**Hands-on Assessment**

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

**Time: 3 Hrs. Mark: 50**

**Instruction to Candidates:**

* It is a controlled open book test. Please follow the guidelines provided by Invigilator.
* The code will be evaluated only after submitting it to the submission folder (server).
* **If the timestamp of the file is greater than the submission time the test will not be evaluated.**
* **You are not supposed to use any network resources.**

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

* You need to code the assignment in the folder containing partially coded project.
* Read the problem statement, examples and the other details provided carefully and implement the solution.
* Download the project **PCWebService** in to your system and unzip it
* A file called **sampleInput.txt** inside your project folder has been provided. You can use this file to test your application
* Root folder of your project should be PCWebService. Run **index.html** inside **parserModule** in live server to verify your code. Your application should not have any run-time/compilation errors for successful verification
* **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
* **DO NOT** write codes that result in infinite loops/infinite recursive calls, because it just won’t work!

**Submission guideline**: Create a folder on the desktop with your **<Empno>.** Copy the **src** folder from the artifacts folder to the <Empno> folder while submission.

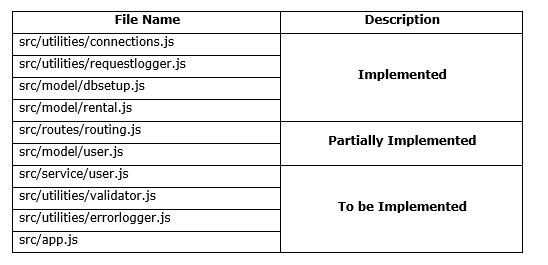
**Note:** Script will not be evaluated in case of incorrect submission

**Problem Description:**

**Pizza Castle Delivery Unit:** Pizza Castle is the newly opened Pizza restaurant in the town, who wants to automate their delivery system for providing hot and yummy pizza to their customers in their home. The following functionalities are required to be implemented.

**Implementation Details of Server side application.**

**In PCWebService folder:**



**src/model/booking.js: (Implemented)**

* This file has **Booking** class which can be used to create Booking object, used for storing and passing **Pizza** details.
* orderId
* deliveryDate
* amount
* pizzaName
* quantity

**src/utilities/requestlogger.js: (Implemented)**

RequestLogger is used for logging the Date and timestamp, request method and URI in logger file **RequestLogger.txt.**

**src/utilities/connections.js: (Implemented)**

* This file contains connection methods for establishing connection & creating model objects that can be used to interact with the MongoDb database
* The **getCollection** method returns a model object of **Booking** collection
* If the connection is not established, it throws an error with message “**Could not connect to Database**” with error status **500.**

**src/utilities/errorlogger.js: (To be implemented)**

* This file has a function **errorLogger**, which should log the full stack trace of the error that is thrown during execution of the program
* If there is any error in the code the entire error should be appended to the **ErrorLogger.txt** file
* If there is some error in appending the error to the **Errorlogger.txt**, it should display the message **“Could not log the errors”** in console
* If error object’s status property is set, then the response status should be set to error objects status value
* Else, the response status should be set to **500** and the error message should be sent as a JSON in the given format {“message” :<<message>>}
* **errorLogger** should be exported as a module

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

* **validateCustomer(customerId)**
  + Write a validation for the **customerId** to ensure that the customer Id entered by the user contains exactly **5 digits** and must **start with** **2**.
  + If the validation fails, throw an error message “**Invalid! CustomerId must start with 2 and should be a 5 digit number**” and the error status should be **406***(Not Acceptable).*
* **validateDate(deliveryDate)**
  + Write a validation for the **deliveryDate** to ensure that the delivery can be on the same day or any other future date.
  + If validation fails, throw an error message **“Date of delivery must be today or greater than today”** and the error status should be **406***(Not Acceptable).*
* **availability(ordObj)**
* Write a validator that will check whether the quantity of each pizza is mentioned or not. If validation fails, throw an error message **“Please check the order again”** and the error status should be 406.
* If the above validation passes, now write a validator that will check whether the Pizza Ordered is in stock or not. The list of Pizza available are provided as an array .Use this array and check for the availability.

If the validation fails, throw an error with message **“One or more Pizza is currently unavailable”** and the error status should be 406.

* **delivery(cost)**
* Write a validator to ensure that delivery is available at amount greater than 250.
* If the above validation fails, throw an error with message **”Delivery is available at amount above 250”** and error status should be 406.

**src/model/user.js: (Partially implemented)**

* This file has an instance of **PizzaOrder** collection and **bookingDb** object
* **bookingDb** object has the following methods
  + generateId()
  + customerDetails (customerId)
  + bookPizza(customerId, bookObj)
  + **bookingDb** object is exported as a module.

