

# 2012480\_Zack\_Costello\_Group\_ Report

*by* Zack Daniel Costello

---

|                |                                                                                                   |                 |       |
|----------------|---------------------------------------------------------------------------------------------------|-----------------|-------|
| FILE           | 24239_ZACK_DANIEL_COSTELLO_2012480_ZACK_COSTELLO_GROUP_REPO<br>RT_312866_1483206263.PDF (352.99K) |                 |       |
| TIME SUBMITTED | 11-DEC-2020 04:59PM (UTC+0000)                                                                    | WORD COUNT      | 3099  |
| SUBMISSION ID  | 139294126                                                                                         | CHARACTER COUNT | 18206 |

# Kartastic: The Design and Implementation of a Car Rental Database Application

Zack Costello

*Student Number: 2012480*

2012480@student.uwtsd.ac.uk

Module: ACCA7000, ACCA7001, ACCA7002

Team Members: Riley Croucher, Rokas Rudys and Ridwan Chowdhury

## I. INTRODUCTION

Over the course of the week 07/12/2020 to 11/12/2020 a week-long group project was assigned, with the goal of designing and building a text-based console application written in Java to interact with database; the scenario was selected by the team from two options at the start of the week. Both the application and the database had to be designed from scratch and had to include certain specific features, ensuring that an Agile workflow is maintained throughout the project.

## II. SCENARIO

The scenario given was to design the database and console application for a car rental company, which the team named Kartastic (a portmanteau of the words 'car' and 'fantastic' with an additional spelling alteration). The application had to allow customers to both rent and track the status of their cars that were in their inventory. It also had to allow the management of the company's cars, including adding new cars and removing old cars from the inventory. Multiple cars of the same make, type, and model had to be possible to add to the inventory and for cars to be able to be rented out to multiple customers at the same time (unless they have all been booked).

### *Required User Operations*

A specific list of required operations were given in the requirements document. These were given as all being performed by the customer. However, there were several items which were likely not intended to be an operation that the customer would perform (such as booking rental vehicles for maintenance and repair), so the first action by the team was to sort the list operations into the categories of user that would perform the action and refine some of the requirements so that they were more clearly defined.

### **All System Users:**

- Register to gain a username and password.
- Login using the name and password.
- View existing bookings:
  - Customers can only view their booking.
  - Rental managers can view all bookings.
- Cancel a booking:
  - Customers can cancel only their booking.
  - Rental managers can cancel any booking.
- Specify the pickup location: country, city and street.
- Browse all cars according to their type, make and availability at a pickup location.
- Search for a specific car make and model at a pickup location.

- Select a rental car to view its availability.

#### **Customers:**

- Book an available vehicle at a pickup location. The stages required are:
  - 1) Select the drop off location.
  - 2) Enter name and home address.
  - 3) Enter their payment details.
  - 4) Confirm booking details.

#### **Rental Managers:**

- Change a vehicle booking.
- The ability to add a new vehicle.
- Suspend bookings.
- Remove a vehicle.
- Book a vehicle for maintenance or repair.
- View all vehicles and their status.
- View all vehicle rentals.
- Search for rentals.
- Cancel a rental.
- Swap a vehicle.
- Record a pickup.
- Record a drop off.
- Request vehicle collection.

### **III. AGILE DESIGN PROCESS**

As the Kartastic team had previous experience using the Scrum framework, this was chosen as the Agile methodology for this project. As the entire project was only 5 days long, some of the ceremonies were by necessity shortened (and the Sprint Reflection entirely omitted) but in general Scrum methods were adhered to. For example, the Product Backlog and Sprint Backlog were effectively combined as the code sprint was 5 days; the number of planning sessions, refinement, and other events would have to be limited in order to not limited progress.

#### *A. Progress Definitions*

It was agreed upon that following definition of ready (DoR) and definition of done (DoD) would be used:

##### *Definition of Ready*

- The team must adequately understand the architecture.
- Stories must be written as a User Story.
- The acceptance criteria must exist and be understood by the entire team.
- There must be no more than 3 acceptance criteria.
- The story must have been estimated by the team.

