Problem 2 – Online Quiz System

Note: In order to run the quiz correctly, please execute the following commands:

1. npm install -g http-server
2. navigate to the folder in which the index.html file is stored
3. http-server -c-1

That way a http server will be started and the quiz is accessible at: http://localhost:8080
This has to be done because the questions are read from a .json file instead of being hardcoded.

If that does not work, you can also **access the quiz** through the following link:
https://mogadenis.github.io/KnowItAll-Challenge/

1. Logical Design

The questions in the quiz are read from a **.json file** containing objects with the following fields: question itself, list of answers and the correct answer, which would be interpreted as **Question** objects.

These questions are then added to a **QuestionPool** which will contain all 50 questions. This class is then responsible for randomly picking a given number of questions from those 50. Finally, we have the **Quiz** class which receives from the QuestionPool the random questions and is then, responsible for giving the questions to the user, one by one, and for keeping track of the **score**.

Regarding the random choice of questions, the QuestionPool class which holds all questions generates an array of indices from 0 to 49, which is then **shuffled** and the Quiz receives a slice (the desired number of questions to be included in the quiz) of an array containing the questions at the random indices in the pool.

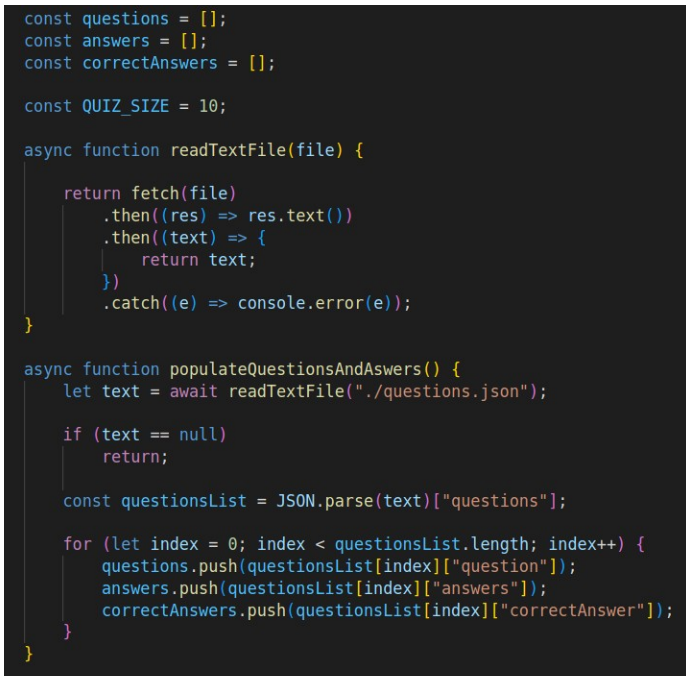
Pseudocode:

```
getRandomIndices(numberOfQuestions) → list<Question> {
    indices = [0...length(allQuestions) - 1]
    shuffle(indices)

    randomQuestions = []

    for i = 0...numberOfQuestions
        randomQuestions.add(allQuestions[indices[i]])

    return randomQuestions
}
```

In the case in which the total number of questions might grow, this algorithm runs in **linear time** (O(n) time complexity). This algorithm also ensures **equal probabilities** for all questions and **no repetitions**.

