

# Android Refactor

Tibor Molnár & Gábor Orosz

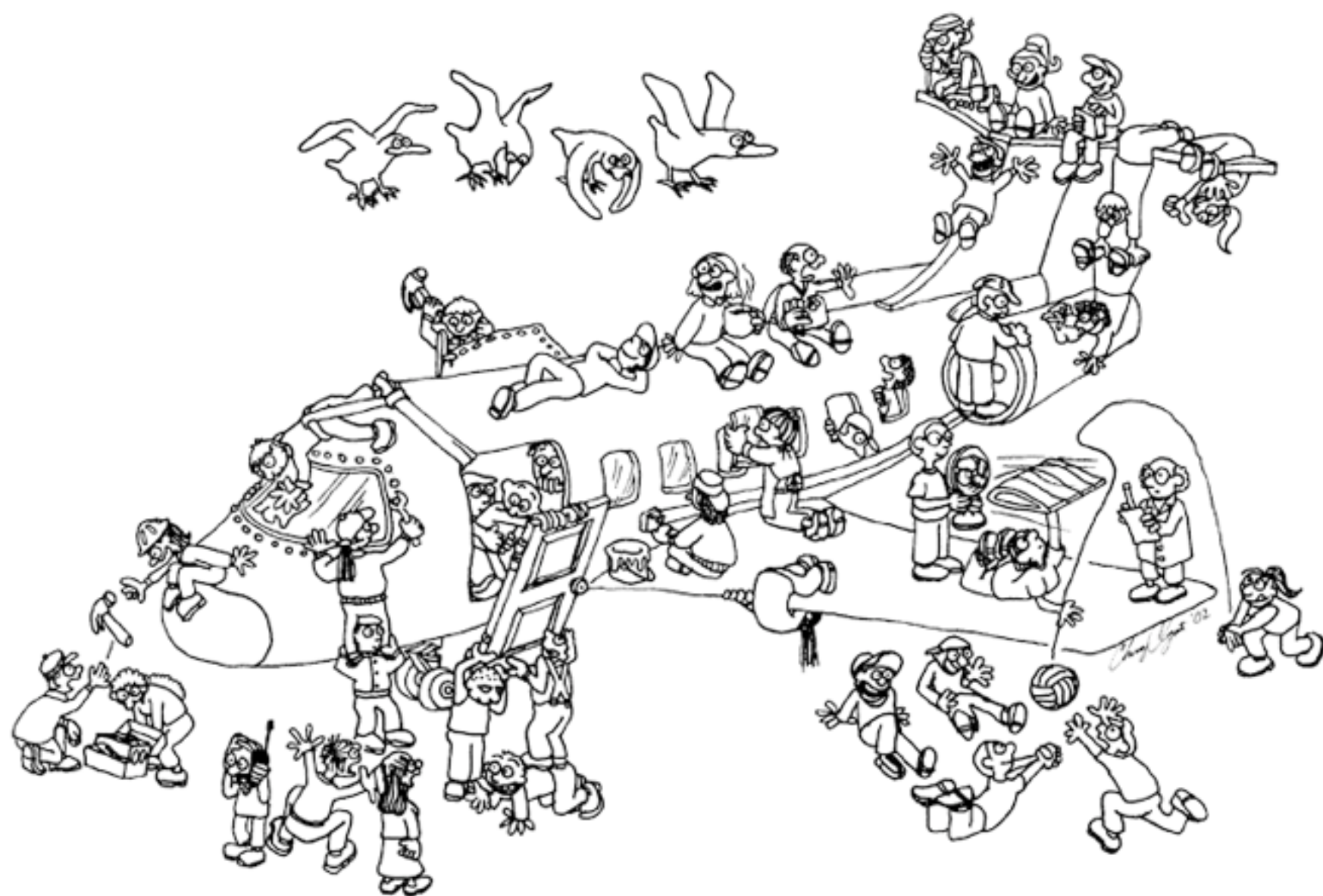
“Nincs az a kód, amit pár év alatt  
ne fogna meg az idő vasfoga.

Általában nem kell ennyi idő!”

*Ex-kolléga bölcsessége (szalonképes verzió)*

# Tartalom

- A legacy kód bemutatása
- A Google hozzáállása régen és most
- Kotlin gyortalpaló
- Android Architecture Components
  - Networking (OkHttp, Retrofit, Picasso)
  - Repository
  - Room
  - Live Data
  - ViewModel
- Mi maradt ki?
- Q&A



**bit.ly/**

**hwsw-android-app**

**hwsw-android-notes**

**hwsw-android-backend**

**[DEMO]**

**A teljes legacy kód bemutatása**

# Legacy problémák #1 - Activity

- Rengeteg dolog van egyben az Activityben, nem válnak el a felelősségi körök:
  - View és kód összekötése
  - User inputok kezelése
  - UI renderelés
  - Activity lifecycle kezelés
  - Networking\* és \*AsyncTask\* bele a kellős közepébe
  - Adatbázis\* kezelés

# Legacy problémák #2 - onCreate

- A **findViewById()** jelen helyzetben nem vészes, lehetne durvább is, láttam osztályokat ahol 15-20-30 elemet is kötöttek be így.
- Ami ellenben már zavaróbb, hogy az **onCreate()**-ben számos más felelősség kör is megjelenik, ezzel mindenképp kezdeni kell majd valamit.



# Legacy problémák #3 - AsyncTask

- Egy **AsyncTask** az **Activity** kellős közepében, ezzel számos probléma lehet.
- Alapvetően nem komplikált konstrukció, de könnyen be lehet nézni dolgokat, trükkös.
- Számos **memory leak** bújhat meg itt, például hivatkozások az **Activity**-be.
- Konfiguráció váltás, az **Activity** lifecycle változás számos problémát okozhat.
- Túl sok **AsyncTask** indítása esetében megtelik a queue, ez pedig crash-ez vezet.

# Legacy problémák #4 - JSON letöltés

- JSON-t akarok letölteni egy REST API-ról, de ez nagyjából 40 sor kódot igényel.
- Csúnya, nehezen átlátható, és számos apró csapdát rejt.

# Legacy problémák #5 - Adatbázis kezelés

- Ismét egy AsyncTask, erről már írtam feljebb.
- Egészen kicsi SQL queryket írunk, de ehhez képest rendkívül hosszú az eredmény.
- A sok string konkatenáció miatt nem tud segíteni az IDE, és az elgépelések csak futási időben jönnek ki.
- Számos apró hiba rejtőzhet ebben, amit nehéz kiszűrni.

**[CAMEO]**

**Kotlin alapozó Ádám**

# Az Android fejlesztés nehéz

|                               | Web      | Android          |
|-------------------------------|----------|------------------|
| Hello World                   | Egyszerű | Egyszerű         |
| Adat letöltés és megjelenítés | Egyszerű | Trükkös          |
| Komplex alkalmazás            | Nehéz    | Nehéz és trükkös |

A Google nem akart állást foglalni számos kérdésben, magukat a keretrendszerben alacsonyabb szintjére pozícionálták, a többit pedig a fejlesztőkre bízták.

**MVC**

**MVP**

**MVVM**

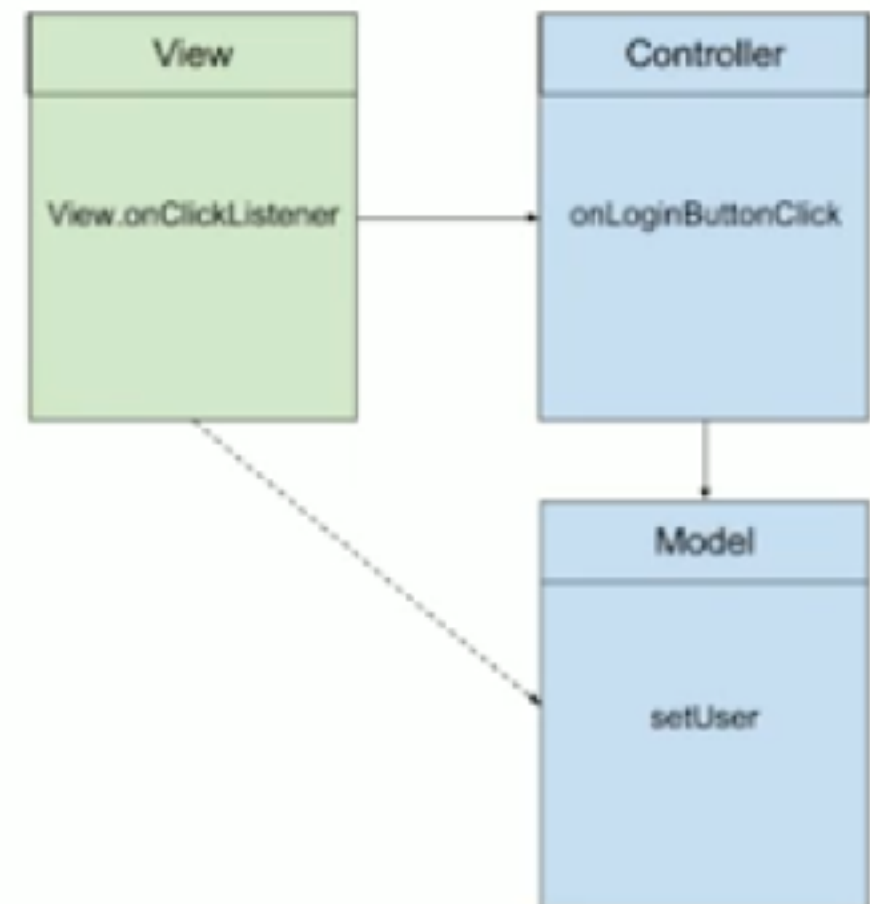
**Clean Architecture**

**VIPER**

**Uber RIBs**

# MVC

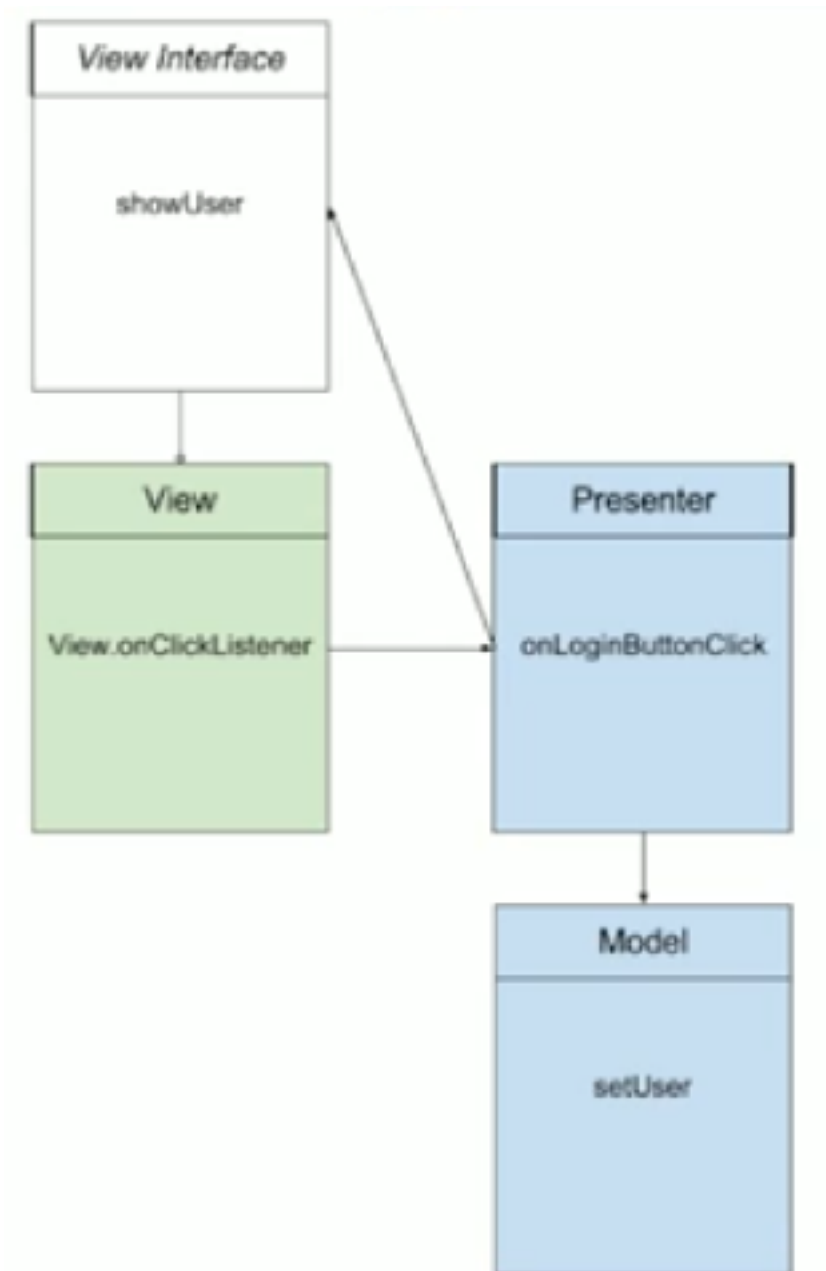
- 80-as évek Smalltalk
- Egy-irányú adatfolyam
- Direkt kapcsolat a **View** és a **Model** között
- Passzol a data bindinghoz





