Movie Theater Startup: Detailed Design Document for Gift Card Feature

Objective

To implement a gift card feature enabling users to gift movie tickets with specified amounts, redeemable by recipients for any movie booking.

Requirements

- Intuitive user interface for buying and sending gift cards.
- Secure payment processing for gift card purchases.
- Multiple denominations for gift cards ($10, $25, $50, $100).
- Email/app notification system for gift card recipients.
- User-friendly redemption process.
- Gift cards with expiration dates (1 year from purchase).
- Real-time balance tracking for gift cards.
- Robust security measures for gift card data.

Out of Scope

1. **Physical Gift Cards**: Production and distribution of physical gift cards are not considered in this design. The focus is solely on digital gift cards.
2. **External Gift Card Vendors**: Integration with external gift card vendors or marketplaces is out of scope. All gift cards are managed internally.
3. **Currency Conversion**: Handling multi-currency transactions or conversions is not included. The feature will support the default currency of the platform.
4. **Advanced Fraud Detection**: While basic security measures are in place, advanced fraud detection mechanisms (like machine learning-based models) are not covered in this phase.
5. **Loyalty Program Integration**: Integration of gift cards with existing loyalty or rewards programs is not considered in the current scope but may be explored in future updates.
6. **Customizable Gift Card Designs**: Offering customizable designs or personalization options for digital gift cards is not included at this stage.

Background

The gift card feature is being introduced as an extension of our existing movie theater booking system. This system already has several key components in place, which will be leveraged for the gift card feature:

1. **Existing Payment Service**: The platform already has a robust Payment Service in place, handling transactions for movie ticket bookings. This service will be utilized for processing gift card purchases, ensuring secure and efficient payment handling without the need for additional payment infrastructure.

2. **User Authentication and Management**: Our platform's UserService, which manages user accounts, authentication, and profile information, will be integral to the gift card feature. It will help in identifying users for purchasing and receiving gift cards, maintaining a seamless user experience.

3. **Movie Booking System**: The core functionality of our platform, the MovieBookingService, already handles movie schedules, seat availability, and booking transactions. The gift card feature will integrate with this service to allow users to apply gift card balances towards movie bookings.

4. **Notification System**: An existing NotificationService, used primarily for booking confirmations and reminders, will be expanded to include notifications related to gift card purchases, redemptions, and expiration alerts.

High-Level Design (HLD)

Microservices Overview

1. **GiftCardService**:

    - **Responsibilities**: Manages gift card creation, redemption, balance checking, and expiration notifications.
    - **Database**: Owns a separate database (`gift_cards_db`) to store gift card information.
    - **Interactions**: Communicates with `PaymentService` for payment processing and `NotificationService` for sending out notifications.

2. **PaymentService**:

    - **Responsibilities**: Handles all payment transactions, including gift card purchases.
    - **Database**: Uses a separate database (`payments_db`) for tracking transactions.
    - **Interactions**: Communicates with `GiftCardService` to confirm payment completion.

3. **NotificationService**:

    - **Responsibilities**: Sends notifications (e.g., email, push) to users, particularly for gift card receipt and expiration alerts.
    - **Database**: May use a small database (`notifications_db`) for logging and managing notifications.
    - **Interactions**: Triggered by `GiftCardService` for sending out gift card-related notifications.

4. **UserService**:

    - **Responsibilities**: Manages user accounts, authentication, and profile information.
    - **Database**: Maintains a user-related database (`users_db`).
    - **Interactions**: Provides user information to `GiftCardService` for gift card issuance and receipt.

5. **MovieBookingService**:

    - **Responsibilities**: Handles movie ticket bookings and applies gift card balances during checkout.
    - **Database**: Has its own database (`bookings_db`) for managing movie bookings.
    - **Interactions**: Interacts with `GiftCardService` for applying gift card balances to bookings.

Microservices Interaction and Database Architecture

- Each microservice has its own dedicated database, ensuring data encapsulation and service independence. This separation aligns with the microservices architecture principle of decentralized data management.
- Services communicate with each other via RESTful APIs or message queues, depending on the requirement for synchronous or asynchronous processing.
- The `GiftCardService` is central to the gift card feature, coordinating with other services for payment processing, notifications, and applying gift card balances during movie bookings.

