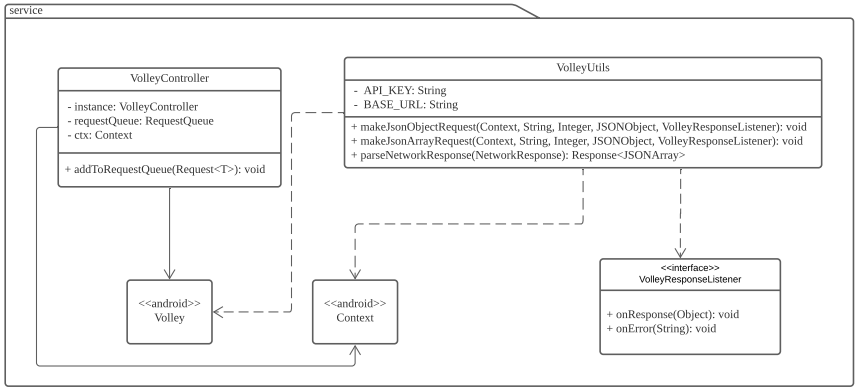
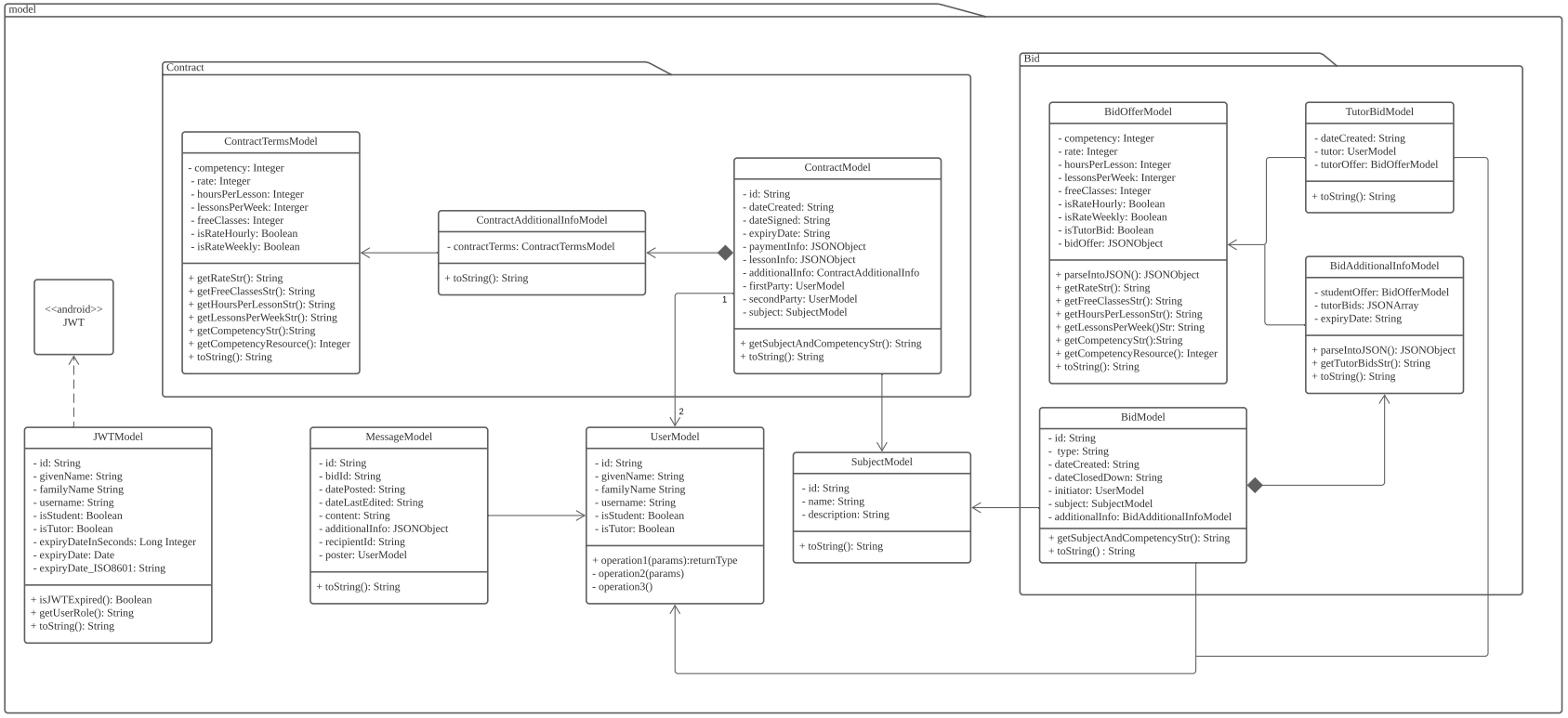
