Object Oriented Analysis and Design with Java

UE20CS352

QUIZZINATOR

NAME	*SRN*
ARYAN KAPOOR	PES2UG20CS069
AVIJIT PANDA	PES2UG20CS078
BHARATH D H	PES2UG20CS086
DOREDLA TIRUMALA RAVI TEJA	PES2UG20CS115

Abstract

We aim to build a quizzing platform for a wide assortment of quizzes which cater to the needs of both quizmaster and participant alike.

The project: “Quizzinator” is a collection of a number of different types of quizzes covering various topics. A user can access/play all of the quiz and can attempt any of the one.

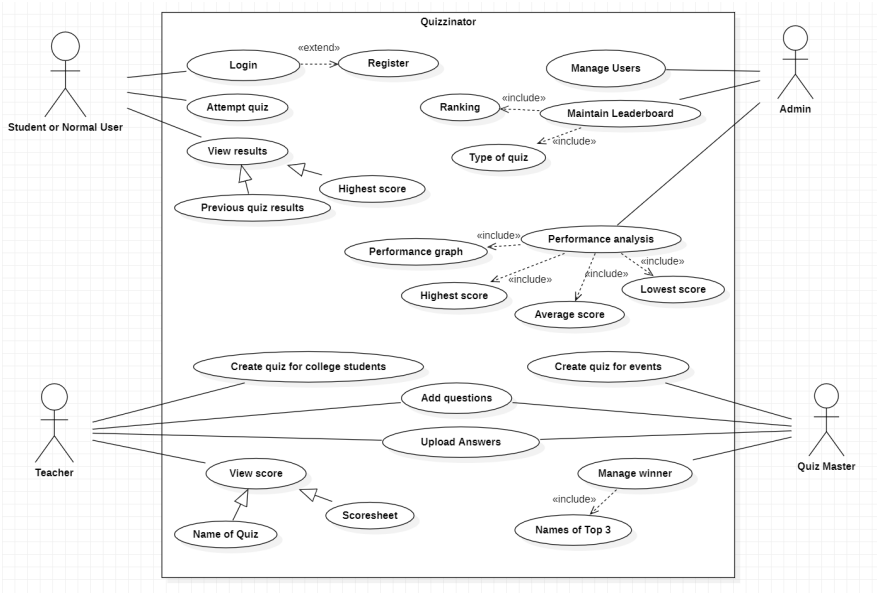
Quizzinator is a user-friendly quiz application which contains : Various quizzes , answers to every question, different formats of quizzes(Rapid fire, hints,Classic,Lifeline) Uploading of question and answer by admin/quizmaster, and to improve the knowledge level of users. Using this application the user and the quizmaster can analyze the scores as well as check the rankings and the highest,average scores.

QUIZZINATOR : A ONE STOP solution for Quizzing.

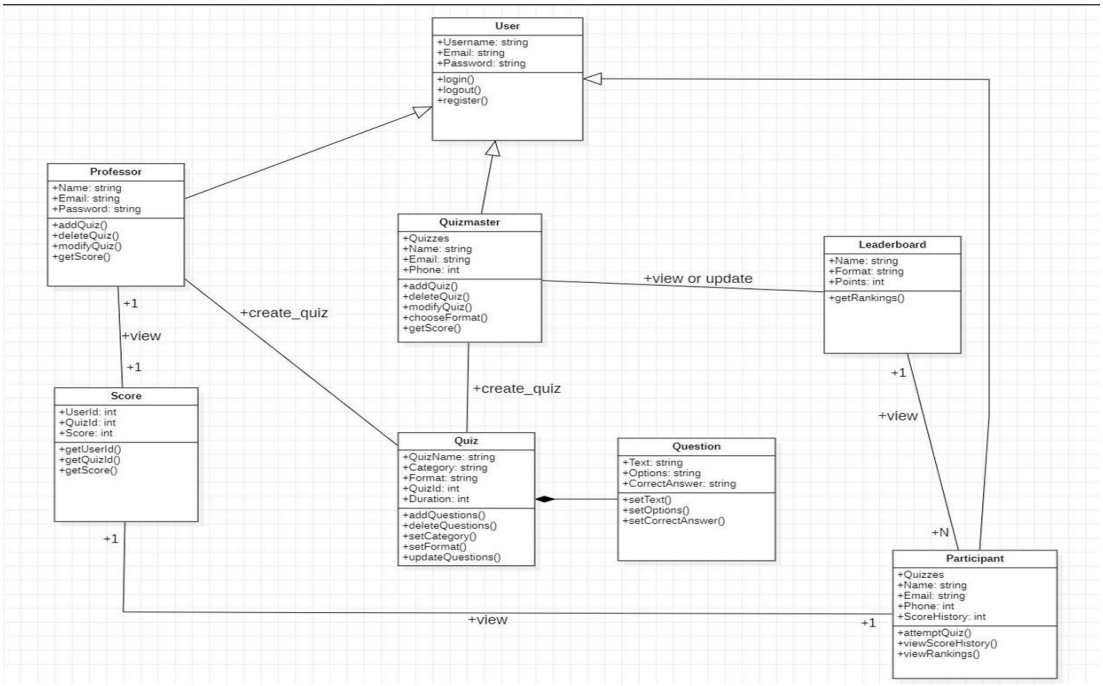
Tools Used:

1. **Spring Boot Web Application**
2. **MySQL Database**

USE CASE DIAGRAM:



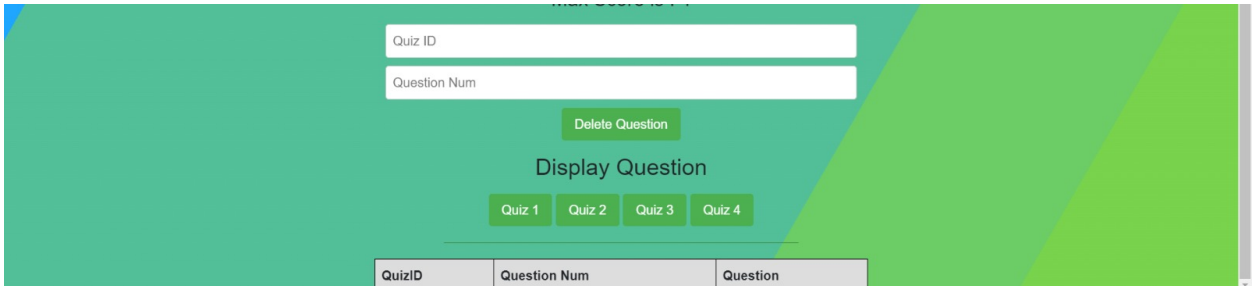
CLASS DIAGRAM:



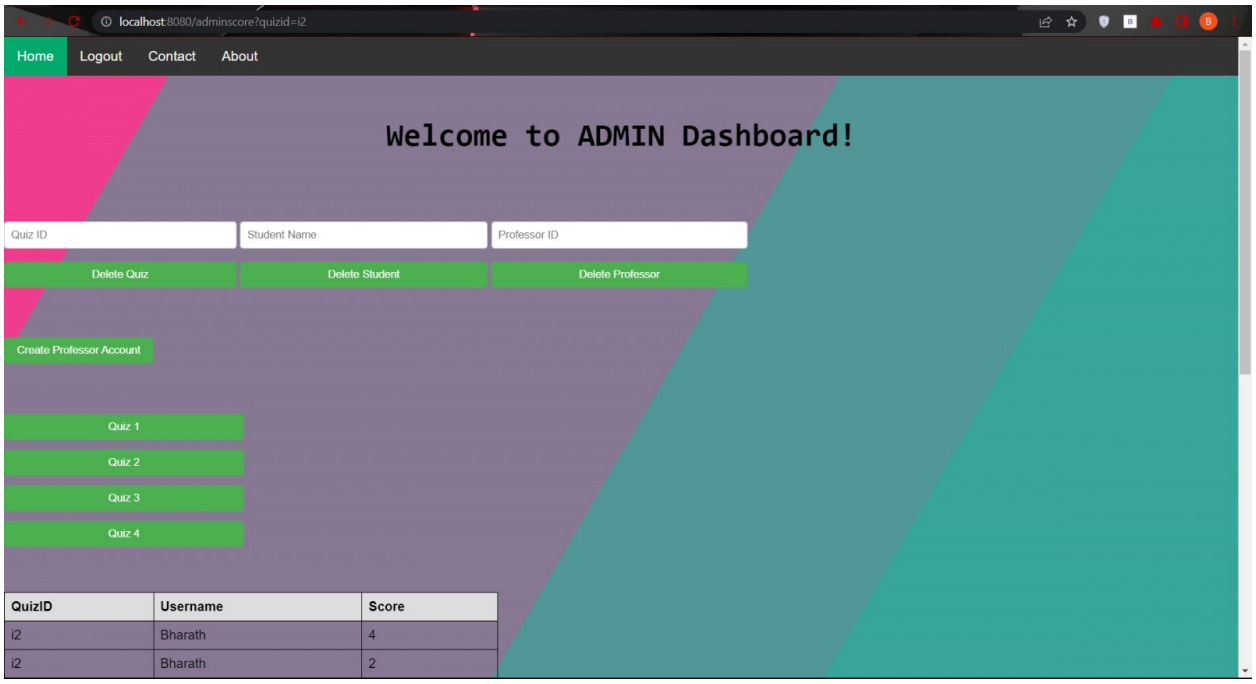
Architecture:

