BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

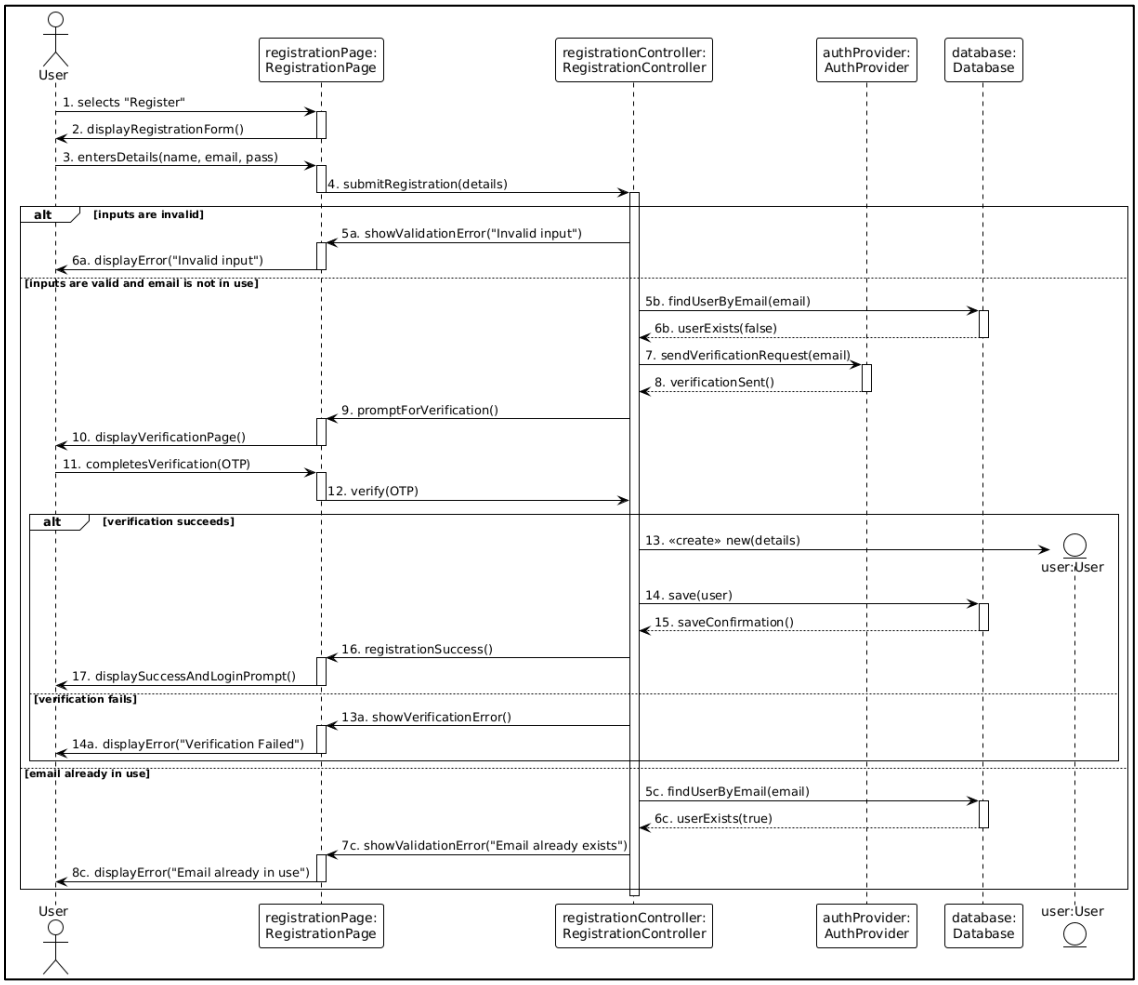
Name:	RAI DEV BRAJESHKUMAR
UID no:	2024301022
Division & Batch:	Division - D & Batch - D
Experiment No:	04

UC-01: Register Account

a. Structured use case specification:

Use Case: Registration
Scenario 1: Successful New User Registration 1. Select "Register" option. 2. Get data for new user (name, email, password). 3. Check validity of user inputs. 4. Check for already registered user via email. 5. Send verification request (e.g., via email link or OTP). 6. Complete verification step. 7. Create a new user account in the system. 8. Update user records in the database. 9. Message "Registration successful". *Alternative 1.1: Invalid Input Provided* At step 3, if inputs are invalid (e.g., weak password, invalid email format). Message "Registration fail: Invalid input provided". *Alternative 1.2: Email Already Exists* At step 4, if the provided email is already registered. Message "Registration fail: Email already in use". *Alternative 1.3: Verification Fails* At step 6, if user provides an incorrect OTP/token or the link expires. Message "Registration fail: Verification failed".

b. Sequence diagram:



c. Method Population in Classes:

RegistrationPage
displayRegistrationForm()
submitRegistration(details)
showValidationError(message)
displayError(message)
promptForVerification()
displayVerificationPage()
completesVerification(OTP)
registrationSuccess()
showVerificationError()
displaySuccessAndLoginPrompt()

RegistrationController
submitRegistration(details)
verify(OTP)
validateInputs()

AuthProvider
sendVerificationRequest(email)

Database
findUserByEmail(email)
save(user)

User
new(details) (Constructor)
Getters/Setters for user attributes...

d. Code snippets:

```
// View Class
public class RegistrationPage {
    private RegistrationController registrationController;

    public RegistrationPage(RegistrationController controller) {
        this.registrationController = controller;
    }

    public void onRegistrationSubmit(UserDetails details) {
        registrationController.submitRegistration(details);
    }

    public void onVerificationSubmit(String otp) {
        registrationController.verify(otp);
    }

    public void displayRegistrationForm() {
        System.out.println("Displaying registration form...");
    }
```

```
    public void displayError(String message) {
        System.out.println("Error: " + message);
    }

    public void showValidationError(String message) {
        // In a real app, this would update the UI to show the error.
        displayError(message);
    }

    public void showVerificationError() {
        displayError("Verification Failed");
    }

    public void displaySuccessAndLoginPrompt() {
        System.out.println("Registration successful! Please log in.");
    }
}

// Controller Class
public class RegistrationController {
    private AuthProvider authProvider;
    private Database database;

    public RegistrationController(AuthProvider authProvider, Database database) {
        this.authProvider = authProvider;
        this.database = database;
    }

    public void submitRegistration(UserDetails details) {
        if (!validateInputs(details)) {
            // In a real app, this would trigger a call back to the view
            System.out.println("Validation failed internally.");
            return;
        }

        if (database.findUserByEmail(details.getEmail()) != null) {
            System.out.println("Email exists check failed internally.");
            return;
        }
```

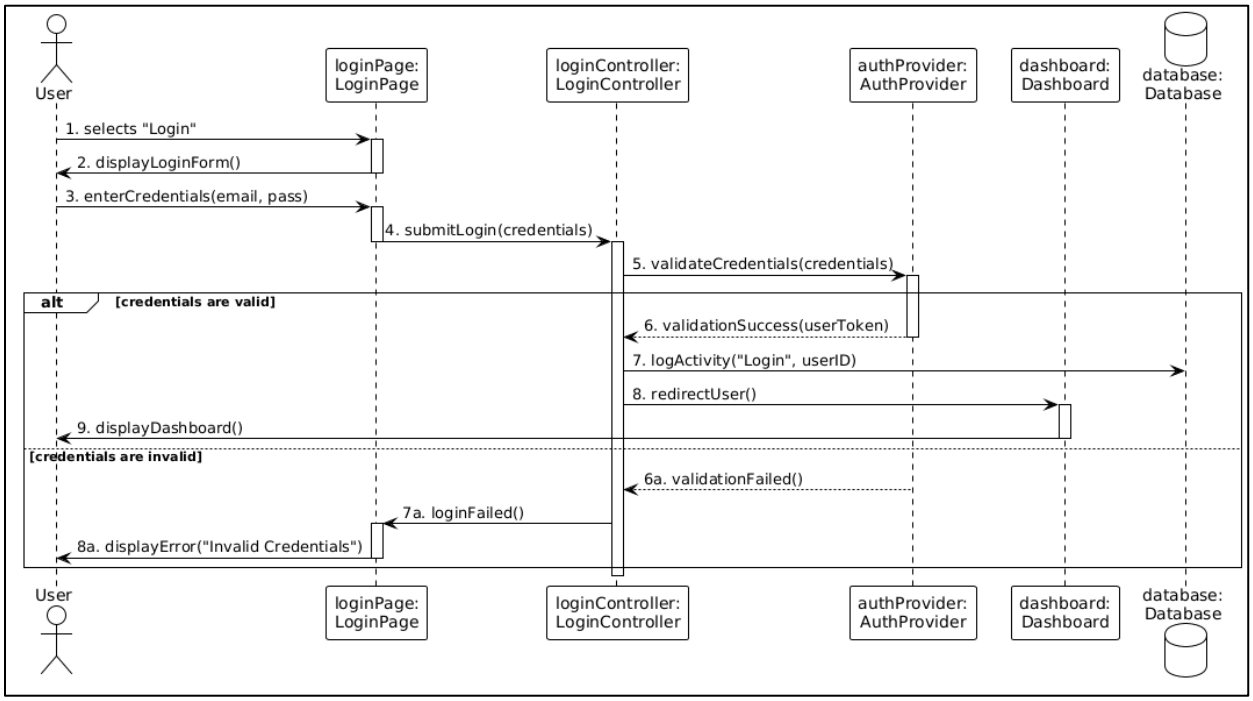
```
        authProvider.sendVerificationRequest(details.getEmail());
        // Logic to handle the verification process would follow...
    }

    public void verify(String otp) {
        // Logic to check OTP validity
        boolean isOtpValid = true; // Placeholder
        if (isOtpValid) {
            // Placeholder for UserDetails that were temporarily stored
            UserDetails details = new UserDetails();
            User newUser = new User(details);
            database.save(newUser);
            // Signal success to the view
        } else {
            // Signal failure to the view
        }
    }

    private boolean validateInputs(UserDetails details) {
        // Logic to validate name, email format, password strength, etc.
        return true; // Placeholder
    }
}

// Entity Class
public class User {
    private String name;
    private String email;
    private String password;

    public User(UserDetails details) {
        this.name = details.getName();
        this.email = details.getEmail();
        this.password = details.getPassword();
    }
    // ... Getters and Setters
}
```

UC-02: Login to Account

a. Structured use case specification:

Use Case: Login
Scenario 1: Successful Login 1. Select "Login" option. 2. Get data for user (email, password). 3. Check for valid and verified user credentials. 4. Log the login activity. 5. Grant access and redirect to the user's dashboard. 6. Message "Login successful". *Alternative 1.1: Invalid Credentials* At step 3, if credentials do not match a registered and verified account. Message "Login fail: Invalid credentials". *Alternative 1.2: Account is Locked* At step 3, if the account is locked due to too many failed attempts. Message "Login fail: Account is locked. Please reset your password".

b. Sequence diagram:



c. Method Population in Classes:

LoginPage
displayLoginForm()
submitLogin(credentials)
loginFailed()
displayError(message)

LoginController
submitLogin(credentials)
redirectUser()

AuthProvider
validateCredentials(credentials)

Dashboard
displayDashboard()

Database
logActivity(action, userID)

d. Code snippets:

```
// View Class
public class LoginPage {
    private LoginController loginController;

    public void onLoginSubmit(Credentials credentials) {
        loginController.submitLogin(credentials);
    }

    public void displayLoginForm() {
        System.out.println("Displaying login form...");
    }

    public void loginFailed() {
        displayError("Invalid Credentials");
    }
```

```
    public void displayError(String message) {
        System.out.println("Login Failed: " + message);
    }
}

// Controller Class
public class LoginController {
    private AuthProvider authProvider;
    private Database database;

    // Constructor for dependency injection
    public LoginController(AuthProvider authProvider, Database database) {
        this.authProvider = authProvider;
        this.database = database;
    }

    public void submitLogin(Credentials credentials) {
        AuthToken token = authProvider.validateCredentials(credentials);

        if (token != null && token.isValid()) {
            database.logActivity("Login", token.getUserId());
            redirectUser();
        } else {
            // In a real app, this would trigger a callback to the LoginPage
            // For example: loginPage.loginFailed();
            System.out.println("Internal check: login failed.");
        }
    }

    private void redirectUser() {
        Dashboard dashboard = new Dashboard();
        dashboard.displayDashboard();
    }
}

// Service Class (Example)
public class AuthProvider {
    public AuthToken validateCredentials(Credentials credentials) {
        // Logic to check credentials against the database.
```

```
        // If successful, create and return a valid AuthToken.
        // Otherwise, return null or an invalid token.
        // Placeholder implementation:
        if ("user@email.com".equals(credentials.getEmail()) &&
"password123".equals(credentials.getPassword())) {
            return new AuthToken("user123", true);
        }
        return new AuthToken(null, false);
    }
}

// A simple data object for the token
public class AuthToken {
    private String userId;
    private boolean valid;

    public AuthToken(String userId, boolean valid) {
        this.userId = userId;
        this.valid = valid;
    }

    public boolean isValid() { return valid; }
    public String getUserId() { return userId; }
}
```