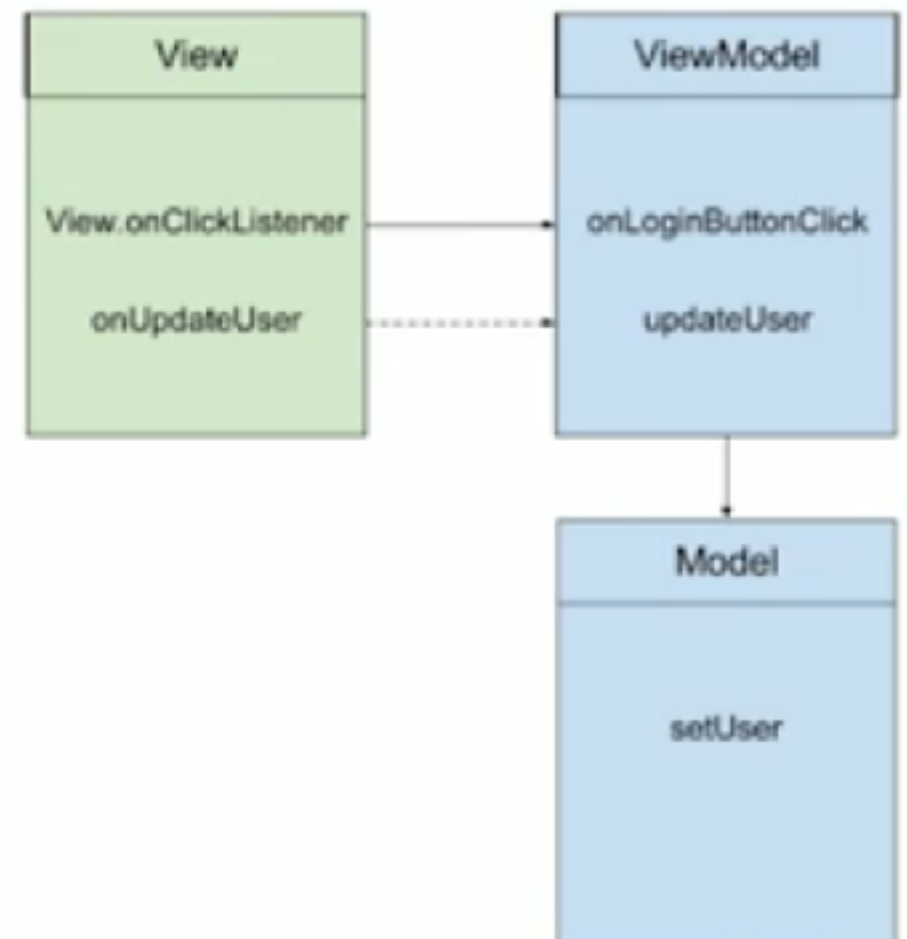
# MVP

- Fő célja, hogy kivigye a logikát a **View**-ből.
- Eltérések az **MVC**-vel szemben:
  - Több-irányú adatfolyam
  - **Nincs** direkt kapcsolat a **View** és a **Model** között, a **Presenter** közvetít (tesztelésnél szuper)
  - Parancs-alapú struktúra



# MVVM

- Minden szempontból a előző kettő között.
- **Nincs** direkt kapcsolat a View és a Model között, ViewModel közvetít
- A data binding ide is nagyon szépen illik.
- Figyelj a két kapcsolatra a View és a ViewModel között.



# Android fájdalmak

*Adat perzisztálás*

*Életciklus menedzselés*

*Alkalmazás modularizálás*

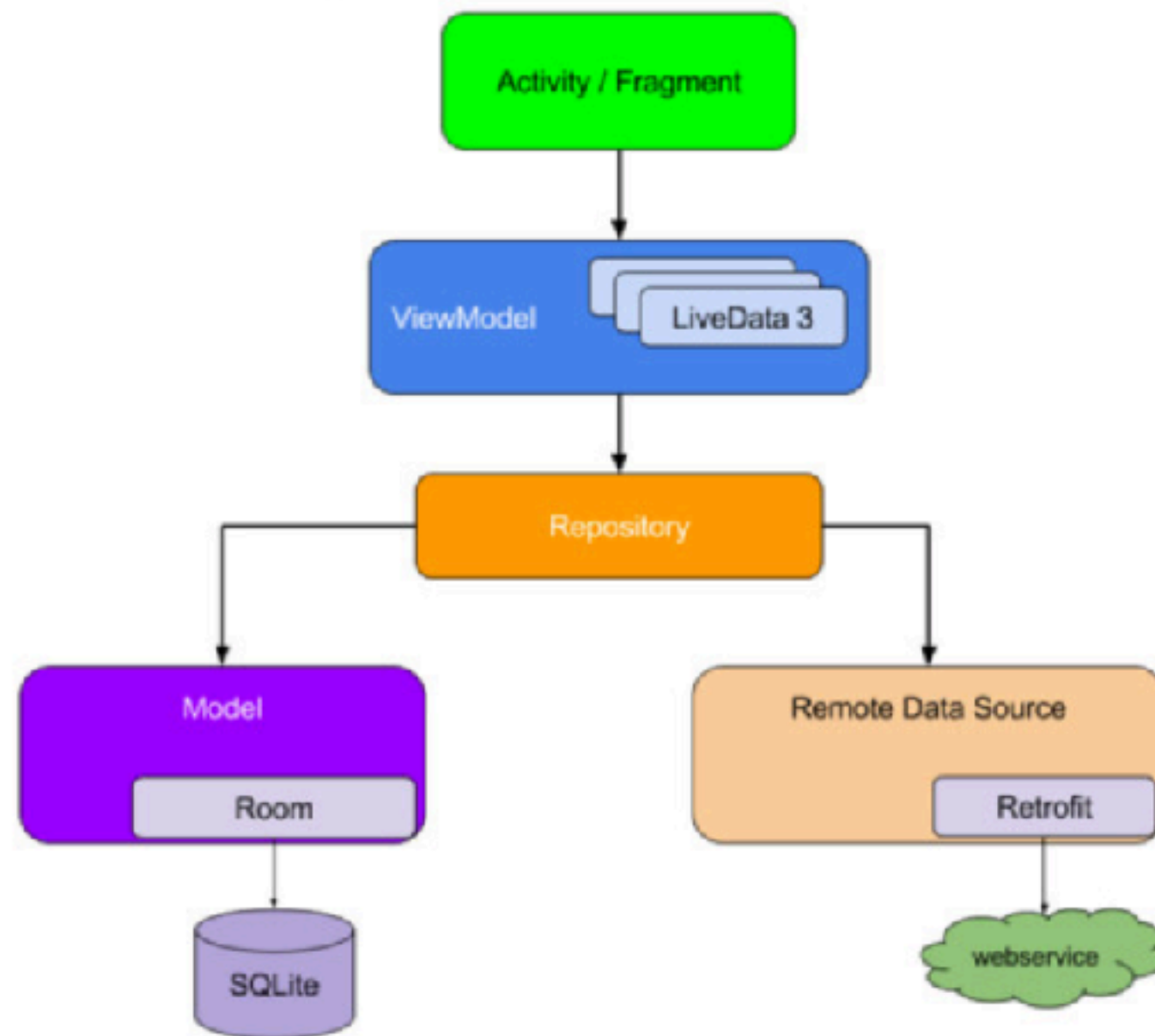
*Adat szinkronizálás*

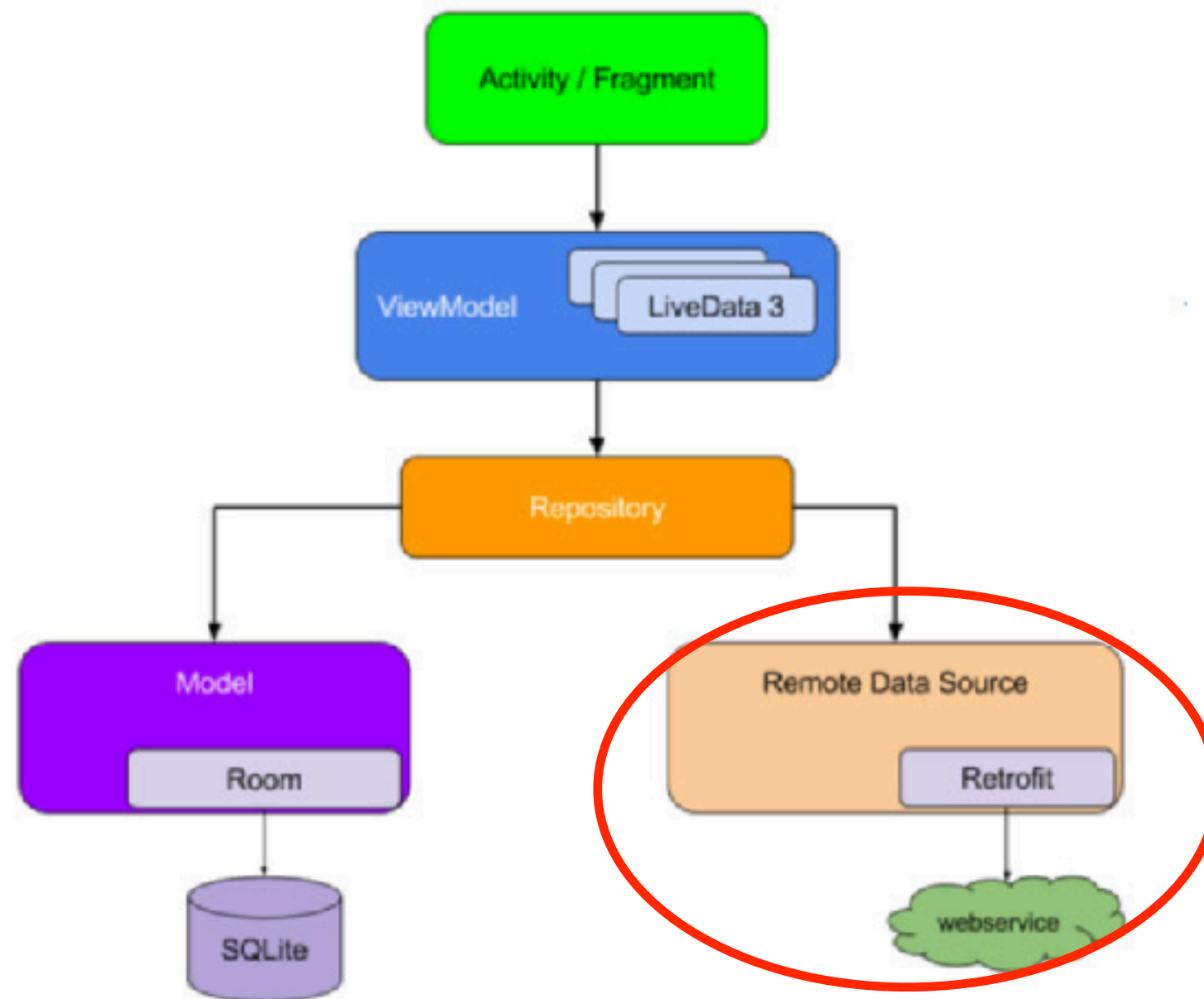
*Trükkös szálkezelés*

*Rengeteg boilerplate*

# Az Architecture Components céljai

- Az egyszerű dolgok tényleg legyenek egyszerűek.
- A bonyolultak legyenek lehetségesek.
- Ne találják fel újra a melegvizet.