##### *Definition of Done*

- The story must have been fully developed.
- All pushes to the main code branch must have been peer reviewed (in the form of approving a pull request).
- All tests must have been passed.
- There must be no outstanding bugs.

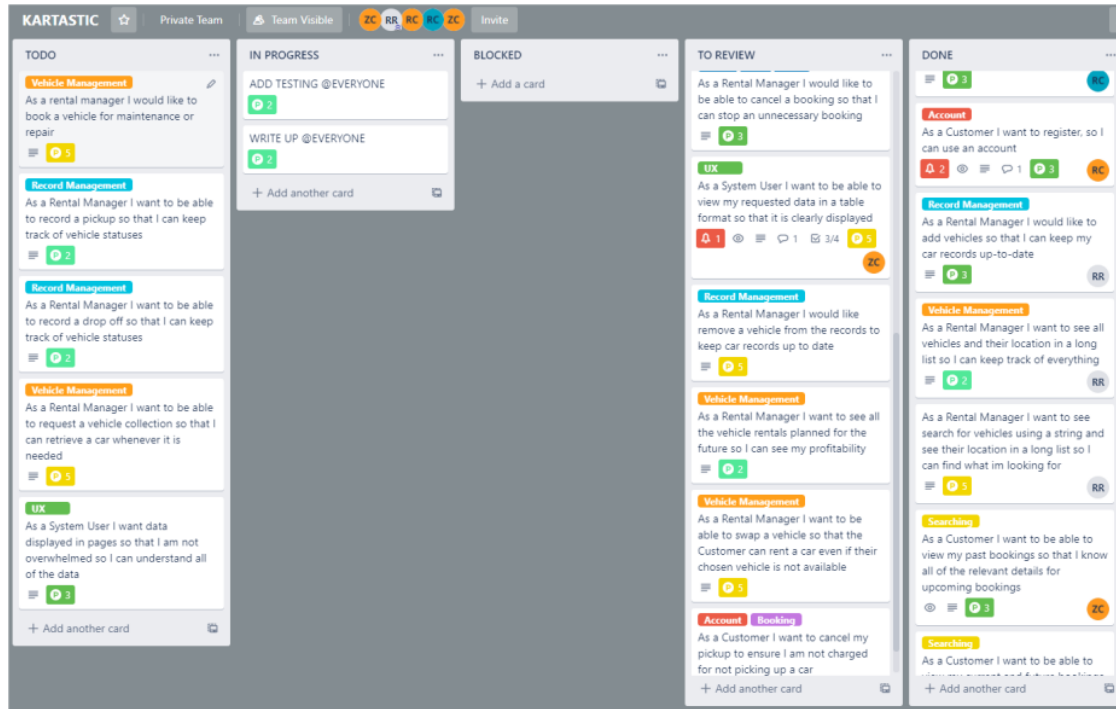


Fig. 1: The Trello board used for the code sprint. The board shown was taken towards the end of the sprint.

### B. Product and Sprint Backlogs

The team chose to use Trello as the platform for the Scrum board as it required minimal setup. The first task, performed collaboratively by the entire team on the morning of Day 1, was to use the requirements document to create the sprint backlog and, where possible, refine them into appropriate scope for the sprint.

The user operations listed Section II were used as the basis for many of the user stories, as they were clearly listed features which did not need a lot of refinement (at least initially).

### C. Scrum Board

The Trello board used for the code sprint is shown in Fig. 1. After the Sprint Backlog had been created, all of the user stories were estimated using Fibonacci numbers according to their perceived difficulty/time to complete. As there were a range of experiences within the team and the team had not previously participated in a code sprint of this scope using Scrum techniques before, discussion of the estimations by each member was particularly important so that all team members could meaningfully contribute to the project.