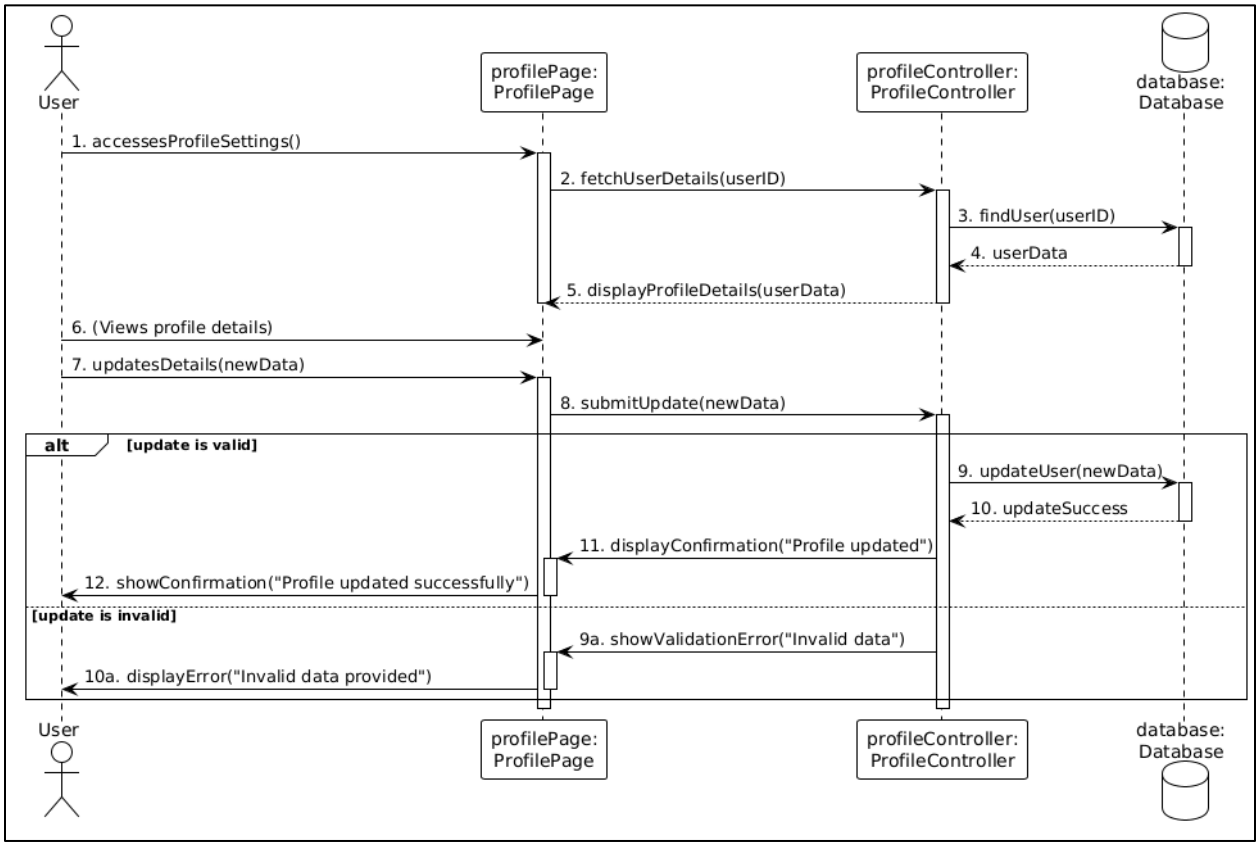
UC-03: Manage Profile

a. Structured use case specification:

Use Case: Manage Profile
Scenario 1: Successful Profile Update 1. Navigate to "Profile" or "Account Settings" page. 2. Get current user data. 3. Display current profile information in an editable form. 4. User updates their information (e.g., name, contact details). 5. Check validity of new data. 6. Update user record in the database. 7. Message "Profile updated successfully". *Alternative 1.1: Invalid Data Provided* At step 5, if the new data is invalid (e.g., invalid phone

number format). Message "Update fail: Invalid data provided". Reject changes and display an error message indicating the required corrections.

b. Sequence diagram:



c. Method Population in Classes:

ProfilePage
fetchUserDetails(userID)
displayProfileDetails(userData)
submitUpdate(newData)
displayConfirmation(message)
showConfirmation(message)
showValidationError(message)
displayError(message)

ProfileController
fetchUserDetails(userID)

submitUpdate(newData)
validateUpdate(data)

Database
findUser(userID)
updateUser(data)

d. Code snippets:

```
// View Class
public class ProfilePage {
    private ProfileController profileController;

    public ProfilePage(ProfileController controller) {
        this.profileController = controller;
    }

    public void onPageLoad(UserID userId) {
        UserData currentUserData = profileController.fetchUserDetails(userId);
        displayProfileDetails(currentUserData);
    }

    public void displayProfileDetails(UserData data) {
        System.out.println("Displaying user profile for: " + data.getName());
    }

    public void onUpdateSubmit(NewData data) {
        profileController.submitUpdate(data);
    }

    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
    }

    public void displayError(String message) {
        System.out.println("Error: " + message);
    }
}

// Controller Class
```

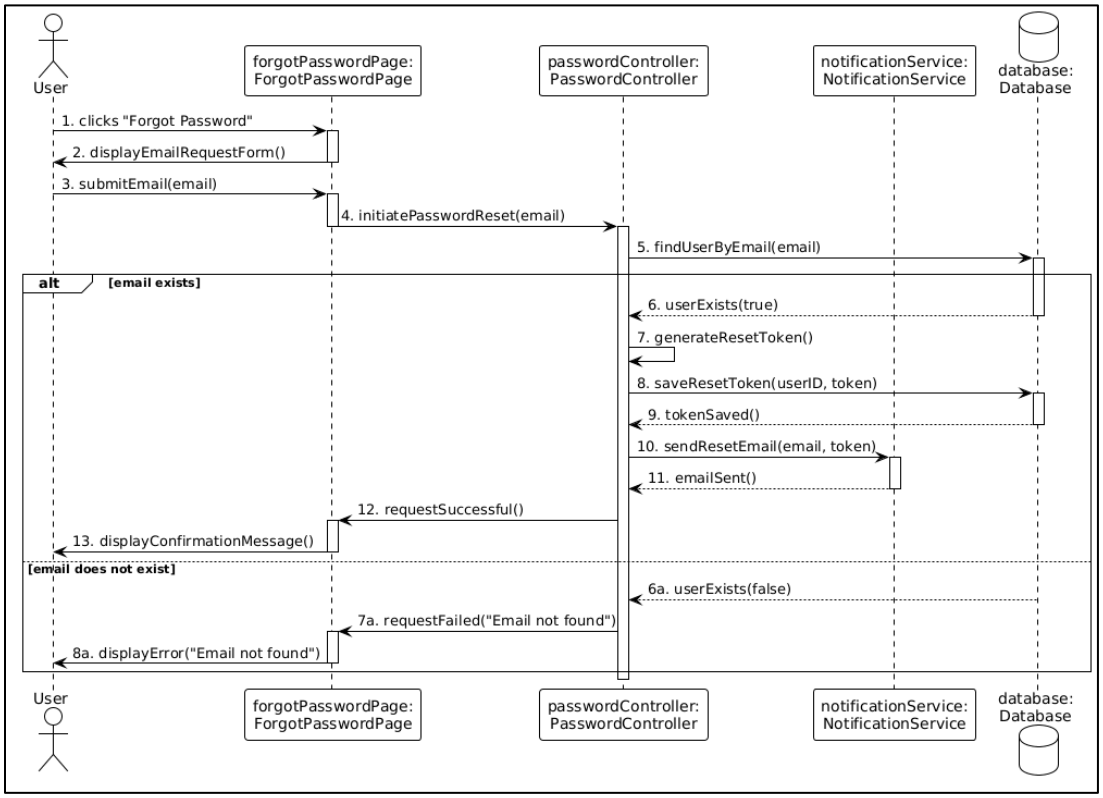
```
public class ProfileController {
    private Database database;

    public ProfileController(Database database) {
        this.database = database;
    }

    public UserData fetchUserDetails(UserID userId) {
        // Fetches user data to be displayed on the profile page
        return database.findUser(userId);
    }

    public void submitUpdate(NewData data) {
        if (validateUpdate(data)) {
            database.updateUser(data);
            // e.g., profilePage.showConfirmation("Profile Updated");
        } else {
            // e.g., profilePage.displayError("Invalid data provided");
        }
    }

    private boolean validateUpdate(NewData data) {
        return true; // Placeholder
    }
}

// Data Transfer Object (DTO) for updated data
public class NewData {
    private UserID userId;
    private String newName;
    private String newContact;

}

public class UserData {
    private String name;
    private String email;
    private String contact;
}
```

UC-04: Forgot Password

a. Structured use case specification:

Use Case: Forgot Password
Scenario 1: Successful Password Reset Link Generation 1. User selects "Forgot Password" option on the login page. 2. System displays a page asking for the user's registered email address. 3. User enters their email address and submits the form. 4. System verifies that the email exists in the database. 5. System generates a unique, secure password reset link/token. 6. System sends the reset link to the user's email address. 7. System displays a confirmation message, "Password reset link has been sent to your email." *Alternative 1.1: Email Not Found* At step 4, if the provided email address is not found in the database. Message "Error: Email address not found."

b. Sequence diagram:



c. Method Population in Classes:

ForgotPasswordPage
displayEmailRequestForm()
submitEmail(email)
requestSuccessful()
displayConfirmationMessage()
requestFailed(message)
displayError(message)

PasswordController
initiatePasswordReset(email)
generateResetToken()

NotificationService
sendResetEmail(email, token)

Database
findUserByEmail(email)
saveResetToken(userID, token)

d. Code snippets:

```
// View Class
public class ForgotPasswordPage {
    private PasswordController passwordController;

    public void displayEmailRequestForm() {
        System.out.println("Please enter your registered email address.");
    }

    public void onSubmitEmail(String email) {
        passwordController.initiatePasswordReset(email);
    }

    public void displayConfirmationMessage() {
        System.out.println("A password reset link has been sent to your email.");
    }
```

```
    public void displayError(String message) {
        System.out.println("Error: " + message);
    }
}

// Controller Class
public class PasswordController {
    private Database database;
    private NotificationService notificationService;

    public PasswordController(Database db, NotificationService notifier) {
        this.database = db;
        this.notificationService = notifier;
    }

    public void initiatePasswordReset(String email) {
        User user = database.findUserByEmail(email);

        if (user != null) {
            String token = generateResetToken();
            database.saveResetToken(user.getId(), token);
            notificationService.sendResetEmail(email, token);
            // In a real app, signal success back to the view
        } else {
            // In a real app, signal failure back to the view
        }
    }

    private String generateResetToken() {
        // Generates a unique, secure, and time-limited token
        return "secureToken12345"; // Placeholder
    }
}

// Service for sending notifications (e.g., email)
public class NotificationService {
    public void sendResetEmail(String email, String token) {
        System.out.println("Sending reset email to " + email + " with token: " + token);
```

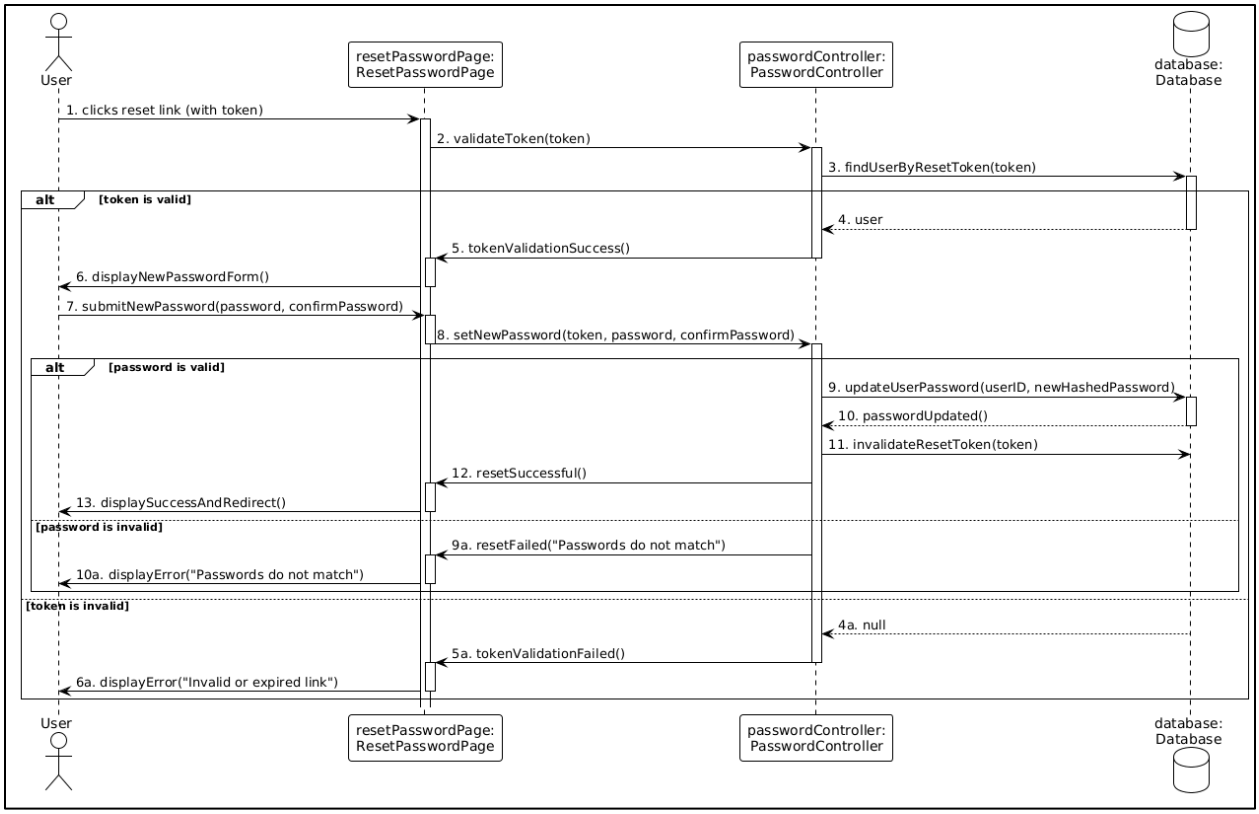
```
        // Logic to connect to an email server and send the message
    }
}

// Entity Class (assuming it has an ID)
public class User {
    private UserID id;
    private String email;

    public UserID getId() {
        return id;
    }
}
```

