COMSATS UNIVERSITY ISLAMABAD, DHAMTHOR CAMPUS

SOFTWARE DESIGN AND ARCHITECTURE

PROJECT

CAR RENTAL APP

Rehman Ghani

Syed Muhammad Usman

Zain Ul Abideen

Version : 1.0

Lab : 3

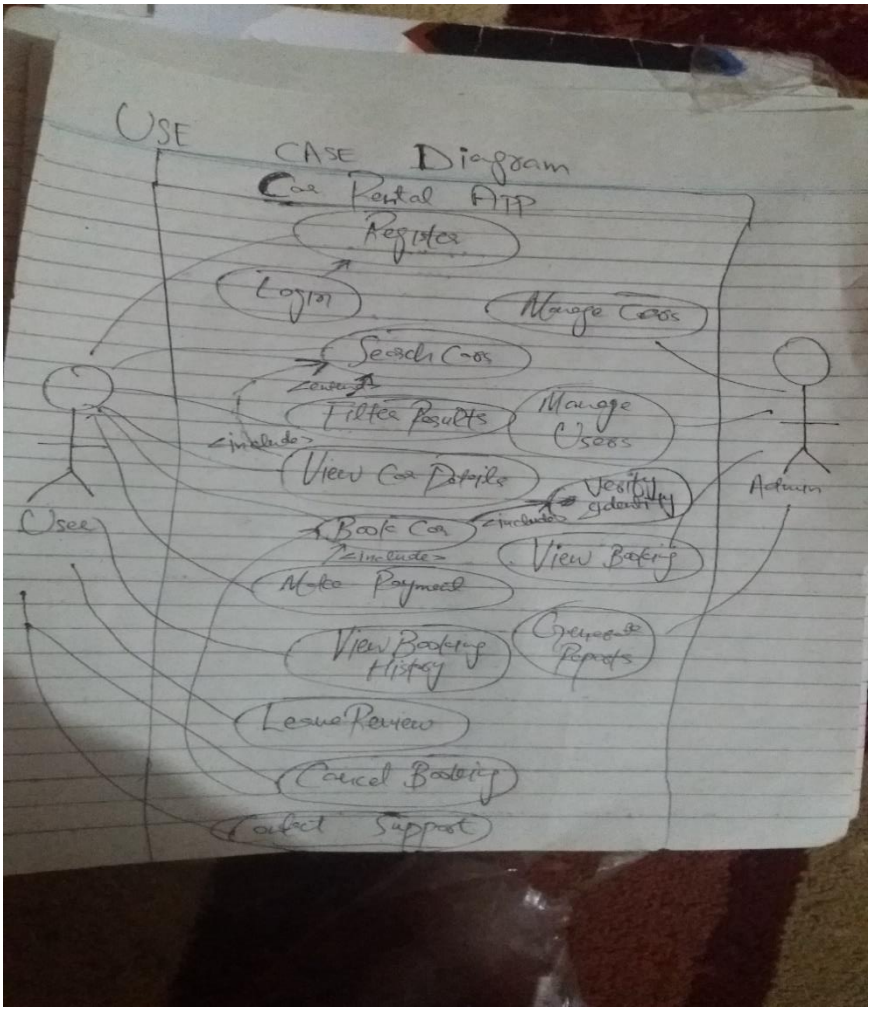
Submitted To: Mukhtiar Zamin

DEPARTEMENT OF SOFTWARE ENGINEERING

CAR RENTAL APP:

A car rental app, which can enhance user experience and streamline the rental process for both customers and operators. Users can create an account and log in using email or phone number. Users can search for available cars based on date, time, and car type. Users can filter results by price range, car brand, type (SUV, sedan, etc.), fuel type, and features. Detailed descriptions of each car, including make, model, year, mileage, and features. Users can select a car, choose rental dates, and reserve the vehicle. Immediate booking confirmation via email or app notification. Options for one-way rentals, hourly, daily, or weekly rates. Support for cash payment and bank transfers. Implementation of secure payment gateways to protect user information. Clear display of rental costs, insurance fees, and taxes. Assistance in case of car breakdowns or accidents. Users can review and sign rental agreements electronically. Admins can add, update, and remove vehicles from the inventory. Users can rate and review their rental experience and vehicles.

