Analysis part

First, before coding the application, a thorough analysis of the work to be done must be handed in. The analysis must include:
1. Some mock-ups of each view of the application
2. A breakdown of each React component that will compose the application
3. A breakdown of each WebSocket event that will be used.
4. A list of the HTTP routes used in the backend if any HTTP routes are used

2. React components

Each view in a React application is composed of components. You must list all the components that make up the application.
For each component, you must specify:
- Its name
- If it receives props from another component
- The state variables needed
- The action required on component load
- The interaction possible on this component (like a button click)
- The WebSocket events it is emitting or listening on

2 - List of react components:

Name: index.js

Props: It does not receive props.
State Variables: None required for this component.
Action on Component Load: This component is responsible for setting up the routing configuration using React Router (BrowserRouter and Routes). It doesn't have any specific actions on component load.
Interactions Possible: This component doesn't handle any user interactions or events itself.
WebSocket Events: There are no WebSocket events being emitted or listened to in this component.

App.jsx

Props: It does not receive props.
State Variables: None required for this component.
Responsible for app-wide layout and routing.
Action on Component Load: Display the welcome screen and main menu options.

MainMenu.jsx

Props: It does not receive props.
State Variables: None required for this component.
Action on Component Load: Display the main menu options for the user to select.
Interactions:
Button to navigate to the Teacher's view. (Host a quiz)
Button to navigate to the Student's view. (Join a quiz)
WebSocket Events: None needed for this component.

(CreateRoom.jsx) (Teacher)

Props: It does not receive props.
State Variables: None required for this component.
Action on Component Load: Display the main host a room options for the Teacher to select the category, difficulty, number of questions and time per question.
Button to start hosting a quiz.

WEBSOCKET:
create-room
Emitted by: CreateRoom
Listened by: Server
Actions: Create a new quiz room with specified details.

room-created-confirmation
Emitted by: Server
Listened by: CreateRoom
Actions: Send the host the quiz questions and answers.

StartRoom.jsx

Props: RoomCode to display
State Variables: numberOfStudents
Displays the room code and the number of students waiting.
There will be a “START” button to start the quiz

WEBSOCKET:
New-room-created:
Emitted by: Server
Listened by: StartRoom
Create a code for the room-id and send it to the client to display it in the start room.

player-joined
Emitted by: Server

Listened by: StartRoom
Actions: Update the player list when a player joins.

start-quiz-host
Emitted by: **StartRoom**
Listened by: Server
Actions: Send a message to the server to begin the quiz and distribute questions to players.

Host-view-Confirmation
Emitted by: Start-Room
Listened by: Server
Actions: Will confirm that this user is the host and make it so that the Teacher can't see the score part and will not be submitting questions or be displayed in the leaderboard.

(JoinQuiz.jsx) (Student)

Props: It does not receive props.
State Variables: None required for this component.
On load it will display two TextBox where the Student will enter his name and the Quiz Room number.
There will also be a button to enter the room.

player-click-joined
Emitted by: JoinQuiz
Listened by: Server
Actions: Send a message to the server when a player joins which updates the dictionary in the server with the player names and updates the waiting list of the host.

Join-room:
Emitted by: Server
Listened by: ***Join-quiz***
Actions: Join the room when the host decides to press the Start button in the host view .

WaitingRoomStudent.jsx

Props: Name of student
State variable: userName to be able to display the user's name to be displayed in the waiting menu.
It displays the name of students and saying to wait for the host to start the quiz
Websocket events:

quiz-started-player
Emitted by: Server
Listened by: WaitingRoomStudent
Actions: Display questions to players and start the timer.

Play.jsx

Props:none
State Variables Needed:

- Timer: keep track of time for each question
- displayScore: show the student their score after each question
- hostView: To determine if it is the teacher's view so we will not be showing him the score part.
- Correct_answer:To display the correct answer after the timer is done.
- activeQuestionIndex: To keep track of the index of the currently active question.
- questions: To store the list of trivia questions fetched from the API.
- quizFinished: To indicate whether the quiz has finished.
- score: To keep track of the player's score.
- isLoading: To indicate whether the API data is currently being fetched.
- selectedCategory: To store the selected category for the quiz.
- query: To handle query parameters using the useSearchParams hook.

