Task & Rewards Management System

This project is a full-stack web application that models a real parent–child responsibility system. It focuses on role-based authentication, clean architecture, and backend-enforced rules.

Problem Statement

- I needed a structured way to assign and track responsibilities
- My son needed motivation and clear rewards
- Many apps mix roles or rely only on frontend enforcement

High-Level Solution

- Role-based system: Parent and Kid
- Tasks earn points
- Points redeem rewards
- Backend enforces all rules

Tech Stack

FRONTEND: REACT + TYPESCRIPT

BACKEND: .NET 8 MINIMAL API

DATABASE: SQLITE (LOCAL) / POSTGRESQL (PRODUCTION READY)

AUTHENTICATION: JWT (JSON WEB TOKENS)

High-Level Architecture

Architecture Overview

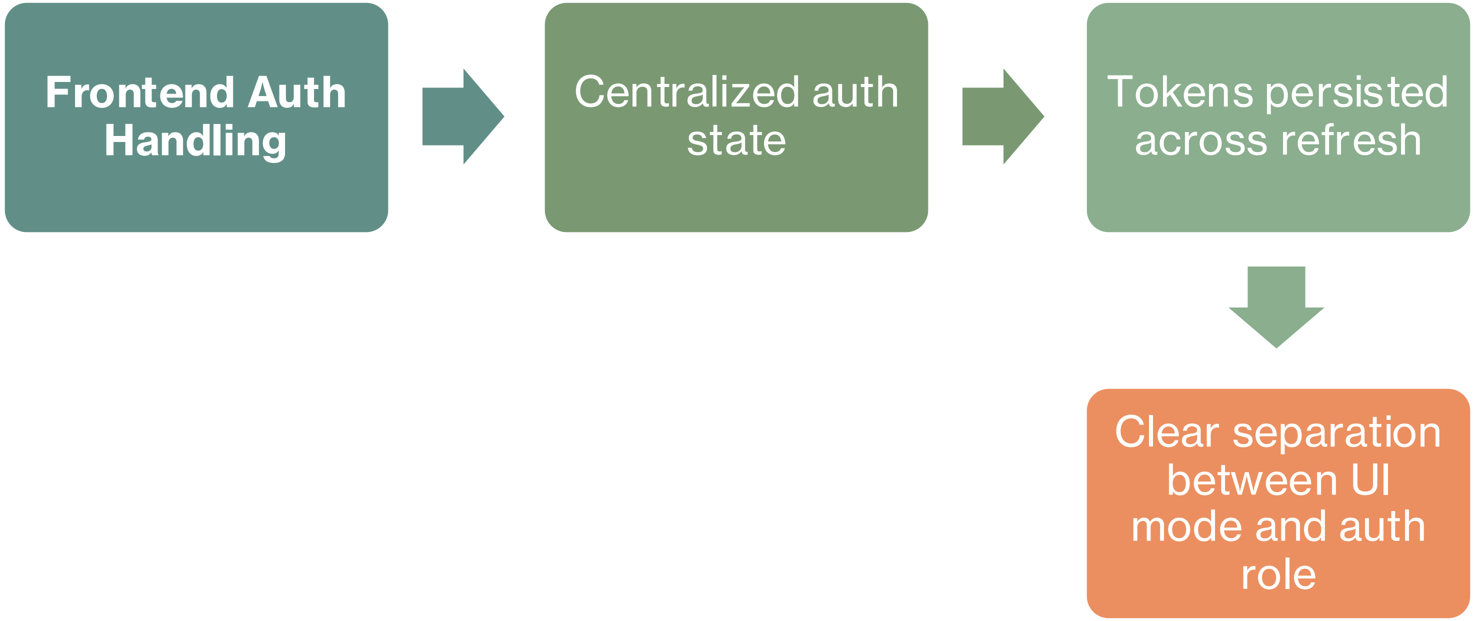
- React frontend communicates via REST APIs
- Shared API client attaches JWT automatically
- Backend validates JWT and enforces policies
- EF Core maps models to the database

