Final Project MPCS 51050 - Event Market

Fernando Rocha Urbano

April 2024

1 General Topic

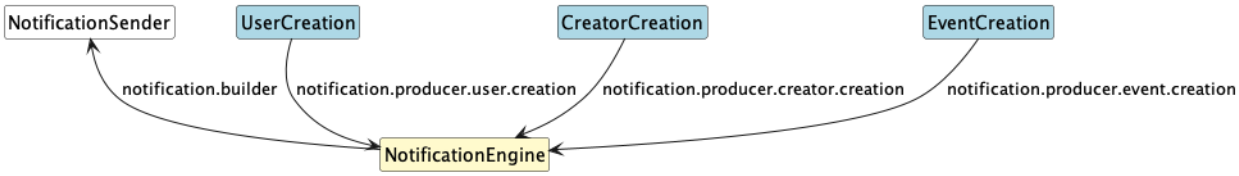
We aim to build an Event Ticket Purchase Platform, including event creation, purchase app, notification system, payment validation, event recommender engine.

In order to facilitate the understanding we highlight the following with specific formatting:

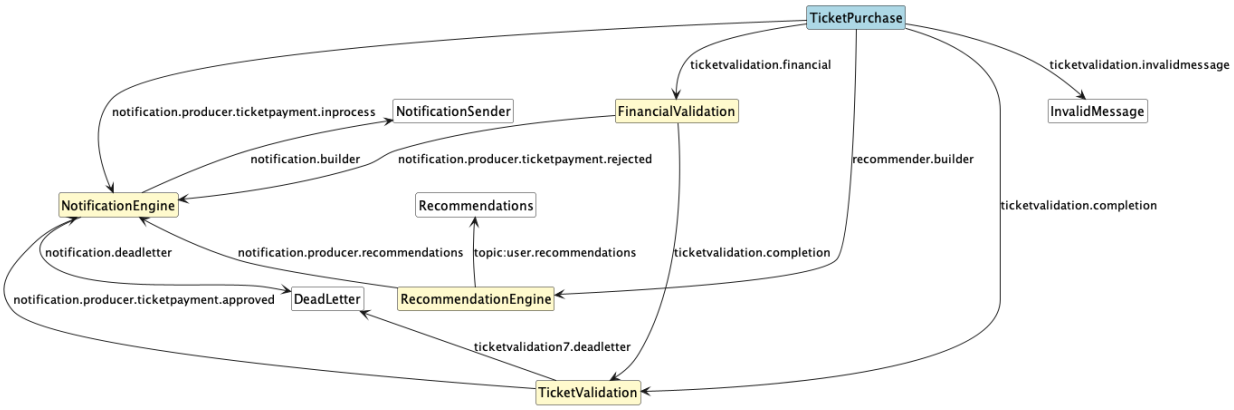
- **Application Names**
- ***Design Patterns***
- *Classes*

2 Project Separation

2.1 Creation Processes



2.2 Ticket Purchase Processes



3 Applications

- Main Application with Database connection:
  - **Event Market: event-market**
- Creation Camel Applications:
  - **Event Creation: event-market-event-creation**
  - **User Creation: event-market-user-creation**
  - **Creator Creation: event-market-event-creation**
- Ticket Purchase Camel Applications:
  - **Ticket Purchase: event-market-ticket-purchase**
  - **Financial Validation: event-market-financial-validation**
  - **Ticket Validation: event-market-ticket-validation**
  - **Event Recommender Engine: event-market-recommendation-engine**
  - **Notification Engine: event-market-notification-engine**

4 Design Patterns

4.1 Iterator

- Inside the **Ticket Purchase** app.
- Iterator pattern is used inside the *TicketTypeIterator* class to iterate over the ticket types of an Event.
- Provides a convenient way for users to browse through various ticket options, helping them understand the minimum age, prices, and types of tickets available for an event.

4.2 Singleton

- Inside the **Financial Validation** app.
- The *FinancialValidator* class is a Singleton, ensuring that a single instance validates ticket requests and manages their statuses.
- The Singleton pattern allows the *FinancialValidator* to track which requests it has validated recently, avoiding redundant checks and improving efficiency.

4.3 Strategy

- Inside the **Financial Validation** app.
- The *PaymentMethod* class acts as a Strategy pattern, providing an interface implemented by different payment classes such as *CheckingsAccountPayment* and *CardPayment*.
- This pattern enables the *Financial Validation* app to support various payment methods flexibly, facilitating future additions or modifications by simply extending the interface.

4.4 Facade

- Inside the **Event Recommender Engine**.
- The *EventRecommender* class acts as a Facade, simplifying the complexity of underlying classes derived from the *AbstractRecommenderModel*.
- This pattern allows for interactions with different recommendation models, making it easier to generate recommendations for users based on recent ticket requests.

