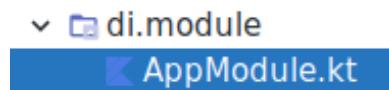
override fun getTheme(): String {
    return prefs.pull("theme", "DEV")
}

override fun setTheme(theme: String) {
    prefs.push("theme", theme)
}
```

Gambar 3.88. Code Lain di Class Session

m. Kemudian buat *package* baru dengan nama *di.module*. Di dalam *package* inilah *dependency injection* dibuat. *Dependency Injection* adalah sebuah teknik untuk mengatur cara bagaimana suatu objek dibentuk ketika terdapat objek lain yang membutuhkan. Sehingga ketika sebuah kode membutuhkan *instance* dari sebuah *object* dan *object* tersebut dibutuhkan dibanyak tempat, kita dapat menggunakan *dependency injection* ini untuk mendapatkan instance tersebut.

Untuk membuat *dependency injection* di aplikasi ini akan menggunakan *library* Koin. Untuk mengatur *dependency injection* menggunakan Koin, buat kotlin *file* baru dengan nama *AppModule.kt* di *package* *di.module*.



Gambar 3.89. *Package DI Module*

Kemudian buat kode untuk *file* tersebut seperti berikut.

```
package com.quick.auditverifikasi.di.module

import ...

private const val TIME_OUT = 60_000

val viewModelModule = module { ... }

val repositoryModule = module { ... }

val appModule = module { ... }

@KtorExperimentalAPI
val ktorModule = module { ... }
```

Gambar 3.90. *Code File Modules.kt*

Setiap *module* berisi instruksi untuk memberitahu *dependency injector* bagaimana cara membuat sebuah *instance* ketika ada yang membutuhkannya.

Buat deklarasi berikut untuk semua view model yang ada di aplikasi ini. Dengan begitu Koin dapat mengenali view model tersebut dan dapat memberikannya Ketika ada yang memintanya.

```
viewModel { this: Scope
    MonitoringSafetyViewModel(get(), get())
}
```

Gambar 3.91. Pendeklarasian View Model

Kemudian buat deklarasi untuk semua view model dan *repository*. Berikut adalah salah satu pendeklarasian *module repository*.

```
single<LoginDataSource> { this: Scope
    LoginRemoteDataSource(get())
}
```

Gambar 3.92. Pendeklarasian *Data Source*

Semua *module* di *file* ini memiliki struktur kode yang sama seperti di atas. Untuk membuat agar *module-module* ini dapat digunakan, *module* tersebut harus dimuat di *function onCreate* di *class Application*.

```
startKoin { this: KoinApplication
    // declare used Android context
    androidContext( androidContext: this@App)
    // declare modules
    modules(listOf(ktorModule, viewModelModule, appModule, repositoryModule))
}
```

Gambar 3.93. Code Untuk Memuat *Module* Di Aplikasi Audit & Verification

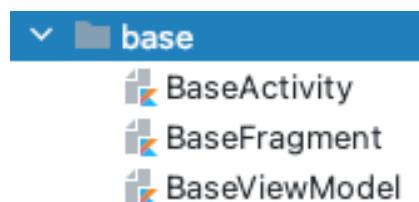
n. Kemudian buat *package ui*. *Package ui* ini berperan sebagai UI Layer.

Peran utama dari UI layer (lapisan presentasi) adalah untuk menampilkan data aplikasi di layar. Setiap kali data berubah, baik karena interaksi pengguna (seperti menekan tombol) atau input eksternal (seperti respons jaringan), UI harus diperbarui untuk mencerminkan perubahan tersebut.



Gambar 3.94. *Package UI*

Untuk *package ui.base*, berisi *base* / dasar *class* untuk semua *class* di *package ui* ini.



Gambar 3.95. *Package Base UI*

Berikut adalah kode untuk *class base activity*. Class ini adalah sebuah *abstract class* yang mewarisi *class app compat activity*, karena *class* ini adalah *abstract class* sehingga *class* ini tidak bisa diinisialisasi secara langsung.

```
package com.quick.auditverifikasi.ui.base

import ...

abstract class BaseActivity : AppCompatActivity() {

    lateinit var loadingOverlay: LoadingOverlay
    var mLastClickTime: Long = 0

    var loadingDialog: InfoSheet? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        MainScope().launch { ... }

        title = this.javaClass.simpleName
            .removeSuffix("Activity")
            .split(Regex("(?=\\p{Upper})"))
            .joinToString(postfix = "", separator = " ")
    }

    open fun onPrefReady() {
    }
}
```

Gambar 3.96. *Class Base Activity* Untuk Semua *Activity* Di Aplikasi

Audit & Verification

Kemudian berikut adalah kode di *class base fragment*.

```
package com.quick.auditverifikasi.ui.base

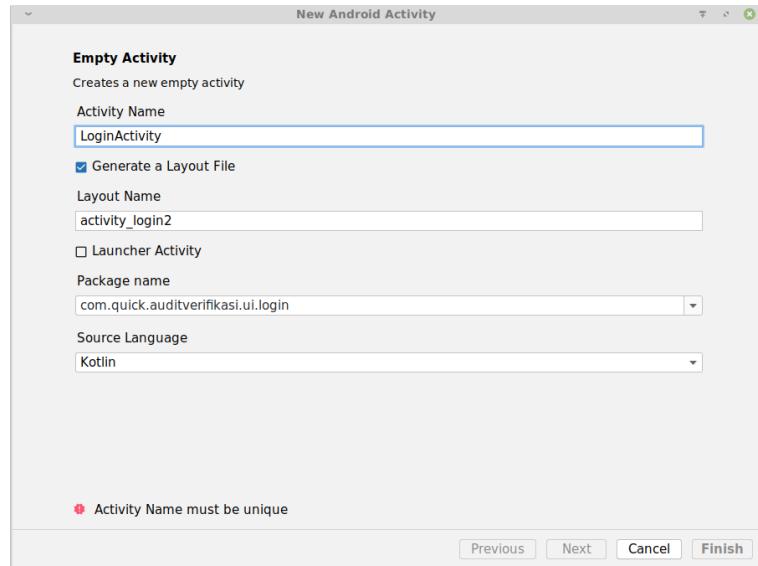
import ...