Handling Latency and Availability Issues

- **Caching**: Implement caching in services like `GiftCardService` and `MovieBookingService` to reduce database read operations, particularly for frequently accessed data like gift card balances and movie schedules.
- **Load Balancing**: Use load balancers to distribute incoming requests evenly across service instances, preventing overloading of any single instance.
- **Circuit Breakers**: Implement circuit breakers in service-to-service communication to prevent cascading failures. For example, if `PaymentService` is down, `GiftCardService` can temporarily halt gift card purchases.

- **Fallback Mechanisms**: Provide fallback options in case of service unavailability. For instance, if `NotificationService` fails, `GiftCardService` can queue the notifications and retry later.
- **Database Replication**: Use database replication techniques to improve data availability and reduce read latency.
- **Monitoring and Alerting**: Continuously monitor service health, performance metrics, and error rates. Set up alerting mechanisms to notify relevant teams in case of anomalies.

Example Scenario: Gift Card Purchase

1. User initiates a gift card purchase via the frontend, which sends a request to `GiftCardService`.
2. `GiftCardService` calls `PaymentService` to process the payment.
3. Once payment is confirmed, `GiftCardService` creates a new gift card record in `gift_cards_db`.
4. `GiftCardService` then requests `NotificationService` to send a confirmation to the recipient.
5. Throughout this process, each service handles its own data in its respective database.

Low-Level Design (LLD)

- **Java Classes**:
  - `GiftCard`: Represents a gift card (`id`, `balance`, `expirationDate`, `recipientEmail`, `purchaserId`).
  - `GiftCardController`: REST controller for handling gift card-related HTTP requests.
  - `GiftCardService`: Business logic for managing gift cards.
  - `GiftCardRepository`: Interface for database operations related to gift cards.

Java Classes and Implementations

GiftCard Class

```java
public class GiftCard {
    private UUID id;
    private BigDecimal balance;
    private LocalDate expirationDate;
    private String recipientEmail;
    private UUID purchaserId;
    private LocalDateTime creationDate;
    private LocalDateTime lastUpdated;
    // Constructor, getters, setters...
}
```

GiftCardController

```
@RestController
@RequestMapping("/giftcards")
public class GiftCardController {
    private final GiftCardService giftCardService;

    @PostMapping
    public ResponseEntity<GiftCardDetails> createGiftCard(@RequestBody GiftCardDto g
        // Implementation to create a gift card
    }

    @PostMapping("/redeem")
    public ResponseEntity<RedeemGiftCardResponse> redeemGiftCard(@RequestBody Redeem
        // Implementation to redeem a gift card
    }

    @GetMapping("/sent")
    public ResponseEntity<List<GiftCardDetails>> getGiftCardsSent(@RequestParam UUID
        // Implementation to retrieve gift cards sent by a user
    }

    @GetMapping("/received")
    public ResponseEntity<List<GiftCardDetails>> getGiftCardsReceived(@RequestParam
        // Implementation to retrieve gift cards received by a user
    }
}
```

GiftCardService

```
@Service
public class GiftCardService {
    private final GiftCardRepository giftCardRepository;

    public GiftCard createGiftCard(GiftCardDto giftCardDto) {
        // Logic to create and save a new GiftCard
    }

    public GiftCard redeemGiftCard(UUID giftCardId, UUID bookingId) {
        // Logic to redeem a gift card
    }
    public List<GiftCard> getGiftCardsSent(UUID purchaserId) {
        // Logic to fetch gift cards sent by a user
    }

    public List<GiftCard> getGiftCardsReceived(String recipientEmail) {
        // Logic to fetch gift cards received by a user
    }
}
```

Repository Layer

The repository layer abstracts the data access logic from the business logic of the application. In the context of our gift card system, repositories would interface with the database to perform CRUD (Create, Read, Update, Delete) operations. Here's an example of how the repository layer might be implemented for the `GiftCardService`:

GiftCardRepository Interface

```
public interface GiftCardRepository {
    GiftCard save(GiftCard giftCard);
    Optional<GiftCard> findById(UUID giftCardId);
    List<GiftCard> findByPurchaserId(UUID purchaserId);
    List<GiftCard> findByRecipientEmail(String recipientEmail);
    void updateBalance(UUID giftCardId, BigDecimal newBalance);
    // Other necessary methods...
}
```

GiftCardRepository Implementation

In the implementation (e.g., using Spring Data JPA), you'd define the interaction with the database, handling queries, and transactions.