Authentication & Roles

Authentication Design

- Parent login issues Parent JWT
- Kid sessions issue Kid-scoped JWT
- Tokens are stateless and role-aware

Frontend Auth State



Frontend Auth Handling

Centralized auth state

Tokens persisted across refresh

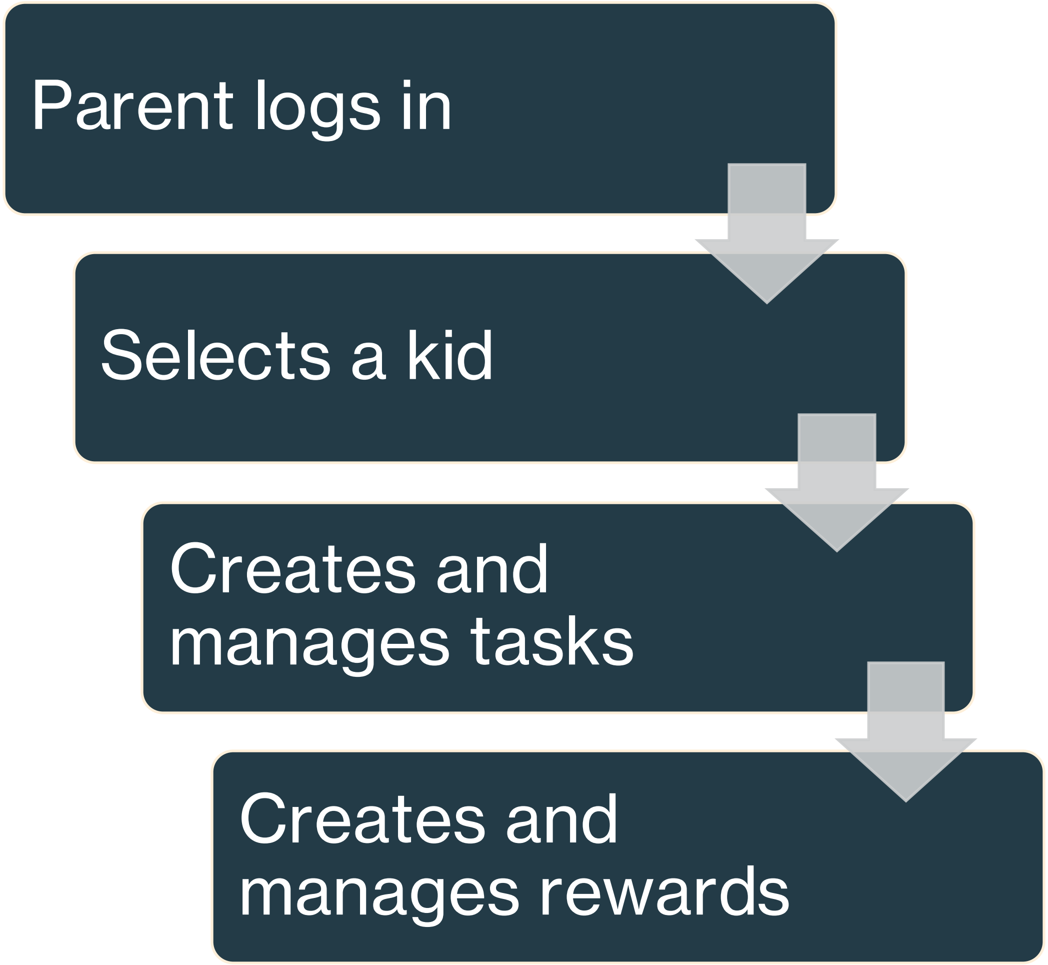
Clear separation between UI mode and auth role

