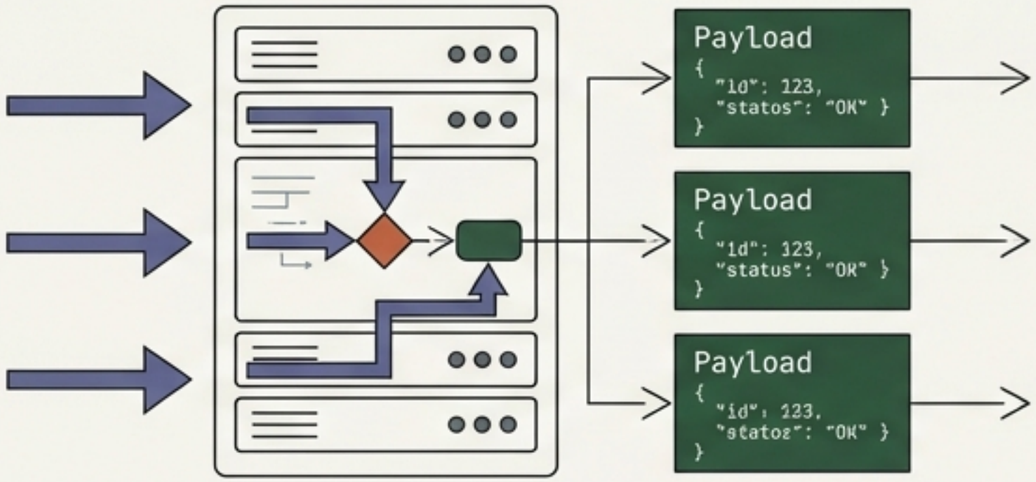
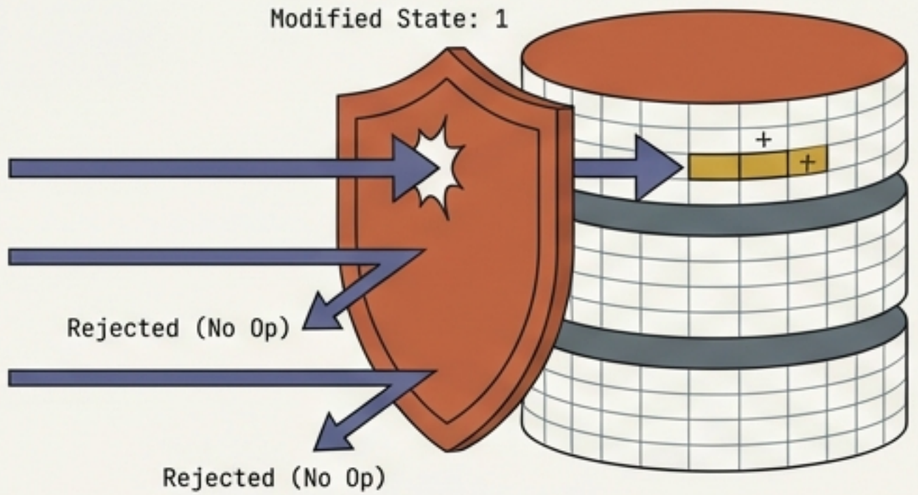
The Misconception



Common Myth: Idempotency means the API server creates and returns the exact same JSON response object every time, regardless of internal state.

The Reality (Definition)