An example user story, with a description showing an agreed-upon acceptance criteria, is shown in Fig. 2. This story in particular was selected as it contains examples of all of the components of Scrum practices. A checklist was added to show the technical steps needed in adding the feature this story describes. It also demonstrates some of the requirements of Agile; the story was moved to *Review* instead of *Done* as the story had not been tested enough. If more time was available, the story would likely have to be returned to *In Progress* upon further peer

review as thorough testing is an integral part of Agile working methods.

Also, due to the time limitations, the scope of some stories inevitably were found to be more difficult than initially estimated (due to inexperience with the technologies involved). However, it was correctly identified that as more stories were completed, code could be reused so that momentum increase as the sprint progressed, which was reflected in the amount of stories moved across the board towards the end of the sprint.

#### *D. Sprint Evaluation*

The amount of progress made during the sprint could be measured using the story points assigned during the story estimation. Trello contains a feature to display the point counts, which is shown for the sprint in Fig. 3

If a Sprint Retrospective was held, there would likely be several common observations made by all members.

There would have been far more emphasis on testing, as whilst some testing was performed the complexity of some of the code meant that completing certain stories took longer for some members than others (due to experience). For example, a generalised function to make SQL queries to the database was created that used the data type `ArrayList<ArrayList<String>>`, i.e. a 2D `ArrayList`. Whilst this allowed for greater flexibility, some members of the group had not worked with this before (amongst other techniques such as the extensive use of Object Oriented programming). This meant that testing was harder to apply to these stories as understanding of these data types is necessary in order to correctly interact with them and know the format of the desired responses.

### IV. DATABASE DESIGN

The database was designed by the entire group due to it otherwise being a major blocker to the major parts of the console application. Therefore, all members contributed to this part of the project. The design was done using Draw.io, with the Entity-Relationship Diagram (ERD) for the database being progressively produced as the design unfolded. The ERD produced is shown in Fig. 4

As shown in the ERD, many different relationships are present between tables, forming a relatively interdependent database. Therefore, normalisation was a necessary process to be carried out.

#### *A. User Story: Adding Database Tables and Seeding*

Seeding the database tables was performed by all members of the group, with each member seeding a selection of the tables. The *JavaFaker* library was used in order to seed fake data into each field, with appropriate data being selected from those available in the library. For example, for a hashed password it was chosen to add random SHA256 hashed keys as a demonstration.

### V. CONSOLE APPLICATION

#### *A. User Story: Adding Table Formatting*

Adding a general table formatting function, which allows the displaying of database query results in a neatly-formatted table, was an important component of the console program

#### *B. User Story: View Existing Bookings as a Customer*

A separate class was created for this story, with the addition of a menu that the customer can access within the console application. This accessed functions in several other classes, as well as making several SQL queries that involved multiple inner joins. The WHERE string is concatenated with the relevant condition depending on the option selected in the menu.



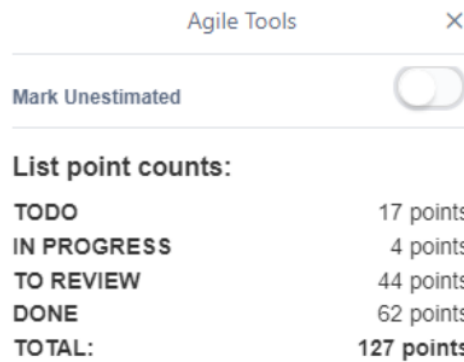


Fig. 3: The point counts for the sprint, calculated from the Scrum board on Trello.