4.5 Template Method

- Inside the **Event Recommender Engine**.
- The Template Method pattern is used by *AbstractRecommenderModel* and its derived classes, implementing some functionalities while leaving the heavy lifting to derived classes.
- This pattern provides a clear structure for creating different recommendation models, allowing derived classes to implement the specific logic needed to generate suggestions for users.

5 Applications and UML Drawings

5.1 Event Creation

App creates a new instances of *Event*.

The *Event* class has:

- at least one *TicketType* instance: for instance, premium, normal, free, etc... which specifies the price, minimum age, total tickets for the type.
- one *Location* instance.
- one *Creator* (each *Creator* can have multiple events).

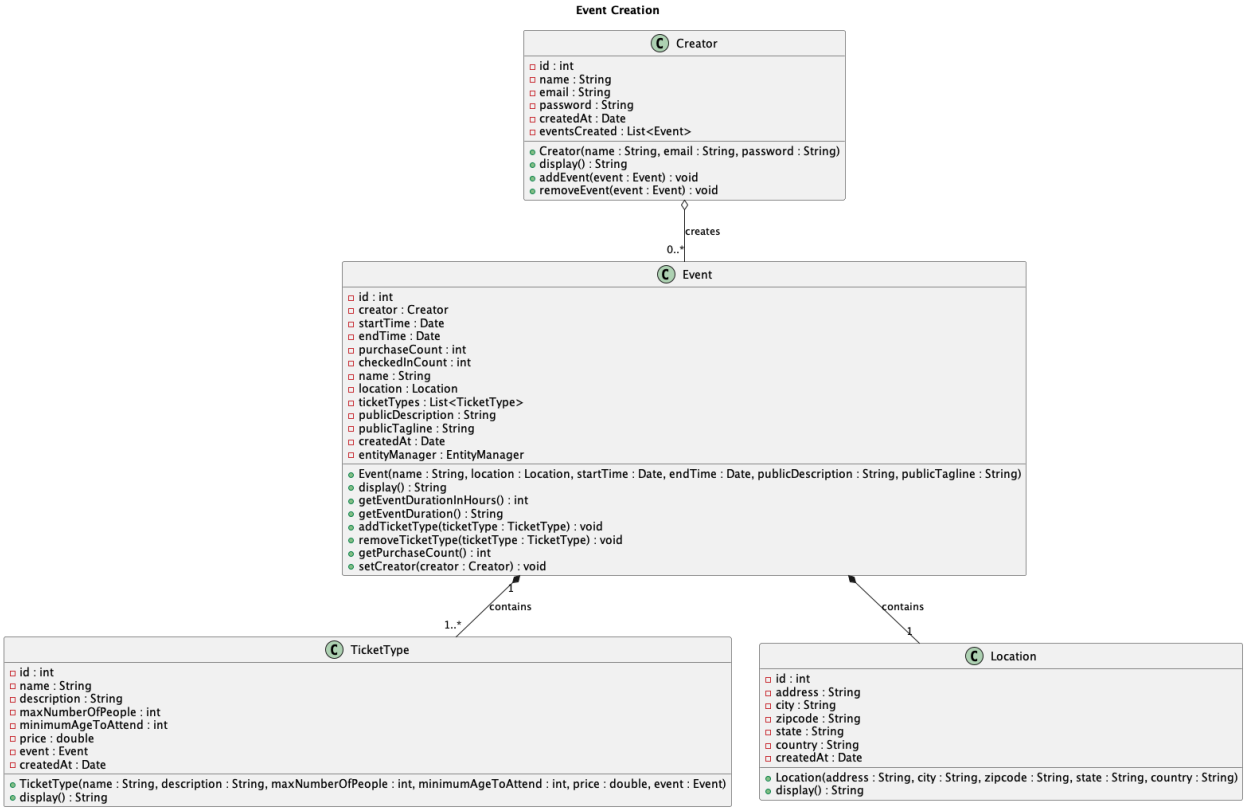
THe application communicates with **Notification Engine**.

5.2 User Creation

Similar to the creation of Event. THe application communicates with **Notification Engine**.

5.3 Creator Creation

Similar to the creation of Event. THe application communicates with **Notification Engine**.



5.4 Ticket Purchase

App allows purchase of tickets from *User*.

The *TicketRequest* is the main class in this process.