Utilised GRASP Principles:

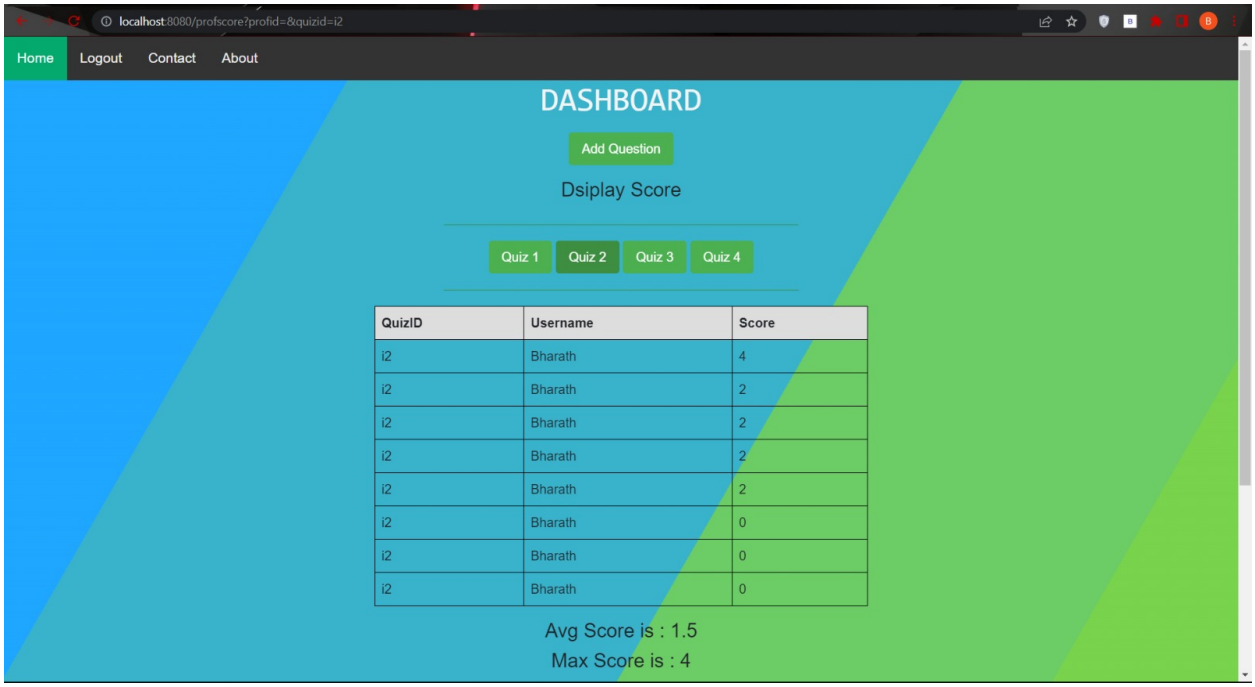
1. Information Expert: Assign responsibility to the object that has the most information required to fulfill it.



Controller: Assign responsibility to the object that manages and coordinates the flow of information and requests between objects.



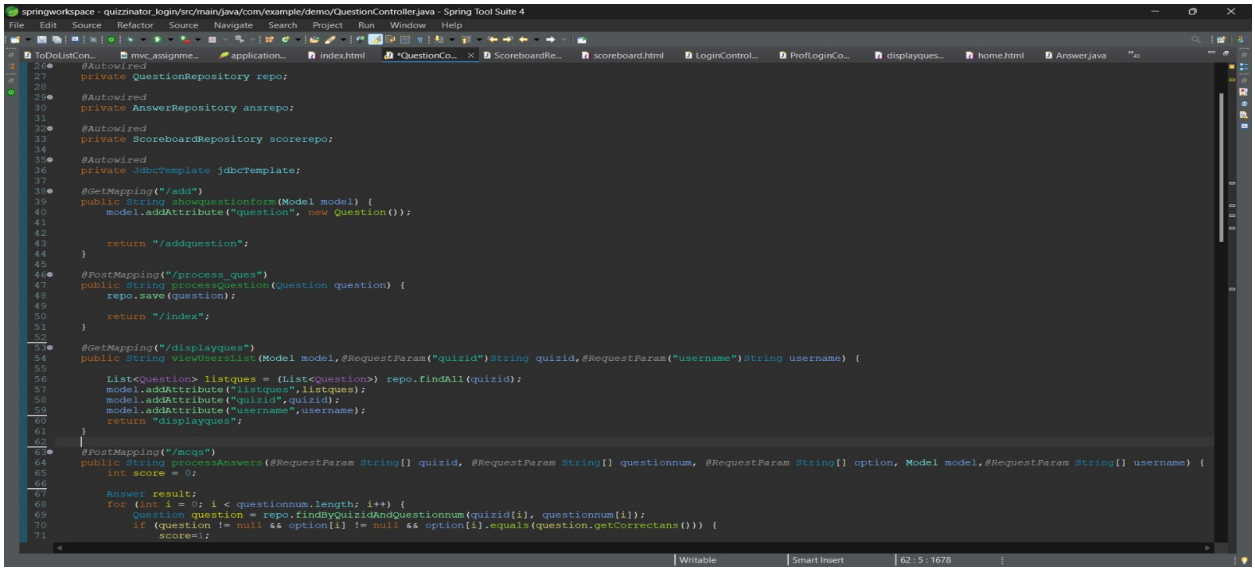
2. High Cohesion: Assign responsibility to the object that has the most related functions to perform.