abstract class BaseFragment : Fragment() {
    lateinit var loadingOverlay: LoadingOverlay
    var mLastClickTime: Long = 0
    lateinit var parent: BaseActivity
}
```

Gambar 3.97. *Class Base Fragment* Untuk Semua *Fragment* Di

Aplikasi Audit & Verification

- o. Kemudian buat *activity* login untuk menampilkan halaman login. Untuk membuat *activity* baru, klik kanan di *package* tempat *activity* tersebut ingin dibuat, kemudian pilih *New > Activity > Empty Activity*.



Gambar 3.98. Jendela Untuk Membuat *Activity* Baru

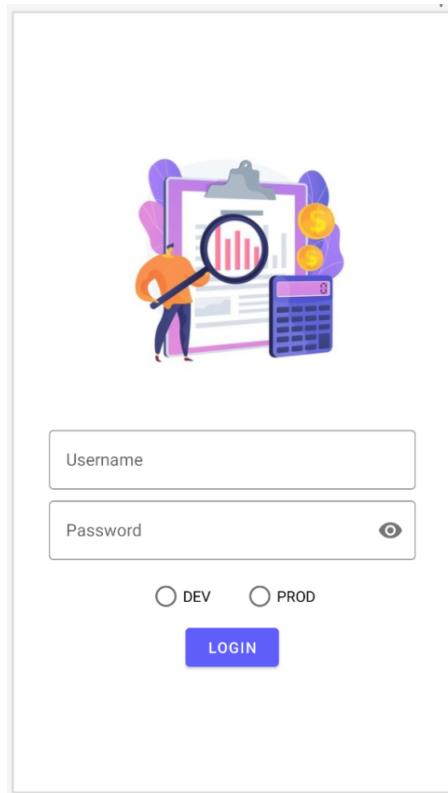
Kemudian akan muncul dialog untuk membuat *activity* baru seperti di atas, beri nama *ActivityLogin* dan pastikan *Generate a Layout File* dipilih, dan juga pilih bahasa pemrograman untuk *activity* ini. Stelah selesai tekan *Finish* untuk membuat *activity*-nya.

- p. Untuk membuat UI di Android, dapat dilakukan dengan menggunakan *file XML*.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Gambar 3.99. Contoh Membuat UI dengan XML

Buka *file XML* dari *activity* login ini dan buat UI-nya menjadi seperti berikut ini.



Gambar 3.100. UI *Activity* Login

- q. Setelah UI dari *activity* login dibuat, kemudian buat kode untuk *activity* login ini. Buka *file ActivityLogin* dan buat

```
package com.quick.auditverifikasi.ui.login

import ...

class LoginActivity : BaseActivity() {

    lateinit var binding: ActivityLoginBinding
    private var session: UserSession = get()

    private val viewModel by viewModel<LoginViewModel>()

    override fun onCreate(savedInstanceState: Bundle?) { ... }
}
```

Gambar 3.101. Code *Activity* Login

Kemudian di *function* `onCreate`, buat kode berikut untuk mengecek apakah *user* sebelumnya sudah login atau belum, jika sudah maka tutup *activity* ini dan buka *activity* main.

```
if (session.isUserLogged()) {
    startActivity(Intent(packageContext: this@LoginActivity, MainActivity::class.java))
    finish()
}
```

Gambar 3.102. Pengecekan Apakah *User* Sudah Login Atau Belum

Kemudian buat kode untuk menghandle *event user* menekan tombol login.

```
binding.buttonLogin.setOnClickListener { ... }
```

Gambar 3.103. Code Untuk Mengatur Click Listener

Ketika *user* menekan tombol login, yang pertama kali dilakukan adalah mengambil *username* dan *password* yang *user* masukan.

```
val username = binding.editTextUsername.text.toString()
val password = binding.editTextPassword.text.toString()
```

Gambar 3.104. Code Untuk Mendapatkan *Username* Dan *Password*

Kemudian melakukan validasi untuk mengecek apakah *username* dan *password* yang *user* masukan valid atau tidak. Jika *username* dan *password* yang dimasukkan valid, maka melanjutkan ke proses selanjutnya, tetapi jika tidak, berhenti dan menampilkan pesan *error* sesuai dengan kondisinya.

```
if (username.trim().isEmpty()) {
    Toast.makeText(
        context: this@LoginActivity,
        text: "Tolong isi username dulu",
        Toast.LENGTH_SHORT
    ).show()
    return@setOnClickListener
}
```

Gambar 3.105. Code Untuk Validasi *Username*

```

        if (password.trim().isEmpty()) {
            Toast.makeText(
                context: this@LoginActivity,
                text: "Tolong isi password dulu",
                Toast.LENGTH_SHORT
            ).show()
            return@setOnClickListener
        }
    }
}

```

Gambar 3.106. Code Untuk Validasi Password

Kemudian setelah proses validasi berhasil, melakukan proses login.

```

viewModel.login(
    username,
    password
).observeOnce(lifecycleOwner: this) { it: Resource<NoData>!
    when (it.status) { ... }
}

```

Gambar 3.107. Code Untuk Melakukan Proses Login

Jika *username* dan *password* yang dimasukkan ada di *database*, maka *user* akan diarahkan ke *activity main* dan menyimpan session bahwa *user* telah login di *shared preferences*.

