

## Criterion C

### Techniques used:

- Servlet
- GSON
- SQLite-jdbc
- JQuery
- User-defined objects
- Objects as data records
- User-defined methods with parameters and appropriate return values
- Database
- Simple selection
- ArrayList
- HashMap
- Loops
- JavaScript + Database manipulation
- Sorting
- Searching
- GUI

## Servlet

One technique that I utilized is Web Servlets. Servlets are responsible for receiving user request and responding to these requests. The use of a Servlet also allows me to create a dynamic webpage, something very important for my product.

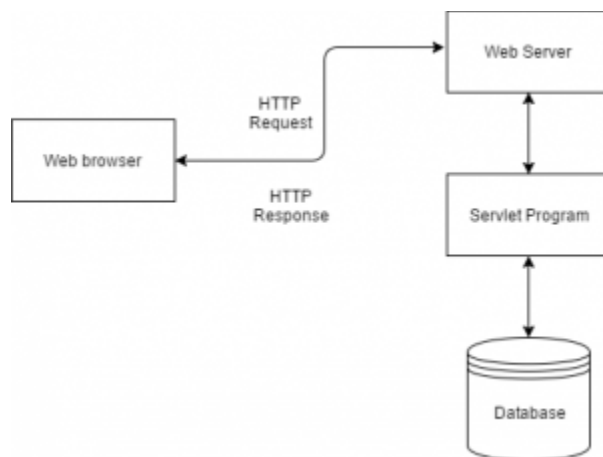


Figure 1. Servlet Architecture (Thakral, Kartik)

Shown below in figure 2, using the request doPost function, it allows me to respond to the user's login attempts securely.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String username = request.getParameter("user");
    String pwd = request.getParameter("password");
    if("interact".equals(username) && "god".equals(pwd)) { // if username and password matches preset details
        response.sendRedirect("Welcome.html");
    }else {
        response.sendRedirect("LoginPage.html");
    }
}
```

Figure 2. Interact Exec login code

Using the doGet function shown in figure 3, I am able to change edit the database through the DALManager class and hence changing the webpage.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    DALManager.getNextQuestion(DALManager.findCurrentGame());
    response.sendRedirect("admin.html");
}
```

Figure 3. doGet method to cycle to next question

## GSON

Using the GSON library, I was able to serialize and deserialize Java objects to JSON so it can be used and displayed in the front end shown in figure 4.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("application/json; charset=utf-8");
    response.setCharacterEncoding("UTF-8");
    ResponseUtil rsp = new ResponseUtil();
    DALManager manager = new DALManager();
    List<Question> listQuestion = manager.LoadQuestions(); // returns list of all questions in SQLite database
    rsp.setCode(200); // HTTP status code - 200 = success
    rsp.setData(listQuestion); // Data in list put into responseUtil
    rsp.setMessage("sucess.");

    Gson agson = new Gson(); //creates new Gson object
    String jsonObj = agson.toJson(rsp); // changes Gson to Json
    PrintWriter out = response.getWriter();
    out.print(jsonObj);
    out.flush(); // printwriter gets sent
}
```

Figure 4. use of GSON library

## JQuery

I used JQuery to help make my code more elegant and easier to maintain. It also gives me access to JavaScript's AJAX request protocol which allows my web application to send and retrieve data

from a server asynchronously.

```

<script src="jquery-3.5.1.js"></script>
<script type="text/javascript">
    function loadQuestionList() {
        $.get("questiondo", function(result, status){
            var q
            console.log(result.data)
            for(i=0; i<result.data.length; i++)
            {
                console.log(result.data[i].question)
                $("#tbl").append("<tr><td>" + result.data[i].questionID + "</td><td>" + result.data[i].question + "</td><td>" + result.data[i].answerchoice + "</td><td>" + result.data[i].answerchoice + "</td><td>" + result.data[i].answerchoice + "</td><td>" + result.data[i].
                    "</td><td>button onclick='\"showEditIdg[\" + result.data[i].questionID + \"\", \"\" + result.data[i].question + \"\", \"\" + result.data[i].answerchoice + \"\", \"\" + result.data[i].answerchoice + \"\", \"\" + result.data[i].
            }
        });
    }
}

```

Figure 5. use of JQuery library

```

<script src="jquery-3.5.1.js"></script>
<script type="text/javascript">
    function checkQuestion(){
        setInterval("getCurrentQuestion()",2000)
    }

    function getCurrentQuestion(){
        $.get("gameholder", function(result, status){
            console.log(result)
            console.log(result.questionID)
            $("#current_num").text(result.description)
            if(result.questionID ==1){
                window.location.assign("comp.html")
            }
        });
    }
}
</script>

```

Figure 6. use of JQuery library

## SQLite-jdbc

Using the SQLite-jdbc library, I can edit the SQLite database through code in JAVA. With this I can delete, edit, add, retrieve data from the databases which is crucial to the success of my project, as shown in figures 7,8,9 below.