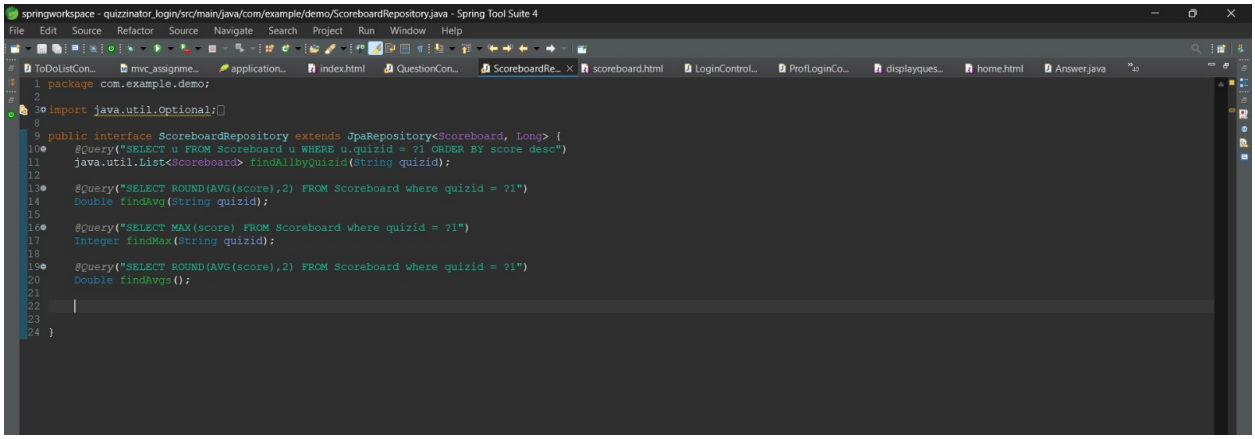


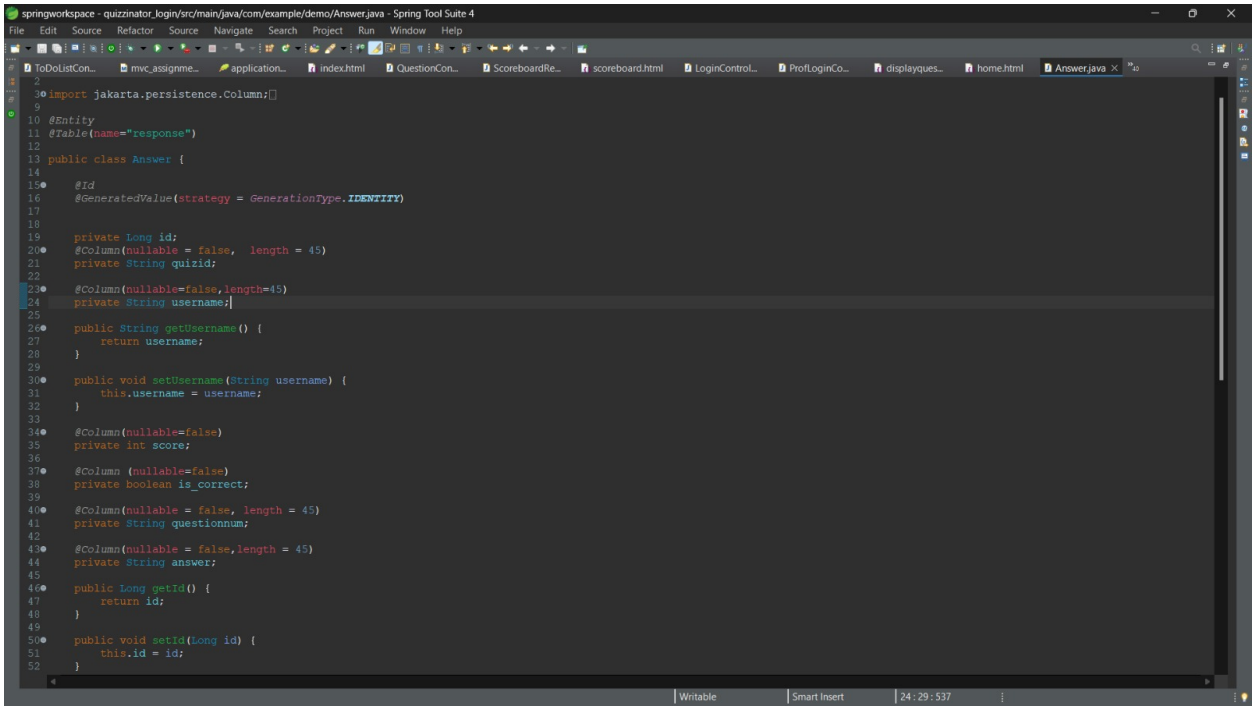
3. Low Coupling: Assign responsibility to the object that has the least amount of dependency on other objects.