UC-05: Reset Password

a. Structured use case specification:

Use Case: Reset Password
Scenario 1: Successful Password Reset 1. User clicks the unique reset link received via email. 2. System validates the reset token (checks if it exists and hasn't expired). 3. System displays a form to set a new password, requiring the new password and a confirmation. 4. User enters and confirms their new password. 5. System validates that the passwords match and meet security requirements (e.g., length, complexity). 6. System updates the user's password in the database. 7. System invalidates the used reset token. 8. System displays a success message and redirects the user to the login page. *Alternative 1.1: Invalid or Expired Token* At step 2, if the reset token is not found or has expired. Message "Error: This password reset link is invalid or has expired." *Alternative 1.2: Passwords Do Not Match or Are Weak* At step 5, if the provided passwords do not match or fail to meet security requirements. Message "Error: Passwords do not match or are too weak. Please try again."

b. Sequence diagram:



c. Method Population in Classes:

ResetPasswordPage
validateToken(token)
tokenValidationSuccess()
tokenValidationFailed()
displayNewPasswordForm()
submitNewPassword(password, confirm)
resetSuccessful()
resetFailed(message)
displayError(message)
displaySuccessAndRedirect()

PasswordController
validateToken(token)
setNewPassword(token, password, confirm)

Database
findUserByResetToken(token)
updateUserPassword(userID, newPassword)
invalidateResetToken(token)

d. Code snippets:

```
// View Class
public class ResetPasswordPage {
    private PasswordController passwordController;

    // Triggered when the user lands on the page from the email link
    public void onPageLoad(String token) {
        passwordController.validateToken(token);
    }

    public void displayNewPasswordForm() {
        System.out.println("Please enter and confirm your new password.");
    }

    public void onSubmitNewPassword(String token, String password, String confirm) {
        passwordController.setNewPassword(token, password, confirm);
    }

    public void displayError(String message) {
        System.out.println("Error: " + message);
    }

    public void displaySuccessAndRedirect() {
        System.out.println("Password has been successfully updated. Please log in.");
        // Logic to redirect to login page
    }
}
public class PasswordController {
    private Database database;
    private NotificationService notificationService; // (from previous UC)
    private ResetPasswordPage resetPage; // Reference to the view

    public PasswordController(Database db, NotificationService notifier, ResetPasswordPage
```

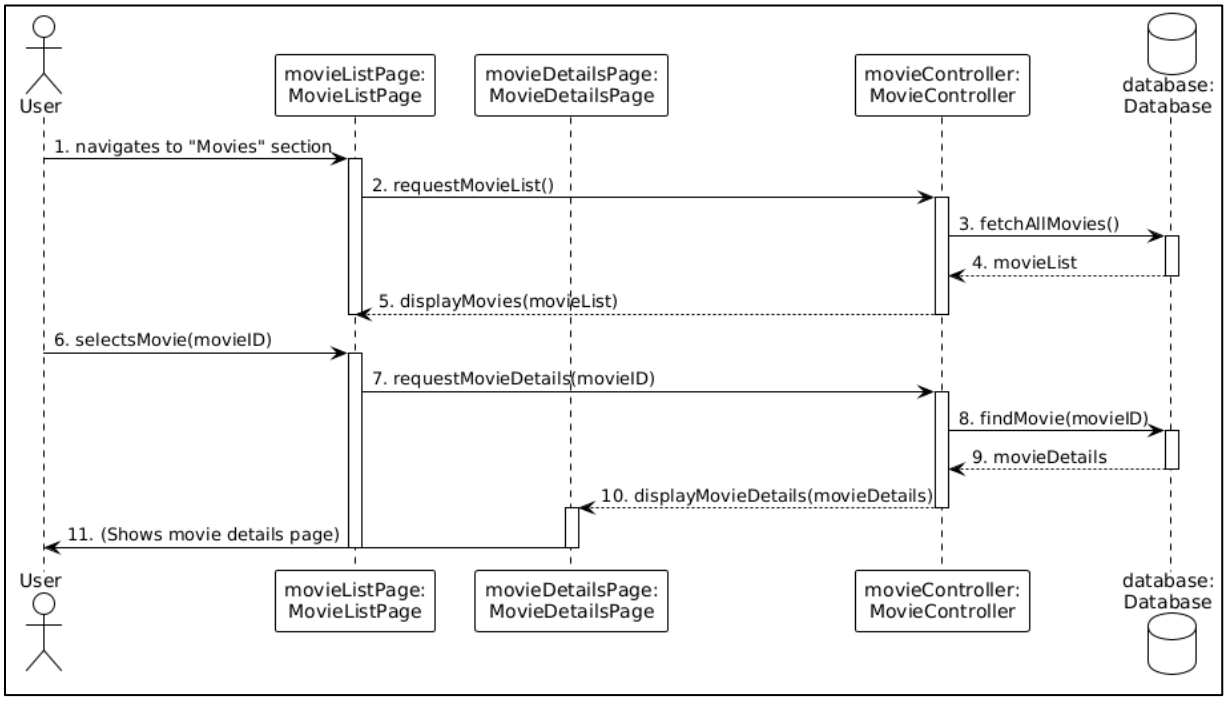
```
page) {
        this.database = db;
        this.notificationService = notifier;
        this.resetPage = page;
    }

    // (Method from previous UC)
    // public void initiatePasswordReset(String email) { ... }

    public void validateToken(String token) {
        User user = database.findUserByResetToken(token);
        if (user != null) {
            resetPage.displayNewPasswordForm();
        } else {
            resetPage.displayError("Invalid or expired link");
        }
    }
    public void setNewPassword(String token, String password, String confirmPassword) {
        if (!password.equals(confirmPassword) || isPasswordWeak(password)) {
            resetPage.displayError("Passwords do not match or are too weak.");
            return;
        }

        User user = database.findUserByResetToken(token);
        if (user != null) {
            // In a real app, hash the password before saving
            database.updateUserPassword(user.getId(), password);
            database.invalidateResetToken(token);
            resetPage.displaySuccessAndRedirect();
        } else {
            resetPage.displayError("Invalid session. Please try again.");
        }
    }
    private boolean isPasswordWeak(String password) {
        // Logic to check password strength
        return password.length() < 8; // Placeholder
    }
}
```

UC-06: Browse Movies

a. Structured use case specification:

Use Case: Browse Movies
Scenario 1: Successfully View Movie List and Details 1. User navigates to the "Movies" or "Now Showing" section. 2. System retrieves a list of currently showing or upcoming movies from the database. 3. System displays the list of movies, showing key information like poster, title, and rating. 4. User selects a specific movie from the list. 5. System retrieves the detailed information for the selected movie. 6. System displays a detailed page for the movie, including its synopsis, cast, showtimes, and an option to watch the trailer. *Alternative 1.1: No Movies Available* At step 2, if there are no movies currently listed in the system. Message "There are currently no movies to display. Please check back later."

b. Sequence diagram:



c. Method Population in Classes:

MovieListPage
requestMovieList()
displayMovies(movieList)
selectsMovie(movieID)

MovieDetailsPage
displayMovieDetails(details)

MovieController
requestMovieList()
requestMovieDetails(movieID)

Database
fetchAllMovies()
findMovie(movieID)

d. Code snippets:

```
// View Class for the list of movies
public class MovieListPage {
    private MovieController movieController;

    // Triggered when the user navigates to the movie list page
    public void onPageLoad() {
        List<MovieSummary> movies = movieController.requestMovieList();
        displayMovies(movies);
    }

    public void displayMovies(List<MovieSummary> movies) {
        System.out.println("--- Now Showing ---");
        if (movies.isEmpty()) {
            System.out.println("No movies to display.");
        } else {
            for (MovieSummary movie : movies) {
                System.out.println("Title: " + movie.getTitle());
            }
```

```
        }
    }

    // Triggered when a user clicks on a movie
    public void onMovieSelect(MovieID movieId) {
        movieController.requestMovieDetails(movieId);
    }
}

// View Class for a single movie's details
public class MovieDetailsPage {
    public void displayMovieDetails(MovieDetails details) {
        System.out.println("--- Movie Details ---");
        System.out.println("Title: " + details.getTitle());
        System.out.println("Synopsis: " + details.getSynopsis());
        System.out.println("Cast: " + details.getCast());
        // Logic to show available showtimes
    }
}


// Controller Class
public class MovieController {
    private Database database;
    private MovieDetailsPage movieDetailsPage; // Reference to the details view

    public MovieController(Database db, MovieDetailsPage detailsPage) {
        this.database = db;
        this.movieDetailsPage = detailsPage;
    }

    public List<MovieSummary> requestMovieList() {
        return database.fetchAllMovies();
    }

    public void requestMovieDetails(MovieID movieId) {
        MovieDetails details = database.findMovie(movieId);
        if (details != null) {
            movieDetailsPage.displayMovieDetails(details);
```

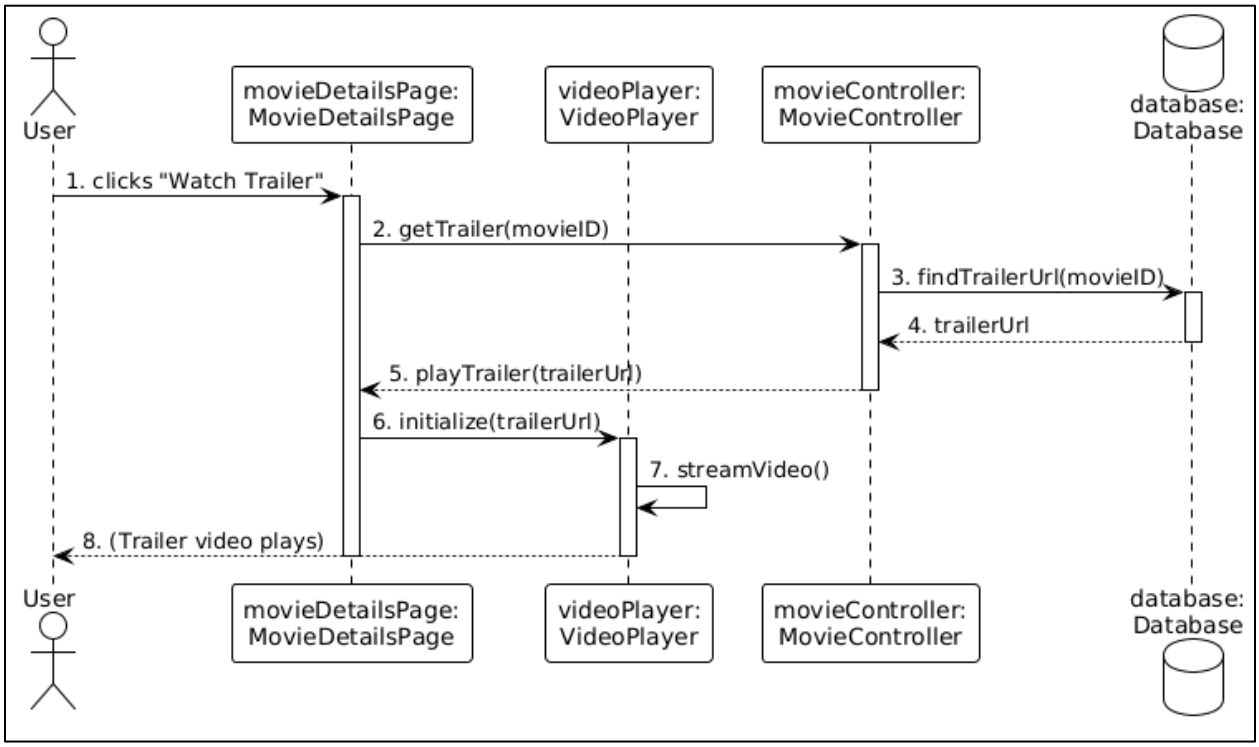
```
        }
    }
}

// Data Transfer Object (DTO) for the movie list
public class MovieSummary {
    private MovieID id;
    private String title;
    private String posterUrl;
    // Getters
}

// Data Transfer Object (DTO) for movie details
public class MovieDetails {
    private String title;
    private String synopsis;
    private List<String> cast;
    private List<Showtime> showtimes;
    // Getters
}
```

UC-07: Watch Trailer

a. Structured use case specification:

Use Case: Watch Trailer
Scenario 1: Successfully Watch Trailer Pre-condition: User is viewing the details page for a specific movie (UC-06). 1. User selects the "Watch Trailer" option on the movie details page. 2. System retrieves the trailer link associated with the movie. 3. System opens a video player modal or navigates to a player page. 4. System begins streaming the trailer. *Alternative 1.1: Trailer Not Available* At step 1, the "Watch Trailer" option is unavailable or disabled if no trailer is linked to the movie.

b. Sequence diagram:



c. Method Population in Classes:

MovieDetailsPage
getTrailer(movieID)
playTrailer(trailerUrl)

MovieController
getTrailer(movieID)

VideoPlayer
initialize(url)
streamVideo()

