
Bilkent University

Department of Computer Engineering

# Senior Design Project

*Project short-name: AeroCast*

# Project Low-Level Design Report

Ahmet Alparslan Çelik, Ömer Berk Uçar, Muhammed Yusuf Satıcı, Yasin İlkağan Tepeli, Ahmet Taha Albayrak

Supervisor: Çiğdem Gündüz Demir
Jury Members: Selim Aksoy and Mustafa Özdal

Progress Report

Feb 12, 2018

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

# 1. Introduction

While planning a holiday or a business trip, everyone once in awhile tries to find the best airfare deal according to their needs. In general, it is a tedious work due to the dynamic changes in prices. Airlines change their prices according to various criteria such as supply and demand, how far the departure date is, which day of the week the flight is, etc. Thus, finding the cheapest ticket by searching manually can be cumbersome. Furthermore, it is necessary to optimally find the best purchase date for a particular route within the traveller's' budget.

With the current technologies, it is possible to track airfares, get notifications in case of a drop or increase in price and even get rough predictions if the price will increase or decrease for a particular route. Nevertheless, the existing methods do not provide user specific airplane predictions. Therefore, in order to purchase the ideal cheapest ticket, travellers will need to use separate services. Moreover, if they were to look for different travelling plans, the number of different routes they need to check will increase making things even more time consuming and complicated.

Aerocast aims to minimize the efforts put into finding the best airfare deal available. It will be a web application, which collects the available data scattered through different web services and analyzes them in a user specific way. Therefore, it will assist travellers to find the best airfare in the minimum possible time. The users can search and track their flight of interest months before the departure and can get elusive intelligence. Regarding to the personal preferences, users can provide preferences such as carrier type or cabin class, Aerocast will list the most related results.

In the report, low level architecture and design of the system are provided. Initially, engineering standards and design trade-offs are listed. Afterwards, packages and their functionalities along with detailed class diagram views are included. Finally, all class

interfaces of all packages are reported to give detailed descriptions for the functionalities of each software component.

## 1.1. Design Trade-Offs

### 1.1.1. Functionality vs Usability

The AeroCast system aims to support various functionalities along with a user-friendly interface. Therefore, it is important for the AeroCast system to have a balance between the variety of the functionalities and the ease of use of the user interface. In the design of the AeroCast system, the user interface is organized in a modular way that the functionalities of the system does not overwhelm or annoy the user. Hence, the user does not access all of the functionalities and the whole content of the system at the same time but rather the functionalities and the content is provided to the user as it is needed. Therefore, AeroCast tries to ensure the usability of the system regardless of how many functionalities it has.

### 1.1.2. Security vs Performance

The AeroCast system aims to protect the user information and flight data stored in its databases. Therefore, AeroCast uses SHA-256 hashing to protect the password information of the users and it limits the number of daily requests to 200 in order to protect the flight data from malicious actions. Also, AeroCast aims to protect its web servers by using the Cloudflare Services. To make trade-off balanced, the flight data and user information won't be encrypted because the decryption of the frequently used data slows down the system's performance. Therefore, Aerocast avoids unnecessary encryption operations.

### 1.1.3. Memory vs Performance

Since the system uses a prediction system and works with Machine Learning, it will need huge amount of data to predict the result more accurately. With this amount of data, the processes will be longer. However, this does not mean memory is the only credential. On the web, speed and performance means everything to the user. Because of that, the system needs to be fast while being accurate. For the predictions, memory will be more important unless the threshold for the performance is crossed.

When the tracked flights and routes need to be updated, system will run the prediction system. Having an accurate prediction will require huge amount of data. However, processing this huge amount of data can make the system slow and decrease the performance. Therefore, a scheduler which determines when to make updates and take data from other websites will be used in order to make the trade-off balanced and not to lose from the both sides. By doing so, the system will continue to perform updates in a proper way although the system has a great amount of space.

### 1.1.4. Compatibility vs Extensibility

The compatibility is an essential part of the design goals. Currently, the AeroCast system is fully compatible with Chrome and Opera web servers and partially compatible with Firefox. Although the compatibility of the client with the web servers mentioned above is important, the most of the functionalities of the system relies on the back-end servers. Therefore, despite the importance of the compatibility of the system, the real significance is given to the extensibility of the back-end system. Hence, To make the back-end system extensible, the back-end is divided into two servers, which deals with different functionalities. Therefore, the programmers can easily extend the prediction or application system without having any effect on the other system.

## 1.2. Engineering Standards

Throughout the report, UML design principles [1] and IEEE citation guideline [2] is followed. UML standard is broadly accepted in describing software components, class interfaces, diagrams, use cases, subsystem decompositions and many more. In this very report, UML language is used to document system design, software components and their functionalities in a modular manner. Moreover, IEEE citation guide is one of the widely used engineering standards in engineering related reports to manage citations. Thus, we continue the tradition and follow its set of rules throughout the report.

## 1.3. Interface Documentation Guidelines

For the report, OOS Standards [3] were followed. As it is indicated in the standards, a class description will follow the detailed hierarchy:

<table>
<tr><td colspan="2">class className</td></tr>
<tr><td colspan="2">Class Description...</td></tr>
<tr><td colspan="2">Attributes</td></tr>
<tr><td colspan="2">TypeOfAttribute attributeName</td></tr>
<tr><td colspan="2">Methods</td></tr>
<tr><td>methodName(Arguments)</td><td>Method Description</td></tr>
</table>

### 1.4. Definitions, Acronyms, and Abbreviations

**API:** Application Programmer Interface.

**RESTful API**: API with representational state transfer.

**Client:** The application that initiates the connection with server.

**Server:** The program that provides the services to the application.

**MongoDB:** NoSQL database.

**HTTP**: Application Layer Protocol.

## 2. Packages

The packages of the AeroCast system is divided into three servers, which are the client, application server and prediction server. In this chapter, the packages of these servers and the classes inside these packages are explained along with UML class diagrams.

### 2.1. Client

Client is responsible from all user interfaces and communication. It renders pages, gets user inputs, sends user requests to application server, receive the response of the application server, and rerender the page according to the response. It has a single package which is Presentation Package. Presentation Package has two responsibilities. First one is handling the graphical user interface, rendering pages, and second responsibility is handling user input and server responses. Presentation Package classes has both controller and view codes therefore client does not need any additional server side controller package.

