Bilkent University

Department of Computer Engineering

Senior Design Project

Low-Level Design Report

Project Name: BookClub

Group Members: Bikem Çamli
Mert Osman Dönmezyürek
Barış Eymür
Mahin Khankishizade
Deniz Şen

Supervisor: Assoc. Prof. Selim Aksoy

Jury Members: Prof. Dr. Fazlı Can
Assoc. Prof. Cigdem Gunduz Demir

Innovation Expert: Dr. Haluk Altunel

Low-Level Design Report

Feb 13, 2019
This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2

Table of Contents

1. Introduction

Reading is a fun and useful activity: we can learn new things by reading a scientific book or we can find ourselves in a totally new fictional world by reading a novel. Unfortunately, every beautiful thing comes with a price. Wouldn't it be nice to exchange our books with our friends' books, instead of paying? If we did that, we would have lessened the price of reading significantly. There are groups of people around the world who have noticed the advantages of book sharing. These people have formed book sharing groups, called book clubs.

The main aim of this project is to create a book trading platform which will serve as a universal "BookClub". The users of this mobile app will be able to exchange their used books with books of other users and they will also be able to buy second-hand books from other users by paying a reasonable price. The users will specify the books that they desire to give away or sell and the ones they want to read. Using this information, BookClub will match users with the owners of books they are looking for. While doing this, it will find the best possible match for a user by considering user locations, user ratings, book prices and some other attributes. Additionally, it will make personal suggestions to its users by trying to understand their taste of books.

In this report, you will find the design trade-offs in several subsections, followed with the high-level explanations of the packages that will be put into both the client and the server-side of the system. Then, the class interfaces will be given and explained in detail.

1.1 Design Trade-Offs

In the design phase of a software project, there are almost always some trade-offs. When facing a trade-off, the designers of the program usually decide to concentrate on a specific feature of a program and design the system accordingly. It usually causes a loss of importance and less consideration of another feature. In the design of BookClub, we faced with some trade-offs and made the following decisions about them.

1.1.1 Extensibility vs Time Efficiency

BookClub will be taking the users' comments and reviews into consideration to improve and shape the system accordingly. In addition, the application will constantly get updates and additional features to keep up with the users' wishes and new trends. This will make our system extensible, however these features will be time-consuming for us. Nevertheless, since user satisfaction is important for us time efficiency can be discarded.

1.1.2 Scalability vs Speed

We wanted BookClub to be scalable. The aim of the implementation is to have fewer changes in the algorithm and core design of the platform when there is a significant number of users. However, to make this possible, we made some sacrifices in the sense of speed. While implementing the server side, we will not make speed our primary concern, i.e., there will not be speed handler packages or related classes in the server-side. Since these packages would consume more time and budget than we estimate, we decided to sacrifice it for the scalability. The algorithm will still work, however the speed will decrease.

1.1.3 Reliability vs Cost

The cost will be analyzed in two parts: reliability vs time cost and reliability vs user cost. In order for the application to be more reliable, it needs to be implemented in a long time because the server and client side both need to be secure and reliable. When it comes to the user cost, since this platform prevents the black market of the books, the users who own non-original books will not want to use it, thus the platform will lose many users because of its reliability.

1.1.4 Adaptability vs Complexity

In our system, the client and server sides work separately and they are connected via RestAPI. While we are writing our client side in Java, we are implementing the server side in Python. With the help of this feature, we can easily adapt our system to different environments. However, designing and implementing our system in this approach increased the complexity of our project. Since adaptability is an important feature that enables our system to reach more users, we thought that an increase in complexity can be ignored.

1.2 Interface Documentation Guidelines

In our report, we named our classes using camel case format, such as "ClassName". In our method and variable names, we followed the same format. However, for variables which denote id, we did not follow the camel case format. We named them by separating words with a "_", such as "variable_id". The reason for us to do this is, Python creates these id variables by following this convention. For simplicity purposes, we wanted our variables to be named in the same way as Python creates them.

We described the details of our classes by generating tables. In the tables, there are names, attributes and methods of each class.

1.3 Engineering Standards

To explain our system design, we generated diagrams. While generating these diagrams, we followed Unified Modeling Language (UML) guidelines. We chose UML because all of the team members were proficient in it. We also used UML to describe use cases, subsystem decomposition and class interfaces of BookClub. For the references, we followed the citation standards of IEEE.

1.3 Use of New Tools and Technologies

- RestAPI - we will implement the API between client and server with the help of RestAPI. All the requests will be handled via this API.
- Django Python Framework - we will use this framework in the server-side because it is very easily adaptable to MVC pattern and easily controls the model side with its controllers. Additionally, since we have machine learning (ML) algorithms in the AlgorithmPackage, Python comes very handy with its ML libraries.
- JSON Data - we will use this type of data for conversating between client and server. This kind of data comes very handy and works perfectly with RestAPI.

1.4 Definitions, acronyms and abbreviations

API:	Application Programming Interface
MVC:	Model View Controller
UML:	Unified Modeling Language
ML:	Machine Learning
SMTP:	Simple Mail Transfer Protocol
XML:	Extensible Markup Language

2. Packages

2.1 Client

Following page contains the package diagram of the client-side.

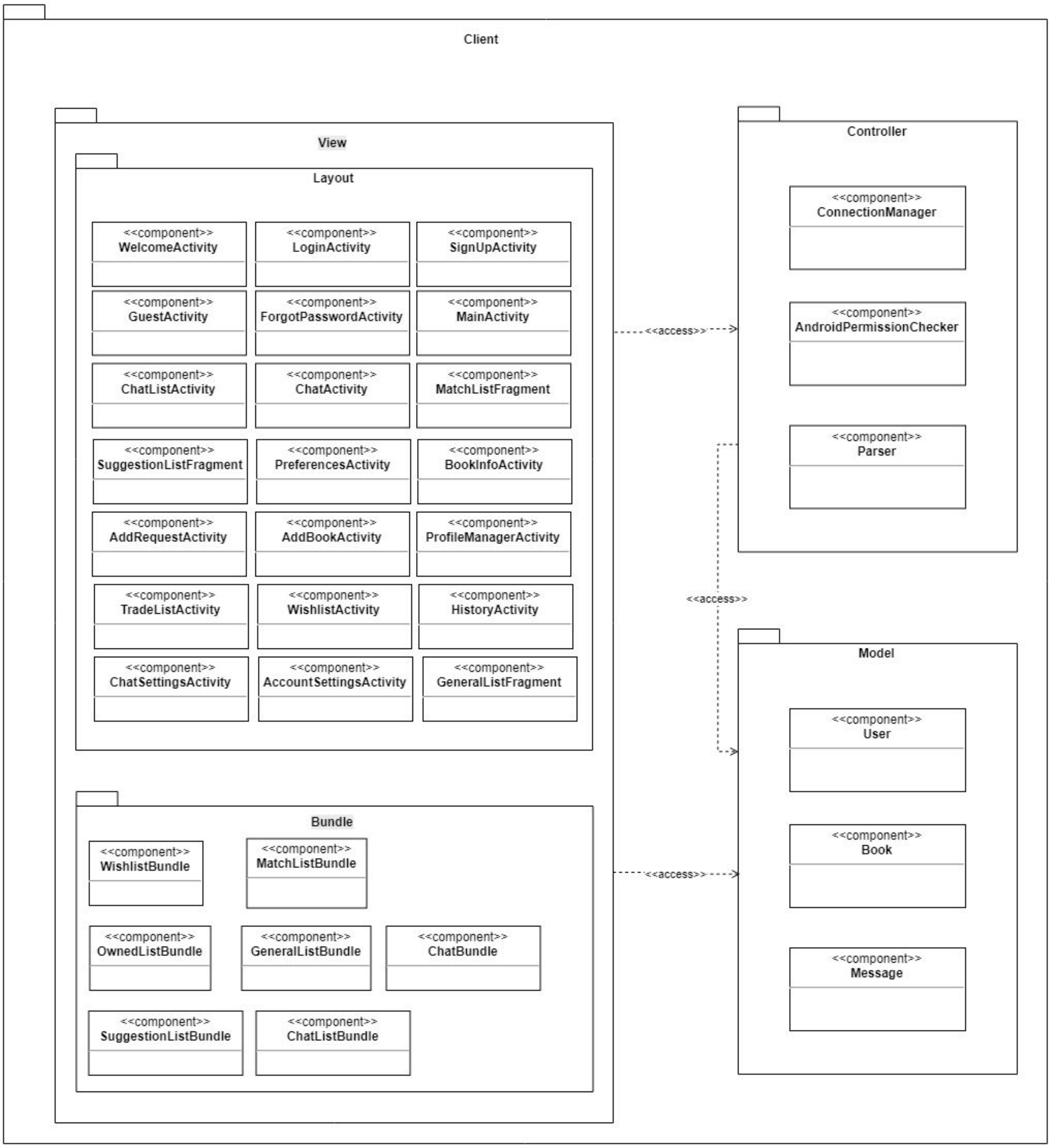

Client
View
Layout
<<component>> WelcomeActivity
<<component>> LoginActivity
<<component>> SignUpActivity
<<component>> GuestActivity
<<component>> ForgotPasswordActivity
<<component>> MainActivity
<<component>> ChatListActivity
<<component>> ChatActivity
<<component>> MatchListFragment
<<component>> SuggestionListFragment
<<component>> PreferencesActivity
<<component>> BookInfoActivity
<<component>> AddRequestActivity
<<component>> AddBookActivity
<<component>> ProfileManagerActivity
<<component>> TradeListActivity
<<component>> WishlistActivity
<<component>> HistoryActivity
<<component>> ChatSettingsActivity
<<component>> AccountSettingsActivity
<<component>> GeneralListFragment
Bundle
<<component>> WishlistBundle
<<component>> MatchListBundle
<<component>> OwnedListBundle
<<component>> GeneralListBundle
<<component>> ChatBundle
<<component>> SuggestionListBundle
<<component>> ChatListBundle
Controller
<<component>> ConnectionManager
<<component>> AndroidPermissionChecker
<<component>> Parser
Model
<<component>> User
<<component>> Book
<<component>> Message
<<access>>
<<access>>
<<access>>

Figure: Package Diagram of Client-Side

2.1.1 View

2.1.1.1 Layout

WelcomeActivity: This class is the first screen that is shown to the user when the application is first launched. The user can then proceed to login or sign up to the system.