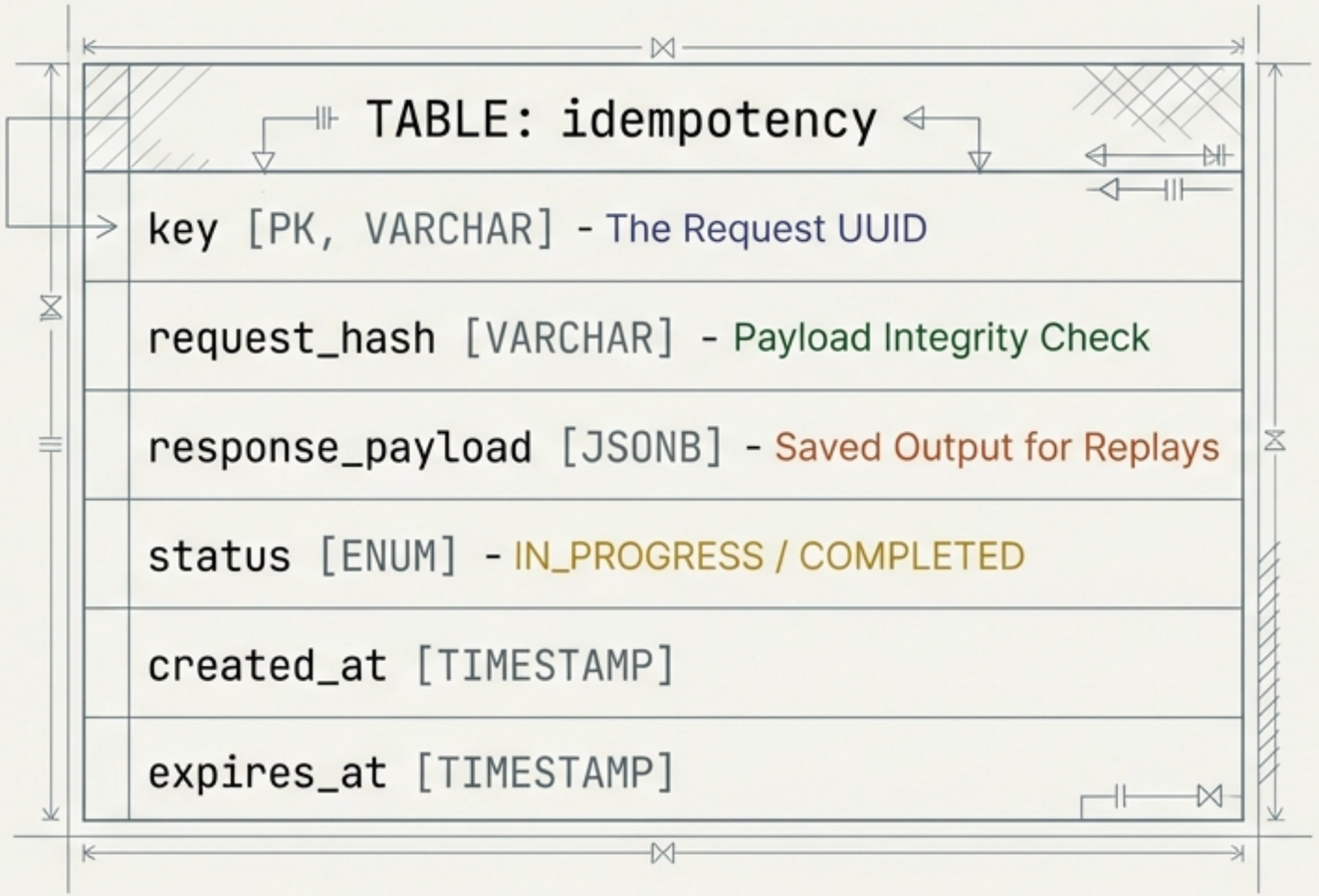


An idempotent API guarantees that multiple identical requests result in **one logical operation and one final state**, without creating duplicates.

The Rule: We care about the state of the database, not just the JSON returned to the client.

Phase 1: The Idempotency Table Anatomy

The System's Memory Log



Purpose & Utility

This table sits *in front* of the reservation logic. It serves a dual purpose:

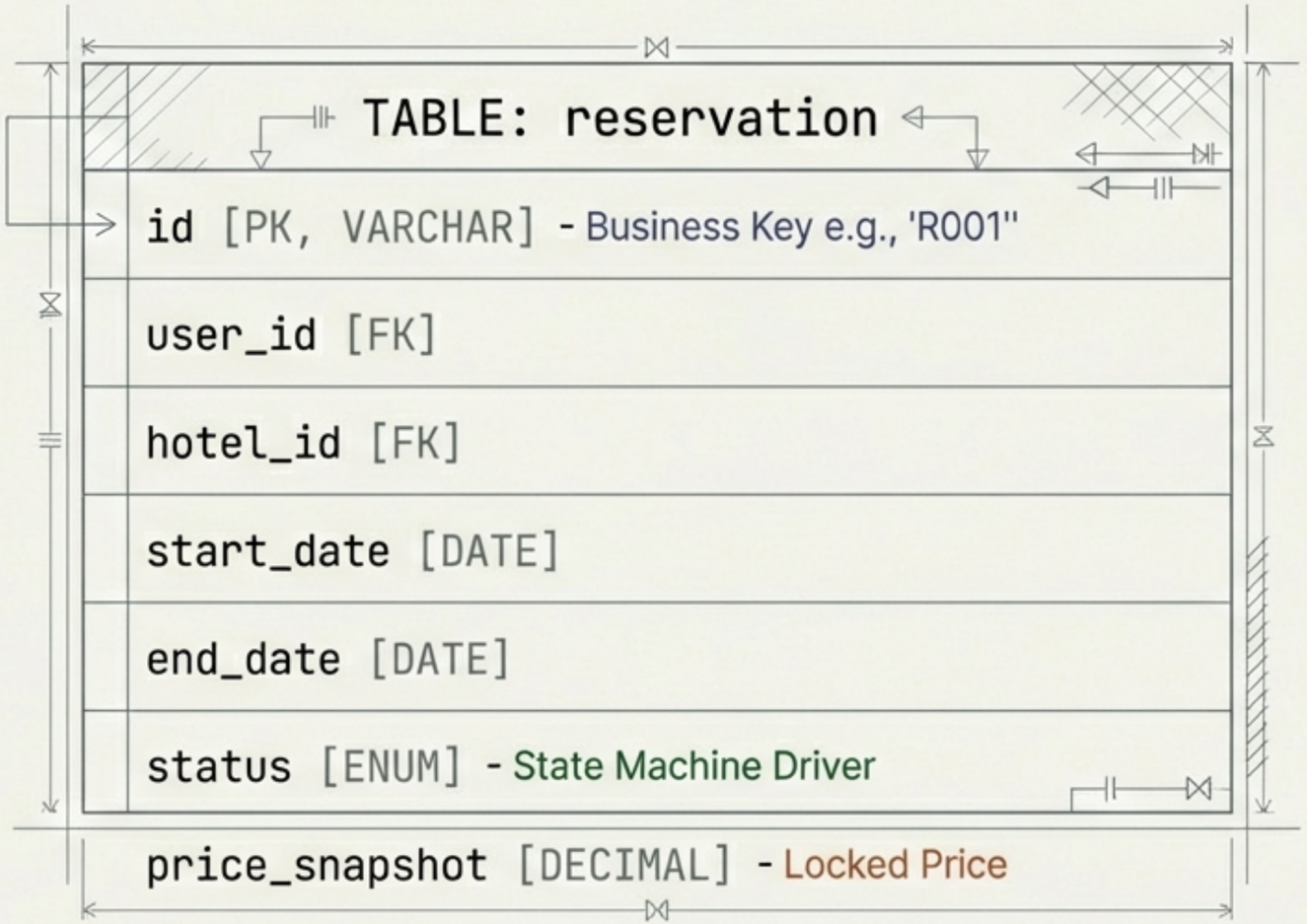
1. **Deduplicating Reserve API calls:** It acts as a lock against double-clicks.
2. **Handling Retries:** It stores the result of the first success to replay it later.