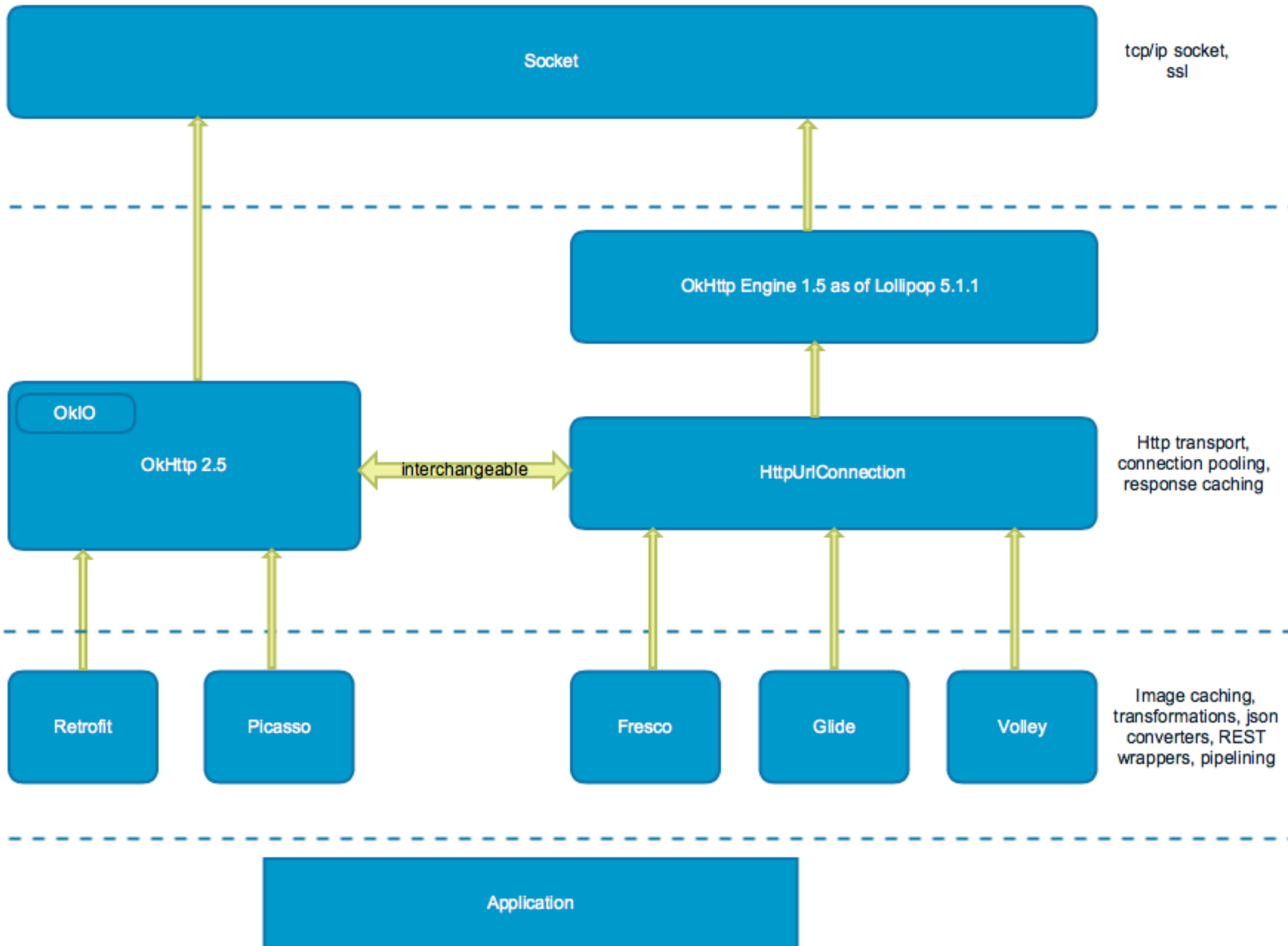
# A hálózat kezelés trükkös

- *Nem tetszik...*

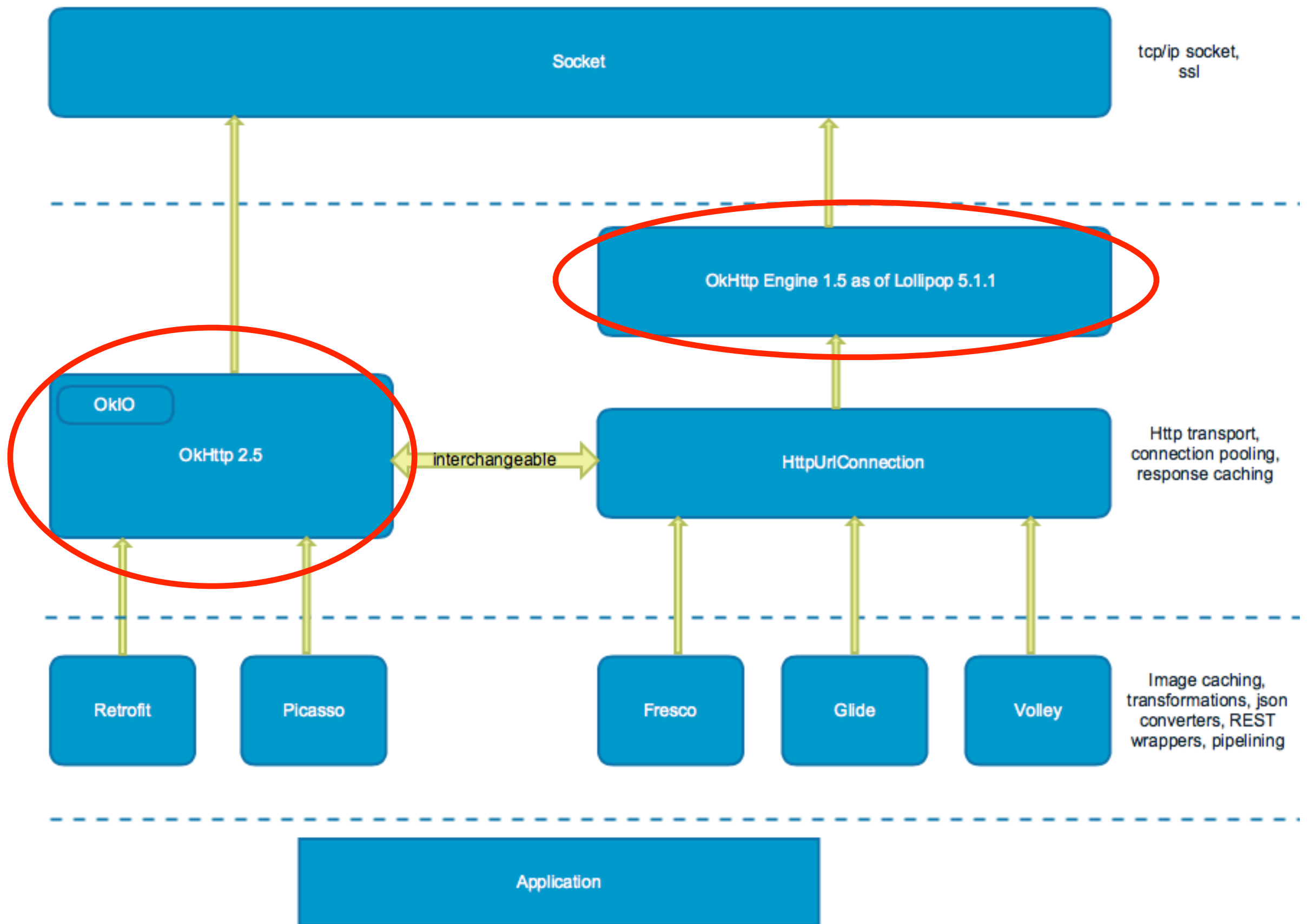
- Sok és kifejezetten imperatív kód.
- Piszok sok osztályt kell megjegyezni és használni.

- *Lehet ezt másképp is...*

- A Square féle **OkHttp**, **Retrofit** és **Picasso** hármas az esetek 99%-ban tökéletesen alkalmazható.
- Érett és sokat bizonyított könyvtárakról van szó.

