Design Rationale

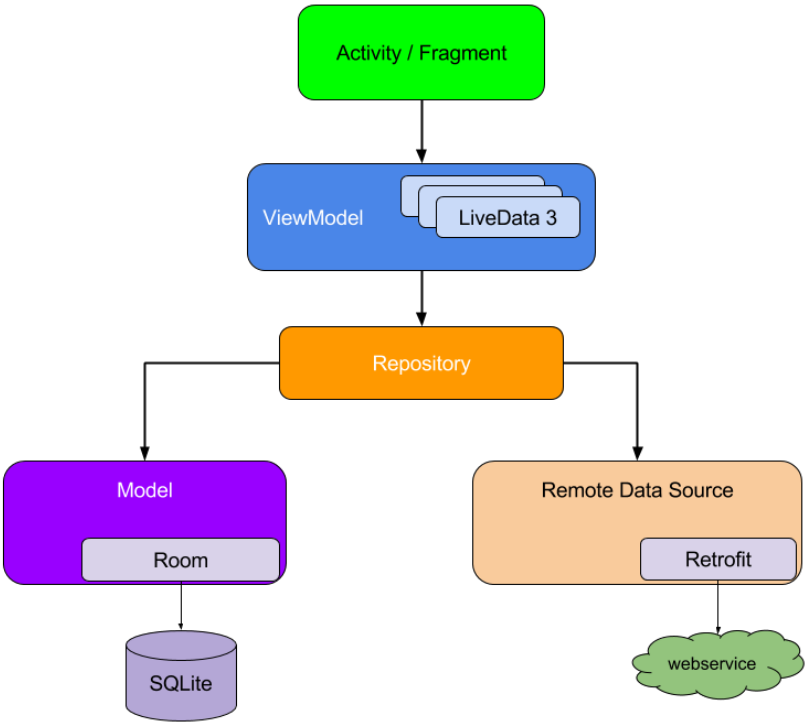
FIT3077 Assignment 3
Team Name - Optimize Prime
Member Names - Sadeeptha Bandara, Kaveesha Nissanka

Architecture

As at Assignment 2, no specific architecture was strictly followed when implementing the application, though there was a distinction between the data model from the remote data source and the application logic.
From the perspective of the MVC pattern , given that the app was an Android application, an Activity acted as both the View and the controller and interacted with the model through interfaces.

For Assignment 3, we have migrated to using the MVVM architecture, which is the recommended architecture for Android applications by Google. Android's Jetpack library provides MVVM architecture components and makes it easier to implement MVVM in the android context.

The MVVM architecture is an extension of the MVC architecture with a ViewModel replacing the Controller in MVC.



Activity/Fragment: This is similar to the View in MVC and is purely responsible for managing the interaction with the UI and uses the ViewModel for application logic and communication with remote data sources.
The activity uses the observer pattern to observe changes from the ViewModel

ViewModel: The ViewModel provides data for a specific activity and contains business logic specific to the activity. It interacts with the repository to access remote data sources. The ViewModels that we have used in the application are extensions of Android Jetpack's ViewModel and are lifecycle-aware. (Aware of when an activity is opened, destroyed, when the device configuration changes)
The ViewModel can observe changes from the repository

Repository: The Repository is a clean API to access the model, and abstracts access to a particular aspect of the model, providing an easy way to access data to the ViewModel. The repository is generally considered to be a part of the model.

Model: As implemented in Assignment 2, we use Retrofit to access the Web service, which allows access to the API endpoints through standardized interfaces.

The reasons for why we have adopted MVVM as our architecture is that it allows for,

1. Separation of concerns
   The Views are in charge of UI rendering and ViewModels are in charge of adapting the data from the remote data source as well as parsing data according to the view's requirements. The repository is in charge of accessing the model data and conducting low level API request operations.

2. Follows Clean code architecture
   Clean code architecture as discussed by Robert C Martin emphasizes that dependencies should point inward towards the inner layers of the application.

   By following the observer pattern, MVVM follows this, while MVC does not because the controller is aware of the activity and is dependent on the activity's implementation.
   The ViewModel does not store a reference to the Activity as such is not dependent on it

3. Easy to extend due to separated concerns and due to only inward dependencies each component can be unit-tested.

4. Recommended by google and provides supporting libraries through Android Jetpack, such as Android's ViewModel, LiveData.

Note: Due to time constraints MVVM has been strictly applied to the most complex activities only.