Exception Handling

Effective exception handling ensures the system remains robust and user-friendly, even when encountering errors. In a microservice architecture, you should consider:

1. **Service-Level Exceptions**: Handle exceptions that occur within a service. For instance, if a payment fails in `PaymentService`, it should throw an appropriate exception that can be handled by `GiftCardService`.

2. **Inter-Service Communication Failures**: Handle failures in communication between services gracefully. Use patterns like Circuit Breaker to prevent cascading failures.

3. **Data Access Exceptions**: Handle exceptions related to data access in the repository layer. For example, handle `SQLExceptions` or `DataAccessExceptions` and translate them into service-level exceptions.

4. **Client-Friendly Error Responses**: Transform exceptions into user-friendly error messages that can be returned to the client. Avoid exposing sensitive error details.

5. **Logging and Monitoring**: Log exceptions for monitoring and debugging purposes. Integrate with tools like ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging.

6. **Fallback Mechanisms**: Implement fallback mechanisms for handling failures gracefully. For example, in case of a failure in `NotificationService`, queue the notifications and attempt to resend them later.

7. **Validation Exceptions**: Validate input data and handle validation exceptions by sending appropriate error responses to the client.

Example of Exception Handling in `GiftCardService`

```
public GiftCard createGiftCard(GiftCardDto giftCardDto) {
    try {
        validateGiftCardDto(giftCardDto);
        // Business logic to create a gift card...
    } catch (DataValidationException e) {
        // Handle validation exceptions
        throw new GiftCardCreationException("Invalid gift card data", e);
    } catch (DataAccessException e) {
        // Handle database exceptions
        log.error("Database access error during gift card creation", e);
        throw new ServiceException("Error creating gift card", e);
    } catch (Exception e) {
        // Generic exception handling
        log.error("Unexpected error during gift card creation", e);
        throw new ServiceException("Unexpected error", e);
    }
}
```

Expanded API Collection

1. Purchase Gift Card API

- **Endpoint**: `POST /giftcards/purchase`

- **Request**:

  ```
  {
    "purchaserId": "uuid-of-purchaser",
    "amount": 50.00,
    "recipientEmail": "recipient@example.com"
  }
  ```

- **Response**:

  ```
  {
    "giftCardId": "uuid-of-giftcard",
    "status": "success",
    "message": "Gift card successfully purchased."
  }
  ```

2. View Gift Cards Sent by a User

- **Endpoint**: `GET /giftcards/sent/{purchaserId}`
- **Response**:

  ```
  {
    "giftCards": [
      {
        "giftCardId": "uuid-of-giftcard-1",
        "amount": 50.00,
        "recipientEmail": "recipient1@example.com",
        "expirationDate": "2024-01-01"
      },
      {
        "giftCardId": "uuid-of-giftcard-2",
        "amount": 25.00,
        "recipientEmail": "recipient2@example.com",
        "expirationDate": "2024-06-01"
      }
    ]
  }
  ```

3. View Gift Cards Received by a User

- **Endpoint**: `GET /giftcards/received/{recipientEmail}`
- **Response**:

```
{
  "giftCards": [
    {
      "giftCardId": "uuid-of-giftcard-1",
      "amount": 30.00,
      "purchaserId": "uuid-of-purchaser-1",
      "expirationDate": "2024-01-01"
    },
    {
      "giftCardId": "uuid-of-giftcard-2",
      "amount": 20.00,
      "purchaserId": "uuid-of-purchaser-2",
      "expirationDate": "2023-12-01"
    }
  ]
}
```

4. Redeem Gift Card API

- **Endpoint**: `POST /giftcards/redeem`
- **Request**:

```
{
  "giftCardId": "uuid-of-giftcard",
  "bookingId": "uuid-of-booking"
}
```

