Traffic Light

**Description**

**Problem Statement**

You are assigned to implement a program that simulates a traffic light system using **enums** in C programming. The program should display the current color of the traffic light and switch to the next color based on a predefined sequence.

Instructions:

1. Define an **enum** type called **TrafficColor** With three constants representing the colors of a traffic light: **RED**, **YELLOW**, and **GREEN**.
2. Declare a variable of type **TrafficColor** to store the current color of the traffic light.
3. Implement a function called **nextColor** that takes a **TrafficColor** parameter and returns the next color in the sequence according to the traffic light pattern (**RED -> GREEN -> YELLOW -> RED**).
4. In the main function, initialize the current color of the traffic light to **TrafficColor**.
5. Implement a function called **printColor** that takes a **TrafficColor** parameter and display the Enum value as:

* **RED: "RED"**
* **GREEN: "GREEN"**
* **YELLOW: "YELLOW"**
* **Note**: Add **'\n'**for the next line after every color display.

**Evaluation Parameters**

**Sample Input**

2

2

1

**Sample Output**

**YELLOW**

GREEN

**Explanation**

->First, 2 define the total number of colors.

->For input purposes, we define 1 as "RED," 2 as "GREEN," and 3 as "YELLOW."

->The current color initially is "GREEN," so after the next color, it becomes "YELLOW."

->The next current color is "RED," so after the next color, it becomes "GREEN."

------------------------------------------------------------------------------------------------------------------

Licious

**Description**

Your task here is to implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications:**

enum definition:

﻿ enum Type {MEAT, FISH, OTHER}

﻿

class definitions:

class Dish:

﻿data members:

﻿final name: String

﻿ final vegetarian: boolean

﻿ final calories: int

final type: Type﻿

visibility: private

﻿

﻿Dish(String name, boolean vegetarian, int calories, Type type): constructor with public visibility

﻿ Define getter with public visibility

﻿ toString(): has been implemented for you

class DishImplementation:

﻿method definition:

﻿ getNameAndCalories(List<Dish> menu):

﻿ return type: List<Dish>

visibility: public

threeHighCaloricDish(List<Dish> menu):

﻿ return type: List<String>

visibility: public

isVegetarian(List<Dish> menu):

return type: boolean

visibility: public

**Task:**

**class Dish:**

**-**define class **Dish** according to the above specifications

class **DishImplementation:**

**Implement the below method for this class**:

* **List<Dish> getNameAndCalories(List<Dish> menu): f**ilter the dishes based on :
* type **OTHERS**
* calories greater than equal **200** and less than **560**

put the filtered dishes into a List of Dish and return the list

* **List<String> threeHighCaloricDish(List<Dish> menu):**fetch the first three dishes with calories greater than **300**, put it into a list and return the list
* **boolean isVegetarian(List<Dish> menu):**return **true** if the list of dish contains vegetarian else return **false**

*Refer sample output for clarity*

**Sample Input**

**List**<Dish> menu = Arrays.asList(

**new** Dish("pork",**false**,800,Type.MEAT),

**new** Dish("french fries", **true**, 530,Type.OTHER),

**new** Dish("rice",**true**,120,Type.OTHER),

**new** Dish("pizza", **true**, 550, Type.OTHER),

**new** Dish("salmon",**false**,450,Type.FISH) );

DishImplementation i = **new** DishImplementation();

i.getNameAndCalories(menu)

i.threeHighCaloricDish(menu)

i.isVegetarian(menu)

**Sample Output**

[Dish{name='french fries', vegetarian=true, calories=530, type=OTHER}, Dish{name='pizza', vegetarian=true, calories=550, type=OTHER}]

----------------------------------------------------------

[pork, french fries, pizza]

----------------------------------------------------------

true

**NOTE**

* You can make suitable function calls and use **the RUN CODE** button to check your **main()** method output.

------------------------------------------------------------------------------------------------------------------

Spring has come

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

**class** Definition:

﻿ **class** SeasonExample:

**enum** definition:

**enum** Season

        SPRING,

        SUMMER,

        FALL,

        WINTER

  visibility: public

method definitions:

**static** getSeason(**int** month):

**return** type: String

visibility: public

﻿

**static** printSeason(Season season):

**return** type: String

visibility: public

**Task**

**Class** **SeasonExample**

**-Define enum Season** according to the above specifications

**-Implement the below methods for this class:**

**->static String getSeason(int month):**

* Take an integer parameter representing the month (1-12) and return the corresponding season based on the traditional division of the year into four seasons:
* Spring: March (3) to May (5)
* Summer: June (6) to August (8)
* Fall: September (9) to November (11)
* Winter: December (12), January (1), and February (2)

**->static String  printSeason(Season season):**

* Take Season parameter and display the month in a string as given:
* SPRING: "SPRING Season"
* SUMMER: "SUMMER Season"
* FALL: "FALL Season"
* WINTER: "WINTER Season"

**Sample Input**

SeasonExample se = **new** SeasonExample();

String season = se.getSeason(7);           se.printSeason(SeasonExample.Season.valueOf(season));

**Sample Output**

**SUMMER** Season

**NOTE:**

* You can make suitable function calls and use **the RUN CODE** button to check your **main()** method output.