LoginActivity: This class is the representation of the screen where the user can enter their credentials and go into the system.

ForgotPasswordActivity: When the user forgets their password, they can use this screen to request a new password email from the system.

GuestActivity: This screen holds 2 pages for the guest user to sign up to the system, and 1 page for them to see the listed books.

SignUpActivity: This screen is for the new users to create an account by giving personal information.

MainActivity: This screen holds the suggestion list, match list and the listed books. For simplicity, these pages are designed to be horizontally scrollable.

ChatListActivity: This screen shows the ongoing and past chats of the user from where they can proceed to continue or start a chat.

ChatActivity: The user and a matched user can chat from this screen thanks to a quite familiar and simple massaging layout.

MatchListFragment: This segment shows the matches that the user is involved with.

SuggestionListFragment: This segment shows the suggested listed books to the user. From here, the user can agree or disagree with a suggestion.

PreferencesActivity: This screen is a navigation list from where the user can proceed to change several kinds of preferences which can be seen in the following sections.

HistoryActivity: This screen shows a list of transactions that the user has made recently.

ChatSettingsActivity: Here, the user can block or unblock a user with whom they have chatted recently.

TradeListActivity: The user can add and remove books that they are willing to use as a trade material in this section.

AccountSettingsActivity: In this section, the user can change personal and account related information.

WishlistActivity: The user sees and manipulates their wished books in this screen.

GeneralListFragment: This section shows the currently listed books in the system regardless of the user's preferences and likings.

BookInfoActivity: This screen shows information and a brief description of a particular book.

AddBookActivity: This screen allows the user to put a new book to their wishlist or tradable book list.

2.1.1.2 Bundle

WishlistBundle: This object is a simple bundle to hold information related to the wishlist list items.

MatchListBundle: This object is a simple bundle to hold information related to the match list's list items.

OwnedListBundle: This object is a simple bundle to hold information related to the lists of owned books list items.

SuggestionListBundle: This object is a simple bundle to hold information related to the suggestion list items.

GeneralListBundle: This object is a simple bundle to hold information related to the general list items.

ChatListBundle: This object is a simple bundle to hold information related to the chatted user list’s list items.

ChatBundle: This object is a simple bundle to hold information related to the messages of a chat that will be printed to a list view.

2.1.2 Controller

ConnectionManager: This class is meant to help the view package. It sends requests to the server and receives responses from the server via RESTful API over HTTP. Its methods can give feedback if the process is done or return the data got from the server in a proper format on the client-side.

Parser: This class eases the conversion of the JSON strings to specific objects and specific objects to JSON strings.

AndroidPermissionChecker: This class checks the required permissions of the application such as the Internet connection and GPS.

2.1.3 Model

User: This class is an object to hold the user’s personal info and account related data.

Book: This class is an object to hold a particular book’s regular and owner related information.

Message: This class maintains the information of each message sent or received during a conversation led through the chat service.

2.2 Server

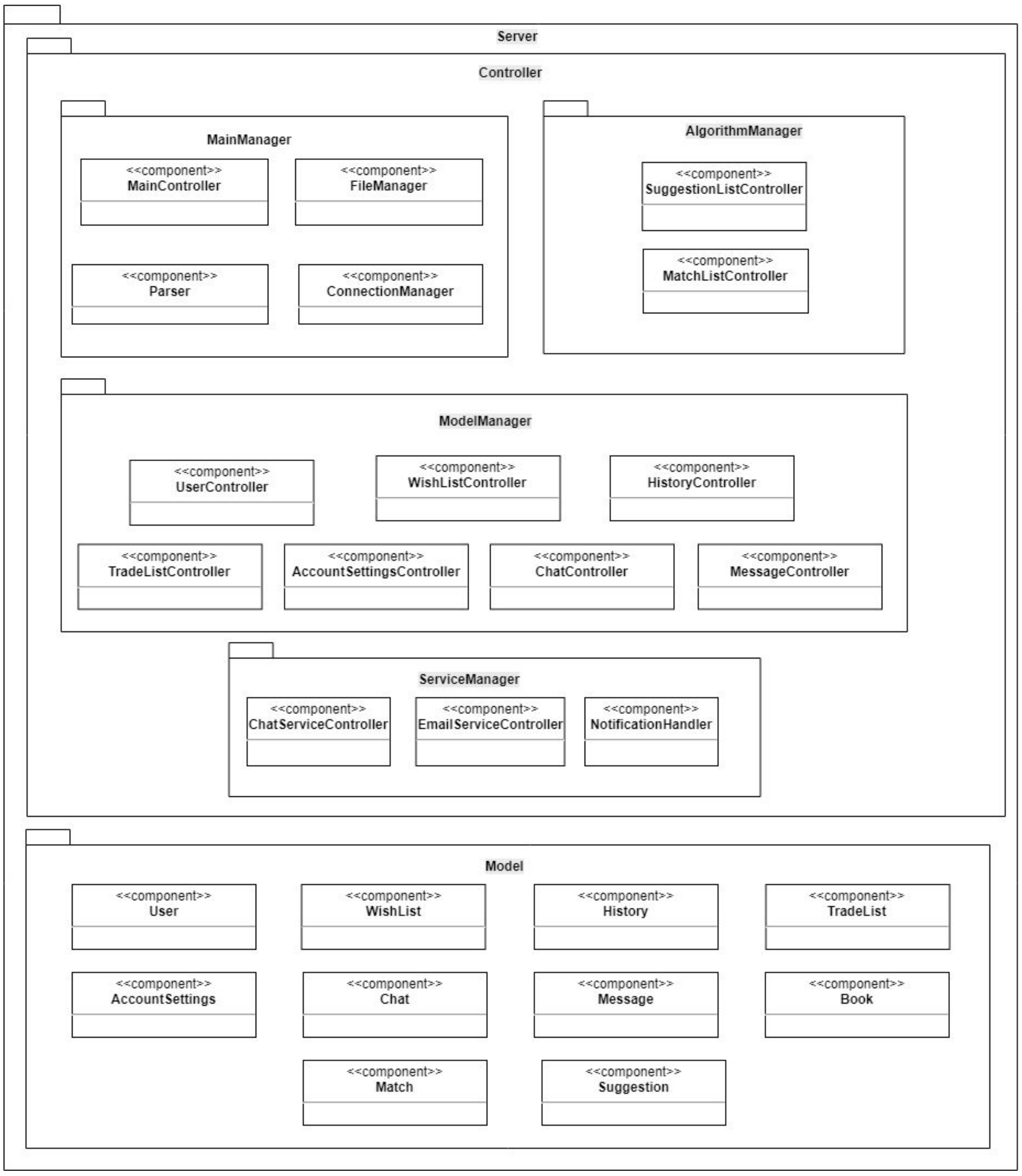

Server
Controller
MainManager
<<component>> MainController
<<component>> FileManager
<<component>> Parser
<<component>> ConnectionManager
AlgorithmManager
<<component>> SuggestionListController
<<component>> MatchListController
ModelManager
<<component>> UserController
<<component>> WishListController
<<component>> HistoryController
<<component>> TradeListController
<<component>> AccountSettingsController
<<component>> ChatController
<<component>> MessageController
ServiceManager
<<component>> ChatServiceController
<<component>> EmailServiceController
<<component>> NotificationHandler
Model
<<component>> User
<<component>> WishList
<<component>> History
<<component>> TradeList
<<component>> AccountSettings
<<component>> Chat
<<component>> Message
<<component>> Book
<<component>> Match
<<component>> Suggestion

Figure 2: Package Diagram of the Server

The responsibility of the server side of the system is to handle different clients, get requests from them, keep data of BookClub users and the application, and manipulate those data by using different components in the server. Server package will be implemented in Python Django on a different

machine (a server) to execute the whole package and keep the data of the application.
The server-side of the system is mainly grouped into two sections as model and controller packages. Controller package itself is divided into four main packages such as MainManager, AlgorithmManager, ServiceManager and ModelManager. Model package includes the classes dealing with the database in it. These packages are deeply analyzed in this section.

2.2.1 Model

Model part of the server is responsible for migrating the database to the application as a class. This way, the controllers will easily modify them within the server. These classes are User, Book, Chat, Message, History, Wishlist, AccountSettings, TradeList, Match and Suggestion classes. These classes do not do anything except literally being a class of a particular table. In other words, these classes will basically be the attributes of a table (columns) in the server.

2.2.2 Controller

Controller part of the server is responsible for coordinating the user input between Model and View parts of the application.

2.2.2.1 MainManager

This package is the main package that includes in itself the classes that are playing the core role of receiver and sender within the server-side. These classes are MainController, FileManager, Parser and ConnectionManager.