Database
findTrailerUrl(movieID)

d. Code snippets:

```
// View Class (Adding to MovieDetailsPage from previous UC)
public class MovieDetailsPage {
```

```
    private MovieController movieController;
    private VideoPlayer videoPlayer;
    private MovieID currentMovieId;

    public void displayMovieDetails(MovieDetails details) {
        this.currentMovieId = details.getId();
        System.out.println("Title: " + details.getTitle());
        // ... display other details
        if (details.hasTrailer()) {
            System.out.println("[Watch Trailer button is enabled]");
        } else {
            System.out.println("[Watch Trailer button is disabled]");
        }
    }

    // Triggered when user clicks the "Watch Trailer" button
    public void onWatchTrailerClick() {
        movieController.getTrailer(currentMovieId);
    }

    public void playTrailer(String trailerUrl) {
        videoPlayer.initialize(trailerUrl);
        videoPlayer.streamVideo();
    }
}


// View Component for video playback
public class VideoPlayer {
    public void initialize(String url) {
        System.out.println("Initializing video player with URL: " + url);
    }

    public void streamVideo() {
        System.out.println("Streaming video... [PRETEND VIDEO IS PLAYING]");
    }
}
```

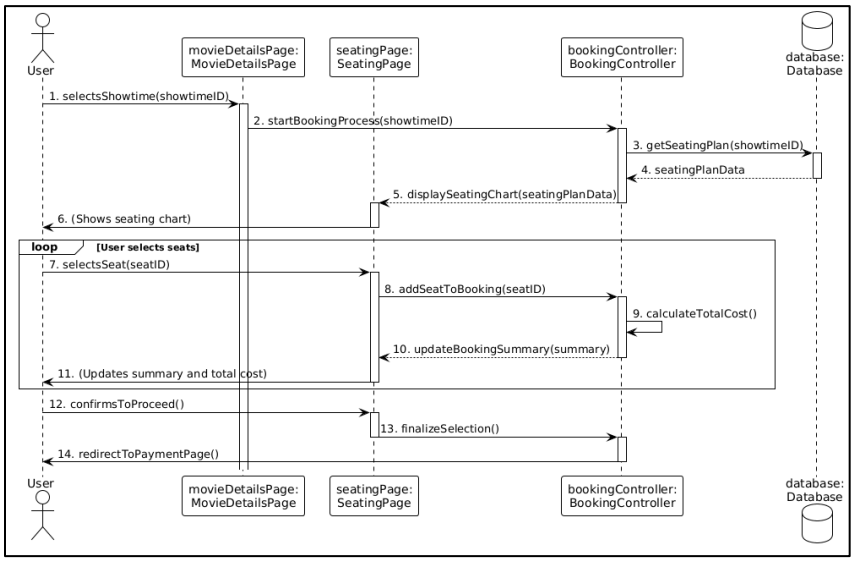
```
// Controller Class (Adding to MovieController from previous UC)
public class MovieController {
    private Database database;
    private MovieDetailsPage movieDetailsPage; // Reference from previous UC

    // ... constructor and other methods from UC-06 ...

    public void getTrailer(MovieID movieId) {
        String trailerUrl = database.findTrailerUrl(movieId);
        if (trailerUrl != null && !trailerUrl.isEmpty()) {
            movieDetailsPage.playTrailer(trailerUrl);
        } else {
            // This case is handled by disabling the button in the view,
            // but could also show an error if clicked somehow.
            System.out.println("No trailer found for this movie.");
        }
    }
}

// Data Transfer Object (Adding trailer info)
public class MovieDetails {
    private MovieID id;
    private String title;
    private String synopsis;
    private boolean hasTrailer; // Flag to enable/disable button in UI
    // Getters
}

// Database Class (New method)
public class Database {
    // ... other methods ...
    public String findTrailerUrl(MovieID movieId) {
        // SQL query to find the trailer URL for the given movie ID
        return "https://example.com/trailer.mp4"; // Placeholder
    }
}
```

UC-08: Book Ticket

a. Structured use case specification:

Use Case: Book Ticket
Scenario 1: Successful Booking Flow Initiation **Pre-condition:** User is logged in and viewing the details for a specific movie. 1. User selects a specific showtime for the movie. 2. System retrieves the seating layout for the corresponding screen, including seat availability. 3. System displays the seating chart. 4. User selects one or more desired seats. 5. As seats are selected, the system calculates and displays a running summary of the selection (number of tickets) and the total cost. 6. User confirms their seat selection and proceeds to payment. 7. The system finalizes the booking details and transitions to the "Process Payment" use case (UC-09). *Alternative 1.1: Seat Becomes Unavailable* At step 4, if a selected seat becomes unavailable while the user is choosing (e.g., booked by another user simultaneously). System displays an alert: "Sorry, one or more selected seats are no longer available. Please choose another seat." The seating chart is refreshed to show the updated availability.

b. Sequence diagram:



c. Method Population in Classes:

MovieDetailsPage
selectsShowtime(showtimeID)

SeatingPage
displaySeatingChart(data)
selectsSeat(seatID)
updateBookingSummary(summary)
confirmsToProceed()

BookingController
startBookingProcess(showtimeID)
addSeatToBooking(seatID)
calculateTotalCost()
finalizeSelection()
redirectToPaymentPage()

Database
getSeatingPlan(showtimeID)

d. Code snippets:

```
// View Class (for seat selection)
public class SeatingPage {
    private BookingController bookingController;

    public void displaySeatingChart(SeatingPlanData data) {
        System.out.println("Displaying seating chart for screen: " + data.getScreenName());
        // Logic to render the seats and their availability
    }

    // Triggered when a user clicks a seat
    public void onSeatSelected(SeatID seatId) {
        bookingController.addSeatToBooking(seatId);
    }

    public void updateBookingSummary(BookingSummary summary) {
```

```
        System.out.println("--- Booking Summary ---");
        System.out.println("Selected Seats: " + summary.getSelectedSeatsCount());
        System.out.println("Total Cost: $" + summary.getTotalCost());
    }
    public void onConfirm() {
        bookingController.finalizeSelection();
    }
}
public class BookingController {
    private Database database;
    private SeatingPage seatingPage;
    private PaymentPage paymentPage; // The view for the next use case
    private Booking currentBooking; // State object for the current booking

    public BookingController(Database db, SeatingPage sp, PaymentPage pp) {
        this.database = db;
        this.seatingPage = sp;
        this.paymentPage = pp;
    }

    public void startBookingProcess(ShowtimeID showtimeId) {
        this.currentBooking = new Booking(showtimeId);
        SeatingPlanData plan = database.getSeatingPlan(showtimeId);
        seatingPage.displaySeatingChart(plan);
    }

    public void addSeatToBooking(SeatID seatId) {
        currentBooking.addSeat(seatId);
        calculateTotalCost();
        BookingSummary summary = currentBooking.getSummary();
        seatingPage.updateBookingSummary(summary);
    }

    private void calculateTotalCost() {
        double pricePerTicket = 15.00; // Placeholder
        double total = currentBooking.getSeats().size() * pricePerTicket;
        currentBooking.setTotalCost(total);
    }
```

```
    public void finalizeSelection() {
        System.out.println("Seat selection finalized. Redirecting to payment...");
        redirectToPaymentPage();
    }

    public void redirectToPaymentPage() {
        paymentPage.displayPaymentForm(currentBooking.getTotalCost());
    }
}

// Represents the state of the booking being created
public class Booking {
    private ShowtimeID showtimeId;
    private List<SeatID> selectedSeats;
    private double totalCost;

    public Booking(ShowtimeID showtimeId) {
        this.showtimeId = showtimeId;
        this.selectedSeats = new ArrayList<>();
    }

    public void addSeat(SeatID seat) { this.selectedSeats.add(seat); }
    public List<SeatID> getSeats() { return selectedSeats; }
    public void setTotalCost(double cost) { this.totalCost = cost; }
    public double getTotalCost() { return totalCost; }

    public BookingSummary getSummary() { /* ... creates summary DTO ... */ }
}
```

UC-09: Process Payment

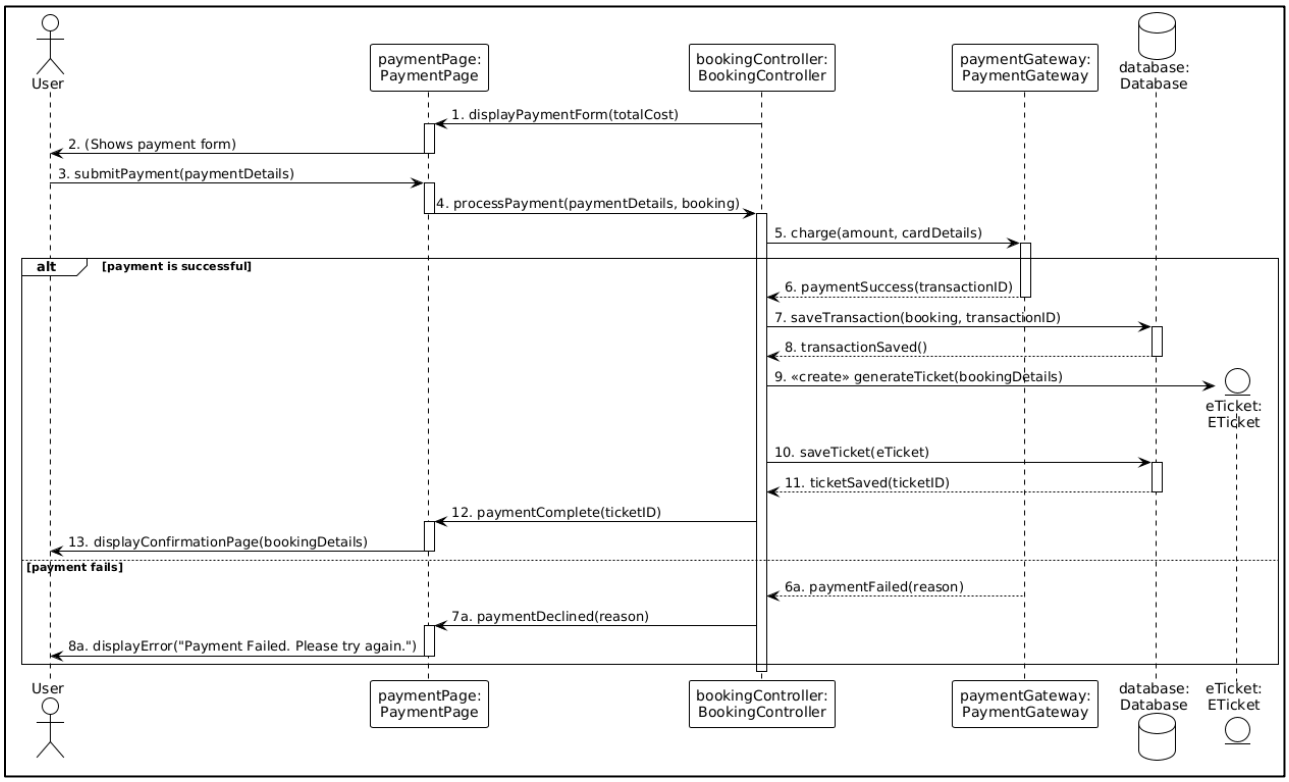
a. Structured use case specification:

Use Case: Process Payment
Scenario 1: Successful Payment Transaction **Pre-condition:** User has confirmed their ticket selection and is at the payment stage (UC-08). 1. System displays a secure payment form, showing the total amount due. 2. User enters their payment information (e.g., credit card

details) and confirms the transaction.
3. System securely transmits the payment data to an external payment gateway.
4. The payment gateway processes the transaction and returns a success response.
5. System receives the success response.
6. System records the successful transaction in the database.
7. System finalizes the booking, marks the seats as sold, and generates an e-ticket record.
8. System displays a confirmation page with booking details and sends a confirmation email.

Alternative 1.1: Payment Gateway Declines Transaction
At step 4, the payment gateway returns a failure response (e.g., insufficient funds, incorrect details). System displays an error message to the user: "Payment failed. Please check your details and try again." The user remains on the payment page to correct their information or try a different payment method.

b. Sequence diagram:



c. Method Population in Classes:

PaymentPage
displayPaymentForm(totalCost)
submitPayment(details)
paymentComplete(ticketID)
displayConfirmationPage(details)
paymentDeclined(reason)
displayError(message)

BookingController
processPayment(details, booking)

PaymentGateway
charge(amount, details)

Database
saveTransaction(booking, transactionID)
saveTicket(eTicket)

ETicket
generateTicket(details) (or Constructor)

d. Code snippets:

```
// View Class for Payment
public class PaymentPage {
    private BookingController bookingController;
    private Booking currentBooking; // State passed from previous step

    public void displayPaymentForm(double totalCost) {
        System.out.println("--- Payment ---");
        System.out.println("Total Amount Due: $" + totalCost);
        // Logic to show payment input fields
    }

    public void onSubmitPayment(PaymentDetails details) {
        bookingController.processPayment(details, currentBooking);
    }
```

```
    public void displayConfirmationPage(BookingDetails details) {
        System.out.println("--- Booking Confirmed! ---");
        System.out.println("Your e-ticket has been generated.");
        // Display summary of the confirmed booking
    }

    public void displayError(String message) {
        System.out.println("Error: " + message);
    }
}

// Controller Class (adding new method to BookingController)
public class BookingController {
    private Database database;
    private PaymentGateway paymentGateway;
    private PaymentPage paymentPage;
    // ... other fields and methods from UC-08 ...

    public void processPayment(PaymentDetails details, Booking booking) {
        PaymentResult result = paymentGateway.charge(booking.getTotalCost(), details);

        if (result.isSuccessful()) {
            database.saveTransaction(booking, result.getTransactionId());
            ETicket ticket = new ETicket(booking); // Create the ticket
            TicketID ticketId = database.saveTicket(ticket);

            // In a real app, signal completion to the view
            paymentPage.paymentComplete(ticketId);
            paymentPage.displayConfirmationPage(booking.getDetails());

        } else {
            // In a real app, signal failure to the view
            paymentPage.paymentDeclined(result.getErrorMessage());
            paymentPage.displayError("Payment Failed. Please try again.");
        }
    }
}
```

