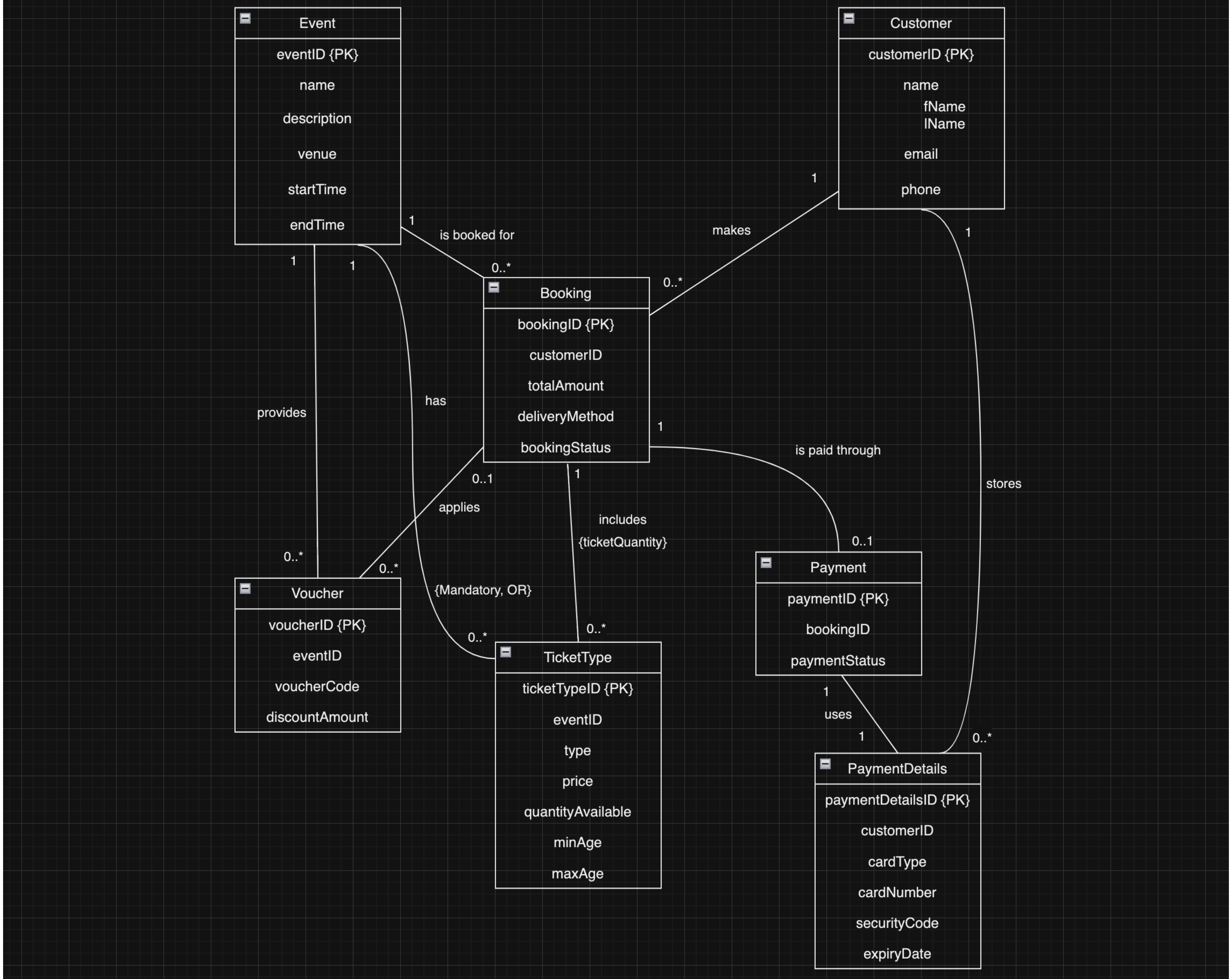
Event
eventID {PK}
name
description
venue
startTime
endTime
Customer
customerID {PK}
name
fName
lName
email
phone
Booking
bookingID {PK}
customerID
totalAmount
deliveryMethod
bookingStatus
Voucher
voucherID {PK}
eventID
voucherCode
discountAmount
TicketType
ticketTypeID {PK}
eventID
type
price
quantityAvailable
minAge
maxAge
Payment
paymentID {PK}
bookingID
paymentStatus
PaymentDetails
paymentDetailsID {PK}
customerID
cardType
cardNumber
securityCode
expiryDate
1
is booked for
0..*
1
makes
0..*
1
provides
0..*
1
has
{Mandatory, OR}
0..*
applies
0..1
0..*
1
includes
{ticketQuantity}
0..*
1
is paid through
0..1
1
uses
1
1
stores
0..*