MainController: This class is the main class, as it is obvious from its name. To be more exact, when MainController receives clear data from Parser or ConnectionManager, the first thing it checks is the action key. For example, if the action key is “login”, MainController understands the request and knows how to handle it. In this case, it would send the data to the UserController class so that it would create a new user in the database. Later UserController sends the confirmation data to the MainController. After confirmation, MainController notifies ConnectionManager.

ConnectionManager: This class is basically a notifier or a notification handler of the server-side. It deals with the client-side's ConnectionManager. When the client-side requests data, the ConnectionManager class gets a notification and receives the data from the client. Later, it sends the data to the Parser and returns the defined data to MainController so that it could spread it to the relevant classes.

FileManager: The purpose of this class is managing important files of the system such as user profile photos, images of books, etc.

Parser: This class plays the second main role in the MainManager package. Namely, other classes are using its parse method for defining the attributes and action keys of the JSON data received from the client-side of the application. With the help of the parse method, Parser class analyzes the data and sends it back to the classes that requested it.

2.2.2.2 ServiceManager

This package is a small-service package that includes in itself the classes that handle small services within the server-side. These classes are EmailServiceController, NotificationHandler and ChatServiceController.

EmailServiceController: This class handles the email service. In other words, when the application needs to notify a user within the email, this class configures the relevant SMTP ports and sends an email to the user effortlessly.

NotificationHandler: This class sends a user notification when something happens within the application. For example, if a user gets a match from another user, NotificationHandler sends a notification to the user via the app. In other words, the user gets a screen notification on his/her phone.

ChatServiceController: This class handles a chat between two users after the match is confirmed from both sides. It opens a new stream between the users and uses web socket to handle the messages. This way, the server does not load the database.

2.2.2.3 AlgorithmManager

This package is responsible for managing the execution of the core algorithm of BookClub. It performs this task by updating match lists and suggestion lists of users according to the results of the algorithm. It contains classes MatchListController and SuggestionListController.

MatchListController: This class is responsible for finding new matches for a user and updating the MatchList of a user accordingly. It does this by running the special match algorithm of BookClub.

SuggestionListController: This class is responsible for creating book suggestions for a user and updating the SuggestionList of a user accordingly.

2.2.2.4 ModelManager

This package is basically a controller package for all the classes of the model package. It has several controllers that handle the API requests of the particular model classes. Since all of them do the same thing, their main goal will be generally explained in this section. These controllers handle main four types of requests (some of the controllers have exceptions, the details are given in the further sections). These requests are POST, GET, DELETE and PUT.

1. When the request is POST, the controller (for example) signs a user up or adds a new item to the database. Thus, POST request is basically for adding or updating the items in the database. These actions are also possible with the PUT requests.
2. When the request is DELETE, the controller deletes an item from the database. For example, when the user wants to delete a book from the wishlist/tradelist or wants to delete a particular conversation from the chat list, this request is used.

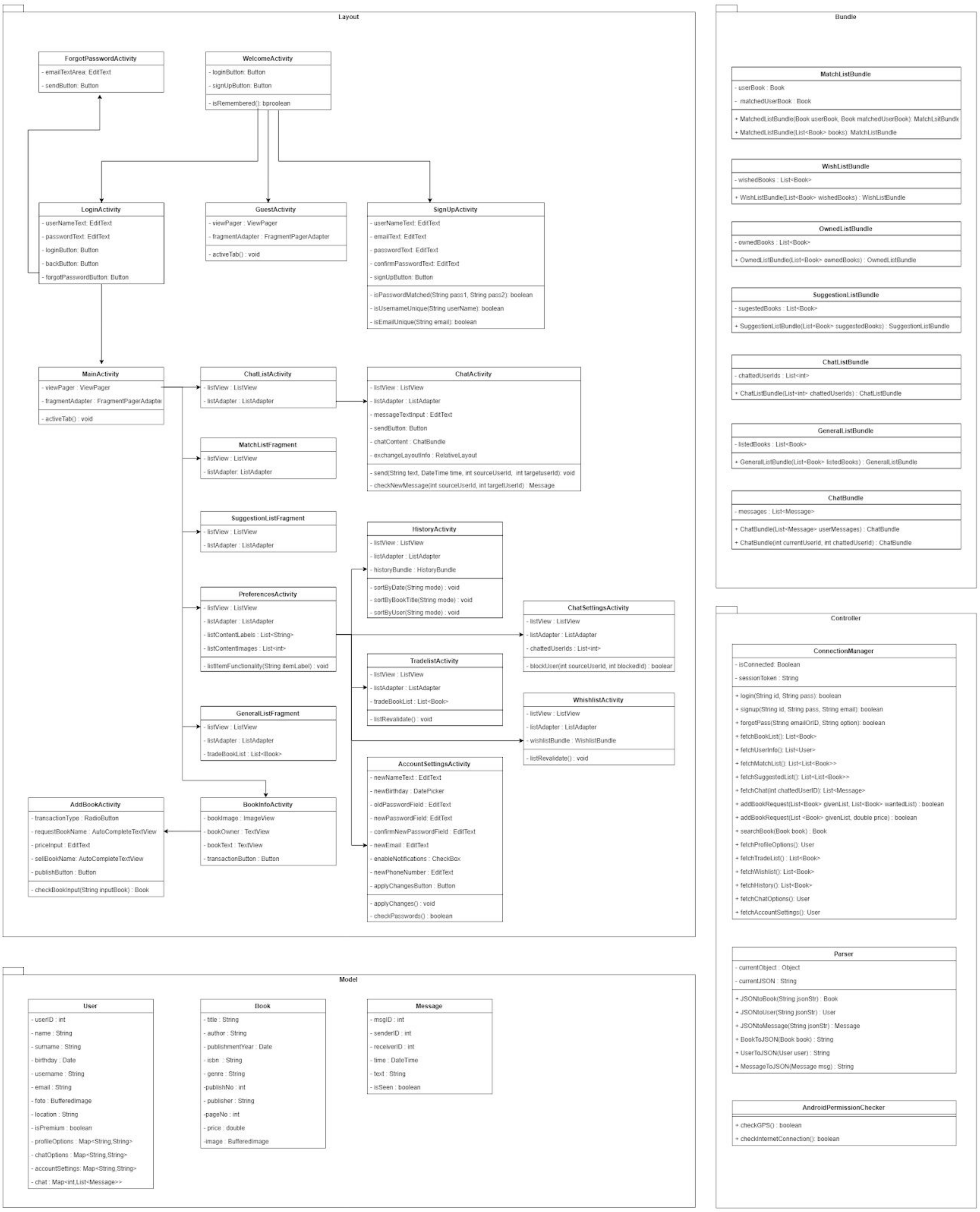
3. When the request is GET, the controller returns a data with the information from the database. It is almost the most used request in the application. For example, when the user wants to see a book's information the GET request is sent to the controller and the information of the book is returned as the JSON data.

3. Class Interfaces

Please note that the class interfaces do not include getter and setter methods since they all will be implemented by default for each class. Also as each object will have a default constructor, for the sake of simplicity across the report layout, these constructors will not be included inside the class interfaces. However, constructors with different than default parameters will be included.

3.1 Client

Following page contains the class diagram of the client-side.
(Figure : Class diagram of Client-Side)



