High Level Details (HLD)

Context

A system to handle movie seat reservations during high-traffic windows. The primary challenge is preventing race conditions (double-booking) while managing temporary seat holds. Can handle 1000+ concurrent users at the same time for the app because of the low memory footprint of Go.

Proposed Approach

1. Use `FOR UPDATE` locking to lock specific seat rows during the reservation transaction.
2. We assign a locked_at timestamp to seats.
3. We then manage the 10 minute seat holds using **Lazy Lock Expiration**. This avoids the overhead of background cleanup processes by validating and resetting expired timestamps at the time of active read/write operations instead of running a cron job.

Architecture

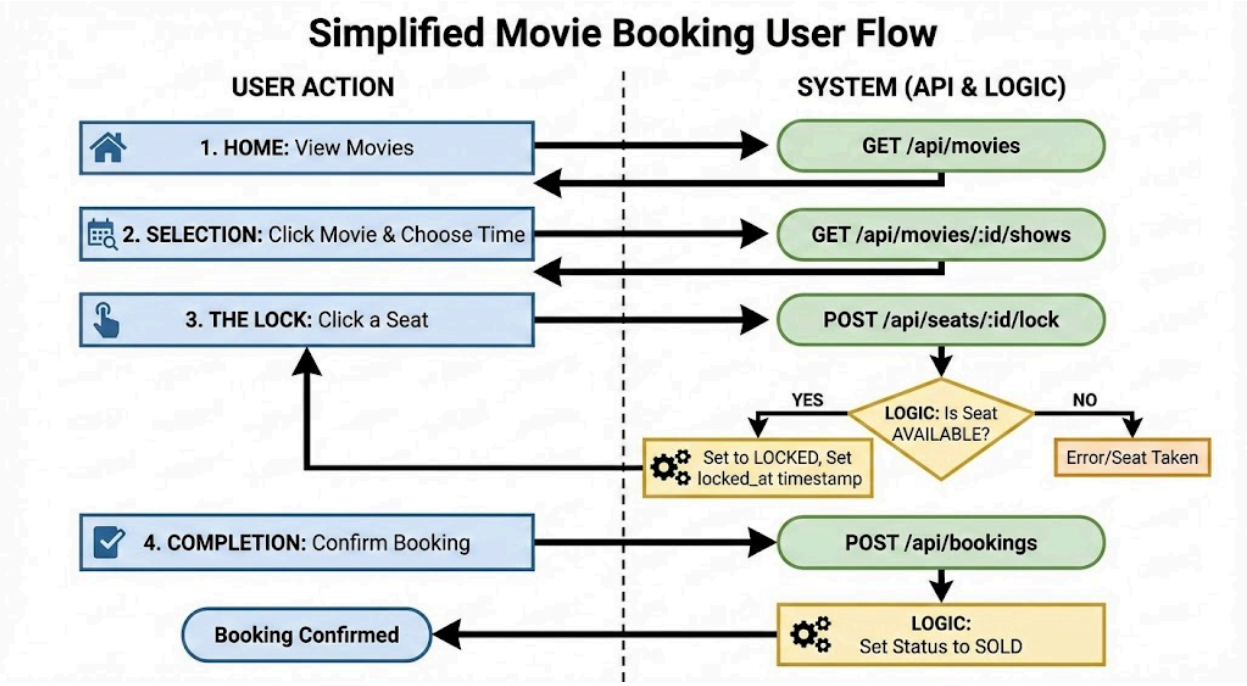
1. A containerized Go monolith
2. DB: MySQL
3. API: RestAPI
4. Deployment: Docker Compose for single-command environment parity.

Implementation Detail

Logic Flow

The core technical challenge is handled in the `POST /api/seats/:id/lock` endpoint.

1. **Open Transaction:** Start a database session.
2. **Row Lock:** Execute `SELECT * FROM show_seats WHERE id = ? FOR UPDATE`. This prevents any other process from even *reading* that seat's status until we are done.
3. **Validate:** Check if `status == 'AVAILABLE'` OR `(status == 'LOCKED' AND locked_at < 10mins ago)`.
4. **Update:** Set status to `LOCKED` and update `locked_at` to the current time.
5. **Commit:** Release the lock.



Rest API Detail

Method	Endpoint	Input	Success Output
POST	/api/v1/login	Email password	JWT Token & User Info
GET	/api/v1/movies	None	List of all movies.
GET	/api/v1/movies/:id/shows	movie_id	List of showtimes for that movie.
GET	/api/v1/shows/:id/seats	show_id	50-seat grid with current statuses.