Entities

Customer Entity: In the design of the Customer entity, it is assumed that every customer is uniquely identifiable by a customerID, which serves as the primary key. This ID facilitates unique identification within the system. Customers' personal details such as name, email, and phone number are also included, which are essential for communication and identification purposes. The assumption here is that each customer has a unique set of these details, particularly the email, which is crucial for booking confirmations and updates.

Event Entity: The Event entity is conceptualised with the understanding that each event within the system is distinct and identifiable by a unique eventID, acting as the primary key. The inclusion of attributes like name, description, venue, start time, and end time is to provide essential information about each event. It is assumed that each event’s name, description and venue are sufficient to distinguish one event from another within the system.

TicketType Entity: For the TicketType entity, the assumption is that ticket types (like adult, child, VIP) are specific to each event and are distinguishable by a ticketTypeID, which is used as the primary key. Attributes like type, price, quantity, minAge, and maxAge available are crucial for ticket sales and management. This entity’s design acknowledges the variability in ticket pricing and availability, which is a common scenario in event management.

Booking Entity: In designing the Booking entity, it is assumed that every booking is unique and can be identified using a bookingID, which serves as its primary key. The total amount, delivery method, and booking status (active, cancelled) are crucial for tracking the booking lifecycle. The inclusion of the delivery method caters to the specification's need for different ticket delivery options (email or physical pickup). The customerID as a foreign key establishes a relationship with the Customer entity, aligning each booking with a specific customer.

Payment and PaymentDetails Entities: The Payment entity is uniquely identified by a paymentID. It is associated with a specific booking and its status (paid, pending) is critical for transaction management. The separation of PaymentDetails, identified by paymentDetailsID, from Payment is based on the need to securely store sensitive payment information like card details, separate from transactional data, adhering to data protection best practices, and also in order to access the payment details in the future.

Voucher Entity: For the Voucher entity, the unique identifier is voucherID. This entity captures discount vouchers, and I assumed their applicability to various events and the possibility of multiple vouchers per event. The addition of a voucherCode attribute allows for the unique identification of each voucher. The discount amount attribute is included to quantify the value of each voucher. Since multiple vouchers can be associated with one booking in my implementation, the voucherID is not included as a foreign key in Booking, but instead as an associative entity BookingVoucher in the relational model.

Relationships and Their Cardinalities

Customer-Booking (Makes) Relationship: The relationship between the Customer and Booking entities is designed as a one-to-many (1 to 0..*) relationship. This design decision is based on the assumption that a single customer can make multiple bookings, but each booking is uniquely associated with one customer. This relationship reflects the typical user behaviour in ticket booking systems, where customers often engage in multiple transactions over time.

Event-Booking (Is Booked For) Relationship: The relationship between Event and Booking is also modelled as a one-to-many (1 to 0..*) relationship. This is based on the understanding that each booking is specific to a single event, but an event can have multiple bookings associated with it. This structure caters to scenarios where popular events may have numerous bookings from different customers.