```

public static int getCurrentQuestion(int id) { //takes in id of the current game
    int currentQuestion=0;
    Connection connection = DBUtil.getConnection();
    if(connection!=null) {
        try {
            String sql = "SELECT HasGameStarted FROM Game WHERE Game_ID="+id; // searches the database using SQLite-jdbc command
            PreparedStatement ps = connection.prepareStatement(sql); // sql statement is stored in a PreparedStatement
            ResultSet rlt = ps.executeQuery(); // PreparedStatement is executed, returns the object ResultSet object
            currentQuestion = rlt.getInt("HasGameStarted");
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return currentQuestion; // returns the current question that the game is on
}

```

Figure 7. use of SQLite-jdbc library to select a specific value in database

```

public static List<Question> loadQuestions() { // returns list of an arraylist of the object Question
    List<Question> list = new ArrayList<>();
    Connection connection = DBUtil.getConnection(); // establishes connection with database
    if(connection!=null) { // if connection is successful
        try {
            String sql = "SELECT * FROM question"; // Select everything from the 'question' table
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) { // whilst there are more questions
                Question question = new Question();
                question.setQuestionID(rlt.getInt("question_id"));
                question.setQuestion(rlt.getString("question_description"));
                question.setAnswerchoiceA(rlt.getString("question_choiceA"));
                question.setAnswerchoiceB(rlt.getString("question_choiceB"));
                question.setAnswerchoiceC(rlt.getString("question_choiceC"));
                question.setAnswerchoiceD(rlt.getString("question_choiceD"));
                question.setCorrectAnswer(rlt.getString("correct_answer").charAt(0));
                question.setTimelimit(rlt.getInt("time_limit"));
                question.setExplanation(rlt.getString("explanation"));
                list.add(question);
            }
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection(); // lastly, close connection with database
            }
        }
    }
    return list;
}

```

Figure 8. use of SQLite-jdbc library to retrieve values in database

```

public static void endGame() {
    Connection connection = DBUtil.getConnection();
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
    LocalDateTime now = LocalDateTime.now(); // gets the current time
    if(connection!=null) {
        try {
            String sql = "UPDATE Game SET HasGameStarted=-1, End_Time='"+dtf.format(now)+"'WHERE HasGameStarted!=-1"; // ends the running game by updating database
            PreparedStatement ps = connection.prepareStatement(sql);
            int rlt= ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
}

```

Figure 9. use of SQLite-jdbc library to update values in database

## User-defined Objects and Objects as data records

Creating objects such as Game, Section, Team allows me to store and retrieve the necessary data easily with the use of setters and getters and creating lists of certain Objects.





- ▼  com.andrew.model
  - >  Game.java
  - >  Question.java
  - >  Team.java

Figure 10. User-defined Objects in my IA

```

public class Game {
    int gameID;
    String gameName;
    String gameStartTime;
    String gameEndTime;
    int hasGameStarted;
    public int getGameID() {
        return gameID;
    }
    public void setGameID(int gameID) {
        this.gameID = gameID;
    }
    public String getGameName() {
        return gameName;
    }
    public void setGameName(String gameName) {
        this.gameName = gameName;
    }
    public String getGameStartTime() {
        return gameStartTime;
    }
    public void setGameStartTime(String gameStartTime) {
        this.gameStartTime = gameStartTime;
    }
    public String getGameEndTime() {
        return gameEndTime;
    }
    public void setGameEndTime(String gameEndTime) {
        this.gameEndTime = gameEndTime;
    }
    public int getHasGameStarted() {
        return hasGameStarted;
    }
    public void setHasGameStarted(int hasGameStarted) {
        this.hasGameStarted = hasGameStarted;
    }
}

```

Figure 11. Game class

```

public static List<Question> loadQuestions() { // returns list of an arraylist of the object Question
    List<Question> list = new ArrayList<>();
    Connection connection = DBUtil.getConnection(); // establishes connection with database
    if(connection!=null) { // if connection is successful
        try {
            String sql = "SELECT * FROM question"; // Select everything from the 'question' table
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) { // whilst there are more questions
                Question aquestion = new Question();
                aquestion.setQuestionID(rlt.getInt("question_id"));
                aquestion.setQuestion(rlt.getString("question_description"));
                aquestion.setAnswerchoiceA(rlt.getString("question_choiceA"));
                aquestion.setAnswerchoiceB(rlt.getString("question_choiceB"));
                aquestion.setAnswerchoiceC(rlt.getString("question_choiceC"));
                aquestion.setAnswerchoiceD(rlt.getString("question_choiceD"));
                aquestion.setCorrectAnswer(rlt.getString("correct_answer").charAt(0));
                aquestion.setTimelimit(rlt.getInt("time_limit"));
                aquestion.setExplanation(rlt.getString("explanation"));
                list.add(aquestion);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection(); // lastly, close connection with database
            }
        }
    }
    return list;
}

```

Figure 12. creating an ArrayList with the object Question

## User-defined Methods (with and without parameters and appropriate return values)

With user-defined methods with both parameters and a return value, I can pass in some values and get a returned value after some operations, as shown in figure 13.