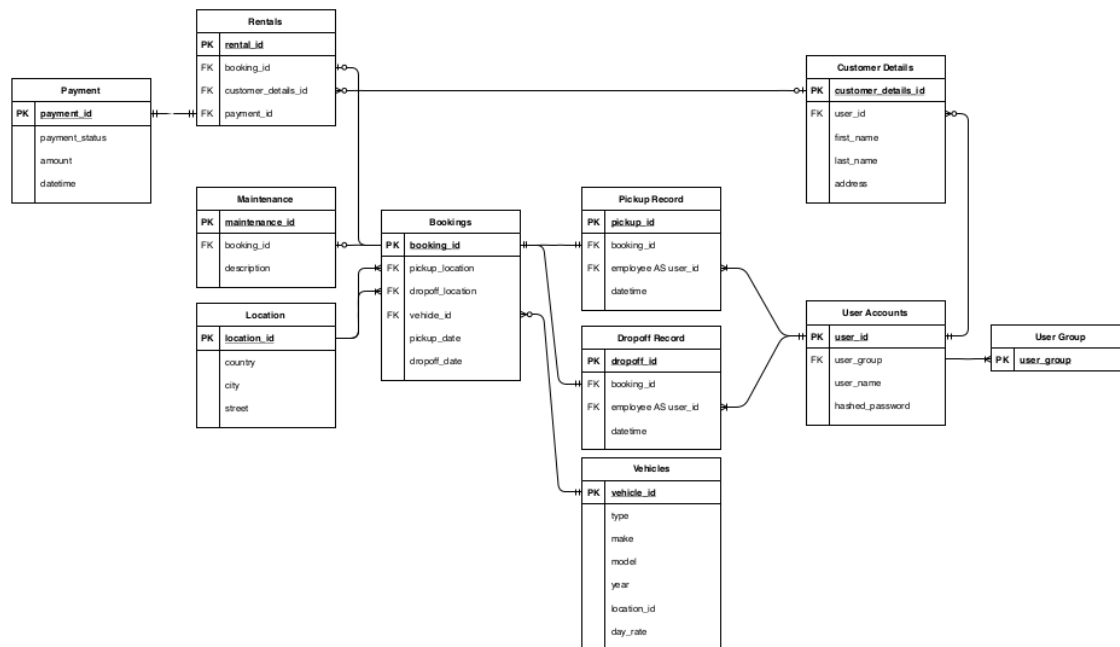


Fig. 4: Entity-Relationship Diagram for the Kartastic database. Primary and Foreign Keys are shown. It is also available in the Documentation folder of the GitHub repository.

## VI. CONTRIBUTIONS

### A. Code for Database Seeding

---

```
/**
 * Generates data for user_accounts table
 *
 * @return true if creation successful
 * @AUTHOR Zack
 */

private static boolean seedUserAccount() throws SQLException {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        stmt = c.createStatement();
        //create known logins
        String sql = "INSERT INTO user_accounts
            (user_group,username,hashed_password) "
            + "VALUES ('customer', 'user', 'password');";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO user_accounts (user_group,username,hashed_password) "
            + "VALUES ('employee', 'employee', 'password');";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO user_accounts (user_group,username,hashed_password) "
            + "VALUES ('manager', 'admin', 'password');";
        stmt.executeUpdate(sql);
        //create unknown other accounts
        Faker faker = new Faker();
        for (int i = 0; i < NUMBER_OF_USERS; i++) {
            sql = "INSERT INTO user_accounts (user_group,username,hashed_password) "
                + "VALUES ('customer', ' " + faker.name().username().replaceAll("'", "")
                + "' , ' " + faker.crypto().sha256() + "');";
            stmt.executeUpdate(sql);
        }
        stmt.close();
        c.close();
        System.out.println("user_accounts - Table seeded successfully");

        return true;
    } catch (Exception e) {
        stmt.close();
        c.close();
        e.printStackTrace();
        return false;
    }
}

/**
 * @return true if creation successful
 * @author Zack
 */

private static boolean createUserAccounts() throws SQLException {
    Statement stmt = null;
    Connection c = null;
```

```

try {
    c = DatabaseHandler.connectDatabase();
    stmt = c.createStatement();
    String sql = "CREATE TABLE user_accounts" +
        "(user_id      INT          GENERATED ALWAYS AS IDENTITY PRIMARY KEY\n" +
        "   NOT NULL," +
        "   user_group   VARCHAR(50) NOT NULL, " +
        "   username     VARCHAR(50) NOT NULL, " +
        "   hashed_password VARCHAR(256) NOT NULL, " +
        "   CONSTRAINT FK_USER_GROUP " +
        "   FOREIGN KEY(user_group) " +
        "   REFERENCES user_group(user_group))";
    stmt.executeUpdate(sql);
    stmt.close();
    c.close();
    System.out.println("user_accounts Table - created successfully");
    return true;
} catch (Exception e) {
    stmt.close();
    c.close();
    e.printStackTrace();
    return false;
}
}

/**
 * Generates data for user_group table
 *
 * @return true if seeding is successful
 * @author Zack
 */
private static boolean seedUserGroup() throws SQLException {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        stmt = c.createStatement();
        //add defined user_groups
        String sql = "INSERT INTO user_group (user_group) "
            + "VALUES ('customer') ";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO user_group (user_group) "
            + "VALUES ('employee') ";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO user_group (user_group) "
            + "VALUES ('manager') ";
        stmt.executeUpdate(sql);
        stmt.close();
        c.close();
        System.out.println("user_group Table - seeded successfully");

        return true;
    } catch (Exception e) {
        stmt.close();
        c.close();
    }
}