### 2.1.1. Presentation Package

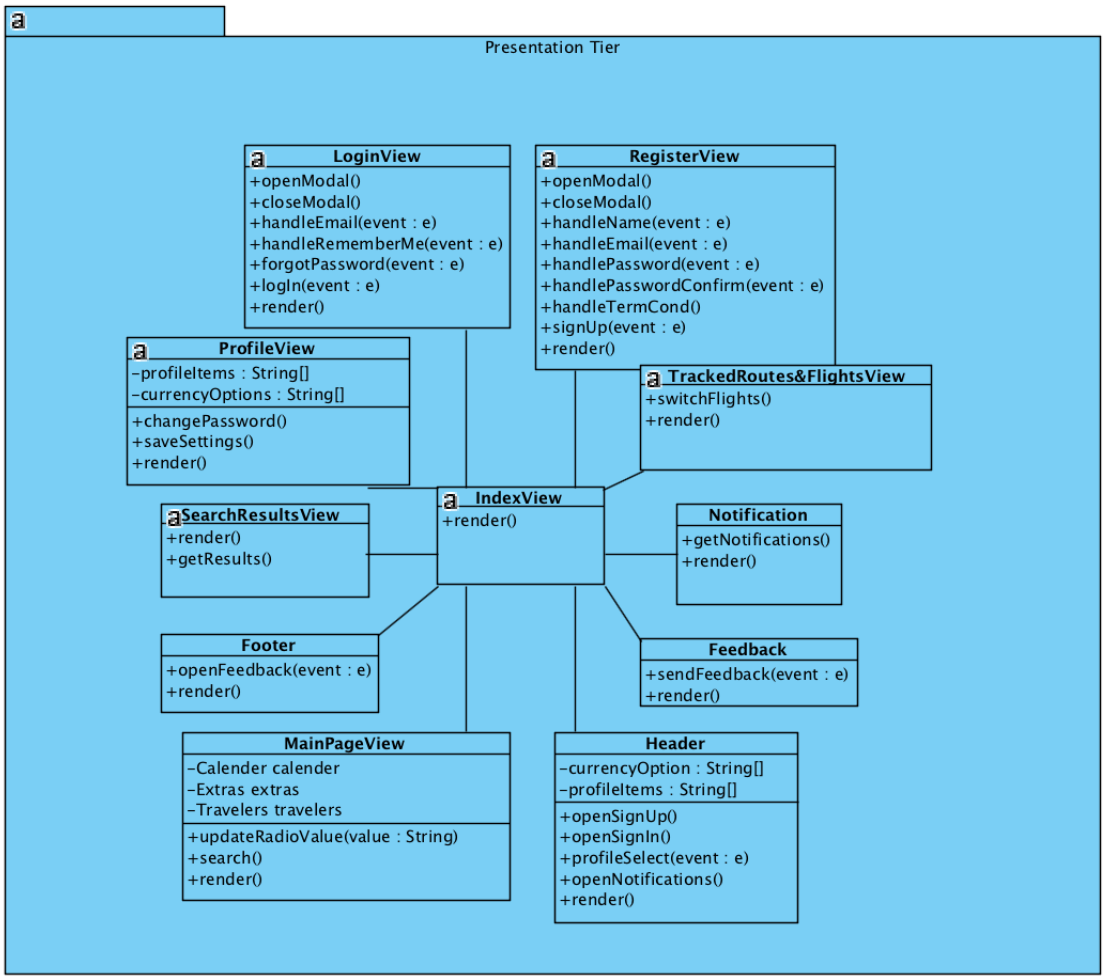


Figure 1 - Presentation Package t

Presentation Tier consists of one main class named as IndexView and 10 other classes. These 10 classes (LoginView, RegisterView, ProfileView, TrackedRoutesAndFlightsView, SearchResultView, Notification, Footer, Header, Feedback, MainPageView, Header) are the components of the web pages. IndexView is the container of the other classes.

**IndexView:** This class consist of 1 Header class, 1 Footer Class and 1 from other classes according to current page. It is basically the core of the Client.

**LoginView:** This class represent the login of the user.

**RegisterView:** This class represents the registration of the user.

**Notification:** This class represents the notification page of the website. It consist of notifications.

**ProfileView:** This class represent the settings of the user where user can change password or save the other settings. It has profileItems and currencyOptions instances as string arrays. It also has changePassword, saveSettings and render methods.

**SearchResultsView**: This class represents the results of the searches of the flights or the routes.

**Footer**: This class represents the Footer of the page and it is included by IndexView in any condition.

**Header:** This class represents the Header of the page and it is included by IndexView in any condition. It consist of currencyOptions and profileItems and functions to open modals.

**TrackedRoutesAndFlightsView**: This class represents the tracked routes and searches of user.

**Feedback**: This class represents the feedback page of the system. It has function to send feedback.

**MainPageView**: This class represents the homepage of the website and consists of the objects Traveler, Calendar and Extras (Extra Options) and it is used for the searches.

### 2.2. Application Server

Application Server is responsible from receiving the user requests coming from the client and handling them with appropriate operations. The application server has two components, which are the logic package and the data package. Logic package handles all communication between client and application and it manages the entities of the application, which are residing in the data package. The data package consists of the entity objects. In addition to these responsibilities, the application server communicates with the

prediction server to handle the forecast related operations. The application server is the main component of the system that contains the most of the functionalities.