- **Response**:

```
{
  "status": "success",
  "message": "Gift card successfully redeemed.",
  "remainingBalance": 20.00
}
```

5. Check Gift Card Balance

- **Endpoint**: `GET /giftcards/balance/{giftCardId}`
- **Response**:

```
{
  "giftCardId": "uuid-of-giftcard",
  "currentBalance": 25.00
}
```

6. Gift Card Expiration Notification

- **Endpoint**: `GET /giftcards/expiration-notice`
- **Response**:

```
{
  "notifiedGiftCards": [
    {
      "giftCardId": "uuid-of-expiring-giftcard-1",
      "recipientEmail": "recipient1@example.com",
      "expirationDate": "2023-05-01"
    },
    {
      "giftCardId": "uuid-of-expiring-giftcard-2",
      "recipientEmail": "recipient2@example.com",
      "expirationDate": "2023-05-15"
    }
  ]
}
```

Sequence Diagram for Issuing a Gift Card

1. **Actors and Objects**:
    - User (Actor)
    - GiftCardController
    - GiftCardService
    - GiftCardRepository
    - PaymentService
    - NotificationService
2. **Flow of Events for Issuing a Gift Card**:
    - The User initiates a request to issue a gift card via the GiftCardController.
    - GiftCardController calls the GiftCardService to process the request.
    - GiftCardService interacts with PaymentService to handle the payment transaction.
    - Once the payment is successful, GiftCardService requests GiftCardRepository to create a new gift card record.
    - GiftCardRepository saves the gift card details in the database.

- GiftCardService then triggers NotificationService to send a notification to the recipient.
- NotificationService sends an email or app notification to the recipient.
- Finally, GiftCardService returns the details of the issued gift card to GiftCardController, which then sends a response back to the User.

Algorithm for Redeeming a Gift Card

1. **Actors and Objects**:
    - User (Actor)
    - GiftCardController
    - GiftCardService
    - MovieBookingService
    - GiftCardRepository
2. **Flow of Events for Redeeming a Gift Card**:
    - The User sends a request to redeem a gift card for a movie booking through GiftCardController.
    - GiftCardController forwards the request to GiftCardService.
    - GiftCardService validates the gift card details and checks the balance.
    - If valid, GiftCardService interacts with MovieBookingService to apply the gift card to the movie booking.
    - MovieBookingService processes the booking and returns the status to GiftCardService.
    - GiftCardService updates the gift card balance in GiftCardRepository.
    - GiftCardRepository updates the gift card record in the database.
    - GiftCardService sends a confirmation of the redemption to GiftCardController.
    - GiftCardController responds to the User with the redemption status.

flow diagram for the gift card issuance process:

```
+-------+      +------------------+      +---------------+
|       |      |                  |      |               |
|  User +----->+ GiftCardController +---->+ GiftCardService |
|       |      |                  |      |               |
+-------+      +------------------+      +-------+-------+
                                                 |
                                                 v
                                           +-------+-------+
                                           |               |
                                           | PaymentService |
                                           |               |
                                           +-------+-------+
                                                 |
                                                    v
                                        +-----------+----------+
                                        |                      |
                                        | GiftCardRepository   |
                                        |                      |
                                        +-----------+----------+
                                                    |
                                                    v
                                        +-----------+-----------+
                                        |                       |
                                        |  NotificationService  |
                                        |                       |
                                        +-----------------------+
```

And for the gift card redemption process:

```
+--------+      +-------------------+      +----------------+
|        |      |                   |      |                |
|  User  +------>+ GiftCardController +---->+ GiftCardService |
|        |      |                   |      |                |
+--------+      +-------------------+      +-------+--------+
                                                    |
                                                    v
                                           +-------+--------+
                                           |                |
                                           | MovieBookingService |
                                           |                |
                                           +--------+-------+
                                                    |
                                                    v
                                        +-----------+----------+
                                        |                      |
                                        | GiftCardRepository   |
                                        |                      |
                                        +----------------------+
```

Database Schema

- SQL script for creating `gift_cards` table:

```sql
CREATE TABLE gift_cards (
    id UUID PRIMARY KEY,
    balance DECIMAL(10, 2),
    expiration_date DATE,
    recipient_email VARCHAR(255),
    purchaser_id UUID,
    creation_date TIMESTAMP,
    last_updated TIMESTAMP
);
```