```

session.startUserSession(
    username,
    password,
    if (binding.radioButtonProd.isChecked)
        AppMode.Production.name
    else AppMode.Development.name
)
loadingOverlay.dismiss()
Toast.makeText(
    context: this@LoginActivity,
    text: "Login berhasil",
    Toast.LENGTH_SHORT
).show()
startActivity(
    Intent(
        packageContext: this@LoginActivity,
        MainActivity::class.java
    )
)
finish()

```

Gambar 3.108. Code Untuk Proses Login Jika Berhasil

Tetapi jika login gagal, maka akan menampilkan pesan *error*-nya.

```
loadingOverlay.dismiss()
showInfo(
    title: it.message?.title ?: "Login Gagal",
    message: it.message?.message ?: "Unknown Error"
) { }
if ((it.message?.message ?: "")  
    .contains( other: "password", ignoreCase: true)
) {
    binding.editTextPassword.error = it.message?.message ?: ""
}
```

Gambar 3.109. Code Ketika Login Gagal

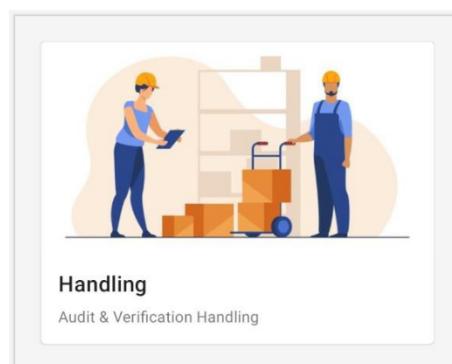
- r. Kemudian buat *activity* main untuk menampilkan halaman menu utama.

Activity ini dibuat dengan nama *MainActivity* dan berada di *package ui.main*.

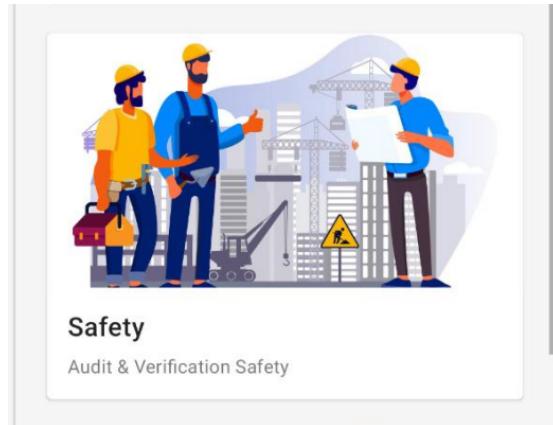


Gambar 3.110. Package Main

Lalu buat UI di *file XML activity main*. *Activity main* ini digunakan sebagai menu utama yang menampilkan daftar sub menu dari aplikasi ini. Sub menu tersebut adalah *Handling*, *Safety* dan *5S*. Setiap sub menu berisi fungsi dan alur yang hampir sama. Dimulai dari membuat audit, kemudian *follow up* audit, verifikasi audit dan yang terakhir *monitoring* data audit.



Gambar 3.111. UI Activity Main Menu *Handling*



Gambar 3.112. UI *Activity main Menu Safety*

s. Setelah itu buat kode di *activity main*-nya.

```
package com.quick.auditverifikasi.ui.main

import ...
class MainActivity : BaseActivity() {

    lateinit var binding: ActivityMainBinding

    private val session: UserSession = get()

    override fun onPrefReady() { ... }

    override fun onCreate(savedInstanceState: Bundle?) { ... }

    override fun onCreateOptionsMenu(menu: Menu): Boolean { ... }

    @ExperimentalCoroutinesApi
    @InternalCoroutinesApi
    override fun onOptionsItemSelected(item: MenuItem): Boolean { ... }

    private fun logout() { ... }

    override fun onBackPressed() { ... }
}
```

Gambar 3.113. *Code Main Activity*

Function onPrefReady() dijalankan ketika shared preferences sudah siap untuk digunakan. Di dalam *onPrefRaedy()* pertama-tama dilakukan pengecekan apakah *user* sudah benar-benar login atau

belum, jika belum maka akan diarahkan kembali ke halaman login, jika sudah melanjutkan ke proses selanjutnya,

```
if (!session.isUserLogged()) {
    startActivity(Intent(packageContext: this, LoginActivity::class.java))
    finish()
    return
}
```

Gambar 3.114. Pengecekan *User Session*

Kemudian dilakukan proses pengaturan tema sesuai dengan jenis mode yang dipilih *user* ketika login, yaitu mode *development* atau *production*.

```
setContentView(R.layout.activity_main)
val themeId = if (App.prefs.pull(
    key: "mode",
    AppMode.Development.name
) == AppMode.Production.name
) R.style.Theme_AplikasiAuditVerifikasiHandling_Prod
else R.style.Theme_AplikasiAuditVerifikasiHandling_Dev
theme.applyStyle(themeId, force: true)
```

Gambar 3.115. Mengatur Tema Di *Activity*

Di dalam *activity* ini terdapat 3 menu utama, yaitu menu *handling*, *safety*, dan *5S*. Setiap menu memiliki daftar kegiatan yang sama yaitu membuat audit, *follow up* audit, memverifikasi audit, dan me-*monitoring* audit. Sehingga buat kode untuk handle event ketika *user* memilih salah satu menu yang ada.

```
binding.menuHandlingCard.setOnSafeClickListener { it: View
    with(Intent(packageContext: this@MainActivity, MainHandlingActivity::class.java)) { this: Intent
        addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
        startActivity(
            intent: this,
            ActivityOptions.makeSceneTransitionAnimation(activity: this@MainActivity).toBundle()
        )
    }
}
```

Gambar 3.116. Listener Ketika *User* Menekan Menu *Handling*

```

binding.menuSafetyCard.setOnSafeClickListener { it: View
    with(Intent(packageContext: this@MainActivity, MainSafetyActivity::class.java)) {
        addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
        startActivity(
            intent: this,
            ActivityOptions.makeSceneTransitionAnimation(activity: this@MainActivity).toBundle()
        )
    }
}
}

```

Gambar 3.117. Listener Ketika User Menekan Menu Safety

Kemudian buat kode untuk event ketika user ingin logout dari aplikasi ini. Code berikut dipanggil ketika user menekan tombol logout di activity main, dan meminta konfirmasi dari user apakah yakin ingin melakukan logout dari aplikasi ini atau tidak.

