**Integrated Assignment**

**Developing a Server Side application using Node & Express**

**Instructions to use the project file provided:**

* Read the problem statement, examples and the other details provided carefully and implement the solution
* Download the project **FBWebService** in to your system and unzip it
* Implement **app.js** as per the requirement and start your server
* Once the server started successfully, hit the **URL** **http://localhost:1050/flight/setupDb** **(GET request)** and

**http://localhost:1050/customer/setupDb** **(GET request)**to configure your mongoose database

* **DO NOT** alter the function name or the argument list of the function that is provided to you
* **DO NOT** add any new functions apart from the one given in the file where you write the solution

**Problem Description:**

**Infy Airlines** wants to automate the process of flight booking and management for which they want the implementation of the following.

* Book tickets in a flight
* View all bookings for any flight
* View all bookings done by a particular customer

**In CustomerMicroservice Folder:**

|  |  |
| --- | --- |
| **File Name** | **Description** |
| src/model/dbsetup.js | **Implemented** |
| src/utilities/connection.js |
| src/utilities/requestLogger.js |
| src/routes/routing.js | **Partially Implemented** |
| src/utilities/errorLogger.js |
| src/service/customer.js |
| src/model/customer.js |
| Customer-config.json |

**utilities/requestLogger.js (Implemented)**

This file consists of a middleware function which logs all the requests made by the users into a text file **RequestLogger.txt.**

**utilities/connection.js (Implemented)**

* This file contains the database schemas and a connection object.
* The connection object has the following two methods to establish connection with the database:

**getCustomerCollection()–** for connecting to Customer schema

The database consists of the following schema’s

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Schema** | **Field Name** | **Type** | **Validations** | **Collection** |
| **customerSchema** | customerId | String | -- | **Customer** |
| customerName | String | -- |
| customerType | String | [Platinum, Gold, Silver] |

**model/dbsetup.js: (Implemented)**

This file contains the data for the schemas and a method to insert the data into the respective database schemas.

**ErrorLogger.js: (To be implemented)**

* This file contains a **errorLogger()** function which should log the full stack trace of the error that may get thrown during execution of the program
* If there is some error in the code the entire error should be appended to the **ErrorLogger.txt** file
* If there is any error in appending the error to the **Errorlogger.txt**, it should display the message **“Failed in logging error”** in console
* If error object’s status property is set, then the **response status** should be setto error objects **status** value
* Else, the **response status** should be setto **500** and the error message should be sent as a **JSON** in the given format **{“message” :<<message>>}**
* **errorLogger.js** should be exported as a module.

**model/customer.js: (partially implemented)**

* Instance of **connection** module is created by importing it
* This file contains **customerDb** object with the following methods:
  + checkCustomer()

**Note**: Details of the mentioned methods are given in the next section

* Finally, **customerDb** object is exported as a module

**customerDb.checkCustomer() :(to be implemented)**

* This method should take **customerId** as a parameter and check whether customer with given customerId exists or not
* A connection should be established to the database by invoking the **getCustomerCollection()** method of the connection object
* It should fetch the customer details from **Customer** collection for the given customerId
* If details found, it should return the customer Details

Else, it should return **null**

**service/customer.js (partially implemented)**

* The required modules should be imported
* **customerService** object is created with the following methods:
  + checkCustomer

**Note:** Details of above methods are given in next section

* Finally, it should export **customerService** as a module

**customerService.checkCustomer(): (to be implemented)**

* This method should not take any parameter and return all the bookings over all the flights
* It should invoke the **checkCustomer()** method of **customerDb** objectwhich in turn returns the object if data found or null

**routing.js: (to be implemented)**

* It should import the required modules.
* The following URI’s should be configured for booking, fetching details & updating bookings respectively

**URI:- /customer**

It should also configure the instance of Router, in order to handle the **GET** request for the given URI.

