ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM

KHOA CÔNG NGHỆ THÔNG TIN

KIẾN TRÚC PHẦN MỀM

Đề tài: Đồ án Kiến Trúc Phần Mềm

PHẦN MỀM HỌC TIẾNG ANH QUA VIDEO CHAT

Bộ môn Công nghệ phần mềm

Khoa Công nghệ thông tin

Đại học Khoa học tự nhiên TP HCM

MỤC LỤC

1 Thông tin cá nhân 3
2 Thang điểm tự đánh giá 3
3 Thông tin dự án 3
4 Công nghệ sử dụng 4
a. Client side 4
b. Server side 4
5 Kỹ thuật và kiến trúc được dùng trong phần mềm 4
a. Software architecture pattern: 4
b. Programming paradigms 11
c. Design Pattern 13
a. Creational pattern: 13
ii. Builder 17
iii. Factory method: 20
iv. Singleton 21
vii. Abstract class 22
b. Structural pattern 22
i. Adapter: 22
ii. ViewHolder 23
iii. Façade 24
iv. Repository 26
c. Behavioral pattern 29
i. Observer 29
ii. Strategy 31

1 Thông tin cá nhân

MSSV	Họ Tên	Email	SDT
1412197	Đoàn Thị Phương Huyền	dtphuyen2506@gmail.com	0969938215
1412363	Trần Thị Nhã	tranthinha160296@gmail.com	
1412477	Đoàn Hiếu Tâm	nhoxbypass@gmail.com	01684934109

2 Thang điểm tự đánh giá

Đánh giá dựa trên tiêu chí tỉ lệ số kĩ thuật áp dụng thành công/số kĩ thuật có thể áp dụng được cho Android. (vì một số kĩ thuật không thể hoặc không phù hợp với Android).

- Software architecture pattern (MVVM): 10.
- Programming paradigm: 10.
- Design pattern: Creational (9.5), Structural (9.75), Behavior (9.75).
- Middleware (8)

Tổng kết: 9.75.

3 Thông tin dự án

English Now là một ứng dụng Android nhằm giúp đỡ mọi người học tiếng Anh thông qua các cuộc đối thoại bằng video chat, viết essay và chat text. Qua đó các kĩ năng speaking, writing và listening sẽ được cải thiện.

Link github: https://github.com/HCMUS-AssignmentWarehouse/EnglishNow-Android-MVVM/

Đồ án được xây dựng phục vụ riêng cho môn Kiến trúc phần mềm.

Dựa trên phần mềm English Now phiên bản iOS mà nhóm đã xây dựng ở học kì trước (trong môn phát triển ứng dụng mobile). Link github của phiên bản iOS kèm demo để hiểu rõ hơn phiên bản Android (Android chưa kịp làm demo):

https://github.com/HCMUS-AssignmentWarehouse/EnglishNow/ . Link video walkthrough: https://www.youtube.com/watch?v=Sw4Gj1eF8is/

4 Công nghệ sử dụng

Môi trường phát triển: **Android Studio 3.0** và **Ubuntu 16.04 LTS**.

a. Client side

English Now sử dụng:

- **Android** framework với minSdk 21 và buildtools version 27.0.1 để xây dựng.
- **OpenTok client sdk** để xây dựng client side cho webRTC
- **Firebase** để xây dựng hệ thống authentication và realtime database.
- **Dagger 2** để ứng dụng dependency injection.
- **RxJava, RxAndroid** để ứng dụng observer pattern và reactive programming.
- **Android Acrchitecture Component** của Google để áp dụng MVVM và LiveData.
- **Retrofit 2** cho HTTP client.
- **EventBus** cho việc giao tiếp giữa các thành phần trong hệ thống bằng message.
- Và một số thư viện UI khác.

Để biết rõ hơn, tham khảo readme của projet tại đây.

b. Server side

Sử dụng **nodeJS**, **expressJS** và **OpenTok server sdk** để tạo một server side đơn giản và deploy lên herokuapp (https://englishnow.herokuapp.com/).

5 Kỹ thuật và kiến trúc được dùng trong phần mềm

Tham khảo danh sách thu gọn các kỹ thuật/kiến trúc đã áp dụng hoặc chưa áp dụng được trong list requirement sau.

a. Software architecture pattern:

Software Architectural pattern:

- [ ] Model-View-Controller (MVC)
- [ ] Model-View-Presenter (MVP)
- [x] Model-View-ViewModel (MVVM)

Vấn đề: Project mang tính chất realtime, cần databinding và cập nhật view theo trạng thái của data liên tục

Giải pháp: Nhóm chọn sử dụng mô hình **Model-View-ViewModel (MVVM)** trong đó:

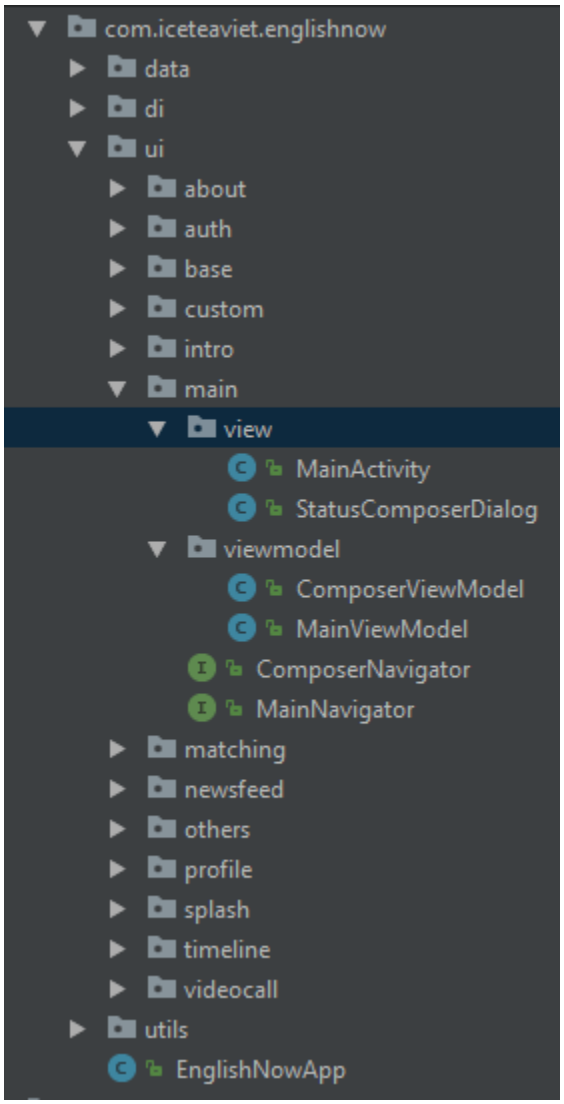
- **View**: đại diện cho giao diện, dùng để hiển thị trạng thái lên màn hình và thông báo cho ViewModel khi có tương tác từ user.
- **ViewModel**: xử lý các dòng dữ liệu, xử lý logic và và chuyển lại cho View.
- **Model**: hay còn gọi là DataModel, là các abstract data source. Được ViewModel sử dụng trong việc lấy và lưu data.