Every *TicketRequest* instance has an instance of:

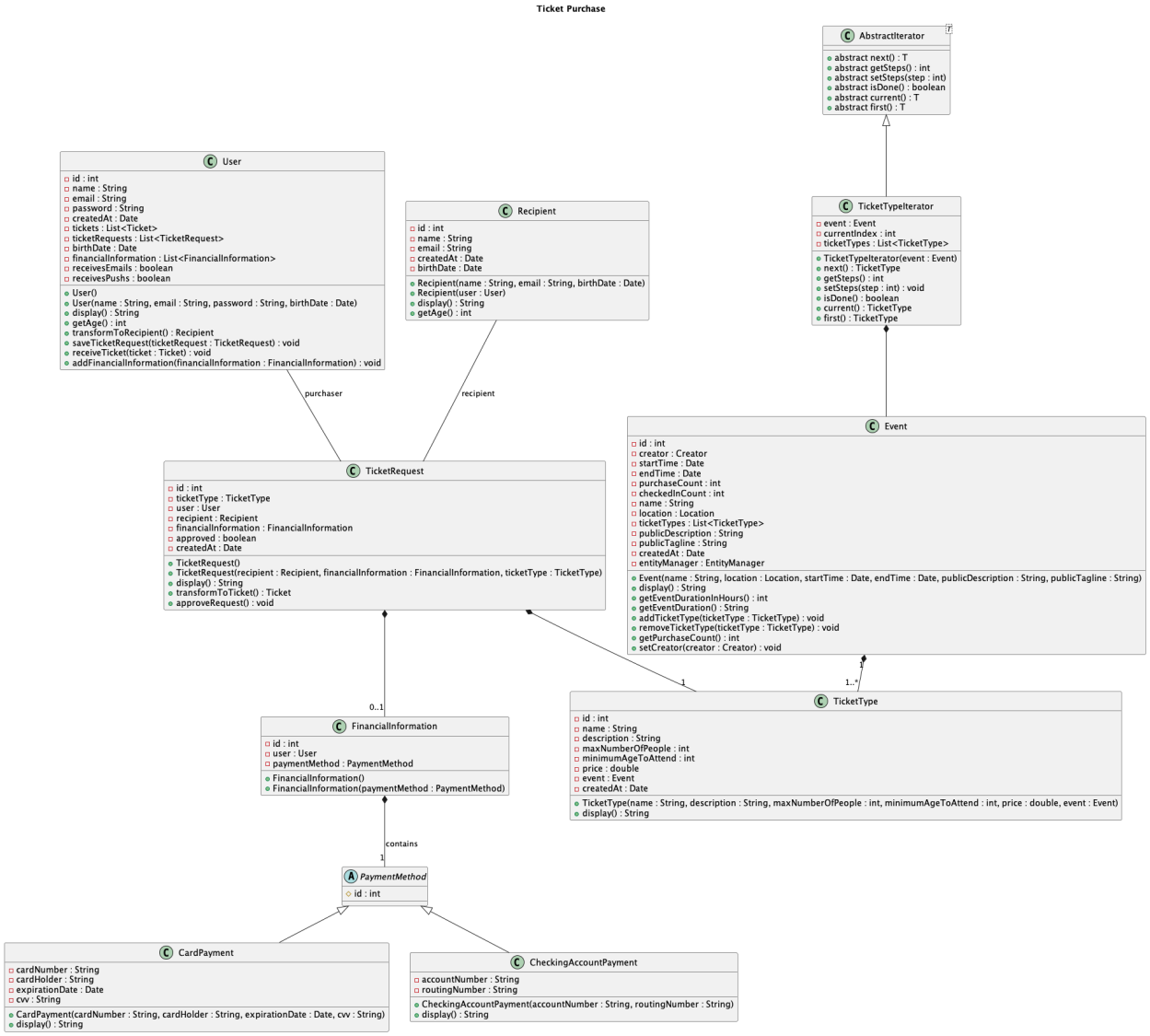
- *TicketType*: the type of ticket the user is buying. A single event allows for multiple *TicketType*'s with different prices, descriptions, etc. A *TicketRequest* instance only has one *TicketType* instance among the ones available for the event.
- *Financial Information*: the information regarding the payment of the ticket. Each instance of it has one instance of a class derived from *PaymentMethod*, which can be *CheckingsAccountPayment* or *CardPayment*. In case the ticket price is 0, the *FinancialInformation* is not necessary.
- *User*: the person purchasing the ticket.
- *Recipient*: the person who will enter the event. An instance of class *User* can be transformed into an instance of the class *Recipient* in case the user is trying to buy a ticket for himself/herself.

Once the *TicketRequest* is completed, it is sent to:

- the **Financial Validation** in case a payment is necessary.
- the **Ticket Validation** in case no payment is necessary.

- the **Event Recommender Engine** regardless of payment options.
- the **Notification Engine** to say that the ticket is being processed.

Inside the **Ticket Purchase** we use the ***Iterator*** pattern with *TicketTypeIterator*, which receives an *Event* and iterates over the ticket types. The iterator is considerably useful for the *User* to understand the minimum age among all the ticket types, the minimum and maximum price of the event, etc...



5.5 Financial Validation

App gets messages from the distributed message queue saying that a *TicketRequest* needs payment validation. Inside the app, The *TicketRequest* is sent to the *FinancialValidator* which ask for a payment process.