USE CASE DIAGRAM:



FULLY DRESSED USE CASE: VIEW BOOKING HISTORY

Use Case Name: View Booking History
This use case describes how a **user** (customer) can view their past and future bookings for car rentals using a web or mobile application. The goal of this use case is to provide the user with an intuitive way to access, review, and manage their booking history. The system must enable the user to not only see the status and details of their previous bookings but also interact with upcoming bookings, such as reviewing rental agreements or potentially canceling the reservation.

Actors Involved:

1. **Primary Actor:**
    - **User (Customer):** The user is the individual who has booked a car rental and wishes to view or manage their booking history. This could involve viewing completed bookings for record-keeping or managing upcoming bookings for future rentals.
2. **Secondary Actors:**
    - **Administrator:** The admin is an indirect actor who may need access to the user's booking history for operational purposes, such as resolving customer issues, managing rental availability, or generating reports for fleet management.
    - **System:** The system represents the application that manages the user's bookings and is responsible for storing, retrieving, and displaying booking data. It also handles payment processing and interactions with external services, such as notifications and payment gateways.

Stakeholders and Interests:

1. **User (Customer):**
    - Wants to view their past and upcoming car bookings, including rental dates, car details, total rental costs, and the booking status (completed, upcoming, or canceled).
    - May need to review rental agreements, payment records, and other details related to past or current bookings for personal or business reasons.
    - Wants a smooth and user-friendly experience when interacting with their booking history.
2. **Car Rental Company:**

- Wants to offer a seamless customer experience, ensuring users can access their booking history easily, leading to greater customer satisfaction and fewer customer support inquiries.
- Needs to keep track of completed and ongoing rentals to manage its fleet efficiently and avoid conflicts in car availability.

3. **Administrator:**
   - May need to access user booking histories for various reasons, including resolving disputes, supporting customer service queries, handling cancellations, or making adjustments to rental agreements.
   - Requires access to historical data for generating reports, auditing booking trends, and understanding rental patterns.

Pre-conditions:

1. **User is authenticated (logged in):**
   Before accessing the booking history, the user must be authenticated by the system. This ensures secure access to personal booking data. The system needs to verify the user's credentials (e.g., username and password or other methods like biometric login or two-factor authentication) before allowing access to any sensitive data, such as booking records.
2. **User has made at least one past or upcoming booking:**
   For this use case to be relevant, the user must have at least one booking recorded in the system, either a completed booking (from the past) or an upcoming rental. If the user has no bookings, the system will return a message such as "No bookings found."
3. **The system has up-to-date booking data stored in the database:**
   The system must maintain an accurate and real-time record of all bookings, ensuring that users see the latest information on their bookings, including rental dates, car availability, and any changes (e.g., cancellations, modifications). The data in the database should be synced and accurate.
4. **The application is functional and has an active connection to the database:**
   If the system is cloud-based, the user's device must have an active internet connection to communicate with the server. The application should be functioning without any service outages or database downtimes to ensure smooth retrieval of booking data.
5. **User knows how to access the booking history section:**
   The user must navigate to the booking history section via an intuitive UI/UX. The system

typically provides this access in a section like "My Bookings" or "Booking History," which is easily discoverable in the app's navigation.

6. **Booking information includes relevant details:**
   The booking records in the system must include all essential information for the user to review. This includes car details, rental dates, total costs, payment statuses, and booking terms. Any missing information can lead to a degraded user experience.