MVVM tách rời trạng thái của View ra khỏi các hành vi của nó, chú trọng vào **event driven** programming. View sẽ có thể có reference tới ViewModel, nhưng ViewModel sẽ **không quan tâm** cũng như **không biết** gì về trạng thái của View, chỉ nhận yêu cầu, lấy data, xử lý và trả kết quả.

MVVM ứng dụng trong project theo hướng **two-way data binding** giữa View và ViewModel với sự hỗ trợ của Android Data Binding bằng cách bật trong **app/build.gradle**

```
dataBinding {
    enabled = true
}
```

Project có cấu trúc như sau:



Nhóm phân chia các package của project theo feature mà không theo layer vì khi ta cần sửa một tính năng gì, ta chỉ cần sửa gọn trong package của nó. Và vì project áp dụng abstract class/ interface rất triệt để cho các class của từng layer nên nếu ta cần thêm code cho tất cả các class của một layer nào đó chỉ cần thêm vào base class của layer đó.

Ta sẽ đến với phần **View trong MVVM**: Trong Android, phần View thường là Activity/Fragment. Tất cả các đoạn code trong View layer không được phép chứa logic hay các thao tác với dữ liệu, vì bản chất của MVVM là tách rời logic và các dòng dữ liệu ra khỏi View để tiện xử lý.

Vì vậy mọi xử lý phải thông qua ViewModel

```java
public class MainActivity extends BaseActivity<ActivityMainBinding, MainViewModel> implements MainNavigator

    @Inject
    protected @MainViewModelProviderFactory
    ViewModelProvider.Factory viewModelFactory;

    @Inject
    protected DispatchingAndroidInjector<Fragment> fragmentDispatchingAndroidInjector;

    protected ActivityMainBinding activityMainBinding;
    private MainViewModel mainViewModel;

    private DrawerLayout drawerLayout;
    private Toolbar toolbar;
    private NavigationView navigationView;
    private ActionBarDrawerToggle drawerToggle;

    private int currentMenuItemId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    @Override
    protected void onResume() {...}

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {...}

    @Override
```

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {...}

@Override
public void onFragmentDetached(String tag) {...}


@Override
public void onBackPressed() {...}

@Override
public MainViewModel getViewModel() {...}

@Override
public int getBindingVariable() { return BR.viewModel; }

@Override
public int getLayoutId() { return R.layout.activity_main; }

@Override
public void onConfigurationChanged(Configuration newConfig) {...}

private void setup() {...}

private void setupDrawerLayout() {...}


private void setupNavigationHeader() {...}

private boolean onNavigationItemSelected(MenuItem menuItem) {...}
```

```
private void showNewsFeedFragment() {...}

private void showConversationMatchingActivity() {...}


private void showProfileFragment() {...}

private void showAboutFragment() {...}

@Override
public void navigateToLoginScreen() {...}

@Override
public void handleError(Throwable throwable) {...}

@Override
public AndroidInjector<Fragment> fragmentInjector() {...}

private void lockDrawer() {...}

private void unlockDrawer() {...}

/**
 * Swaps fragments in the main content view
 */
private void selectFragment(Fragment fragment, String tag) {...}

private void addFragment(Fragment fragment, String tag) {...}
```

Tiếp theo ta sẽ đến với phần **ViewModel trong MVVM**: ViewModel đảm nhiệm các phần tính toán logic, và xử lý các dòng dữ liệu

```java
public class MainViewModel extends BaseViewModel<MainNavigator> {

    private final ObservableField<String> appVersion = new ObservableField<>( value: "1.0");
    private final ObservableField<String> userName = new ObservableField<>();
    private final ObservableField<String> userEmail = new ObservableField<>();
    private final ObservableField<String> userProfilePicUrl = new ObservableField<>( value: "");
    private final ObservableBoolean isShowNavUsername = new ObservableBoolean( value: false);
    private final ObservableBoolean isShowNavEmail = new ObservableBoolean( value: false);


    public MainViewModel(DataManager dataManager, SchedulerProvider schedulerProvider) {
        super(dataManager, schedulerProvider);
    }

    public void updateAppVersion(String version) { appVersion.set(version); }

    public void onNavigationViewCreated() {
        final String currentUserName = getDataManager().getCurrentUserDisplayName();
        if (currentUserName != null && !currentUserName.isEmpty()) {
            userName.set(currentUserName);
            isShowNavUsername.set(true);
        } else {
            isShowNavUsername.set(false);
        }

        final String currentUserEmail = getDataManager().getCurrentUserEmail();
        if (currentUserEmail != null && !currentUserEmail.isEmpty()) {
            userEmail.set(currentUserEmail);
            isShowNavEmail.set(true);
```

```java
            isShowNavEmail.set(true);
        } else {
            isShowNavEmail.set(false);
        }

        final String profilePicUrl = getDataManager().getCurrentUserPhotoUrl();
        if (profilePicUrl != null && !profilePicUrl.isEmpty()) {
            userProfilePicUrl.set(profilePicUrl);
        }
    }


    public void logout() {
        setIsLoading(true);
        getDataManager().logoutFirebase();
        setIsLoading(false);
        getNavigator().navigateToLoginScreen();
    }

    public ObservableField<String> getAppVersion() { return appVersion; }

    public ObservableField<String> getUserName() { return userName; }

    public ObservableField<String> getUserEmail() { return userEmail; }

    public ObservableField<String> getUserProfilePicUrl() { return userProfilePicUrl; }

    public ObservableBoolean getIsShowNavEmail() { return isShowNavEmail; }

    public ObservableBoolean getIsShowNavUsername() { return isShowNavUsername; }
}
```

ViewModel giao tiếp với View thông qua một interface **Navigator** được View implement các method:

```
public interface MainNavigator {
    void navigateToLoginScreen();

    void handleError(Throwable throwable);
}
```

Cuối cùng là **Model trong MVVM**: Model chịu trách nhiệm làm các "vật chứa" để ViewModel thao tác với database (read/write/remove).