* It should invoke **checkCustomer()** of **customerService**, passing customerId as parameter retrieved from query which in turn returns a promise wrapped response
* If the promise is successful it should populate the JSON response with the customer data returned by the **checkCustomer()** of **customerService**
* Else if the promise is failed it should forward the control to next handler by passing the error object as parameter

**app.js: (To be implemented)**

* All the required modules should be imported
* Middleware’s should be organized properly for their appropriate functionality

**customer-config.json : (Partially Implemented)**

* The application should listen to requests coming through port **3333** with the **service name** as ‘**customer’** and the **API end point** of this microservice should be ‘**customer**’

**In FlightMicroservice Folder:**

|  |  |
| --- | --- |
| **File Name** | **Description** |
| src/model/dbsetup.js | **Implemented** |
| src/model/flightBooking.js |
| src/utilities/connection.js |
| src/utilities/requestLogger.js |
| src/routes/routing.js | **Partially Implemented** |
| src/utilities/errorLogger.js |
| src/utilities/validator.js |
| src/service/flight.js |
| src/model/flight.js |
| Flight-config.json |
| app.js |

**model/flightBooking.js: (Implemented)**

* This class is used for storing and passing the **flight booking** details
* It converts a generic object to **FlightBooking** object.

**utilities/RequestLogger.js (Implemented)**

This file consists of a middleware function which logs all the requests made by the users into a text file **RequestLogger.txt.**

**utilities/connection.js (Implemented)**

* This file contains the database schemas and a connection object.
* The connection object has the following two methods to establish connection with the database:
* **getFlightCollection() –** for connecting to flightSchema

The database consists of the following schema’s

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Schema** | **Field Name** | **Type** | **Validations** | **Collection** |
| **flightBookingSchema** | customerId | String | -- | **----** |
| bookingId | Number | Unique |
| noOfTickets | Number | min-1, max-5 |
| bookingCost | Number | min-0 |
| **flightSchema** | flightId | String | -- | **Flight** |
| AircraftName | String | -- |
| Fare | Number | -- |
| availableSeats | Number | -- |
| Status | String | [Running, Cancelled] |
| Bookings | **[flight Booking Schema]** | default : [] |

**model/dbsetup.js: (Implemented)**

This file contains the data for the schemas and a method to insert the data into the respective database schemas.

**ErrorLogger.js: (To be implemented)**

* This file contains a **errorLogger()** function which should log the full stack trace of the error that may get thrown during execution of the program
* If there is some error in the code the entire error should be appended to the **ErrorLogger.txt** file
* If there is any error in appending the error to the **Errorlogger.txt**, it should display the message **“Failed in logging error”** in console
* If error object’s status property is set, then the **response status** should be setto error objects **status** value
* Else, the **response status** should be setto **500** and the error message should be sent as a **JSON** in the given format **{“message” :<<message>>}**
* **errorLogger.js** should be exported as a module.

**utilities/validator.js: (To be implemented)**

This file contains a Validator object with two methods i.e. **validateFlightId()** and **validateBookingId()**

* **validateFlightId()**
* This method should accept **flightId** as a parameter and validate it
* It should check if the **flightId** matches the following pattern: -

**<part1>>-<<part2>>**

**part1** -> **IND**

**part2**->three digits (where first digit cannot be 0)

**e.g. IND-101**

* If the validation fails, it should throw an error with message **“Error in flight Id”** after setting error status as **400**
* **validateBookingId()**
  + This method should accept **bookingId** as a parameter and validate it
  + It should check that the bookingId should always be of 4 digits only
  + If the validation fails, it should throw an error with message **“Error in booking Id”** after setting the error status as **400**

**model/flight.js: (partially implemented)**

* Instance of **connection** module is created by importing it
* This file contains **flightDb** object with the following methods:
  + generateId()
  + getAllbookings()
  + getbookingsByFlightId()
  + checkAvailability()
  + customerBookingsByFlight()
  + bookFlight()

**Note**: Details of the mentioned methods are given in the next section

* Finally, **flightDb** object is exported as a module

**flightDb.generateID() :(Implemented)**

* + This method generates unique **booking Ids** for each new document that gets inserted
  + The generated **bookingId** is returned

**flightDb.getAllBookings() :(to be implemented)**

* This method should not take any parameter and return all the bookings made in all the flights.
* A connection should be established to the database by invoking the **getFlightCollection()** method of the connection object
* It should fetch only the bookings field from all the flights.
* If bookings found, it should return the same **bookings** array
* Else, it should return **null**