Post-conditions:

1. **User successfully views their booking history:**
   The user is able to access a list of all past and upcoming bookings, including key information such as car model, rental period (start and end dates), total cost, booking status (completed, upcoming, or canceled), and payment confirmation.
2. **Booking details are displayed accurately and without errors:**
   The system accurately retrieves and presents the user's booking history. The information displayed must match what is stored in the system, including payment details, car availability, and rental terms. Any errors in displaying this information (e.g., outdated bookings, incorrect data) could lead to customer dissatisfaction or confusion.
3. **User is able to perform further actions:**
   From the booking history, the user has several options for interacting with their bookings:
   - **View more detailed information:** The user can click on individual bookings to see additional details, such as rental agreements, payment records, and any extra charges.
   - **Cancel upcoming bookings (if applicable):** If the booking is still upcoming and within the cancellation window, the user should have the option to cancel the reservation and potentially receive a refund or credit, depending on company policy.
   - **Download documents:** The user may have the option to download rental agreements, payment receipts, or invoices for their records (useful for business travelers).
4. **User receives notifications (if applicable):**
   The system can optionally send a notification to the user regarding their upcoming bookings, changes to bookings, or reminders (e.g., a notification reminding the user about an upcoming rental one day before it starts).
5. **System logs the action for analytics and future reference:**
   The system logs the user's actions, such as when and how often they access their booking

history. These logs can be used for future auditing, customer service inquiries, or business analytics (e.g., to track how frequently users interact with their booking history).

6. **Error-free data retrieval:**
   The system retrieves and displays data without issues, and the process concludes without any errors. In case of an error (e.g., database retrieval failure), an appropriate message is shown to the user, and the system may suggest they retry or contact support.

Main Success Scenario (Basic Flow):

This section details the steps that occur when the user successfully views their booking history without any errors or interruptions.

1. **The user logs into the system and navigates to the "Booking History" section:**
   The user accesses the app or website, logs into their account using their credentials, and selects the "Booking History" option from the navigation menu (e.g., from a user dashboard or profile section).
2. **System fetches the booking history from the database:**
   The system queries the database for all bookings associated with the user's account. The system checks both completed bookings (past rentals) and upcoming rentals that are still active. The system retrieves all relevant data, such as:
   - Car model and details (e.g., make, model, year)
   - Rental start and end dates
   - Total rental cost (including any taxes or additional fees)
   - Booking status (e.g., Completed, Ongoing, Upcoming, or Canceled)
   - Payment status (e.g., Paid, Unpaid, Pending)
3. **System displays a list of past and upcoming bookings:**
   The system presents a list of the user's bookings in a clear and concise manner. Each booking is typically displayed as a summary, with essential details such as the car model, rental period, and total cost. For each booking, there will also be an indication of whether the booking is completed or upcoming, along with any status updates (e.g., pending payment, canceled, or refunded).
4. **User selects a specific booking to view more details:**
   The user clicks on or taps a booking to view additional details. For example, they might want to review a rental agreement, see the payment breakdown, or view specific details about the car they rented (e.g., make, model, features, etc.).

5. **System retrieves and displays full booking details:**
   Once the user selects a booking, the system retrieves and displays the full details of the selected booking. This page might include:
   - Rental agreement (e.g., terms and conditions, responsibilities)
   - Payment details (e.g., breakdown of rental cost, taxes, additional fees)
   - Car details (e.g., model, make, color, engine specifications)
   - Special instructions (e.g., where to pick up or drop off the car)
   - Cancellation policy (for upcoming bookings)
   - Total charges and payment confirmation
6. **User reviews the booking details and may take further action:**
   The user reviews the detailed booking information. Depending on the status of the booking, they may:
   - **Cancel an upcoming booking:** If the booking is still in the future and within the allowed cancellation window, the user may cancel the booking.
   - **Modify upcoming booking:** In some systems, the user may have the option to change the booking, such as adjusting rental dates, upgrading/downgrading the car, or adding extras (e.g., insurance).
   - **Download documents:** The user may download or save rental agreements or receipts for personal or business use.
   - **Contact customer support:** If there are discrepancies or issues with the booking, the user can contact customer service for assistance.
7. **User can return to the booking history overview or exit:**
   After reviewing a specific booking, the user can either return to the list of bookings to view other entries or exit the booking history section. The system allows the user to navigate freely within the app or website.

Alternative Flows:

1. **Alternative Flow 1: No Booking History Available**
   - If the user has never made a booking or the system cannot find any records of past or upcoming bookings:
     1. The system displays a message such as “No bookings found” or “You have no past or upcoming bookings.”