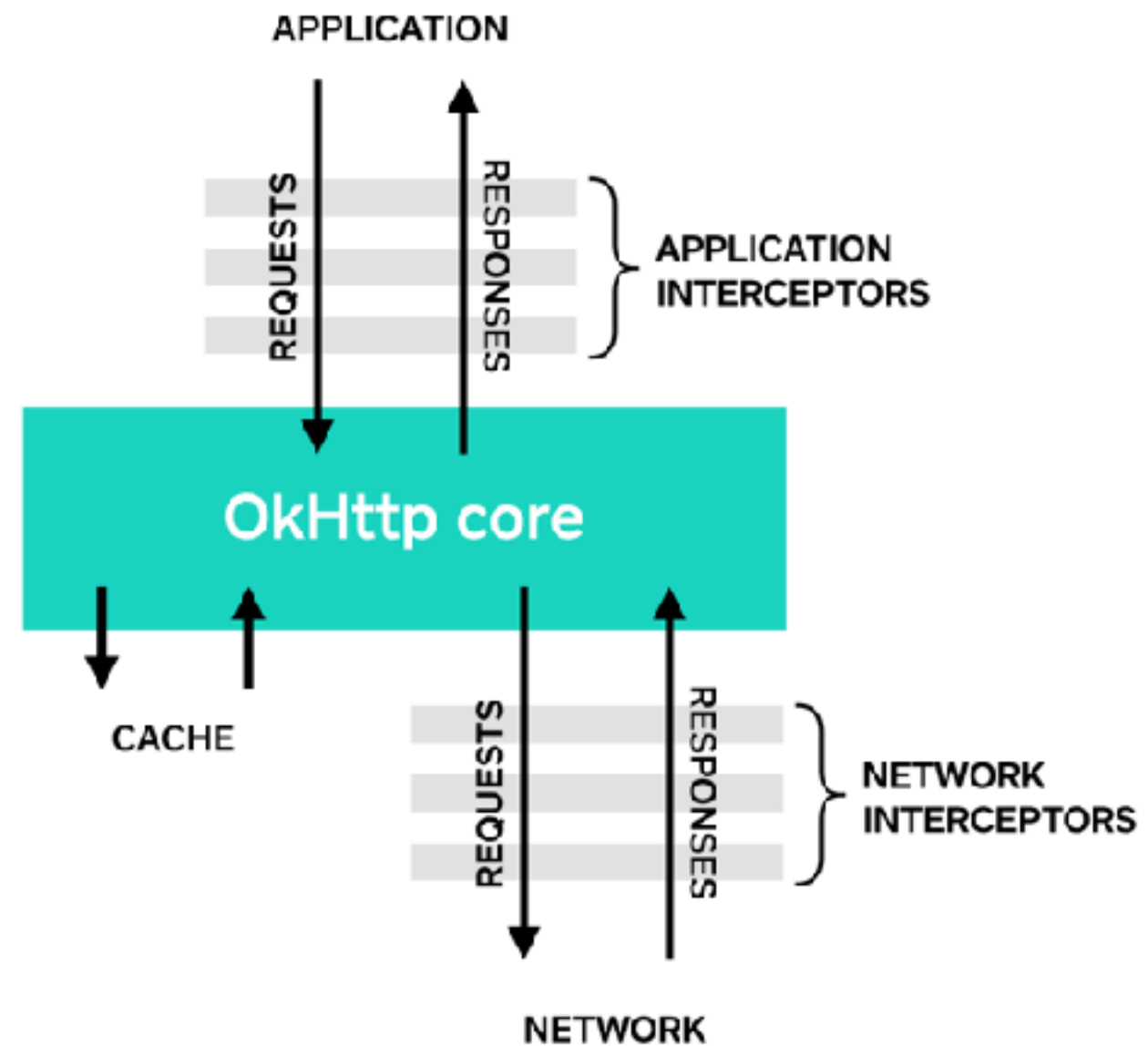






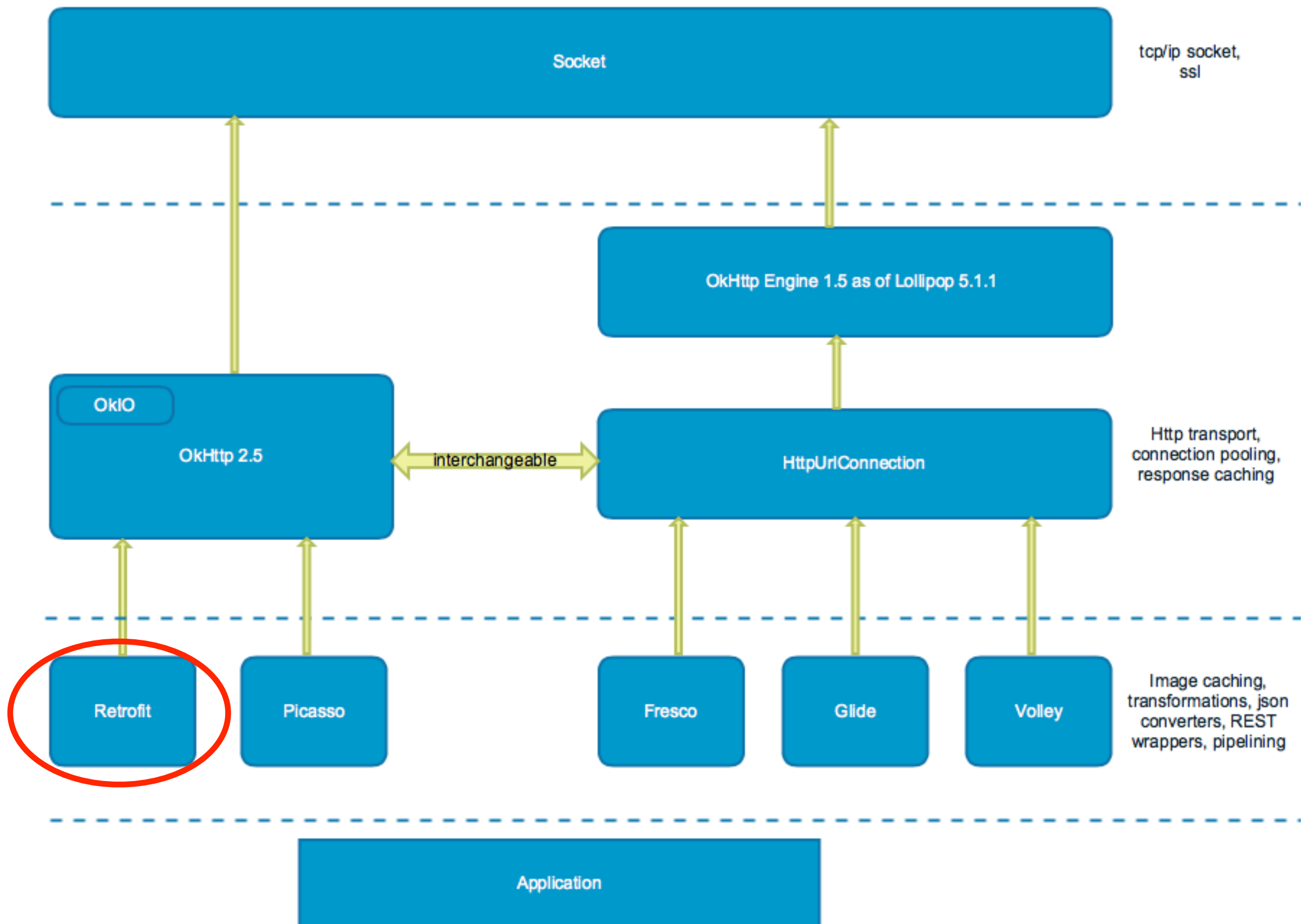
# OkHttp

- Alacsony szintű könyvtárról van szó, az Android alap **HTTP stack**ét cseréli ki modernebbre.
- Előnyei:
  - **HTTP/2,**
  - **GZip,**
  - **Cachelés,**
  - **IPv4 és IPv6** váltások,
  - Hálózati problémák kezelése.



# Pár sor a konfiguráció

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()  
    .writeTimeout(60L, TimeUnit.SECONDS)  
    .readTimeout(60L, TimeUnit.SECONDS)  
    .connectTimeout(60L, TimeUnit.SECONDS)  
    .build();
```



# Retrofit

Piszok egyszerű deklaratív interface-en keresztül lehet vele leírni a REST API-k kezelést.

```
interface BGGApiDefinition {  
    @GET("restaurants")  
    fun getRecommendations() : Call<List<RecommendationEntity>>  
  
    @POST("restaurants")  
    fun addRestaurant(@Body recommendation : RecommendationEntity) : Call<RecommendationEntity>  
}
```

# Egyszerű használni

```
Gson gson = new GsonBuilder().create();

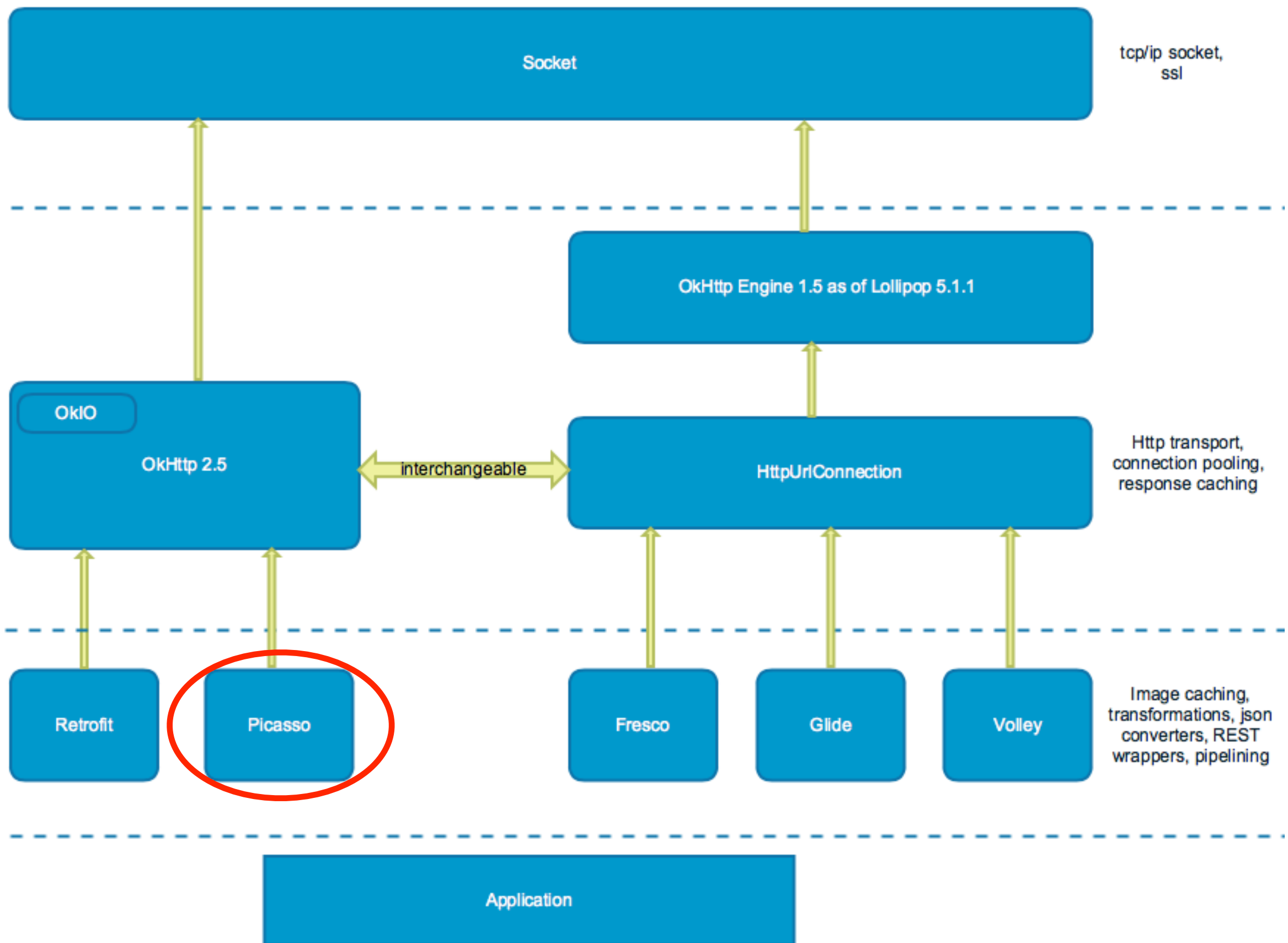
apiDefinition = new Retrofit.Builder()
    .addConverterFactory(GsonConverterFactory.create(gson))
    .client(okHttpClient)
    .baseUrl(BuildConfig.API_BASE_URL)
    .build()
    .create(BGGApiDefinition.class);
```

