🧾 Detailed Lecture Notes: Idempotency in Booking Systems

📘 Overview

This lecture explores the problem of **duplicate bookings in transactional systems** (e.g., hotel/flight reservations), and introduces **idempotency** as a reliable backend pattern to avoid inconsistencies. We assume no prior experience with the topic and explain from first principles.

✅ Problem Statement: Double Bookings

🎯 What is Double Booking?

When a user accidentally or intentionally sends the same booking request multiple times, the backend might create multiple bookings and charge the user multiple times.

🛠️ Example:

User clicks a **“Book Now”** button:

- First click → Booking 1 created ✅
- Second click (by mistake or due to lag) → Booking 2 created ❌

This can cause:

- Revenue loss due to refunds
- Frustration due to duplicate charges
- Poor user experience

❌ Naive Solution: Disable Button on Frontend

💡 Idea:

Use HTML or JavaScript to disable the button after the first click:

```
<button disabled>Book Now</button>
```

Or using JavaScript:

```
button.addEventListener('click', () => {
  button.disabled = true;
  makeBooking();
});
```

🚨 Why This is Not Enough

1. **DevTools Bypass**: User can re-enable the button from browser developer tools.
2. **JavaScript Disabled**: Browser might have JS turned off.
3. **Third-Party Clients**: Someone might hit your API using Postman, curl, etc.

Conclusion: Frontend controls can be bypassed. **Always validate on the server side.**

✅ Real Solution: Backend-Driven Idempotency

📖 Definition of Idempotency

> An operation is **idempotent** if it can be applied multiple times without changing the result beyond the first time.

🔁 Examples in HTTP:

- **GET /users/123**: Always returns the same user → Idempotent
- **DELETE /users/123**: Deletes the user once → Further calls have no effect → Idempotent
- **POST /bookings**: Typically creates new resources → Not idempotent by default ❌

🎯 Our Goal

Make **POST /bookings** idempotent to prevent multiple charges and bookings for the same user action.

🧠 Implementation Strategy

1. Generate an Idempotency Key (client side)

- Unique identifier for the request
- Example: `UUIDv4` (universally unique identifier)
- Can be generated using libraries:

```
// JavaScript (frontend)
import { v4 as uuidv4 } from 'uuid';
const idempotencyKey = uuidv4();
```

2. Send It in the API Request

```
POST /bookings HTTP/1.1
Idempotency-Key: 123e4567-e89b-12d3-a456-426614174000
Content-Type: application/json

{
  "userId": 101,
  "roomId": 201,
  "paymentDetails": {...}
}
```

3. Handle It on the Server

- Save the key along with the response when processing a request for the first time.
- If the same key is received again, **return the stored response instead of processing again**.

🌐 Pseudo Code (Node.js + Express):

```
const cache = {}; // Can be Redis, DB table, etc.

app.post('/bookings', async (req, res) => {
  const key = req.headers['idempotency-key'];
  if (!key) return res.status(400).send({ error: 'Missing Idempotency Key'

  if (cache[key]) {
    return res.status(200).send(cache[key]); // Return cached response
  }

  const result = await createBooking(req.body); // Booking logic
  cache[key] = result; // Save response

  res.status(200).send(result);
});
```

> 🔁 Replace `cache` with Redis or a DB table in production for persistence and scalability.

🏗️ Real-Life Example: Flipkart Flights via Cleartrip

🎓 Scenario:

- Flipkart allows flight bookings.

- Internally uses Cleartrip’s API (their subsidiary).
- When a user clicks “Book”, they are redirected to a URL with a unique itinerary ID.

```
https://www.cleartrip.com/flights/itinerary/abc123xyz
```

🔍 What Happened Behind the Scenes?

- A **temporary booking** (draft) was created.
- You’re now on a screen to fill traveler and payment info.
- If you refresh or revisit → still same booking session (idempotent behavior).

📐 Design Considerations

🧱 DB Table for Idempotency (SQL Schema Example)

```
CREATE TABLE idempotency_keys (
  id VARCHAR(255) PRIMARY KEY,
  user_id INT,
  request_hash TEXT,
  response_body TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);
```

Use `request_hash` (optional) to ensure content hasn’t changed maliciously.

🔄 TTL Cleanup

- Store `idempotency_keys` with TTL (time-to-live) to prevent DB bloat.
- Use Redis or a cron job to clean up old entries.

🧪 Student Exercise

🎯 Task:

Build a **hotel booking API** that handles:

1. Bookings with room availability.
2. Prevents duplicate bookings for same idempotency key.
3. Uses in-memory store (or Redis) for key tracking.

✨ Optional Add-ons:

- Add retry logic from frontend.
- Show toast/snackbar if duplicate request detected.
- Save metadata like `createdAt`, `status`, etc.

📌 Recap & Takeaways

Concept

Summary

Problem

Users accidentally make double bookings

Naive Fix

Disable button (insecure, bypassable)

Ideal Fix

Backend-enforced idempotency

Key Mechanism

Use Idempotency-Key header to deduplicate requests

Real-World Example

Flipkart → Cleartrip URL redirection with draft booking

> Idempotency isn't just good practice. It's **essential** for any app that deals with **money, inventory, or critical resources**.

❓ FAQs

Q1: Can I use Idempotency for `GET` APIs?

Yes, but it's redundant. `GET` is already idempotent by design.

Q2: Should every `POST` API be idempotent?

Not always. Only those dealing with **critical resource creation**, **money**, or **booking** should.

Q3: What if I get a different payload with the same key?

Reject the request. Or store a hash of the original request to compare.