```

```

        e.printStackTrace();
        return false;
    }
}

/**
 * @return true if creation successful
 * @author zack
 */
private static boolean createUserGroup() throws SQLException {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        stmt = c.createStatement();
        String sql = "CREATE TABLE user_group " +
            "(user_group VARCHAR(50) PRIMARY KEY NOT NULL)";
        stmt.executeUpdate(sql);
        stmt.close();
        c.close();
        System.out.println("user_group Table - created successfully");
        return true;
    } catch (Exception e) {
        stmt.close();
        c.close();
        e.printStackTrace();
        return false;
    }
}

/**
 * Generates data for customer_details table
 *
 * @return true if creation successful
 * @author zack
 */
private static boolean seedCustomerDetails() throws SQLException {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        stmt = c.createStatement();
        Faker faker = new Faker();
        //NOT ALL CUSTOMER USERS HAVE THEIR DETAILS ON THE SYSTEM BUT USER SHOULD NO
        //HAVE THEIR DETAILS MORE THAN ONCE
        for (int i = 0; i < NUMBER_OF_USERS * RATIO_OF_DETAILS; i++) {
            String sql = "INSERT INTO customer_details (user_id, first_name,
                last_name, address) "
                + "VALUES ((SELECT user_id FROM user_accounts WHERE user_group =
                    'customer' " +
                    " AND user_id NOT IN (select user_id FROM customer_details) ORDER
                    BY RANDOM() LIMIT 1)," +
                    " ' " + faker.name().firstName().replaceAll("'", "") + "', " +
                    " ' " + faker.name().lastName().replaceAll("'", "") + "', " +
                    " ' " + faker.address().streetAddress().replaceAll("'", "") + "');"
        }
    }
}

```

```

        stmt.executeUpdate(sql);
    }
    stmt.close();
    c.close();
    System.out.println("customer_details Table - seeded successfully");
    return true;
} catch (Exception e) {
    stmt.close();
    c.close();
    e.printStackTrace();
    return false;
}
}

/**
 * @return true if creation successful
 * @author zack
 */
private static boolean createCustomerDetails() throws SQLException {
    Statement stmt = null;
    Connection c = null;
    try {
        c = DatabaseHandler.connectDatabase();
        stmt = c.createStatement();
        String sql = "CREATE TABLE customer_details" +
            "(customer_details_id INT GENERATED ALWAYS AS IDENTITY PRIMARY" +
            " KEY NOT NULL," +
            " user_id INT NOT NULL, " +
            " first_name VARCHAR(50) NOT NULL, " +
            " last_name VARCHAR(256) NOT NULL, " +
            " address VARCHAR(256) NOT NULL, " +
            " CONSTRAINT FK_USER_ID " +
            " FOREIGN KEY(user_id) " +
            " REFERENCES user_accounts(user_id) " +
            " ON UPDATE CASCADE " +
            " ON DELETE CASCADE)";
        stmt.executeUpdate(sql);
        stmt.close();
        c.close();
        System.out.println("customer_details Table - created successfully");
        return true;
    } catch (Exception e) {
        stmt.close();
        c.close();
        e.printStackTrace();
        return false;
    }
}
}