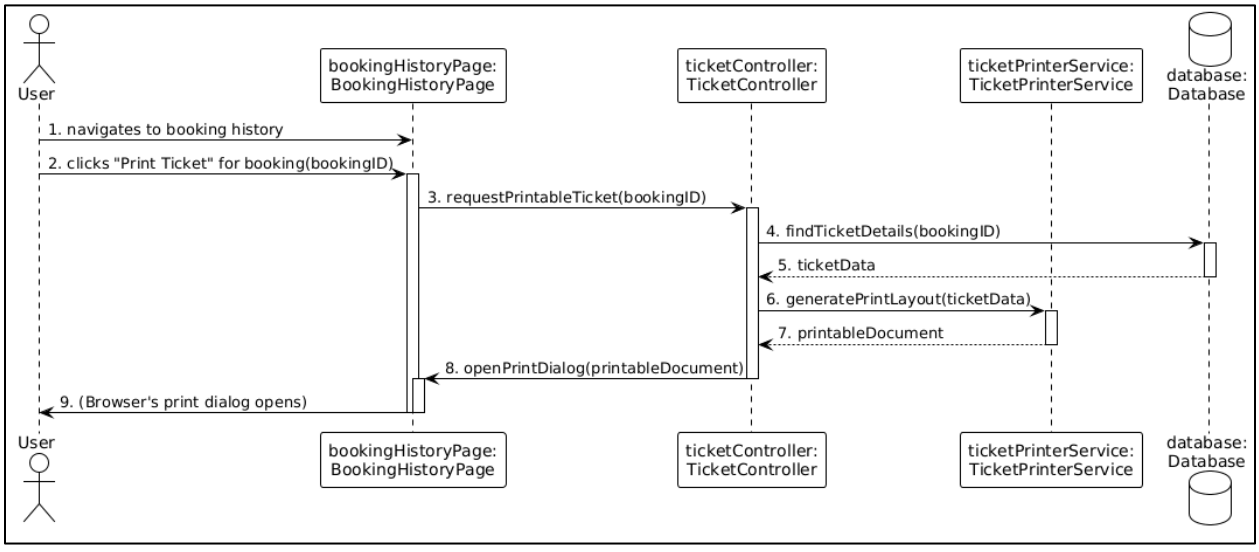
```
// Represents the external payment service
public class PaymentGateway {
    public PaymentResult charge(double amount, PaymentDetails details) {
        // Simulates communication with a real payment provider like Stripe or PayPal
        System.out.println("Charging $" + amount + " to card ending in " +
details.getCardLastFour());
        // Placeholder logic
        if (details.isValid()) {
            return new PaymentResult(true, "txn_12345", null);
        } else {
            return new PaymentResult(false, null, "Insufficient funds");
        }
    }
}

// Entity Class for the ticket
public class ETicket {
    private TicketID id;
    private Booking bookingInfo;

    public ETicket(Booking booking) {
        this.bookingInfo = booking;
        // Generate a unique ID, QR code, etc.
    }
    // Getters
}

// DTO for payment result
public class PaymentResult {
    private boolean successful;
    private String transactionId;
    private String errorMessage;
    // Constructor and getters
}
```

UC-10: Print E-Ticket

a. Structured use case specification:

Use Case: Print E-Ticket
Scenario 1: Successfully Print E-Ticket **Pre-condition:** User has a successfully booked ticket in their account. 1. User navigates to their booking history or views the confirmation page for a specific booking. 2. User selects the "Print Ticket" or "View Ticket" option. 3. System retrieves all the necessary details for the booking (movie, showtime, seats, QR code, etc.). 4. System formats the details into a printable e-ticket layout (e.g., a PDF or a specific HTML view). 5. System opens the browser's print dialog to allow the user to print the ticket or save it as a PDF. *Alternative 1.1: Booking Not Found* At step 3, if the specified booking ID is invalid or does not belong to the user. Message "Error: The requested booking could not be found."

b. Sequence diagram:



c. Method Population in Classes:

BookingHistoryPage
requestPrintableTicket(bookingID)
openPrintDialog(document)

TicketController
requestPrintableTicket(bookingID)

TicketPrinterService
generatePrintLayout(data)

Database
findTicketDetails(bookingID)

d. Code snippets:

```
// View Class (e.g., a page showing past bookings)
public class BookingHistoryPage {
    private TicketController ticketController;

    // Triggered when user clicks the "Print Ticket" button for a booking
    public void onPrintTicketClick(BookingID bookingId) {
        ticketController.requestPrintableTicket(bookingId);
    }

    public void openPrintDialog(PrintableDocument document) {
        System.out.println("--- Preparing Ticket for Printing ---");
        System.out.println(document.getContent());
        System.out.println("Opening browser's print dialog...");
    }
}


// Controller Class
public class TicketController {
    private Database database;
    private TicketPrinterService printerService;

    public TicketController(Database db, TicketPrinterService service) {
        this.database = db;
        this.printerService = service;
    }

    public void requestPrintableTicket(BookingID bookingId) {
        TicketData data = database.findTicketDetails(bookingId);
```

```
        if (data != null) {
            PrintableDocument document = printerService.generatePrintLayout(data);
            // In a real app, this would be a callback to the view
            // new BookingHistoryPage(this).openPrintDialog(document);
        } else {
            // Signal error back to the view
            System.out.println("Error: Booking not found.");
        }
    }
}


// Service Class responsible for formatting the ticket
public class TicketPrinterService {
    public PrintableDocument generatePrintLayout(TicketData data) {
        System.out.println("Generating print layout for ticket...");
        // Logic to format data into HTML or PDF content
        String content = "Movie: " + data.getMovieTitle() + "\n" +
                         "Showtime: " + data.getShowtime() + "\n" +
                         "Seats: " + data.getSeats();
        return new PrintableDocument(content);
    }
}

// DTO for all data needed for a ticket
public class TicketData {
    private String movieTitle;
    private String showtime;
    private List<String> seats;
    private String qrCodeUrl;
    // Getters
}

// Represents the formatted document to be printed
public class PrintableDocument {
    private String content; // Could be HTML, PDF bytes, etc.

    public PrintableDocument(String content) {
        this.content = content;
```

```
    }

    public String getContent() {
        return content;
    }
}
```

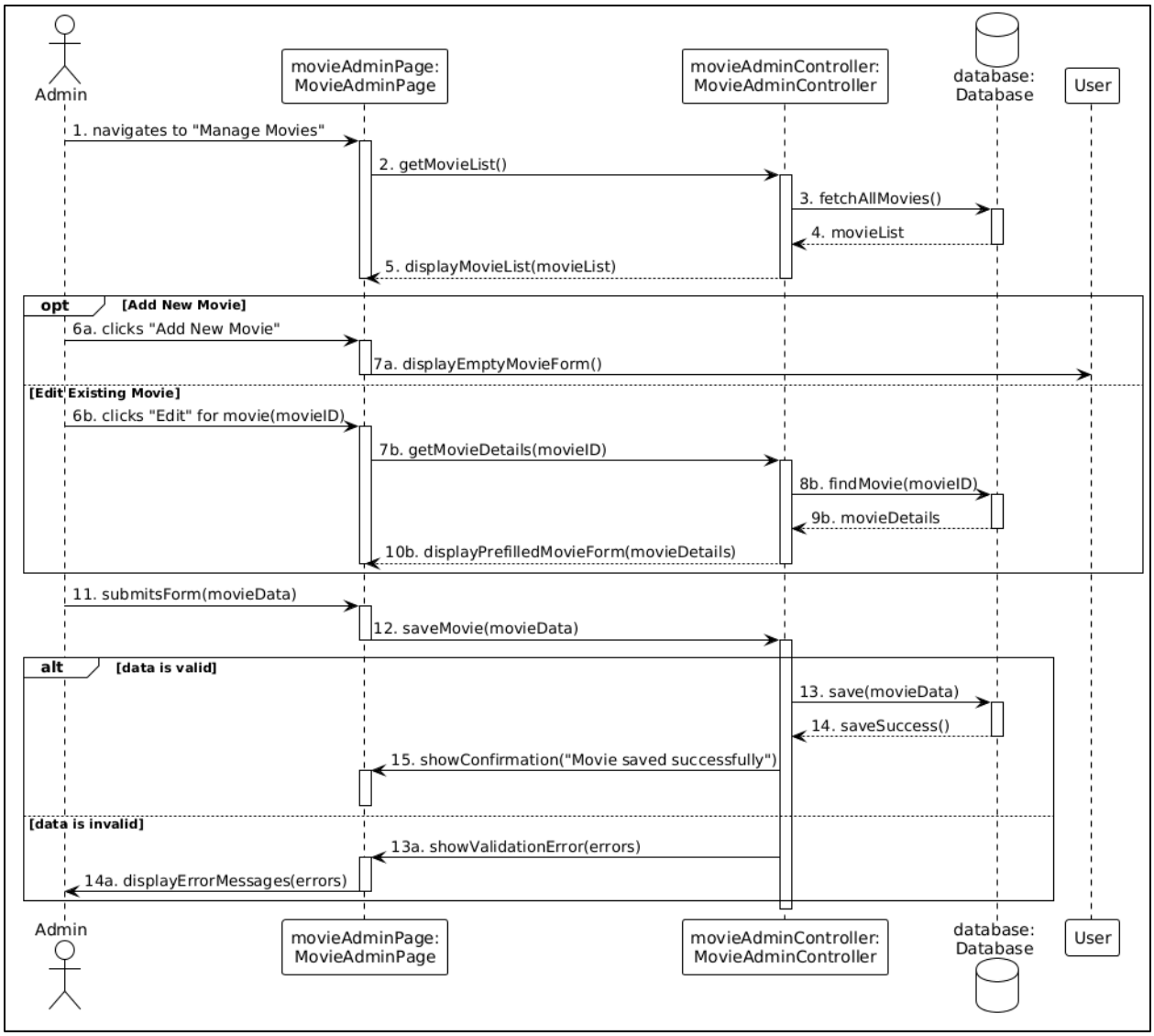
UC-11: Manage Movies

a. Structured use case specification:

Use Case: Manage Movies
Scenario 1: Admin Successfully Adds a New Movie **Pre-condition:** Admin is logged into the administrative panel. 1. Admin navigates to the "Manage Movies" section. 2. System displays a list of all movies in the catalog. 3. Admin selects the "Add New Movie" option. 4. System displays an empty form for movie details (e.g., title, synopsis, cast, poster image, trailer link). 5. Admin enters the details for the new movie and submits the form. 6. System validates the submitted data. 7. System creates a new movie record in the database. 8. System displays a success message, "Movie added successfully," and refreshes the movie list. *Scenario 2: Admin Successfully Edits an Existing Movie* **Pre-condition:** Admin is logged into the administrative panel and viewing the movie list. 1. Admin selects the "Edit" option for an existing movie. 2. System displays a form pre-filled with the selected movie's current details. 3. Admin modifies the movie's details and submits the form. 4. System validates the submitted data. 5. System updates the existing movie record in the database. 6. System displays a success message, "Movie updated successfully," and refreshes the movie list. *Alternative 2.1: Invalid Data Submitted* At step 6 (*Scenario 1*) or step 4 (*Scenario 2*), if the submitted data is invalid (e.g., missing title).

System displays an error message next to the relevant form fields, indicating the required corrections. The submitted data is preserved in the form for easy editing.

b. Sequence diagram:



c. Method Population in Classes:

MovieAdminPage
getMovieList()
displayMovieList(list)
displayEmptyMovieForm()
getMovieDetails(movieID)
displayPrefilledMovieForm(details)

submitsForm(data)
showConfirmation(message)
showValidationError(errors)
displayErrorMessages(errors)

MovieAdminController
getMovieList()
getMovieDetails(movieID)
saveMovie(data)
validateMovieData(data)

Database
fetchAllMovies()
findMovie(movieID)
save(movieData)

d. Code snippets:

```
// View Class for Admin Movie Management
public class MovieAdminPage {
   private MovieAdminController movieAdminController;

   public void onPageLoad() {
      List<MovieSummary> movies = movieAdminController.getMovieList();
      displayMovieList(movies);
   }

   public void displayMovieList(List<MovieSummary> movies) {
      System.out.println("--- Movie Catalog ---");
      // Logic to display movies in a table with Edit/Delete buttons
   }

   public void onAddNewMovieClick() {
      displayEmptyMovieForm();
   }

   public void displayEmptyMovieForm() {
      System.out.println("Displaying blank form for new movie...");
   }
```

```
    public void onEditMovieClick(MovieID movieId) {
        MovieDetails details = movieAdminController.getMovieDetails(movieId);
        displayPrefilledMovieForm(details);
    }

    public void displayPrefilledMovieForm(MovieDetails details) {
        System.out.println("Displaying form to edit: " + details.getTitle());
    }

    public void onSubmitForm(MovieData data) {
        movieAdminController.saveMovie(data);
    }

    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
        onPageLoad(); // Refresh the list
    }

    public void displayErrorMessages(List<ValidationError> errors) {
        System.out.println("Error: Please correct the following issues:");
        for (ValidationError error : errors) {
            System.out.println("- " + error.getField() + ": " + error.getMessage());
        }
    }
}


// Controller Class for Admin
public class MovieAdminController {
    private Database database;
    private MovieAdminPage view; // Reference to the view

    public MovieAdminController(Database db, MovieAdminPage view) {
        this.database = db;
        this.view = view;
    }

    public List<MovieSummary> getMovieList() {
        return database.fetchAllMovies();
```

```
    }

    public MovieDetails getMovieDetails(MovieID movieId) {
        return database.findMovie(movieId);
    }

    public void saveMovie(MovieData data) {
        List<ValidationError> errors = validateMovieData(data);
        if (errors.isEmpty()) {
            database.save(data);
            view.showConfirmation("Movie saved successfully");
        } else {
            view.displayErrorMessages(errors);
        }
    }

    private List<ValidationError> validateMovieData(MovieData data) {
        List<ValidationError> errors = new ArrayList<>();
        if (data.getTitle() == null || data.getTitle().isEmpty()) {
            errors.add(new ValidationError("title", "Title cannot be empty."));
        }
        // Add other validation rules...
        return errors;
    }
}

// DTO representing data from the form
public class MovieData {
    private MovieID id; // Can be null for new movies
    private String title;
    private String synopsis;
    // Getters and Setters
}
```