```

private fun logout() {
    showQuestion(
        title: "Konfirmasi",
        message: "Apa anda yakin ingin logout ?",
        {},
        onPositive = {
            session.endUserSession()
            startActivity(Intent(packageContext: this@MainActivity, LoginActivity::class.java))
            finish()
        }
    )
}

```

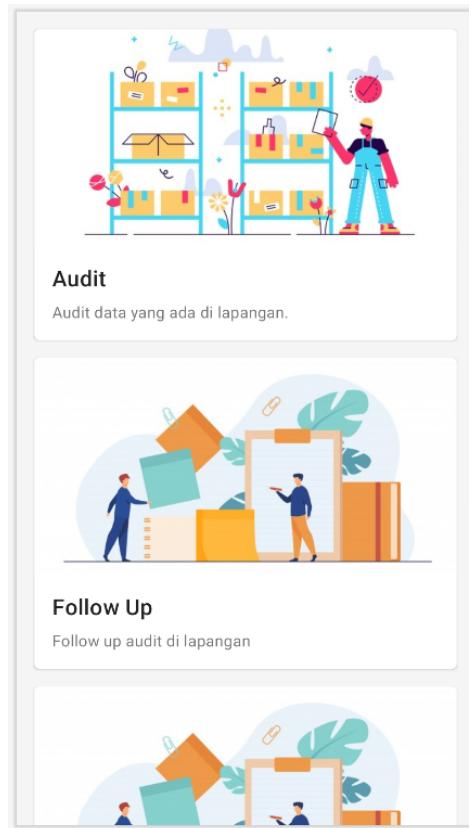
Gambar 3.118. Code Konfirmasi Sebelum Logout

- Kemudian buat package baru untuk sub menu pertama yaitu menu *handling*. package ini berada di ui.*handling*.



Gambar 3.119. Package Handling

Setelah itu buat *activity main handling* dan dengan UI seperti berikut.



Gambar 3.120. UI Main Activity

Setelah UI dari *activity* ini dibuat, kemudian buat kode untuk *activity* ini.

```
package com.quick.auditverifikasi.ui.handling

import ...

class MainHandlingActivity : BaseActivity() {

    lateinit var binding: ActivityMainHandlingBinding
    private var session: UserSession = get()

    override fun onCreate(savedInstanceState: Bundle?) { ... }

    override fun onBackPressed() { ... }
}
```

Gambar 3.121. Activity Main Handling

Hal utama yang dilakukan *activity* ini adalah menampilkan semua menu dari sum menu *handling* ini, dan mengarahkan ke menu

yang dipilih *user*. Sehingga buat kode untuk meng-handle event *user* memilih menu tertentu dan mengarahkan ke menu yang dituju.

```

setSingleClickListener(binding.auditCard, clickInterval: 2000) {
    val intent = Intent(
        packageContext: this@MainHandlingActivity,
        AuditHandlingActivity::class.java
    )
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    startActivity(intent)
}

setSingleClickListener(binding.verifikasiCard, clickInterval: 2000) {
    val intent =
        Intent(
            packageContext: this@MainHandlingActivity,
            VerifikasiHandlingActivity::class.java
        )
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    startActivity(intent)
}

setSingleClickListener(binding.followUpCard, clickInterval: 2000) {
    val intent =
        Intent(
            packageContext: this@MainHandlingActivity,
            FollowUPHandlingActivity::class.java
        )
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    startActivity(intent)
}

setSingleClickListener(binding.monitoringCard, clickInterval: 2000) {
    val intent =
        Intent(
            packageContext: this@MainHandlingActivity,
            MonitoringHandlingActivity::class.java
        )
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    startActivity(intent)
}

```

Gambar 3.122. Button Listener Untuk Setiap Menu

Di *activity* ini juga ada permintaan konfirmasi ketika *user* menekan tombol kembali.

```

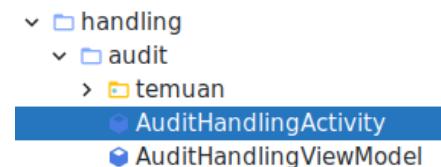
override fun onBackPressed() {
    showQuestion(
        title: "Konfirmasi",
        message: "Apa anda yakin ingin kembali ke menu utama ?",
        {},
        onPositive = {
            super.onBackPressed()
        }
}

```

Gambar 3.123. Konfirmasi Sebelum Kembali Ke Menu Utama

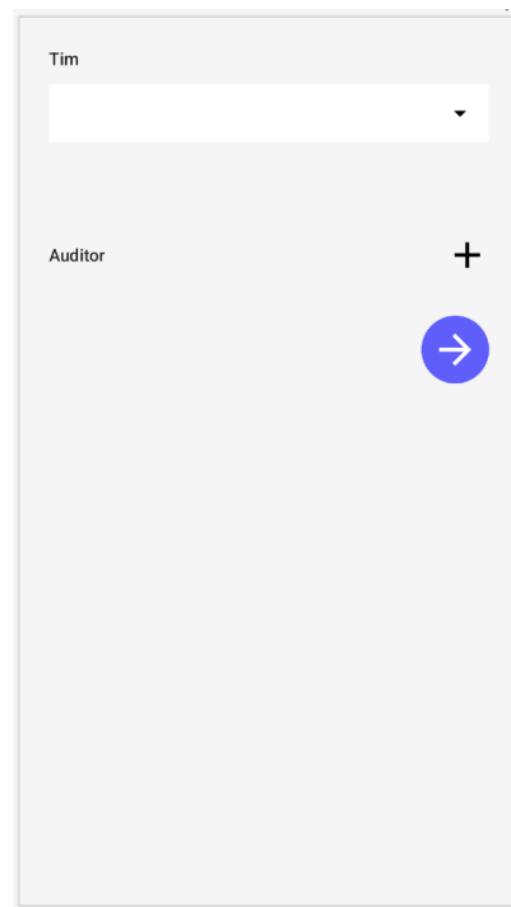
Jika *user* mengkonfirmasi maka akan kembali ke menu utama, tetapi jika tidak maka tidak akan melakukan apa-apa.

- u. Kemudian buat *activity audit handling* untuk membuat audit baru di sub menu *handling* ini.



Gambar 3.124. Package Audit Handling

Kemudian buat UI untuk *activity audit handling* seperti berikut.



Gambar 3.125. UI Activity Audit Handling

Setelah UI-nya jadi, buat kode untuk *activity audit handling*-nya.