```

public static int TeamEnter(String team, int gamecode) { // takes in the team name and the gamecode
    Connection connection = DBUtil.getConnection();
    Random rnd = new Random();
    int n = 10000 + rnd.nextInt(900000);
    int teamid = gamecode+n; // generates TeamID
    if(connection!=null) {
        try {
            Statement statement = connection.createStatement();
            Statement statement2 = connection.createStatement();
            String sql = "insert into Team(Team_ID, Team_Name)values('"+teamid+"','"+team+"')"; // adds team into database
            String sql2 = "insert into Game_Team_bridge(Game_ID, Team_ID, Team_points)values('"+gamecode+"','"+teamid+"','"+0+"')";
            int rlt = statement.executeUpdate(sql);
            int rlt2 = statement.executeUpdate(sql2);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return teamid; // returns the unique TeamID of the team that just joined
}

```

Figure 13. TeamEnter method that takes in two variables and returns a team ID of the newly entered team

Some other user-defined methods have parameters but no return value with the method declaration of void. These methods help me complete an action with given information, such as in figure 14, given the id of a question, it deletes it from the database.

```
public static void deleteQuestion(int id) {
    DALManager manager = new DALManager();
    List<Question> list = manager.LoadQuestions();
    int i=0;
    while(list.get(i).getQuestionID()!=id) {
        i++;
    }
    DALManager.saveQuestion(list.get(i), "delete");
}
```

Figure 14. deleteQuestion method that takes in the id of a question and deletes the question in database.

Other user-defined methods have a return value but no parameters. These are used to perform an operation and return a value as a result of the operation. Such as in figure 15, the method loadGame() starts a new game and returns the unique game id generated.