Layout
ForgotPasswordActivity
- emailTextArea: EditText
- sendButton: Button
WelcomeActivity
- loginButton: Button
- signUpButton: Button
- isRemembered(): bproolean
LoginActivity
- userNameText: EditText
- passwordText: EditText
- loginButton: Button
- backButton: Button
- forgotPasswordButton: Button
GuestActivity
- viewPager : ViewPager
- fragmentAdapter : FragmentPagerAdapter
- activeTab() : void
SignUpActivity
- userNameText: EditText
- emailText: EditText
- passwordText: EditText
- confirmPasswordText: EditText
- signUpButton: Button
- isPasswordMatched(String pass1, String pass2): boolean
- isUsernameUnique(String userName): boolean
- isEmailUnique(String email): boolean
MainActivity
- viewPager : ViewPager
- fragmentAdapter : FragmentPagerAdapter
- activeTab() : void
ChatListActivity
- listView : ListView
- listAdapter : ListAdapter
ChatActivity
- listView : ListView
- listAdapter : ListAdapter
- messageTextInput : EditText
- sendButton: Button
- chatContent : ChatBundle
- exchangeLayoutInfo : RelativeLayout
- send(String text, DateTime time, int sourceUserId, int targetuserId): void
- checkNewMessage(int sourceUserId, int targetUserId) : Message
MatchListFragment
- listView : ListView
- listAdapter: ListAdapter
SuggestionListFragment
- listView : ListView
- listAdapter : ListAdapter
HistoryActivity
- listView : ListView
- listAdapter : ListAdapter
- historyBundle : HistoryBundle
- sortByDate(String mode) : void
- sortByBookTitle(String mode) : void
- sortByUser(String mode) : void
PreferencesActivity
- listView : ListView
- listAdapter : ListAdapter
- listContentLabels : List<String>
- listContentImages : List<int>
- listItemFunctionality(String itemLabel) : void
ChatSettingsActivity
- listView : ListView
- listAdapter : ListAdapter
- chattedUserIds : List<int>
- blockUser(int sourceUserId, int blockedId) : boolean
TradelistActivity
- listView : ListView
- listAdapter : ListAdapter
- tradeBookList : List<Book>
- listRevalidate() : void
WhishlistActivity
- listView : ListView
- listAdapter : ListAdapter
- wishlistBundle : WishlistBundle
- listRevalidate() : void
GeneralListFragment
- listView : ListView
- listAdapter : ListAdapter
- tradeBookList : List<Book>
AccountSettingsActivity
- newNameText : EditText
- newBirthday : DatePicker
- oldPasswordField : EditText
- newPasswordField: EditText
- confirmNewPasswordField : EditText
- newEmail : EditText
- enableNotifications : CheckBox
- newPhoneNumber : EditText
- applyChangesButton : Button
- applyChanges() : void
- checkPasswords() : boolean
AddBookActivity
- transactionType : RadioButton
- requestBookName : AutoCompleteTextView
- priceInput : EditText
- sellBookName: AutoCompleteTextView
- publishButton : Button
- checkBookInput(String inputBook) : Book
BookInfoActivity
- bookImage : ImageView
- bookOwner : TextView
- bookText : TextView
- transactionButton : Button
Bundle
MatchListBundle
- userBook : Book
- matchedUserBook : Book
+ MatchedListBundle(Book userBook, Book matchedUserBook): MatchLsitBundle
+ MatchedListBundle(List<Book> books): MatchListBundle
WishListBundle
- wishedBooks : List<Book>
+ WishListBundle(List<Book> wishedBooks) : WishListBundle
OwnedListBundle
- ownedBooks : List<Book>
+ OwnedListBundle(List<Book> ownedBooks) : OwnedListBundle
SuggestionListBundle
- sugestedBooks : List<Book>
+ SuggestionListBundle(List<Book> suggestedBooks) : SuggestionListBundle
ChatListBundle
- chattedUserIds : List<int>
+ ChatListBundle(List<int> chattedUserIds) : ChatListBundle
GeneralListBundle
- listedBooks : List<Book>
+ GeneralListBundle(List<Book> listedBooks) : GeneralListBundle
ChatBundle
- messages : List<Message>
+ ChatBundle(List<Message> userMessages) : ChatBundle
+ ChatBundle(int currentUserId, int chattedUserId) : ChatBundle
Controller
ConnectionManager
- isConnected: Boolean
- sessionToken : String
+ login(String id, String pass): boolean
+ signup(String id, String pass, String email): boolean
+ forgotPass(String emailOrID, String option): boolean
+ fetchBookList(): List<Book>
+ fetchUserInfo(): List<User>
+ fetchMatchList(): List<List<Book>>
+ fetchSuggestedList(): List<List<Book>>
+ fetchChat(int chattedUserID): List<Message>
+ addBookRequest(List<Book> givenList, List<Book> wantedList) : boolean
+ addBookRequest(List <Book> givenList, double price) : boolean
+ searchBook(Book book) : Book
+ fetchProfileOptions(): User
+ fetchTradeList() : List<Book>
+ fetchWishlist(): List<Book>
+ fetchHistory(): List<Book>
+ fetchChatOptions(): User
+ fetchAccountSettings(): User
Parser
- currentObject : Object
- currentJSON : String
+ JSONtoBook(String jsonStr) : Book
+ JSONtoUser(String jsonStr) : User
+ JSONtoMessage(String jsonStr) : Message
+ BookToJSON(Book book) : String
+ UserToJSON(User user) : String
+ MessageToJSON(Message msg) : String
AndroidPermissionChecker
+ checkGPS() : boolean
+ checkInternetConnection(): boolean
Model
User
- userID : int
- name : String
- surname : String
- birthday : Date
- username : String
- email : String
- foto : BufferedImage
- location : String
- isPremium : boolean
- profileOptions : Map<String,String>
- chatOptions : Map<String,String>
- accountSettings: Map<String,String>
- chat : Map<int,List<Message>>
Book
- title : String
- author : String
- publishmentYear : Date
- isbn : String
- genre : String
-publishNo : int
- publisher : String
-pageNo : int
- price : double
-image : BufferedImage
Message
- msgID : int
- senderID : int
- receiverID : int
- time : DateTime
- text : String
- isSeen : boolean

3.1.1 View

3.1.1.1 Bundle Package

3.1.1.1.1 WishlistBundle Class

Class: WishlistBundle
This class is used to hold the data for customized wishlist list items.
Attributes:
private List<Book> wishedBooks: this attribute will store the books that are in the wishlist of the user
Functions:

3.1.1.1.2 MatchListBundle Class

Class: MatchListBundle
This class is used to hold the data for customized matchlist list items.
Attributes:
private Book userBook: this attribute will hold the book of the current user to be written to the list item. **private Book matchedUserBook:** this attribute will hold the book of the matched user to be written to the list item.
Functions:
public MatchedListBundle(List<Book> books): constructor that converts the list of books to two different Book objects.

3.1.1.1.3 OwnedListBundle Class

Class: OwnedListBundle
This class is used to hold the data for customized owned books' list's list items.
Attributes:

private List<Book> ownedBooks: Holds the books which are owned by the user.
Functions:

3.1.1.1.4 SuggestionListBundle Class

Class: SuggestionListBundle
This class is used to hold the data for customized suggestion lists’ list items.
Attributes:
private List<Book> suggestedBooks: This attribute holds the books that are suggested to the user by the program.
Functions:

3.1.1.1.5 GeneralListBundle Class

Class: GeneralListBundle
This class is used to hold the data for customized general book lists’ list items.
Attributes:
private List<Book> listedBooks : holds the books that are put into trade and can be seen by guest users.
Functions:

3.1.1.1.6 ChatListBundle Class

Class: ChatListBundle
This class is used to hold the data for customized chat lists’ list items.
Attributes:

private List<int> chattedUserIds: holds the user ids whom the user can chat with.
Functions:

3.1.1.1.7 ChatBundle Class

Class: ChatBundle
This class is used to hold the data for customized message lists' list items.
Attributes:
private List<Message> messages: holds the messages of a conversation.
Functions:
public ChatBundle(int currentUserId, int chattedUserId): constructor that gets the message list using the user ids of two sides of the conversation.

3.1.1.2 Layout Package

Please note that each activity has an onCreate() method by default and neither of the activities mentioned in the following section will contain an explanation regarding this method. The arguments of the onCreate() methods will not be changed. Also, the functionalities of the screen items (views) will be given inside the onCreate() methods using anonymous inner classes.

3.1.1.2.1 WelcomeActivity Class

Class: WelcomeActivity
This activity is the Java code of the welcome screen, working along with the XML file welcome_activity.xml. This activity is the Android manifest's main activity.
Attributes:
private Button loginButton : object of the button which brings login screen.

private Button signUpButton : the object of the button which brings the signup page.
Functions:
private boolean isRemembered() : gathers the user credentials from the device and automatically authenticates if it finds valid credentials.

3.1.1.2.2 LoginActivity Class

Class: LoginActivity
This activity is the Java code of the login screen, working along with the XML file login_activity.xml.
Attributes:
private EditText userNameText : takes the username as input **private EditText passwordText :** takes password as input while showing dots only for the characters. **private Button loginButton :** starts the authentication process. **private Button backButton :** takes the screen back to welcome screen. **private Button forgotPasswordButton :** brings the forgot password screen.
Functions:

3.1.1.2.3 ForgotPasswordActivity Class

Class: ForgotPasswordActivity
This activity is the Java code of the forgot password screen, working along with the XML file forgot_password_activity.xml.
Attributes:
private EditText emailTextArea : takes input for the email address that the system will send a new password request mail. **private Button sendButton :** starts the password change procedure.
Functions:

3.1.1.2.4 GuestActivity Class

Class: GuestActivity
This activity is the Java code of the guest screen, working along with the XML file guest_activity.xml.
Attributes:
private ViewPager viewPager : holds the horizontally scrollable fragments that are general list, a preview of a suggestion list and preview of a match list which will take the user to the signup screen. **private FragmentPagerAdapter fragmentAdapter :** adapts the fragments into the ViewPager object.
Functions:
private void activeTab(): changes the currently shown fragment. Called inside the FragmentPagerAdapter.

3.1.1.2.5 SignUpActivity Class

Class: SignUpActivity
This activity is the Java code of the signup screen, working along with the XML file sign_up_activity.xml.
Attributes:
private EditText userNameText : takes input for the username. **private EditText emailText:** takes input for the email address. **private EditText passwordText:** takes password input. **private EditText confirmPasswordText:** takes the confirming password input. **private Button signUpButton:** starts the signup procedure.
Functions:
private boolean isPasswordMatched(String pass1, String pass2) : every time the user types in a character for the confirming password, this method controls if two password inputs match. **private boolean isUsernameUnique(String userName) :** every time the user types in a character for the username, this function checks if the current username is unique. **private boolean isEmailUnique(String email) :** every time the user types in a character for the email address, this function checks if the current entry is unique.

3.1.1.2.6 MainActivity Class

Class: MainActivity
This activity is the Java code of the main screen, working along with the XML file main_activity.xml. Note that this activity is not the main activity in terms of Android manifest entries.
Attributes:
private ViewPager viewPager : holds the swipable fragments that are general list, suggestion list and match list. **private FragmentPagerAdapter fragmentAdapter :** adapts the fragments into the ViewPager object.
Functions:
private void activeTab(): changes the currently shown fragment. Called inside the FragmentPagerAdapter.

3.1.1.2.7 ChatListActivity Class

Class: ChatListActivity
This activity is the Java code of the chat list screen, working along with the XML file chat_list_activity.xml.
Attributes:
private ListView listView : scrollable list that contains the chats of the user. **private ListAdapter listadapter :** adapts the data of the list items into the list item views and gives them particular functionalities.
Functions:

3.1.1.2.8 ChatActivity Class

Class: ChatActivity
This activity is the Java code of the chat screen, working along with the XML file chat_activity.xml.