```
@IgnoreExtraProperties
public class VideoCallSession implements Serializable {
    @PropertyName("learner_uid")
    private String learnerUid;

    @PropertyName("speaker_uid")
    private String speakerUid;

    @PropertyName("learner_token")
    private String learnerToken;

    @PropertyName("speaker_token")
    private String speakerToken;

    @PropertyName("session_id")
    private String sessionId;

    public VideoCallSession() {...}

    public String getLearnerUid() { return learnerUid; }

    public void setLearnerUid(String learnerUid) { this.learnerUid = learnerUid; }

    public String getSpeakerUid() { return speakerUid; }

    public void setSpeakerUid(String speakerUid) { this.speakerUid = speakerUid; }

    public String getLearnerToken() { return learnerToken; }

    public void setLearnerToken(String learnerToken) { this.learnerToken = learnerToken; }
```

Để sử dụng Model, ViewModel cần một **Façade** tổng hợp các thao tác gọi là **AppDataManager**, một concrete class của **interface DataManager**

```java
public interface DataManager extends ApiHelper, FirebaseHelper, PreferencesHelper {
    UserDataSource getUserRepository();

    NewsFeedItemDataSource getNewsFeedItemRepository();

    MediaDataSource getMediaRepository();

    VideoCallSessionDataSource getVideoCallSessionRepository();

    Observable<List<StatusItemData>> fetchAllStatusItemData();
}
```

```java
@Singleton
public class AppDataManager implements DataManager {
    private final Context context;
    private final ApiHelper apiHelper;
    private final FirebaseHelper firebaseHelper;
    private final PreferencesHelper preferencesHelper;
    private final UserDataSource userRepository;
    private final NewsFeedItemDataSource newsFeedItemRepository;
    private final MediaDataSource mediaRepository;
    private final VideoCallSessionDataSource videoCallSessionRepository;

    @Inject
    public AppDataManager(Context context, FirebaseHelper firebaseHelper, PreferencesHelper preferenc
                          UserDataSource userRepository, NewsFeedItemDataSource newsFeedItemRepositor
                          VideoCallSessionDataSource videoCallSessionRepository) {...}

    @Override
    public NewsFeedItemDataSource getNewsFeedItemRepository() { return newsFeedItemRepository; }

    @Override
    public UserDataSource getUserRepository() { return userRepository; }

    @Override
    public MediaDataSource getMediaRepository() { return mediaRepository; }

    @Override
    public VideoCallSessionDataSource getVideoCallSessionRepository() {
        return videoCallSessionRepository;
    }
}
```

Lý do tại sao không tạo class trực tiếp mà sử dụng Interface rồi implement lại là vì cần thiết cho việc thiết kế các testcase và sẽ được giải thích trong phần sau.

b. Programming paradigms

Programming paradigms

- [x] Reactive programming: Programming with asynchronous data streams. In computing, reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change.

Vấn đề: project là phần mềm tương tác online 90%, **dữ liệu và trạng thái của các thành phần trong app luôn thay đổi liên tục**.

Giải pháp: nhóm quyết đỉnh áp dụng **Reactive Programming** để tương tác với **các dòng dữ liệu bất đồng bộ** giữa app và server.

Ví dụ: Trong khi thực hiện việc matching giữa hai user để tạo room cho video chat. ViewModel cần biết khi nào việc matching hoàn thành để lấy thông tin ra và chuyển sang màn hình khác, ta sử dụng PublishSubscriber để tạo ra datastream liên kết giữa ViewModel và các data handler.

```
private PublishSubject<OpenTokRoom> matchingSubject;
```

ViewModel subscribe vào datastream này và chờ sợ thay đổi của trạng thái và dữ liệu.

```
handler = new ConversationMatchingHandler(dataManager, schedulerProvider, getCompositeDisposable(),

//Subscribe to data-stream to know when the matching complete and get the data
getCompositeDisposable().add(handler.getMatchingSubject()
        .subscribeOn(schedulerProvider.io())
        .observeOn(schedulerProvider.ui())
        .subscribe(room -> {
            //Finish -> start video call activity
            getNavigator().navigateToVideoCallScreen(room.getSessionId(), room.getToken());
        }, throwable -> getNavigator().handleError(throwable)));
```

Khi dữ liệu thay đổi, ta thêm data vào data-stream và trigger event cho các subscriber biết mà cập nhật dữ liệu.

```
if (session.getSessionId() != null && !session.getSessionId().isEmpty()
        && session.getSpeakerToken() != null && !session.getSpeakerToken().isEmpty()
        && session.getLearnerToken() != null && !session.getLearnerToken().isEmpty()) {
    //Successfully, add data to stream and send onNext event back to ViewModel
    matchingSubject.onNext(currentRoom);
    matchingSubject.onComplete();
} else {
    //Continue waiting
}
```

c. Design Pattern

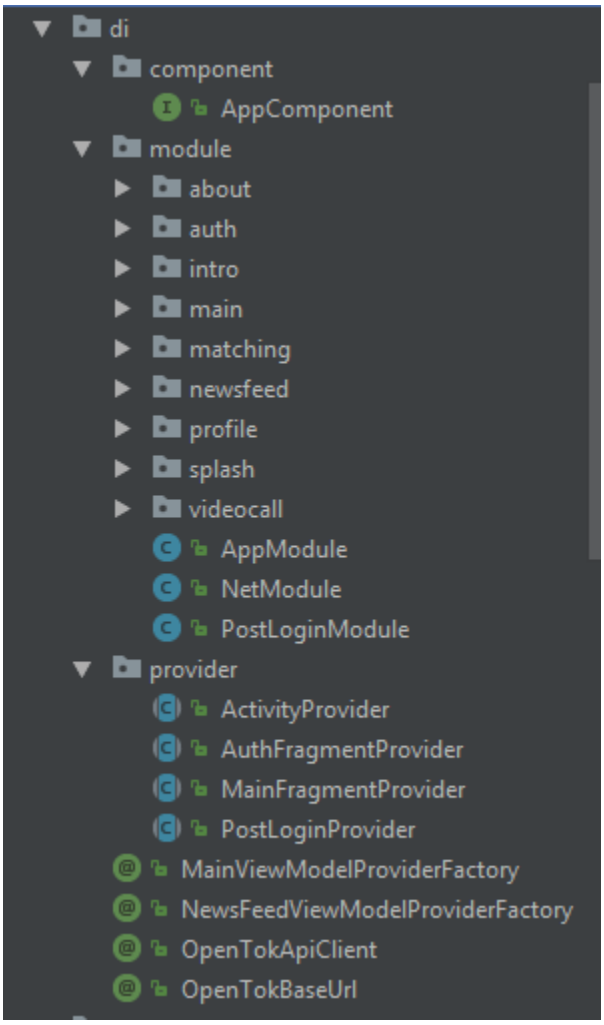
a. Creational pattern:

Creational:

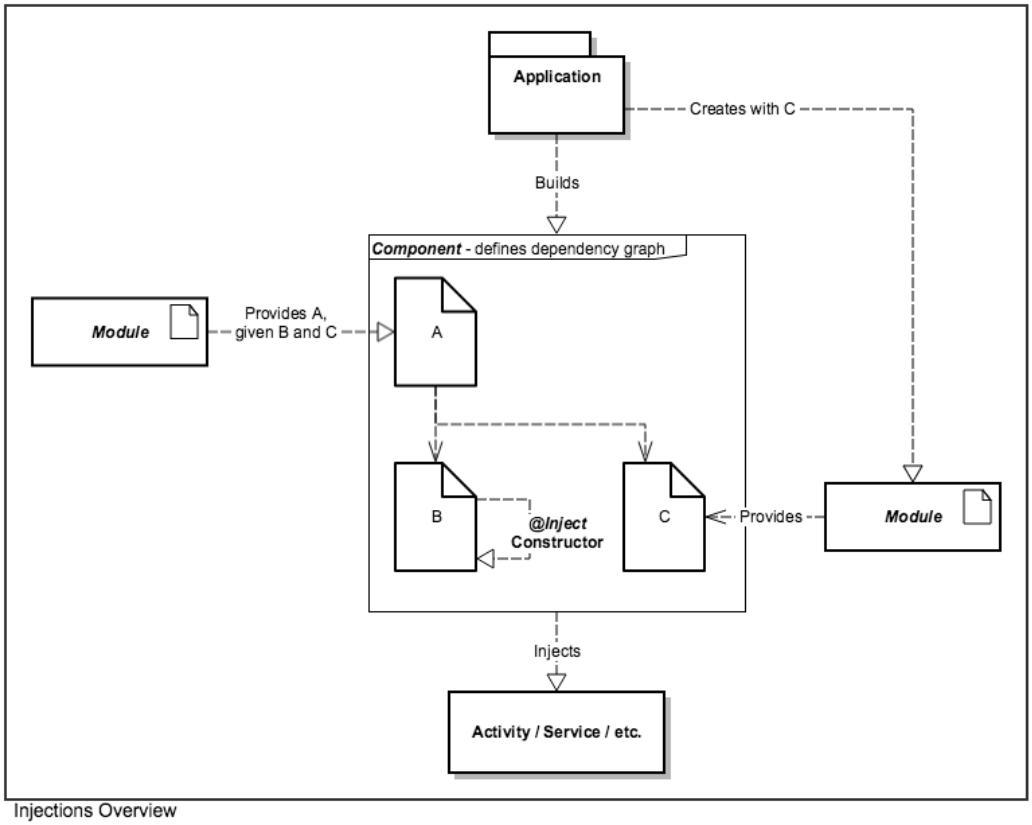
- [x] Dependency Injection (DI): Is a design pattern that demonstrates how to create loosely coupled classes.
- [x] Builder: constructs complex objects by separating construction and representation.
- [ ] Abstract factory: groups object factories that have a common theme.
- [x] Factory method: creates objects without specifying the exact class to create.
- [ ] Prototype: creates objects by cloning an existing object.
- [x] Singleton: restricts object creation for a class to only one instance.
- [ ] Create new object by instance:
- [x] Abstract class.

i. **Dependency Injection (DI)**: Vì project có rất nhiều các thành phần được dùng chung và có thể tái sử dụng nên nhóm quyết định sử dụng Dagger 2 để apply dependency injection để tạo các object đó trước ở ngoài và inject vào các chỗ cần thiết.

Project package structure được sắp xếp như sau:



Mô hình thường thấy nhất khi áp dụng DI trong Android là:



Injections Overview

Ở đây ta thấy có ba thành phần chính: **Module**, **Component** và **những nơi cần sử dụng dependencies**.

Ta đi vào tìm hiểu **@Module**: Là nơi cung cấp các dependency cần thiết cho Component. Ví dụ **NetModule.java, AppModule.java**:

```java
@Module
public class NetModule {
    // Dagger will only look for methods annotated with @Provides
    @Provides
    @OpenTokBaseUrl
    String provideOpenTokBaseUrl() { return ApiEndPoint.OPENTOK_BASE_URL; }

    @Provides
    @Singleton
    Cache provideOkHttpCache(Application application) {
        int cacheSize = 10 * 1024 * 1024; // 10 MiB
        Cache cache = new Cache(application.getCacheDir(), cacheSize);
        return cache;
    }

    @Provides
    @Singleton
    Gson provideGson() {
        GsonBuilder gsonBuilder = new GsonBuilder();
        gsonBuilder.setFieldNamingPolicy(FieldNamingPolicy.LOWER_CASE_WITH_UNDERSCORES);
        return gsonBuilder.create();
    }

    @Provides
    @Singleton
    OkHttpClient provideOkHttpClient(Cache cache) {
        OkHttpClient.Builder client = new OkHttpClient.Builder();
        client.cache(cache);
        return client.build();
    }
```

Các annotation **@Provides** đánh dấu đây là các dependency sẽ được cung cấp cho component, các annotation **@Singleton** đánh dấu các object này chỉ cần tạo một lần và sử dụng xuyên suốt trong project.
Có rất nhiều @Module được tạo ra trong project.

Tiếp theo ta đến với **@Component**: Component là một graph. Chúng ta xây dựng Component, Component sẽ cung cấp các **injected instance** bằng cách sử dụng các @Module. Ví dụ **AppComponent.java**:

```java
@Singleton
@Component(modules = {AndroidInjectionModule.class, AppModule.class, NetModule.class, ActivityProvider.class})
//define what objects should be included as part of the dependency chain by modules
public interface AppComponent {
    void inject(EnglishNowApp app);
    // Note that the activities, services, or fragments that are allowed to request the dependencies declared b
    // (by means of the @Inject annotation) should be declared in this class with individual inject() methods
    //void inject(SplashActivity activity);
    // void inject(MyFragment fragment);
    // void inject(MyService service);

    @Component.Builder
    interface Builder {
        @BindsInstance
        Builder application(Application application);

        AppComponent build();
    }
}
```

Sau khi setup xong, mỗi khi cần dependency nào chúng ta có thể inject vào bằng 2 cách.

Field inject:

```java
public class StatusFragment extends BaseFragment<FragmentStatusBinding, StatusViewModel
    @Inject
    protected @NewsFeedViewModelProviderFactory
    ViewModelProvider.Factory mViewModelFactory;

    @Inject
    protected StatusAdapter statusAdapter;

    @Inject
    protected LinearLayoutManager mLayoutManager;
```

Và constructor inject

```java
@Singleton
public class AppApiHelper implements ApiHelper {
    private Retrofit openTokClient;

    @Inject
    public AppApiHelper(@OpenTokApiClient Retrofit openTokClient) {
        this.openTokClient = openTokClient;
    }

    @Override
    public Single<OpenTokRoom> getOpenTokRoomInfo(String roomName) {
        OpenTokRoomService service = openTokClient.create(OpenTokRoomService.class);
        return service.roomInfo(roomName);
    }
}
```

ii. Builder

Vấn đề: Các model của project có rất nhiều biến thể và có các **optional field**, cần tối giản quá trình tạo lập object.

Giải pháp: Áp dụng Builder pattern để tạo các đối tượng phức tạp chỉ bằng các câu lệnh đơn giản.

Ví dụ:

Khi khởi tạo User lúc đăng ký, ta chưa có nhiều thông tin về User đó, vậy khi đó ta chỉ cần tạo User với các thông tin cần thiết để push vào database, sau này khi có đầy đủ thì ta có thể update lại thông tin.

Lưu ý: Vì project có nhiều loại user nền cần dùng kế thừa/đa hình. Vì vậy Builder cũng được ứng dụng cho base class **AbstractUser.java**, sử dũng **Generic<T>** trong java.