------------------------------------------------------------------------------------------------------------------

Collections in java

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

**class** **definitions**:

**class** **ArrayListOps**:

 method definitions:

  convertArrayListtoInt(**int** n):

**return** type: ArrayList<Integer>

      visibilty: **public**

  reverse(ArrayList<Integer> list):

**return** type: ArrayList<Integer>

      visibilty: **public**

You don't need to implement the main() method. It has already been implemented as a part of the test-cases. It contains an **ArrayList** of integers.

**Task:**

Your task is to create a **Source**and implement the following:

* **convertArrayListtoInt(int n):**Method to create an **arrayList** with number of components **n** and ***set***components to **0**
* **reverse(ArrayList<Integer> list):** Method to ***Reverse* list**

**Important:**

* To check your program, you can use the **main()** method (in Source class) given in the stub. You can make suitable function calls and use **RUN CODE** button to check your main() function output.**﻿**

**Sample Input**

ArrayList<Integer> list = **new** ArrayList<Integer>(Arrays.asList(10, 25, 33, 28, 10, 12));

**Sample Output**

[0, 0, 0, 0]

[12, 10, 28, 33, 25, 10]

**Allowed technologies**

* Java 8

------------------------------------------------------------------------------------------------------------------

Jsp Librarian

**Description**

As part of a new online book club platform, you've been tasked with developing a book entry feature. Users should be able to add books to the platform's database seamlessly. To ensure a smooth user experience, you must handle various scenarios, such as successful book entries and errors during the process.

**JSP Pages**

**index.jsp**

1. Create a book entry form with the following fields:

* **Title** (input type text)
* **Author** (textarea)
* **Genre** (dropdown with options: Fiction, Non-Fiction, Mystery, Romance)
* **Publication Year** (input type number)
* Include a **submit button** with the value "**AddBook**" and id as "**addBook**"

**success.jsp**

* If the book entry is successful, display a success message:**"Book added successfully**".

**error.jsp**

* If there is an error during book entry (e.g., empty fields or invalid year), display an error message: "**Failed to add book**".

**Book**

Implement a Book class to represent book data with the following private fields:

* title
* author
* genre
* publicationYear

Include appropriate constructors, getters, and setters.

**AddBookServlet**

Implement a servlet to handle book entry.

* Retrieve parameters (title, author, genre, publicationYear) from the request.
* Check if all parameters are not null and not empty.
* Validate the publication year to ensure it is a positive integer.
* If the parameters are valid, create a new Book object and redirect the user to the success page **{proxy\_uri}/success.jsp**.
* If parameters are invalid, redirect the user to the error page **{proxy\_uri}/error.jsp.**

**Note**

* **Environment Variable Retrieval:** The servlet accesses an environment variable named "VSCODE\_PROXY\_URI" using System.getenv() method. This variable contains the URI of a proxy server, which is crucial for redirecting users to appropriate JSP pages.
* **Redirecting Users:** Depending on the authentication result, the servlet redirects users to either a success or error JSP page. The redirection URLs are constructed based on the proxy URI retrieved from the environment variable.

------------------------------------------------------------------------------------------------------------------

Spring Cab Booking management

**Description**

**Problem Description**

In the bustling city of Metroville, **ZoomRide**, a popular ride-hailing platform, aims to offer seamless transportation experiences for its users. To enhance the platform's efficiency, the **RideService** layer must be implemented with three key functionalities: booking a ride, retrieving recent completed rides, and ending a ride.

**a. model package**:

->Contains the Driver, RideDetails and User entity class with attributes, Constructors, getters and setters.