Event-TicketType (Has) Relationship with {Mandatory, OR}: The Event-TicketType relationship is represented as a one-to-many (1 to 0..*) relationship, along with a {Mandatory, OR} constraint. This implies that each event must offer at least one type of ticket (mandatory), but it can offer multiple types (OR). This design decision reflects the diversity in ticketing options for events, such as adult, child, or VIP tickets, ensuring that every event has at least one ticket type available for sale.

Event-Voucher (Provides) Relationship: Similar to the previous relationships, the Event-Voucher relationship is a one-to-many (1 to 0..*) relationship. This structure is based on the premise that an event can have multiple vouchers applicable to it, but each voucher is specifically tied to a particular event. This relationship captures the promotional strategies of events where different vouchers may be offered to attract diverse customer segments.

Booking-TicketType (Includes) Relationship with {ticketQuantity}: The relationship between Booking and TicketType is modelled with a many-to-many (1 to 0..*) cardinality, including an attribute {ticketQuantity} on the relationship. This is an essential design choice, as it captures the quantity of each type of ticket included in a booking. It addresses the scenario where a single booking can contain various ticket types in different quantities, reflecting the customer's purchase choices.

Booking-Payment (Is Paid Through) Relationship: The Booking-Payment relationship is defined as a one-to-one (1 to 0..1) relationship. This indicates that each booking leads to a single payment transaction. However, there is a possibility of a booking not having a completed payment (hence 0..1), which could occur in cases of booking cancellation or payment failure.

Payment-PaymentDetails (Uses) Relationship: The relationship between Payment and PaymentDetails is a one-to-one (1 to 1) relationship.

This decision comes from the assumption that each payment transaction uses a unique set of payment details. This relationship is critical for linking transactional data to financial information securely and storing it for future purposes.

Booking-Voucher (Applies) Relationship: The Booking-Voucher relationship is characterised as a many-to-many (0..1 to 0..*) relationship. This configuration allows for the application of none, one, or multiple vouchers to a booking, and conversely, a voucher can be applied to multiple bookings. This is to accommodate various discounting strategies employed by events.

Customer-PaymentDetails (Stores) Relationship: The relationship between the Customer and PaymentDetails entities is modelled as a one-to-many (1 to 0..*) relationship. This design choice is based on the understanding that a single customer may choose to store multiple sets of payment details in the system. The cardinality of this relationship reflects real-world usage where customers often have multiple payment methods.

Relational model

Customer(<u>customerID</u>, fName, lName, email, phone)
Primary key: <u>customerID</u>

Event(<u>eventID</u>, name, description, venue, startTime, endTime)
Primary key: <u>eventID</u>

TicketType(<u>ticketTypeID</u>, eventID, type, price, quantityAvailable, minAge, maxAge)
Primary key: <u>ticketTypeID</u>
Foreign key: eventID references Event(eventID)

Booking(<u>bookingID</u>, customerID, totalAmount, deliveryMethod, bookingStatus)
Primary key: <u>bookingID</u>
Foreign key: customerID references Customer(customerID)

Payment(<u>paymentID</u>, bookingID, paymentStatus)
Primary key: <u>paymentID</u>
Foreign key: bookingID references Booking(bookingID)

PaymentDetails(paymentDetailsID, customerID, cardType, cardNumber, securityID, expiryDate)
Primary key: paymentDetailsID
Foreign key: customerID references Customer(customerID)

Voucher(voucherID, eventID, voucherCode, discountAmount)
Primary key: voucherID
Foreign key: eventID references Event(eventID)

BookingTicket(bookingID, ticketTypeID, ticketQuantity)
Primary keys: bookingID, ticketTypeID
Foreign keys:

- bookingID references Booking(bookingID)
- ticketTypeID references TicketType(ticketTypeID)

BookingVoucher(bookingID, voucherID)
Primary keys: bookingID, voucherID
Foreign keys:

- bookingID references Booking(bookingID)
- voucherID references Voucher(voucherID)