Design Patterns

Observer Pattern
The observer pattern is applied throughout the layers of the application using Android Jetpack's LiveData, which is observable data.

API responses abstracted using the CallAdapter in the Repository are observed by specific ViewModels that require that specific repository. The activity itself listens to the changes in the ViewModel and updates the UI accordingly.

Singleton and Facade
Each repository follows the singleton pattern.The singleton is enforced by the dependency injection framework Dagger2.
Further, as used in assignment 2, the RetrofitClient used for API requests is a singleton that is available globally. Both repositories and the RetrofitClient act as Facades to the model and the Remote respectively.

Factory Pattern
The factory pattern is used for generating ViewModels, for use within individual activities. Each ViewModel class is injected to the ViewModelFactory, which constructs them.

Design Principles

Dependency Injection
Dependency Injection is applied in order to prevent the accumulation of dependencies in the layers of the application.

To apply this, the dependency injection framework Dagger2 is used, and it is used to inject the application context as well as the Repositories to the ViewModels. The ViewModels themselves are injected to each activity, using a ViewModel factory which is in charge of constructing the ViewModels.

Liskov Substitution Principle
The ViewModels are subclassed using a CommonViewModel and each child follows the Liskov substitution principle by being individually substitutable to wherever the

CommonViewModel is used. CommonViewModel itself subclasses ViewModel, provided by the Android Jetpack library.

Single Responsibility Principle
Each activity is encapsulated with the logic relevant to that particular activity only and in the case of MVVM, is specific to the UI logic of the application.
Further, the model classes used in the application have been improved over their usage in Assignment 2, by encapsulating the attributes further.

Dependency Inversion

All repository classes implement an interface relevant to the particular model that it abstracts so that it provides a clean interface for the ViewModels to use and it is very helpful inorder to define which operations are needed, inverting the direction of dependencies.

Abstraction

As a notable improvement over Assignment 2, API calls have been abstracted into a generic Network adapter that supports LiveData.
Previously, all the API calls were made within a particular activity and contained duplicate code for handling the request.

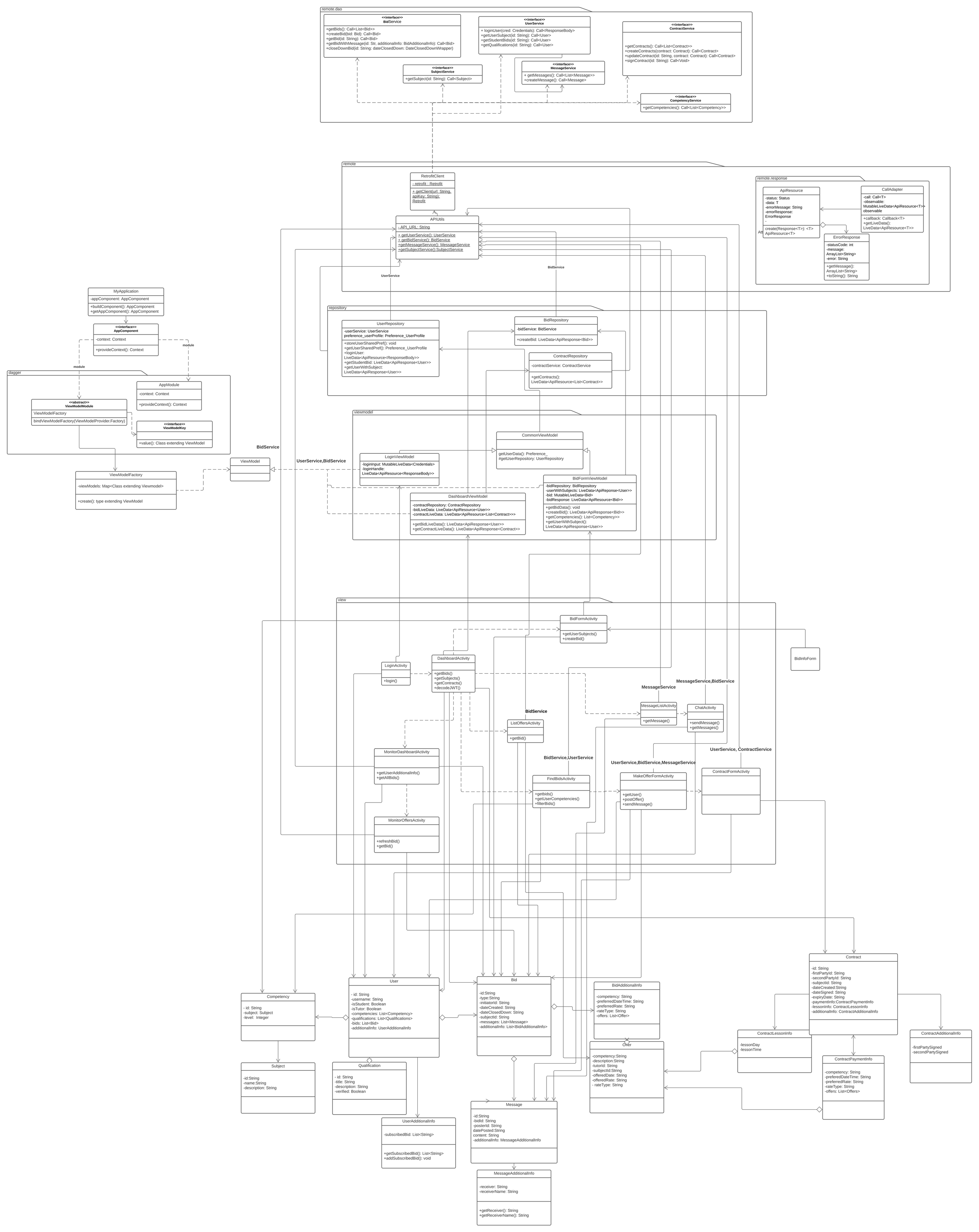
This has been avoided by using a generic Adapter “CallAdapter”, to make API requests. The API request is intercepted at each level (ViewModel, Repository), so that necessary data handling can be done and the activity can only concern itself with UI logic.