The MVC architecture can be implemented in the following way:

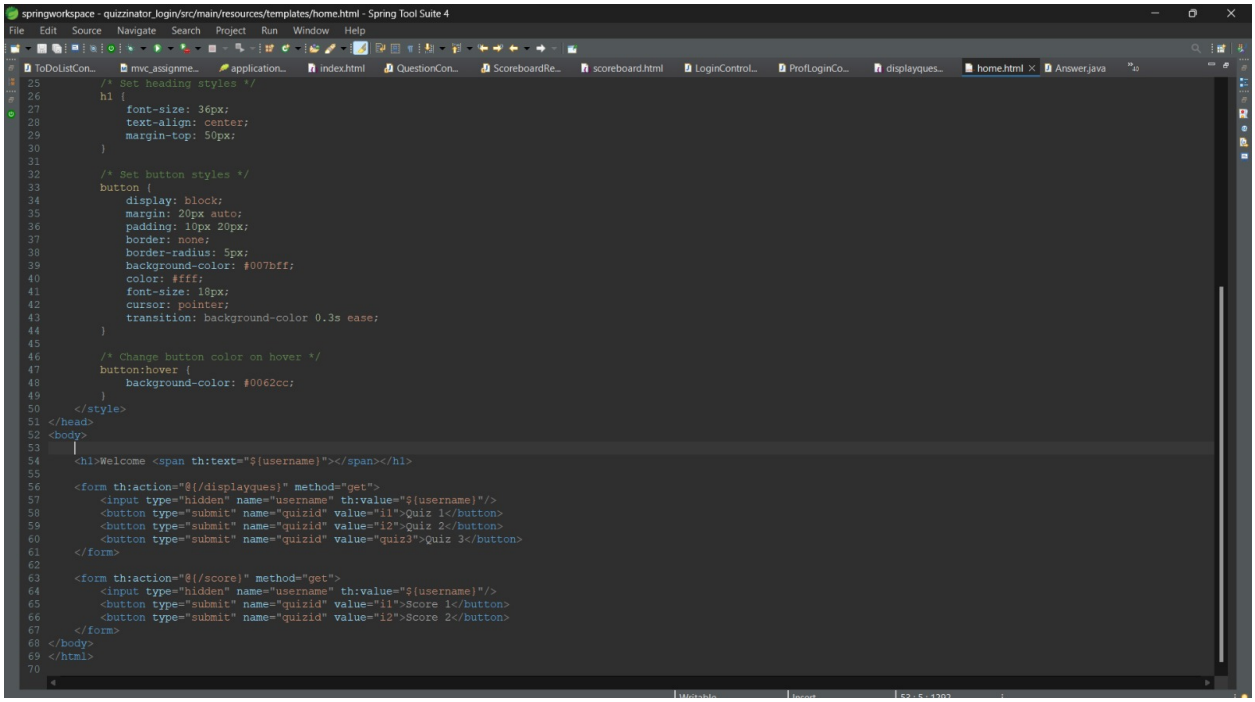
Model: The model represents the data and business logic of the quiz app. In this case, the model would include the quiz questions, answer options, correct answers, and other related data. The model would also contain the logic for generating random questions, checking answers, and calculating the score.