2. The user is returned to the main menu or dashboard, where they can explore other options (e.g., browsing available cars or making a new booking).
3. The use case ends here, as there are no bookings to display.

2. **Alternative Flow 2: System Error in Fetching Data**
    - If the system encounters an error (e.g., database connection failure) while trying to retrieve the user's booking history:
        1. The system displays an error message indicating that the booking data cannot be retrieved at the moment (e.g., "We're experiencing technical issues. Please try again later").
        2. The system may log the error for further investigation by technical support.
        3. The user is given the option to retry or contact customer support if the issue persists.
        4. If the system remains unavailable, the use case ends with an error message.
3. **Alternative Flow 3: User Cancels Viewing Process**
    - If the user decides not to proceed with viewing their booking history (e.g., they click the back button or navigate away from the booking history section):
        1. The system cancels the current operation and returns the user to the previous screen (e.g., the main dashboard or profile section).
        2. The system stops fetching the booking data, and the use case ends without displaying the booking history.

Trigger:

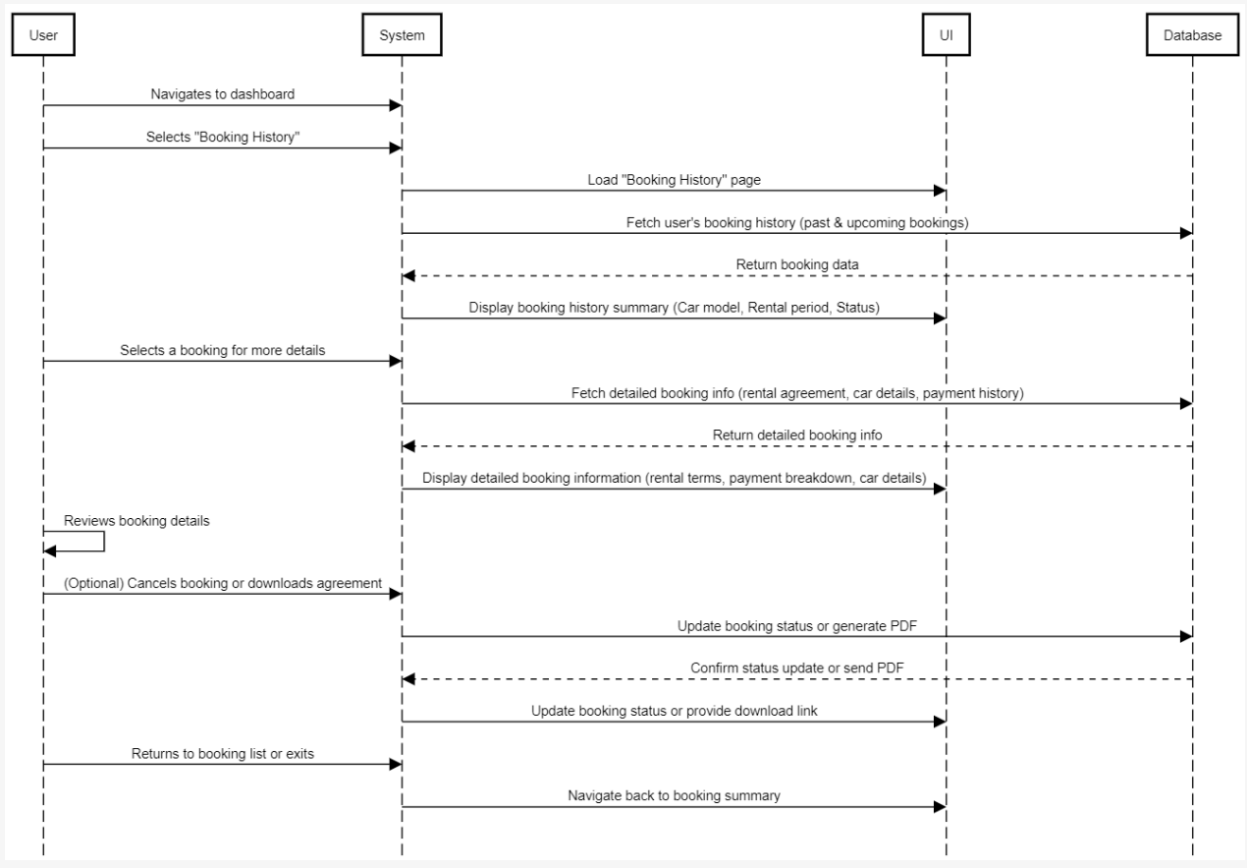
The use case is triggered when the user selects the "Booking History" option from the application menu. This action typically occurs when the user wants to review their past or upcoming bookings for reference, record-keeping, or management purposes.