- **Database Schema**:
  - Table `gift_cards`:
    - `id` (Primary Key, UUID)
    - `balance` (Decimal)
    - `expiration_date` (Date)
    - `recipient_email` (String)
    - `purchaser_id` (Foreign Key, UUID from users table)
    - `creation_date` (Timestamp)
    - `last_updated` (Timestamp)
- **Code Changes**:
  - Add new endpoints in `GiftCardController`:
    - `createGiftCard(GiftCardDto giftCardDto)`: POST endpoint to create a new gift card.
    - `redeemGiftCard(String giftCardId, String bookingId)`: POST endpoint to redeem a gift card.

Database Changes

- Creation script for `gift_cards` table with appropriate indexes for faster queries.

1. Creating a New Gift Card

```
INSERT INTO gift_cards (id, balance, expiration_date, recipient_email, purchaser_id,
VALUES (UUID(), 50.00, '2024-01-01', 'recipient@example.com', 'purchaser-uuid', CURR
```

2. Redeeming a Gift Card

- **Updating the Balance**:

  ```
  UPDATE gift_cards
  SET balance = balance - redeemed_amount, last_updated = CURRENT_TIMESTAMP
  WHERE id = 'giftcard-uuid' AND balance >= redeemed_amount;
  ```

- **Checking the Remaining Balance**:

```
SELECT balance
FROM gift_cards
WHERE id = 'giftcard-uuid';
```

3. Viewing Gift Cards Sent by a User

```
SELECT id, balance, expiration_date, recipient_email
FROM gift_cards
WHERE purchaser_id = 'purchaser-uuid';
```

4. Viewing Gift Cards Received by a User

```
SELECT id, balance, expiration_date, purchaser_id
FROM gift_cards
WHERE recipient_email = 'recipient@example.com';
```

5. Checking Gift Card Balance

```
SELECT balance
FROM gift_cards
WHERE id = 'giftcard-uuid';
```

6. Fetching Expiring Gift Cards for Notification

```
SELECT id, recipient_email, expiration_date
FROM gift_cards
WHERE expiration_date BETWEEN CURRENT_DATE AND DATE_ADD(CURRENT_DATE, INTERVAL 30 DA
```

Queries and Corresponding Indexes

1. **Creating a New Gift Card**:
   - **Query**:

     ```
     INSERT INTO gift_cards (id, balance, expiration_date, recipient_email, pur
     VALUES (?, ?, ?, ?, ?, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
     ```

   - **Index**: No additional index is needed for insert operations.
2. **Redeeming a Gift Card**:

- **Update Balance Query**:

  ```
  UPDATE gift_cards
  SET balance = balance - ?, last_updated = CURRENT_TIMESTAMP
  WHERE id = ? AND balance >= ?;
  ```

- **Check Balance Query**:

  ```
  SELECT balance
  FROM gift_cards
  WHERE id = ?;
  ```

- **Index**:
  - Primary index on `id` (usually default for primary key).
  - Consider an additional index on `balance` if balance-related queries are frequent.

3. **Viewing Gift Cards Sent by a User**:

   - **Query**:

     ```
     SELECT id, balance, expiration_date, recipient_email
     FROM gift_cards
     WHERE purchaser_id = ?;
     ```

   - **Index**:
     - Index on `purchaser_id` for faster lookups based on the purchaser.

4. **Viewing Gift Cards Received by a User**:

   - **Query**:

     ```
     SELECT id, balance, expiration_date, purchaser_id
     FROM gift_cards
     WHERE recipient_email = ?;
     ```

   - **Index**:
     - Index on `recipient_email` for efficient querying based on the recipient.

5. **Checking Gift Card Balance**:

   - **Query**:

     ```
     SELECT balance
     FROM gift_cards
     WHERE id = ?;
     ```

   - **Index**: Already covered under the primary index on `id`.

6. **Fetching Expiring Gift Cards for Notification**:

    - **Query**:

      ```sql
      SELECT id, recipient_email, expiration_date
      FROM gift_cards
      WHERE expiration_date BETWEEN ? AND ?;
      ```

    - **Index**:
        - Index on `expiration_date` for efficient range queries.