```java
//Base builder class to construct variant type of users
public static class Builder<T extends Builder<T>> {
    protected String email;
    protected String username;
    protected String profilePic; //Optional field

    public T setUsername(String username) {
        this.username = username;
        return (T) this;
    }

    public T setEmail(String email) {
        this.email = email;
        return (T) this;
    }

    public T setProfilePic(String profilePic) {
        this.profilePic = profilePic;
        return (T) this;
    }
}
```

Trong class con **User.java**

```java
public static class Builder extends AbstractUser.Builder<Builder> {
    private List<Skill> skillList;
    private int conversations; //Optional field
    private int drinks; //Optional field
    private int pizzas; //Optional field
    private List<NewsFeedItem> newsFeedItems; //Optional field

    public User build() { return new User( builder: this); }

    public Builder setSkillList(List<Skill> skillList) {
        this.skillList = skillList;
        return this;
    }

    public Builder setConversations(int conversations) {
        this.conversations = conversations;
        return this;
    }

    public Builder setDrinks(int drinks) {
        this.drinks = drinks;
        return this;
    }

    public Builder setPizzas(int pizzas) {
        this.pizzas = pizzas;
        return this;
    }
```

Sử dụng:

```java
.registerFirebaseWithEmail(new RegisterMessage.ServerRequest(email, username, password))
.subscribe(authResult -> {
    // Sign in success, update UI with the signed-in user's information
    FirebaseUser firebaseUser = authResult.getUser();

    if (firebaseUser != null) {
        //push new user to firebase
        User user = new User.Builder()
                .setEmail(email)
                .setUsername(username)
                .setProfilePic(firebaseUser.getPhotoUrl() == null ? "" : firebaseUser.getPhotoUrl()
                .build();
        getDataManager().getUserRepository().createOrUpdate(firebaseUser.getUid(), user);

        //Go to post login activity
        getNavigator().navigateToPostLoginScreen();
    }
    setIsLoading(false);
```

Trong bài Builder pattern còn được sử dụng với các **NewsFeedItem**:

```java
@Override
public Status buildNewsFeedItem() {
    Status.Builder builder = new Status.Builder()
            .setOwnerUid(ownerUid)
            .setOwnerUsername(displayName)
            .setContent(content)
            .setPhotoUrl(photoUrl)
            .setTimestamp(System.currentTimeMillis());


    return builder.build();
}
```

iii. Factory method:

Vấn đề: Khi áp dụng MVVM ở màn hình NewsFeed sẽ có 2 tab Status và Video tương ứng với 2 list dữ liệu khác nhau (một list các status dạng text kèm hình, một list các video). Điều đó đồng nghĩa với việc cần có 2 ViewModel khác nhau nhưng lại có những mối quan hệ với nhau.

Giải pháp: Áp dụng Factory method để tạo ra các ViewModel nào cần điều kiện khi tạo.

Ta tạo class **ViewModelProviderFactory.java**

```java
/**
 * Created by Genius Doan on 29/12/2017.
 * <p>
 * A provider factory pattern which responsible to instantiate multi-type of ViewModels.
 * Used if the ViewModels has a parameterize constructor.
 * <p>
 * Inside this factory, we could even use dependency injection to get everything we need.
 * move the responsibility of creating view models into a separate class which already made code bet
 */

public class ViewModelProviderFactory<V> implements ViewModelProvider.Factory {

    private V viewModel;

    public ViewModelProviderFactory(V viewModel) { this.viewModel = viewModel; }

    @Override
    public <T extends ViewModel> T create(Class<T> modelClass) {
        if (modelClass.isAssignableFrom(viewModel.getClass())) {
            return (T) viewModel;
        }
        throw new IllegalArgumentException("Unknown class name");
    }

}
```

Ta xét các điều kiện sau đó gọi hàm **ViewModelProviderFactory#get()** để tạo các object

```
@Override
public StatusViewModel getViewModel() {
    statusViewModel = ViewModelProviders.of((MainActivity) getActivity(), mViewModelFactory).get(StatusViewModel.class);
    return statusViewModel;
}
```

```
@Override
public VideoStatusViewModel getViewModel() {
    videoStatusViewModel = ViewModelProviders.of((MainActivity) getActivity(), mViewModelFactory).get(VideoStatusViewModel
    return videoStatusViewModel;
}
```

iv. Singleton

Vấn đề: Có những object chỉ cần **tạo một lần** và sử dụng nhiều nơi trong project.

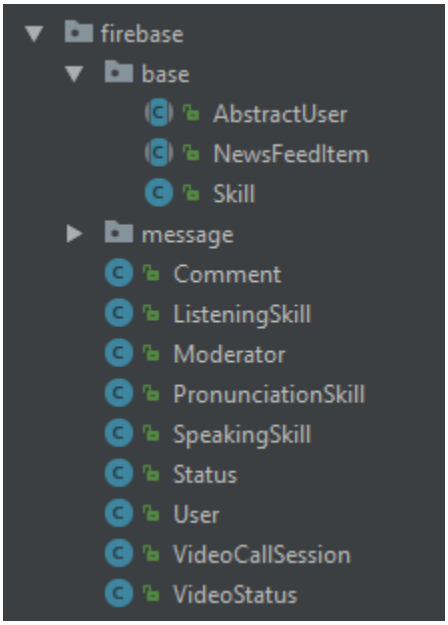
Giải pháp: Nếu object đó tạo bằng DI ta sử dụng annotation @Singleton. Nếu không ta tạo một biến **static** để lưu instance, dùng **private/protected constructor** để giới hạn việc tạo lập instance của class đó.

Ví dụ:

```
public class NewsFeedPostingExecutor {
    private static final String TAG = NewsFeedPostingExecutor.class.getSimpleName();
    private static NewsFeedPostingExecutor instance = null;
    private DataManager dataManager;

    private NewsFeedPostingExecutor() {
        //Singleton
    }

    public static NewsFeedPostingExecutor getInstance() {
        //Lazy init
        if (instance == null) {
            instance = new NewsFeedPostingExecutor();
        }

        return instance;
    }

    public void setDataManager(DataManager dataManager) { this.dataManager = dataManager; }

    public void execute(NewsFeedPostingTicket ticket) {
        NewsFeedItem item = ticket.buildNewsFeedItem();
        if (item != null)
            ticket.post(dataManager, item);
        else
            AppLogger.e(TAG, ...objects: "Cannot post newsfeed item");
    }
}
```

vii. Abstract class

Project áp dụng abstract class cho **tất cả** các model chính. Khi cần mở rộng để thêm các object trong cùng một nhóm chỉ cần extends base abstract class và implement thêm các thông tin cần thiết mà không boiterplate code.



b. Structural pattern

Structural:

- Composite: composes zero-or-more similar objects so that they can be manipulated as one object.
- Adapter (also known as Wrapper): allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- ViewHolder: used for View caching - Holder (arbitrary) object holds child widgets of each row and when row is out of View then findViewById() won't be called but View will be recycled and widgets will be obtained from Holder
- Facade: provides a simplified interface to a large body of code:
- Repository: can be viewed as a special kind of Façade. For Mediating between the domain and data mapping layers using a collection-like interface for accessing domain objects.
- Marker interface: used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

i. Adapter:

Vấn đề: Trong việc hiển thị và cập nhật danh sách các newsfeed item, ta cần giao tiếp giữa hai thành phần khác nhau hoàn toàn là **list dữ liệu trong memory** và các **layouts/views**.

Giải pháp: Sử dụng Adapter pattern của Android.

Tạo base class **BaseRecyclerViewAdapter<T>** để làm khung, đồng thời giữ một List dữ liệu **mItems**