```
public static int loadGame() {
    Connection connection = DBUtil.getConnection();
    Game game = new Game(); // creates new Game object
    List<Game> list = new ArrayList<>();
    int id = GameManagement.generateGameCode(); // generates game code
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss"); // gets current date and time
    DateTimeFormatter yr = DateTimeFormatter.ofPattern("yyyy"); // gets current year
    LocalDateTime now = LocalDateTime.now();
    String name = "TriviaNight"+yr.format(now);
    if(connection!=null) {
        try { // adds game data into database
            String sql = "insert into game(Game_ID, Game_Name, Start_Time, HasGameStarted)values('"+id+"','"+name+"','"+dtf.format(now)+"'";
            Statement statement = connection.createStatement();
            int rlt = statement.executeUpdate(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return id;
}
```

Figure 15. loadGame that starts a new game by adding the information into the database.

## Database Structure

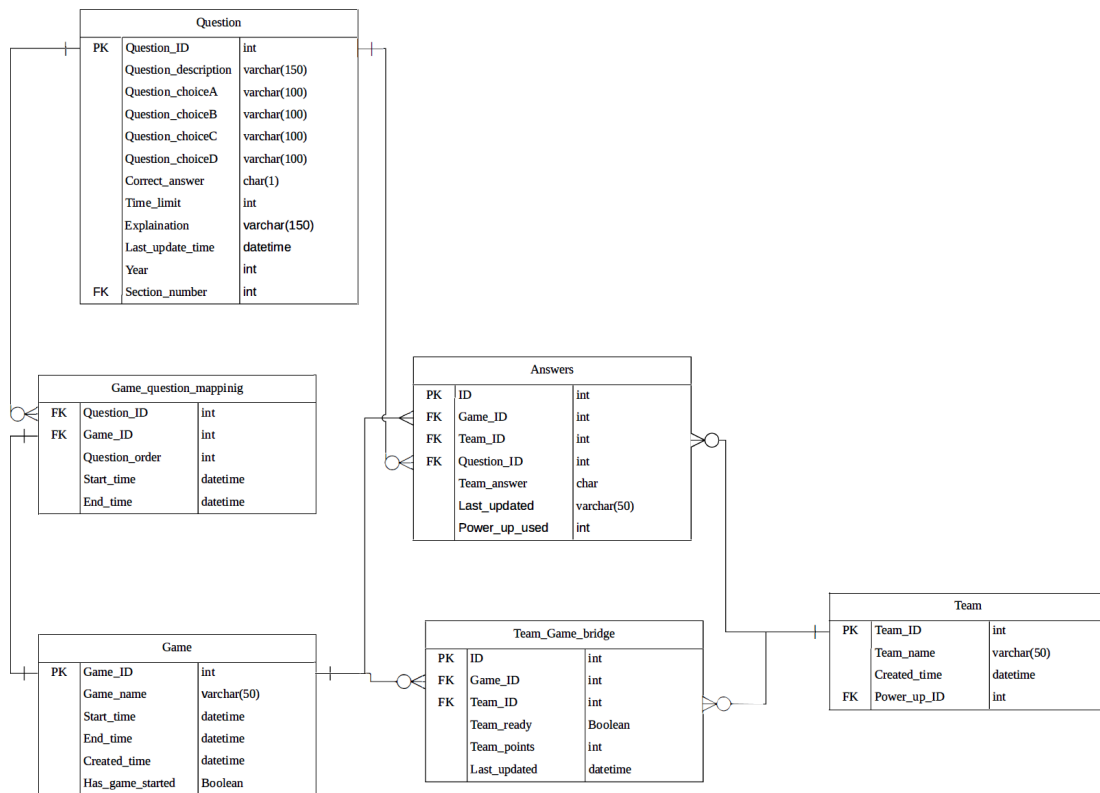


Figure 16. ERM diagram of database showing its structure and relations.

My database structure was designed with normalization in mind to help reduce data redundancy. With the use of Primary and Foreign keys as marked on figure 16, I can split a large table into smaller tables and link the tables using relationships, reducing redundancy and making sure that data is stored logically.

Validation is achieved by varying the field types upon creation to ensure that the correct data type is stored, checking the presence of some variables making sure it is not left blank, and for some variables making sure the entered information is unique.

Name	Data type	大小	比例	Not null?	Key
question_id	integer			<input checked="" type="checkbox"/>	
question_description	TEXT			<input checked="" type="checkbox"/>	
question_choiceA	TEXT			<input checked="" type="checkbox"/>	
question_choiceB	TEXT			<input checked="" type="checkbox"/>	
question_choiceC	TEXT			<input checked="" type="checkbox"/>	
question_choiceD	TEXT			<input checked="" type="checkbox"/>	
correct_answer	TEXT			<input checked="" type="checkbox"/>	
time_limit	integer			<input checked="" type="checkbox"/>	
explanation	TEXT			<input checked="" type="checkbox"/>	
last_update_time	TEXT			<input checked="" type="checkbox"/>	
year	integer			<input checked="" type="checkbox"/>	
section_number	INTEGER			<input checked="" type="checkbox"/>	
showAnswer	TEXT			<input checked="" type="checkbox"/>	
ID	real	▼		<input checked="" type="checkbox"/>	1

Figure 17. database settings



## Simple Selection

If statements are used to ensure that the correct action is carried out based on input of user. In figure 18, a question is either added, deleted, or edited based on which button the user presses.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String act = request.getParameter("act");
    System.out.println(act); // debugging
    if ("add".equalsIgnoreCase(act)) { // add new question
        int id = Integer.parseInt(request.getParameter("questionID"));
        String question = request.getParameter("description");
        String choiceA = request.getParameter("choiceA");
        String choiceB = request.getParameter("choiceB");
        System.out.println("choiceB");
        String choiceC = request.getParameter("choiceC");
        String choiceD = request.getParameter("choiceD");
        int timelimit = Integer.parseInt(request.getParameter("time"));
        char correctChoice = request.getParameter("correctanswer").charAt(0);
        String explanation = request.getParameter("explanation");
        QuestionManagement.createQuestion(id, question, choiceA, choiceB, choiceC, choiceD, timelimit, correctChoice, explanation);
    } else if ("edit".equalsIgnoreCase(act)) { // edit existing question
        int id = Integer.parseInt(request.getParameter("questionID"));
        String question = request.getParameter("description");
        String choiceA = request.getParameter("choiceA");
        String choiceB = request.getParameter("choiceB");
        String choiceC = request.getParameter("choiceC");
        String choiceD = request.getParameter("choiceD");
        int timelimit = Integer.parseInt(request.getParameter("time"));
        char correctChoice = request.getParameter("correctanswer").charAt(0);
        String explanation = request.getParameter("explanation");
        QuestionManagement.updateQuestion(id, question, choiceA, choiceB, choiceC, choiceD, timelimit, correctChoice, explanation);
    } else if ("delete".equalsIgnoreCase(act)) { // delete existing question
        int id = Integer.parseInt(request.getParameter("questionID"));
        QuestionManagement.deleteQuestion(id);
    }
    response.sendRedirect("Question.html"); // refreshes page
    //doGet(request, response);
}
```

Figure 18. doPost method that either adds, edits, or deletes question based on user selection.