The Expiry Strategy

The `expires_at` column allows the system to 'forget' old keys after 24 hours, preventing table bloat while covering the maximum reasonable retry window.

Phase 1: The Reservation Table Anatomy

The Business Logic Driver



Separation of Concerns

While the Idempotency table protects the API infrastructure, this table protects the inventory logic. It tracks the lifecycle of the booking from draft to confirmation.

The State Machine

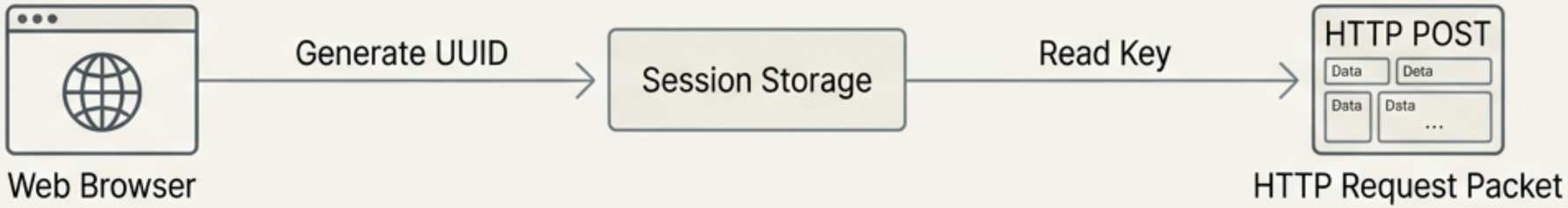
The `status` column is the most critical field. It enforces strict transitions (e.g., STARTED → CONFIRMED) and prevents illegal operations, such as paying for a reservation that has already timed out.

Tutor Tip

Note: We record the `price_snapshot` here. If the hotel changes rates 1 minute later, this user's draft price must be honored.

Step 1: The Client-Side Genesis

Persistence begins before the network call.



Left Panel (Code Block - JetBrains Mono)

```
// Inside the Browser
const bookingKey = generateUUID();

// PERSIST IMMEDIATELY
sessionStorage.setItem("booking_key", bookingKey);

// Send Key in Header
fetch("/reservations", {
  method: "POST",
  headers: {
    "Idempotency-Key": bookingKey
  },
  body: JSON.stringify(bookingDetails)
});
```

Right Panel (Explanation - Inter)

Why Client-Side Generation?

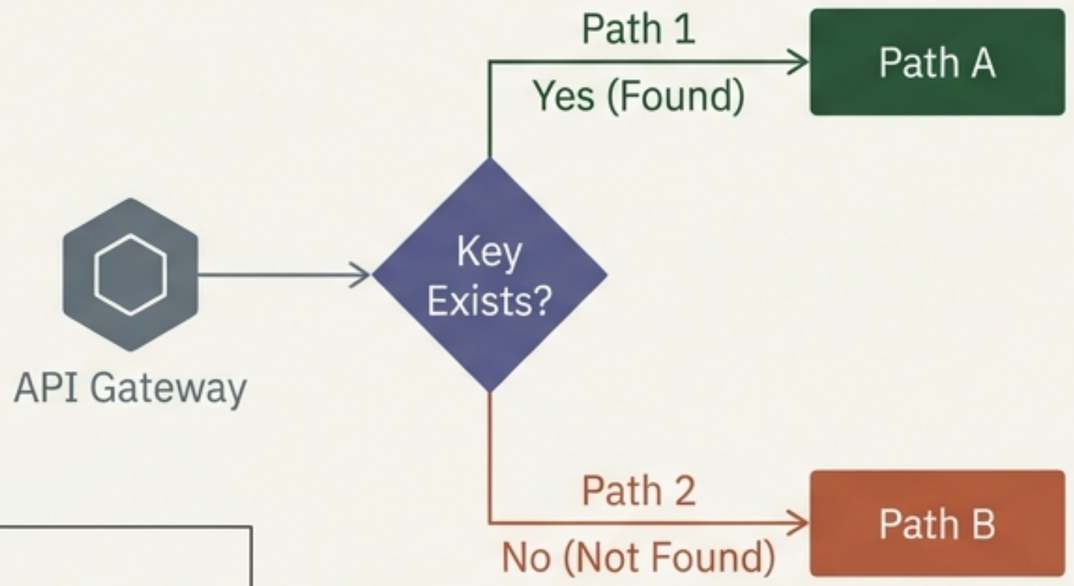
If the network fails *before* the server responds, the client must be able to **retry**. If the server generated the key, a network failure would lose that key forever. By generating it in the browser, the client owns the identity of the request.