```
public abstract class BaseRecyclerViewAdapter<T> extends RecyclerView.Adapter<RecyclerView.ViewHolder> {
    protected List<T> mItems = new ArrayList<>();

    public void setItems(List<T> items) {
        /*
        if (this.mItems == items)
            return;
            */

        mItems = items;
        notifyDataSetChanged();
    }

    protected int indexOf(T item) { return mItems.indexOf(item); }

    @Override
    public int getItemCount() { return mItems.size(); }
}
```

Và class con **StatusAdapter.java** sử dụng hàm **bindStatusItemView()** để convert từ dữ liệu sang layout với thông tin và trạng thái tương ứng.

```
public class StatusAdapter extends BaseRecyclerViewAdapter<StatusItemViewModel> {
    protected Context mContext;

    public StatusAdapter(Context context) {
        super();
        mContext = context;
    }

    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        StatusItemBinding binding = DataBindingUtil.inflate(LayoutInflater.from(mContext),
                R.layout.status_item, parent,  attachToParent: false);
        return new StatusItemViewHolder(binding);
    }

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
        bindStatusItemView(holder, position);
    }

    private void bindStatusItemView(RecyclerView.ViewHolder holder, int position) {
        StatusItemViewHolder viewHolder = (StatusItemViewHolder) holder;
        viewHolder.onBind(position);
    }
```

ii. ViewHolder

Vấn đề: Trong Android, việc tìm view từ các id rất **tốn thời gian và tài nguyên của máy**, với một list dữ liệu lớn mỗi lần chúng ta muốn cập nhật thông tin đều phải tìm view bằng cách gọi **findViewById()** thì cực kì mất thời gian và **ảnh hưởng đến hiệu năng**.

Giải pháp: Sử dụng ViewHolder pattern, chúng ta sẽ **cache/hold** các instance của các View cần thiết lại và lấy ra **tái sử dụng** khi cần cập nhật dữ liệu của View.

```
//ViewHolder pattern
public class StatusItemViewHolder extends BaseViewHolder {
    private StatusItemBinding statusItemBinding;

    public StatusItemViewHolder(StatusItemBinding binding) {
        super(binding.getRoot());
        statusItemBinding = binding;
    }

    @Override
    public void onBind(int position) { statusItemBinding.setViewModel(mItems.get(position)); }
}
```

iii. Façade

Vấn đề: Trong project có những thao tác được thực hiện **nhiều lần** theo một **trình tự nhất định**, sử dụng **nhiều nơi** trong ứng dụng. Lập trình viên khó khăn khi phải nhớ mình cần làm cái gì, mặc khác nếu copy/paste những đoạn code đó thì sẽ tạo ra **boiterplate code**.

Giải pháp: Sử dụng Façade pattern, tạo một **Handler/Executor/Manager** đại diện, cung cấp cho nó những thành phần cần thiết và chỉ cần thực thi bất cứ chỗ nào cần.

Ví dụ: class **ConversationMatchingHandler.java**

```
public class ConversationMatchingHandler {
    private static final String TAG = ConversationMatchingHandler.class.getSimpleName();
    private DataManager appDataManager;
    private SchedulerProvider schedulerProvider;
    private CompositeDisposable compositeDisposable;
    private VideoCallSessionDataSource repository;
    private PublishSubject<OpenTokRoom> matchingSubject;
    private OpenTokRoom currentRoom = null;

    public ConversationMatchingHandler(DataManager dataManager, SchedulerProvider schedulerProvider, Compos
        appDataManager = dataManager;
        this.schedulerProvider = schedulerProvider;
        this.compositeDisposable = compositeDisposable;
        repository = dataSource;
        matchingSubject = PublishSubject.create();
    }

    public PublishSubject<OpenTokRoom> getMatchingSubject() { return matchingSubject; }

    public void startConversationMatching(String uid) {
        //Use Firebase helper to do conversation matching to match uid of learner and speaker
        Single<String> matchingSingle = appDataManager.doConversationMatching(uid);
        matchingSingle
                .subscribeOn(schedulerProvider.io())
                .observeOn(schedulerProvider.ui())
                .subscribe(new SingleObserver<String>() {
                    @Override
                    public void onSubscribe(Disposable d) {
                        //Add to composite disposable to dispose together with other observer when app sto
                        compositeDisposable.add(d);
                    }
```

```
                    @Override
                    public void onSuccess(String opponentUid) {
                        //Complete, remove learner from available list (available_learners node on
                        appDataManager.removeLearnerFromAvailableList(uid);

                        //Create conversation room
                        createConversationRoom(uid, opponentUid);
                    }

                    @Override
                    public void onError(Throwable e) {
                        e.printStackTrace();
                        matchingSubject.onError(e);
                    }
                });
    }

    public void stopConversationMatching(String uid) {
        appDataManager.removeLearnerFromAvailableList(uid);
    }

    private void createConversationRoom(String uid, String opponentUid) {...}

    private void waitForSessionCreated(String sessionId) {...}

    private void createOrUpdateSession(OpenTokRoom openTokRoom, String uid) {...}

    private String buildRoomName(String uid, String opponentUid) {...}
```

Sử dụng trong ViewModel một cách đơn giản, chỉ cần gọi 1 trong 2 hàm cần sử dụng, mọi thông tin về việc nó vẫn hành thế nào ta không cần quan tâm.

```
public void onFindButtonClicked() {
    if (!isFinding) {
        getNavigator().changeViewsToFindingMode();
        if (getNavigator().selfCheckRequiredPermissions()) {
            startFinding();
        } else {
            getNavigator().requestPermissions();
        }
    } else {
        getNavigator().changeViewsToNormalMode();
        stopFinding();
    }

    isFinding = !isFinding;
}

private void stopFinding() {
    handler.stopConversationMatching(getDataManager().getCurrentUserUid());
}

public void startFinding() {
    handler.startConversationMatching(getDataManager().getCurrentUserUid());
}
```

Ngoài ra trong project còn áp dụng Façade pattern cho NewsFeedPostingExecutor, và các data manager khác.

iv. Repository

Vấn đề: Vì project thao tác với dữ liệu rất nhiều, nên cần một cách để thống nhất và dễ dàng thao tác với dữ liệu.

Giải pháp: Sử dụng Repository pattern để ViewModel có thể dễ dàng tương tác với data layer.

Ví dụ:

```java
public class UserRepository implements UserDataSource {
    private static final String USER_PROFILE = "user_profile";

    private FirebaseAuth mAuth;
    private FirebaseDatabase database;