**bookingDb.generateId(): (Implemented)**

* + This method generates unique **order Ids** for each new order that has been taken.
  + It returns the newly generated **orderId**

**bookingDb.customerDetails (customerId): (To Be Implemented)**

* This method is used to check whether the given customer is existing or not for the given **customer Id**
* If details of the customer in the **PizzaOrder** collection for the given **customerId** exists returns the customer object.
* Else, return **null**

**bookingDb.bookPizza(customerId,bookObj): (To Be Implemented)**

* This method should book the order for a given **Customer.**
* It should invoke the **generateId** method of **bookingDb** which in turn returns the newly generated **orderId** wrapped inside *promise* object, it should populate the newly generated **orderId** into the **bookObj** object.
* Then establish connection with **PizzaOrder** collection by invoking **getCollection** method of **connection.js**
* It should update the order details in **bookObj** to **orders** in PizzaOrder collection
* It should check for the **updation** of the new document into the **PizzaOrder** collection. On successful updation, return **bookObj.**
* Else return **null**

**src/service/user.js: (To be implemented)**

* Import and create instances of required **modules**
* **orderService** object is created with the following method:
  + viewcustomer(customerId)
  + order(customerId, orderObj)
  + customerdetails(customerId)
  + calculateamount(bookObj)
* **orderService** should be exported as a module

**orderService.viewCustomer(customerId): (To be Implemented)**

* This method should check whether the customer details are available or not for the given customerId
* It should invoke **customerDetails** method of **model/user.js** by passing customerId as parameter, which in turn returns customer object or null
* If it returns customer object return the same
* Else throw an error with a message **“Customer details not found”** and status 404

**orderService.customerDetails(customerId): (To be Implemented)**

* This method should return the order details for the given customerId
* It should validate the customerId using validateCustomer method of validator.js
* It should invoke **viewCustomer** method of **orderService** by passing the customerId as the parameter, which returns the customer object
* If the customer has ordered any pizza then return the **order details array**
* Else it should throw an error with a message **“No Order Details found”** and status **404**

**orderService.calculateAmount(bookObj): (To be Implemented)**

* This method in **orderService** should take **bookObj** object as parameter and based on the Pizza Ordered and quantity, it should calculate the cost.
* A object named as **items** is given which has the key as Pizza Name and the value to be their corresponding cost.
* With reference to the object items, Calculate the cost of one type of Pizza by the following method:

Amount = quantity\*Cost of Pizza

Similarly calculate the total cost of the order by adding all the amount of different Pizza if any.

* Set the amount property of Booking object with the calculated total cost

**orderService.order(customerId, orderObj): (To be Implemented)**

* This method in **orderService** should take order object as parameter and then finalize the order for the given customer
* It should validate the **customerId** by invoking **validateCustomer()** function by passing the **customerId** as the parameter.
* It should also validate the delivery date by invoking **validateDate()** function by passing the **deliverydate** as the parameter.
* Similarly validate the available Pizza by invoking **availability()** function by passing the **orderObj** as the parameter.
* It should check if the given **customer** existsfor the given **customerId** by invoking the **viewCustomer(customerId)** method of **orderService**
* Then invoke **calculateAmount(bookObj)** of the **orderService** to calculate the total amount.
* Then call the **delivery()** function of validator by providing **total calculated cost** as the parameter and validate whether delivery is feasible or not.
* Then invoke **bookPizza()** method of **model/user.js** by passing **customerId** and **bookObj** as parameter, which in turn return a **bookObj** object wrapped inside a *promise*, return the same object.
* If the return value from the **bookPizza()** method of **model/user.js** is **NULL,** throw an error with a message “**Failed to update data”** and status **500**

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

* Import all the required modules

For PUT **/book/:customerId**

* It should configure the instance of *Router* to handle the put request with an URI **‘/book/:customerId’**
* Once request is received:
  + Convert the generic object inside request body to book object using the Booking class in **model/booking.js**
  + Convert the customerId from the request URL to Number type
* Invoke **order()** of **service/user.js** by passing the **customerId** and **bookingObj** object, which in turn returns a *promise* containing **order** object
* If the promise is successful, send the JSON response in the below format:

***{"message": “Order Succesfully placed with " + <orderId>+ “, amount to be paid Rs:“+ <total amount> }***

* Else if the *promise* is failed, forward the control to next handler by passing the error object as parameter

For GET **/book/:customerId**

* It should configure the instance of *Router* to handle the GET request with an URI **‘/book/:customerId’**
* Once request is received:
  + Convert the customerId from the request URL to Number type
* Invoke **customerdetails()** of **service/user.js** by passing the **customerId**, which in turn returns a *promise* containing **array of order details**
* If the promise is successful, send the array of order details as response
* Else if the *promise* is failed, forward the control to next handler by passing the error object as parameter
* **Routing.js** should be exported as a module

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

* Import all the required modules
* Organize middlewares in proper order to make the application function appropriately
* All the requests sent to the application should be logged
* All the Errors thrown from the application should be logged
* The application should listen to requests coming through port **3000**