Package Principles
The packages were made in the application, taking into account cohesion and coupling and how relevant they are to each other.

Refactoring techniques

Composing Methods is a technique that can be used to streamline methods and remove code duplication. And one instance where we did this was to handle the API responses using a generic CallAdapter, that is sent through to each Activity, which reduces most of the duplicate code.

We also organised our data better to help with our data handling. Organising Data was used in instances when the userData was needed. A common sharedPreference(Android local storage) that can be used across the whole app was implemented to get the necessary data. Repositories for most of the java objects were also implemented to get the data from those objects easier.


remote.dao
<<interface>>
BidService
+getBids(): Call<List<Bid>>
+createBid(bid: Bid): Call<Bid>
+getBid(id: String): Call<Bid>
+getBidWithMessage(id: Str, additionalInfo: BidAdditionalInfo): Call<Bid>
+closeDownBid(id: String: dateClosedDown: DateClosedDownWrapper)
<<interface>>
UserService
+ loginUser(cred: Credentials): Call<ResponseBody>
+getUserSubject(id: String): Call<User>
+getStudentBids(id: String): Call<User>
+getQualifications(id: String): Call<User>
<<interface>>
ContractService
+getContracts(): Call<List<Contract>>
+createContracts(contract: Contract): Call<Contract>
+updateContract(id: String, contract: Contract): Call<Contract>
+signContract(id: String): Call<Void>
<<interface>>
SubjectService
+getSubject(id: String): Call<Subject>
<<interface>>
MessageService
+ getMessages(): Call<List<Message>>
+createMessage(): Call<Message>
<<interface>>
CompetencyService
+getCompetencies(): Call<List<Competency>>
remote
RetrofitClient
- retrofit : Retrofit
+ getClient(url: String, apiKey: String): Retrofit
APIUtils
- API_URL: String
+ getUserService(): UserService
+ getBidService(): BidService
+getMessageService(): MessageService
+getSubjectService():SubjectService
UserService
BidService
remote.response
ApiResource
-status: Status
-data: T
-errorMessage: String
-errorResponse: ErrorResponse
-
create(Response<T>): <T> ApiResource<T>
CallAdapter
-call: Call<T>
-observable: MutableLiveData<ApiResource<T>> observable
+callback: Callback<T>
+getLiveData(): LiveData<ApiResource<T>>
Alt
ErrorResponse
-statusCode: int
-message: ArrayList<String>
-error: String
+getMessage(): ArrayList<String>
+toString(): String
repository
UserRepository
-userService: UserService
preference_userProfile: Preference_UserProfile
+storeUserSharedPref(): void
+getUserSharedPref(): Preference_UserProfile
+loginUser: LiveData<ApiResource<ResponseBody>>
+getStudentBid: LiveData<ApiResponse<User>>
+getUserWithSubject: LiveData<ApiResponse<User>>
BidRepository
-bidService: BidService
+createBid: LiveData<ApiResponse<Bid>>
ContractRepository
-contractService: ContractService
+getContracts(): LiveData<ApiResource<List<Contract>>
viewmodel
CommonViewModel
getUserData(): Preference_
#getUserRepository: UserRepository
LoginViewModel
-loginInput: MutableLiveData<Credentials>
-loginHandle: LiveData<ApiResource<ResponseBody>>
BidFormViewModel
-bidRepository: BidRepository
-userWithSubjects: LiveData<ApiReponse<User>>
-bid: MutableLiveData<Bid>
-bidResponse: LiveData<ApiResource<Bid>>