UC-12: Delete Movie

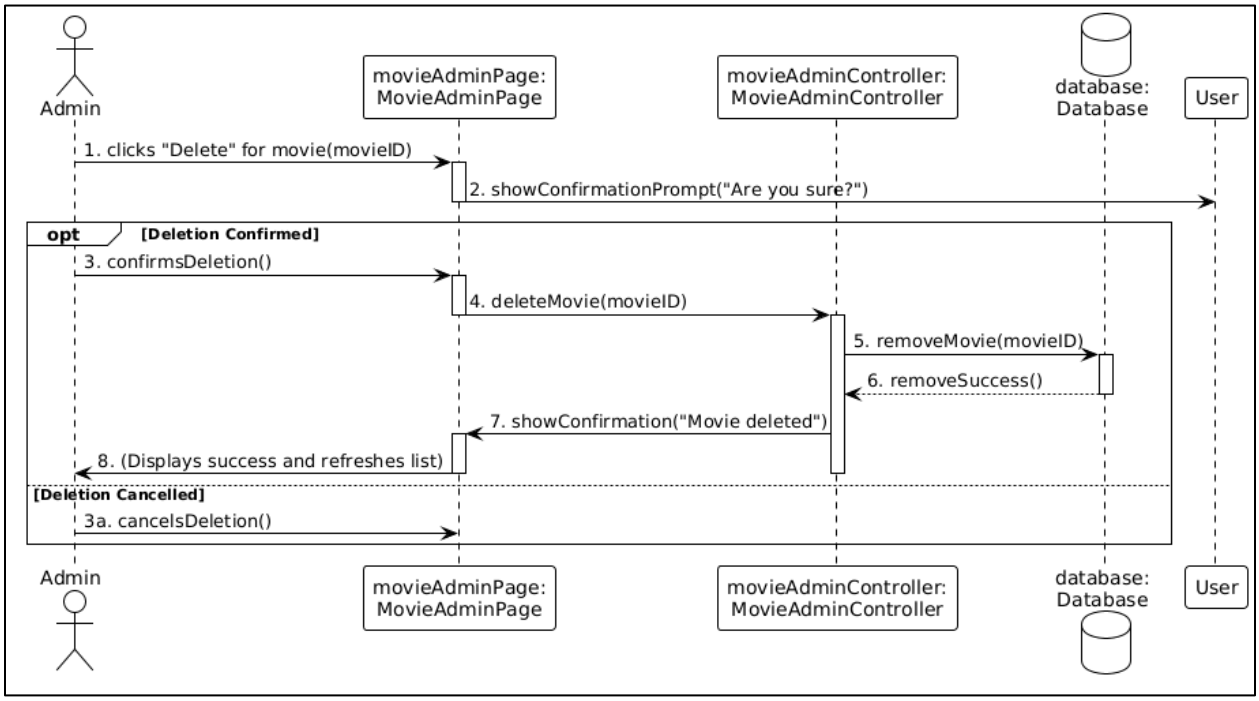
a. Structured use case specification:

Use Case: Delete Movie
Scenario 1: Admin Successfully Deletes a Movie

Pre-condition: Admin is logged in and viewing the movie list (UC-11).
1. Admin selects the "Delete" option for a specific movie.
2. System displays a confirmation prompt (e.g., "Are you sure you want to permanently delete this movie?").
3. Admin confirms the deletion.
4. System removes the movie record from the database.
5. System displays a success message, "Movie deleted successfully," and refreshes the movie list.

Alternative 1.1: Deletion is Cancelled
At step 3, if the Admin cancels the action at the confirmation prompt.
No action is taken, and the system returns to the movie list.

b. Sequence diagram:



c. Method Population in Classes:

MovieAdminPage
showConfirmationPrompt(message)
confirmsDeletion()
cancelsDeletion()
deleteMovie(movieID)

showConfirmation(message)

MovieAdminController
deleteMovie(movieID)

Database
removeMovie(movieID)

d. Code snippets:

```
// View Class (Adding to MovieAdminPage from previous UC)
public class MovieAdminPage {
    private MovieAdminController movieAdminController;
    private MovieID movieToDelete; // Store the ID of the movie targeted for deletion

    // ... methods from UC-11 ...

    // Triggered when admin clicks the delete button in the movie list
    public void onDeleteClick(MovieID movieId) {
        this.movieToDelete = movieId;
        showConfirmationPrompt("Are you sure you want to permanently delete this movie?");
    }

    public void showConfirmationPrompt(String message) {
        System.out.println("CONFIRMATION: " + message + " [Yes/No]");
    }

    // Triggered when admin confirms the action in the prompt
    public void onConfirmDeletion() {
        if (movieToDelete != null) {
            movieAdminController.deleteMovie(movieToDelete);
        }
    }

    public void onCancelDeletion() {
        this.movieToDelete = null;
        System.out.println("Deletion cancelled.");
    }

    // This is a callback from the controller, not a direct user action method
```

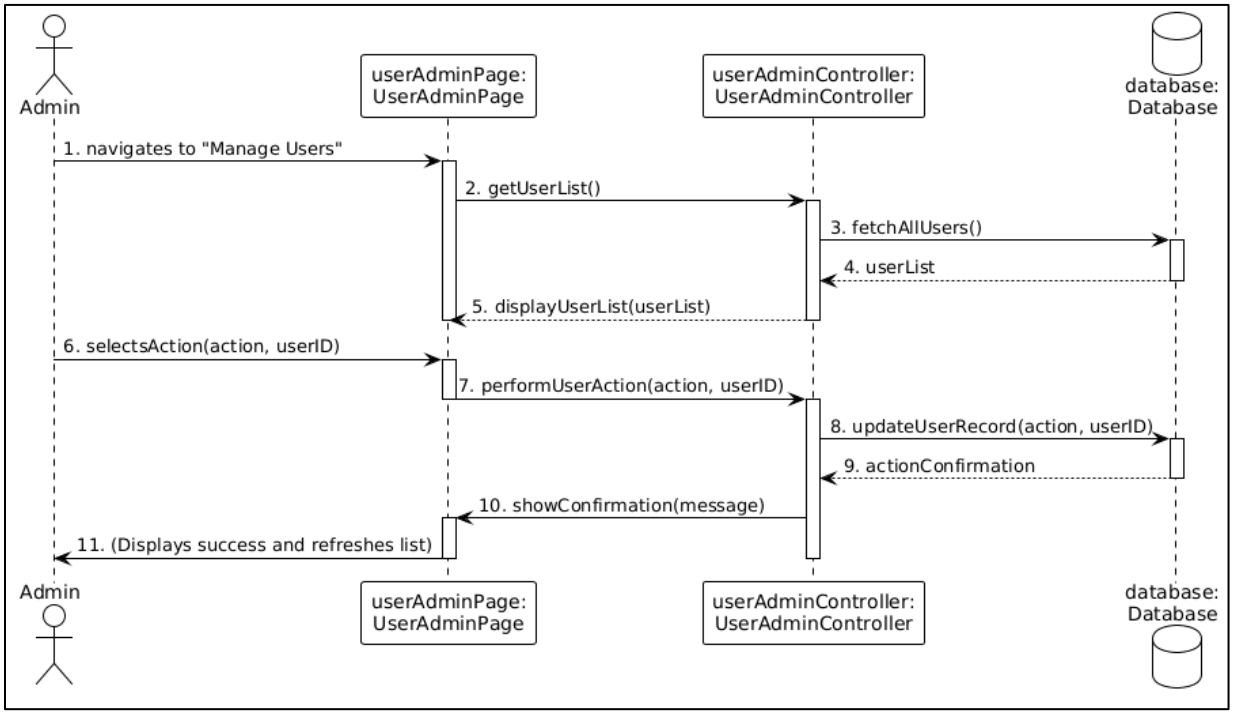
```
    // It's called after the controller has processed the request
    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
        onPageLoad(); // Refresh the list
    }
}


// Controller Class (Adding to MovieAdminController from previous UC)
public class MovieAdminController {
    private Database database;
    private MovieAdminPage view;

    // ... constructor and other methods from UC-11 ...

    public void deleteMovie(MovieID movieId) {
        // Optional: Add business logic checks here, e.g., cannot delete if movie has active
screenings.
        boolean success = database.removeMovie(movieId);

        if (success) {
            view.showConfirmation("Movie deleted successfully");
        } else {
            // In a real app, you'd show an error, e.g., "Movie could not be deleted."
        }
    }
}

// Database Class (New method)
public class Database {

    public boolean removeMovie(MovieID movieId) {
        System.out.println("Deleting movie with ID: " + movieId);
        return true; // Placeholder
    }
}
```

UC-13: Manage Users

a. Structured use case specification:

Use Case: Manage Users
Scenario 1: Admin Views and Acts on User Accounts **Pre-condition:** Admin is logged into the administrative panel. 1. Admin navigates to the "Manage Users" section. 2. System retrieves a list of all registered user accounts from the database. 3. System displays the list of users with key details (e.g., name, email, status). 4. Admin selects a user and chooses an action to perform (e.g., "Deactivate Account", "Reset Password"). 5. System executes the requested action (e.g., updates the user's status in the database). 6. System displays a confirmation of the action taken. 7. The user list is refreshed to show the updated information. *Alternative 1.1: Action Cannot Be Performed* At step 5, if a business rule prevents the action (e.g., trying to deactivate the last remaining admin account). System displays an error message: "Action cannot be performed."

b. Sequence diagram:



c. Method Population in Classes:

UserAdminPage
getUserList()
displayUserList(list)
selectsAction(action, userID)
performUserAction(action, userID)
showConfirmation(message)

UserAdminController
getUserList()
performUserAction(action, userID)

Database
fetchAllUsers()
updateUserRecord(action, userID)

d. Code snippets:

```
// View Class for Admin User Management
public class UserAdminPage {
    private UserAdminController userAdminController;

    // Triggered on page load
    public void onPageLoad() {
        List<UserData> users = userAdminController.getUserList();
        displayUserList(users);
    }

    public void displayUserList(List<UserData> users) {
        System.out.println("--- User Accounts ---");
        for (UserData user : users) {
            System.out.println("User: " + user.getName() + ", Email: " + user.getEmail() + ", Status:
" + user.getStatus());
        }
        // UI would show a table with action buttons (Deactivate, etc.)
    }

    // Triggered when admin clicks an action button for a user
    public void onUserAction(ActionType action, UserID userId) {
```

```
        userAdminController.performUserAction(action, userId);
    }

    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
        onPageLoad(); // Refresh the list of users
    }
}

// Enum to represent possible actions
public enum ActionType {
    DEACTIVATE_USER,
    ACTIVATE_USER,
    RESET_PASSWORD
}


// Controller Class for Admin
public class UserAdminController {
    private Database database;
    private UserAdminPage view;

    public UserAdminController(Database db, UserAdminPage view) {
        this.database = db;
        this.view = view;
    }

    public List<UserData> getUserList() {
        return database.fetchAllUsers();
    }

    public void performUserAction(ActionType action, UserID userId) {
        boolean success = database.updateUserRecord(action, userId);

        if (success) {
            String message = "";
            switch (action) {
                case DEACTIVATE_USER:
                    message = "User has been deactivated.";
```

```
                break;
            case ACTIVATE_USER:
                message = "User has been activated.";
                break;
            // other cases
        }
        view.showConfirmation(message);
    } else {
        // In a real app, show an error message.
    }
  }
}


// Database Class (New methods)
public class Database {
  // ... other methods ...

  public List<UserData> fetchAllUsers() {
    System.out.println("Fetching all user records...");
    // SQL SELECT query to get all users
    return new ArrayList<UserData>(); // Placeholder
  }

  public boolean updateUserRecord(ActionType action, UserID userId) {
    System.out.println("Performing action '" + action + "' on user ID: " + userId);
    // SQL UPDATE statement based on the action
    // e.g., UPDATE users SET status = 'inactive' WHERE id = ?
    return true; // Placeholder
  }
}


// DTO for user data
public class UserData {
  private UserID id;
  private String name;
  private String email;
  private String status;
```

```
    // Getters
}
```