In figure 19, the validity of the game code is checked and an action is performed based on if the code is valid or not.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    String team = request.getParameter("team");
    int game = Integer.parseInt(request.getParameter("code"));
    DALManager mal = new DALManager();
    if (mal.codecheck(game)) { // if gamecode is linked to an active game, the team is entered into the game
        DALManager.saveTeamInfo(mal.TeamEnter(team, game), team);
        response.sendRedirect("ContestantPage.html");
    }
    else {
        response.sendRedirect("GameCodeError.html"); // else redirected to error page
    }
}
```

Figure 19. doPost method that checks the validity of code entered in the front end by user

## ArrayList

ArrayLists are used to store information about questions, teams, and sections as it does not have a fixed size. The number of questions, teams and sections for the contest change from year to year and it is faster than LinkedList and Vectors for direct access. Uses shown in figure 20. I also used a ArrayList to store ArrayLists for my leaderboard algorithm as shown in figures 21, 22.

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("application/json; charset=utf-8");
    response.setCharacterEncoding("UTF-8");
    ResponseUtil rsp = new ResponseUtil();
    DALManager manager = new DALManager();
    List<Question> listQuestion = manager.LoadQuestions(); // returns list of all questions in SQLite database
    rsp.setCode(200); // HTTP status code - 200 = success
    rsp.setData(listQuestion); // Data in list put into responseUtil
    rsp.setMessage("sucess.");
}

```

Figure 20. ArrayList used to store Questions

```

public static ArrayList<ArrayList<Integer>> displayRanking() { // returns an arraylist of arraylists
    int id = DALManager.findCurrentGame();
    Connection connection = DBUtil.getConnection();
    HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>(); // HashMap mapping TeamID to Points
    ArrayList<ArrayList<Integer>> leaderboard = new ArrayList<ArrayList<Integer>>();
    if(connection!=null) {
        try {
            String sql = "SELECT * FROM Game_Team_bridge WHERE Game_ID="+id;
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) {
                hm.put(rlt.getInt("Team_ID"),rlt.getInt("Team_points"));
            }
            ArrayList<Integer> temp = new ArrayList<Integer>(hm.keySet()); // temporary array that holds the id of teams
            ArrayList<Integer> points = new ArrayList<Integer>(hm.values()); // array that holds the amount of points each team has
            ArrayList<Integer> teams = new ArrayList<Integer>(); // empty array
            Collections.sort(points); // points sorted in ascending order
            while (!temp.isEmpty()) {
                for (int i=0;i<points.size();i++) { // enclosed for loop to try match the teams with the points that is sorted
                    for (int j=0; j<temp.size();j++){
                        if (hm.get(temp.get(j))==points.get(i)) {
                            teams.add(temp.get(j));
                            temp.remove(j);
                            break;
                        }
                    }
                }
            }
            leaderboard.add(teams);
            leaderboard.add(points);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return leaderboard; // returns arraylist of arraylists with the first arraylist being the teams and second being points, both in
    // ascending order of ranking
}

```

Figure 21. ArrayList used to store other ArrayLists

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("application/json; charset=utf-8");
    response.setCharacterEncoding("UTF-8");
    ResponseUtil rsp = new ResponseUtil();
    DALManager manager = new DALManager();
    ArrayList<ArrayList<Integer>> rank = TeamManagement.displayRanking();
    ArrayList<String> teams=TeamManagement.returnNames(rank.get(0));
    ArrayList<ArrayList<Object>> ranking = new ArrayList<ArrayList<Object>>(); // arraylist of object arraylists to store both String and int
    ArrayList<Object> team = new ArrayList<Object>();
    ArrayList<Object> points = new ArrayList<Object>();
    for (String i:teams) {
        team.add(i);
    }
    for (int i:rank.get(1)) {
        points.add(i);
    }
    ranking.add(team);
    ranking.add(points);
}

```

Figure 22. ArrayList of all Questions in the database stored as a Question Object

# HashMap

```
public static ArrayList<ArrayList<Integer>> displayRanking() { // returns an arraylist of arraylists
    int id = DALManager.findCurrentGame();
    Connection connection = DBUtil.getConnection();
    HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>(); // HashMap mapping TeamID to Points
    ArrayList<ArrayList<Integer>> leaderboard = new ArrayList<ArrayList<Integer>>();
    if(connection!=null) {
        try {
            String sql = "SELECT * FROM Game_Team_bridge WHERE Game_ID="+id;
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) {
                hm.put(rlt.getInt("Team_ID"),rlt.getInt("Team_points"));
            }
            ArrayList<Integer> temp = new ArrayList<Integer>(hm.keySet()); // temporary array that holds the id of teams
            ArrayList<Integer> points = new ArrayList<Integer>(hm.values()); // array that holds the amount of points each team has
            ArrayList<Integer> teams = new ArrayList<Integer>(); // empty array
            Collections.sort(points); // points sorted in ascending order
            while (!temp.isEmpty()) {
                for (int i=0;i<points.size();i++) { // enclosed for loop to try match the teams with the points that is sorted
                    for (int j=0; j<temp.size();j++){
                        if (hm.get(temp.get(j))==points.get(i)) {
                            teams.add(temp.get(j));
                            temp.remove(j);
                            break;
                        }
                    }
                }
            }
            leaderboard.add(teams);
            leaderboard.add(points);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return leaderboard; // returns arraylist of arraylists with the first arraylist being the teams and second being points, both in
                        // ascending order of ranking
}
```

Figure 23. Use of a HashMap mapping TeamID to team points

The use of a HashMap allowed me to associate the ID of the teams to their score, making it easier for me to keep track of the scores of each team, shown in figure 23. HashMaps are extremely efficient for this purpose, using a hash function to quickly map the key to a data value, while not taking up extra space from statically allocating arrays.

## Loops

Enhanced For loops are used when it is not important for us to know the index of the current element such as when cycling through an ArrayList as shown in figure 24. A while loop is used when we are unsure of the number of cycles a loop needs to be ran as shown in figure 25. A nested For loop is also used in figure 24 to compare values of two data structures. For loops are also used to cycle through values of an array, such as the stored JSON values shown in figure 26.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("application/json; charset=utf-8");
    response.setCharacterEncoding("UTF-8");
    ResponseUtil rsp = new ResponseUtil();
    DALManager manager = new DALManager();
    ArrayList<ArrayList<Integer>> rank = TeamManagement.displayRanking();
    ArrayList<String> teams=TeamManagement.returnNames(rank.get(0));
    ArrayList<ArrayList<Object>> ranking = new ArrayList<ArrayList<Object>>(); // arraylist of object arraylists to store both String and int
    ArrayList<Object> team = new ArrayList<Object>();
    ArrayList<Object> points = new ArrayList<Object>();
    for (String i:teams) {
        team.add(i);
    }
    for (int i:rank.get(1)) {
        points.add(i);
    }
    ranking.add(team);
    ranking.add(points);
}
```