Route Protection

Client-Side Route Guarding

- Routes protected by role
- Automatic redirects for wrong role
- Safe fallbacks

Parent User Flow



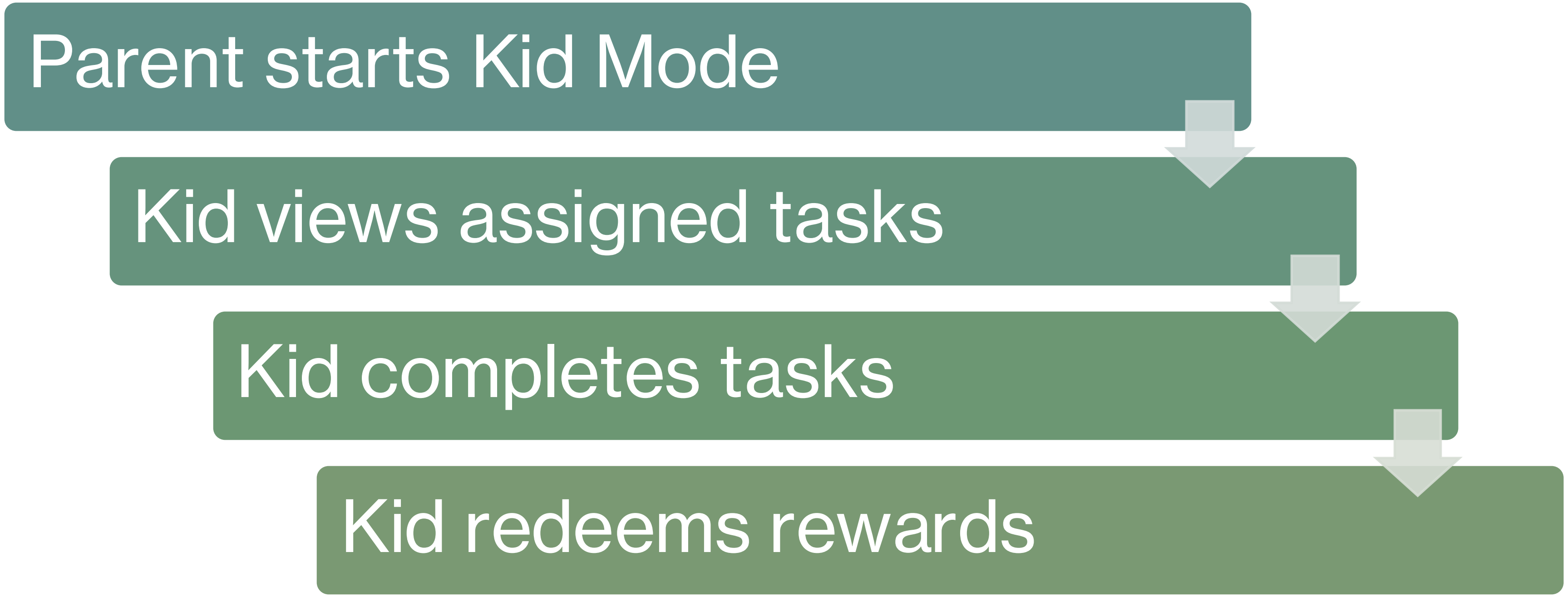
Parent logs in

Selects a kid

Creates and manages tasks

Creates and manages rewards

Kid User Flow



Parent starts Kid Mode

Kid views assigned tasks

Kid completes tasks

Kid redeems rewards

Role-Based UI

Single Page, Dual Behavior

- Same page supports Parent and Kid
- UI actions vary by role
- Backend always enforces permissions

Task Lifecycle

Tasks

1. Parent creates tasks
2. Tasks assigned to a kid
3. Kid completes task
4. Backend awards points

Rewards Lifecycle

Rewards

1. Parent defines rewards and costs
2. Kid redeems rewards
3. Backend validates points
4. Points deducted atomically

Points Ledger

POINTS & HISTORY

BALANCE STORED FOR FAST READS

LEDGER STORES FULL HISTORY

EARN AND SPEND TRANSACTIONS TRACKED

EF Core & Data Integrity

Entity Framework Core

- Maps models to tables
- Configures relationships and delete behavior
- Supports provider abstraction