The SessionStorage Safety Net

We save the key to `sessionStorage` immediately. If the user accidentally refreshes the page, the JavaScript reloads, finds the existing key, and sends it again. This is crucial for the "Refresh Scenario".

Step 2: The Gatekeeper

The ‘Check-Before-Act’ Protocol




```
-- The very first query executed
SELECT response_payload, status
FROM idempotency
WHERE key = :bookingKey;
```

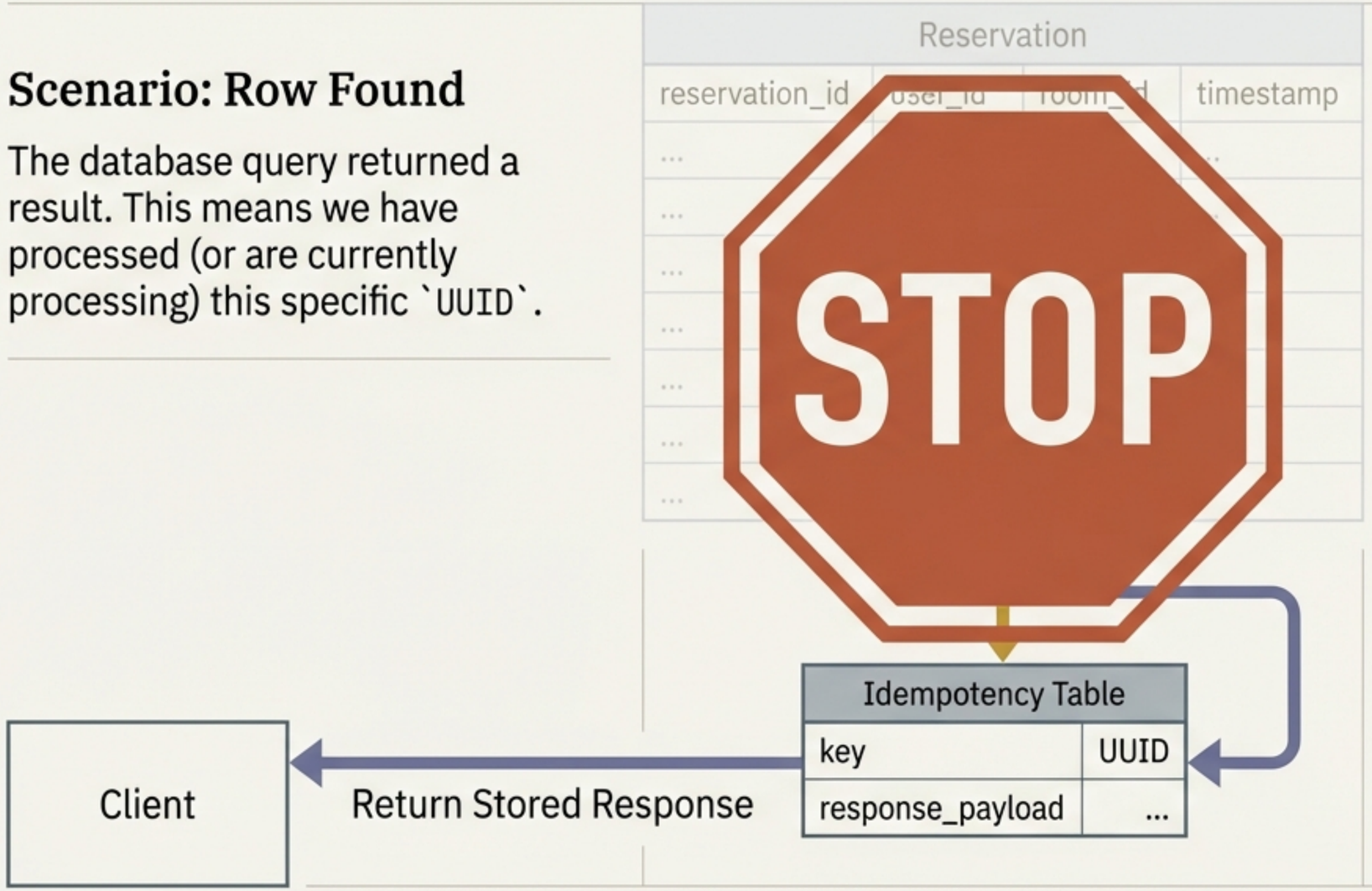
The Entry Logic
Before executing any expensive business logic or booking a room, the system queries the history. This is a low-cost `SELECT` operation indexed by the Primary Key.
The Decision Matrix
The result of this query dictates the entire execution flow. We are determining if this is a **Retry** (a returning signal) or a **New Request** (a fresh signal).