```
apiService.getRecommendations().enqueue(object: Callback<List<RecommendationEntity>> {
    override fun onFailure(call: Call<List<RecommendationEntity>>?, t: Throwable?) {
        Log.e(TAG.toString(), "Cannot fetch recommendations from network.", t)
    }

    override fun onResponse(call: Call<List<RecommendationEntity>>?,
        response: Response<List<RecommendationEntity>>?) {
        val items = response?.body()
        // ...
    }
})
```

[DEMO]

*A hálózati komponens  
refaktorálása Retrofittel*





# Picasso

**A képek kezelése nem egyszerű történet, számos use-case-t le kell tudni fedni:**

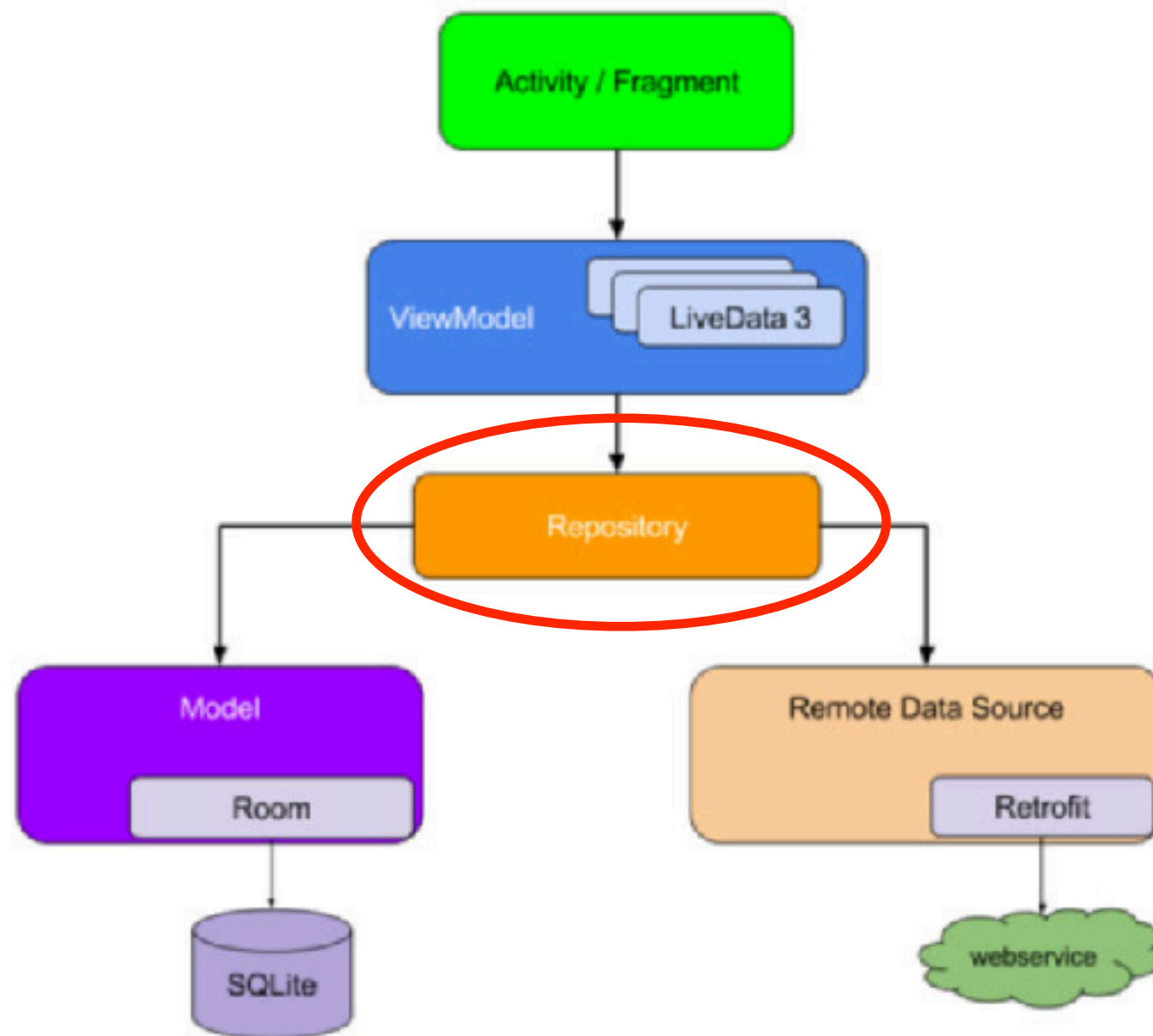
- *ImageView*-t újrafelhasználás.
- Letöltés megszakítás.
- Kép-transzformációk kevés memóriával.
- Cachelés memóriába és háttértárra.

```
Picasso.with(getContext())
    .load(recommendation.getImageURL())
    .placeholder(R.drawable.food)
    .error(R.drawable.food)
    .into(viewHolder.foodImageView, new Callback() {
        @Override
        public void onSuccess() {
        }

        @Override
        public void onError() {
            Log.e(TAG, "Error downloading image from " + recommendation.getImageURL());
        }
    });
```

[DEMO]

*A képkezelési komponens  
refaktorálása Picassoval*



# Meghízott az Activity

- ***Nem tetszik...***

- Kifejezetten testes osztályról van szó.
- Számos felelősség gyűlt össze egy osztályban.
  - UI komponensek kirajzolása
  - Felhasználói interakciók kezelése
  - UI mentése és visszaállítása konfiguráció változásnál
  - Adatok betöltése
  - Adatok feldolgozása

- ***Fogyókúrára kellene fogni...***

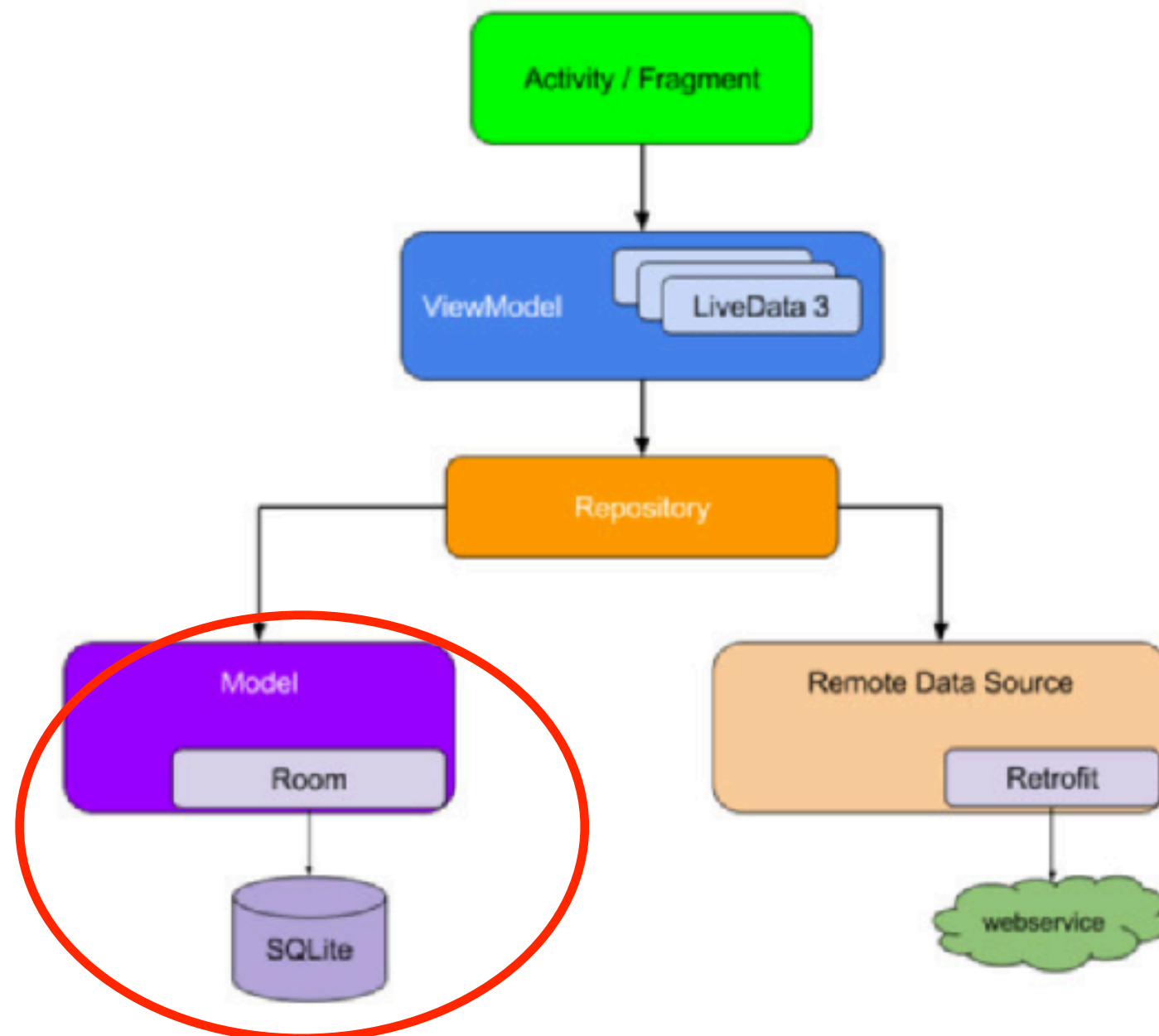
- Szeparáljuk jobban a kódot, mindent, ami az adatkezeléssel kapcsolatos szervezzük ki.
- Egyaránt beszélek itt a hálózat, illetve az adatbázis kódról.

# Repository

- Rengeteg felelősség összpontosul az *Activity*ben. Jobban akarjuk szeparálni a kód részeit.
- Kiszervezzük a teljes adatkezelést a **View**ból a **Repository**ba, legyen az *hálózattal* vagy *adatbázissal* kapcsolatos.
- *A kitűzött cél:*
  - Használja az adott időkereten belül lokálisan perzisztált adatokat.
  - Próbáljon meg frissebbet letölteni.

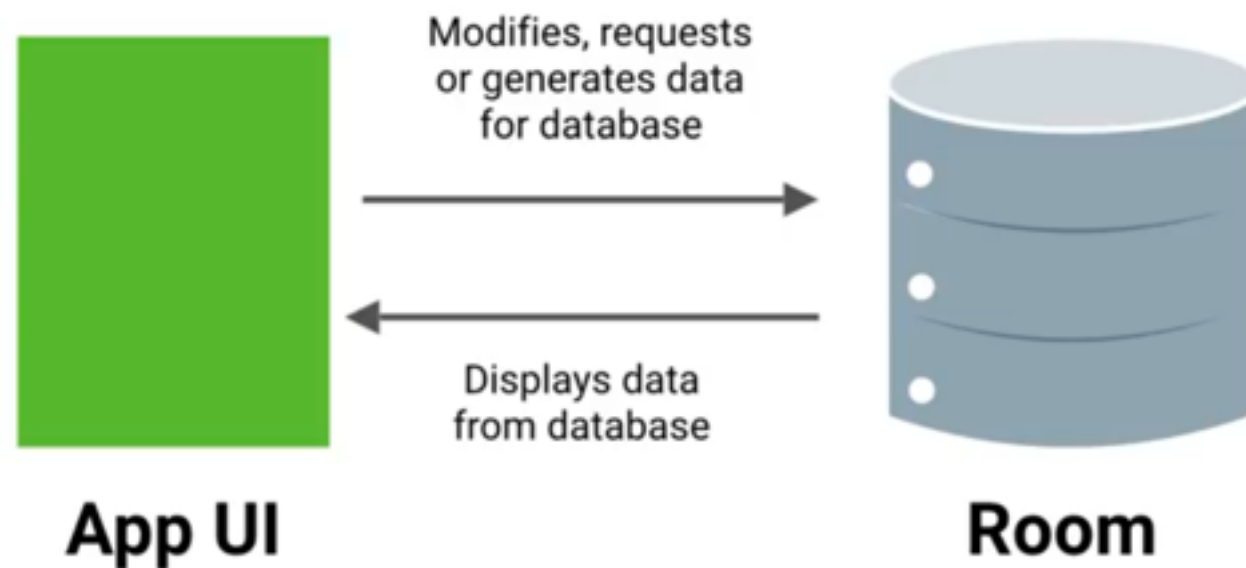
[DEMO]

*A Repository* komponens bevezetése



# Room

Retrofit egyszerűségű, deklaratív SQL-Java mappelő könyvtár.





# A adatbázis kezelés keményen kézimunkás

- *Nem tetszik...*
  - Sok és kifejezetten imperatív kód.
  - Piszok sok osztályt kell megjegyezni és használni.
  - String konkatenálásnál bukod az *IDE* segítségét.
- *Lehet ezt másképp is...*
  - Itt lesz a *Room* kiváló hasznunkra.
  - Még alig jött ki az 1.0, de eddig elég biztató.
  - Hasznos segítséget ad hozzá az *Android Studio*.

# *Ezeket memorizáld*

## POJO

