CONCEPT STUDY · INTERACTIVE PROTOTYPE

Slotwise: High-Concurrency Appointment Scheduling

Optimistic distributed slot reservation, client-side lease countdown timers, and zero-loss patient intake recovery for high-demand clinical practices.

INDUSTRY VERTICAL
Healthcare & Wellness

DEMONSTRATED FOCUS
Resource Booking & Specialist Scheduling

SIMULATED PROTOTYPE STACK
Next.js / TypeScript / React

PROPOSED PRODUCTION RUNTIME
React Native / Expo + Redis Leases

Executive Summary

In specialized medical centers and high-demand practices, scheduling windows create intense concurrency surges. Hundreds of patients access portals simultaneously to claim limited specialist appointments. Traditional booking architectures create double-booking collisions and form resets when multiple users attempt to secure the same appointment.

Slotwise implements an ephemeral distributed lease model backed by Redis keyspaces and client-side reactive state machines. Selecting a slot places a guaranteed 5-minute exclusive hold token while the patient completes clinical intake notes. If the timer lapses, the slot is gracefully recycled while the user's typed inputs are 100% preserved and mapped to adjacent openings in a single tap.

High-Concurrency Schedulers: Why Standard Systems Break

Appointment scheduling in healthcare is fundamentally different from generic ecommerce checkout. Patients are asked to input complex medical history, referral IDs, and symptom descriptions—a cognitive task that requires several minutes on a mobile device.

The Primary UX Pathologies of Relational Schedulers:

1. The Late-Stage Checkout Collision: In standard SQL architectures, slots remain "available" in the database until an `INSERT INTO appointments` statement executes. Two patients can both spend five minutes entering symptoms for the 09:30 AM slot, only for the second patient to be rejected at the final payment click.

2. Punitive Intake Discard: When the booking fails, the application re-routes the user to the blank calendar view. The patient loses their typed symptom notes and insurance verification data, triggering immediate booking abandonment.

3. Race Condition Double-Bookings: When two requests hit the API within a 50-millisecond window without distributed table locks, both can evaluate `is_available = true` and succeed, double-booking the physician.

Slotwise solves these systemic flaws by shifting from pessimistic final-checkout validation to **optimistic distributed reservation leasing**.

Architectural Blueprint: Ephemeral Redis Leases

The core technical innovation in Slotwise is the clean separation between *temporary reservation leases* and *permanent appointment records*:

```
// 1. Atomic Redis Lease Acquisition (Node.js / Redis)

const acquired = await redis.set(
  'slot:lease:' + providerId + ':' + slotTimestamp,
  JSON.stringify({ userId, intakeDraftId, heldAt: Date.now() }),
  'NX',
  'EX',
  300 // 5-minute hard TTL
);
```

Key architectural properties of this approach:

- **Zero Race Windows:** Redis single-threaded execution guarantees that only one caller receives `OK` for the key. Competitors immediately receive a `HELD_BY_ANOTHER` response code.
- **Self-Expiring Locks:** Unlike database row locks that risk hanging if the client drops connection, Redis TTL automatically cleans up expired leases with zero database garbage accumulation.
- **WebSocket Keyspace Notifications:** Redis publishes keyspace events on expiration, prompting all connected mobile clients to update their availability grids in real time.

User Experience Decisions: Graceful Recovery UX

Technical locking mechanisms are meaningless if the user experience creates anxiety. Slotwise designs every state to build trust:

1. Transparent Hold Badge

When a slot is tapped, a green banner secures the reservation: “Hold Secured: 09:30 AM (04:58)”. The ticking countdown provides psychological certainty.

2. Local Intake Auto-Save

Patient names, symptom descriptions, and contact info persist to local storage on every keystroke, decoupling form data from server session state.

3. Non-Destructive Expiration

When the timer expires, an informative amber card surfaces: “Slot released to waiting list, but your clinical intake is 100% saved.”

4. One-Tap Adjacent Swap

The app automatically evaluates Dr. Vance's next opening (10:15 AM) and provides a 1-tap swap button that re-locks the new slot with all intake intact.

Interface State Sequence: 5 Core Screens

Screen 01 · Availability Grid

Real-time appointment calendar showing Dr. Vance's open, high-demand, and booked morning slots.

`State: Slot Selection · Open`

Screen 02 · Clinical Intake

Symptom description and phone number input with continuous local auto-save status indicator.

`State: Auto-Saved Intake · Form Active`

Screen 03 · Active Hold Countdown

Prominent 5-minute lease timer, fee breakdown ($180.00, $25 copay), and checkout lock.

`State: Leased (04:58) · Exclusive Lock`

Screen 04 · Expiration & Swap

Amber notification indicating expired lease with recommended 10:15 AM slot and preserved intake.

`State: Expired · Suggested 10:15 AM`

Edge Case Resilience & Concurrency Guardrails

Resilience Matrix:

1. Network Dropout During Payment Handshake: If the patient loses cellular connectivity during Stripe token authorization, the lease remains protected for the duration of its 5-minute TTL. An idempotent charge token ensures payment is processed exactly once upon reconnect.

2. Simultaneous Multi-Client Claim: If 100 users click the same open slot at 09:00:00.001 AM, Redis atomic `SET NX` ensures that exactly 1 user succeeds. The remaining 99 users receive an instantaneous push notification rendering the slot “Held” without UI hang.

3. Provider Emergency Blockout: If a clinic administrator blocks out Dr. Vance's morning schedule, active lease holders are notified via WebSocket with priority rescheduling privileges before general release.

Technical Specification: Prototype vs. Native Runtime

Simulated Web Prototype (Delivered):

Built with Next.js 15, React 19, and TypeScript. Implements an in-memory client state machine that simulates Redis distributed lease acquisition, 1-second countdown ticks, expiration events, and intake form preservation.

Target Production Runtime (Specification):

- **Mobile Client:** React Native / Expo with TypeScript.
- **State Management:** TanStack Query + Zustand with persistent MMKV local storage.
- **Backend Gateway:** Node.js / Fastify microservices with Redis Cluster.
- **Distributed Locks:** Redlock algorithm across multi-node Redis instances for high availability.
- **Payments:** Stripe Payment Intents with pre-authorized hold captures.

Observed Metrics, Target Thresholds & Handover

OBSERVED CYCLE TIME

35 seconds

Average time to hold, complete intake, and confirm reservation in prototype.

INTAKE DATA RETENTION

100%

Zero form fields lost across all simulated timer expiration swaps.

Engineering Handover Terms:

Upon contract execution, FFNA delivers the complete React Native codebase, Redis lease orchestration lambdas, end-to-end load test scripts (k6 simulating 5,000 concurrent booking requests), and architectural documentation under full client intellectual property assignment.