->The **Driver** entity should have the following attributes:

* **id** : The unique identifier for the driver (Long, Primary Key).
* **availability** : Indicates if the driver is currently available. It uses a BIT(1) type to represent true (available) or false (not available). (Boolean).
* **name** : The name of the driver, with a maximum length of 255 characters. It can be NULL if not specified. (String).
* **rating** : The rating of the driver, representing their performance. This field is mandatory and cannot be NULL.(Double)
* **vehicleDetails** : Contains details about the vehicle used by the driver, with a maximum length of 255 characters. This field can also be NULL.(String)

->The **User** entity should have the following attributes:

* **id** : The unique identifier for the user (Long, Primary Key).
* **email** : The email address of the user, with a maximum length of 255 characters. It can be NULL.(String)
* **name** : The name of the user, with a maximum length of 255 characters. This field can be NULL if not specified.(String)
* **paymentMethod** : Indicates the payment method used by the user (e.g., credit card, cash). This field has a maximum length of 255 characters and can be NULL.(String)
* **phone** : The contact number of the user, with a maximum length of 255 characters. It can also be NULL.(String)

->The **RideDetails** entity should have the following attributes:

* **id** : The unique identifier for the ride. (Long, Primary Key).
* **driverId** : The ID of the assigned driver for this ride (Foreign Key referencing the driver table).(Long)
* **endLocation** : The destination of the ride, with a maximum length of 255 characters. This field is mandatory and cannot be NULL.(String)
* **endTime** : The end time of the ride. This field can be NULL if the ride is ongoing.(LocalDateTime)
* **fare** : The fare for the ride, which is mandatory and cannot be NULL.(Double)
* **rideStatus** : The current status of the ride, which can be "Ongoing" or "Completed". This field is mandatory and cannot be NULL.(String)
* **startLocation** : The starting point of the ride, with a maximum length of 255 characters. This field is mandatory and cannot be NULL.(String)
* **startTime** : The start time of the ride. This field is mandatory and cannot be NULL.(LocalDateTime)
* **userId** : The ID of the user who booked the ride (Foreign Key referencing the user table).(Long)

**b. repository package**: Contains the interface DriverRepository, RideDetailsRepository and UserRepository extending JpaRepository.

**c.** **service package**: Contains RideService class with some method signature performing business logics

**d.** **controller package**: Contains RideController with skeleton CRUD API endpoints.

**Task 1: Book a Ride**

**Complete the method with logic as per the requirement.**

* **Function Name**: bookRide(Long userId, String startLocation, String endLocation, double distance):
* **Return Type**: RideDetails
* **Functionality**: If no drivers are available, throw an exception indicating "No available drivers". Otherwise, calculate the fare based on the distance (Rs.25 per KM) and return the ride details, including the first driver available and the fare calculated.

**Task 2: Get Recent Completed Rides**

**Complete the method with logic as per the requirement.**

* **Function Name**: getRecentCompletedRides(Long userId)
* **Return Type**: List
* **Functionality**: Retrieve completed rides for the user within the last 30 minutes. Return a list of RideDetails that match the criteria.

**Task 3: End a Ride**

**Complete the method with logic as per the requirement.**

* **Function Name**: endRide(Long rideId)
* **Return Type**: RideDetails
* **Functionality**: End the specified ride by updating its status and setting the end time. If the ride ID does not exist, throw an exception indicating**"Ride not found"**.

**NOTE:**

* The required endpoints are provided in the **RideController.java** file so that you can check them easily.

------------------------------------------------------------------------------------------------------------------

SpringBoot Automative Parts management

**Description**

**Problem Description**

The automotive supply chain management system at a major vehicle manufacturer has been facing challenges with managing parts inventory. The system needs to efficiently handle orders for parts, monitor stock levels, and predict when restocking will be necessary to avoid production delays. As part of a larger overhaul of the inventory system, your team is tasked with implementing several key functionalities for managing parts and orders in the company's backend system.

**a. model package**:

->Contains the Orders and Part entity class with attributes, Constructors, getters and setters.

->The **Orders** entity should have the following attributes:

* **id** : The unique identifier for the order (Long, Primary Key).
* **partId** : The ID of the part associated with the order (Long).
* **quantity** : The quantity of the part ordered (Integer).
* **status** : The current status of the order (String)-IN\_PROGRESS, COMPLETED, CANCELLED.
* **orderDate** : The date of the Order Placed.(Date).

->The **Part** entity should have the following attributes:

* **id** : The unique identifier for the part (Long, Primary Key).
* **name** : The name of the part (String).
* **manufacturer** : The manufacturer of the Part. (String).
* **price** : The price of the part (Double).
* **stockQuantity** : The current stock level of the part (Integer).
* **minStockLevel** : The minimum stock level to be maintained of the part (Integer).

b. **repository package**: Contains the interface OrderRepository and PartRepository extending JpaRepository.

c. **service package**: Contains OrderService and PartService class with some method signature performing business logics

d. **controller package**: Contains OrderController and PartController with skeleton CRUD API endpoints.

**Requirements**

A. Your task is to Implement Service Layer Implementation in **PartService**.

Implement the logic in the service layer using PartRepository and OrderRepository.