```
package com.quick.auditverifikasi.ui.handling.audit

import ...

class AuditHandlingActivity : BaseActivity() {

    private val viewModel: AuditHandlingViewModel by viewModel()

    lateinit var binding: ActivityAuditBinding

    private lateinit var auditorSearchModels: ArrayList<SearchModelTwo>
    private lateinit var searchDialogCompat: SimpleSearchDialogCompat<SearchModelTwo>

    private var resultLauncher =
        registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { ... }

    override fun onCreate(savedInstanceState: Bundle?) { ... }

    private fun observeViewModel() { ... }

    override fun onBackPressed() { ... }
}
```

Gambar 3.126. *Code Activity Audit Handling*

Kemudian di *function onCreate()* buat kode berikut yang akan dijalankan ketika *user* menambah auditor baru. Jika *user* yang dipilih sudah ada, maka akan menampilkan pesan peringatan bahwa *user* yang dipilih sudah ditambahkan sebelumnya.

```
binding.buttonAddAuditor.setOnSafeClickListener { it: View
    searchDialogCompat = SimpleSearchDialogCompat(
        context: this@AuditHandlingActivity,
        "Pilih Auditor",
        "Masukan nama auditor yang ingin dipilih",
        filter: null,
        auditorSearchModels
    ) model@{ dialog: BaseSearchDialogCompat<*>, item: SearchModelTwo, _: Int →
        for (value in viewModel.selectedAuditor) {
            if (value == item) {
                Toast.makeText(
                    context: this@AuditHandlingActivity,
                    text: "${item.title} sudah ada",
                    Toast.LENGTH_SHORT
                ).show()
                return@model
            }
        }
        viewModel.selectedAuditor.add(item)
        binding.listAuditor.adapter = ArrayAdapter(
            context: this@AuditHandlingActivity,
            R.layout.view_text,
            viewModel.selectedAuditor.map { it.title }
        )
        dialog.dismiss()
    }
    searchDialogCompat.show()
}
```

Gambar 3.127. *Code Untuk Menambah Data Auditor*

Kemudian buat kode untuk meng-handle ketika *user* ingin menghapus data auditor yang sudah dimasukkan. Sebelum data auditor benar-benar akan dihapus, lakukan konfirmasi dari *user*.

```
binding.listAuditor.onItemLongClickListener =
    AdapterView.OnItemLongClickListener { _, AdapterView<>?, _: View?, position: Int, _: Long →
        showQuestion(
            title: "Konfirmasi",
            message: "Apa anda yakin ingin menghapus auditor " + viewModel.selectedAuditor[position].title + "?",
            onPositive = {
                viewModel.selectedAuditor.removeAt(position)
                binding.listAuditor.adapter = ArrayAdapter(
                    context: this@AuditHandlingActivity,
                    android.R.layout.simple_list_item_1,
                    viewModel.selectedAuditor.map { it.title }
                )
            },
            onNegative = {}
        )
        false ^OnItemLongClickListener
    }
```

Gambar 3.128. Code Untuk Menghapus Data Auditor

Kemudian buat kode untuk meng-handle ketika *user* menekan tombol *next* ke *activity* selanjutnya. Sebelum ke *activity* selanjutnya untuk menambah audit baru, dilakukan pengecekan apakah semua input yang ada sudah dimasukkan dengan benar oleh *user* atau belum, jika belum akan dimunculkan pesan *error*. Tetapi jika sudah benar semua, maka akan di arahkan ke *activity* data temuan audit *handling*.

```
binding.ivNext.setOnSafeClickListener { it: View
    val intent =
        Intent(
            packageContext: this@AuditHandlingActivity,
            AuditDataTemuanHandlingActivity::class.java
        )
    if (binding.spinnerTim.selectedItem.toString()
        .isBlank() || binding.spinnerTim.selectedItem.toString() == "Pilih"
    ) {
        showInfo(title: "Warning", message: "Tim tidak boleh kosong") { }
        return@setOnSafeClickListener
    }
    if (viewModel.selectedAuditor.isEmpty()) {
        showInfo(title: "Warning", message: "Auditor tidak boleh kosong") { }
        return@setOnSafeClickListener
    }
    intent.putExtra(name: "tim", binding.spinnerTim.text.toString())
    intent.putStringArrayListExtra(
        name: "auditor",
        java.util.ArrayList(viewModel.selectedAuditor.map { it.id })
    )
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    resultLauncher.launch(intent)
}
```

Gambar 3.129. Code Untuk Event *User* Menekan Tombol *Next*

Kemudian yang terakhir buat kode di *function* `observeViewModel()`, *function* ini dijalankan untuk mendapatkan data tim dan auditor dari Rest API dan menampilkannya ke UI. Dalam kode ini juga terdapat mekanisme untuk meng-*handle* berbagai kemungkinan ketika melakukan *network request*. Seperti menampilkan pesan *error* ketika *network request* gagal.