Figure 24. Use of enhanced for loops

```

public static ArrayList<ArrayList<Integer>> displayRanking() { // returns an arraylist of arraylists
    int id = DALManager.findCurrentGame();
    Connection connection = DBUtil.getConnection();
    HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>(); // HashMap mapping TeamID to Points
    ArrayList<ArrayList<Integer>> leaderboard = new ArrayList<ArrayList<Integer>>();
    if(connection!=null) {
        try {
            String sql = "SELECT * FROM Game_Team_bridge WHERE Game_ID="+id;
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) {
                hm.put(rlt.getInt("Team_ID"),rlt.getInt("Team_points"));
            }
            ArrayList<Integer> temp = new ArrayList<Integer>(hm.keySet()); // temporary array that holds the id of teams
            ArrayList<Integer> points = new ArrayList<Integer>(hm.values()); // array that holds the amount of points each team has
            ArrayList<Integer> teams = new ArrayList<Integer>(); // empty array
            Collections.sort(points); // points sorted in ascending order
            while (!temp.isEmpty()) {
                for (int i=0;i<points.size();i++) { // enclosed for loop to try match the teams with the points that is sorted
                    for (int j=0; j<temp.size();j++){
                        if (hm.get(temp.get(j))==points.get(i)) {
                            teams.add(temp.get(j));
                            temp.remove(j);
                            break;
                        }
                    }
                }
                leaderboard.add(teams);
                leaderboard.add(points);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return leaderboard; // returns arraylist of arraylists with the first arraylist being the teams and second being points, both in
    // ascending order of ranking
}

```

Figure 25. Use of nested for loops and while loop

```

<script src="jQuery-3.5.1.js"></script>
<script type="text/javascript">
function loadQuestionList() {
    $.get("Questiondo", function(result, status){
        var q
        console.log(result.data)
        for(i=0; i<result.data.length; i++)
        {
            console.log(result.data[i].question)
            $('#qul').append("<tr><td>"+result.data[i].questionId+"</td><td>"+result.data[i].question+"</td><td>"+result.data[i].answerchoice+"</td><td>"+result.data[i].answerchoice+"</td><td>"+result.data[i].answerchoice+"</td><td>"+result.data[i].
                "</td><td>button onclick='\"showEditDlg\"edit','\""+result.data[i].questionId+"\""+result.data[i].question+"\""+result.data[i].answerchoice+"\""+result.data[i].answerchoice+"\""+result.data[i].
            ));
        }
    });
}

```

Figure 26. Used for loop used to cycle through JSON information stored in front end

## JavaScript + Database manipulation

Using the JS, the setInterval function specifically, I can constantly search the database for changes and adjust the page base on it as shown by figure 26 and 27 below.

Game_ID	Game_Name	Start_Time	End_Time	Game_Process
142516	TriviaNight20:2020/10/28 1	(Null)		0
604230	TriviaNight20:2020/09/29 0	2020/09/29		-1

Figure 27. "Game" table from SQLite database

```

<script src="jquery-3.5.1.js"></script>
<script type="text/javascript">
    function checkQuestion(){
        setInterval("getCurrentQuestion()",2000)
    }

    function getCurrentQuestion(){
        $.get("gameholder", function(result, status){
            console.log(result)
            console.log(result.questionID)
            $("#current_num").text(result.question)
        });
    }
</script>
<head>
<meta charset="ISO-8859-1">
<title>Host</title>
</head>
<body onload="checkQuestion()">
<h1>Question:</h1>
<div>
    <span id="current_num"></span>
</div>
</body>
</html>

```

Every few seconds the program will run the method which checks and returns the value of the Game\_Process column. If the value equals to 0 it means the game is in the 'welcome screen', anything bigger than one gives us the question number, and finally if it is -1 it means that the game has ended.