**updatePartStock(Long partId, int quantity):**

**Return Value**: The updated Part object after the stock quantity has been successfully modified.

**Functionality**:

* Retrieves the part by its ID (partId) from the repository. If no part is found, an exception is thrown with the message **"Part not found"**
* Validates the new stock level by checking if the update would result in a negative stock quantity. If so, it throws an exception with the message **"Insufficient stock"**
* If the stock level is valid, the method updates the part's stock quantity and saves the changes in the repository.

**getLowStockParts():**

* **Return Type**: List
* **Functionality**: Complete the method to return a list of parts where the stock quantity is below a defined threshold(20).

**predictRestockDate(Long partId)**

**Return Value**:

* Return an integer value representing the number of days until the part’s stock reaches the buffer level.
* Return 0 if the stock is at or below the buffer stock, indicating immediate restock is required.

**Functionality**:

* Retrieves the part by its ID (partId) from the repository. If the part does not exist, the method throws a RuntimeException with the message**"Part not found"**
* Uses a predefined average daily consumption rate to estimate how long the current stock will last before reaching a predefined buffer stock level.
* If the current stock is already below or equal to the buffer stock, it returns 0, indicating that restocking is needed immediately.

**Assumptions**:

* Average Daily Consumption: Assumed to be a fixed value (5.0 units per day).
* Buffer Stock: A constant buffer of 10 units is maintained as the minimum stock level before restocking is triggered.

**Calculation**:

* If the current stock is greater than the buffer stock, the method calculates how many days it will take for the stock to fall to the buffer level, given the average daily consumption rate.
* The formula for predicting restock days is:

daysUntilOutOfStockInteger

=

(

currentStock

−

bufferStock

)

averageDailyConsumption

daysUntilOutOfStock=

averageDailyConsumption

(currentStock−bufferStock)

* The result is rounded to the nearest whole number and returned as an integer.

B. Your task is to Implement Service Layer Implementation in **OrderService**. Implement the logic in the service layer using OrderRepository and PartRepository.

Complete the methods with logic as per the requirement.

**createOrder(Long partId, int quantity):**

* This method creates an order for a part with a given partId and quantity.Complete the method to fetch a part, reduce the stock quantity by the requested amount in current requested Order, and create a new order with the status **"IN\_PROGRESS".**
* If the partId is invalid (i.e., no part is found), throw a RuntimeException with the message: **"Part not found"**
* If there is insufficient stock for the given quantity, throw a RuntimeException with the message: **"Insufficient stock"**

**processBulkOrders(List<Orders> orders):**

* This method processes a list of orders in bulk, using the createOrder method for each order. It collects successfully created orders and returns them.
* If an order fails during the creation process (due to reasons like part not being found or insufficient stock), an error message is logged: **"Failed to create order for part ID " + order.getPart().getId() + ": " + e.getMessage()**.

**completeOrder(Long orderId)**

* Complete the method to fetch an order by its ID and update the status of the order to **"COMPLETED"**.
* If the orderId is invalid or the order is not found, it throws a RuntimeException with the message: **"Order not found".**

C. Your task is to implement a set of **REST** services that exposes the endpoints.

| API Route | API Type | Success Response Code | Validation Error Code |

|---------------------------|----------|------------------------|------------------------|

| /api/orders | POST | 201 | 400 |

| /api/orders/bulk | POST | 200 | 400 |

| /api/orders/{id}/complete | PUT | 200 | 404 |

| /api/orders | GET | 200 | N/A |

| /api/orders/{id} | GET | 200 | 404 |

**Task 7: REST API Endpoints in OrderController**

**POST request to /api/orders**

* Create Order: This endpoint creates a new order for a specific part.
* **HTTP Status Code**:
* 201 - For a successful order creation.
* 400 - If parameters are invalid or missing.

**POST request to /api/orders/bulk**

* Process Bulk Orders: This endpoint processes a list of orders in bulk.
* **HTTP Status Code**:
* 200 - For successful bulk order processing.
* 400 - If the request body is null or empty.
* 500 - If no orders could be placed due to an internal error.

**PUT request to /api/orders/{id}/complete**

* Complete Order: This endpoint marks an order as completed.
* **HTTP Status Code**:
* 200 - If the order is successfully marked as complete.

**GET request to /api/orders**

* Get All Orders: This endpoint retrieves a list of all orders.
* **HTTP Status Code**:
* 200 - If the request is successful.

**GET request to /api/orders/{id}**

* Get Order by ID: This endpoint retrieves the details of a specific order by its ID.
* **HTTP Status Code**:
* 201 - If the order is found.
* 404 - If the order with the given ID is not found.