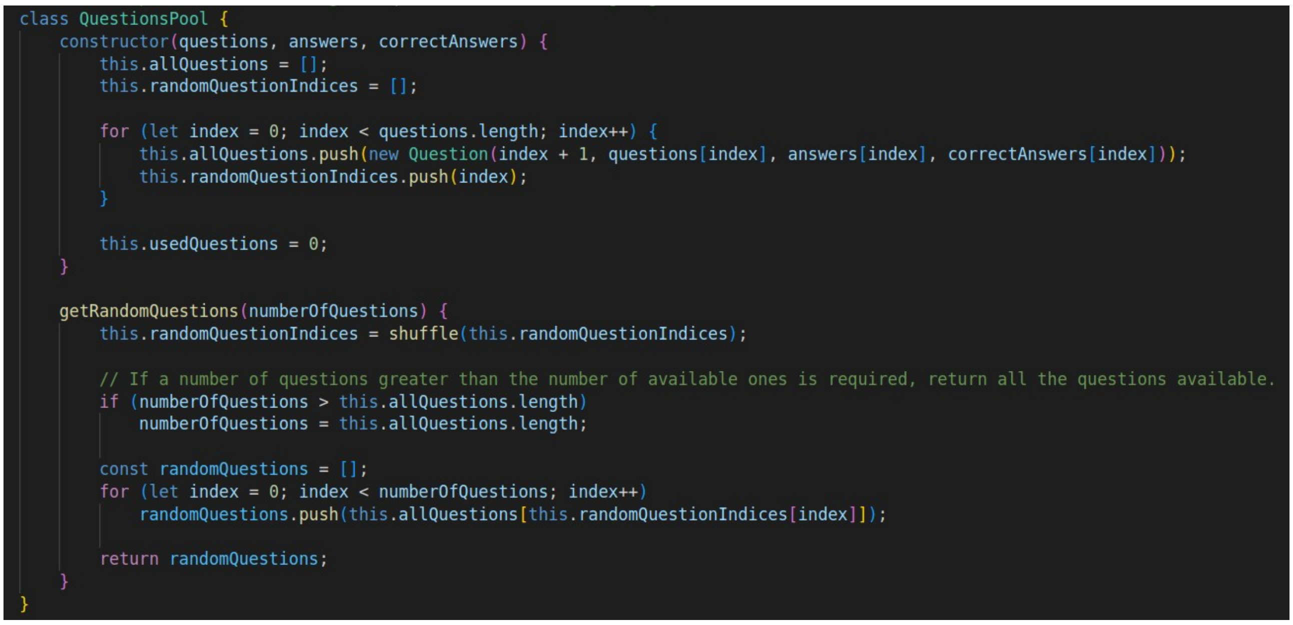
The score is stored in the Quiz class; an answer, after it is submitted by the user, is evaluated and if the answer is right, the score is incremented.

2. Algorithm implementation



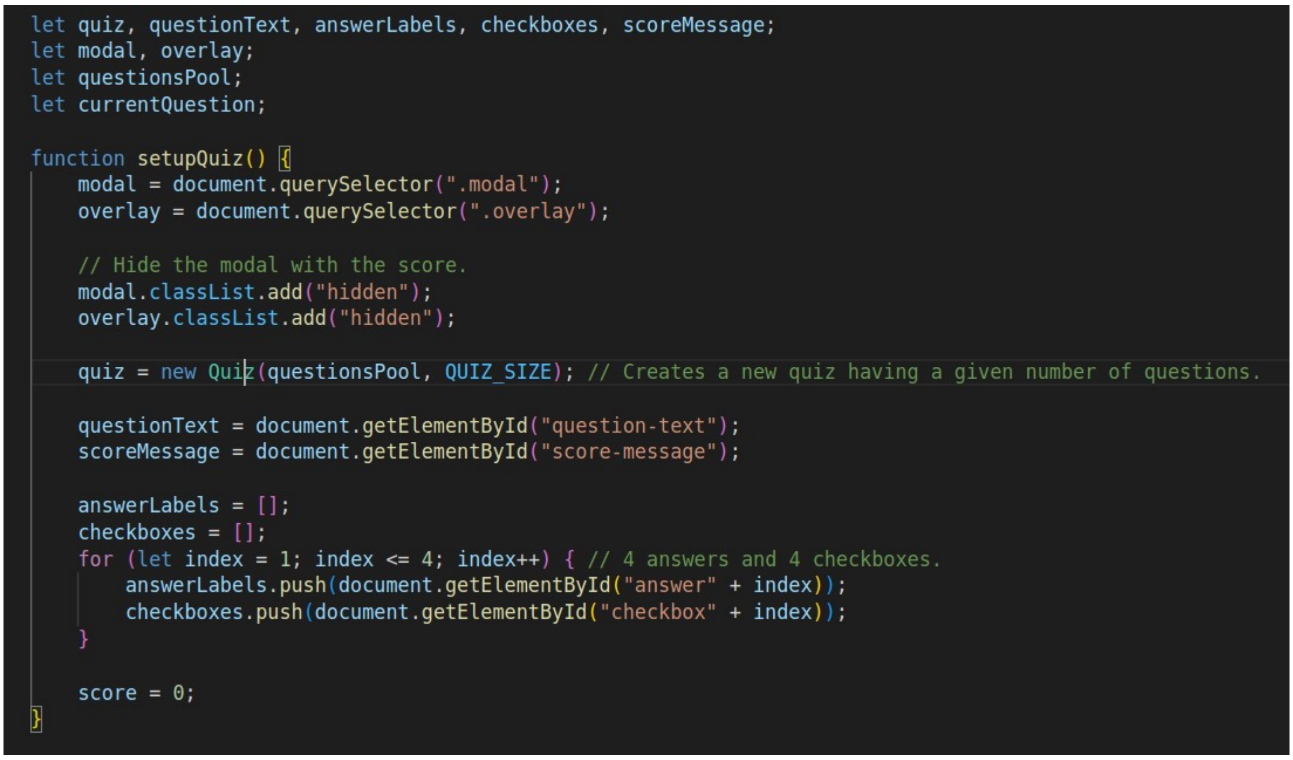
```javascript
const questions = [];
const answers = [];
const correctAnswers = [];

const QUIZ_SIZE = 10;

async function readTextFile(file) {

    return fetch(file)
        .then((res) => res.text())
        .then((text) => {
            return text;
        })
        .catch((e) => console.error(e));
}

async function populateQuestionsAndAswers() {
    let text = await readTextFile("./questions.json");

    if (text == null)
        return;

    const questionsList = JSON.parse(text)["questions"];

    for (let index = 0; index < questionsList.length; index++) {
        questions.push(questionsList[index]["question"]);
        answers.push(questionsList[index]["answers"]);
        correctAnswers.push(questionsList[index]["correctAnswer"]);
    }
}
```