The *FinancialValidator* is a ***Singleton*** because it aims to track which instances of *TicketRequest* it has

checked recently: this allows the *FinancialValidator* to avoid wasting time checking twice the same *TicketRequest*.

The *TicketRequest* instance has an instance of *FinancialInformation*, which has one instance of a class derived from *PaymentMethod*.

The available classes are:

- *CardPayment*
- *CheckingsAccountPayment*

PaymentMethod is a ***Strategy*** pattern.

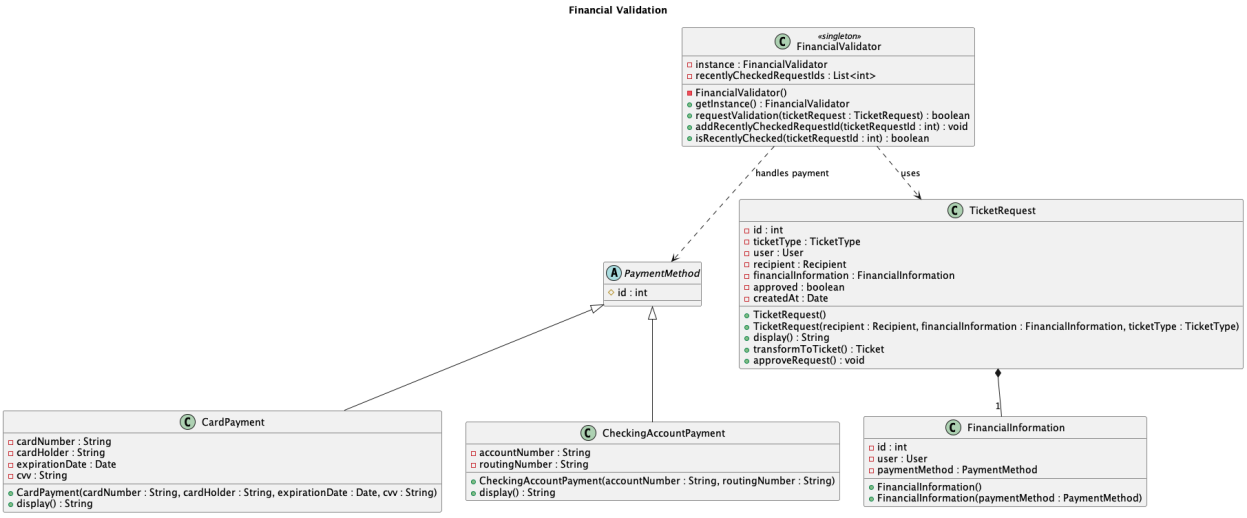
The *FinancialValidator* must be able to handle both classes.

Therefore, if there is an addition/change in those classes, the *FinancialValidator* must be changed/incremented as well.

Finally, the *FinancialValidator* sends a request to the desired bank to approve the payment (here we simplified the implementation and just randomly approve or reject the payment).

The application sends message to the

- the **Ticket Validation** in case the payment is accepted.
- the **Notification Engine** in case the payment is rejected.