------------------------------------------------------------------------------------------------------------------

SpringBoot Delivery Application

**Description**

Implement REST APIs to facilitate managing delivery orders and customer details for a delivery application system. The APIs should enable users to update delivery order information based on specific criteria, calculate average delivery duration, and get the count of delivered orders for all customers.

Here is an example of a Customer JSON object:

{

"customerId": 1,

"name": "Charlie Adams",

"email": "charlie.adams@example.com",

"phoneNumber": "+1212345678"

}

Here is an example of a DeliveryPerson JSON object:

{

"deliveryPersonId": 1,

"name": "Alice Blue",

"vehicleType": "Bike",

"status": "Active"

}

Here is an example of a DeliveryOrder JSON object:

{

"orderId": 3,

"customerId": 5,

"deliveryPerson": {

"deliveryPersonId": 4,

"name": "Sarah Yellow",

"vehicleType": "Car",

"status": "Active"

},

"items": "Book,Notebook",

"pickupLocation": "456 Walnut St",

"deliveryLocation": "789 Pine Ln",

"status": "PLACED",

"orderTime": "2024-07-04T08:00:00",

"pickupTime": "2024-07-04T08:20:00",

"deliveryTime": "2024-07-04T09:00:00"

}

You are provided with the implementation of the models required for all the APIs. Your task is to implement a set of REST services that expose the endpoints, enabling users to update delivery order information based on specific criteria, calculate average delivery duration, and get the count of delivered orders for all customers.

| API Route | API Type | Success Response Code| Validation Error Code|

|---------------------------------------------------------|----------|--------------------- |----------------------|

| /api/delivery-orders/{orderId}/status | PUT | 200 | 400, 404, 500 |

| /api/delivery-orders/average-delivery-duration | GET | 200 | 404, 500 |

| /api/delivery-orders/delivered-orders-count-by-customer | GET | 200 | 404, 500 |

**Task 1: Service Layer Implementation in DeliveryOrderService**

Implement the logic in the service layer using DeliveryOrderRepository, CustomerRepository, and DeliveryPersonRepository:

**Method Details**:

* **updateOrderStatus()**: It should update the status of a delivery order using the specified orderId and status. If the orderId is found, save the the delivery order status. If the orderId is not found, a NoSuchElementException should be thrown with the message "Order not found".
* **calculateAverageDeliveryDuration()**: It should calculate the average delivery duration (in minutes) for all the orders that are delivered. The duration is the time between pickupTime and deliveryTime, rounded to two decimal places. If there are no delivered orders, return null. For delivered orders, return the average delivery duration rounded to two decimal places.
* **getDeliveredOrdersCountForAllCustomers()**: It should fetch all delivery orders from the repository, filter out the orders with a status of "**DELIVERED**" and retrieve the count of delivered("DELIVERED") orders for each customer. The result should return a map where the keys are customer names and the values are the counts of their delivered orders.

**Task 2: REST API Endpoints in DeliveryOrderController**

**PUT request to /api/delivery-orders/{orderId}/status**

Updates the status of a delivery order using the order ID and the specified status. The status must be either "DELIVERED" or "CANCELLED".

**Request Body**: String representing the new status ("DELIVERED" or "CANCELLED").

**Response Body**: String indicating the updated status of the delivery order

**HTTP Status Code**:

* 200 - If the status is updated successfully - "Delivery Status Updated Successfully".
* 400 - If the request body contains an invalid status - "Invalid status".
* 404 - If the given orderId is not found or there is not any details for the specific orderId.
* 500 - If there is an internal server error while updating the status- "An error occurred while updating the order status".

**GET request to /api/delivery-orders/average-delivery-duration**

Calculates and retrieves the average delivery duration in minutes up to 2 decimal places.

**Response Body**: The average delivery duration (in minutes) should be of a double data type.

**HTTP Status Code**:

* 200 - If you received the average delivery duration.
* 404 - If there are no delivered orders.
* 500 - Error in calculating average delivery duration.

**GET request to /api/delivery-orders/delivered-orders-count-by-customer**

Retrieves the count of delivered orders for each customer.

**Response Body**: JSON object representing customer names and their respective delivered order counts

**HTTP Status Code**:

* 200 - If you received the customer order counts.
* 404 - If there are no customers with delivered orders.
* 500 - Error in receiving the customer order counts.

Complete the given project so that it passes all the test cases when running the provided unit tests.