```
private fun observeViewModel() {

    viewModel.fetchTim().observeOnce(lifecycleOwner: this) { it: Resource<List<String>>!
        when (it.status) {
            Resource.Status.LOADING → { ... }
            Resource.Status.SUCCESS → {
                binding.shimmerTim.visibility = View.GONE
                binding.spinnerTim.visibility = View.VISIBLE

                it.data?.let { list →
                    binding.spinnerTim.attachDataSource(list)
                }
            }
            Resource.Status.ERROR → { ... }
        }
    }

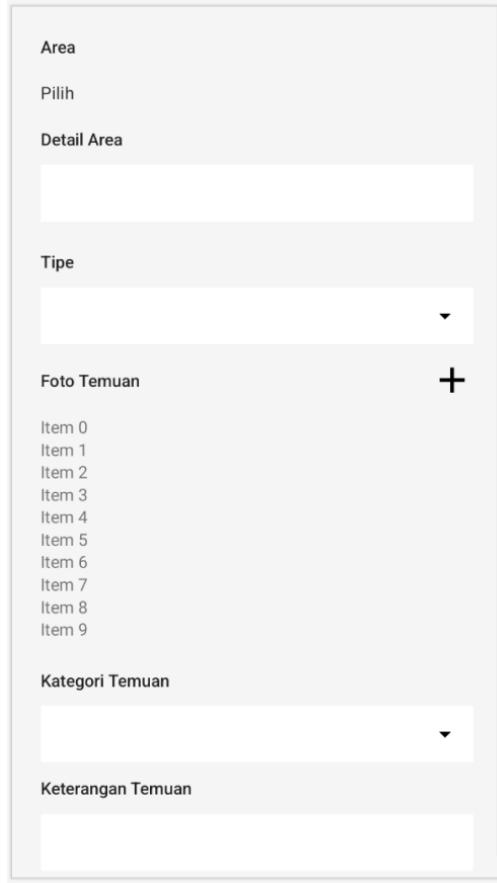
    viewModel.fetchAuditor().observeOnce(lifecycleOwner: this) { it: Resource<Map<String, String>>!
        when (it.status) {
            Resource.Status.LOADING → { ... }
            Resource.Status.SUCCESS → {
                binding.shimmerAuditor.visibility = View.GONE
                binding.buttonAddAuditor.visibility = View.VISIBLE

                auditorSearchModels.clear()
                it.data?.let { list →
                    list.forEach { that →
                        auditorSearchModels.add(SearchModelTwo(that.key, that.value))
                    }
                }
            }
            Resource.Status.ERROR → { ... }
        }
    }
}
```

Gambar 3.130. Code Untuk Mendapatkan Data Time Dan Auditor

- v. Kemudian buat *activity* data temuan audit *handling*. *Activity* ini dibuat di `package ui.handling.temuan` dengan nama `AuditDataTemuanHandling`. *Activity* ini digunakan untuk membuat data temuan audit dan menyimpannya ke *database*.

Setelah *activity* dibuat, buat kode untuk UI-nya seperti berikut.



Gambar 3.131. UI Untuk *Activity* Membuat Audit Baru

Kemudian buat kode untuk *activity*-nya seperti berikut.

```
package com.quick.auditverifikasi.ui.handling.audit.temuan

import ...

@KtorExperimentalAPI
class AuditDataTemuanHandlingActivity : BaseActivity() {

    lateinit var binding: ActivityAuditDataTemuanBinding
    lateinit var adapter: CustomFragmentPagerAdapterItemAdapter
    private var currentPagePosition = 0
    private var session: UserSession = get()

    private val viewModel by viewModel<AuditDataTemuanHandlingViewModel>()

    @InternalCoroutinesApi
    override fun onCreate(savedInstanceState: Bundle?) { ... }

    override fun onCreateOptionsMenu(menu: Menu): Boolean { ... }

    @InternalCoroutinesApi
    fun saveTemuan() { ... }
}
```

Gambar 3.132. Code *Activity Audit Data Temuan Handling*

Kemudian buat kode di *function saveTemuan()*, *function* ini dijalankan ketika *user* menyimpan data temuan yang sudah dimasukkan.

```
@InternalCoroutinesApi
fun saveTemuan() {
    var fragments: MutableList<DataTemuanHandlingFragment> = ArrayList()

    for (i in 0 until adapter.count) { ... }

    if (fragments.isEmpty()) { ... }

    fragments = fragments.filter { it.isChanged() }.toMutableList()

    if (fragments.isEmpty()) { ... }

    val listOfTemuan = mutableListOf<DataTemuanHandling>()

    for ((index, fragment) in fragments.withIndex()) { ... }

    showQuestion(
        title: "Konfirmasi",
        message: "Apakah anda yakin sudah selesai menginputkan temuan ini ? \n${fragments.joinToString(
            separator = "\n"
        )} { \"Temuan ${it.pageIndex + 1} di area ${it.binding.textPilihArea.text}\" }",
        onPositive = {
            saveAuditRecursive(fragments, listOfTemuan)
        },
        onNegative = { })
}
```

Gambar 3.133. Code Untuk Menyimpan Temuan Audit

Di dalam *function* ini juga dilakukan pengecekan data temuan yang telah dimasukkan, jika data yang dimasukkan tidak benar maka proses akan berhenti dan menampilkan pesan *error*-nya.