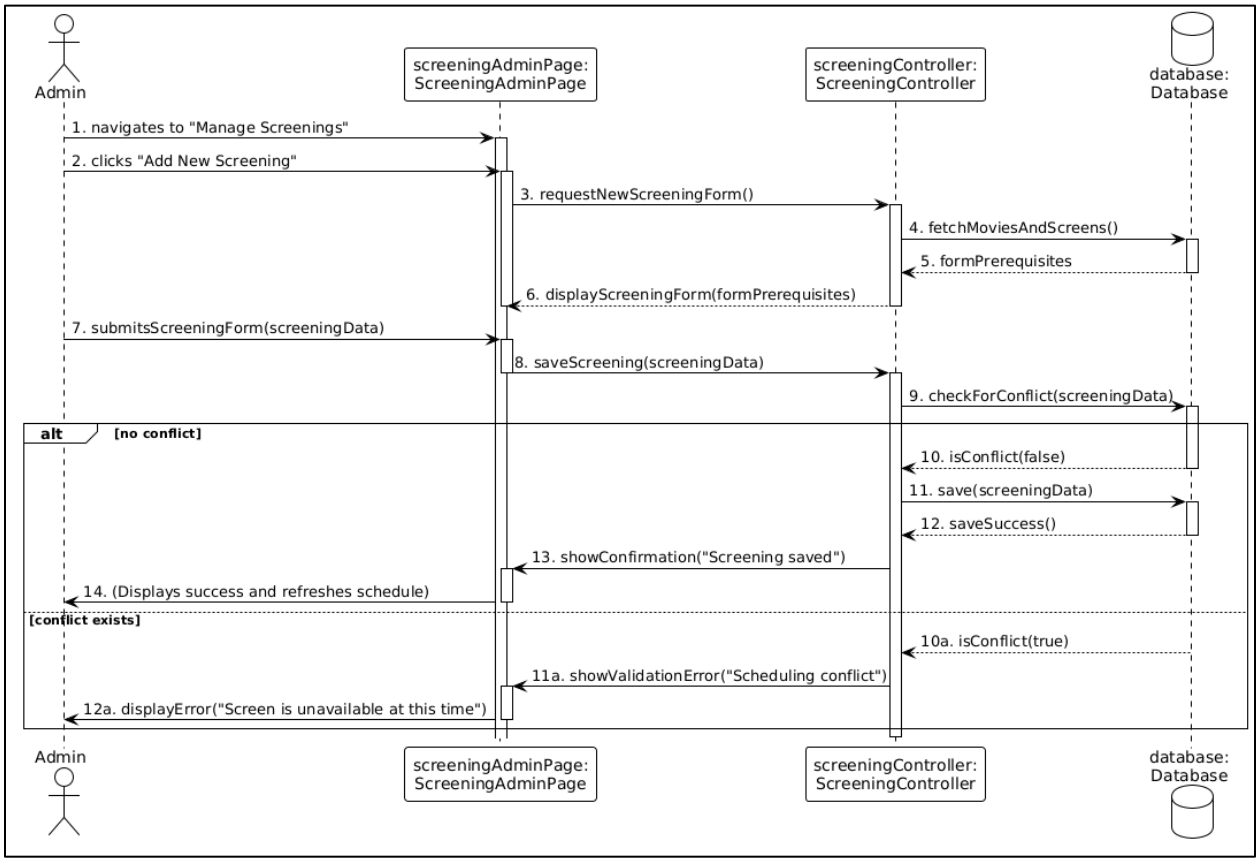
UC-14: Manage Screenings

a. Structured use case specification:

Use Case: Manage Screenings
Scenario 1: Admin Successfully Creates a New Screening **Pre-condition:** Admin is logged into the administrative panel. 1. Admin navigates to the "Manage Screenings" or "Schedule" section. 2. System displays the current schedule of screenings. 3. Admin selects an option to add a new screening. 4. System displays a form to create a new screening. 5. Admin selects a movie, a screen, a date, and a time, then submits the form. 6. System validates the form, checking for potential scheduling conflicts (e.g., the same screen is not booked at the same time). 7. System saves the new screening record to the database. 8. System displays a success message and refreshes the schedule. *Scenario 2: Admin Successfully Updates an Existing Screening* **Pre-condition:** Admin is logged in and viewing the screening schedule. 1. Admin selects an existing screening to edit. 2. System displays a form pre-filled with the screening's details. 3. Admin updates the details (e.g., changes the time) and submits the form. 4. System validates the updated information for scheduling conflicts. 5. System updates the screening record in the database. 6. System displays a success message and refreshes the schedule. *Alternative 2.1: Scheduling Conflict Detected* At step 6 (*Scenario 1*) or step 4 (*Scenario 2*), if the

selected screen is already booked for the specified date/time. System displays an error message: "Scheduling conflict detected. The selected screen is unavailable at this time."

b. Sequence diagram:



c. Method Population in Classes:

ScreeningAdminPage
requestNewScreeningForm()
displayScreeningForm(data)
submitsScreeningForm(data)
showConfirmation(message)
showValidationError(message)
displayError(message)

ScreeningController
requestNewScreeningForm()
saveScreening(data)

Database
fetchMoviesAndScreens()
checkForConflict(data)
save(screeningData)

d. Code snippets:

```
// View Class for Screening Management
public class ScreeningAdminPage {
    private ScreeningController screeningController;

    public void onAddNewScreeningClick() {
        screeningController.requestNewScreeningForm();
    }

    // Displays the form with dropdowns for movies and screens
    public void displayScreeningForm(FormPrerequisites data) {
        System.out.println("Displaying screening creation form.");
        // Logic to populate movie and screen selectors from 'data'
    }

    // Triggered when admin submits the form to create or update a screening
    public void onSubmitForm(ScreeningData screeningData) {
        screeningController.saveScreening(screeningData);
    }

    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
        // Logic to refresh the schedule view
    }

    public void displayError(String message) {
        System.out.println("Error: " + message);
    }
}


// Controller Class
public class ScreeningController {
```

```
    private Database database;
    private ScreeningAdminPage view;

    public ScreeningController(Database db, ScreeningAdminPage view) {
        this.database = db;
        this.view = view;
    }

    public void requestNewScreeningForm() {
        FormPrerequisites data = database.fetchMoviesAndScreens();
        view.displayScreeningForm(data);
    }

    public void saveScreening(ScreeningData data) {
        boolean hasConflict = database.checkForConflict(data);

        if (!hasConflict) {
            database.save(data);
            view.showConfirmation("Screening has been successfully scheduled.");
        } else {
            view.displayError("Scheduling conflict detected. The selected screen is unavailable at
this time.");
        }
    }
}

// Database Class (New methods)
public class Database {
    // ... other methods ...

    public FormPrerequisites fetchMoviesAndScreens() {
        // Fetches all movies and all available screens to populate form dropdowns
        System.out.println("Fetching movies and screens for form...");
        return new FormPrerequisites(); // Placeholder
    }

    public boolean checkForConflict(ScreeningData data) {
        // SQL query to check if any other screening exists for the given
        // screenID at the specified dateTime.
```

```
        System.out.println("Checking for scheduling conflicts...");
        return false; // Placeholder for no conflict
    }

    public boolean save(ScreeningData data) {
        // SQL INSERT or UPDATE statement to save the screening details.
        System.out.println("Saving screening to database...");
        return true; // Placeholder
    }
}


// DTO for screening data from the form
public class ScreeningData {
    private MovieID movieId;
    private ScreenID screenId;
    private LocalDateTime dateTime;
    // Getters and Setters
}

// DTO to hold data needed to populate the form
public class FormPrerequisites {
    private List<MovieSummary> movies;
    private List<Screen> screens;
    // Getters
}
```

UC-15: Cancel Screening

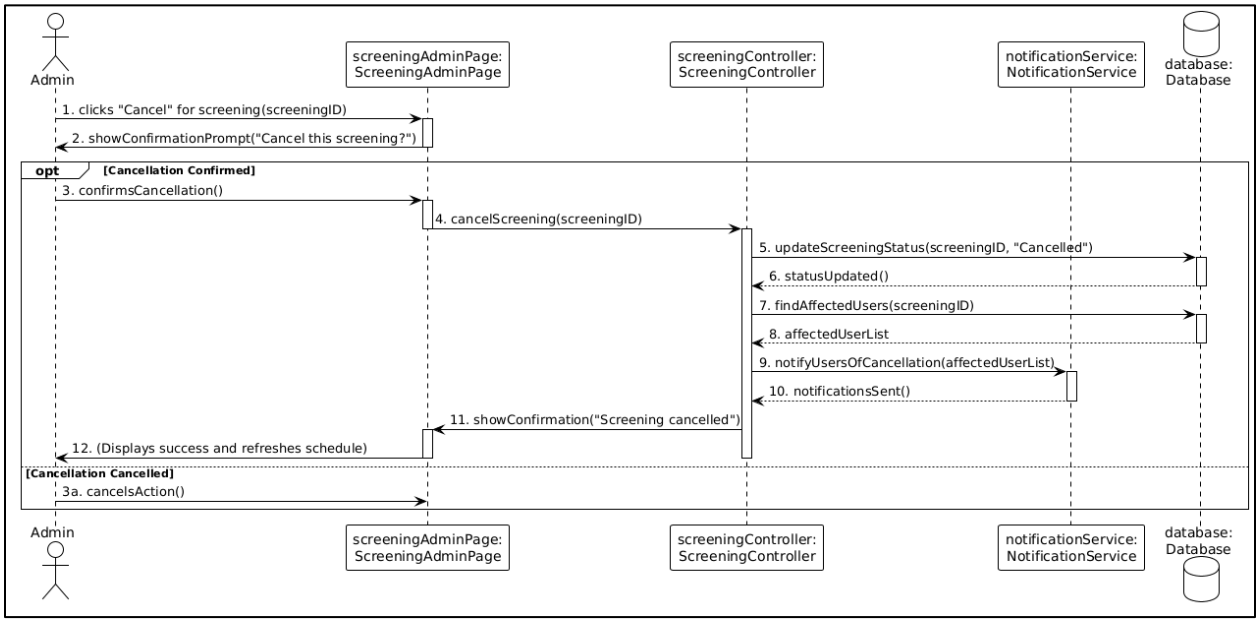
a. Structured use case specification:

Use Case: Cancel Screening
Scenario 1: Admin Successfully Cancels a Screening **Pre-condition:** Admin is logged in and viewing the screening schedule. 1. Admin selects the "Cancel" option for a specific upcoming screening. 2. System displays a confirmation prompt. 3. Admin confirms the cancellation. 4. System updates the screening's status to "Cancelled" in the database. 5. System identifies any users who have booked tickets

for this screening.
6. System triggers notifications (e.g., email) to affected customers and may initiate an automated refund process.
7. System displays a success message, "Screening has been cancelled," and refreshes the schedule.

Alternative 1.1: Cancellation is Aborted
At step 3, if the Admin cancels the action at the confirmation prompt.
No action is taken.

b. Sequence diagram:



c. Method Population in Classes:

ScreeningAdminPage
showConfirmationPrompt(message)
confirmsCancellation()
cancelsAction()
cancelScreening(screeningID)
showConfirmation(message)

ScreeningController
cancelScreening(screeningID)

NotificationService

notifyUsersOfCancellation(userList)

Database
updateScreeningStatus(id, status)
findAffectedUsers(screeningID)

d. Code snippets:

```
// View Class (Adding to ScreeningAdminPage from previous UC)
public class ScreeningAdminPage {
    private ScreeningController screeningController;
    private ScreeningID screeningToCancel;

    // ... methods from UC-14 ...

    // Triggered when admin clicks "Cancel" on a screening in the schedule
    public void onCancelClick(ScreeningID screeningId) {
        this.screeningToCancel = screeningId;
        showConfirmationPrompt("Are you sure you want to cancel this screening? This will notify
all booked customers.");
    }

    public void showConfirmationPrompt(String message) {
        System.out.println("CONFIRM: " + message + " [Yes/No]");
    }

    public void onConfirmCancellation() {
        if (screeningToCancel != null) {
            screeningController.cancelScreening(screeningToCancel);
        }
    }

    public void onCancelAction() {
        this.screeningToCancel = null;
        System.out.println("Cancellation aborted.");
    }

    public void showConfirmation(String message) {
        System.out.println("Success: " + message);
        // Refresh the schedule view
```

```
    }
}

// Controller Class (Adding to ScreeningController from previous UC)
public class ScreeningController {
    private Database database;
    private NotificationService notificationService;
    private ScreeningAdminPage view;

    // ... constructor and other methods ...

    public void cancelScreening(ScreeningID screeningId) {
        // Step 1: Update the screening status
        database.updateScreeningStatus(screeningId, "Cancelled");

        // Step 2: Find and notify affected users
        List<User> affectedUsers = database.findAffectedUsers(screeningId);
        if (!affectedUsers.isEmpty()) {
            notificationService.notifyUsersOfCancellation(affectedUsers);
            // In a real app, refund logic would be triggered here
        }

        view.showConfirmation("Screening has been cancelled.");
    }
}


// Service Class (Adding to NotificationService from UC-04)
public class NotificationService {
    // ... other methods ...

    public void notifyUsersOfCancellation(List<User> users) {
        System.out.println("Notifying " + users.size() + " users of cancellation...");
        for (User user : users) {
            // Logic to send email/SMS to each user
            System.out.println("  - Sending notification to " + user.getEmail());
        }
    }
}
```

```
// Database Class (New methods)
public class Database {
   // ... other methods ...

   public boolean updateScreeningStatus(ScreeningID screeningId, String status) {
      System.out.println("Updating screening " + screeningId + " to status '" + status + "'");
      // SQL UPDATE statement
      return true; // Placeholder
   }

   public List<User> findAffectedUsers(ScreeningID screeningId) {
      System.out.println("Finding users who booked for screening " + screeningId);
      // SQL query to join bookings and users tables for a given screeningId
      return new ArrayList<User>(); // Placeholder
   }
}
```