These are the methods responsible for reading the contents of the .json file and **parsing** them; the questions, answers and correctAnswers are stored in lists.



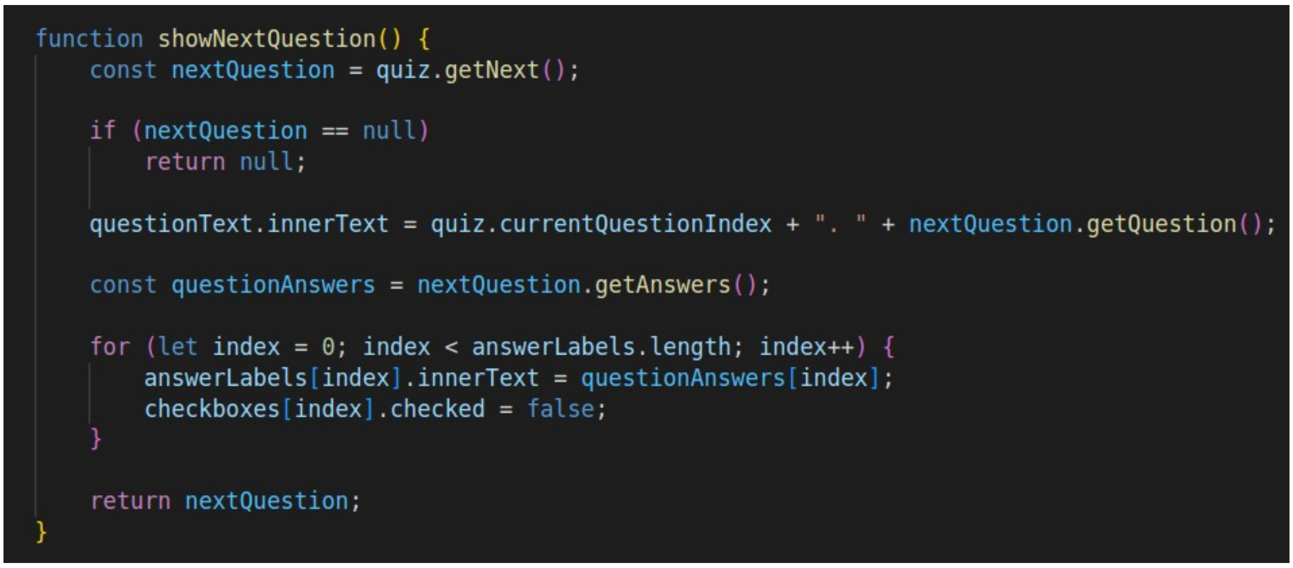
```javascript
class QuestionsPool {
    constructor(questions, answers, correctAnswers) {
        this.allQuestions = [];
        this.randomQuestionIndices = [];

        for (let index = 0; index < questions.length; index++) {
            this.allQuestions.push(new Question(index + 1, questions[index], answers[index], correctAnswers[index]));
            this.randomQuestionIndices.push(index);
        }

        this.usedQuestions = 0;
    }

    getRandomQuestions(numberOfQuestions) {
        this.randomQuestionIndices = shuffle(this.randomQuestionIndices);

        // If a number of questions greater than the number of available ones is required, return all the questions available.
        if (numberOfQuestions > this.allQuestions.length)
            numberOfQuestions = this.allQuestions.length;

        const randomQuestions = [];
        for (let index = 0; index < numberOfQuestions; index++)
            randomQuestions.push(this.allQuestions[this.randomQuestionIndices[index]]);

        return randomQuestions;
    }
}
```