Attributes:
private ListView listView : scrollable list that holds the messages **private ListAdapter listAdapter:** adapts the messages into the list view. **private EditText messageTextInput:** takes message input. **private Button sendButton:** sends the content of messageTextInput view. **private ChatBundle chatContent :** holds the current chat content to not occupy the server. **private RelativeLayout exchangeLayoutInfo :** holds the content of the chat's subject such as the books that are being traded. **private Thread messageChecker :** thread to check if there are new messages every 5 seconds.
Functions:
private void send(String text, DateTime time, int sourceUserId, int targetUserId) : sends the message to the user. **private Message checkNewMessages(int sourceUserId, int targetUserId) :** gathers new messages from the server. It is the main function of a worker thread.

3.1.1.2.9 MatchListFragment Class

Class: MatchListFragment
This fragment is the Java code of the match list part of the scrollable view pager, working along with the XML file match_list_fragment.xml.
Attributes:
private ListView listView : scrollable list that holds the matches. **private ListAdapter listAdapter:** adapts the match list items into the list view.
Functions:

3.1.1.2.10 SuggestionListFragment Class

Class: SuggestionListFragment
This fragment is the Java code of the suggestion list part of the scrollable view pager, working along with the XML file suggestion_list_fragment.xml.
Attributes:

private ListView listView : scrollable list that holds the suggestions. **private ListAdapter listAdapter:** adapts the suggestion list items into the list view.
Functions:

3.1.1.2.11 PreferencesActivity Class

Class: PreferencesActivity
This activity is the Java code of the preferences screen, working along with the XML file preferences_activity.xml.
Attributes:
private ListView listView : scrollable list that holds the preference tab items. **private ListAdapter listAdapter:** adapts the preference list items into the list view. **private List<String> listContentLabel :** contains the list item labels of the preferences list. **private List<int> listContentImages :** contains the integer representations of the icons of the list images.
Functions:
private void listItemFunctionality(String itemLabel) : uses intent to change screen based on which item is pressed.

3.1.1.2.12 HistoryActivity Class

Class: HistoryActivity
This activity is the Java code of the history screen, working along with the XML file history_activity.xml.
Attributes:
private ListView listView : scrollable list that holds the history. **private ListAdapter listAdapter:** adapts the history list items into the list view. **private HistoryBundle historyBundle :** holds the history information of the user.

Functions:
private void sortByDate(String mode) : sorts the history list based on their dates and the parameter mode which can either be “ascending” or “descending”. **private void sortByBookTitle(String mode) :** sorts the history list based on their book titles and the parameter mode which can either be “ascending” or “descending”. **private void sortByUser(String mode) :** sorts the history list based on their user names and the parameter mode which can either be “ascending” or “descending”.

3.1.1.2.13 ChatSettingsActivity Class

Class: ChatSettingsActivity
This activity is the Java code of the chat settings screen, working along with the XML file chat_settings_activity.xml.
Attributes:
private ListView listView : scrollable list that holds the chatted users. **private ListAdapter listAdapter:** adapts the user list items into the list view. **private List<int> chattedUserIds :** holds the ids of the users with whom the current user has chatted at some point.
Functions:
private boolean bloackUser(int sourceUserId, int blockedUserId): blocks the second argument user from the chat list of the first argument user.

3.1.1.2.14 TradeListActivity Class

Class: TradeListActivity
This activity is the Java code of the trade list screen, working along with the XML file trade_list_activity.xml.
Attributes:
private ListView listView : scrollable list that holds the tradable books. **private ListAdapter listAdapter:** adapts the tradable books’ list items into the list view. **private List<Book> tradeBookList:** holds the tradable books.

Functions:
private void listRevalidate(): reconstructs the tradable book list after adding a new book to the list.

3.1.1.2.15 AccountSettingsActivity Class

Class: AccountSettingsActivity
This activity is the Java code of the account settings screen, working along with the XML file account_settings_activity.xml.
Attributes:
private EditText newNameText: input for the personal name. **private DatePicker newBirthday:** date picker to select a new birthday. **private EditText oldPasswordField:** input for the current password if the user wants to change the password. **private EditText newPasswordField:** input for the new password if the user wants to change the current password. **private EditText confirmPasswordField:** input for the confirmed new password if the user wants to change the current password. **private EditText newEmail :** input for new email address. **private CheckBox enableNotifications:** check box to enable/disable the notifications from the application. **private EditText newPhoneNumber:** input for the new phone number. **private Button applyChangesButton:** triggers the changes.
Functions:
private void applyChanges(): sends a series of queries to make all the requested changes. **private boolean checkPasswords():** checks if the passwords match.

3.1.1.2.16 WishlistActivity Class

Class: WishlistActivity
This activity is the Java code of the wish list screen, working along with the XML file wishlist_activity.xml.
Attributes:
private ListView listView: scrollable list that holds the wished books. **private ListAdapter listAdapter:** adapts the wished books' list items into the list view.

private WishListBundle wishlistBundle: holds the wished book objects.
Functions:
private void listRevalidate(): reconstructs the tradable book list after adding a new book or removing a book from the list.

3.1.1.2.17 GeneralListFragment Class

Class: GeneralListFragment
This fragment is the Java code of the listed books' list, working along with the XML file general_list_fragment.xml.
Attributes:
private ListView listView: scrollable list that holds the listed books. **private ListAdapter listAdapter:** adapts the listed books' list items into the list view. **private List<Books> tradeBookList:** list of the listed books to hold in memory.
Functions:

3.1.1.2.18 BookInfoActivity Class

Class: BookInfoActivity
This activity is the Java code of the book information screen, working along with the XML file book_info_activity.xml.
Attributes:
private ImageView bookImage : image of the book. **private TextView bookOwner :** text of the owner of the book. **private TextView bookText :** description of the book. **private Button transactionButton :** starts the indicated transaction when pressed.
Functions:

3.1.1.2.19 AddBookActivity Class

Class: AddBookActivity
This activity is the Java code of the book adding screen, working along with the XML file add_book_activity.xml.
Attributes:
private RadioButton transactionType : either sell of trade option that will be indicated by the user. **private AutoCompleteTextView requestBookName :** the title of the requested book. Automatically completes as the user types the title in. **private EditText priceInput :** if the radio button is set to sell, this option will allow the user to type a price in. **private AutoCompleteTextView sellBookName :** the title of the sold/traded book. Automatically completes as the user types the title in. **private Button publishButton :** publishes the book announcement.
Functions:
private Book checkBookInput(String inputBook) : checks the title of the book as the user types it in.

3.1.2 Controller

3.1.2.1 ConnectionManager Class

Class: ConnectionManager
This class is used to send a request to the server or receive a result from the server over RESTful API. Also, it puts the taken results into the required object format in the client.
Attributes:
private boolean isConnected: stores the data if there is an active connection. **private String sessionToken:** stores the token that is got from the server and used to verify that the current connection is valid. To achieve it, the user needs to login.
Functions:
public boolean login(int id, String password): sends a POST request to the server. If the user is authenticated, the method recieves and sets a sessionToken, and returns true. Otherwise, returns false.

public boolean signup(int id, String pass, String email): sends a POST request to the server to signup. If the process is done successfully, returns true.
public boolean forgotPass(String emailOrUsername, String option): sends a POST request to the server. If such username or email exists, a recover mail is sent by the server and this method returns true. Otherwise, returns false.
public List<Book> fetchBookList(): sends a GET request to the server. When the responded JSON string is arrived, it is converted into Book list and returned.
public User fetchUserInfo(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string is arrived, it is converted into User and returned.
public List<List<Book>> fetchMatchList(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string is arrived, it is converted into a list of a Book list and returned.
public List<List<Book>> fetchSuggestedList(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string is arrived, it is converted into a list of a Book list and returned.
public List<Message> fetchChat(int chattedUserID): sends a POST request with sessionToken to the server if token was set. When the responded JSON string is arrived, it is converted into list of Message objects and returned.
public boolean addBookRequest(List<Book> givenList, List<Book> wantedList): sends a POST request with sessionToken, given and wanted book list in JSON format to the server if token was set. When the response is positive, the method returns true.
public boolean addBookRequest(List<Book> givenList, double price): sends a POST request with sessionToken, given list, and price of the books in JSON format to the server if token was set. When the response is positive, the method returns true.
public Book searchBook(Book book): sends a GET request with a Book object in JSON format to the server. When the responded JSON string is arrived, it is converted into Book and returned.
public User fetchProfileOption(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into User and returned.
public List<Book> fetchTradeList(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into Book list and returned.
public List<Book> fetchWishlist(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into Book list and returned.
public List<Book> fetchHistory(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into Book list and returned.

public User fetchChatOptions(): sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into User and returned. **public User fetchAccountSettings():** sends a POST request with sessionToken to the server if token was set. When the responded JSON string arrived, it is converted into User and returned.

3.1.2.2 Parser Class

Class: Parser
This class is used to convert a JSON string into a specific object or a specific object into a JSON string.
Attributes:
public Object currentObject: stores the current object that is going to be parsed into string. the object can be Book, User or Message. **public String currentJSONstr:** stores the current JSON string that is going to be parsed into a Book, a User or a Message object.
Functions:
receiveData(data, actionKey): This function sends notifications related to the SuggestionList class to the MainManager. **storeData():** this function will store the data taken from the database in the dataframe attribute

3.1.2.3 AndroidPermissionChecker Class

Class: AndroidPermissionChecker
This class is used to check the required permission of the application. It can be accessed by any view package classes.
Attributes:
Functions:
public boolean checkGPS(): checks the GPS permission and returns true if it is given. **public boolean checkInternetConnection():** checks the Internet connection and returns true if it is open.