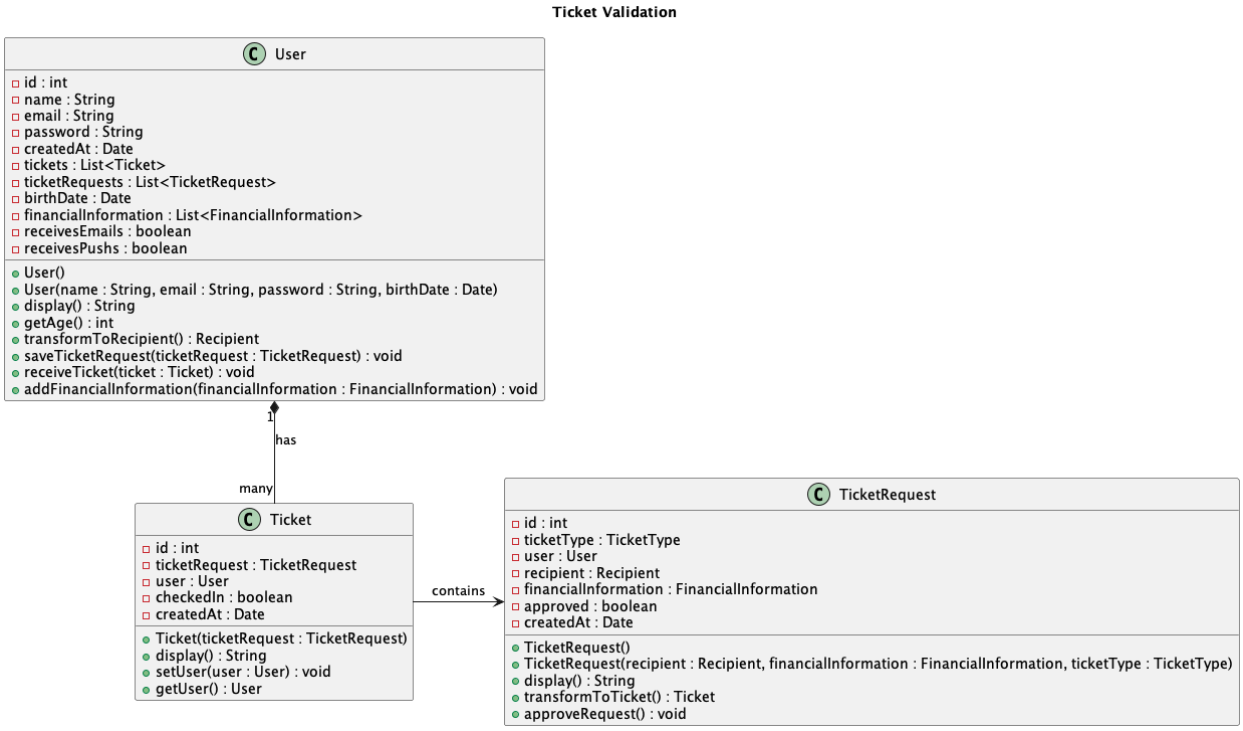
5.6 Ticket Validation

App gets messages from the distributed message queue saying that the ticket for that *User* has been approved.

The *Ticket* instance is created with a *TicketRequest* instance inside. The *TicketRequest* contains all the information about the *Event* inside it.

The *Ticket* instance is added to the *User*. More specifically, the *Ticket* instance is added to a list of owned tickets and remove the instance of *TicketRequest* from the list of requested tickets.

Finally, a message is sent to the **Notification Engine**.



5.7 Event Recommender Engine

App creates recommendations to the *User* based on recent purchases.

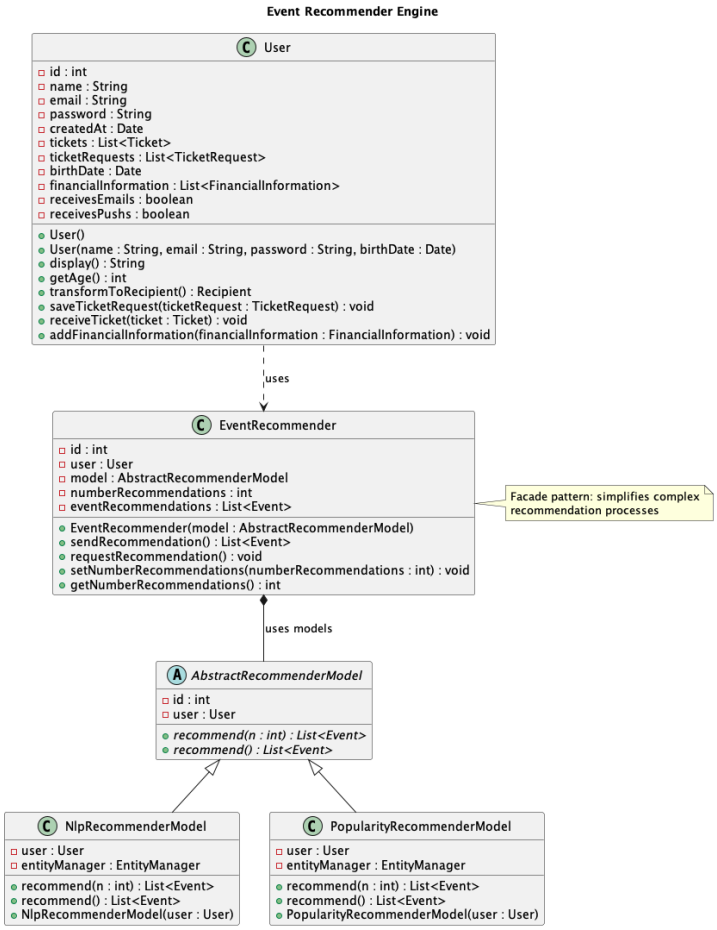
The event recommender engine receives a message everytime a *User* uses the **Ticket Purchase** application.

The *EventRecommender* class takes *User* and a model name (the one that is currently being used) and creates a recommendation for the user based on recent ticket requests.

The *EventRecommender* works as a ***Facade*** as it simplifies the underlying complexity of classes derived from the *AbstractRecommenderModel*, like *NlpRecommenderModel* and *PopularityRecommenderModel*.

We have a ***Template Method*** with the *AbstractRecommenderModel* and its derived classes. The *AbstractRecommenderModel* implements some of the functionalities of the model, but leaves the heavy lifting of creating suggestions for the derived classes.