SEQUENCE OF EVENTS FOR "VIEW BOOKING HISTORY" USE CASE:

This section outlines the series of steps that take place between the user and the system when the user chooses to view their booking history.

1. **User Navigates to Dashboard**:
   The user begins by navigating to the dashboard, where various options and sections are available for interaction.
2. **User Selects "Booking History"**:
   From the dashboard or menu, the user selects the "Booking History" option to view their past and upcoming bookings.
3. **System Loads the Booking History Page**:
   The system responds by loading the "Booking History" page, where it prepares to display the user's booking information.
4. **System Fetches Booking History from the Database**:
   The system queries the backend database to retrieve all bookings associated with the logged-in user. This includes both completed (past) bookings and upcoming (future) bookings that have not yet started.
5. **System Displays Booking History Summary**:
   Once the booking data is retrieved, the system presents a summary of each booking to the user. Key details such as the car model, rental period, and booking status (e.g., Completed, Upcoming, Canceled) are shown in an organized format.
6. **User Selects a Specific Booking for More Details**:
   The user can click on any specific booking from the list to view more detailed information.
7. **System Fetches Detailed Booking Information**:
   Upon selection, the system sends a request to the database to retrieve the full details of the selected booking. This includes the rental agreement, car details, payment history, and any applicable terms.
8. **System Displays Detailed Booking Information**:
   The system displays all the detailed information to the user. This includes the rental agreement, car specifications, payment breakdown (rental cost, taxes, fees), and the cancellation policy (if applicable).
9. **User Reviews Booking Details**:
   The user reviews the detailed booking information. At this stage, they can either take further action or simply check the details.
10. **User (Optional) Cancels Booking or Downloads Rental Agreement**:
    If the user wishes, they can either cancel an upcoming booking or download a copy of the rental agreement. The system will then update the booking status in the database or generate a PDF file of the rental agreement for download.
11. **System Updates Booking Status or Provides PDF Download**:
    The system updates the booking status in the database if the user cancels a booking and reflects this change in the booking summary. Alternatively, the system generates and provides a link to download the rental agreement as a PDF.

12. **User Returns to Booking List or Exits**:
    After reviewing the details or taking any additional action, the user can return to the booking history summary page or exit the booking section altogether.



COMMUNICATION DIAGRAM:

This communication diagram illustrates how various system components interact during the **"View Booking History"** use case, focusing on key user actions such as viewing booking history, canceling a booking, and downloading a rental agreement. The diagram shows how different parts of the system, including the **UserInterface**, **BookingHistoryController**, **Booking Component**, and **NotificationService**, communicate to fulfill user requests.

Key Components and Their Roles:

1. **User**:
    - The user interacts with the application interface to perform actions like viewing booking history, canceling a booking, or downloading a rental agreement.

2. **View Bookings (User Interface)**:
    - This is the app's user interface (UI) that directly interacts with the user, managing all the booking-related inputs and outputs.
    - It receives user commands such as viewing bookings, canceling bookings, or downloading agreements, and forwards these requests to the **BookingHistoryController**.
    - After receiving data from the backend, it displays booking history or details to the user.
3. **BookingHistoryController**:
    - This is the central controller that mediates between the **View Bookings** UI and the system's backend logic.
    - It processes requests coming from the UI and communicates with the **Booking Component** to fetch or update booking-related data.
    - It handles three main requests:
        - **View Booking History**: Fetches a list of bookings for the user.
        - **Cancel Booking**: Sends a request to cancel a specific booking.
        - **Download Agreement**: Sends a request to retrieve a rental agreement.
4. **Booking Component**:
    - This component stores and manages all booking-related data. It is responsible for retrieving or updating the booking information as needed.
    - It provides:
        - **Retrieve Data**: Retrieves booking history or specific booking details based on the user's request.
        - **Update Booking Status**: Handles booking cancellations and updates the status of the booking in the system.
        - **Retrieve Agreement**: Fetches the rental agreement for download by the user.
    - After retrieving the data or updating the status, it sends the required information back to the **BookingHistoryController**.
5. **NotificationService**:

- This component is responsible for sending notifications to the user about important booking-related events.
- It performs three main tasks:
  - **Send Confirmation**: Sends a confirmation notification when a booking is successfully made or updated.
  - **Send Reminder**: Sends reminders about upcoming bookings to the user.
  - **Send Cancellation Notice**: Notifies the user when a booking is canceled.

Detailed Flow of Communication:

1. Viewing Booking History:

- **User Action**: The user initiates a request to view their booking history.
- **View Bookings UI**: The request is forwarded to the **BookingHistoryController**.
- **BookingHistoryController**: It processes the request and asks the **Booking Component** to fetch the user's bookings by calling getBookings(userID).
- **Booking Component**: It retrieves the list of bookings for the user from the database and returns the data to the **BookingHistoryController**.
- **BookingHistoryController**: Once the data is received, it sends the bookings information back to the **View Bookings UI**.
- **View Bookings UI**: The user interface then displays the list of bookings for the user to view.

2. Canceling a Booking:

- **User Action**: The user selects a booking and requests to cancel it.
- **View Bookings UI**: This request is forwarded to the **BookingHistoryController**.
- **BookingHistoryController**: It processes the request and sends a command to the **Booking Component** to update the booking status by calling updateBookingStatus(bookingID).
- **Booking Component**: The system updates the booking status to "canceled" in the database and confirms the change back to the **BookingHistoryController**.
- **BookingHistoryController**: After confirming the update, it notifies the **NotificationService**.

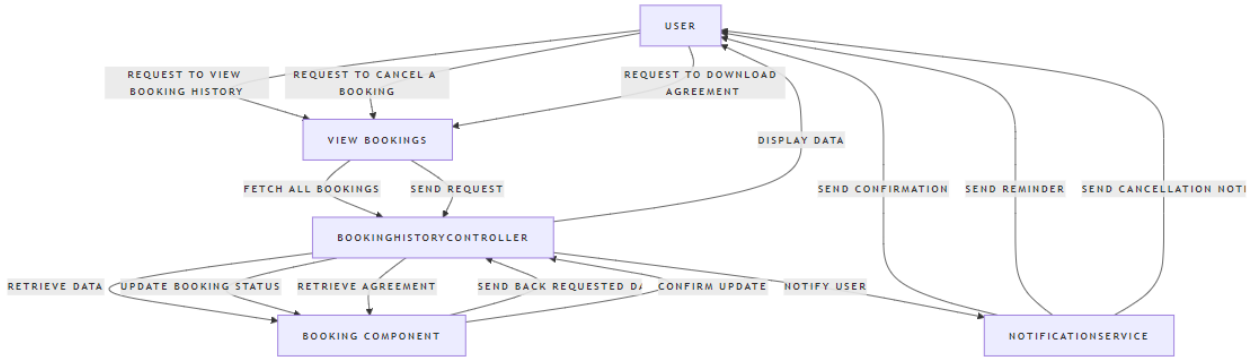
- **NotificationService**: The notification service sends a **Cancellation Notice** to the user, informing them that the booking has been successfully canceled.
- **View Bookings UI**: Finally, the UI is updated to reflect the cancellation, and the user is informed about the status update.

3. Downloading a Rental Agreement:

- **User Action**: The user requests to download the rental agreement for a specific booking.
- **View Bookings UI**: This request is forwarded to the **BookingHistoryController**.
- **BookingHistoryController**: It processes the request and sends a command to the **Booking Component** to fetch the rental agreement by calling fetchAgreement(bookingID).
- **Booking Component**: The system retrieves the rental agreement and sends the file back to the **BookingHistoryController**.
- **BookingHistoryController**: After receiving the rental agreement, it forwards the data back to the **View Bookings UI**.
- **View Bookings UI**: The UI then provides the user with a download option to access the rental agreement file.