3.1.3 Model

3.1.3.1 Book Class

Class: Book
This class is used to store and manipulate the information of a book in the system. Defining all the variables is not a must
Attributes:
private String title: stores the title of the book **private String author:** stores the author of the book **private Date publishmentYear:** stores the publish year of the book **private String isbn:** stores the isbn of the book **private String genre:** stores the genre of the book **private int publishNo:** stores the publish number of the book **private String publisher:** stores the publisher of the book **private int pageNo:** stores the page number of the book **private String price:** stores the price of the book **private BufferedImage image:** stores the cover image of the book
Functions:

3.1.3.2 User Class

Class: User
This class is used to store and manipulate the information of a user in the system. Defining all the variables is not a must.
Attributes:
private int userID: stores the ID number of the user **private String name:** stores the name of the user **private String surname:** stores the surname of the user **private Date birthday:** stores the birthday of the user **private String username:** stores the username of the user **private String email:** stores the email address of the user **private BufferedImage foto:** stores the profile picture of the user **private String location:** stores the place where the user lives **private boolean isPremium:** is a checker if the account is premium **private Map<String, String> profileOptions:** stores the profile options of the user. The first is the keyword of the option. The second is the value. **private Map<String, String> chatOptions:** stores the chat options of the

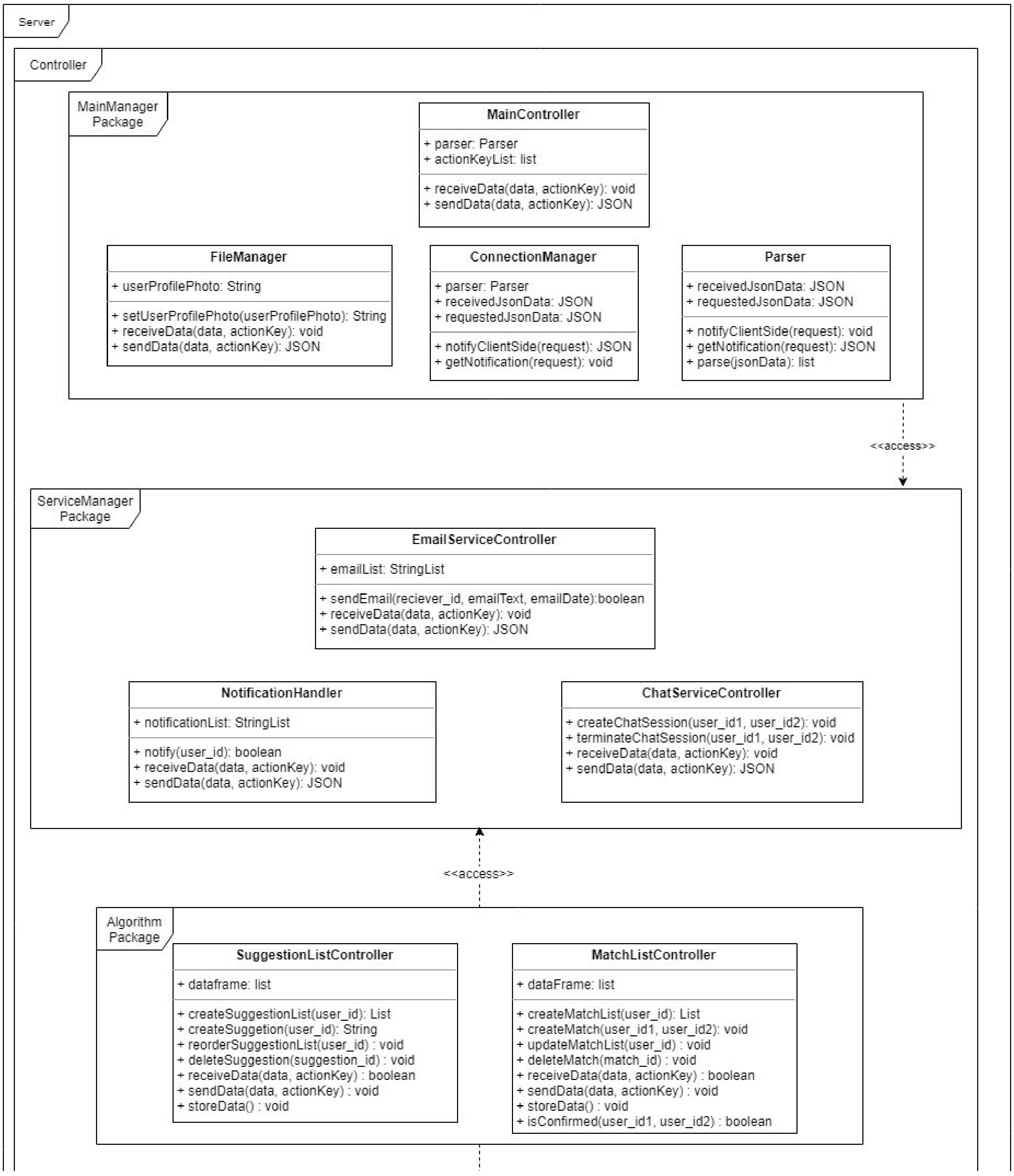
user. The first is the keyword of the option. The second is the value. **private Map<String, String> accountSettings:** stores the account settings of the user. The first is the keyword of the option. The second is the value. **private Map<int, <Message>> chat:** stores the chat data of the user. The first is the ID of the other user who is communicated. The second is the list of the message data.
Functions:

3.1.3.3 Message Class

Class: Message
This class is used to store and manipulate the data of a message in a chat. All variables need to be defined.
Attributes:
private int msgID: stores the ID of the message **private int senderID:** stores the sender ID of the message **private int receiverID:** stores the receiver ID of the message **private DateTime time:** stores the sent time of the message **private String text:** stores the content of the message **private boolean isSeen:** stores the data if the message is seen
Functions:

3.2 Server

Following two pages contains the class diagram of the Server-Side. These diagrams indicate one whole diagram.



Server
Controller
MainManager Package
MainController
+ parser: Parser
+ actionKeyList: list
+ receiveData(data, actionKey): void
+ sendData(data, actionKey): JSON
FileManager
+ userProfilePhoto: String
+ setUserProfilePhoto(userProfilePhoto): String
+ receiveData(data, actionKey): void
+ sendData(data, actionKey): JSON
ConnectionManager
+ parser: Parser
+ receivedJsonData: JSON
+ requestedJsonData: JSON
+ notifyClientSide(request): JSON
+ getNotification(request): void
Parser
+ receivedJsonData: JSON
+ requestedJsonData: JSON
+ notifyClientSide(request): void
+ getNotification(request): JSON
+ parse(jsonData): list
<<access>>
ServiceManager Package
EmailServiceController
+ emailList: StringList
+ sendEmail(reciever_id, emailText, emailDate):boolean
+ receiveData(data, actionKey): void
+ sendData(data, actionKey): JSON
NotificationHandler
+ notificationList: StringList
+ notify(user_id): boolean
+ receiveData(data, actionKey): void
+ sendData(data, actionKey): JSON
ChatServiceController
+ createChatSession(user_id1, user_id2): void
+ terminateChatSession(user_id1, user_id2): void
+ receiveData(data, actionKey): void
+ sendData(data, actionKey): JSON
<<access>>
Algorithm Package
SuggestionListController
+ dataframe: list
+ createSuggestionList(user_id): List
+ createSuggetion(user_id): String
+ reorderSuggestionList(user_id) : void
+ deleteSuggestion(suggestion_id) : void
+ receiveData(data, actionKey) : boolean
+ sendData(data, actionKey) : void
+ storeData() : void
MatchListController
+ dataFrame: list
+ createMatchList(user_id): List
+ createMatch(user_id1, user_id2): void
+ updateMatchList(user_id) : void
+ deleteMatch(match_id) : void
+ receiveData(data, actionKey) : boolean
+ sendData(data, actionKey) : void
+ storeData() : void
+ isConfirmed(user_id1, user_id2) : boolean

Figure : Class diagram of Server-Side (Part 1)