**Example Responses:**

**PUT request to /api/delivery-orders/1/status:**

Request Body (text):

**DELIVERED**

The response code is 200 and the response body should return a message as follows: plaintext

**Request**: PUT - /api/delivery-orders/1/status

**Delivery** Status Updated Successfully

**GET request to /api/delivery-orders/average-delivery-duration:**

The response code is 200 and the response body, when converted to JSON, is as follows:

36.67

**GET request to /api/delivery-orders/delivered-orders-count-by-customer:**

The response code is 200 and the response body, when converted to JSON, is as follows:

**Request**: GET - /api/delivery-orders/delivered-orders-count-by-customer

{

"Eve Black": 2,

"David Wilson": 1,

"Frank White": 2,

"Charlie Adams": 1,

"Grace Green": 1

}

**Allowed technologies**

*All available technologies for project based : backend problems enabled.*

------------------------------------------------------------------------------------------------------------------

JSP Quiz App

**Description**

**Problem Description**

You are tasked with creating an Online Quiz Application using JavaServer Pages (JSP) and Servlets. The application should allow users to select a topic, answer a series of questions related to that topic, and then receive a pass or fail result based on their answers. The application uses session management to track user progress and simulate quiz data using Java data structures.

**Constraints:**

* The application should be implemented using JavaServer Pages (JSP), Servlets, and Java classes.
* Quiz questions and answers should be simulated using Java data structures (e.g., ArrayList, HashMap).
* No database should be used in this implementation.
* The user interface should be simple and intuitive, focusing on functionality rather than aesthetics.

**JSP Pages:**

**index.jsp:**

Serves as the homepage where users can select a quiz topic.

**Features**:

* **Title**: **Select Quiz Topic**
* **Quiz Topic Selection**: A dropdown list with the name **"topic"** to select the quiz topic. Available options are **"Java"** and **"CSS"**.
* **Start Quiz Button**: A submit button labeled **"Start Quiz"** to start the quiz based on the selected topic.
* **Quiz Questions**: Displays a list of questions, shown only if the questions are available. Each question is followed by multiple radio button options to select the answer with the name **answer\_${index}**, where index is the index of the question and value **${option}**.
* **Submit Quiz Button**: A submit button labeled **"Submit Quiz"** to submit the selected answers for evaluation.
* **Quiz Result**: Displays the result of the quiz after submission, showing the user's score or feedback.

**Your** Result: Pass

**error.jsp :**

**Features**:

* **Title**: **Error Page**
* **Heading**: Displays a main heading **"Error"** to indicate the occurrence of an error.
* **Error Message**: Displays a paragraph with a generic error message stating, **"An unexpected error occurred. Please try again later."**
* **Specific Error Details**: Conditionally displays a specific error message if available. This message is shown in a paragraph with the text **"Error details: ${error}"** using JSTL to check if the error attribute is not empty.
* **Navigation**: Includes a link labeled **"Back to Home"** to navigate back to the home page (index.jsp).

An unexpected error occurred. Please try again later.

Error details:

Back to Home

**Java Classes:**

**Quiz:**

* Represents a quiz for a specific topic.
* **Attributes**: A collection of questions and the corresponding correct answers.
* **Methods**: Get the list of questions, validate answers, and calculate the score.

**Question:**

* Represents an individual quiz question.
* **Attributes**: Question text, options (multiple-choice answers), and the correct answer.
* **Methods**: Get and set the question attributes.

**QuizData:**

* To build Quiz which will have list of questions.
* **Attributes**: Map<String, Quiz> quizzes, List javaQuestions.
* **Methods**: Get quiz by topic.

**QuizServlet:**

**Purpose**: Serves as the controller handling user requests, managing the quiz flow, and forwarding results to the JSP pages.

**Functional Requirements**:

* **Quiz Selection**:

1. The servlet should handle **GET** requests to select a quiz based on a topic provided as a request parameter (**topic**).
2. It should retrieve the corresponding quiz from a data source (represented by **QuizData.getQuizByTopic(topic)**).
3. If a quiz is found for the given topic, it should store the quiz and its questions in the session and request attributes, then forward the user to the main quiz page (**index.jsp**).
4. If no quiz is found for the topic, an error message should be displayed, indicating that no quiz was found for the provided topic.

* **Quiz Submission and Evaluation**:

1. The servlet should handle **POST** requests to process the quiz answers submitted by the user.
2. It should retrieve the current quiz from the session. If no quiz is found, it redirects to an error page (**error.jsp**).
3. It should iterate over the quiz questions, check the user's answers against the correct answers, and count the number of correct responses.
4. Based on the number of correct answers should determine if the user has passed or failed the quiz by comparing the correct count against the quiz's passing score.
5. The result (**"Pass" or "Fail"**) should be set as a request attribute along with the quiz questions, and the user should be forwarded back to the main quiz page (**index.jsp**) to display the results.

* **Error Handling**:

1. If a **GET** request is made with an invalid or missing topic, an error message **"Quiz not found for topic: " + topic** should be displayed on the quiz page.
2. If a **POST** request is made with incomplete answers (e.g., missing answers), the user should be redirected to an error page.