A **QuestionsPool** object is then created, which upon its creation takes the previous three lists and creates and stores Question objects. This class has the method which returns a number of **random questions** from the list using the approach explained earlier.



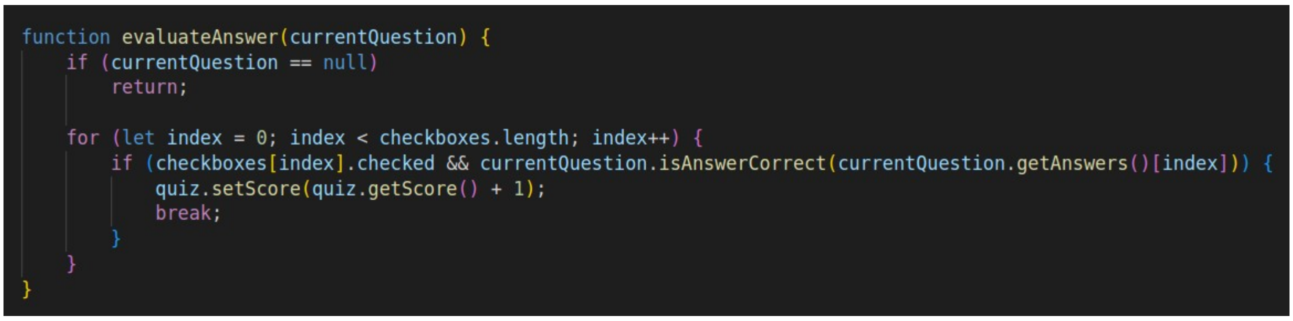
```
let quiz, questionText, answerLabels, checkboxes, scoreMessage;
let modal, overlay;
let questionsPool;
let currentQuestion;

function setupQuiz() {
    modal = document.querySelector(".modal");
    overlay = document.querySelector(".overlay");

    // Hide the modal with the score.
    modal.classList.add("hidden");
    overlay.classList.add("hidden");

    quiz = new Quiz(questionsPool, QUIZ_SIZE); // Creates a new quiz having a given number of questions.

    questionText = document.getElementById("question-text");
    scoreMessage = document.getElementById("score-message");

    answerLabels = [];
    checkboxes = [];
    for (let index = 1; index <= 4; index++) { // 4 answers and 4 checkboxes.
        answerLabels.push(document.getElementById("answer" + index));
        checkboxes.push(document.getElementById("checkbox" + index));
    }

    score = 0;
}
```

Before starting a quiz, the function **setupQuiz()** is called in order to obtain from the HTML document references to the answer labels, checkboxes, question's text and the score message, and also to a modal and overlay which are used for **displaying the score** at the end.