Path A: The Safety Net (Retry Logic)

Handling Duplicates and Refreshes

Scenario: Row Found

The database query returned a result. This means we have processed (or are currently processing) this specific `UUID`.



The Action: Return & Stop

1. Return the `response_payload` stored in the idempotency row.
2. Terminate the thread.

The Consequence: No Mutation

This is the core value proposition. Even if the user clicks 'Reserve' 50 times in one second, or refreshes the page while the network is lagging, the `INSERT INTO reservation` command is never reached. The database state remains pristine.

Path B: The Placeholder (New Request)

Locking the Intention

```
INSERT INTO idempotency
  (key, status, created_at, expires_at)
VALUES
  (:bookingKey, 'IN_PROGRESS', now(),
    now() + interval '24 hours');
```

Claiming the Key

We insert the row immediately with a status of `IN_PROGRESS`. This acts as a semaphore. If a parallel request with the same key arrives 1 millisecond later, it will violate the Primary Key constraint and fail/wait.

Why 24 Hours?

The expiry is set to 24 hours to cover extended network outages. If a user loses signal and retries the next day, we still want to recognize the key, though likely we would prompt them to start over if the session is stale.

🗣️ **Tutor Tip:** Crucial Sequencing: We insert into the idempotency table *before* touching the reservation table. This marks the transaction as “active” and traceable.

Step 3: The Draft Creation

Instantiating the Business Object

```
INSERT INTO reservation
  (id, user_id, hotel_id, start_date, end_date,
  status, created_at, expires_at)
VALUES
  ('R001', :userId, :hotelId, :startDate, :endDate,
  'STARTED', now(), now() + interval '15 minutes');
```

Status: STARTED

We generate a Business ID (`R001`). The status is explicitly set to `STARTED`, not `CONFIRMED`.

The 15-Minute Timer

This expiry is different from the idempotency expiry. This limits how long a user can hold a “draft” before they must pay. It’s a cleanup mechanism for abandoned carts.

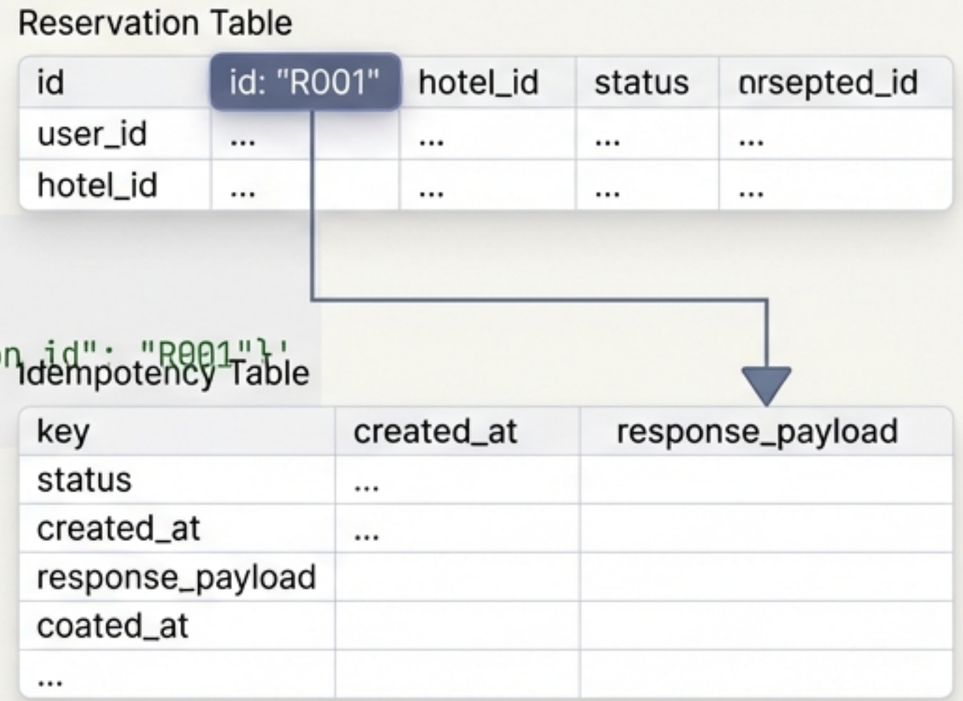
CRITICAL CONSTRAINTS

1. This is NOT a confirmed booking.
2. Inventory is NOT yet locked.
3. Other users can still book this room.

Step 4: Closing the Idempotency Loop

Linking Request to Result

```
UPDATE idempotency
SET status = 'COMPLETED',
    response_payload = '{"reservation_id": "R001"}'
WHERE key = :bookingKey;
```



The Commit Point

We update the idempotency row from `IN_PROGRESS` to `COMPLETED`. We embed the newly created Reservation ID (`R001`) into the payload.