Indexes Summary:

1. **Primary Key Index on `id`**: This is the default index created for the primary key of the table.
2. **Index on `purchaser_id`**: Optimizes queries to find all gift cards sent by a specific user.
3. **Index on `recipient_email`**: Speeds up queries to find all gift cards received by a specific user.
4. **Index on `expiration_date`**: Useful for queries involving expiration date ranges, especially for sending notifications.

Performance & System Health

- Monitoring API response times, especially during high-traffic periods.
- Database performance metrics: query execution time, index efficiency.

Launch Plan

- Initial release to a small user group for beta testing.
- Gradual rollout to the wider audience over two weeks.
- Rollback plan in case of critical issues.

Metrics & Monitoring

Success Metrics

Success metrics help in assessing the positive impact of the feature.

1. **Increase in Total Sales**: Measure the percentage increase in overall ticket sales due to the usage of gift cards.
2. **Number of Gift Cards Sold**: Track the total number of gift cards sold over a given period.
3. **Redemption Rate**: The percentage of gift cards redeemed compared to those sold.

Monitoring: Success metrics are typically tracked using business analytics tools. For website analytics, libraries like ReactGA (Google Analytics for React) can be used to track events like gift card purchases and redemptions.

Regression Metrics

Regression metrics alert us to any negative impacts the new feature might have.

1. **Checkout Abandonment Rate**: Any increase in the abandonment rate of the ticket purchasing process after introducing gift cards.
2. **Page Load Time**: Monitoring if the introduction of new gift card-related pages affects the overall load time of the website.
3. **Error Rates**: Increase in errors or issues during the gift card purchase or redemption process.

Monitoring: For web applications, tools like Sentry can be integrated for real-time error tracking and alerts. For performance, React-based applications can utilize libraries like `react-perf-devtool` to monitor rendering performance and diagnose potential issues.

Usage Metrics

Usage metrics provide insights into how the feature is being used.

1. **User Engagement with Gift Card Feature**: Track how many users interact with the gift card feature, including viewing, purchasing, and redeeming.
2. **Average Gift Card Value**: The average value of gift cards purchased.
3. **Frequency of Gift Card Purchases**: How often users purchase gift cards.

Monitoring: To monitor interactions on a React-based frontend, libraries like `react-event-tracker` can be used to track and send data on specific user actions related to the gift card feature. This data can then be analyzed to gain insights into user behavior.

Implementation in a React Application

- For tracking events (like clicks on the ‘Buy Gift Card’ button), ReactGA can be used. It can send data to Google Analytics, which then provides insights and trends.
- To catch and report errors, especially those affecting the user experience, Sentry can be integrated into the React app. Sentry provides real-time error tracking and is valuable for

identifying issues in production.
- `react-perf-devtool` can be integrated into the development environment to monitor and analyze the performance of React components, helping to identify any performance regressions introduced by new features.

Example Usage

```
import ReactGA from 'react-ga';
import * as Sentry from '@sentry/react';

// Initialize ReactGA and Sentry
ReactGA.initialize('YOUR_GOOGLE_ANALYTICS_TRACKING_ID');
Sentry.init({ dsn: 'YOUR_SENTRY_DSN' });

// Example of tracking a gift card purchase
ReactGA.event({
  category: 'Gift Card',
  action: 'Purchase',
  label: 'Gift Card Purchase'
});

// Example of Sentry capturing an exception
try {
  // Code that might throw an exception
} catch (error) {
  Sentry.captureException(error);
}
```

Server-Driven Configurations

- Feature flags for enabling/disabling the gift card feature.

Testing Plan

- Automated unit tests for new classes and methods.
- Integration tests covering the entire flow of purchasing and redeeming gift cards.
- Load testing to simulate high usage scenarios.

Work Estimates

- GiftCardService development: 2 weeks.
- UI Integration: 1 week.
- Testing and QA: 1 week.
- Total estimate: 4 weeks.

Future Work

- Adding customization options for gift cards (designs, messages).
- Integration with loyalty programs for additional benefits.