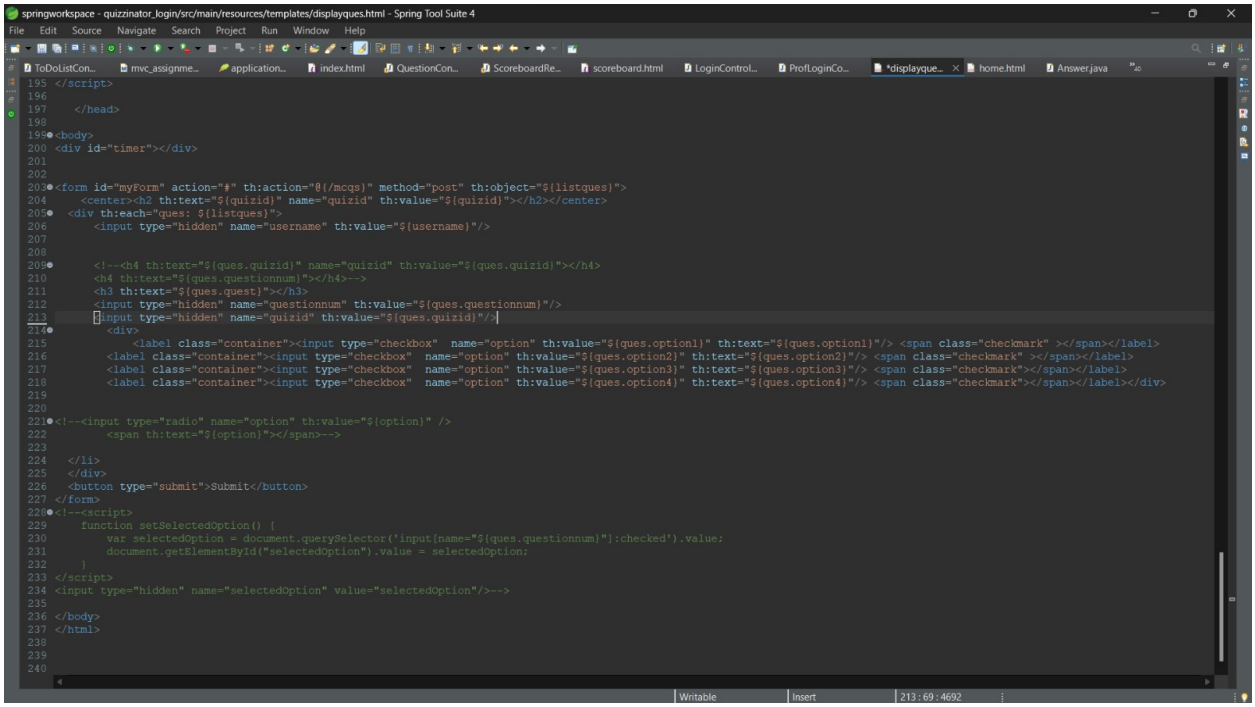






View: The view represents the user interface of the quiz app. In this case, the view would include the screens and UI elements used to display the quiz questions and answer options to the user. The view would also be responsible for handling user input, such as selecting an answer option and submitting the answer.


springworkspace - quizzinator_login/src/main/resources/templates/home.html - Spring Tool Suite 4
File Edit Source Navigate Search Project Run Window Help
ToDoListCon... mvc_assignme... application... index.html QuestionCon... ScoreboardRe... scoreboard.html LoginControl... ProfLoginCo... displayques... home.html Answer.java
/* Set heading styles */
h1 {
font-size: 36px;
text-align: center;
margin-top: 50px;
}
/* Set button styles */
button {
display: block;
margin: 20px auto;
padding: 10px 20px;
border: none;
border-radius: 5px;
background-color: #007bff;
color: #fff;
font-size: 18px;
cursor: pointer;
transition: background-color 0.3s ease;
}
/* Change button color on hover */
button:hover {
background-color: #0062cc;
}
</style>
</head>
<body>
<h1>Welcome </h1>
<form th:action="@{/displayques}" method="get">
<input type="hidden" name="username" th:value="${username}"/>
<button type="submit" name="quizid" value="i1">Quiz 1</button>
<button type="submit" name="quizid" value="i2">Quiz 2</button>
<button type="submit" name="quizid" value="quiz3">Quiz 3</button>
</form>
<form th:action="@{/score}" method="get">
<input type="hidden" name="username" th:value="${username}"/>
<button type="submit" name="quizid" value="i1">Score 1</button>
<button type="submit" name="quizid" value="i2">Score 2</button>
</form>
</body>
</html>


springworkspace - quizzinator_login/src/main/resources/templates/displayques.html - Spring Tool Suite 4
File Edit Source Navigate Search Project Run Window Help
ToDoListCon... mvc_assignme... application... index.html QuestionCon... ScoreboardRe... scoreboard.html LoginControl... ProfLoginCo... *displayque... home.html Answer.java
</script>
</head>
<body>