Future-Proofing

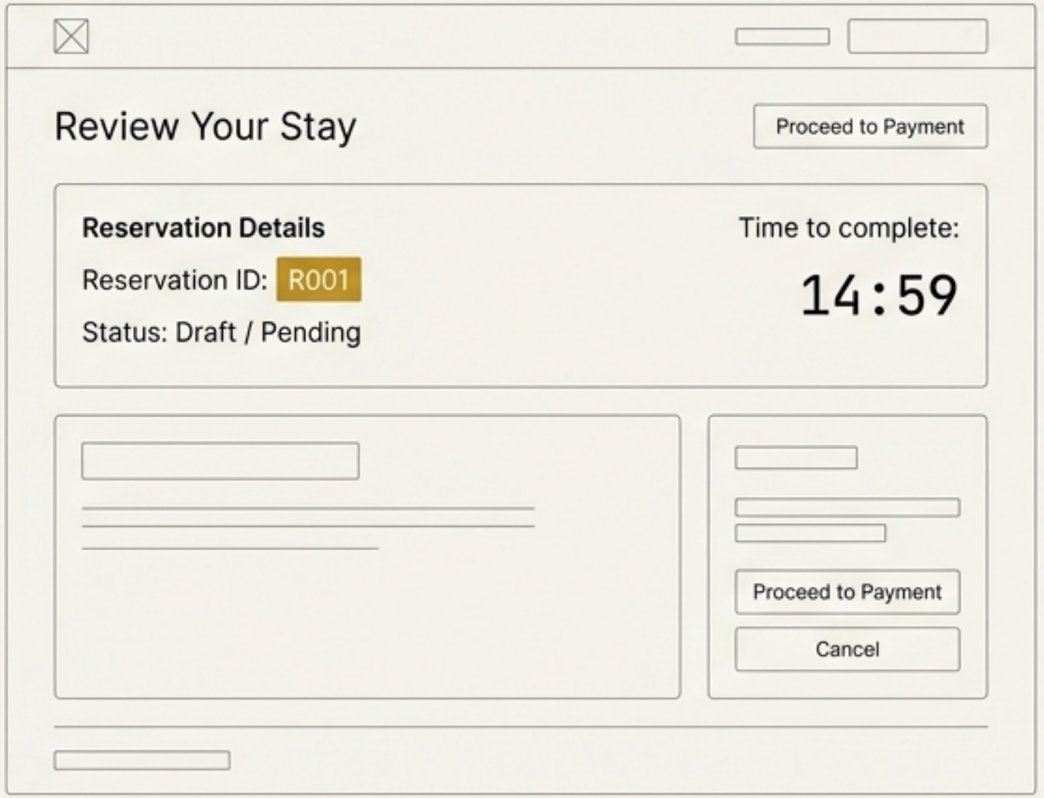
If the user retries this request later, the 'Safety Net' (Path A) will read this exact JSON blob and return `{"reservation_id": "R001" }`, making it look like a fresh success.

Step 5: The User Feedback Loop

Rendering the Draft

Return Payload

The API responds with HTTP 200 OK and the JSON body containing the ID `R001`.