    @Inject
    public UserRepository(FirebaseAuth auth, FirebaseDatabase database) {
        this.mAuth = auth;
        this.database = database;
    }

    @Override
    public Observable<User> fetch() {
        return Observable.create(emitter -> {...});
    }

    @Override
    public Single<User> fetchOnce(String uid) {
        return Single.create(emitter -> {...});
    }

    @Override
    public void remove(String userUid) {
        database.getReference(USER_PROFILE).child(userUid).removeValue();
    }

    @Override
    public void createOrUpdate(String userUid, User user) {
        database.getReference(USER_PROFILE).child(userUid).setValue(user);
```

```java
public class NewsFeedItemRepository implements NewsFeedItemDataSource {
    private static final String STATUS = "status";

    private FirebaseDatabase database;

    @Inject
    public NewsFeedItemRepository(FirebaseDatabase database) { this.database = database; }

    @Override
    public Observable<List<Status>> fetchAll() {
        return Observable.create(emitter -> database.getReference(STATUS)
                .addValueEventListener(new ValueEventListener() {
                    @Override
                    public void onDataChange(DataSnapshot dataSnapshot) {
                        List<Status> statuses = new ArrayList<>();
                        for (DataSnapshot itemSnapshot : dataSnapshot.getChildren()) {
                            Status status = itemSnapshot.getValue(Status.class);
                            statuses.add(status);
                        }

                        emitter.onNext(statuses);
                    }

                    @Override
                    public void onCancelled(DatabaseError databaseError) {
                        emitter.onError(databaseError.toException());
                    }
                }));
    }
```

Mặc khác khi áp dụng Repository pattern nói riêng và các data handler/helper nói chúng, nhóm có tạo các **interface DataSource** làm **phần khung** trước sau đó mới implement lại (một biến thể của Template pattern)

Việc này giúp ta có thể dễ dàng tạo thêm một class tương tự (**UserRepositoryMock**/**UserRepositoryFake**) để thực hiện **Mock** dữ liệu cho các **unit test/intergration test**.

Trong project còn các Repository khác như MediaRepository, VideoCallSessionRepository

v. Marker interface

Vấn đề: Trong project Android để chuyển dữ liệu giữa các Activity chúng ta convert nó thành byte-stream để gửi đi và tạo lại object từ byte-stream nhận được. Vì vậy cần một thứ gì đó **đánh dấu hiệu** là một object nào đó có thể gửi như vậy.

Giải pháp: Sử dụng marker interface Serializable của Android

Ví dụ:

```
@IgnoreExtraProperties
public class User extends AbstractUser implements Serializable {
    @PropertyName("skills")
    private List<Skill> skillList;

    @PropertyName("conversations")
    private int conversations;

    @PropertyName("drinks")
    private int drinks;

    @PropertyName("pizzas")
    private int pizzas;

    @PropertyName("newsfeed_item")
    private List<NewsFeedItem> newsFeedItems;

    public User() {
        //For deserialize
    }
```

c. Behavioral pattern

Behavioral:

- Iterator: accesses the elements of an object sequentially without exposing its underlying representation.
- Observer: is a publish/subscribe pattern which allows a number of observer objects to see an event.
- Strategy: allows one of a family of algorithms to be selected on-the-fly at runtime
- Template method: defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.

i. Observer

Vấn đề: Vì là một app phụ thuộc vào realtime database, nên sẽ có rất nhiều đối tượng trong app như user info, newsfeed items, chat message, incoming request,... phụ thuộc vào dữ liệu và các đối tượng khác.

Giải pháp: Sử dụng **Observable/Single** của RxJava hoặc **LiveData** của Android. Khi realtime database cập nhật trạng thái mới, thì các thành phần trong app phải cập nhật thay đổi theo.

Ví dụ:

Để fetch và observe dữ liệu của các status ta sử dụng hàm **fetchAll().** Trong hàm này ta tạo ra một object kiểu Observable (có thể observe được). Mỗi khi realtime database cập nhật, ta dùng **emmiter** gửi một event kèm theo dữ liệu bằng **emitter.onNext()**.

```
@Override
public Observable<List<Status>> fetchAll() {
    return Observable.create(emitter -> database.getReference(STATUS)
            .addValueEventListener(new ValueEventListener() {
                @Override
                public void onDataChange(DataSnapshot dataSnapshot) {
                    List<Status> statuses = new ArrayList<>();
                    for (DataSnapshot itemSnapshot : dataSnapshot.getChildren()) {
                        Status status = itemSnapshot.getValue(Status.class);
                        statuses.add(status);
                    }

                    emitter.onNext(statuses);
                }

                @Override
                public void onCancelled(DatabaseError databaseError) {
                    emitter.onError(databaseError.toException());
                }
            }));
}
```

Trong ViewModel hoặc bất cứ chỗ nào cần nhận event từ Observable trên, ta tạo một observer và subscribe vào. Mỗi khi có **event** được **emit**, các observer sẽ được **notify**, sau đó lấy dữ liệu trạng thái mới ra và cập nhật.

```
private void loadNewsFeedItems() {
    getCompositeDisposable().add(getDataManager()
            .getNewsFeedItemRepository()
            .fetchAll()
            .subscribeOn(getSchedulerProvider().io())
            .observeOn(getSchedulerProvider().ui())
            .subscribe(statusList -> {
                if (statusList != null) {
                    statusItemsLiveData.setValue(getViewModelList(statusList));

                    getNavigator().setPlaceHolderTextVisible(statusList.size() == 0);
                }
            }, throwable -> throwable.printStackTrace()));
}
```

Hoặc sử dụng LiveData của Android.

Khai báo trong class **StatusFragment.java**

```
private final ObservableArrayList<StatusItemViewModel> statusItems = new ObservableArrayList<>();
private final MutableLiveData<List<StatusItemViewModel>> statusItemsLiveData;
```

Tạo các Observer để observe vào LiveData.