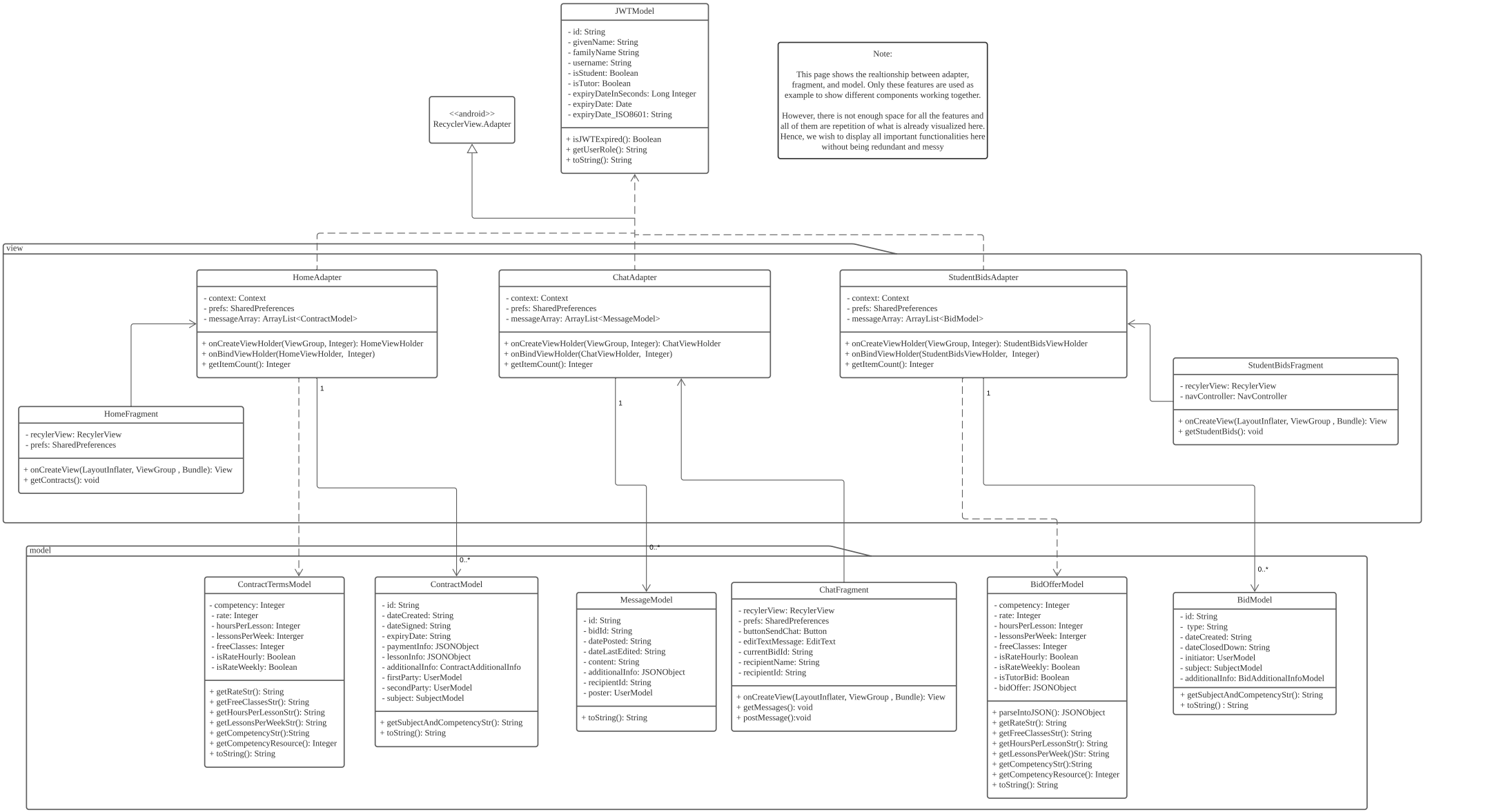