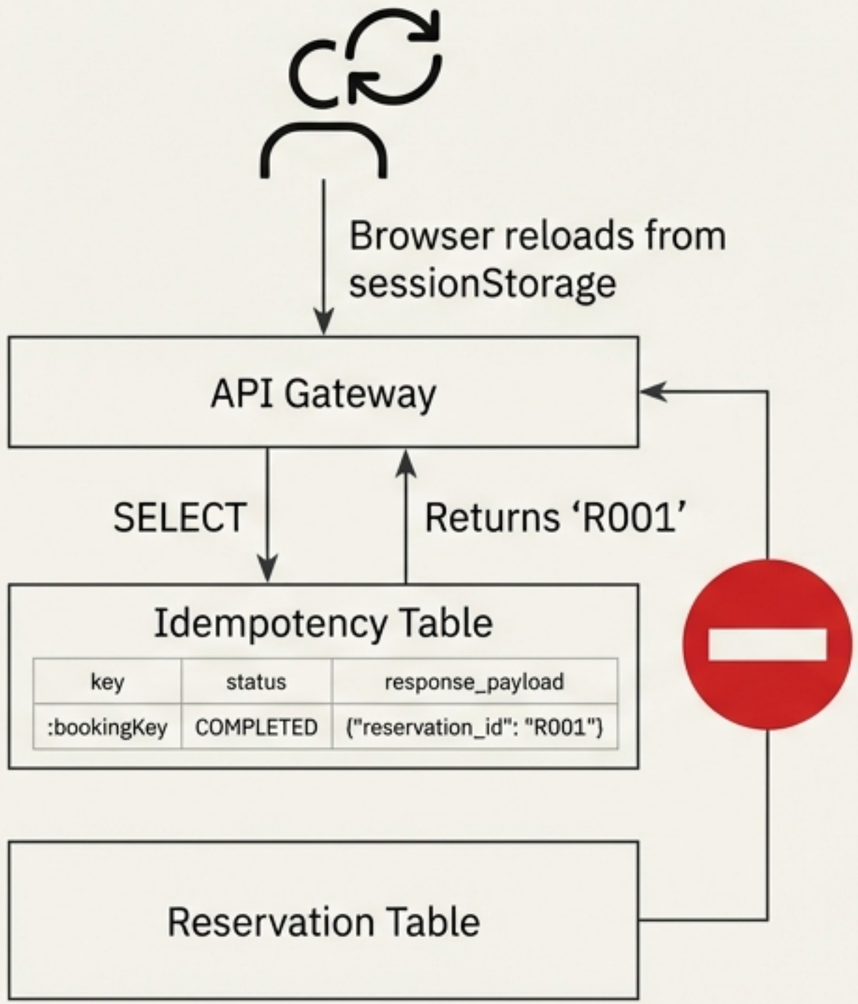
The UX Context

At this stage, the user believes they have "reserved" the room, but technically the hotel room is still free for others to take. This is a soft reservation.

The user must now proceed to payment to finalize the lock.

Step 6: The Refresh Scenario

Testing the Resilience



No New Reservation Created.

Because the browser persisted the key in `sessionStorage` (Slide 4), the refresh sends the the *_exact same_* UUID.

The backend logic hits ‘Path A’ (Slide 6), finds the `COMPLETED’ row, and simply returns `R001’ again. The database does not insert a new row `R002’.

Step 7: Intermediate Transitions

Advancing the State Machine

Backend Logic

- **Action**: User clicks “Next / Confirm Details”.
- **Endpoint:** POST /reservations/R001/confirm-details

```
UPDATE reservation
SET status = 'DETAILS_CONFIRMED'
WHERE id = 'R001' AND status = 'STARTED';
```

Explanation

The Checkpoint
We use an `UPDATE` statement that includes the *current state* in the `WHERE` clause. This ensures we only move forward from a valid state.

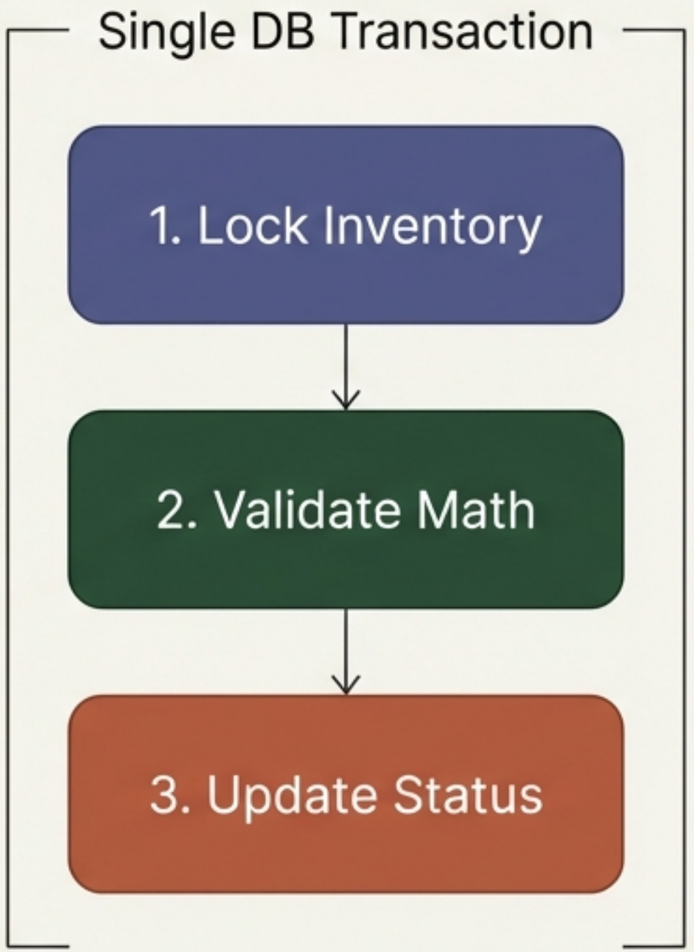
Idempotency by Logic
If the user clicks this button twice, the second request will try to update a row where status is *already* DETAILS_CONFIRMED. The `WHERE status = 'STARTED'` clause will fail to match, resulting in 0 rows updated. The system detects this and returns the current state safely.

The Critical Path: The Final Transaction

Steps 8 & 9 - Atomic Commitment

User Action: Click "Complete Booking & Pay".

Header: `Payment-Idempotency-Key: pay-uuid`



A New Idempotency Context

Notice we use a *new* key (`pay-uuid`). The draft creation and the payment confirmation are two distinct idempotent operations. We cannot reuse the draft key here.