### 2.2.1. Logic Package

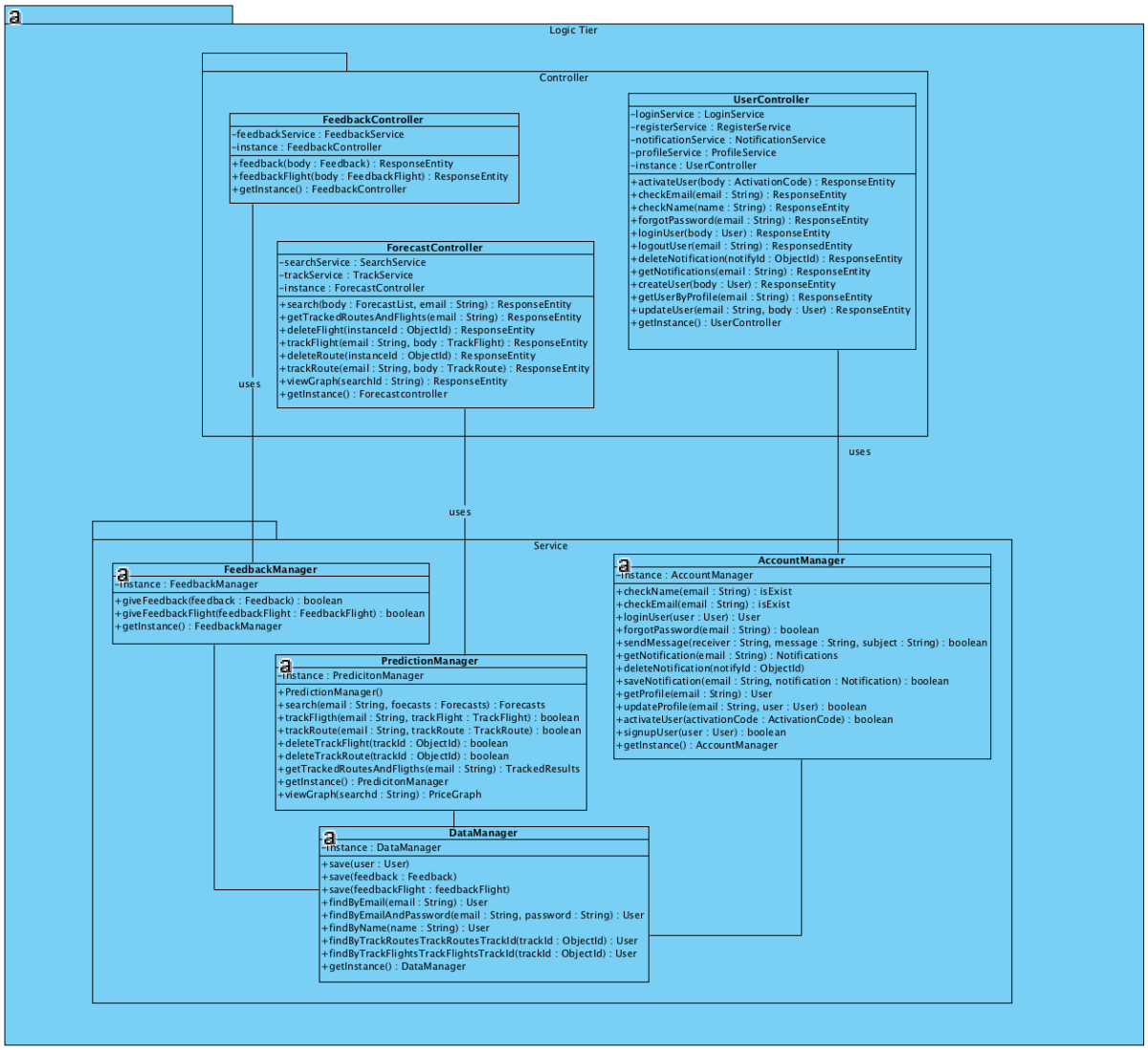


Figure 2 - Logic Package

Logic package consists of the controller package and the service package. The controller package has three classes, which are the feedback controller, the user controller and the forecast controller. These controllers receive HTTP requests from the client and sends the content of these requests to the corresponding services implemented in the service

package. The corresponding classes in the service package either handle these requests by themselves or propagate these requests to the Prediction server. After the responses for the requests are generated by the service package classes or the Prediction Server, the responses are returned to the controller classes and the controller classes return these responses to the client. In addition to these request response cycles, the classes in the service package are responsible from managing the database insertion and retrievals.

**FeedbackController:** This class is a singleton class that receives the client's requests about the feedback and responses the client requests.

**ForecastController:** This class is a singleton class that receives the client's requests about the forecast and responses the client requests.

**UserController:** This class is a singleton class receives the clients requests about the user and responses the client requests.

**FeedbackManager:** Feedback Manager class is the singleton data class that connects to feedback database. It manages the feedback requests and work with other managers to store the feedback.

**PredictionManager:** Prediction Manager class is the connection point of Application Server and Machine Learning Server. It is a singleton class and it manages the forecast functionality which needs the Machine Learning Server work.

**DataManager:** DataManager class is singleton data class that manages all database related requests. When it gets the request it connects to the database and modifies it according to the request.

**AccountManager:** AccountManager class is a singleton class that manages the account related processes. It controls modification requests of accounts, login and register processes. It works with other managers to store the new accounts into the database or update the database with the modifications.

### 2.2.2. Data Package

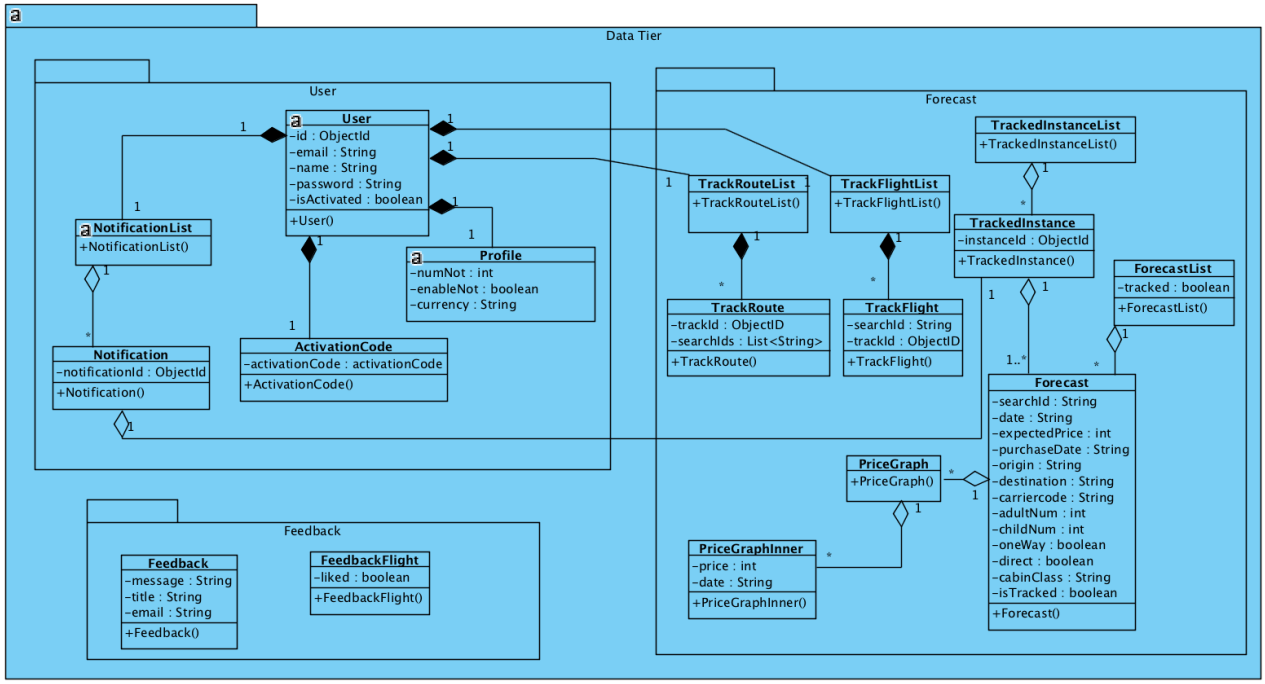


Figure 3 - Data Package

Data Package consists of three packages named as user, forecast and feedback. User package keeps the classes related to the registered users. User package consists of user, profile, notification, notifications and activation code classes. Forecast package includes the classes related to the forecasts and the tracked entities. Track Routes and Track Flights objects are persistent objects that are stored inside the database along with the user objects whereas forecast, price graph and tracked results objects are passed to the client inside the HTTP responses and they are not stored inside the database.