FIT3077 Assignment 2
Team Ouroboros

This first page shows the individual packages for model and service respectively, since all the packages do not fit in this page.

Hence, the other pages will explain inter_package interactions such as how fragment, adapter, and model work together for a specific feature




JWTModel
- id: String
- givenName: String
- familyName String
- username: String
- isStudent: Boolean
- isTutor: Boolean
- expiryDateInSeconds: Long Integer
- expiryDate: Date
- expiryDate_ISO8601: String
+ isJWTExpired(): Boolean
+ getUserRole(): String
+ toString(): String
Note:
This page shows the realtionship between adapter, fragment, and model. Only these features are used as example to show different components working together.
However, there is not enough space for all the features and all of them are repetition of what is already visualized here. Hence, we wish to display all important functionalities here without being redundant and messy
<<android>>
RecyclerView.Adapter
view
HomeAdapter
- context: Context
- prefs: SharedPreferences
- messageArray: ArrayList<ContractModel>
+ onCreateViewHolder(ViewGroup, Integer): HomeViewHolder
+ onBindViewHolder(HomeViewHolder, Integer)
+ getItemCount(): Integer
ChatAdapter
- context: Context
- prefs: SharedPreferences
- messageArray: ArrayList<MessageModel>
+ onCreateViewHolder(ViewGroup, Integer): ChatViewHolder
+ onBindViewHolder(ChatViewHolder, Integer)
+ getItemCount(): Integer
StudentBidsAdapter
- context: Context
- prefs: SharedPreferences
- messageArray: ArrayList<BidModel>
+ onCreateViewHolder(ViewGroup, Integer): StudentBidsViewHolder
+ onBindViewHolder(StudentBidsViewHolder, Integer)
+ getItemCount(): Integer
HomeFragment
- recylerView: RecylerView
- prefs: SharedPreferences
+ onCreateView(LayoutInflater, ViewGroup , Bundle): View
+ getContracts(): void
StudentBidsFragment
- recylerView: RecylerView
- navController: NavController
+ onCreateView(LayoutInflater, ViewGroup , Bundle): View
+ getStudentBids(): void
1
1
1
1
model
0..*
0..*
0..*
ContractTermsModel
- competency: Integer
- rate: Integer
- hoursPerLesson: Integer
- lessonsPerWeek: Interger
- freeClasses: Integer
- isRateHourly: Boolean
- isRateWeekly: Boolean
+ getRateStr(): String
+ getFreeClassesStr(): String
+ getHoursPerLessonStr(): String
+ getLessonsPerWeekStr(): String
+ getCompetencyStr():String
+ getCompetencyResource(): Integer
+ toString(): String
ContractModel
- id: String
- dateCreated: String
- dateSigned: String
- expiryDate: String
- paymentInfo: JSONObject
- lessonInfo: JSONObject
- additionalInfo: ContractAdditionalInfo
- firstParty: UserModel
- secondParty: UserModel
- subject: SubjectModel
+ getSubjectAndCompetencyStr(): String
+ toString(): String
MessageModel
- id: String
- bidId: String
- datePosted: String
- dateLastEdited: String
- content: String
- additionalInfo: JSONObject
- recipientId: String
- poster: UserModel
+ toString(): String
ChatFragment
- recylerView: RecylerView
- prefs: SharedPreferences
- buttonSendChat: Button
- editTextMessage: EditText
- currentBidId: String
- recipientName: String
- recipientId: String
+ onCreateView(LayoutInflater, ViewGroup , Bundle): View
+ getMessages(): void
+ postMessage():void
BidOfferModel
- competency: Integer
- rate: Integer
- hoursPerLesson: Integer
- lessonsPerWeek: Interger
- freeClasses: Integer
- isRateHourly: Boolean
- isRateWeekly: Boolean
- isTutorBid: Boolean
- bidOffer: JSONObject
+ parseIntoJSON(): JSONObject
+ getRateStr(): String
+ getFreeClassesStr(): String
+ getHoursPerLessonStr(): String
+ getLessonsPerWeek()Str: String
+ getCompetencyStr():String
+ getCompetencyResource(): Integer
+ toString(): String
BidModel
- id: String
- type: String
- dateCreated: String
- dateClosedDown: String
- initiator: UserModel
- subject: SubjectModel
- additionalInfo: BidAdditionalInfoModel
+ getSubjectAndCompetencyStr(): String
+ toString() : String