Atomicity

From this point on, every SQL command happens inside a single transaction wrapper. Either **all** of them succeed, or **none** of them happen. This is the only way to prevent data drift.

Step 9.1: Pessimistic Locking

Preventing Race Conditions

```
SELECT booked_rooms, total_rooms
FROM room_type_inventory
WHERE hotel_id = :hotelId
  AND date BETWEEN :start AND :end
FOR UPDATE;
```

The “FOR UPDATE” Clause

This is the most technically critical line in the system. It tells the database: “I am reading these rows because **I intend to write** to them. **Block anyone else from reading them** until I am done.”

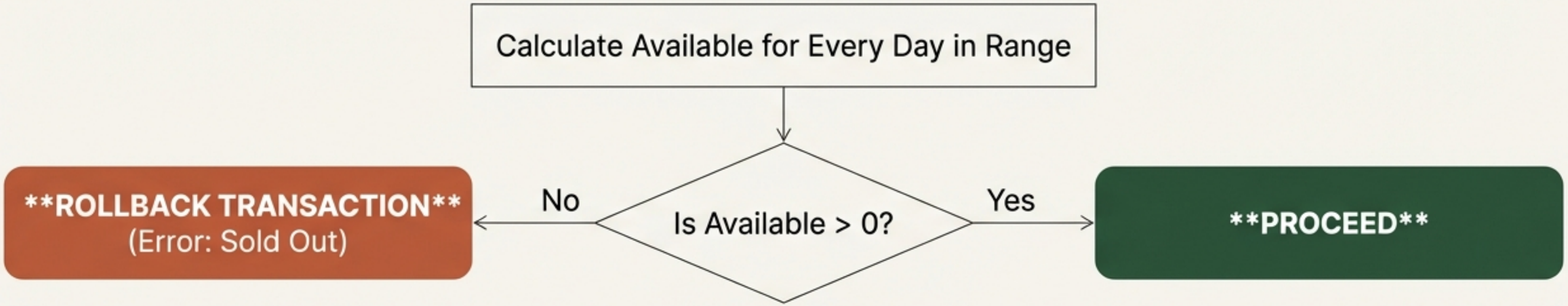
Stopping the Race

If User A and User B both try to book the last room at the exact same millisecond, the room at the exact same millisecond, the database will force one to wait until the other finishes. This serialized access prevents overselling.

Step 9.2: Application Level Validation

The Mathematics of Availability

```
Available = Total + OverbookingLimit - Booked
```



While holding the lock (from the previous slide), we perform the math. We check every single date in the booking range. If even one night is sold out, we must abort the entire transaction immediately to release the locks for other waiting users.

Step 9.3 & 9.4: The Permanent Mutation

The Point of No Return

```sql
-- 1. Consume Inventory
UPDATE room_type_inventory
SET booked_rooms = booked_rooms + 1
WHERE ...;

-- 2. Finalize Reservation
UPDATE reservation
SET status = 'CONFIRMED'
WHERE id = 'R001';

-- 3. Release Locks
COMMIT;
```

Inventory Consumption

We permanently increment the `booked_rooms` counter. Because of the lock, this operation is safe.

Status Confirmation

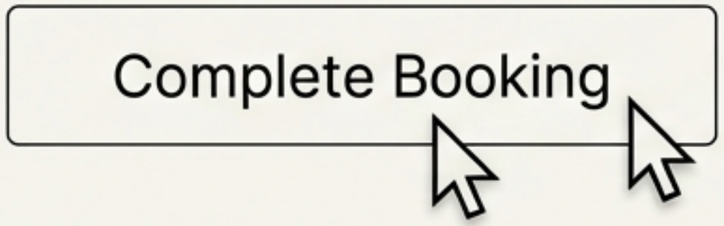
We flip the reservation status to `CONFIRMED`. The user now owns the room.

`COMMIT`

The transaction ends. The database saves changes to disk and releases the `FOR UPDATE` locks, allowing other users to proceed.

Step 10: Handling the Double-Click

Idempotency at the Finish Line



Scenario: Duplicate Payment Click
The user is impatient and clicks “Pay” twice. The second request arrives while the first is just finishing.

1. Check: `SELECT status FROM reservation WHERE id = 'R001'`
2. Result: `CONFIRMED`
3. Action: **Short-Circuit Success**

Checklist of Avoided Disasters (Green Checks)
[✅] No Inventory Update (Prevented -1 room)
[✅] No Payment Retry (Prevented double charge)
[✅] No Duplicate Row (Prevented data corruption)

Synthesis: The Guarantee Matrix

Mapping Problems to Architectural Solutions

The Problem	The Architectural Solution
Duplicate ‘Reserve’ Click	**Idempotency Table** (Check-before-Act / Primary Key Constraint)
Browser Page Refresh	**SessionStorage** (Client-side UUID persistence)
Double ‘Pay’ Click	**State Machine** (Status Check) + **Payment Idempotency Key**
Overselling Inventory	**Pessimistic Locking** (`SELECT ... FOR UPDATE`)