**User:** This class represents the registered users of the application. The users have a unique email and name. The class also keeps the hashed versions of the user's password and activation code.

**Profile:** This class represents the preferences of the user such as currency and enable notifications. It also keeps the number of notification user has.

**Activation Code:** This class represents the activation code of the user.

**Notification List:** This class is an array of notification objects that the user has.

**Notification:** This class represents a notification of the user. Each notification has a unique id and a tracked instance.

**Tracked Instance List:** This class is a list of tracked instances and it is returned to the client whenever the user displays the tracked routes and flights on the screen.

**Tracked Instance**: This class represents an entity that the user has tracked. Tracked Instance has a unique instance id and it has a list of forecasts.

**Forecast List**: This class represents a list of forecast objects.

**Forecast:** This class represents a forecast generated by the machine learning algorithm allocated in the machine learning server. ML server returns the forecast results to the application server whenever it is needed and this forecast results are kept in this class along with the input values.

**Price Graph**: This class keeps an array of price and date values for generating the price history graph.

**Price Graph Inner**: This class represents a pair of date and airfare.

**Track Route List**: This class represents an array of tracked routes.

**Track Flight List**: This class represents an array of tracked flights.

**Track Route**: This class consist of a unique track id and a list of search ids corresponding to the forecasts stored in the database.

**Track Flight**: This class consist of a unique track id and a search id corresponding to a specific forecast stored in the database.

**Feedback**: This class represents the feedback given to the system by the user.

**Feedback Flight**: This class represents the feedback given to the flight forecast by the user.

### 2.3. Prediction Server

Prediction Server is responsible from generating, updating and returning forecasts as well as collecting and storing flight information. The prediction server has two components, which are the logic package and the data package. Logic package handles all communication between the application server and the prediction server and it makes airfare predictions based on the flight data by using the machine learning algorithms. The data package is responsible from collecting the data that is used by the logic package and it also contains the entity objects of the prediction server. The prediction server is the component of the system that handles all machine learning operations.

#### 2.3.1. Logic Package

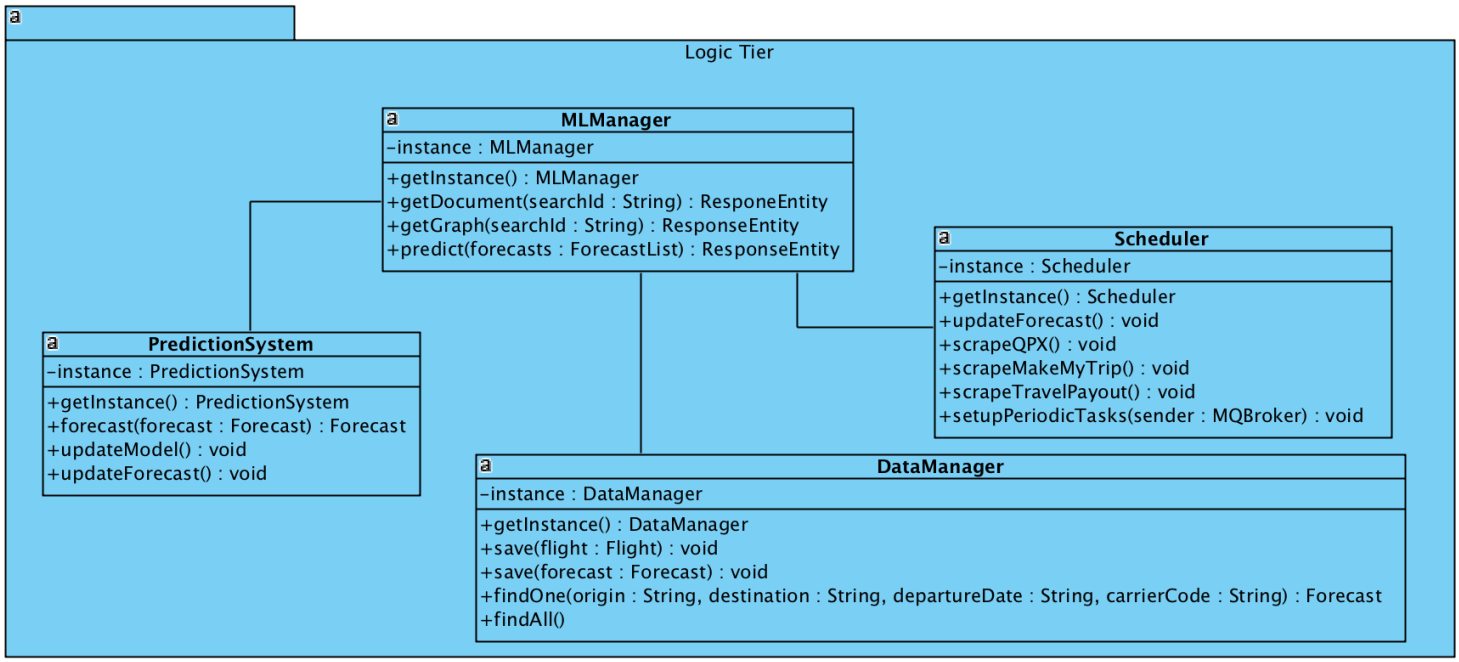


Figure 4 - Logic Package

Performing prediction, scheduling scrapers to collect data periodically, serialize & deserialize the flight data are among responsibilities of this particular package.

**ML Manager**: It is a Singleton class that is responsible for routing the related requests to related Manager classes: performing predictions, storing particular objects in database and scheduling the scrapers.

**Data Manager**: It is a Singleton class that is responsible for saving objects into database.

**Prediction System**: It is a Singleton class that is responsible for performing forecasts given an initialized forecast object.

**Scheduler**: It is a Singleton class that is responsible for managing the scheduled periodic tasks to crawl the updated flight data.