It sends messages to the **Notification Engine**.



5.8 Notification Engine

The app translates internal information of the system into proper notifications for the users, allowing the market department to modify the content to better address the *User* or *Creator*.

It sends message to the Notification Sender (which could later just send the notification to the user or creator.)

6 EIP Patterns

1. **Message Channel and Endpoint:** How we connect an application to the channel. It is used in every single application.
2. **Content-Based Router:** Routes each message to the correct recipient based on message content.
3. **Dead-Letter Channel:** The messaging system determines that it cannot or should not deliver a message. It is used (in our case and generally) for messages that cannot be processed successfully after a certain amount of tries.
4. **Invalid-Message Channel:** Channel for messages that are malformed or fail validation checks and cannot be processed as expected.

5. **Message Translator:** Used between other filters or applications to translate one data format into another.
6. **Wire Tap:** Inserts a simple Recipient List into the channel that publishes each incoming message to the main channel and a secondary channel.
7. **Normalizer:** Routes each message type through a custom Message Translator so that the resulting messages match a common format.
8. **Content Enricher:** Access an external data source in order to augment a message with missing information.
9. **Multicast:** Allows a message to be sent to multiple recipients or destinations simultaneously (not in the book, but included in Camel: https://camel.apache.org/components/4.4.x/eips/multicast-eip.html).

6.1 Event-Market Applications

6.2 event-market-user-creation: User Creation Producer

1. **Message Channel**

```
to("jms:queue:notification.producer.user.creation.queue")
```

Listing 1: Message Channel for User Creation

2. **Point-to-Point Channel:** Inherited and implemented by this type of queue.

6.3 event-market-creator-creation: Creator Creation Producer

1. **Message Channel**

```
to("jms:queue:notification.producer.creator.creation.queue")
```

Listing 2: Message Channel for Creator Creation

2. **Point-to-Point Channel:** Inherited and implemented by this type of queue.

6.4 event-market-event-creation: Event Creation Producer

1. **Message Channel**

```
to("jms:queue:notification.producer.event.creation.queue")
```

Listing 3: Message Channel for Event Creation

2. **Point-to-Point Channel:** Inherited and implemented by this type of queue.

6.5 event-market-notification-engine: Notification Engine Producer

1. **Message Channel**

```
from("jms:queue:notification.producer.user.creation.queue")
from("jms:queue:notification.producer.creator.creation.queue")
from("jms:queue:notification.producer.event.creation.queue")
from("jms:queue:notification.producer.ticketpayment.inprocess.queue")
from("jms:queue:notification.producer.ticketpayment.rejected.queue")
from("jms:queue:notification.producer.ticketpayment.approved.queue")
from("jms:queue:notification.producer.recommendations.queue")
```

Listing 4: Message Channels for Notification Engine

2. **Content-Based Router**

```
choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.user.creation.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.creator.creation.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.event.creation.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.ticketpayment.inprocess.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.ticketpayment.rejected.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.ticketpayment.approved.queue")

choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
    .otherwise()
        .to("jms:queue:notification.builder.recommendations.queue")
```

Listing 5: Content-Based Router for Notification Engine

3. **Point-to-Point Channel:** Inherited and implemented by this type of queue.

4. **Content Enricher**

```
if (user != null) {
    notificationJson.put("receivesEmails", user.getReceivesEmails());
    notificationJson.put("receivesPushs", user.getReceivesPushs());
} else if (creator != null) {
```

```
    notificationJson.put("receivesEmails", true);
    notificationJson.put("receivesPushs", true);
}
```

Listing 6: Content Enricher for Notification Engine

5. **Dead-Letter Channel**

```
choice()
    .when(header("validMessage").isEqualTo(false))
        .to("jms:queue:notification.deadletter.queue")
```

Listing 7: Dead-Letter Channel for Notification Engine

6.6 event-market-ticket-purchase: Ticket Purchase Producer

1. **Message Channel**

```
to("jms:queue:ticketvalidation.deadletter.queue")
to("jms:queue:ticketvalidation.financial.queue")
to("jms:queue:ticketvalidation.completion.queue")
to("jms:queue:notification.producer.ticketpayment.inprocess.queue")
to("jms:queue:recommender.builder.queue")
```

Listing 8: Message Channel for Ticket Purchase

2. **Content-Based Router**

```
.when(simple("${body[validPrice]} == false"))
    .to("jms:queue:ticketvalidation.deadletter.queue")
.otherwise()
    .multicast().stopOnException()
        .parallelProcessing()
        .to("direct:processTicket", "direct:inProcessNotification", "direct:
    inProcessRecommender")
    .end()
.end()
```

Listing 9: Content-Based Router for Ticket Purchase

3. **Invalid-Message Channel**

```
.choice()
    .when(simple("${body[validPrice]} == false"))
        .log("Sending to ticketvalidation.invalidmessage.queue")
        .to("jms:queue:ticketvalidation.invalidmessage.queue")
```

