Perfect—let’s design the **database schema** for the Payment Gateway module. This schema supports merchant registration, payment lifecycle management, refunds, and webhook simulation. I’ll lay it out with tables, relationships, and notes on constraints.

🗄️ Payment Gateway Database Schema

1. **Merchant Table**

Column	Type	Constraints	Notes
`id`	BIGINT (PK)	Auto-increment	Unique merchant identifier
`name`	VARCHAR(255)	NOT NULL	Merchant’s display name
`api_key`	VARCHAR(255)	NOT NULL, UNIQUE	Secure API key for authentication
`created_at`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Registration time

---

2. **Payment Table**

Column	Type	Constraints	Notes
`id`	BIGINT (PK)	Auto-increment	Unique payment identifier
`merchant_id`	BIGINT (FK)	REFERENCES Merchant(id)	Links payment to merchant
`amount`	DECIMAL(12,2)	NOT NULL	Payment amount
`currency`	VARCHAR(10)	NOT NULL	ISO currency code (e.g., USD, CAD)
`description`	VARCHAR(255)		Optional order description
`status`	ENUM	Values: `PENDING` , `SUCCESS` , `FAILED` , `REFUNDED`	Lifecycle state

Column	Type	Constraints	Notes
`created_at`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Payment creation time
`updated_at`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE	Last status update

---

3. **Transaction Table** (optional, for audit trail)

Column	Type	Constraints	Notes
`id`	BIGINT (PK)	Auto-increment	Transaction record
`payment_id`	BIGINT (FK)	REFERENCES Payment(id)	Associated payment
`action`	ENUM	Values: `CREATE`, `COMPLETE`, `REFUND`	What happened
`status`	ENUM	Values: `SUCCESS`, `FAILED`	Outcome of action
`timestamp`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	When action occurred
`reason`	VARCHAR(255)		Optional (e.g., refund reason)

---

4. **WebhookLog Table** (optional, for simulation)

Column	Type	Constraints	Notes
`id`	BIGINT (PK)	Auto-increment	Log entry
`payment_id`	BIGINT (FK)	REFERENCES Payment(id)	Associated payment
`status`	ENUM	Values: `SUCCESS`, `FAILED`, `REFUNDED`	Status sent
`delivered_at`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	When webhook was delivered
`endpoint`	VARCHAR(255)		Merchant endpoint (simulated)

🔗 Relationships

- **Merchant → Payment:** One-to-many (a merchant can create many payments).
- **Payment → Transaction:** One-to-many (each payment can have multiple lifecycle events).
- **Payment → WebhookLog:** One-to-many (each payment can trigger multiple webhook notifications).

⚖️ Constraints & Notes

- **API Key Security:** Store API keys hashed if you want production realism (e.g., SHA-256 + salt).
- **Currency Validation:** Enforce ISO codes via check constraint or validation logic.
- **Status Transitions:** Business logic ensures valid transitions (e.g., `PENDING → SUCCESS/FAILED → REFUNDED`).
- **Auditability:** Transaction table ensures you can reconstruct payment history.

✅ This schema aligns perfectly with the API contract you already have. It gives you persistence for merchants, payments, lifecycle events, and webhook logs.

Would you like me to also sketch out the **JPA entity classes** (Java code skeletons) that map directly to these tables? That way you'll have a ready-to-code blueprint for your Spring Boot implementation.