### 2.3.2. Data Package

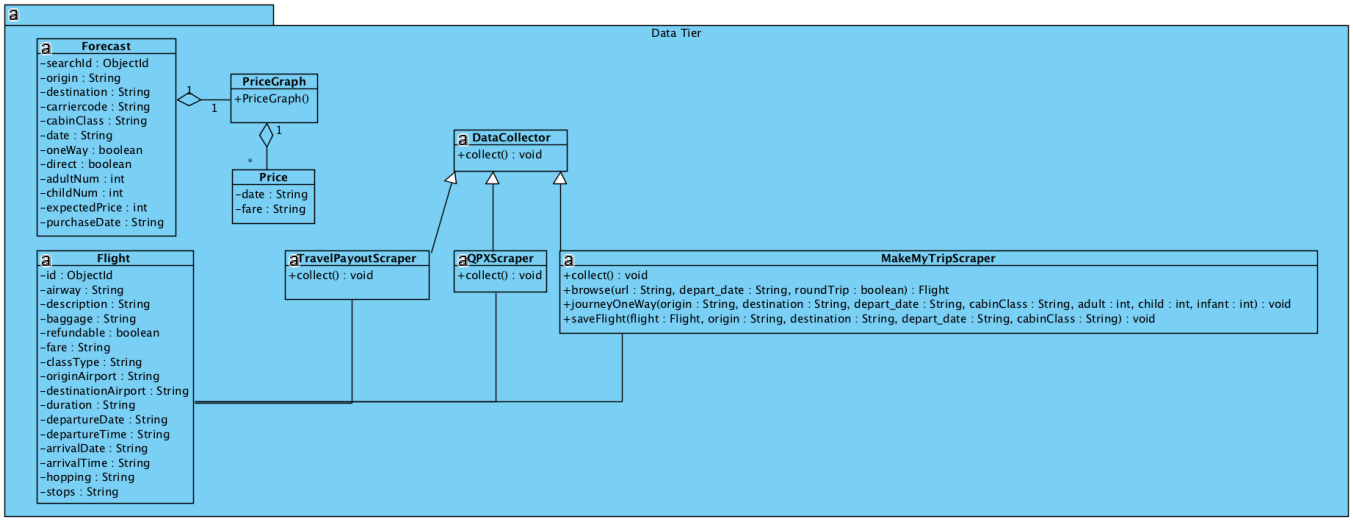


Figure 5 - Data Package

Data Package is responsible from the management and storage of the flight and forecast related data. It also uses its web scrapers to collect data from websites. All entities of the machine learning server is managed by this package.

**Data Collector:** This class is the super class for the scraper scripts. It has a collect method for scraping web content.

**Travel Payout Scraper**: This class overrides the collect method of the data collector to collect data from travel payout website.

**QPX Scraper**: This class overrides the collect method of the data collector to collect data from google qpx website.

**Make My Trip**: This class overrides the collect method of data collector to collect data from make my trip website. It uses browse method to retrieve data and save flight method to store the newly obtained flight data.

**Forecast**: This class represents the forecast including the input parameters generated by machine learning algorithm.

**Flight:** This class represents the flight data that is generated by the scrapers.

**Price Graph**: This class keeps an array of price and date values for generating the price history graph.

**Price**: This class represents a pair of date and airfare.

# 3. Class Interfaces

The class interfaces consist of the interfaces of the client, the application server and the prediction server. The class interfaces of these servers and the methods of these classes are explained in detail below.

## 3.1. Client

| **class IndexView** |
|---|
| This class consists of other classes and it is the base class of the client package. |

| **class LoginView** | |
|---|---|
| This class receives the user credentials and check them through Java server and allows or not to login to the user. | |
| **Methods** | |
| openModel() | Popup Model which contains Login interface opens. |
| closeModel() | Close the Popup Model |
| handleEmail(event) | Checks whether the email is appropriate or not |
| handlePassword(event) | Checks whether the password is valid or not |
| handleRememberMe(event) | Handles whether Login information will be saved or not for future logins. |
| forgotPassword(event) | Handles the situation of forgetting and renewing the password. |
| logIn(event) | Sends user credentials to the server and logs in if it is true. |

| **class RegisterView** | |
|---|---|
| This class receives the user informations and send them to Java Server to make them a registered user. | |
| **Methods** | |
| openModel() | Popup Model which contains Register interface opens. |
| closeModel() | Close the Popup Model |
| handleName(event) | Checks whether the name is appropriate or not |
| handleEmail(event) | Checks whether the email is valid or not |
| handlePassword(event) | Checks whether the password is valid or not |
| handlePasswordConfirm(event) | Checks whether the two passwords are same. |
| handleTerm() | Handles the accepting the Terms and Conditions. |
| SignUp(event) | Sends the user credentials to the server for registering. |

| **class Notification** | |
|---|---|
| Notification class shows users the notifications where it gets from the Java Server. | |
| **Methods** | |
| getNotifications() | It gets the notifications of the user from Java Server. |

| **class ProfileView** | |
|---|---|
| ProfileView class controls the profile settings. It takes user credentials and sends them to the java server to update. | |
| **Attributes** | |
| String[] profileItems | |
| String[] currencyOptions | |
| **Methods** | |
| changePassword() | It sends the new password of the user to the Java Server to change the password. |
| currencyOptions() | It changes and sends the new currency to Java Server for the user. |

| **class searchResultsView** | |
|---|---|
| searchResultsView class sends search options to the java server and gets the result. It renders the result list. | |
| **Methods** | |
| getSearchResults() | It gets the response of the search by sending the options. |

| **class MainPageView** | |
|---|---|
| MainPageView class is the class of the main page. | |
| **Attributes** | |
| Calendar calendar | |
| Travelers travelers | |
| Extras extras | |
| **Methods** | |
| updateRadioValue(String value) | It changes the value of the radio button to decide whether the trip one way or return.. |
| search() | It sends the credentials to the Java Server and opens the searchResult page. |

| **class Header** | |
|---|---|
| This class is the base class of top view of the website. It contains login and register components, profile menu. It is one of the main page navigator classes. | |
| **Attributes** | |
| String[] currencyOption | |
| String[] profileItems | |
| **Methods** | |
| openSignUp() | It creates and calls the LogIn class. |
| openSignIn() | It creates and calls the Register class. |
| profileSelect(event) | It controls the navigation of the user. |
| openNotifications() | It calls the Notification class to the root of the page. |

| **class TrackedFlightsSearchesView** | |
|---|---|
| TrackedFlightsSearchesView class gets tracked instances from the java server and render the tracked instance list. | |
| **Methods** | |
| switchFlights() | It changes the content into Flights. |
| switchRoutes() | It changes the page content into Routes. |

| **class Footer** | |
|---|---|
| This class is the base class of bottom view of the website. It contains navigation links to about, contact, and feedback pages. | |
| **Methods** | |
| openFeedback(event) | It calls the Notification class. |

| **class Feedback** | |
|---|---|
| Feedback class posts user feedbacks to the java server. | |
| **Methods** | |
| sendFeedback(event) | It sends the feedback by user to the Java Server |

## 3.2. Application Server

Application Server consists of the logic package and the data package. The classes, attributes and methods inside the logic and data packages are explained in detail below.