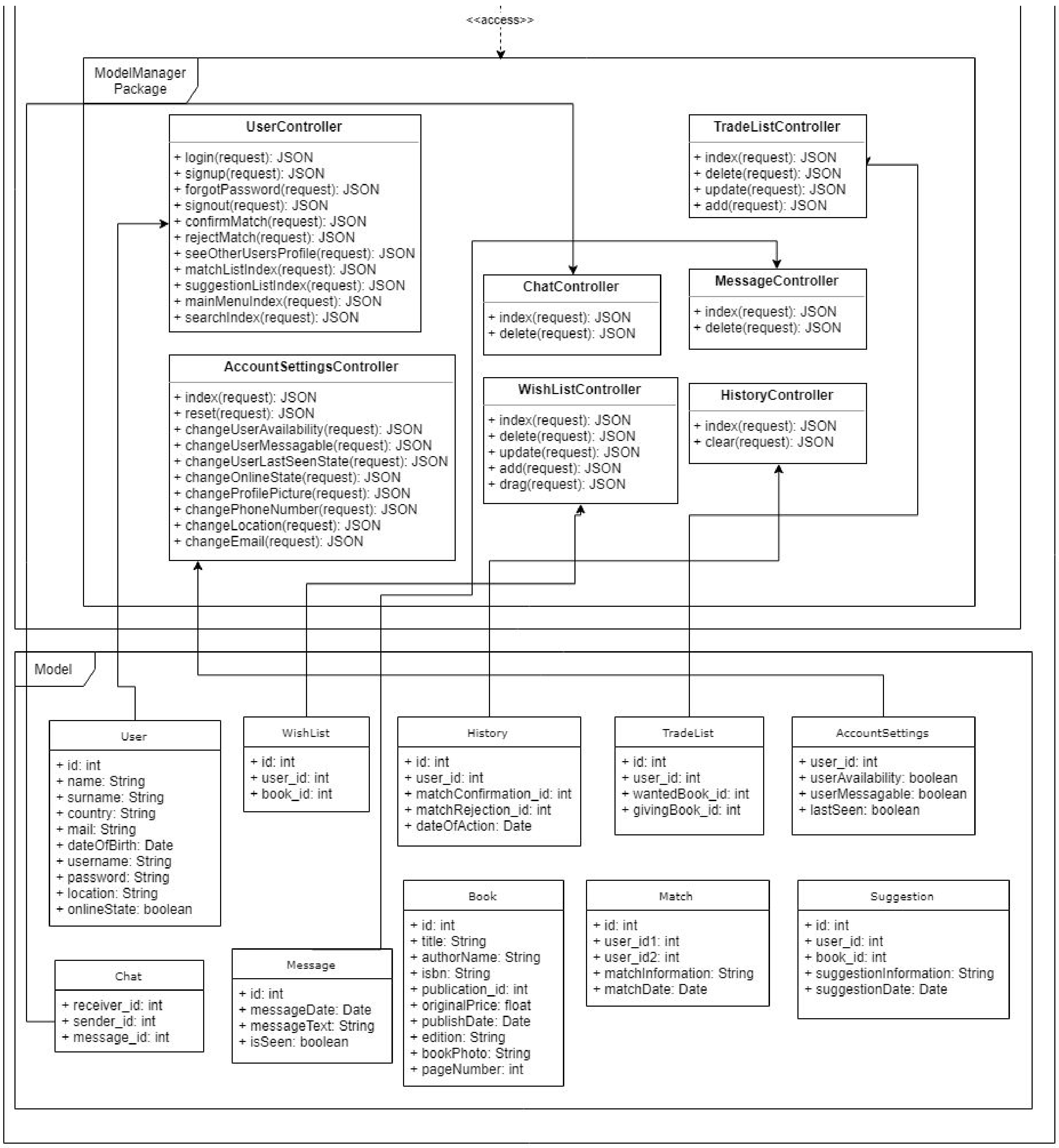


Figure : Class diagram of Server-Side (Part 2)

3.2.1 Controller

3.2.1.1 AlgorithmManager Package

3.2.1.1.1 SuggestionListController Class

Class: SuggestionListController
This class is used to create and update the suggestion lists of users.
Attributes:
dataframe: this attribute will store the data that is going to be modified
Functions:
createSuggestionList(user_id): This function takes user_id as a parameter and creates a new suggestion list for that user. **createSuggestion(user_id):** This function updates the suggestion list of a user by creating new suggestions according to some factors like user's previous activity etc. **reorderSuggestionList(user_id):** When the wishlist of the user is changed, this function will reorder the suggestion list of a user. **deleteSuggestion(suggestion_id):** When user accept or delete a suggestion, this function delete this suggestion from the SuggestionList. **receiveData(data, actionKey):** This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to call the related functions of SuggestionListController class with the notifications. **sendData(data, actionKey):** This function sends notifications related to the SuggestionList class to the MainManager. **storeData():** this function will store the data taken from the database in the dataframe attribute

3.2.1.1.2 MatchListController Class

Class: MatchListController
This class is used to create and update the match lists of users.
Attributes:
dataframe: this attribute will store the data that is going to be modified
Functions:

createMatchList(user_id): This function takes user_id as a parameter and creates a new match list for that user. **createMatch(user_id1, user_id2):** This function updates the match list of a user by creating new match according to some factors like user's previous activity etc. **updateMatchList(user_id):** This function will update the match list of a user when necessary. **deleteMatch(match_id):** When user accept or delete a match, this function delete this match from the MatchList for both users. **receiveData(data, actionKey):** This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to call the related functions of MatchListController class with the notifications. **sendData(data, actionKey):** This function sends notifications related to the MatchListController class to the MainManager. **storeData():** This function will store the data taken from the database in the dataframe attribute i**sConfirmed(user_id1, user_id2):** This function will return whether a match is accepted by both users or not. If match is accepted, it will modify the user and book states accordingly.

3.2.1.2 ServiceManager Package

3.2.1.2.1 EmailServiceController Class

Class: EmailServiceController
This class is used to control and manipulate the email sending functionality of BookClub. This is necessary to notify the users about updates and other BookClub news.
Attributes:
emailList: This attribute is a list of email templates.
Functions:
sendEmail(receiver_id, emailText, emailDate) : This function sends the emailText to the user with the taken receiver_id and returns a boolean value that shows email send successfully. **receiveData(data, actionKey) :** This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to create an emailText and emailDate according to the actionKey and calling sendEmail function with these parameters. **sendData(data, actionKey):** This function sends notifications to the

MainManager.

3.2.1.2.2 NotificationHandler Class

Class: NotificationHandler
This class is used to control the notifications of BookClub, which will be displayed in the Android devices of the users.
Attributes:
notificationList: This attribute is a list of notification templates.
Functions:
notify(user_id): This function is used to send notifications to user devices (client side). **receiveData(data, actionKey):** This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to call the related functions of NotificationHandler class with the notifications. **sendData(data, actionKey):** This function sends notifications related to the notification service to the MainManager.

3.2.1.2.3 ChatServiceController Class

Class: ChatServiceController
This class is used to control and manipulate the chat functionality of BookClub.
Attributes:
This class does not require any attributes.
Functions:
createChatSession (user_id1, user_id2): This function creates a chat session between the users taken by parameters. **terminateChatSession (user_id1, user_id2):** This function terminates a chat session when a trade is completed. **receiveData(data, actionKey):** This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to call the related functions of ChatServiceController class with the notifications.

sendData(data, actionKey): This function sends notifications related to the chat service to the MainManager.

3.2.1.3 MainManager Package

3.2.1.3.1 MainController Class

Class: MainController
This class is implemented for being a main class of the server side. Which means, it is going to send the data taken from ConnectionManager, parse it with the help of a Parser class and send it to the relevant classes with the specific actionKey taken from the actionKeyList.
Attributes:
parser: This is an instance of the Parser class, that will call the parse method for parsing the data received from the ConnectionManager class **actionKeyList:** this attribute will store the names of all possible actions within the application
Functions:
receiveData(data, actionKey): This function takes data from ConnectionManager and analyzes the actionKey, later to send to the relevant classes **sendData(data, actionKey):** This function sends notifications to the related controller classes depending on the actionKey

3.2.1.3.2 FileManager Class

Class: FileManager
This class is used to manipulate necessary files (book images, user profile photos etc.).
Attributes:
userProfilePhoto: this attribute stores the user’s profile picture. It takes it from the server. When the user uploads a new photo the old photo is deleted and renewed by FileManager.
Functions:
setUserProfilePhoto(userProfilePhoto): This function sets the user’s profile photo by updating it in the database

receiveData(data, actionKey): This function takes notifications from MainManager and returns a boolean value that shows data received successfully. Purpose of this function is to call the related functions of FileManager class with the notifications. **sendData(data, actionKey):** This function sends notifications related to the file service to the MainManager.

3.2.1.3.3 Parser Class

Class: Parser
This class is implemented for parsing any data came from ConnectionManager or MainController class. Which means, it analyzes the data, defines the actionKey and the attributes. Later, it sends them back to MainController or ConnectionManager.
Attributes:
receivedData: this attribute stores the data received from the ConnectionManager **requestedData:** this attribute stores the data that is being requested from the MainController class
Functions:
parse(jSonData): this method parses the data received and sends it via the relevant methods (receiveData, sendData) **receiveData(data, actionKey):** This function takes data from ConnectionManager and analyzes the actionKey and attributes with the parse method **sendData(data, actionKey):** This function sends the attributes and actionKey back to MainController

3.2.1.3.4 ConnectionManager Class

Class: ConnectionManager
Connection Manager class is responsible for connecting the server side to the client side. When this class is notified by ConnectionManager on the client side, it will send notifications to the related classes in server side.
Attributes:
receivedJsonData: this attribute stores the data received from the client-side

requestedJsonData: this attribute stores the data that is requested from the cliend-side. It is initialized and sent via the server-side **parser:** this is an instance of the Parser class, that will call the parse method for parsing the data received from the client-side
Functions:
notifyClientSide(request): this method notifies the client-side about the requested procedure's end. Namely, after receiving the final data from the server-side, it notifies the client side about it, so that the client-side could take it. **getNotification(request):** this method gets the notification from the client-side and informs the server-side about it.