* **Session Management**:

1. The servlet should use the **HTTP session** to maintain the state of the quiz and the user's progress.

* **Navigation Flow**:

1. The servlet forwards users to **index.jsp** for both displaying the quiz and showing the results.
2. For any errors encountered, users are redirected to **error.js**p.

**index.jsp**

A screenshot of a questionnaire

Description automatically generated

**error.jsp**

A black text on a white background

Description automatically generated

**Start Server:**

This option from the **Run** menu will start the server. You can view the **Live Preview** once the server is started in multiple ways:

* Click the **Preview in Editor** option that pops up at the bottom-right corner.
* Click the **Open Preview** option in the Run menu.

**Run Test:**

* To test your code, hit the **Test** button from the **Run** menu.
* There are tests to ensure the correctness of each method. You can use these tests while debugging or checking your implementation.

**Note:**

* **Session Management**: The servlet uses session management to maintain the user's quiz progress across multiple requests.
* **Error Handling**: The application includes basic error handling for cases such as invalid input and ensuring all questions are answered before submission.
* **Environment Variable Retrieval**: The servlet accesses an environment variable named VSCODE\_PROXY\_URI using the System.getenv() method. This variable contains the URI of a proxy server, which is crucial for redirecting users to appropriate JSP pages.

------------------------------------------------------------------------------------------------------------------

Jsp Library Management

**Description**

**Problem Description**

You are tasked with creating a Library Management Application using JavaServer Pages (JSP) and Servlets. The application allows users to search for books, view details about specific books, and manage their reading list by adding or removing books. The application simulates data using Java data structures and manages user sessions to store the reading list.

**Constraints**

* The application should be implemented using JavaServer Pages (JSP), Servlets, and Java classes.
* Book data should be simulated using Java data structures (e.g., ArrayList, HashMap).
* No database should be used in this implementation.
* The user interface should be simple and focused on functionality.

**JSP Pages**

**index.jsp**

**Purpose**: Serves as the homepage where users can search for books and view the search results.

**Features**:

**Title:** Online Library

**Display Available Books**:

* **Heading**: A heading with text **"Available Books"** is displayed at the top of the page.
* **List of Books**: List of books is displayed, showing the title and author of each book.
* **Details Link**: Each book entry includes a **"View Details"** link that directs to the **LibraryServlet** with the action parameter set to **viewDetails** and the index of the book in the list.

**Book Search Functionality**:

* **Search Form:** A form that allows users to search for books by title or author.
* **Text Input Field**: A required input field with the name as **keyword** where users can enter keywords for the search.
* **Hidden Input Field**: An input field with the name as **action** and value as **search** to specify the action for the servlet.
* **Submit Button**: A button labeled **"Search"** to submit the search query.

**Display Search Results**:

* **Heading**: A heading labeled **"Search Results"** is displayed if there are search results.
* **List of Search Results**: A dynamically generated list displaying the search results.
* **Book Details**: For each search result, the book's title and author are shown.
* **View Details Link**: Each book in the search results includes a **"View Details"** link that directs to the **LibraryServlet** with the action parameter set to **viewDetails** and the index of the book in the list.

**Add to Reading List Link**:

* Each book in the search results includes an **"Add to Reading List"** link that directs to the **LibraryServlet** with the action parameter set to **addToReadingList** and the index of the book in the list.

**Navigation**:

* **View Reading List Link**: A link labeled **"View Reading List"** that navigates to the readingList.jsp page to view the user's reading list.

**Logic and Data Handling**:

* **Data Retrieval**: A list of all books is retrieved using the method BookData.getallBook() and stored in the request scope for JSTL processing.
* **Conditionally Rendered Content**: Content blocks for available books and search results are conditionally rendered based on whether the lists (bookList and searchResults) are empty or not.

A screenshot of a computer

Description automatically generated

**bookDetails.jsp**

**Purpose**: Displays detailed information about a selected book.

**Features**:

* **Title**: **Book Details**
* **Book Details**: Shows the title, author, year, and description of the selected book as shown below.

A screenshot of a computer

Description automatically generated

* **Add to Reading List Button**: A button to add the book to the user's reading list labeled **"Add to Reading List"**.
* **Back to Search Results Link**: A link to navigate back to the home page (index.jsp) labeled **"Back to Home"**.

**readingList.jsp**

**Purpose**: Displays the user's current reading list.

**Features**:

* **Title**: **Your Reading List**
* **Reading List**: Shows the list of books added by the user. Each book in the list includes the title, author, and a **"Remove"** link to remove the book from the reading list.
* **Back to Home Link**: A link to navigate back to the homepage, labeled **"Home"**.