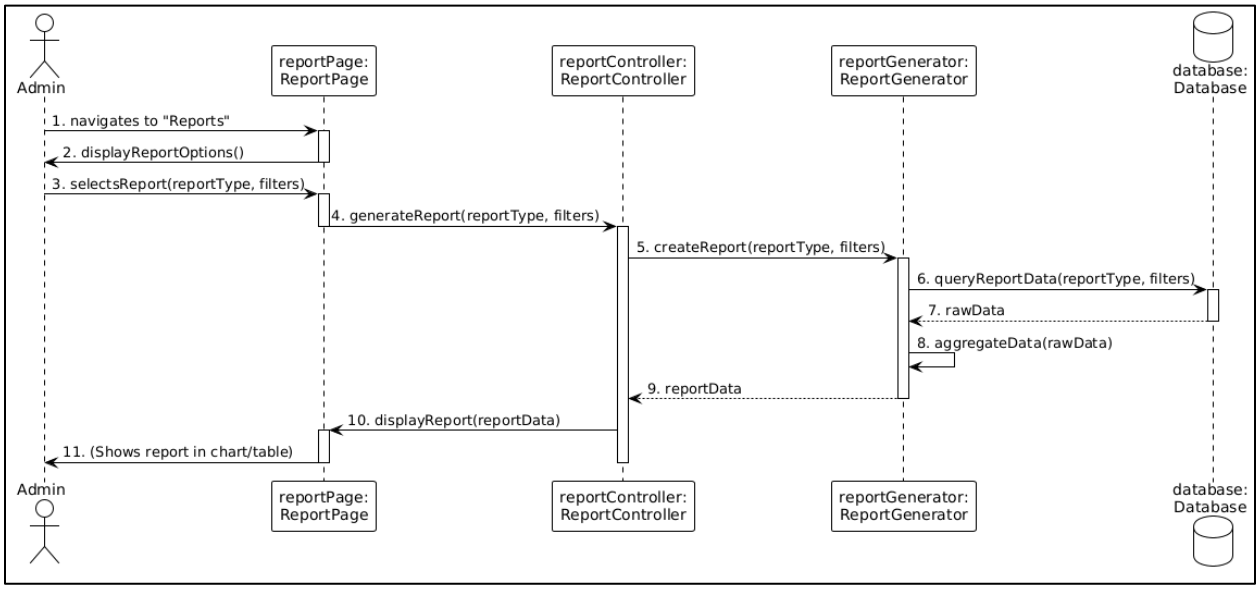
UC-16: View Report

a. Structured use case specification:

Use Case: View Report
Scenario 1: Admin Successfully Generates and Views a Report **Pre-condition:** Admin is logged into the administrative panel. 1. Admin navigates to the "Reports" section. 2. System displays a list of available report types (e.g., "Sales Report", "Booking Statistics", "Movie Performance"). 3. Admin selects a report type. 4. System presents options to filter the report (e.g., by a date range). 5. Admin applies filters and requests to generate the report. 6. System processes the request: queries the database, retrieves the relevant data, and aggregates it. 7. System displays the generated report in a suitable format (e.g., a chart, a table).

Alternative 1.1: No Data for Report
At step 6, if no data is found for the selected report type and filters.
System displays a message: "No data available for the selected criteria."

b. Sequence diagram:



c. Method Population in Classes:

ReportPage
displayReportOptions()
selectsReport(type, filters)
generateReport(type, filters)
displayReport(data)

ReportController
generateReport(type, filters)

ReportGenerator
createReport(type, filters)
queryReportData(type, filters)
aggregateData(rawData)

Database
queryReportData(type, filters)

d. Code snippets:

```
// View Class
public class ReportPage {
    private ReportController reportController;

    public void displayReportOptions() {
        System.out.println("Please select a report to generate:");
        System.out.println("1. Sales Report");
        System.out.println("2. Movie Performance");
    }

    // Triggered when admin selects options and clicks "Generate"
    public void onGenerateReport(ReportType type, ReportFilters filters) {
        reportController.generateReport(type, filters);
    }

    public void displayReport(ReportData data) {
        System.out.println("--- Report: " + data.getTitle() + " ---");
        if (data.isEmpty()) {
            System.out.println("No data available for the selected criteria.");
        } else {
            // Logic to render data as a chart or table
            System.out.println(data.getFormattedContent());
        }
    }
}

// Controller Class
public class ReportController {
    private ReportGenerator reportGenerator;
    private ReportPage view;

    public ReportController(ReportGenerator generator, ReportPage view) {
        this.reportGenerator = generator;
        this.view = view;
    }

    public void generateReport(ReportType type, ReportFilters filters) {
        ReportData reportData = reportGenerator.createReport(type, filters);
```

```
        view.displayReport(reportData);
    }
}
public class ReportGenerator {
    private Database database;

    public ReportGenerator(Database db) {
        this.database = db;
    }

    public ReportData createReport(ReportType type, ReportFilters filters) {
        List<RawData> rawData = database.queryReportData(type, filters);
        ReportData reportData = aggregateData(rawData, type);
        return reportData;
    }

    private ReportData aggregateData(List<RawData> rawData, ReportType type) {
        System.out.println("Aggregating raw data for " + type + " report...");
        return new ReportData("Generated Report", "Formatted content"); // Placeholder
    }
}

// Database Class (New method)
public class Database {
    // ... other methods ...

    public List<RawData> queryReportData(ReportType type, ReportFilters filters) {
        System.out.println("Querying database for report: " + type);
        // This method would have complex SQL queries based on the report type and filters
        // e.g., SELECT movie_title, COUNT(ticket_id) FROM bookings JOIN ... GROUP BY
movie_title
        return new ArrayList<RawData>(); // Placeholder
    }
}

// DTOs for reporting
public enum ReportType { SALES, PERFORMANCE }
public class ReportFilters { private Date startDate; private Date endDate; /* ... */ }
public class ReportData {
```

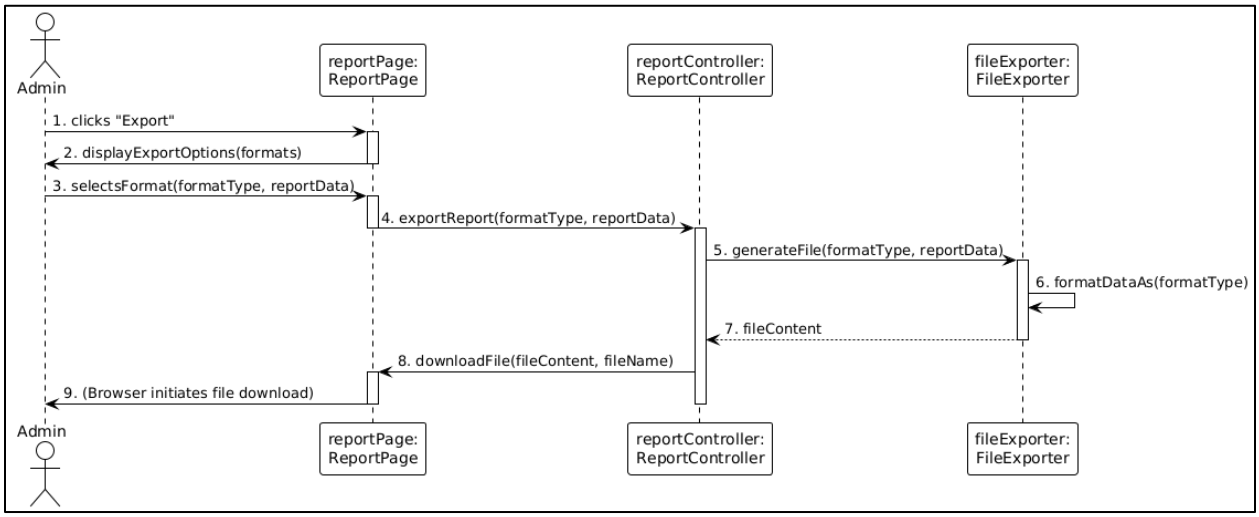
```
    private String title;
    private String formattedContent;
    // Constructor, getters, and isEmpty() method
}
```

UC-17: Export Report

a. Structured use case specification:

Use Case: Export Report
Scenario 1: Admin Successfully Exports a Report **Pre-condition:** Admin is currently viewing a generated report (UC-16). 1. Admin selects the "Export" option. 2. System presents a choice of file formats (e.g., CSV, PDF). 3. Admin chooses a file format. 4. System takes the currently displayed report data. 5. System generates a file in the selected format containing the report data. 6. System initiates a file download in the Admin's browser. *Alternative 1.1: Export Format Not Supported* At step 3, if an unsupported format is somehow requested. Message "Error: The selected export format is not supported."

b. Sequence diagram:



c. Method Population in Classes:

ReportPage
displayExportOptions(formats)
selectsFormat(type, data)
exportReport(type, data)
downloadFile(content, name)

ReportController
exportReport(type, data)

FileExporter
generateFile(type, data)
formatDataAs(type)

d. Code snippets:

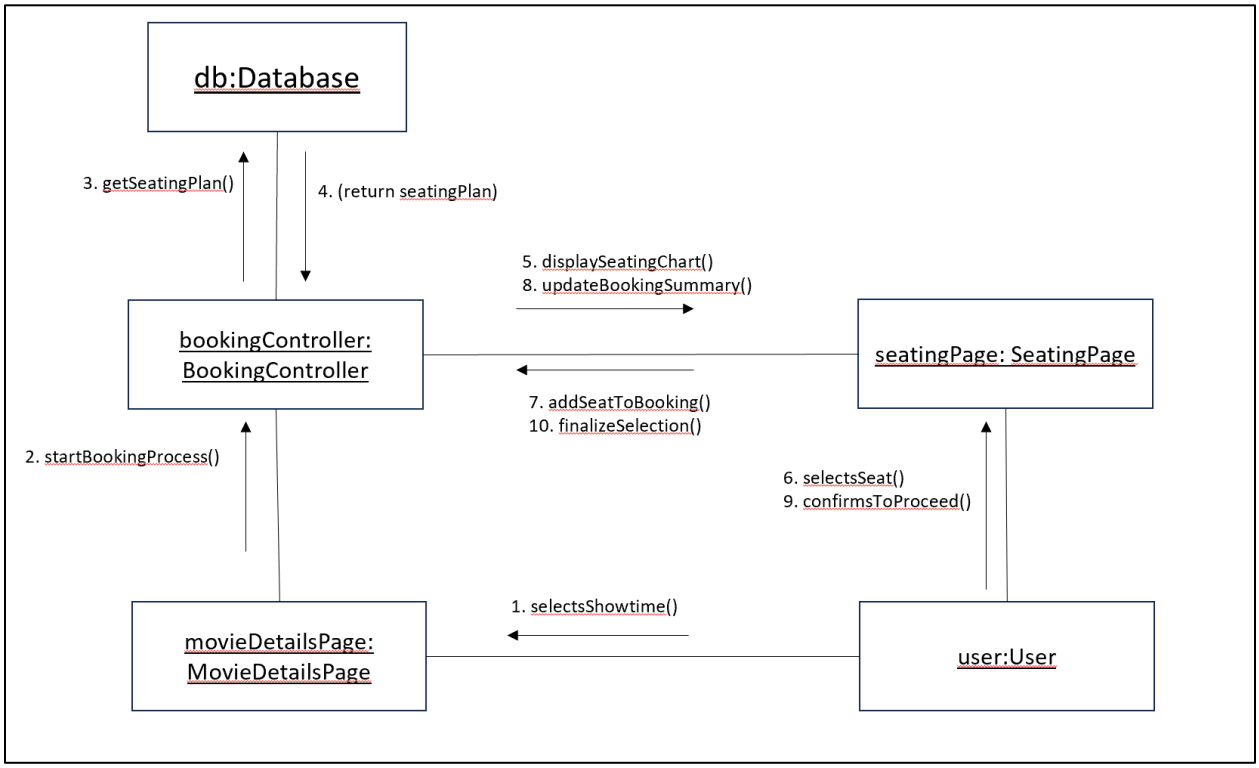
```
// View Class (Adding to ReportPage from previous UC)
public class ReportPage {
    private ReportController reportController;
    private ReportData currentReportData; // Assumes the data from the viewed report is stored

    // ... methods from UC-16 ...

    public void displayReport(ReportData data) {
        this.currentReportData = data;
        // ... display logic from previous UC ...
    }

    // Triggered when admin clicks "Export"
    public void onExportClick() {
        displayExportOptions(new String[]{"CSV", "PDF"});
    }

    public void displayExportOptions(String[] formats) {
        System.out.println("Select an export format: " + String.join(", ", formats));
    }

    // Triggered when admin selects a format
```

```
    public void onFormatSelected(ExportFormat format) {
        if (currentReportData != null) {
            reportController.exportReport(format, currentReportData);
        }
    }

    public void downloadFile(FileContent file, String fileName) {
        System.out.println("Initiating download for file: " + fileName);
        // Browser/framework-specific logic to trigger a file download
    }
}


// Controller Class (Adding to ReportController from previous UC)
public class ReportController {
    // ... fields from previous UC ...
    private FileExporter fileExporter;

    public ReportController(ReportGenerator generator, FileExporter exporter, ReportPage view)
{
        this.reportGenerator = generator;
        this.fileExporter = exporter;
        this.view = view;
    }

    // ... generateReport method from previous UC ...

    public void exportReport(ExportFormat format, ReportData data) {
        FileContent file = fileExporter.generateFile(format, data);
        String fileName = data.getTitle().replaceAll(" ", "_") + "." +
format.toString().toLowerCase();
        view.downloadFile(file, fileName);
    }
}


// Service class for file generation logic
public class FileExporter {
    public FileContent generateFile(ExportFormat format, ReportData data) {
```

```
        System.out.println("Generating " + format + " file for report: " + data.getTitle());
        String content;
        switch (format) {
            case CSV:
                // Logic to convert report data to comma-separated values
                content = "header1,header2\nvalue1,value2"; // Placeholder
                break;
            case PDF:
                // Logic to use a PDF library to generate a PDF document
                content = "[PDF Content Bytes]"; // Placeholder
                break;
            default:
                throw new UnsupportedOperationException("Format not supported");
        }
        return new FileContent(content.getBytes());
    }
}

// DTOs for exporting
public enum ExportFormat { CSV, PDF }
public class FileContent {
    private byte[] data;
    public FileContent(byte[] data) { this.data = data; }
    public byte[] getData() { return data; }
}
```

Collaboration Diagrams:

UC-08: Collaboration Diagram for Booking a Ticket



UC-11: Collaboration Diagram for Editing a Movie (Admin)