3.2.1.4 ModelManager Package

3.2.1.4.1 UserController Class

Class: UserController Class
This class is a User model's controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
login(request): this function handles the user's login request **signup(request):** this function handles the user's singup request **forgotPassword(request):** this function handles the user's forgot password request **signout(request):** this function handles the user's signout request **confirmMatch(request):** this function handles the user's confirmed match request **rejectMatch(request):** this function handles the user's rejected match request **seeOtherUserProfile(request):** this function handles the request created when the user checks other user's profile **matchlistIndex(request):** this function returns the data of the match list when the user wants to see it **suggestionlistIndex(request):** this function returns the data of the suggestion list when the user wants to see it **mainMenuIndex(request):** this function returns the data of the main menu when the user refreshes it or clicks on the main menu **searchIndex(request):** this function returns the data of the search results

when the user searches something

3.2.1.4.2 WishListController Class

Class: WishListController Class
This class is a Wishlist model’s controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the data of the wishlist for the specific user **delete(request):** this function is activated when the user deletes a book from the wishlist **update(request):** this function is activated when the user updates the book in the wishlist **add(request):** this function is activated when the user adds a new book to the wishlist **drag(request):** this function is activated when the user reorders the wishlist books

3.2.1.4.3 HistoryController Class

Class: HistoryController Class
This class is a History model’s controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the data of the history for the specific user **clear(request):** this function is activated when the user clears the whole history

3.2.1.4.4 TradeListController Class

Class: TradeListController Class
This class is a Tradelist model's controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the data of the tradelist for the specific user **delete(request):** this function is activated when the user deletes a trade instance from the tradelist **update(request):** this function is activated when the user updates the trade instance in the tradelist **add(request):** this function is activated when the user adds a new trade instance to the tradelist

3.2.1.4.5 AccountSettingsController Class

Class: AccountSettingsController Class
This class is a AccountSettings model's controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the data of the user's current privacy settings **reset(request):** this function is activated when the user resets the privacy settings **changeUserAvailablity(request):** this function handles the user's change the availability request **changeUserMessagable(request):** this function handles the user's change messagability state request **changeLastSeenState(request):** this function handles the user's last seen state request **changeOnlineState(request):** this function handles the user's change online state request **changeProfilePicture(request):** this function handles the user's change profile picture request

changePhoneNumber(request): this function handles the user's change phone number request **changeLocation(request):** this function handles the user's change location request **changeEmail(request):** this function handles the user's change email request

3.2.1.4.6 ChatController Class

Class: ChatController Class
This class is a Chat model's controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the chat list of the specific user **delete(request):** this function is activated when the user deletes a chat instance

3.2.1.4.7 MessageController Class

Class: MessageController Class
This class is a Message model's controller class
Attributes:
This class is a controller class which handles the requests for the corresponding model, thus it does not require any attributes.
Functions:
index(request): this function returns the message list of the specific chat instance that user selected to see **delete(request):** this function is activated when the user deletes a message instance from the chat

3.2.2 Model

3.2.2.1 User Class

Class: User Class
This class is a model class for the user table from the database
Attributes:
id: this attribute represents the user id column **name:** this attribute represents the user's name column **surname:** this attribute represents the user's surname column **country:** this attribute represents the user's country column **mail:** this attribute represents the user's mail column **phoneNumber:** this attribute represents the user's phone number column **dateOfBirth:** this attribute represents the user's date of birth column **username:** this attribute represents the user's username column **password:** this attribute represents the user's password column **location:** this attribute represents the user's location column. It is needed for the GPS purposes **onlineState:** this attribute is a boolean attribute and represents the user's online/offline state column **profilePicture:** this attribute stores the name of the user's profile photo in the database **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.2 WishList Class

Class: WishList Class
This class is a model class for the wishlist table from the database
Attributes:
id: this attribute represents the wishlist id column **user_id:** this attribute represents the user's id column inside the wishlist table as a foreign key **book_id:** this attribute represents the book's id column inside the wishlist table as a foreign key **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:

This class does not require any functions, because it is a table representation class.

3.2.2.3 History Class

Class: History Class
This class is a model class for the history table from the database
Attributes:
id: this attribute represents the history id column **user_id:** this attribute represents the user's id column inside the history table as a foreign key **matchConfirmation_id:** this attribute represents the confirmed match transaction id column inside the history table as a foreign key. It can be null if no match process happened in the specified date **matchRejection_id:** this attribute represents the rejected match transaction id column inside the history table as a foreign key. It can be null if no match process happened in the specified date **dateOfAction:** this attribute represents the date of the match **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.4 TradeList Class

Class: TradeList Class
This class is a model class for the tradelist table from the database
Attributes:
id: this attribute represents the wishlist id column **user_id:** this attribute represents the user's id column inside the tradelist table as a foreign key **wantedBook_id:** this attribute represents the user's wanted book's id column inside the tradelist table as a foreign key **givingBook_id:** this attribute represents the user's giving book's id column inside the tradelist table as a foreign key **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM

Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.5 AccountSettings Class

Class: AccountSettings Class
This class is a model class for the account table from the database
Attributes:
user_id: this attribute represents the user’s id column inside the account table as a foreign key **userAvailability:** this attribute is a boolean attribute that will be turned on or off depending on the user’s availability case **userMessagable:** this attribute is a boolean attribute that will be turned on or off depending on the user’s chat availability state **lastSeen:** this attribute is a boolean attribute that will be turned on or off depending on the user’s desire to showcase their last seen time or not. **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.6 Chat Class

Class: Chat Class
This class is a model class for the chat table from the database
Attributes:
receiver_id: this attribute represents the receiver user’s id in the chat table as a foreign key **sender_id:** this attribute represents the sender user’s id in the chat as a foreign key **message_id:** this attribute represents the sended message’s id in the chat table as a foreign key **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM

Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.7 Message Class

Class: Message Class
This class is a model class for the message table from the database
Attributes:
id: this attribute represents the sent message’s id in the message table **messageText:** this attribute represents the sent message’s text in the message table **messageDate:** this attribute represents the sent message’s date in the message table **isSeen:** this attribute is a boolean attribute and it represents the sent message’s state; seen or not seen **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.8 Book Class

Class: Book Class
This class is a model class for the book table from the database
Attributes:
id: this attribute represents the book id column **title:** this attribute represents the book’s name column **authorName:** this attribute represents the book’s author’s name column **isbn:** this attribute represents the book’s isbn column **publication_id:** this attribute represents the publication id of the book **originalPrice:** this attribute represents the book’s original price column **publishDate:** this attribute represents the book’s publish date column **edition:** this attribute represents the book’s edition column **bookPhoto:** this attribute represents the book’s image’s name in the database

pageNumber: this attribute represents the book’s page number column **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.9 Match Class

Class: Match Class
This class is a model class for the match table from the database
Attributes:
id: this attribute represents the match id column **user_id1:** this attribute represents the first matched user’s id column **user_id2:** this attribute represents the second matched user’s id column **matchInformation:** this attribute represents the information about the specific match instance **book_id:** this attribute represents the matched book’s id column **matchDate:** this attribute represents the date of the match **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

3.2.2.10 Suggestion Class

Class: Suggestion Class
This class is a model class for the suggestion table from the database
Attributes:
id: this attribute represents the suggestion id column **user_id:** this attribute represents the user’s id column **suggestionInformation:** this attribute represents the information about the specific suggestion instance

book_id: this attribute represents the matched book's id column **suggestionDate:** this attribute represents the date of the suggestion **created_at:** this attribute is a default column created by Python ORM **updated_at:** this attribute is a default column created by Python ORM
Functions:
This class does not require any functions, because it is a table representation class.

4. Glossary

RestAPI: It is also known as a RESTful API. This is a web-service API that creates communication within the web. It breakdowns the transaction into small modules and tries to serve them at the same time, which makes it powerful [1].

JSON Data: JSON is an abbreviation for JavaScript Object Notation format. It is a language-independent format for storing data and transporting it. It is easy to read for humans and easy to parse for machines. It is frequently used as a data type in server-client communication [2].

XML file: Extensible Markup Language is a data file in which the data stored in both machine-readable and human-readable way [3].

MVC: The MVC is an architectural design pattern that separates the system into three main components which are model, view and controller [4].

Django Python Framework: Django is a Python Web framework [5].

SMTP: Simple Mail Transfer Protocol (SMTP) is an Internet standard for email delivery protocol [6].

Client Side: The client side of the system is responsible for managing the interaction of the software with the user [7].

Server Side: The server side of the system is responsible for system operations and data management [7].

ORM: Object-relational mapping is a mechanism for easy access and manipulates data. In this case, it was used to manipulate the database from the server [8].

5. References

[1] “What is RESTful API? .[Online]. Available:
https://searchmicroservices.techtarget.com/definition/RESTful-API .
[Accessed: 16 Feb, 2019].

[2] “What is JSON?”.[Online]. Available:
https://www.w3schools.com/whatis/whatis_json.asp . [Accessed: 16 Feb, 2019].

[3] “What is an XML file? .[Online]. Available:
” https://fileinfo.com/extension/xml . [Accessed: 15 Feb, 2019].

[4] “MVC Framework- Introduction” .[Online]. Available:
https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. [Accessed: 14 Feb, 2019].

[5] “Django”.[Online]. Available:
https://www.djangoproject.com/ . [Accessed: 14 Feb, 2019].

[6] “Simple Mail Transfer Protocol” .[Online]. Available:
https://www.geeksforgeeks.org/simple-mail-transfer-protocol-smtp/.
[Accessed: 16 Feb, 2019].

[7] “Client- Server Architecture” .[Online]. Available:
https://www.techopedia.com/definition/438/clientserver-architecture .
[Accessed: 08 Feb, 2019].

[8] “object-relational mapping (ORM)”   .[Online]. Available:
https://searchwindevelopment.techtarget.com/definition/object-relational-mapping [Accessed: 16 Feb, 2019].