A close-up of a list

Description automatically generated

**error.jsp**

**Purpose**: Displays error messages when specific errors occur, such as trying to add a book that is already in the reading list.

**Features**:

* **Title**: **Error Page**
* **Error Message**: Displays a paragraph with a generic error message stating **"An unexpected error occurred. Please try again later."**
* **Specific Error Details**: Conditionally displays a specific error message if available. This message is shown in a paragraph with the text of the error using JSTL to check if the error attribute is not empty.
* **Navigation**: Includes a link labeled **"Back to Home"** to navigate back to the home page (index.jsp).

A close-up of a white background

Description automatically generated

**Java Classes**

**Book**

**Purpose**: Represents an individual book in the library.

**Attributes**:

* title: The title of the book.
* author: The author of the book.
* description: A brief description of the book.

**Methods**:

* Getters and setters for each attribute.
* equals() and hashCode() methods to compare books and ensure unique entries in the reading list.

**BookData**

**Purpose**: Simulates a data source containing a list of books.

**Attributes**:

* bookList: A list of Book objects representing the library's collection.

**Methods**:

* getAllBooks(): Returns the list of all books in the library.

**LibraryServlet**

**Purpose**: Serves as the controller for handling user requests related to searching for books, viewing book details, managing a reading list, and navigating to the appropriate JSP pages.

**Functional Requirements**:

**Book Search**:

* The servlet should handle **GET** requests with an action parameter set to **"search"**.
* It retrieves a keyword from the request parameters and decodes it.
* It searches for books in the bookList whose title or author contains the provided keyword.
* If no keyword is provided or if the search results are empty, it forwards the user back to the main page (index.jsp).
* The search results and the full book list are set as request attributes and the user is forwarded to the main page (index.jsp).

**View Book Details**:

* The servlet should handle **GET** requests with an action parameter set to **"viewDetails"**.
* It retrieves the index of the selected book from the request parameters.
* It fetches the corresponding book from the **bookList** using the provided index.
* The selected book and its index are set as request attributes, and the user is forwarded to a detailed view page (bookDetails.jsp).

**Add to Reading List**:

* The servlet should handle **GET** requests with an action parameter set to **"addToReadingList"**.
* It retrieves the index of the selected book and fetches it from the bookList.
* The selected book is added to the user's reading list, which is stored in the HTTP session.
* If the book is already in the reading list, an error message **"Already Added in Reading List !"** is set as a request attribute and the user is forwarded to an error page (error.jsp).
* Otherwise, the updated reading list is set as a request attribute, and the user is forwarded to the reading list page (readingList.jsp).

**Remove from Reading List**:

* The servlet should handle **GET** requests with an action parameter set to **"removeFromReadingList"**.
* It retrieves the index of the book to be removed from the request parameters.
* The corresponding book is removed from the user's reading list stored in the HTTP session.
* The updated reading list is set as a request attribute, and the user is forwarded to the reading list page (readingList.jsp).

**Default Action**:

* If no specific action is provided in the request, the servlet forwards the user to the main page (index.jsp ) displaying the full list of books.

**Error Handling**:

* If no keyword is provided in the search or if the search results are empty, the user is forwarded back to the main page (index.jsp).
* If a user tries to add a book to the reading list that is already present, an error message is displayed, and the user is forwarded to an error page (error.jsp).

**Session Management**:

* The servlet uses the HTTP session to maintain the state of the user's reading list, ensuring that books added to the reading list persist across requests.

**Navigation Flow**:

* Users are primarily forwarded to index.jsp for viewing the full book list or search results.
* For viewing detailed information about a book, users are forwarded to bookDetails.jsp.
* The readingList.jsp page is used to display the user's reading list.
* For errors (like adding a duplicate book), users are forwarded to error.jsp.

**Start Server**

This option from the **Run** menu will start the server. You can view the **Live Preview** once the server is started in multiple ways:

* Click the **Preview in Editor** option that pops up at the bottom-right corner.
* Click the **Open Preview** option in the Run menu.

**Run Test**

* To test your code, hit the **Test** button from the **Run** menu.
* There are tests to ensure the correctness of each method. You can use these tests while debugging or checking your implementation.

**Note**

* **Session Management**: The servlet uses session management to maintain the user's reading list across multiple requests.
* **Error Handling**: The application includes error handling for cases such as invalid input and duplicate entries in the reading list.
* **Environment Variable Retrieval**: The servlet accesses an environment variable named VSCODE\_PROXY\_URI using the System.getenv() method. This variable contains the URI of a proxy server, which is crucial for redirecting users to appropriate JSP pages.