The webpage will change based on the changes in the database.

Figure 28. html code for changing page for different questions.

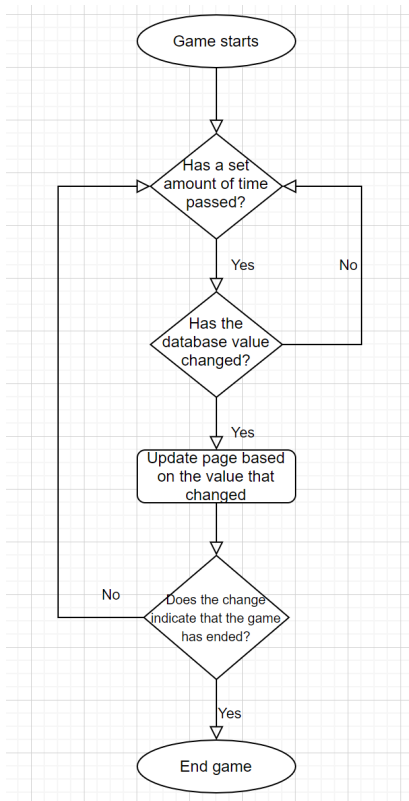


Figure 29. html code for changing page for different questions.

## Sorting

Sorting of ranking of the teams for the leaderboard is done through first sorting the points of the teams, and then matching the teams with the points. This is done through the use of a HashMap and a nested for loop to compare values.



Figure 30. Sorting using HashMap and comparing the values with sorted points ArrayList

## Searching

Linear search is used in the program when searching for values in the database. Linear search is the most practical due to the small size of the database and values of the database being unsorted.

```
public static boolean codecheck(int code) { // checks if the gamecode exists
    Connection connection = DBUtil.getConnection();
    if(connection!=null) {
        try {
            String sql = "SELECT * FROM Game";
            PreparedStatement ps;
            ps = connection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) { // linear search in database to see if the gamecode exists
                if (code == rlt.getInt("Game_ID")) {
                    return true; // if yes returns true
                }
            }
            return false;
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(connection!=null) {
                DBUtil.closeConnection();
            }
        }
    }
    return false;
}
```

Figure 31. codeCheck method which uses linear search to search for game code entered in parameter

```

public static int findCurrentGame() {
    int game = -1;
    Connection dbConnection = DBUtil.getDbConnection();
    if(dbConnection!=null) {
        try {
            String sql = "SELECT Game_ID FROM Game WHERE Game_Process>-1";
            PreparedStatement ps = dbConnection.prepareStatement(sql);
            ResultSet rlt = ps.executeQuery();
            while(rlt.next()) {
                game = rlt.getInt("Game_ID");
                break;
            }

            rlt.close();
            ps.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if(dbConnection!=null) {
                try {
                    dbConnection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    return game;
}

```

Figure 32. findCurrentGame method which uses linear search to search for the active game