```
private void subscribeToLiveData() {
    statusViewModel.getStatusItemsLiveData().observe(getBaseActivity(), statusItemViewModels -> {
        statusViewModel.setStatusItems(statusItemViewModels);
    });
}
```

Mỗi khi giá trị của LiveData thay đổi, các observer sẽ được notify để cập nhật trạng thái mới.

ii. Strategy

Vấn đề: Trong project, user có thể chọn post các status thông thường (text kèm hình ảnh) hoặc video status. Việc lựa chọn cách post **diễn ra run-time**.

Giải pháp: Sử dụng Strategy kết hợp với Façade để post các loại Status.

Ví dụ:

Đầu tiên ta tạo base class cho các strategy:

```
/**
 * Created by Genius Doan on 10/01/2018.
 * <p>
 * Provide skeleton/interface for Strategy Pattern that used to posting newsfeed item
 */

//TODO: Support factory
public interface NewsFeedPostingTicket<T extends NewsFeedItem> {
    T buildNewsFeedItem();

    void post(DataManager dataManager, T item);
}
```

Implement các Strategy

```java
public class StatusPostingTicket implements NewsFeedPostingTicket<Status> {
    private String ownerUid;
    private String displayName;
    private String content;
    private String photoUrl = ""; //Optional

    public StatusPostingTicket(String ownerUid, String displayName, String content) {...}

    public void setPhotoUrl(String photoUrl) { this.photoUrl = photoUrl; }

    @Override
    public Status buildNewsFeedItem() {
        Status.Builder builder = new Status.Builder()
                .setOwnerUid(ownerUid)
                .setOwnerUsername(displayName)
                .setContent(content)
                .setPhotoUrl(photoUrl)
                .setTimestamp(System.currentTimeMillis());

        return builder.build();
    }

    @Override
    public void post(DataManager dataManager, Status status) {
        dataManager.getNewsFeedItemRepository().createOrUpdate(status);
    }
}
```

```java
public class VideoStatusPostingTicket implements NewsFeedPostingTicket<VideoStatus> {
    private String ownerUid;
    private String displayName;
    private String videoUrl;

    public VideoStatusPostingTicket(String uid, String userDisplayName, String videoUrl) {
        ownerUid = uid;
        displayName = userDisplayName;
        this.videoUrl = videoUrl;
    }

    @Override
    public VideoStatus buildNewsFeedItem() { return null; }

    @Override
    public void post(DataManager dataManager, VideoStatus item) {

    }
}
```

Chúng ta có thể tạo thêm Strategy nếu sau này muốn mở rộng project ra để post audio, file, văn bản,..

Sử dụng Façade làm context để thực thi các Strategy, tạo class **NewsFeedPostingExecutor.java**:

```java
public class NewsFeedPostingExecutor {
    private static final String TAG = NewsFeedPostingExecutor.class.getSimpleName();
    private static NewsFeedPostingExecutor instance = null;
    private DataManager dataManager;

    private NewsFeedPostingExecutor() {
        //Singleton
    }

    public static NewsFeedPostingExecutor getInstance() {
        //Lazy init
        if (instance == null) {
            instance = new NewsFeedPostingExecutor();
        }

        return instance;
    }

    public void setDataManager(DataManager dataManager) { this.dataManager = dataManager; }

    public void execute(NewsFeedPostingTicket ticket) {
        NewsFeedItem item = ticket.buildNewsFeedItem();
        if (item != null)
            ticket.post(dataManager, item);
        else
            AppLogger.e(TAG, ...objects: "Cannot post newsfeed item");
    }
}
```

Sử dụng trong các ViewModels:

```java
public void doPostStatus(String content) {
    String uid = getDataManager().getCurrentUserUid();
    String userDisplayName = getDataManager().getCurrentUserDisplayName();

    if (uid == null || userDisplayName == null) {
        Toast.makeText(context, "User authentication not found!", Toast.LENGTH_SHORT).show();
        return;
    }

    StatusPostingTicket ticket = new StatusPostingTicket(uid, userDisplayName, content);
    if (queuedMedia != null) {
        ticket.setPhotoUrl(queuedMedia.getUploadUrl().getURL());
    }

    NewsFeedPostingExecutor.getInstance().execute(ticket); //Execute strategy
}

public void doPostVideoStatus() {
    String uid = getDataManager().getCurrentUserUid();
    String userDisplayName = getDataManager().getCurrentUserDisplayName();

    if (uid == null || userDisplayName == null || queuedMedia == null) {
        Toast.makeText(context, "User authentication not found!", Toast.LENGTH_SHORT).show();
        return;
    }

    VideoStatusPostingTicket ticket = new VideoStatusPostingTicket(uid, userDisplayName, queuedMe

    NewsFeedPostingExecutor.getInstance().execute(ticket); //Execute strategy
}
```

d. Middleware

Middleware:

Transport:

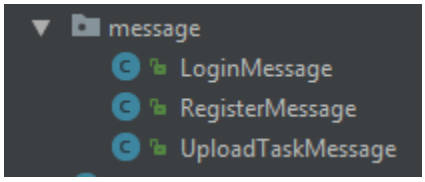
- [ ] Distributed object: Handle, register, dictionary,..
- [x] Message.

Vấn đề: Project ngoài việc thao tác dữ liệu với database, còn phải luân chuyển **transport** dữ liệu **giữa các thành phần của hệ thống** (Activity/Fragment/Service,..)

Giải pháp: Sử dụng Transport Message.

Ví dụ:

Tạo các class model Message để chứa dữ liệu cần truyền




```java
/**
 * Created by Genius Doan on 06/01/2018.
 * <p>
 * Store information about a upload task response
 * Use it as a message for response information from data manager back to View-Model.
 */
public class UploadTaskMessage {
    private double progress;
    private Uri result;

    public UploadTaskMessage(double progress, Uri result) {
        this.progress = progress;
        this.result = result;
    }

    public UploadTaskMessage(double progress) { this.progress = progress; }

    public double getProgress() { return progress; }

    public Uri getResult() { return result; }
}
```

Sau đó nạp thông tin vào và truyền dữ liệu qua lại giữa ViewModel/Data manager

Bên gửi:

```
private void putFile(Uri data, StorageReference reference, ObservableEmitter e) {
    UploadTask uploadTask = reference.putFile(data);

    // Register observers to listen for when the download is done or if it fails
    uploadTask.addOnProgressListener(taskSnapshot -> {
        double progress = (100.0 * taskSnapshot.getBytesTransferred()) / taskSnapshot.getTotalBy
        if (progress < 100)
            e.onNext(new UploadTaskMessage(progress));
    }).addOnFailureListener(exception -> {
        // Handle unsuccessful uploads
        e.onError(exception);
    }).addOnSuccessListener(taskSnapshot -> {
        // taskSnapshot.getMetadata() contains file metadata such as size, content-type, and dow
        Uri downloadUrl = taskSnapshot.getDownloadUrl();
        e.onNext(new UploadTaskMessage( progress: 100, downloadUrl));
        e.onComplete();
    });
}
```

Bên nhận

```
item.setUploadRequest(getDataManager().getMediaRepository().putPhoto(item.getLocalUri()));
getCompositeDisposable().add(
        item.getUploadRequest()
                .subscribeOn(getSchedulerProvider().io())
                .observeOn(getSchedulerProvider().ui())
                .subscribe(new Consumer<UploadTaskMessage>() {
                    int lastProgress = -1;

                    @Override
                    public void accept(UploadTaskMessage uploadTaskMessage) throws Exception {
                        //Parse messages to get information about the Uploading Task
                        int progress = (int) Math.round(uploadTaskMessage.getProgress());
                        if (progress == lastProgress)
                            return;

                        //Update
                        item.getPreview().setProgress(progress);
                        if (progress == 100)
                            onUploadSuccess(item, uploadTaskMessage.getResult());
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        AppLogger.d( s: "Upload request failed. " + throwable.getMessage());
                        onUploadFailure(item, throwable);
                    }
                })
);
```