+getBidData(): void
+createBid(): LiveData<ApiResponse<Bid>>
+getCompetencies(): List<Competency>>
+getUserWithSubject(): LiveData<ApiResponse<User>>
DashboardViewModel
-contractRepository: ContractRepository
-bidLiveData: LiveData<ApiResource<User>>
-contractLiveData: LiveData<ApiResource<List<Contract>>>
+getBidLiveData(): LiveData<ApiResponse<User>>
+getContractLiveData(): LiveData<ApiResponse<Contract>>
ViewModel
UserService,BidService
BidService
MyApplication
-appComponent: AppComponent
+buildComponent(): AppComponent
+getAppComponent(): AppComponent
<<interface>>
AppComponent
-context: Context
+provideContext(): Context
module
module
dagger
AppModule
-context: Context
+provideContext(): Context
<<abstract>>
ViewModelModule
ViewModelFactory
bindViewModelFactory(ViewModelProvider.Factory)
<<interface>>
ViewModelKey
+value(): Class extending ViewModel
ViewModelFactory
-viewModels: Map<Class extending Viewmodel>
+create(): type extending ViewModel
view
BidFormActivity
+getUserSubjects()
+createBid()
BidInfoForm
LoginActivity
+login()
DashboardActivity
+getBids()
+getSubjects()
+getContracts()
+decodeJWT()
MessageService
MessageService,BidService
MessageListActivity
+getMessage()
ChatActivity
+sendMessage()
+getMessages()
BidService
ListOffersActivity
+getBid()
UserService, ContractService
BidService,UserService
UserService,BidService,MessageService
MonitorDashboardActivity
+getUserAdditionalInfo()
+getAllBids()
FindBidsActivity
+getbids()
+getUserCompetencies()
+filterBids()
MakeOfferFormActivity
+getUser()
+postOffer()
+sendMessage()
ContractFormActivity
MonitorOffersActivity
+refreshBid()
+getBid()
Competency
- id: String
-subject: Subject
-level: Integer
User
- id: String
-username: String
-isStudent: Boolean
-isTutor: Boolean
-competencies: List<Competency>
-qualifications: List<Qualifications>
-bids: List<Bid>
-additionalInfo: UserAdditionalInfo
Bid
-id:String
-type:String
-initiatorId: String
-dateCreated: String
-dateClosedDown: String
-subjectId: String
-messages: List<Message>
-additionalInfo: List<BidAdditionalInfo>
BidAdditionalInfo
-competency: String
-preferredDateTime: String
-preferredRate: String
-rateType: String
-offers: List<Offer>
Contract
-id: String
-firstPartyId: String
-secondPartyId: String
-subjectId: String
-dateCreated:String
-dateSigned: String
-expiryDate: String
-paymentInfo:ContractPaymentInfo
-lessonInfo: ContractLessonInfo
-additionalInfo: ContractAdditionalInfo
ContractLessonInfo
-lessonDay
-lessonTime
ContractAdditionalInfo
-firstPartySigned
-secondPartySigned
Offer
-competency:String
-description:String
-tutorId: String
-suibjectId:String
-offeredDate: String
-offeredRate: String
- rateType: String
ContractPaymentInfo
-competency: String
-preferedDateTime: String
-preferredRate: String
-rateType: String
-offers: List<Offers>
Subject
-id:String
-name:String
-description: String
Qualification
- id: String
-title: String
-description: String
-verified: Boolean
UserAdditionalInfo
-subscribedBid: List<String>
+getSubscribedBid(): List<String>
+addSubscribedBid(): void
Message
-id:String
-bidId: String
-posterId: String
datePosted:String
content: String
-additionalInfo: MessageAdditionalInfo
MessageAdditionalInfo
-receiver: String
-receiverName: String
+getReceiver(): String
+getReceiverName(): String