### 3.2.1. Logic Package

Logic Package consists of the controller package and the service package. The classes, attributes and methods inside the controller and service packages are explained in detail below.

#### 3.2.1.1. Controller Package

| **class FeedbackController** | |
|---|---|
| This class receives the client's requests about the feedback and responds to the client requests. | |
| **Attributes** | |
| FeedbackService feedbackService | |
| FeedbackController instance | |
| **Methods** | |
| public ResposeEntity feedback(Feedback body) | This method calls the feedback manager to handle the feedback. |
| public ResposeEntity feedbackFlight(FeedbackFlight body) | This method calls the feedback manager to handle the flight feedback. |
| Public FeedbackController getInstance() | This method represents that the class is a singleton class. |

| **class ForecastController** | |
|---|---|
| This class receives the client's requests about the forecast and responds the client requests. | |
| **Attributes** | |
| SearchService feedbackService | |
| TrackService trackService | |
| ForecastController instance | |
| **Methods** | |
| public ResposeEntity search(Feedback body, String email) | This method calls the prediction manager to handle the search operation. |
| public ResposeEntity getTrackedRoutesAndFlights(String email) | This method calls the prediction manager to retrieve the tracked routes and flights. |
| public ResponseEntity deleteFlight(ObjectId instanceId) | This method calls the prediction manager to handle the delete flight operation. |
| public ResponseEntity trackFlight(String email, TrackFlight body) | This method calls the prediction manager to track the flight. |
| public ResponseEntity deleteRoute(ObjectId instanceId) | This method calls the prediction manager to handle the delete route operation. |
| public ResponseEntity trackRoute(String email, TrackRoute, body) | This method calls the prediction manager to track the route. |
| public ResponseEntity viewGraph(String searchId) | This method calls the prediction manager to retrieve the price-date graph. |
| public ForecastController getInstance() | This method represents that the class is a singleton class. |

| **class UserController** | |
|---|---|
| This class receives the client's requests about the user and responds the client requests. | |
| **Attributes** | |
| LoginService loginService | |
| RegisterService loginService | |
| NotificationService notificationService | |
| ProfileService profileService | |
| UserController instance | |
| **Methods** | |
| public ResposeEntity activateUser(ActivationCode body) | This method calls the account manager to activate the user account. |
| public ResposeEntity checkEmail(String email) | This method calls the account manager to check the correctness of email. |
| public ResposeEntity checkName(String name) | This method calls the account manager to check the correctness of name. |
| public ResposeEntity forgotPassword(String email) | This method calls the account manager to handle forgot password operation. |
| public ResposeEntity loginUser(String user) | This method calls the account manager to handle login operation. |
| public ResposeEntity deleteNotification(ObjectId notiftyId) | This method calls the account manager to delete the notification. |
| public ResposeEntity getNotifications(String email) | This method calls the account manager to retrieve the notifications. |
| public ResposeEntity createUser(String email) | This method calls the account manager to create user account. |
| public ResposeEntity getUserByProfile(String email) | This method calls the account manager to retrieve user profile. |
| public ResposeEntity updateUser(String email, User body); | This method calls the account manager to update the user profile. |

| | |
|---|---|
| public UserController getInstance() | This method represents that the class is a singleton class. |

#### 3.2.1.2. Service Package

| **class FeedbackManager** | |
|---|---|
| Feedback Manager class is the data class that connects to the feedback database. It manages the feedback requests and work with other managers to store the feedback. | |
| **Attributes** | |
| FeedbackManager instance | |
| **Methods** | |
| public boolean giveFeedback(Feedback feedback) | This method checks the feedback and persist the data. |
| public boolean giveFeedbackFlight(FeedbackFlight feedbackFlight) | This method checks the the feedback flight and persist the data. |
| public FeedbackManager getInstance() | This method represents that the class is a singleton class. |

| **class PredictionManager** | |
|---|---|
| Prediction Manager class is the connection point of Application Server and Machine Learning Server. It manages the forecast functionality which needs the Machine Learning Server work. | |
| **Attributes** | |
| PredictionManager instance | |
| **Methods** | |
| public List<Forecast> search(Forecast body, String email) | This method contacts the Python server to retrieve best and cheapest flights. |
| public List<TrackedResult> getTrackedRoutesAndFlights(String email) | This method contacts the Python server to retrieve user's tracked flight and search. |
| public boolean deleteTrackFlight(ObjectId instanceId) | This method sends request to Python server to delete user's tracked flight. |
| public boolean trackFlight(String email, TrackFlight body) | This method sends request to Python server to add user's tracked flight. |
| public boolean deleteTrackRoute(ObjectId instanceId) | This method sends request to Python server to delete user's tracked route. |
| public boolean trackRoute(String email, TrackRoute body) | This method sends request to Python server to add user's tracked route. |
| public PriceGraph viewGraph(String searchId) | This method retrieves the price-date graph and returns to the client. |
| public PredictionManager getInstance() | This method represents that the class is a singleton class. |

| **class AccountManager** | |
|---|---|
| AccountManager class manages the account related processes. It controls modification requests of accounts, login and register processes. It works with other managers to store the new accounts into the database or update the database with the modifications | |
| **Attributes** | |
| AccountManager instance | |
| **Methods** | |
| public boolean activateUser(ActivationCode body) | This method checks the activation code and activates the user account. |
| public isExist checkEmail(String email) | This method checks the uniqueness of the user email |
| public isExist checkName(String name) | This method checks the uniqueness of the user name |
| public boolean forgotPassword(String email) | This method sends a recovery email to the user's email address. |
| public User loginUser(String user) | This method logs the user into the system. |
| public boolean deleteNotification(ObjectId notiftyId) | This method deletes the notification identified by the notifyId |
| public List<Notification> getNotifications(String email) | This method retrieves the notifications of the user from the database. |
| public boolean saveNotification(String email, Notification notification) | This method saves the new notification into the database. |
| public User getProfile(String email, User user); | This method retrieves the user profile. |
| public boolean updateProfile(String email, User user); | This method updates the user profile. |
| public boolean sendMessage(String reciever, String message, String subject) | This method sends messages to the receiver. |
| public boolean signupUser(User user) | This method creates a new user account and saves it into the database. |