Action: Fetch trivia questions from the API based on the selected category and other parameters.
Interactions Possible: Clicking the answer options triggers the selectAnswerHandler function, which updates the score and moves to the next question if applicable.

WebSocket events:

next-question
Emitted by: Server
Listened by: Play.
Actions: Show the next / first question and restart the timer

player-answered
Emitted by: Play after selecting an answer.
Listened by: Server.
Actions: Send the selected answer to the server for scoring.

question-display
Emitted by: Server after processing the player's answer.
Listened by: Play component.
Actions: Display the correct answer to update the score and stop the timer.

quiz-ended

Emitted by: Server after all questions are answered.
Listened by: Play component.
Actions: Show the final score and result.

Host-view
Emitted by: Server
Listened by: Play component
Actions: Will change the host view and make it so that the Teacher can't see the score part and will not be submitting questions or be displayed in the leaderboard.

Question.jsx

Props Received:

- question: The main question text.
- correct_answer: The correct answer to the question.
- incorrect_answers: An array of incorrect answer options.
- selectAnswerHandler: A function to handle the selection of an answer option.

State variable: None
Action: It displays the questions and the answers
Interaction: The answers buttons and whenever an answer is clicked then the correct answer displays
Websocket events:

submit-answer
Emitted by: PlayerView
Listened by: Server
Actions: Send the selected answer to the server for being displayed

Spinner.jsx

Props: None
State variable: None
Action: Displays a text: "Loading…"
Interaction: None
Websocket events:None

Leaderboard.jsx

Props: Name of students and their points
State variable: Name and points of students
Action: Displays the name and the points of all students in descending order in a table
Interaction: None

Websocket events:
leaderboard-updated
Emitted by: Server
Listened by: leaderboard
Actions: receive the points and the names of the players as a ranking to update the leaderboard

Row.jsx

Props received:

- name: A string representing the name to be displayed in the first <td> element.
- category: A string representing the category to be displayed in the second <td> element.
- score: A value representing the score to be displayed in the third <td> element.

State variable: None
Action: Displays the name and point of a single student
Interaction: None
WebSocket events: none

3. WebSocket events

Next, you must list all the WebSocket events that will be used in your application. For each event, specify:
- Its name
- Which part of the application emits it
- Which part of the application is listening for that event
- The actions that must be performed when this event occurs (datastore updates, new event fired, etc.)

4. HTTP requests

If any HTTP requests are made, then a list of the HTTP requests must be provided. For each request, specify:
- The route (path and parameters)
- The HTTP method
- The payload sent
- The actions that must be performed when this request occurs

POST /api/create-room

Method: POST
Payload: Quiz details (category, timer, difficulty, number of questions)
Actions: Create a new quiz room with the provided details.

```
const categoryName = localStorage.getItem("CategoryName");
    if (!selectedCategory) {
      setSelectedCategory({ id: query.get("categoryId"), name:
categoryName });
      return;
    }

    const url =
`${openTDhost}?amount=${numberOfQuestions}&category=${selectedCategory.id}
&difficulty=hard`;

    setIsLoading(true);

    async function fetchTrivia() {
      const triviaResponse = await fetch(url);
```

```
    const body = await triviaResponse.json();

    if (body.results) {
      setQuestions(body.results);
    }

    setTimeout(() => {
      setIsLoading(false);
    }, 500);
  }
```

Datastore vs Database
The backend server will keep the data stored about the current quiz rooms.
Reflect on what will be the data format and what data needs to be kept in the server. The data does not have to be persistent, so you can simply use JavaScript variables to hold the data. That means that the data will be wiped on server restart, but that is acceptable.
The need for a persistent database is not required here. A team can decide to use a database to keep data persistence, but it is not required to.
If a database is used, then the documents schema should be provided in the analysis.

For the storage of the leaderboard score we will be using a dictionary in the server-side that will be updated every time the client-side submits an answer.