```
if (fragments.isEmpty()) {
    showInfo( title: "Warning", message: "Tolong isi minimal satu temuan") { }
    return
}

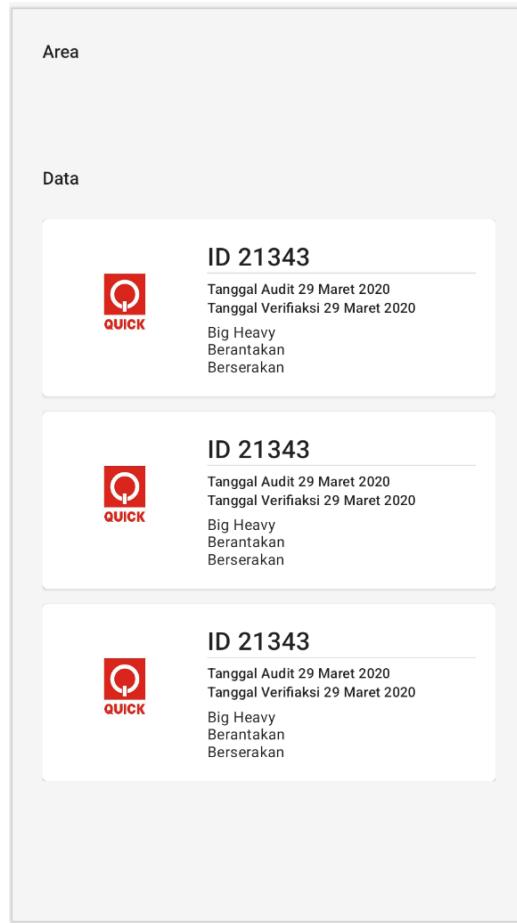
fragments = fragments.filter { it.isChanged() }.toMutableList()

if (fragments.isEmpty()) {
    showInfo( title: "Warning", message: "Tidak ada perubahan yang terdeteksi ") { }
    return
}
```

Gambar 3.134. Validasi Sebelum Menyimpan Data Audit

w. Kemudian buat *activity follow up audit handling*. *Activity* ini digunakan untuk mem-*follow up* audit yang telah dibuat oleh audior, proses *follow up* ini dilakukan oleh *auditee*. Buat *activity* baru dengan nama *FollowUpHandlingActivity* di package *ui.handling.followup*.

Setelah *activity* terbuat, buat kode untuk UI dengan nama *activity_select_area.xml* seperti berikut. File XML ini akan digunakan sebagai UI untuk semua menu kecuali audit, seperti di menu *follow up*, verifikasi dan *monitoring*.



Gambar 3.135. UI Untuk Menampilkan Audit Untuk Area Tertentu

Setelah itu buat kode untuk *activity*-nya. seperti berikut.

```
package com.quick.auditverifikasi.ui.handling.followup

import ...

class FollowUPHandlingActivity : BaseActivity() {
    private val GALLERY_IMAGE_REQ_CODE = 102
    private val CAMERA_IMAGE_REQ_CODE = 103

    lateinit var binding: ActivitySelectAreaBinding

    private lateinit var searchModels: ArrayList<SearchModel>
    private lateinit var searchDialogCompat: SimpleSearchDialogCompat<SearchModel>

    private var session: UserSession = get()

    private val viewModel by viewModel<FollowUpHandlingViewModel>()

    private fun followUp(idAudit: Int, keterangan: String) {...}

    override fun onCreate(savedInstanceState: Bundle?) {...}

    private fun fetchFollowUp(area: String) {...}

    private fun showVerificationDetail(data: FollowUpHandling) {...}

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {...}

    private fun observeViewModel() {...}
}
```

Gambar 3.136. *Code Activity Follow up Handling*

Kemudian buat kode di *function* `fetchFollowUp()`, *function* ini dipanggil untuk mendapatkan semua data audit *handling* yang belum di-*follow up* dan menampilkannya ke UI.

```
private fun fetchFollowUp(area: String) {
    if (area.isBlank() || area == "Pilih") {
        binding.epoxyRecyclerView.clear()
        return
    }
    viewModel.fetchFollowUp(area).observe(owner: this@FollowUPHandlingActivity) { resource →
        when (resource.status) {
            Resource.Status.LOADING → {...}
            Resource.Status.SUCCESS → {...}
            Resource.Status.ERROR → {...}
        }
    }
}
```

Gambar 3.137. *Code Untuk Mendapatkan Semua Audit Yang Bisa Di-
follow up*

Setelah itu buat kode di *function followUp()*, *function* ini dipanggil ketika *user* menyimpan data *follow up* sesuai audit yang telah dipilih. Setelah proses penyimpanan dijalankan, akan ditampilkan pesan sesuai dengan hasil proses penyimpanannya.

```
private fun followUp(idAudit: Int, keterangan: String) {
    viewModel.followUp(
        idAudit,
        keterangan,
        session.getUser()
    ).observeOnce(
        lifecycleOwner: this@FollowUPHandlingActivity
    ) { it: Resource<NoData>!
        when (it.status) {
            Resource.Status.LOADING → loadingOverlay.show()
            Resource.Status.SUCCESS → {
                viewModel.deleteAllImage()
                loadingOverlay.dismiss()
                showInfo(
                    title: "Success",
                    message: "Sukses Follow up audit"
                ) {
                    fetchFollowUp(binding.textPilihArea.text.toString())
                }
            }
            Resource.Status.ERROR → {
                viewModel.deleteAllImage()
                loadingOverlay.dismiss()
                showInfo(
                    title: it.message?.title ?: "Error",
                    message: it.message?.message ?: "Unknown Error"
                ) {}
            }
        }
    }
}
```

Gambar 3.138. Code Untuk *Follow up Audit*

- x. Kemudian buat *activity* baru dengan nama *VerifikasiHandlingActivity* di *package ui.handling.verifikasi*. *Activity* ini digunakan untuk memverifikasi data audit yang telah difollow up oleh *auditee*, proses verifikasi ini dilakukan oleh auditor yang telah membuat audit ini.

Activity ini tetap menggunakan UI yang sama dengan *activity* *follow up*. Untuk kode *activity*-nya buat seperti berikut.