| | |
|---|---|
| public AccountManager getInstance() | This method represents that the class is a singleton class. |

| **class DataManager** | |
|---|---|
| DataManager class is data class that manages all database related requests. When it gets the request it connects to database and modifies it according to the request. | |
| DataManager instance | |
| **Methods** | |
| public void save(User user) | This method inserts the user into persistent NOSQL database |
| public void save(Feedback feedBack) | This method inserts the feedback about web application into persistent NOSQL database |
| public void save(FeedbackFlight feedbackFlight) | This method inserts the feedback about flight into persistent NOSQL database |
| public User findByEmail(String email) | This method retrieves the User by given email |
| public User findByEmailAndPassword(ObjectId instanceId) | This method retrieves the User by given email and password |
| public User findByName(String email, TrackRoute body) | This method retrieves the User by given name |
| public User findByTrackRoutesTrackRoutesTrackedId(ObjectId trackedId) | This method retrieves the User by given track routes object id. |
| public User findByTrackFlightsTrackFlightsTrackId(ObjectId trackedId) | This method retrieves the User by given track flight object id. |
| public DataManager getInstance() | This method represents that the class is a singleton class. |

### 3.2.2. Data Package

Data Package consists of the user package, the forecast package and the feedback package. The classes, attributes and methods of these three packages are explained below.

#### 3.2.2.1. User Package

| **class User** | |
|---|---|
| This class represents the registered users of the application. The users have a unique email and name. The class also keeps the hashed versions of the user's password and activation code. | |
| **Attributes** | |
| ObjectId id | |
| String name | |
| String email | |
| String password | |
| boolean isActivated | |
| **Methods** | |
| public User() | Constructor |

| **class NotificationList** | |
|---|---|
| This class is an array of notification objects that the user has. | |
| **Attributes** | |
| List<Notification> notification | |
| **Methods** | |
| public NotificationList() | Constructor |

| **class Notification** | |
|---|---|
| This class represents a notification of the user. Each notification has a unique id and a tracked instance. | |
| **Attributes** | |
| ObjectId notificationId | |
| **Methods** | |
| public Notification() | Constructor |

| **class Profile** | |
|---|---|
| This class represents the preferences of the user such as currency and enable notifications. It also keeps the number of notification user has. | |
| **Attributes** | |
| int numNot | |
| boolean enableNot | |
| String currency | |
| **Methods** | |
| public Profile() | Constructor |

| **class ActivationCode** | |
|---|---|
| This class represents the activation code of the user. | |
| **Attributes** | |
| ActivationCode activationCode | |
| **Methods** | |
| public ActivationCode() | Constructor |

#### 3.2.2.2. Forecast Package

| **class TrackedInstanceList** | |
|---|---|
| This class is a list of tracked instances and it is returned to the client whenever the user displays the tracked routes and flights on the screen. | |
| **Attributes** | |
| List<TrackedInstance> trackedInstance | |
| **Methods** | |
| public TrackedInstanceList() | Constructor |

| **class TrackedInstance** | |
|---|---|
| This class represents an entity that the user has tracked. Tracked Instance has a unique instance id and it has a list of forecasts. | |
| **Attributes** | |
| ObjectId instanceId | |
| **Methods** | |
| public TrackedInstance() | Constructor |

| **class ForecastList** | |
|---|---|
| This class represents a list of forecast objects. | |
| **Attributes** | |
| boolean tracked | |
| List<Forecast> forecast | |
| **Methods** | |
| public ForecastList() | Constructor |

| **class Forecast** | |
|---|---|
| This class represents a forecast generated by the machine learning algorithm allocated in the machine learning server. ML server returns the forecast results to the application server whenever it is needed and this forecast results are kept in this class along with the input values. | |
| **Attributes** | |
| String searchId | |
| String date | |
| int expectedPrice | |
| String purchaseDate | |
| String origin | |
| String destination | |
| String carriercode | |
| int adultNum | |
| int childNum | |
| boolean oneWay | |
| boolean direct | |
| String cabinClass | |
| boolean isTracked | |
| **Methods** | |
| public Forecast() | Constructor |

| **class PriceGraph** | |
|---|---|
| This class keeps an array of price and date values for generating the price history graph. | |
| **Attributes** | |
| List<PriceGraphInner> priceGraph | |
| **Methods** | |
| public PriceGraph() | Constructor |

| **class PriceGraphInner** | |
|---|---|
| This class represents a pair of date and airfare. | |
| **Attributes** | |
| int price | |
| String date | |
| **Methods** | |
| public PriceGraphInner() | Constructor |

| **class TrackRouteList** | |
|---|---|
| This class represents an array of tracked routes. | |
| **Attributes** | |
| List<TrackRoute> trackRoute | |
| **Methods** | |
| public TrackRouteList() | Constructor |

| **class TrackFlightList** | |
|---|---|
| This class represents an array of tracked flights. | |
| **Attributes** | |
| List<TrackFight> trackFlight | |
| **Methods** | |
| public TrackFlightList() | Constructor |

| **class TrackRoute** | |
|---|---|
| This class consists of a unique track id and a list of search ids corresponding to the forecasts stored in the database. | |
| **Attributes** | |
| List<String> searchIds | |
| ObjectId trackId | |
| **Methods** | |
| public TrackRoute() | Constructor |

| **class TrackFlight** | |
|---|---|
| This class consists of a unique track id and a search id corresponding to a specific forecast stored in the database. | |
| **Attributes** | |
| String searchId | |
| ObjectId trackId | |
| **Methods** | |
| public TrackFlight() | Constructor |

### 3.2.2.3. Feedback Package

| **class Feedback** | |
|---|---|
| This class represents the feedback given to the system by the user. | |
| **Attributes** | |
| String message | |
| String title | |
| String email | |
| **Methods** | |
| public Feedback() | Constructor |

| **class FeedbackFlight** | |
|---|---|
| This class represents the feedback given to the flight forecast by the user. | |
| **Attributes** | |
| boolean liked | |
| **Methods** | |
| public FeedbackFlight() | Constructor |

### 3.3. Prediction Server

Prediction Server consists of the logic package and the data package. The classes, attributes and methods inside the logic and data packages are explained in detail below.