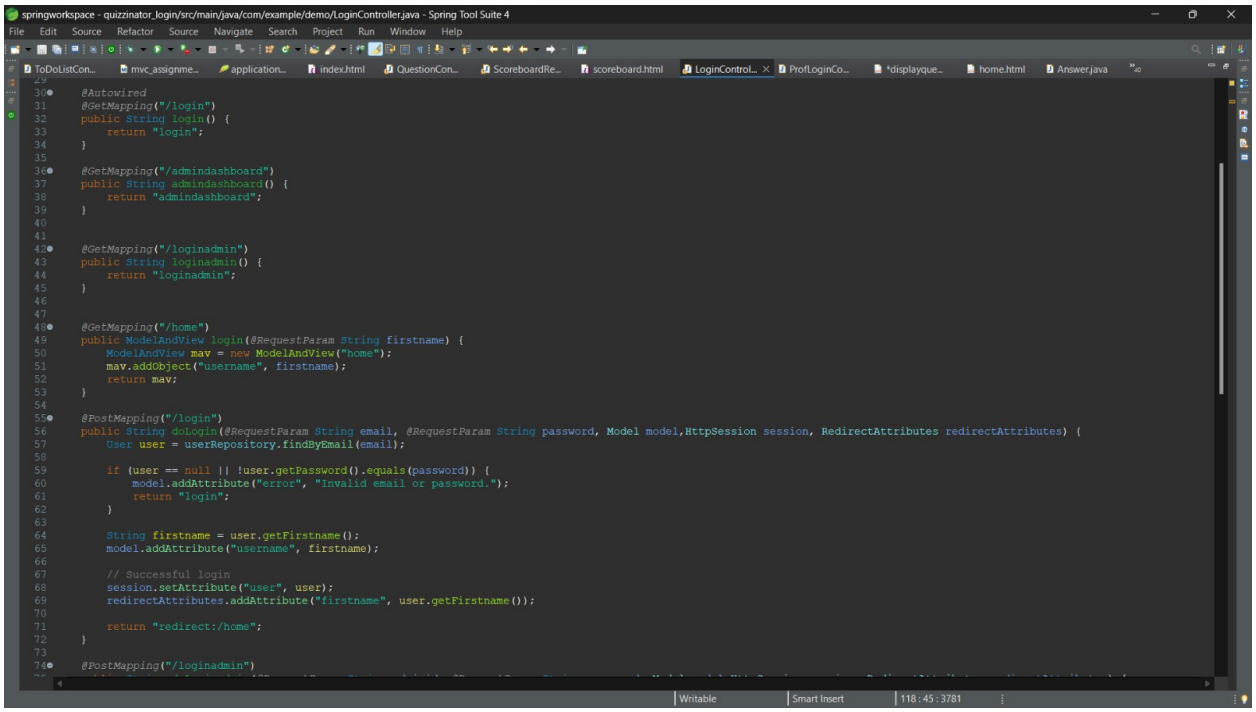
<form id="myForm" action="#" th:action="@{/mcqs}" method="post" th:object="${listques}">
<h2 th:text="${quizid}" name="quizid" th:value="${quizid}"></h2>