```
@Entity(tableName = "users")
public class User {

    @PrimaryKey
    @ColumnInfo(name = "user_id")
    public String id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @ColumnInfo(name = "job_title")
    public String jobTitle;

    public int age;

}
```

## DAO

```
@Dao
public interface UserDao {

    @Insert(onConflict = IGNORE)
    void insertUser(User user);

    @Query("SELECT * FROM User")
    public List<User> findAllUsers();

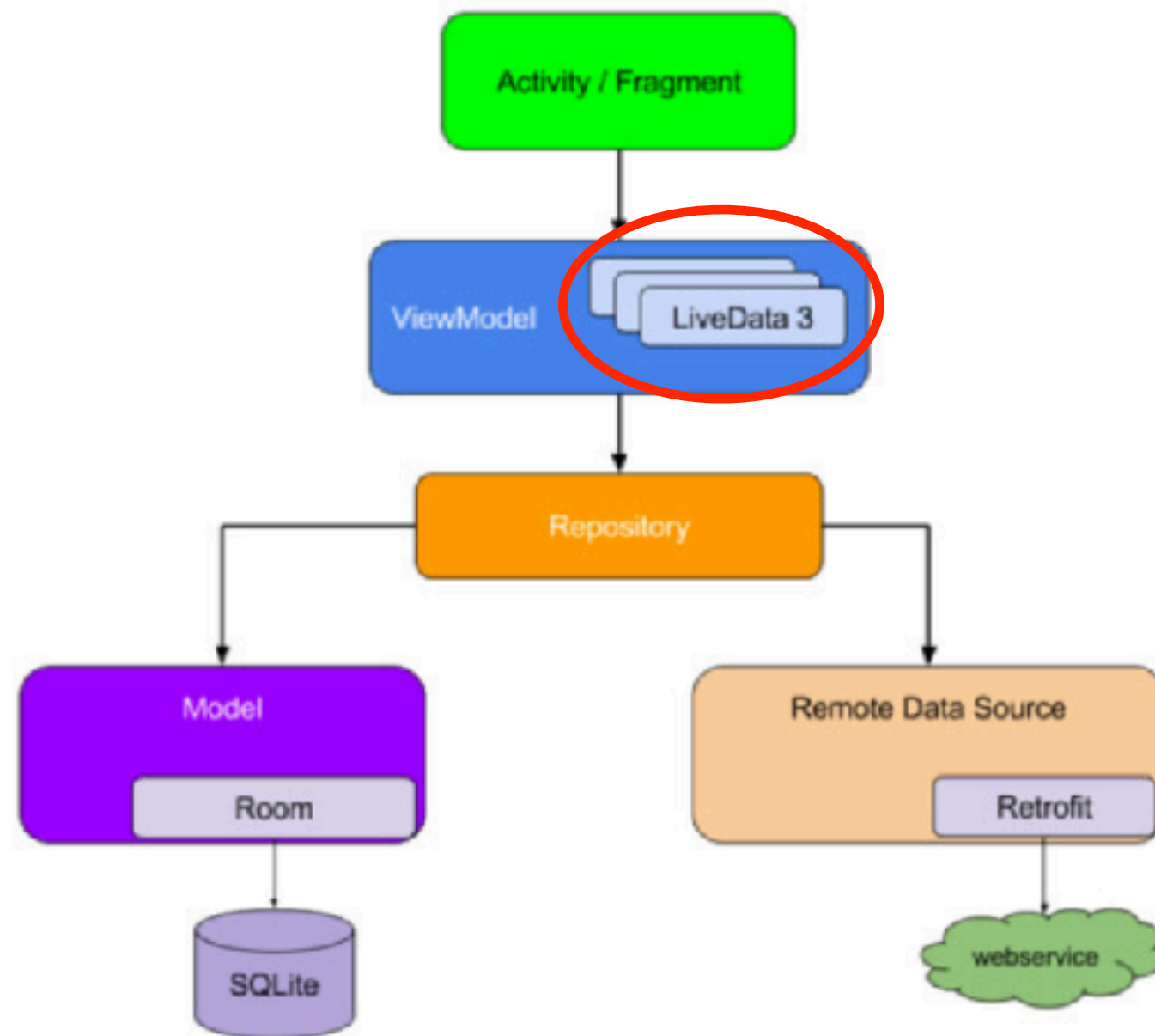
    @Update(onConflict = REPLACE)
    void updateUser(User user);

    @Query("DELETE FROM User")
    void deleteAllUsers();

}
```

[DEMO]

*Az adatbázis komponens  
refaktorálása Roommal*

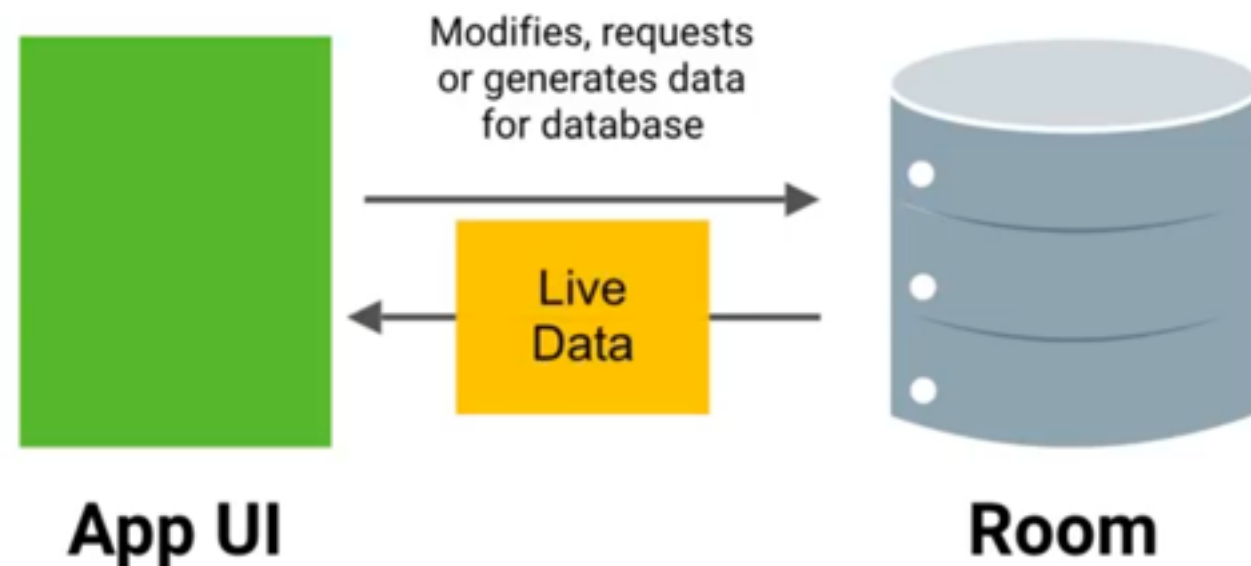


# Adatok szinkronizációja

- *Nem tetszik...*
  - Alapvetően nem vészes rész, de azért kódolás.
  - Egy data binding jellegű megoldás átláthatóbb.
- *Lehet ezt másképp is...*
  - Itt lesz a Live Data kiváló hasznunkra.
  - Még alig jött ki az 1.0, de eddig elég biztató.

# Mi a LiveData?

A **LiveData** tömören egy megfigyelhető (*observable*) adat tároló. Értesíti a megfigyelőket (*observers*), amikor valami adatváltozás történik. Így azok frissíthetik az alkalmazás *UI*-át.



# Ezeket memorizáld

- **Két új osztály:**
  - *LiveData*
  - *MutableLiveData*
- **Egy új metódus:**
  - *.observe()*

# Egyszerűen konfigurálható

```
MutableLiveData<String> dayOfWeek = new MutableLiveData<>();  
  
dayOfWeek.observe(this, data -> {  
    mTextView.setText(dayOfWeek.getValue() + " is a good day.");  
});
```





# Érték módosítása

```
dayOfWeek.setValue("Friday");
```



# Kiválóan együttműködik a *Room*

```
@Dao
public interface UserDao {

    @Insert(onConflict = IGNORE)
    void insertUser(User user);

    @Query("SELECT * FROM User")
    public LiveData<List<User>> findAllUsers();

    @Update(onConflict = REPLACE)
    void updateUser(User user);

    @Query("DELETE FROM User")
    void deleteAllUsers();