```
package com.quick.auditverifikasi.ui.handling.verifikasi

import ...

class VerifikasiHandlingActivity : BaseActivity() {
    lateinit var binding: ActivitySelectAreaBinding

    private var session: UserSession = get()

    private val viewModel by viewModel<VerifikasiHandlingViewModel>()

    private var searchModels: ArrayList<SearchModel>? = null
    private var searchDialogCompat: SimpleSearchDialogCompat<SearchModel>? = null

    override fun onCreate(savedInstanceState: Bundle?) { ... }

    private fun fetchVerifikasi(paramArea: String = "") { ... }

    private fun showVerifikasiDetail(data: VerifikasiHandling) { ... }

    private fun closeAudit(idAudit: Int) { ... }

    private fun openAudit(idAudit: Int, alasanMasihOpen: String) { ... }

    private fun observeViewModel() { ... }
}
```

Gambar 3.139. *Code Activity Verifikasi Handling*

Kemudian buat kode di *function* *fetchVerifikasi()*, *function* ini akan dipanggil untuk mendapatkan semua data audit yang bisa diverifikasi sesuai area tertentu dan menampilkannya ke UI.

```
private fun fetchVerifikasi(paramArea: String = "") {
    var area = paramArea
    if (area == "") {
        area = binding.textPilihArea.text.toString()
    }
    if (area.isBlank() || area == "Pilih") {
        binding.epoxyRecyclerView.clear()
        return
    }
    viewModel.fetchVerifikasi(area).observe(owner: this) { resource →

        when (resource.status) {
            Resource.Status.LOADING → { ... }
            Resource.Status.SUCCESS → { ... }
            Resource.Status.ERROR → { ... }
        }
    }
}
```

Gambar 3.140. *Code Untuk Mendapatkan Data Verifikasi*

Kemudian buat kode di *function closeAudit()*, *function* ini dipanggil untuk menyimpan data verifikasi yang telah diputuskan oleh *user* apakah audit tersebut sudah bisa di-*close* atau masih *open*. Sudah di-*close* artinya data temuan audit tersebut sudah diperbaiki, namun sebaliknya jika *open* berarti data temuan audit tersebut belum diperbaiki dengan benar.

```
private fun closeAudit(idAudit: Int) {
    viewModel.closeAudit(
        idAudit,
        session.getUser()
    ).observeOnce(
        lifecycleOwner: this@VerifikasiHandlingActivity
    ) { it: Resource<NoData>!
        when (it.status) {
            Resource.Status.LOADING → loadingOverlay.show()
            Resource.Status.SUCCESS → {
                loadingOverlay.dismiss()
                showInfo(
                    title: "Success",
                    message: "Sukses CLOSE audit"
                )
                fetchVerifikasi()
            }
            Resource.Status.ERROR → {
                loadingOverlay.dismiss()
                showInfo(
                    title: it.message?.title ?: "Error",
                    message: it.message?.message
                        ?: "Unknown Error"
                )
            }
        }
    }
}
```

Gambar 3.141. Code Untuk Menutup Audit

- y. Kemudian buat *activity* baru dengan nama *MonitoringHandlingActivity* di *package ui.handling.monitoring*. *Activity* ini digunakan untuk menampilkan semua data audit yang ada di sub menu *handling* ini, baik yang belum di-*follow up* sama sekali, sudah di-*follow up* tetapi belum

diverifikasi ataupun yang sudah diverifikasi dan selesai proses auditnya. Aktiviy ini juga masih menggunakan UI yang sama dengan *activity follow up* dan verifikasi.

Setelah *activity*-nya terbuat, buat kode untuk *activity* tersebut seperti berikut.

```
package com.quick.auditverifikasi.ui.handling.monitoring

import ...
import ...

class MonitoringHandlingActivity : BaseActivity() {

    lateinit var binding: ActivitySelectAreaBinding

    private val viewModel by viewModel<MonitoringHandlingViewModel>()

    lateinit var searchModels: ArrayList<SearchModel>
    private var searchDialogCompat: SimpleSearchDialogCompat<SearchModel>? = null

    override fun onCreate(savedInstanceState: Bundle?) { ... }

    private fun observeViewModel() { ... }

    private fun fetchMonitoring(area: String) { ... }

    private fun showMonitoringDetail(list: List<MonitoringHandling>, data: MonitoringHandling) { ... }
}
```

Gambar 3.142. *Code Activity Monitoring Handling*

Kemudian buat kode untuk *function fetchMonitoring()*, *function* ini dipanggil untuk mendapatkan semua data audit untuk sub menu *handling* dan menampilkan datanya ke UI.

```
private fun fetchMonitoring(area: String) {
    if (area.isBlank() || area == "Pilih") {
        binding.epoxyRecyclerView.clear()
        return
    }
    viewModel.fetchMonitoring(area).observe(owner: this) { resource →

        when (resource.status) {
            Resource.Status.LOADING → { ... }
            Resource.Status.SUCCESS → { ... }
            Resource.Status.ERROR → { ... }
        }
    }
}
```

Gambar 3.143. *Code Untuk Mendapatkan Data Monitoring Dari Server*

z. Setelah semua *activity* untuk sub menu *handling* sudah dibuat, ulangi langkah tersebut untuk membuat *activity* di submenu yang lain. Karena kode untuk semua sub menu hampir mirip dan yang membedakan adalah jenis auditnya saja.

6. Quality Control

Setelah aplikasi selesai dibuat, perlu dilakukan pengecekan pada semua fitur di aplikasi ini apakah ada *bug* atau pun *error*. Jika ada *bug* atau *error* perlu diperbaiki sampai masalah tersebut terselesaikan.

Setelah selesai laporan ke atasan bahwa aplikasi ini sudah selesai dan kemudian atasan dan *user* akan mengetes aplikasi ini apakah sesuai dengan kebutuhan atau belum.

7. Kesimpulan

Aplikasi Audit & Verification adalah aplikasi yang digunakan untuk melakukan audit di kawasan CV. Karya Hidup Sentosa Tusono. Aplikasi ini digunakan untuk menginput audit yang dilakukan oleh auditor jika ada audit yang ditemukan saat patroli. Jika ditemukan audit di suatu seksi, maka dengan menggunakan aplikasi ini untuk membuat audit baru.