```

---

## B. Code for Adding Table Formatting

---

```

/**

```

```

    * Chooses the largest width from either the header string or the data string (to
      avoid overlapping)
    *
    * @param rows
    * @param paddingSize
    * @return
    * @author Zack
    */
private static int[] chooseColumnWidth(ArrayList<ArrayList<String>> rows, int
paddingSize) {
    int arraySize = rows.get(0).size();

    int[] columnWidthsFormatted = new int[arraySize];
    for (ArrayList<String> row : rows) {
        for (int i = 0; i < arraySize; i++) {
            columnWidthsFormatted[i] = Math.max(columnWidthsFormatted[i],
                row.get(i).length() + paddingSize);
        }
    }

    return columnWidthsFormatted;
}

/**
 * Print a border line of the table
 *
 * @param columnWidths
 * @author Zack
 */
private static String printTableLine(int[] columnWidths) {
    String tableLineEdge = "+";
    String tableLineMiddle = "-";
    StringBuilder line = new StringBuilder();
    line.append(String.format("%s", tableLineEdge));
    for (int i = 0; i < columnWidths.length; i++) {
        for (int j = 0; j < columnWidths[i]; j++) {
            line.append(String.format("%s", tableLineMiddle));
        }
        line.append(String.format("%s", tableLineEdge));
    }
    line.append(String.format("%n"));
    return line.toString();
}

/**
 * Prints entire row of the table data (formatted)
 *
 * @param rows
 * @param columnWidths
 * @author Zack
 */
private static String printTableRow(ArrayList<String> rows, int[] columnWidths) {
    String tableMiddleEdge = "|";
    int arraySize = rows.size();
    StringBuilder line = new StringBuilder();

```

```

        for (int i = 0; i < arraySize; i++) {
            // (columnWidths[i] - 2) to account for the table edges
            line.append(String.format("%s %- " + (columnWidths[i] - 2) + "s ",
                tableMiddleEdge, rows.get(i)));
        }
        line.append(String.format("%s\n", tableMiddleEdge));
        return line.toString();
    }

    /**
     * Function to call to display entire table
     *
     * @param rows
     * @param paddingSize
     * @author Zack
     */
    public static void displayTable(ArrayList<ArrayList<String>> rows, int
        paddingSize) {
        System.out.println(createTable(rows, paddingSize));
    }

    /**
     * Function to call to create entire table string
     *
     * @param rows
     * @param paddingSize
     * @author Zack
     */
    protected static String createTable(ArrayList<ArrayList<String>> rows, int
        paddingSize) {
        ArrayList<String> names = rows.get(0);
        int[] columnWidths = chooseColumnWidth(rows, paddingSize);
        StringBuilder table = new StringBuilder();
        table.append(printTableLine(columnWidths));
        table.append(printTableRow(rows.get(0), columnWidths));
        table.append(printTableLine(columnWidths));
        for (int i = 1; i < rows.size(); i++) {
            table.append(printTableRow(rows.get(i), columnWidths));
        }
        table.append(printTableLine(columnWidths));
        return table.toString();
    }
}

```

---

### C. Code for Adding Viewing Bookings Functionality

---

```

import java.util.ArrayList;

public class ViewCustomerBookings {
    private static final String[] VIEW_BOOKING_DESCRIPTION = new String[]{"Show this
        menu", "View all bookings", "View current and future bookings", "View past
        bookings"};
    private static final String[] VIEW_BOOKING_COMMANDS = new
        String[]{"HELP", "ALL", "CURRENT", "PAST"};
}