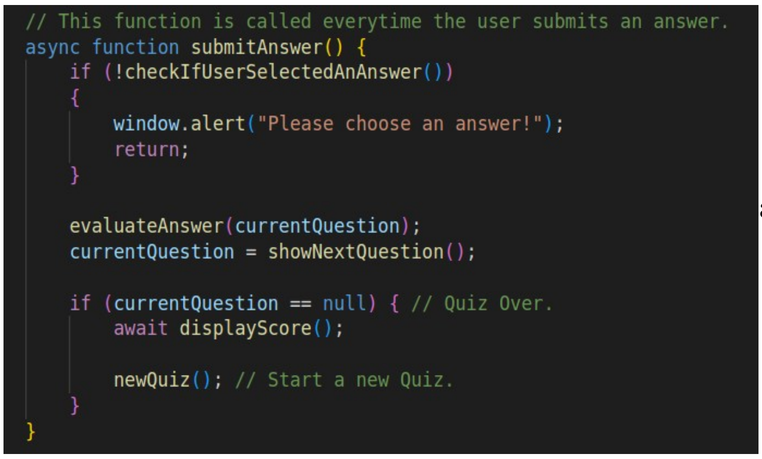
```
function showNextQuestion() {
    const nextQuestion = quiz.getNext();

    if (nextQuestion == null)
        return null;

    questionText.innerText = quiz.currentQuestionIndex + ". " + nextQuestion.getQuestion();

    const questionAnswers = nextQuestion.getAnswers();

    for (let index = 0; index < answerLabels.length; index++) {
        answerLabels[index].innerText = questionAnswers[index];
        checkboxes[index].checked = false;
    }

    return nextQuestion;
}
```

The function above changes the current question, updates the four answers and clears the checkboxes. The quiz is done when this function returns a null value, that is why the question displayed is also returned.



```
function evaluateAnswer(currentQuestion) {
    if (currentQuestion == null)
        return;

    for (let index = 0; index < checkboxes.length; index++) {
        if (checkboxes[index].checked && currentQuestion.isAnswerCorrect(currentQuestion.getAnswers()[index])) {
            quiz.setScore(quiz.getScore() + 1);
            break;
        }
    }
}
```

Each time an answer is **submitted**, it is evaluated inside this function which basically parses the four checkboxes and when it finds a checked one, it verifies if it corresponds to the right answer, if so, the **score is incremented**.



```
// This function is called everytime the user submits an answer.
async function submitAnswer() {
    if (!checkIfUserSelectedAnAnswer())
    {
        window.alert("Please choose an answer!");
        return;
    }

    evaluateAnswer(currentQuestion);
    currentQuestion = showNextQuestion();

    if (currentQuestion == null) { // Quiz Over.
        await displayScore();

        newQuiz(); // Start a new Quiz.
    }
}
```

When the user submits an answer, that action triggers the execution of this function which checks if the user chose an answer, and if so, it evaluates it and the game goes to the next question. If that question is null, then the quiz is over, the **score is displayed** and new quiz starts.

3. Class and Database representation

Each question and its answers are brought together in the **Question** entity and the questions are all stored in a **QuestionPool**. This question pool does the randomization and gives to the Quiz entity the question which will be presented to the user.

In my code, choices/answers of the user are not stored and then evaluated at the end, they are actually extracted after submitting each answer and also evaluated on the spot, so the score is computed incrementally during the quiz.

Progress during the quiz is tracked by an internal counter in the Quiz class which represents the index of the question displayed. When this value exceeds the number of questions in the quiz, the **getNext()** method will return a null value which will signify that the **quiz ended**.

In a **class diagram**, the Question entity is the smallest structure, then the QuestionPool is a container/collection of such Question objects, and finally the Quiz class has a reference to a question pool from which it obtains the desired number of problems in the quiz. If we would like to evaluate the answers given by the user at the end of the game, we could store the choices inside the Quiz structure and then parse the questions and answers in parallel, checking if the current choice matches the correct answer of the current question.

In a **database diagram**, the questions are stored in a Question table and the Quiz table actually has a **randomized slice** of a given number of question from the previously mentioned table. The choices could be stored in a separate table, each one of the having a foreign key which would refer to the ID of the question they answer.

Since each answer is given to a single, unique question, it means that between questions and answers we have a one-to-one relationship. Also, between the Question table and the Quiz we have a one-to-one relationship because each question appears **only once** in the Quiz (no repetitions).