#### 3.3.1. Logic Package

| **class MLManager** | |
|---|---|
| This class is responsible for responsible for routing the related requests to related Manager classes: DataManager, PredictionSystem and Scheduler. | |
| **Attributes** | |
| MLManager instance | |
| **Methods** | |
| public MLManager getInstance() | It returns a reference to the Singleton class object. |
| public ResponseEntity getDocument(String searchId) | It returns a forecast result from Mongo database in Prediction Server. |
| public ResponseEntity getGraph(String searchId) | It returns price history from Mongo database in Prediction Server. |
| public ResponseEntity predict(ForecastList forecasts) | Given forecast objects initialized with flight data, it performs prediction and store the results in the forecast objects passed as parameter. Afterwards, it saves a copy of these results in the Mongo database in Prediction Server and also return them as a ResponseEntity object. |

| **class DataManager** | |
|---|---|
| This class is responsible for both serializing objects to the virtual disk and deserializing objects from virtual disk. | |
| **Attributes** | |
| DataManager instance | |
| **Methods** | |
| public DataManager getInstance() | It returns a reference to the Singleton class object. |
| public void save(Flight flight) | It saves flight objects to virtual memory. |
| public void forecast(Forecast forecast) | Given a forecast object initialized with flight data, it performs prediction and store the results in the forecast object passed as parameter. |
| public Forecast findOne(String origin, String destination, String departureDate, String carrierCode) | It loads Forecast object to memory from Mongo database. |
| public ForecastList findAll() | It loads all of the Forecast objects in the database to memory. |

| **class PredictionSystem** | |
|---|---|
| This class is responsible for performing forecasts given an initialized forecast object | |
| **Attributes** | |
| PredictionSystem instance | |
| **Methods** | |
| public PredictionSystem getInstance() | It returns a reference to the Singleton class object. |
| public forecast(Forecast forecast) | Given a forecast object, the function performs |
| public void updateModel() | It retrain the prediction model with the new data available after the last training. |
| public void updateForecast() | It iterates over the forecasts already stored in the database and updates the prediction and notifies the corresponding users. |

| **class Scheduler** | |
|---|---|
| This class is responsible for managing the scheduled periodic tasks to crawl the updated flight data | |
| **Attributes** | |
| Scheduler instance | |
| **Methods** | |
| public Schedular getInstance() | It returns a reference to the Singleton class object. |
| public void updateForecast() | It schedule a update forecast task by using the updateForecaset function of PredictionSystem class. |
| public void scrapeQPX() | It fetches flight information from Google QPX platform. |
| public void scrapeMakeMyTrip() | It fetches flight information from Make My Trip platform. |
| public void scrapeTravelPayout() | It fetches flight information from Travel Payout platform. |
| public void setupPeriodicTasks(MQBroker sender) | Given a task, it adds it to a queue such that the task is executed periodically. |

### 3.3.2. Data Package

| **class Forecast** | |
|---|---|
| This class represents the forecast including the input parameters generated by machine learning algorithm. | |
| **Attributes** | |
| String searchId | |
| String date | |
| int expectedPrice | |
| String purchaseDate | |
| String origin | |
| String destination | |
| String carriercode | |
| int adultNum | |
| int childNum | |
| boolean oneWay | |
| boolean direct | |
| String cabinClass | |
| **Methods** | |
| public Forecast() | Constructor |

| **class PriceGraph** | |
|---|---|
| This class keeps an array of price and date values for generating the price history graph. | |
| **Attributes** | |
| List<Price> priceGraph | |
| **Methods** | |
| public PriceGraph() | Constructor |

| **class PriceGraphInner** | |
|---|---|
| This class represents a pair of date and airfare. | |
| **Attributes** | |
| int fare | |
| String date | |
| **Methods** | |
| public PriceGraphInner() | Constructor |

| **class DataCollector** | |
|---|---|
| This class is the super class for the scraper scripts. It has a collect method for scraping web content. | |
| **Methods** | |
| public void collect() | The collect method inherited and overridden by the scraper classes. |

| **class TravelPayoutScraper** | |
|---|---|
| This class overrides the collect method of the data collector to collect data from travel payout website. | |
| **Methods** | |
| public void collect() | This method connects to the travel payout API and retrieves the flight data from the user searches made in the last two days. |

| **class QPXScraper** | |
|---|---|
| This class overrides the collect method of the data collector to collect data from google qpx website. | |
| **Methods** | |
| public void collect() | This method reads the origin, destination and departure date values from a json file and collects data from the google qpx api based on the json input. |

| **class MakeMyTripScraper** | |
|---|---|
| This class overrides the collect method of data collector to collect data from make my trip website. It uses browse method to retrieve data and save flight method to store the newly obtained flight data. | |
| **Methods** | |
| public void collect() | This method generates a list of origins, destinations, departure dates and cabin classes. It calls the journey one way method and passes the input parameters to that method. |
| public Flight browse(String url, String depart_date, boolean roundTrip) | This method gets the content of the given URL by using Selenium and retrieves the relevant flight data from the web page's content. It calls the save flight method to save the flight data. |
| public void saveFlight(Flight flight, String origin, String destination, String depart_date, String cabinClass) | This method creates a json file from the flight data and saves the json into the mongo database. |
| public Flight journeyOneWay(String origin, String destination, String depart_date, String cabinClass, int adult, int child, int infant) | This method generates the search URL based on the input parameters and calls the browse method to start scraping the content of the web page with the given URL. |

| **class Flight** | |
|---|---|
| This class represents the flight data that is generated by the scrapers. | |
| **Attributes** | |
| ObjectId id | |
| String airway | |
| int fare | |
| String originAirport | |
| String destinationAirport | |
| String description | |
| String baggage | |
| boolean refundable | |
| String duration | |
| String flightType | |
| String hopping | |
| String stops | |
| String departureDate | |
| String departureTime | |
| String arrivalDate | |
| String arrivalTime | |
| **Methods** | |
| public Flight() | Constructor |

## 4. Glossary

**Flight :** An entity, which is created by using the user's input and has an origin, destination, departure date, carrier code, purchase date forecast and expected price.

**Route :** Result of the user search that consists of a list of flights. It created with the chosen constraints.

**Track :** Following a flight or route including getting updates and notifications for the tracked flight or route.

**Scraper:** Scripts used for collecting data from various websites.

## 5. References

[1] IBM, "UML - Basics," June 2003. [Online]. Available: http://www.ibm.com/developerworks/rational/library/769.html.

[2] IEEE, "IEEE Citation Reference," September 2009. [Online]. Available: https://m.ieee.org/documents/ieeecitationref.pdf. [Accessed 10 February 2018].

[3] *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.