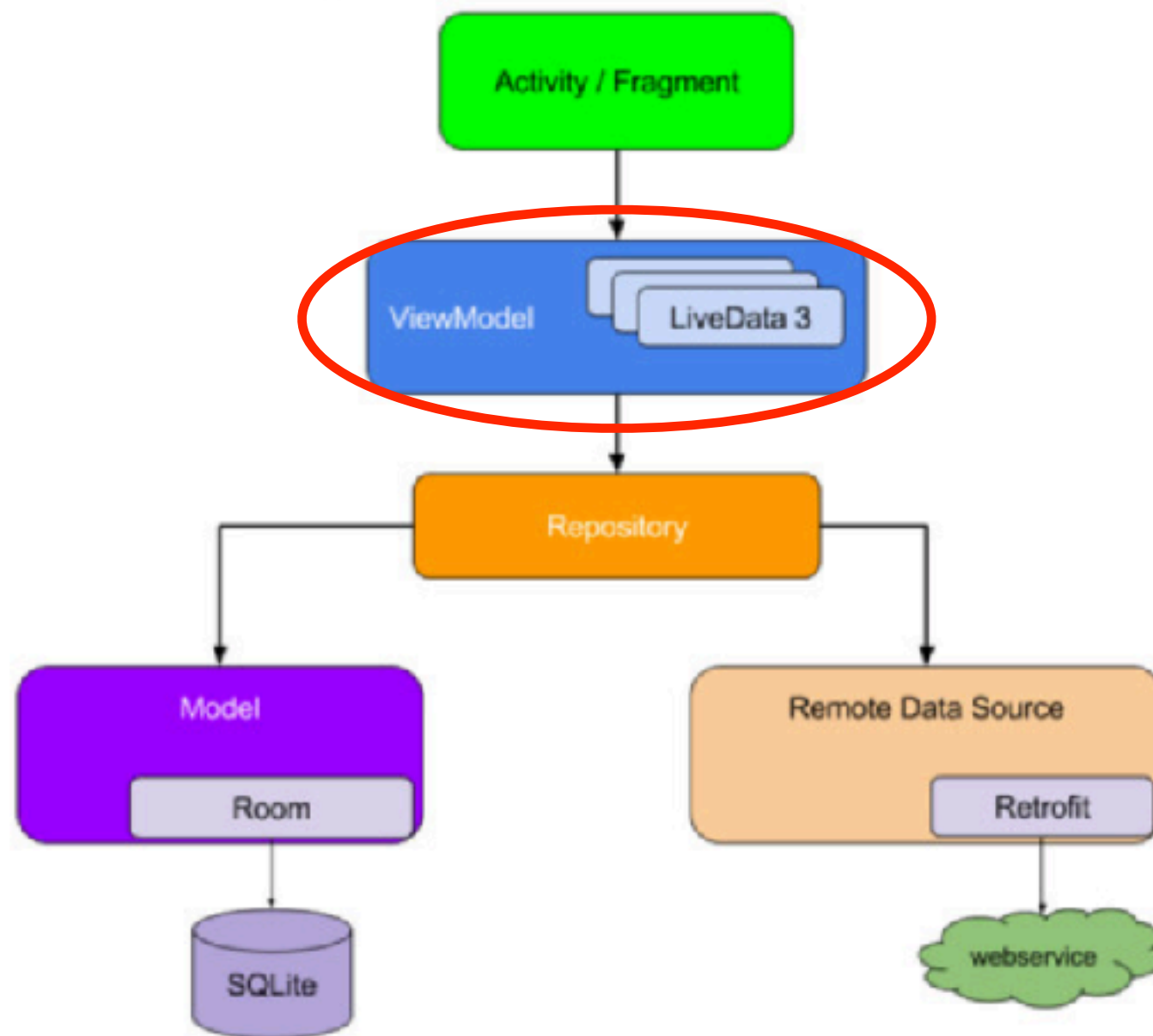
}
```

```
userLiveData.observe(this, users -> {
    mUserRecyclerViewAdapter.replaceItems(users);
    mUserRecyclerViewAdapter.notifyDataSetChanged();
});
```



**[DEMO]**

***Az *Live Data* komponens  
bevezetése***



# Meghízott az Activity

- ***Nem tetszik...***

- Azért már erősen fogyogat.
- Számos felelősség gyűlt össze egy osztályban.
  - UI komponensek kirajzolása
  - Felhasználói interakciók kezelése
  - *UI* mentése és visszaállítása konfiguráció változásnál
  - Adatok betöltése
  - Adatok feldolgozása

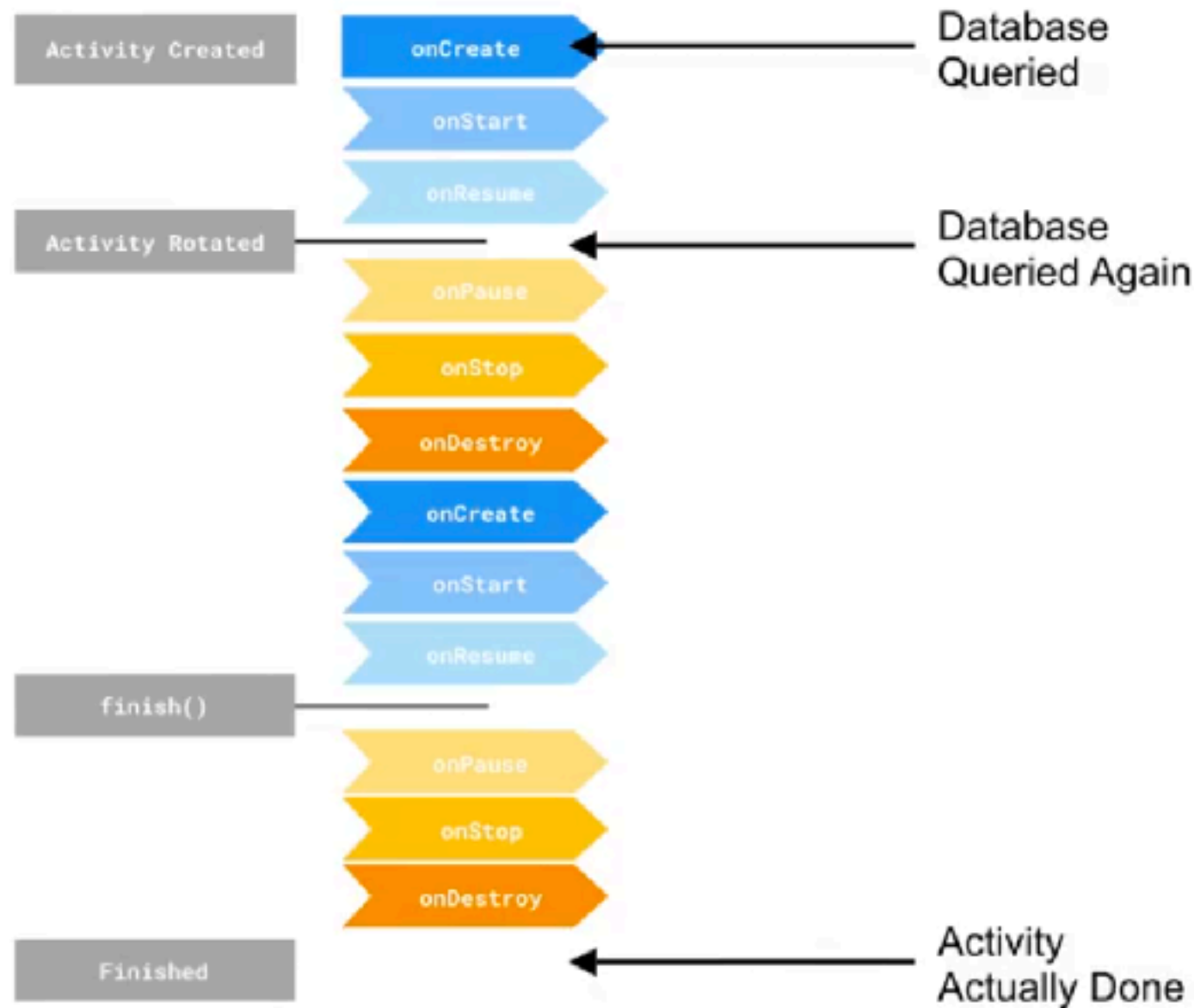
- ***Fogyókúrára kellene fogni...***

- Szeparáljuk még jobban a kódot.
- Mindent, ami üzleti logika, vagy adatkezelés, száműzzük az *Activity*ből.
- Küszöböljük ki az *Activity* életciklusból adódó problémákat és pazarlást.

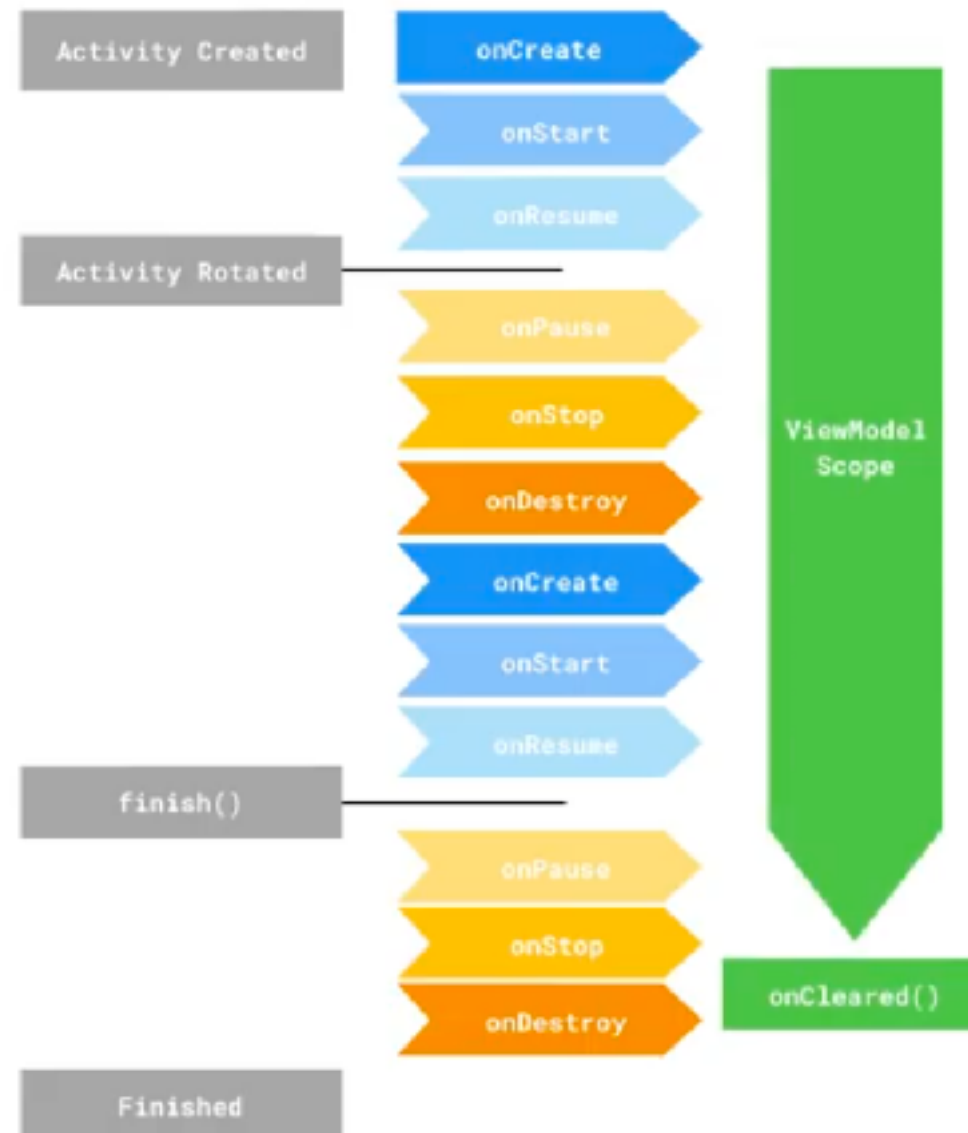
# Alap Activity lifecycle



# Pazarló újra hívni a queryket



# A ViewModel réteg elfedi a problémát





# Mi is az a ViewModel?

A **ViewModel** egy olyan objektum, amely adatot szolgáltat a *UI* komponenseknek, és túléli a konfiguráció változásokat.



# Ezeket memorizáld

- **Három új osztály:**
  - *ViewModel*,
  - *AndroidViewModel*,
  - *ViewModelProviders*.
- **Egy új metódus:**
  - *ViewModelProviders.of()*.

# Piszok egyszerű használni

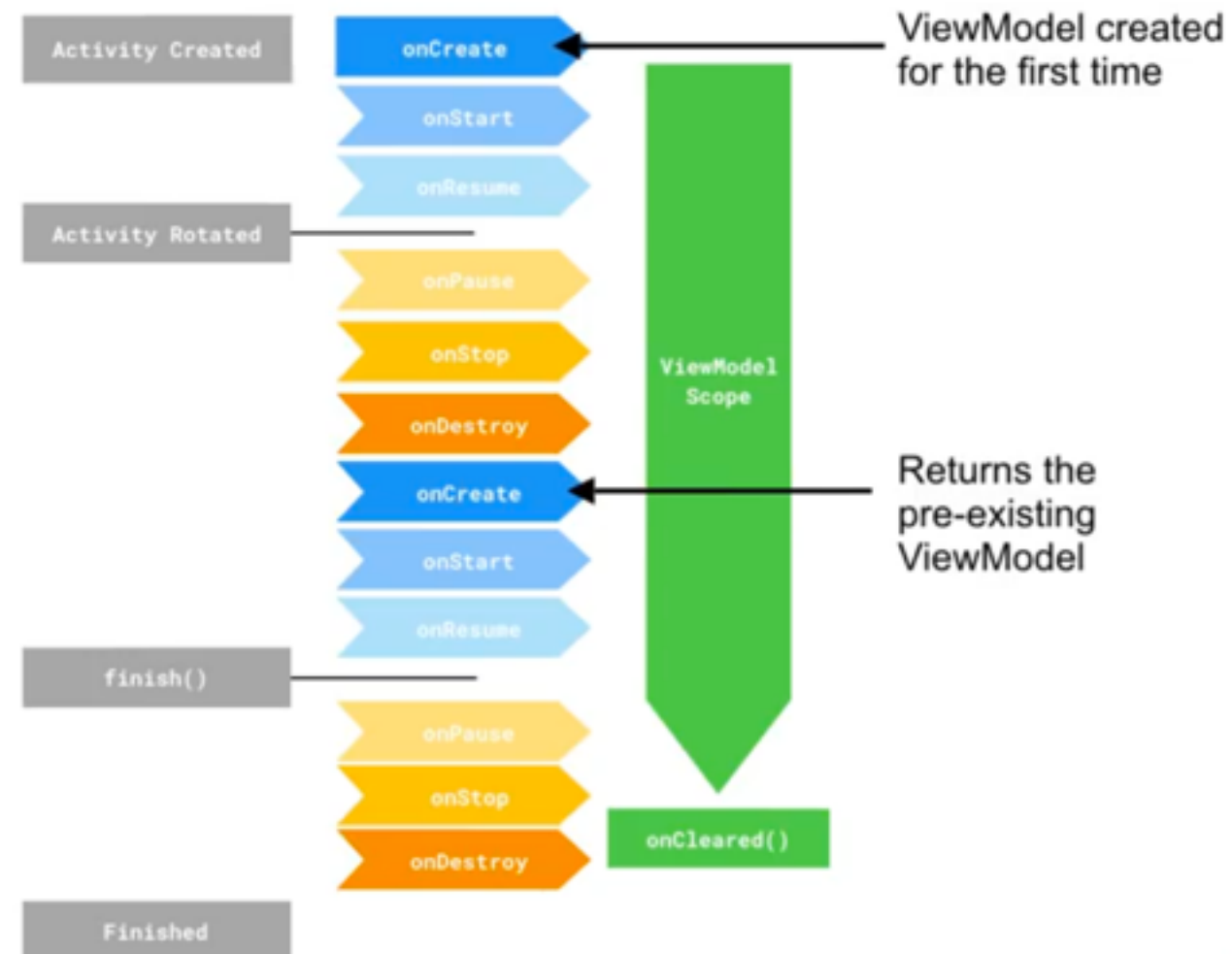
Új osztály