```

```

/**
 * Menu for viewing customer bookings
 * @return true
 * @author Zack
 */

public static boolean customerSelectBookingType() {
    System.out.println(customerViewBookingDetailedOptions());
    boolean back = false;
    while (true) {
        if(back) System.out.println(customerViewBookingDetailedOptions());
        String input = KartasticUtility.getStringMenuInput(VIEW_BOOKING_COMMANDS,
            true);
        switch (input) {
            case "ALL": case "CURRENT": case "PAST":
                customerViewBookings(input);
                break;
            case "BACK":
                // Go back to previous menu
                return true;
            case "HELP":
                System.out.println(customerViewBookingDetailedOptions());
                back = false;
                break;
        }
    }
}

/**
 * Displays all existing bookings (excluding past bookings)
 * @author Zack
 */
public static void customerViewBookings(String input) {
    System.out.printf("%nYour existing bookings are:%n%n");

    String whereQuery = "u.user_id = CAST(? AS int)";

    switch (input) {
        case "ALL":
            break;
        case "CURRENT":
            whereQuery = whereQuery + " AND (pickup_date >= NOW())";
            break;
        case "PAST":
            whereQuery = whereQuery + " AND (pickup_date < NOW())";
            break;
    }

    ArrayList<ArrayList<String>> queryViewExistingBookings =
        DatabaseHandler.getData("b.booking_id, r.rental_id, cd.customer_details_id,
            u.user_id, " +
            "v.make || ' ' || v.model || ' ' || v.year AS vehicle, " +
            "ll.street || ', ' || ll.city || ', ' || ll.country AS
            pickup_location, b.pickup_date, " +

```

```

        "l2.street || ', ' || l2.city || ', ' || l2.country AS
        dropoff_location, b.dropoff_date",
    "booking b " +
        "INNER JOIN location AS l1 ON (b.pickup_location = l1.location_id)
        " +
        "INNER JOIN location AS l2 ON (b.dropoff_location = l2.location_id)
        " +
        "INNER JOIN vehicles AS v ON (b.vehicle_id = v.vehicle_id) " +
        "INNER JOIN rental as r ON (b.booking_id = r.booking_id) " +
        "INNER JOIN customer_details cd ON (r.customer_details_id =
        cd.customer_details_id)" +
        // "INNER JOIN user_accounts u ON (cd.user_id = u.user_id)", 4);
        "INNER JOIN user_accounts u ON (cd.user_id = u.user_id)",
    whereQuery, new String[]{String.valueOf(KartasticUtility.currentUserID)});

    if (queryViewExistingBookings.size() <= 1) {
        System.out.printf("----- No bookings found! -----%n%n");
    }
    else {
        KartasticUtility.displayTable(queryViewExistingBookings, 4);
    }
}

/**
 * Prints all options from ViewCustomerBookings class
 *
 * @author Zack
 */
protected static String customerViewBookingDetailedOptions() {
    return
        KartasticUtility.createDetailedOptions(VIEW_BOOKING_DESCRIPTION, VIEW_BOOKING_COMMAND);
}
}

```

---

# 2012480\_Zack\_Costello\_Group\_Report

---

## GRADEMARK REPORT

---

### FINAL GRADE

77 / 100

### GENERAL COMMENTS

#### Instructor

**Software design evaluation (30): 25**

**Database evaluation and implementation (30): 23**

**Coding and Testing (30): 23**

**Reflection (10): 6**

A good, cohesive report which has clearly been well-structure and considered. However, a summary section would have provided clarity, in which you could have outlined your more personal thoughts and conclusions.

The report reads well, but ends a little too abruptly.

A well-written and fully featured solution to the problem. The code is clean, though would benefit from a few more comments in places. The structure of the code is logical and well-considered.

The vehicle class appears to be well tested, but the remaining code lacks coverage.

Clear representation of the database requirements represented through the ERD. Be careful to test for data duplication, there's some rationalisation possible around rentals/payments. Otherwise a good solution.

PAGE 2

---

PAGE 3

---

PAGE 4

---

PAGE 5

---

PAGE 6

---

PAGE 7

---

PAGE 8

---

PAGE 9

---

PAGE 10

---

PAGE 11

---

PAGE 12

---

PAGE 13

---

PAGE 14

---