4. Sending Notifications:

- The **NotificationService** handles all notifications related to booking activities. It performs the following tasks based on system actions:
    - **Send Confirmation**: When a booking is made or updated, it sends a confirmation message to the user.
    - **Send Reminder**: It sends reminders before the booking's scheduled date.
    - **Send Cancellation Notice**: It notifies the user when a booking is canceled, confirming that the request was successful.



CLASS DIAGRAM:

The diagram represents a **comprehensive class diagram** for the "Book Car" use case in a car rental application, and it adheres to GRASP (General Responsibility Assignment Software Patterns) principles. Below is an explanation of the diagram's elements and relationships:

Classes and GRASP Principles:

1. **UserInterface**:
    - **Responsibility**: Manages user interactions, such as displaying car details, confirming bookings, and showing payment options.
    - **GRASP Principle**: **Controller**, as it coordinates interactions between the user and the business logic.
2. **BookingHistoryController**:
    - **Responsibility**: Acts as a **Controller** that handles all requests related to car booking and manages the flow between the UserInterface, Booking, and other services.
    - **GRASP Principle**: **Controller**, responsible for orchestrating actions like checking availability, processing payments, and sending notifications.
3. **Car**:
    - **Responsibility**: Stores data related to each car, such as model, rental price, and availability. It provides methods to check availability and update the car's status.
    - **GRASP Principle**: **Information Expert**, as it holds all the information needed to determine car availability.
4. **User**:

- **Responsibility**: Represents the customer who is renting the car. Stores personal and payment details.
- **GRASP Principle**: **Information Expert**, as it manages its own data and interactions.

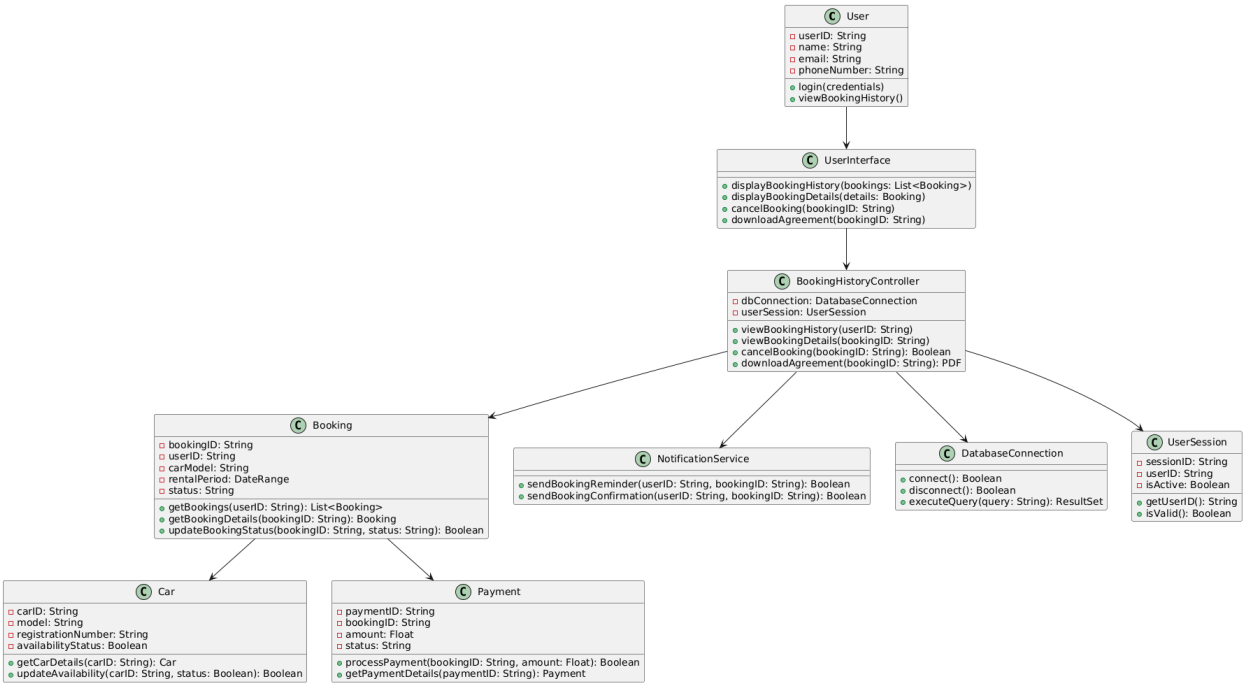
5. **Payment**:
   - **Responsibility**: Manages payment-related tasks, including processing and verifying payments.
   - **GRASP Principle**: **Information Expert**, as it handles all payment-related logic and ensures proper transactions.
6. **Booking**:
   - **Responsibility**: Manages individual car bookings, including details like the user, car, start and end dates, and status.
   - **GRASP Principle**: **Information Expert**, as it knows all the details about a booking and is responsible for updating booking statuses.
7. **NotificationService**:
   - **Responsibility**: Sends notifications, such as booking confirmations and reminders, to the user.
   - **GRASP Principle**: **Low Coupling**, as it works independently to send notifications without direct dependence on the booking logic.
8. **DatabaseConnection** and **UserSession**:
   - **Responsibility**: Manages connections to the database and handles user session details.
   - **GRASP Principle**: **Low Coupling**, as these components abstract the lower-level details of database connections and session management.