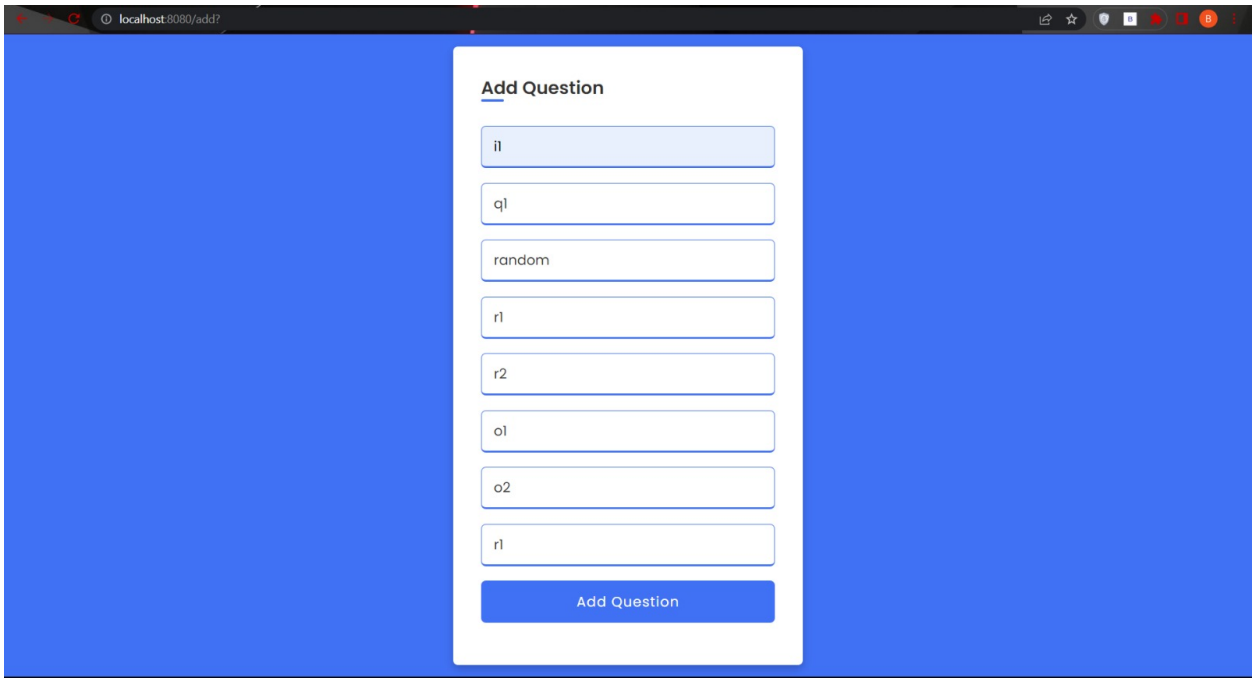
<input type="hidden" name="username" th:value="${username}"/>
<!--<h4 th:text="${ques.quizid}" name="quizid" th:value="${ques.quizid}"></h4>
<h4 th:text="${ques.questionnum}"></h4>-->
<h3 th:text="${ques.quest}"></h3>
<input type="hidden" name="questionnum" th:value="${ques.questionnum}"/>
<input type="hidden" name="quizid" th:value="${ques.quizid}"/>

<label class="container"><input type="checkbox" name="option" th:value="${ques.option1}" th:text="${ques.option1}"/> </label>
<label class="container"><input type="checkbox" name="option" th:value="${ques.option2}" th:text="${ques.option2}"/> </label>
<label class="container"><input type="checkbox" name="option" th:value="${ques.option3}" th:text="${ques.option3}"/> </label>
<label class="container"><input type="checkbox" name="option" th:value="${ques.option4}" th:text="${ques.option4}"/> </label>
<!--<input type="radio" name="option" th:value="${option}" />
-->

<button type="submit">Submit</button>
</form>
<!--<script>
function setSelectedOption() {
var selectedOption = document.querySelector('input[name="${ques.questionnum}"]:checked').value;
document.getElementById("selectedOption").value = selectedOption;
}
</script>
<input type="hidden" name="selectedOption" value="selectedOption"/>-->
</body>
</html>
Writable Insert 213 : 69 : 4692

Controller: The controller acts as an intermediary between the model and view, handling user input and updating the model and view as needed. In this case, the controller would receive user input from the view, validate the input, and update the model accordingly. It would also update the view with the current question and answer options, as well as the user's score.




localhost:8080/add?
Add Question
i1
q1
random
r1
r2
o1
o2
r1
Add Question

DESIGN PATTERNS USED:

1. Factory Method Pattern: The Question class is an abstract class that defines the **getQuestion()** method. This method is used as a factory method to create different types of questions, such as MCQQuestion and TrueFalseQuestion. The Quiz class uses this method to create the questions for the quiz.

2. Strategy Pattern: The Answer class is an abstract class that defines the **evaluate()** method. This method is implemented by the concrete MCQAnswer and TrueFalseAnswer classes, which represent different strategies for evaluating answers to multiple-choice and true/false questions, respectively.

3. Singleton Pattern: The Quiz class uses a private constructor and a static instance variable to implement the Singleton pattern, ensuring that only one instance of the Quiz class can exist at a time.

4. Template Method Pattern: The Quiz class defines the **run()** method, which serves as a template method that defines the overall structure of the quiz. The MCQQuestion and TrueFalseQuestion classes override the **display()** method, which is a hook method that allows the subclasses to customize the behavior of the template method.

These are the main design patterns we have used in our project QUIZZINATOR

INDIVIDUAL CONTRIBUTIONS:

Aryan Kapoor - Handled the Student/User Class and Frontend

Avijit Panda - Handled the Professor Class and Frontend

Bharath D H - Handled the Admin and Scoreboard Class

Ravi Teja - Handled the Question Class

Github link : https://github.com/Aryan-11-Kapoor/QuizApp_OOADJ