PATCH	/api/v1/seats/:id/lock	seat_id	{ "message": "Locked" }
POST	/api/v1/bookings	seat_id, show_id	{ "message": "Booked" }

1. Authentication

- **Purpose: Verify user and provide a stateless JWT.**
- **Payload:** `{"email": "alex@example.com", "password": "###yr538nf>"}`
- **Response:**
  JSON

```
JSON

{
  "token": "eyJhbGciOiJIUzI1Ni...",
  "user": { "id": 1, "name": "Alex Smith" }
}
```

2. **Seat Locking (The Concurrency Core)**

- **Purpose:** Uses a `PATCH` to perform a partial update on the seat status.
- **Payload:** `{"status": "LOCKED"}`
- **Response:**
  JSON

```
JSON

      { "message": "Locked", "expires_at": "2024-05-20T10:10:00Z"
      }
```

3. Final Booking

- **Purpose:** Converts the temporary lock into a permanent sale.

- **Payload:** `{"show_id": 101, "seat_id": 501}`
- **Response:**
  JSON

```JSON
{
  "booking_id": 782,
  "status": "CONFIRMED",
  "message": "Ticket sent to your email."

      }
```

4. Movies: GET `/api/v1/movies`

- **Payload:** None
- **Response**: `[{"id": 1, "title": "...", "desc": "..."}]`

5. Show details: GET `/api/v1/movies/:id/shows`

- **Payload**:None (ID in URL)
- **Response**: `[{"id": 101, "theatre": "PVR", "time": "2024-..."}]`

6. Seat Status: GET`/api/v1/shows/:id/seats`

- **Payload**:None (ID in URL)
- **Response**: `[{"id": 501, "name": "A1", "status": "AVAILABLE"}]`

Database Updates

```Go
type Movie struct {
	ID                uint    `gorm:"primaryKey" json:"id"`
	Title             string  `json:"title"`
```

```
	Description		string	`json:"description"`
	DurationMins		int	`json:"duration_mins"`
	Content rating		string	`json:"rating"`
	Rating			string	`json:"rating"`
}

type Theatre struct {
	ID		uint		`gorm:"primaryKey" json:"id"`
	Name		string		`gorm:"not null" json:"name"`
	Location	string		`json:"location"`
}

type Show struct {
	ID			uint		`gorm:"primaryKey" json:"id"`
	MovieID			uint		`json:"movie_id"`
	TheatreId		string		`json:"theatre_id"`
	StartTime		time.Time	`json:"start_time"`
}

type User struct {
	ID		uint	`gorm:"primaryKey" json:"id"`
	Email		string	`gorm:"uniqueIndex;not null" json:"email"`
	PasswordHash	string	`json:"-"`
	Name		string	`json:"name"`
}

type ShowSeat struct {
	ID	uint		`gorm:"primaryKey" json:"id"`
	ShowID	uint		`gorm:"not null;index" json:"show_id"`
	SeatName string		`json:"seat_name"`
	Status	string		`json:"status"` // AVAILABLE, LOCKED, SOLD
	LockedAt *time.Time	`json:"locked_at"`
	UserID	*uint		`json:"user_id"` // WHO locked this seat?
}

type Booking struct {
	ID		uint		`gorm:"primaryKey" json:"id"`
	UserID		uint		`gorm:"not null;index" json:"user_id"`
	ShowID		uint		`gorm:"not null" json:"show_id"`
	SeatID		uint		`gorm:"not null" json:"seat_id"`
	CreatedAt	time.Time	`json:"created_at"`
}
```

Security Considerations

1. Statelessness: The API is designed to be stateless to support horizontal scaling.
2. Query Validation: All `POST` requests are validated to ensure `seat_id` and `show_id` exist before initiating a database lock.
3. Transaction Safety: Transactions are explicitly rolled back on error to prevent partial data writes or stale locks.
4. Authentication (JWT): The backend issues a JSON Web Token (JWT) upon successful login. This token must be sent in the `Authorization` header for any "Lock" or "Book" requests.
5. Password Hashing: Passwords are never stored in plain text. We use **bcrypt** to hash passwords before saving them to the MySQL database.

Cost Considerations

1. Resource Efficiency: Because Go is a compiled language with a small memory footprint, the entire stack can run on a single entry-level VPS (e.g., 1GB RAM).
2. Using indexes helps us in achieving faster queries which combined with small footprint of Go languages allows us to have faster response times.
3. Instead of paying for extra tools like a background 'cleaner' or a separate memory-bank (Redis), we do everything inside our existing Database. This keeps our setup simple, runs on cheaper servers, and means fewer things can break.