Relationships and Associations:

1. **UserInterface - BookingHistoryController**:
   - **Association**: The UserInterface uses the BookingHistoryController to start the booking process.
   - **Multiplicity**: 1-to-1, as a single interface interacts with a single controller.
2. **BookingHistoryController - Car**:
   - **Association**: The controller checks the car's availability and updates its status.

- **Dependency**: The BookingHistoryController depends on the Car class to retrieve and update car information.

3. **BookingHistoryController - User**:
    - **Association**: The controller accesses the user details during the booking process.
    - **Multiplicity**: 1-to-1, as one user is associated with each booking.
4. **BookingHistoryController - Payment**:
    - **Association**: The controller initiates and manages the payment process.
    - **Dependency**: The BookingHistoryController depends on the Payment class to process transactions.
5. **BookingHistoryController - NotificationService**:
    - **Dependency**: The controller sends notifications to the user through the NotificationService after a booking is confirmed.
6. **Car - Booking**:
    - **Association**: Each booking corresponds to a single car being rented.
    - **Multiplicity**: 1-to-1, as one booking corresponds to one car.
7. **User - Booking**:
    - **Aggregation**: A user can have multiple bookings over time, meaning there's a 1-to-many relationship between users and bookings.



GRASP Principles Applied:

1. **Controller (BookingHistoryController)**:
    - The controller class coordinates all tasks and actions between the user interface and other domain logic, ensuring that the use case for booking a car is handled appropriately.
2. **Information Expert (Car, Payment, Booking)**:
    - Each class manages its own data, ensuring high cohesion. For example, the Car class handles availability checks, while the Payment class processes financial transactions.
3. **Low Coupling**:
    - Dependencies are minimized between unrelated classes. The UserInterface doesn't need to know how payments are processed; it delegates that to the BookingHistoryController, which in turn works with the Payment class.
4. **High Cohesion**:
    - Each class has a clearly defined responsibility. For instance, the Car class only manages car-related data, and the NotificationService class is only responsible for sending notifications, ensuring the system remains well-organized and maintainable.

This class diagram, with its focus on separation of concerns, reflects a well-architected system that adheres to the principles of **Low Coupling** and **High Cohesion**, ensuring both flexibility and maintainability.

Conclusion:

The "View Booking History" use case provides users with a seamless way to view and manage their past and upcoming bookings in a car rental system. The system ensures data integrity, offers flexibility for user interactions (such as viewing details or canceling bookings), and handles errors gracefully. By adhering to GRASP principles, the system maintains low coupling, high cohesion, and clear responsibilities for each component, ensuring a robust and scalable design. This detailed elaboration provides a thorough understanding of the user journey, system operations, and technical architecture involved in this use case.