Listing 10: Invalid-Message Channel for Ticket Purchase

4. **Wire Tap**

```
.to("direct:processTicket", "direct:inProcessNotification", "direct:
    inProcessRecommender")
```

Listing 11: Wire Tap for Ticket Purchase

5. **Message Translator**

```
.marshal().json(JsonLibrary.Jackson)  // Convert Map to
```

Listing 12: Message Translator for Ticket Purchase

6. **Multicast**

```
.otherwise()
    .multicast().stopOnException()
        .parallelProcessing()
        .to("direct:processTicket", "direct:inProcessNotification", "direct:
    inProcessRecommender")
    .end()
.end()
```

Listing 13: Multicast for Ticket Purchase

6.7 event-market-financial-validation: Financial Validation Producer

1. **Message Channel**

```
from("jms:queue:ticketvalidation.financial.queue")
```

Listing 14: Message Channel for Financial Validation

2. **Content-Based Router**

```
choice()
    .when(simple("${body[financiallyValid]} == true"))
        .to("jms:queue:ticketvalidation.completion.queue")
    .otherwise()
        .to("jms:queue:notification.producer.ticketpayment.rejected.queue")
```

Listing 15: Content-Based Router for Financial Validation

6.8 event-market-ticket-validation: Ticket Validation Producer

1. **Message Channel**

```
from("jms:queue:ticketvalidation.completion.queue")
```

Listing 16: Message Channel for Ticket Validation

2. **Point-to-Point Channel:** Inherited and implemented by this type of queue.

3. **Content-Based Router**

```
choice()
    .when(simple("${body[message]} == 'ticket-payment-approved'"))
        .to("jms:queue:notification.producer.ticketpayment.approved.queue")
    .otherwise()
        .to("jms:queue:ticketvalidation.deadletter.queue")
```

Listing 17: Content-Based Router for Ticket Validation

4. **Message Translator**

```
.marshal().json(JsonLibrary.Jackson)
.marshal().json(JsonLibrary.Jackson)
```

Listing 18: Message Translator for Ticket Validation

5. **Normalizer**

```
.process(exchange -> {
    Object body = exchange.getIn().getBody();
    if (body instanceof String) {
        String jsonBody = (String) body;
        @SuppressWarnings("unchecked")
        Map<String, Object> messageData = objectMapper.readValue(jsonBody, Map.class);
        exchange.getIn().setBody(messageData);
    } else if (body instanceof Map) {
        // No conversion needed, body is already a Map
    } else {
        throw new IllegalArgumentException("Unsupported message format: " + body.
    getClass());
    }
})
```

Listing 19: Normalizer for Ticket Validation

6. **Dead-Letter Channel**

```
.choice()
    .when(simple("${body[message]} == 'ticket-payment-approved'"))
        .to("jms:queue:notification.producer.ticketpayment.approved.queue")
    .otherwise()
        .to("jms:queue:ticketvalidation.deadletter.queue")
```

Listing 20: Dead-Letter Channel for Ticket Validation

6.9 event-market-recommendation-engine: Recommendation Engine Producer

1. **Message Channel**

```
from("jms:queue:recommender.builder.queue")
```

Listing 21: Message Channel for Recommendation Engine

2. **Message Translator**

```
.unmarshal().json(JsonLibrary.Jackson, Map.class)  // Unmarshal
```

Listing 22: Message Translator for Recommendation Engine

3. **Content Enricher**

```
.process(new Processor() {
    @Override
    public void process(Exchange exchange) throws Exception {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

        try {
            @SuppressWarnings("unchecked")
            Map<String, Object> messageData = exchange.getIn().getBody(Map.class);
            int userId = (int) messageData.get("userId");

            // Fetch User from the database
            User user = em.find(User.class, userId);
            if (user == null) {
                throw new Exception("User not found for id: " + userId);
            }

            // Create the recommender model and set EntityManager
            NlpRecommenderModel model = new NlpRecommenderModel(user);
```

```
            model.setEntityManager(em);

            // Or use PopularityRecommenderModel if needed
            // PopularityRecommenderModel model = new PopularityRecommenderModel(user);
            // model.setEntityManager(em);

            EventRecommender eventRecommender = new EventRecommender(model);
            eventRecommender.requestRecommendation();
            List<Event> recommendations = eventRecommender.sendRecommendation();

            // Extract event IDs from the recommendations
            List<Integer> eventIds = recommendations.stream()
                .map(Event::getId)
                .collect(Collectors.toList());

            // Prepare recommendation message
            Map<String, Object> recommendationMessage = new HashMap<>();
            recommendationMessage.put("eventIds", eventIds);
            recommendationMessage.put("userId", userId);
            recommendationMessage.put("message", "event-recommendation");

            String jsonMessage = objectMapper.writeValueAsString(recommendationMessage)
    ;
            exchange.getIn().setBody(jsonMessage);

            em.getTransaction().commit();
        } catch (Exception e) {
            if (em.getTransaction().isActive()) {
                em.getTransaction().rollback();
            }
            throw e;
        } finally {
            em.close();
        }
    }
})
```