**flightDb.** **getbookingsByFlightId() :(to be implemented)**

* This method should take flightId as a parameter and return all the bookings made in that flight
* A connection should be established to the database by invoking the **getFlightCollection()** method of the connection object
* It should fetch only the bookings detail from the flight collection for the given flightId.
* If bookings are found, it should return the same **bookings** array
* Else it should return **null**

**FlightBookinDb.checkAvailability() :(to be implemented)**

* This method should take **flightId** as a parameter and check whether flight with given flightId exists or not
* A connection should be established to the database by invoking the **getFlightCollection()** method of the connection object
* It should fetch the customer details from **Flight** collection for the given flightId
* If details found, it should return the flight Details
* Else, it should return **null**

**flightDb.** **customerBookingsByFlight () :(to be implemented)**

* This method should take **customerId** and **flightId** as a parameter and return an array of all the bookings made by the customer in the given flight
* A connection should be established to the database by invoking the **getFlightCollection()** method of the connection object
* It should fetch all the bookings made by a customer for the given flightId
* If bookings are found, it should return the same array of **bookings**
* Else, it should return **null**

**flightDb.bookFlight() :(to be implemented)**

* This method should accept **FlightBooking** Object as a parameter, book tickets for a valid customer, update the number of seats available for the flight and the walletAmount of the customer.
* A connection should be established to the database by invoking the **getFlightCollection()** method of the connection object
* It should invoke **generateId()** method of **flightDb** to generate unique booking Id for each flight booking.
* A new document should be inserted into the **bookings** column of Flight collection with the values of **customerId, bookingId, noOfTickets, bookingCost.**
* It should check for the successful insertion of document into the collection
* If successfully inserted, it should update the number of seats available in the **Flight** Collection for the corresponding **flightId**
  + It should also check for successful updation of seats available in the **Flight** collection.
  + If successfully updated, it should return the bookingId.
  + Else, if the seats available for the flight does not get updated, it should throw an error with message **“Seats not updated”** after setting the error status as **502 (Bad gateway)**.
* Else, it should throw an error with message **“Booking failed”** with status **500**

**service/flight.js (partially implemented)**

* The required modules should be imported
* **flightService** object is created with the following methods:
  + getAllBookings()
  + getBookingsByFlightId()
  + customerBookingsByFlight()
  + bookFlight()

**Note:** Details of above methods are given in next section

* Finally, it should export **flightService** as a module

**flightService.getAllBookings() : (to be implemented)**

* This method should not take any parameter and return all the bookings over all the flights
* It should invoke the **getAllBookings()** method of **flightBookingDb** objectwhich in turn returns the array of booking details of all the flights
* If the returned value is **null,** it should throw an error with message **“No Bookings is found in any flight”** after setting error status as **404**
* Else, it should return the same array of bookings

**flightService.getbookingsByFlightId() :(to be implemented)**

* This method should take flightId as a parameter and return the corresponding booking details
* It should invoke the **getbookingsByFlightId()** of **flightBookingDb** by passing **flightId** which in turn returns the corresponding booking details for that flight
* If the returned value is **null,** it should throw an error with message ***“No Bookings found in <<flightId>>”*** after setting error status as **404**
* Else, it should return the same booking details

**flightService.customerBookingsByFlight() : (to be implemented)**

* This method should take customerId and flightId and return the corresponding customer details
* It should invoke the **checkAvailability()** method of **flightBookingDb** by passing flightId as parameter,which in turn returns the corresponding flight details
* If the returned value is null, it should throw an error with message **“Invalid FlightId!! Enter a valid FlightId to view Details”**
* Else, it should invoke the **customerBookingsByFlight()** method of **flightBookingDb** by passing customerId and flightId as parameter which in turn returns the booking details for a customer in the given flightId.
* If the returned value is **null,** it should throw error with message ***“No Bookings found for <<customerId>> in <<flightId>>”*** after setting error status as **404**
* Else, it should return the same booking details