```
public class UserListViewModel extends AndroidViewModel {  
    private AppDatabase mDatabase;  
    private LiveData<List<User>> users;  
  
    public UserListViewModel(Application application) {  
        super(application);  
        mDatabase = AppDatabase.getDb(getApplication());  
        users = mDatabase.userModel().findAllUsers();  
    }  
  
    // Getters, setters ...  
}
```

\*Activity

```
userListViewModel = ViewModelProviders.of(this).get(UserListViewModel.class);  
  
userListViewModel.getUsers().observe(this, users -> {  
    mUserRecyclerViewAdapter.replaceItems(users);  
    mUserRecyclerViewAdapter.notifyDataSetChanged();  
});
```

# ViewModel életciklus

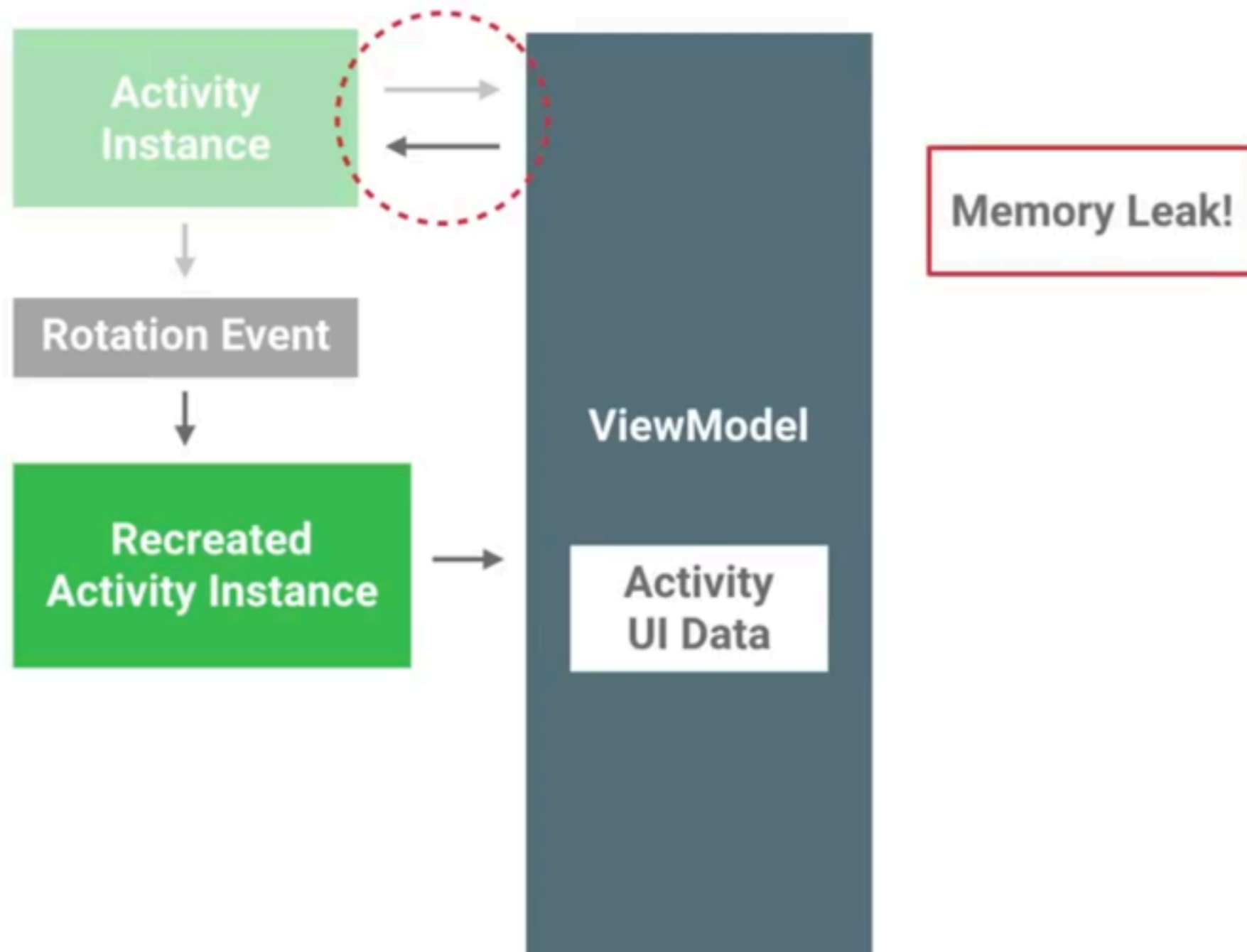


# Szeeparált felelősségek

**ViewModel**  
Hold UI Data

**Activity**  
Drawing UI  
Receiving User  
Interactions

**Repository**  
API for saving and  
loading app data



# Szükség van az onSaveInstancera

ViewModel stores...

All the user's data:  
User id, name, age, birthday,  
profile images, entire family  
history...

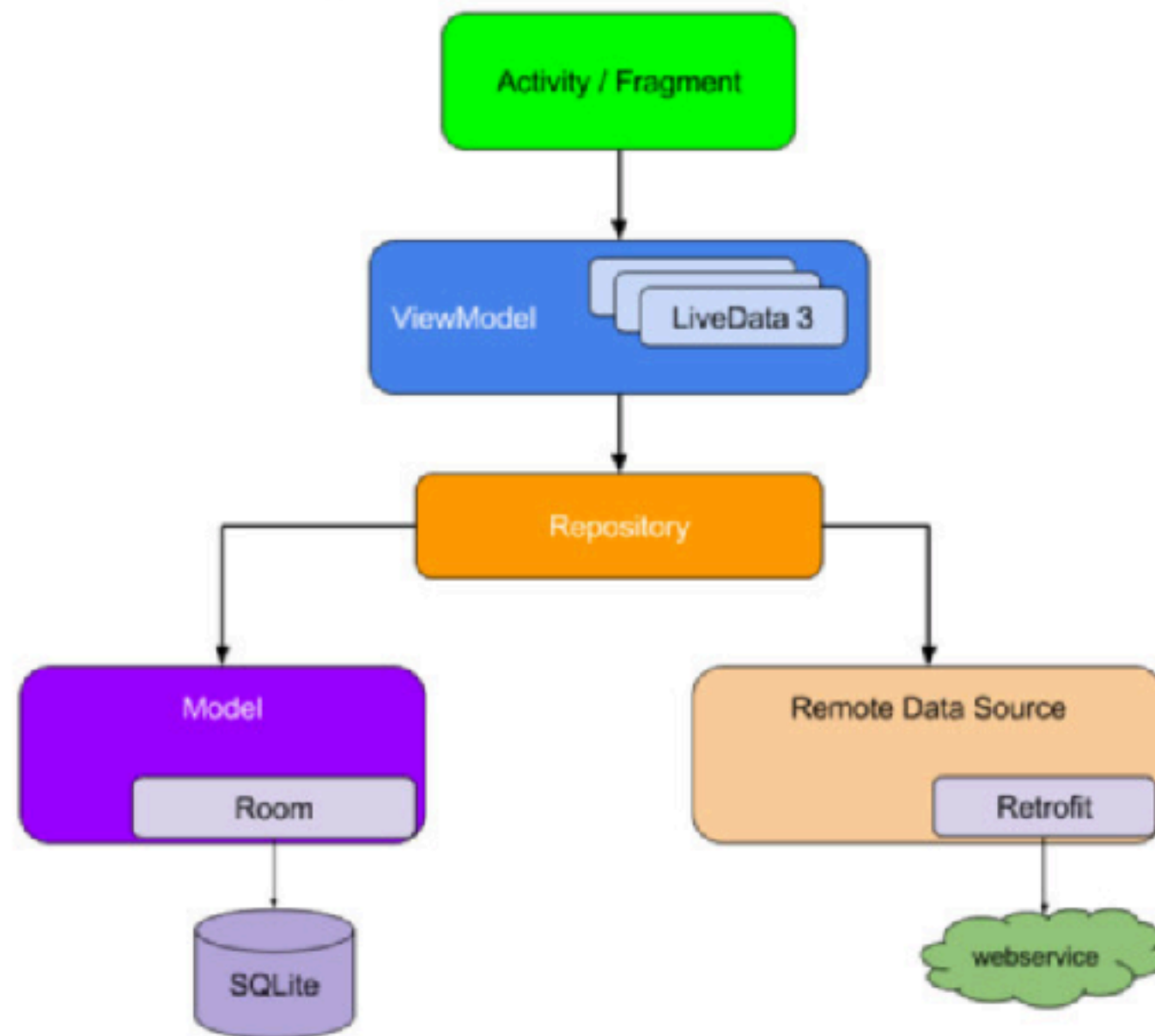
onSaveInstanceState() saves..

User id

**[DEMO]**

***Az *ViewModel* komponens  
bevezetése***





# Mi marad ki?

- Több képernyő és funkció bevezetése
  - Térkép kezelés
  - Új elem felvétele
  - Elem törlése
  - Login
- Komplex network kezelés
  - Login / Firebase Authentication
  - Agresszív cacheing
  - Teljes CRUD lefedése
- Data binding
- Kódbázis egyszerűsítés Ankoval
- Több interface bevezetése és DI (Dagger 2)
- GPS adatok és a LiveData
- Navigáció komponens bevezetése
- Engedélyek (permission) kezelés
- UI kód refaktorálása
  - ListView helyett RecyclerView
  - CoordinatorLayout és FlexBox layout, haladóbb szinten
  - Komponensre bontás
  - Animációk
- Room adatbázis migrálás
- MutableLiveData
- Hatékony loggolási technikák
- Logika- és UI tesztelése
- Firebase (Crashlytics & Analytics)

**Q&A**

**Köszönjük a  
figyelmet!**