Security Enforcement

Security Model

- ✓ JWT validated on every request
- Role-based authorization policies
- Ownership checks in endpoints

Why This Design Works

KEY STRENGTHS

CLEAR SEPARATION OF CONCERNS

STATELESS AUTHENTICATION

BACKEND-FIRST SECURITY

REAL-WORLD MODELING

Future Improvements

Next Steps

- Notifications
- Task scheduling
- Mobile-optimized UI
- Analytics for parents

Closing

SUMMARY

FULL-STACK OWNERSHIP

SECURE ROLE-BASED DESIGN

CLEAN, SCALABLE ARCHITECTURE

Code Reference Map

Where Things Live

Authentication state: AuthContext.tsx

HTTP client and JWT attachment: api.ts

Route protection: RequireRole.tsx

Core role-based UI: KidsRewardsPage.tsx

Data contracts: types.ts

Backend auth and policies: Program.cs

EF Core mappings and ledger: AppDbContext.cs

Interview Q&A: Authentication

Auth Design Decisions

Why JWT instead of server sessions?

- JWTs are stateless and scale well.
- Every request is self-contained and verifiable.

Why separate Parent and Kid tokens?

- Prevents role confusion.
- Makes backend authorization explicit and reliable.

Interview Q&A: Security

Preventing Abuse and Cheating

- UI checks are convenience only.
- Backend validates role and ownership on every request.
- Points are only modified server-side.
- Ledger entries are written atomically with balance updates.
- If someone tampers with the frontend, the backend still blocks invalid actions.

Interview Q&A: Frontend Design

Why One Page for Parent and Kid?

Using one page reduces duplication.

- Behavior is driven by role, not routes.
- Easier to maintain and extend.
- Backend remains the authority for enforcement.

Interview Q&A: Data Modeling

Why a Points Ledger?

Balances are optimized for fast reads.

- Ledger provides full history and auditability.
- EF Core enforces relationships and delete behavior.
- This pattern scales well as features grow.

Final Takeaway

What This Project Shows

- Real-world role-based system design
- Secure backend-first enforcement
- Thoughtful frontend architecture

A. Authorization & Role Enforcement

Files

- `Program.cs`
- `RequireRole.tsx`

Backend Code Reference (Program.cs)

"builder.Services.AddAuthorization(options =>"

Frontend Code Reference (RequireRole.tsx)

"if (required.length > 0 && (!activeRole ||
   !required.includes(activeRole))) {"

B. Frontend Architecture & API Layer

Files

- `api.ts`
- `AuthContext.tsx`

API Service Layer (api.ts)

```
"export const getTasks = async (kidId: string)
  =>"
```

Auth Sync Logic (AuthContext.tsx)

"SetApiRoleToken(auth.activeRole, auth);"

C. Routing, Mode Switching, and UI State

Files

- `App.tsx`
- `AuthContext.tsx`

Role vs UI Mode Separation (App.tsx)

```
"const isKidMode = auth?.uiMode === "Kid";"
```

Route Mirroring Logic (App.tsx)

```
"if (pathname.startsWith("/parent/kids"))"
```

D. Data Integrity & Transactions

File

- `Program.cs`

Task Completion Logic

"if (task.IsComplete) return Results.Ok(task);"

E. Points Ledger & EF Core Mapping

File

- `AppDbContext.cs`

Ledger Configuration

"modelBuilder.Entity<PointTransaction>(entity =>"

F. Scalability & Extension Readiness

Files

- `Program.cs`
- `Api.ts`

Stateless Auth Pipeline (Program.cs)

"app.UseAuthentication();"

API Ready for Growth (api.ts example)

"export const getTasks = async (kidId: string, page?: number) =>"