FIT3077 Assignment 2 Design Rationale

Written by Team Ouroboros

Architectural Pattern

The software architecture of this project closely follows the Model-View-Presenter (MVP) design pattern. This architecture makes use of data containers in model, which interacts with view (layout resources) through a middleman known as presenter, which essentially controls all activities when the model (data) wants to communicate with view (UI) and vice versa.

As the actual implementation is done using native Android for mobile applications, the most popular and efficient approaches are either MVP or MVVM (Model-View-ViewModel) patterns. Currently, MVP is more preferred by the Android community compared to other patterns [1].

Nevertheless, this architecture is very suitable for the scope of the project. Since there are numerous tasks within one functionality (e.g. bids), it is difficult to implement functionalities in large chunks, which will inevitably create bloated and 'god' classes when each task is not modularized into smaller groups. The MVP design pattern is perfect to isolate specific tasks within a functionality by enforcing **one-to-one mapping of Presenter to View**[2].

This supports the idea that each presenter (adapter) only have to deal with one view (fragment) to allow setting/getting data, and perform actions on the UI efficiently, by allowing all logic for a specific task to be handled by one presenter instead of having multiple tasks carried out by the same presenter, or communicating with multiple views from one presenter.

The only potential drawback of the architecture is high complexity due to modularizing each task. When the codebase is extended with new features, it might be difficult to track and manage a large number of fragments and adapters.

Design Principles

Single Responsibility Principle (SRP)

The modularity of the MVP pattern allows smaller classes which have their own unique purpose; adding logic to only one component, the presenter, prevents multiple conflicts by making code testing easier compared to other design patterns. This allows the system to have less bloated classes and reduces redundancy of logic across all presenters.

Open/Close Principle (OCP)

The MVP architecture always provides information on which abstractions are extensively useful [3]. This means that SOLID principles can be easily applied, especially OCP. As modularity and isolation of responsibility is strictly followed in MVP, we can ensure that the new functionalities can be easily extended by simply creating new presenters and views for multiple tasks which together introduce the new extension. As for existing presenters and views, no change is needed when adding new features.

References

[1] Vogel, K., (2020). Android Architecture with MVP or MVVM - Tutorial. Retrieved from this website.

[2] Phan, M., (2019). MVP architectural pattern. Retrieved from https://ducmanhphan.github.io/2019-08-05-MVP-architectural-pattern/

[3] Almeida, J., (2018). SOLID Networking: The Open / Closed Principle. Retrieved from https://medium.com/@codavel/solid-networking-the-open-closed-principle-343af2f57406