**flightService.bookFlight() :(to be implemented)**

* This method should take **FlightBooking** object as a parameter and book the mentioned number of tickets for an existing Flight.
* It should validate the flightId by invoking appropriate validator method.
* It should invoke **checkAvailability()** method of **flightBookingDb** object by passing the flightId entered by the user as a parameter which in turn returns flight details from the Flight collection
* If the returned value is **null**, it should throw an error with message **“Flight Unavailable”** after setting error status **404**
* If the status of the returned flight is **“Cancelled”,** it should throw an error with message **“Sorry for the Inconvinience... *<<flightId>>* is cancelled!!”** after setting error status **400**
* If the **availableSeats** of the returned flight is **0,** it should throw an error with message **“Flight *<<flightId>>* is already full!!”** after setting appropriate error status
* If the **availableSeats** of the returned flight isless than the required **nOOfTickets,** it should throw an error with message **“Flight almost Full... Only <<availableSeats>> left!!”** after setting error status as **406**
* Else, it should assign the bookingCost value to the **FlightBooking** object **(bookingCost = noOfTickets \* fare)**
* It should invoke the **bookFlight()** method of **flightBookingDb** object which in turn returns the booking details
* Finally, it should return the same booking details

**routing.js: (to be implemented)**

* It should import the required modules.
* The following URI’s should be configured for booking, fetching details & updating bookings respectively

**URI:- /flight/getAllBookings**

It should also configure the instance of Router, in order to handle the **GET** request for the given URI.

* It should invoke **getAllBookings()** of **flightService**, which in turn returns a promise wrapped response
* If the promise is successful it should populate the JSON response with the bookings returned by the **getAllBookings()** of **flightService**
* Else if the promise is failed it should forward the control to next handler by passing the error object as parameter

**URI:- /flight/bookingsByFlight/:flightId**

It should also configure the instance of Router, in order to handle the **GET** request for the given URI

* It should invoke **getbookingsByFlightId()** of **flightService** objectby passing the **flightId** obtained from the **request params** object, which in turn returns a promise wrapped response
* If the promise is successful it should populate the JSON response with the bookings returned by the **getbookingsByFlightId()** of **flightService**
* Else if the promise is failed it should forward the control to next handler by passing the error object as parameter

**URI:- /flight/customerBookings/:customerId/:flightId**

It should also configure the instance of Router, in order to handle the **GET** request for the given URI

* It should create a **UMF message** with the following data
  + to: “**customer:[get]/customer?customerId= << customerId retrieved from params>>**“
  + from: ‘Flight\_service’
  + body: {}
* After the message is created, make an API request and send the above created UMF message, which in turn returns a response inside a promise.
* If the response.data value is **null,** it should throw error with message ***“*Invalid CustomerId!! Enter a valid customerId to view Details*”*** after setting error status as **404**
* Else, it should invoke **customerBookingsByFlight()** of **flightService** objectby passing the **customerId** and **flightId** obtained from the **URL**, which in turn returns a promise wrapped response
* If the promise is successful it should populate the JSON response with the bookings returned by the **customerBookingsByFlight()** of **flightService**
* Else if the promise is failed it should forward the control to next handler by passing the error object as parameter.

**URI:- /flight/bookFlight**

* It should configure the instance of Router, to handle the **post** request for the given URI.
* Once the request is received, it should create and populate **FlightBooking** object with the values present inside the request body.
* It should create a **UMF message** with the following data
  + to: “**customer:[get]/customer?customerId= << customerId retrieved from params>>**“
  + from: ‘Flight\_service’
  + body: {}
* After the message is created, make an API request and send the above created UMF message, which in turn returns a response inside a promise.
* If the response.data value is **null,** it should throw error with message ***“*Invalid CustomerId!! Enter a valid customerId to view Details*”*** after setting error status as **404**
* Else, it should invoke **bookFlight()** method of **fBookingService** by passing the **FlightBooking** object, which in turn returns a promise wrapped response
* If the promise is successful it should populate the JSON response in the given format: ***{"message": "Flight booking is successful with booking Id: <<bookingId>>”}*** with status as **201**
* Else if the promise fails, it should forward the control to next handler by passing the error object as parameter

**app.js: (Partially implemented)**

* All the required modules should be imported
* Middleware’s should be organized properly for their appropriate functionality

**Flight-config.json : (Partially Implemented)**

* The application should listen to requests coming through port **2222** with the **service name** as ‘**flight**’ and the **API end point** of this microservice should be ‘**flight**’