Listing 23: Content Enricher for Recommendation Engine

7 Design Patterns Code

7.1 Singleton Pattern

```
public class FinancialValidator {
    private static FinancialValidator instance;

    private FinancialValidator() {
        recentlyCheckedRequestIds = new ArrayList<>();
    }

    public static FinancialValidator getInstance() {
        if (instance == null) {
            instance = new FinancialValidator();
        }
        return instance;
    }
}
```

Listing 24: Singleton Pattern for Financial Validator

7.2 Iterator Pattern

```
public abstract class AbstractIterator<T> {
    public abstract T next();
    public abstract int getSteps();
    public abstract void setSteps(int step);
    public abstract boolean isDone();
    public abstract T current();
    public abstract T first();
}

public class TicketTypeIterator extends AbstractIterator<TicketType> {
    private Event event;
    private int currentIndex;
    private List<TicketType> ticketTypes;

    public TicketTypeIterator(Event event) {
        this.event = event;
        this.ticketTypes = event.getTicketTypes();
        this.currentIndex = 0;
    }

    @Override
    public TicketType next() {
        if (isDone()) {
            throw new IllegalStateException("No more ticket types");
        }
        TicketType ticketType = ticketTypes.get(currentIndex);
        currentIndex++;
        return ticketType;
    }

    // other overridden methods...
}
```

Listing 25: Iterator Pattern for Ticket Types

7.3 Template Method Pattern

```
public abstract class AbstractRecommenderModel {
    public abstract List<Event> recommend(int n);
    public abstract List<Event> recommend();
}

public class NlpRecommenderModel extends AbstractRecommenderModel {
    @Override
    public List<Event> recommend(int n) {
        // implementation...
    }

    @Override
    public List<Event> recommend() {
        // implementation...
    }
}

public class PopularityRecommenderModel extends AbstractRecommenderModel {
    @Override
    public List<Event> recommend(int n) {
        // implementation...
    }
```

```
23
24      @Override
25      public List<Event> recommend() {
26          // implementation...
27      }
28  }
```

Listing 26: Template Method for Recommender Models

7.4 Facade Pattern

```
1   public class EventRecommender {
2       private int id;
3       private User user;
4       private AbstractRecommenderModel model;
5       private int numberRecommendations;
6       private List<Event> eventRecommendations;
7
8       public EventRecommender(AbstractRecommenderModel model) {
9           this.user = model.getUser();
10          this.model = model;
11          this.numberRecommendations = 3;
12          this.eventRecommendations = new ArrayList<>();
13      }
14
15      public List<Event> sendRecommendation() {
16          return eventRecommendations;
17      }
18
19      public void setNumberRecommendations(int numberRecommendations) {
20          this.numberRecommendations = numberRecommendations;
21      }
22
23      // ...
24  }
```

Listing 27: Facade for Event Recommender

7.5 Strategy Pattern

```
1   public abstract class PaymentMethod {
2       @Id
3       @GeneratedValue(strategy = GenerationType.IDENTITY)
4       protected int id;
5
6       // common fields if any
7
8       // Getters and Setters
9       public int getId() {
10          return id;
11      }
12
13      public void setId(int id) {
14          this.id = id;
15      }
16  }
```

Listing 28: Strategy Pattern for Payment Method

```java
public class CardPayment extends PaymentMethod {
    @Column(nullable = false)
    private String cardNumber;

    @Column(nullable = false)
    private String cardHolder;

    @Temporal(TemporalType.DATE)
    @Column(nullable = false)
    private Date expirationDate;

    @Column(nullable = false)
    private String cvv;

    public CardPayment() {
    }

    public CardPayment(String cardNumber, String cardHolder, Date expirationDate, String cvv
    ) {
        this.cardNumber = cardNumber;
        this.cardHolder = cardHolder;
        this.expirationDate = expirationDate;
        this.cvv = cvv;
    }

    // ...
}
```

Listing 29: Card Payment Implementation

```java
public class CheckingAccountPayment extends PaymentMethod {
    @Column(nullable = false)
    private String accountNumber;

    @Column(nullable = false)
    private String routingNumber;

    public CheckingAccountPayment() {
    }

    public CheckingAccountPayment(String accountNumber, String routingNumber) {
        this.accountNumber = accountNumber;
        this.routingNumber = routingNumber;
    }

    // ...
}
```

Listing 30: Checking Account Payment Implementation