## GUI

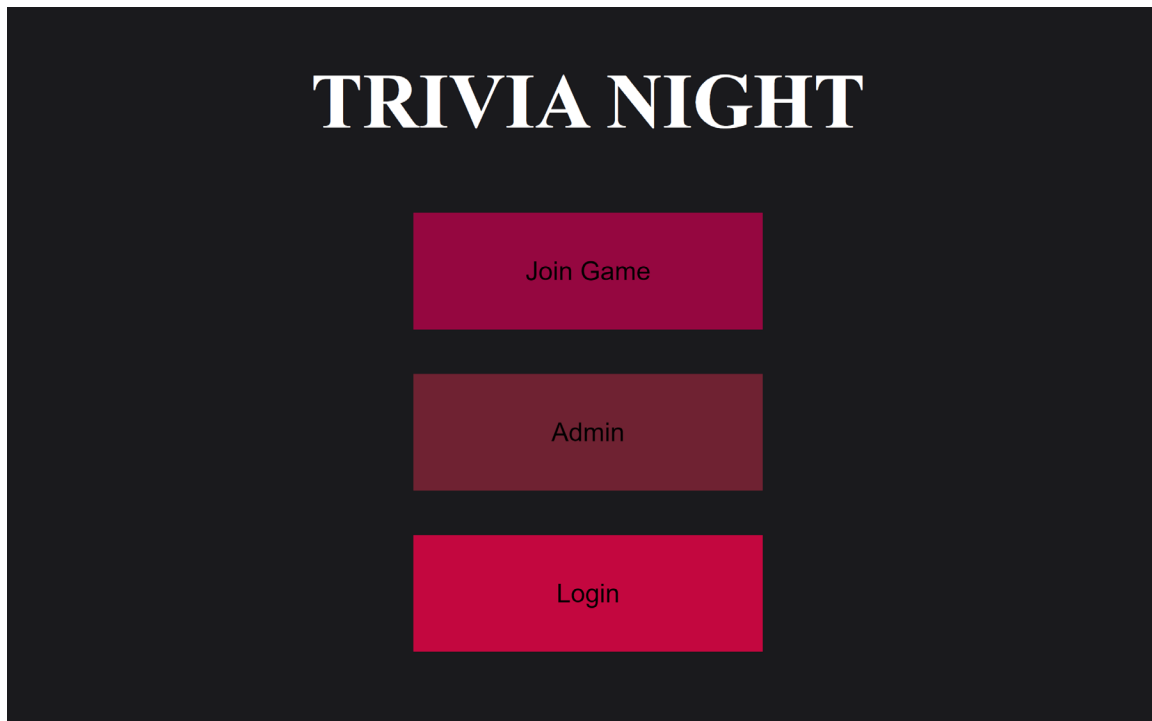


Figure 33. Front page of program

Figure 33. admin page of program

[Next Question](#) [Previous Question](#) [Show Explanation](#) [End Game](#)

### Current Question:

How much is bitcoin worth right now?

Tue Feb 23 2021 19:45:41 GMT+0800 (China Standard Time)

Ranking	Team	Points
1	nnnn	300
2	andrew and his friends	200
3	bandrew	100
4	bandrew	100
5	bandrew	0
6	ands	0

10

### How much is bitcoin worth right now?

50 thousand us dollars

50k thousand us dollars

500k thousand us dollars

5m thousand us dollars

Figure 33. competition page of program for contestants



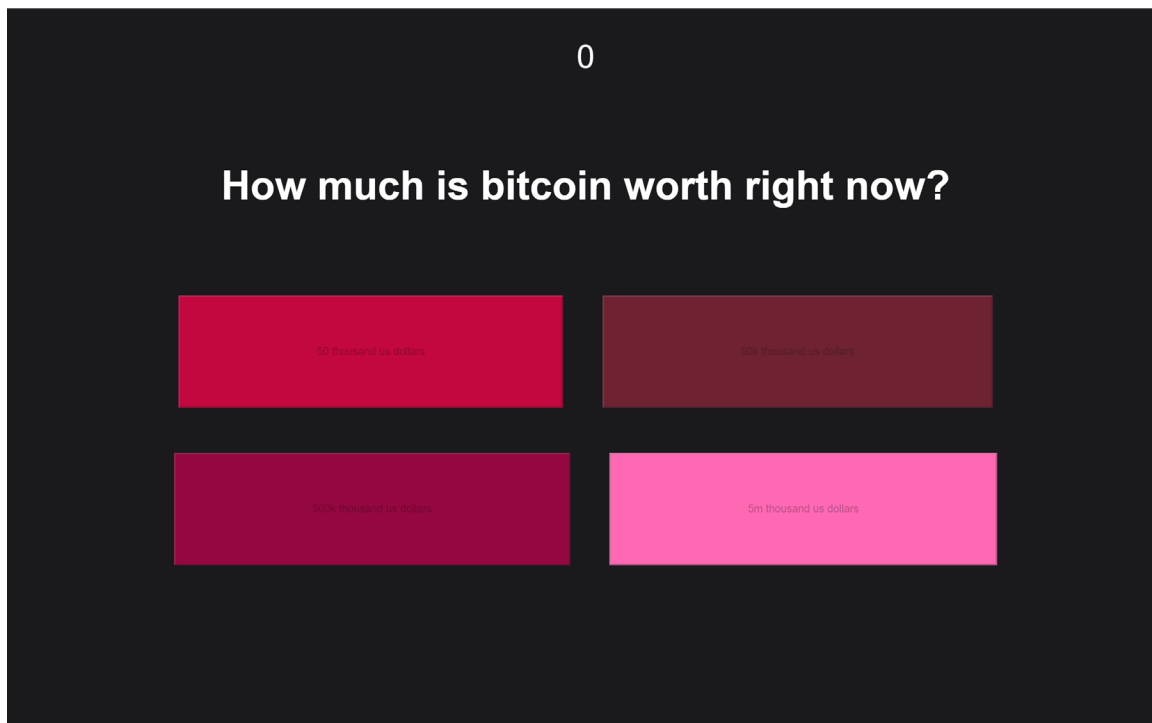


Figure 33. When timer = 0, buttons become unresponsive

With the use of html and css, I am able to make my GUI very user friendly and straight forward and fulfilling requests from my client.

WORD = 879

#### Bibliography:

Thakral, Kartik. "Introduction to Java Servlets." *GeeksforGeeks*, 23 Oct. 2019, [www.geeksforgeeks.org/introduction-java-servlets/](http://www.geeksforgeeks.org/introduction-java-servlets/).