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

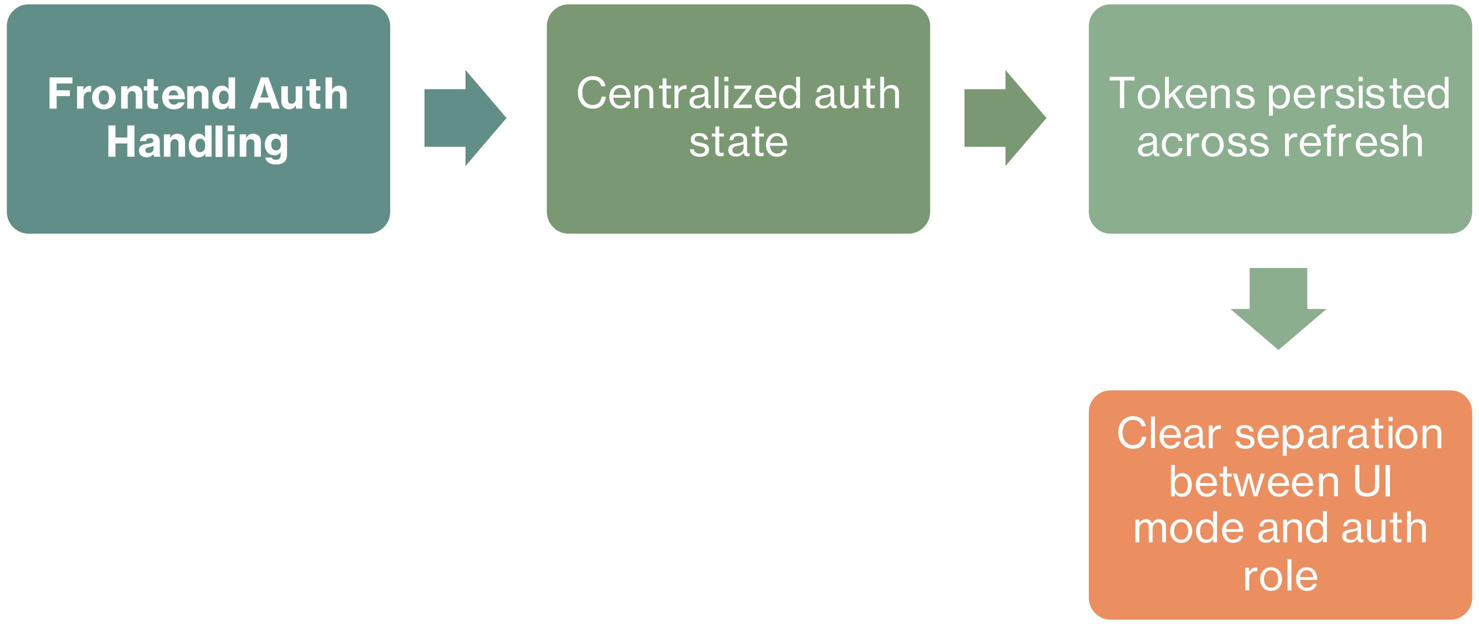
- React frontend communicates via REST APIs
- Shared API client attaches JWT automatically
- Backend validates JWT and enforces policies
- EF Core maps models to the database

Authentication & Roles

Authentication Design

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

Frontend Auth State

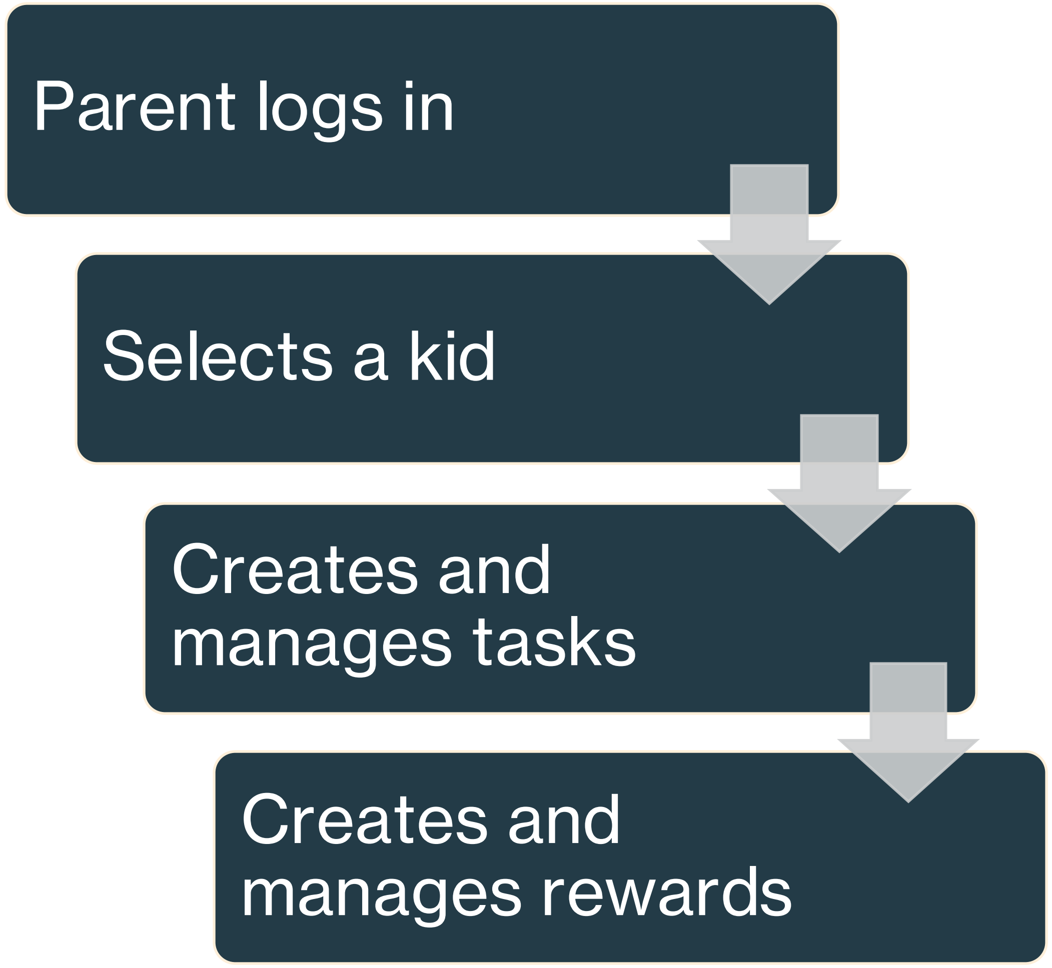

Frontend Auth Handling
Centralized auth state
Tokens persisted across refresh
Clear separation between UI mode and auth role

Route Protection

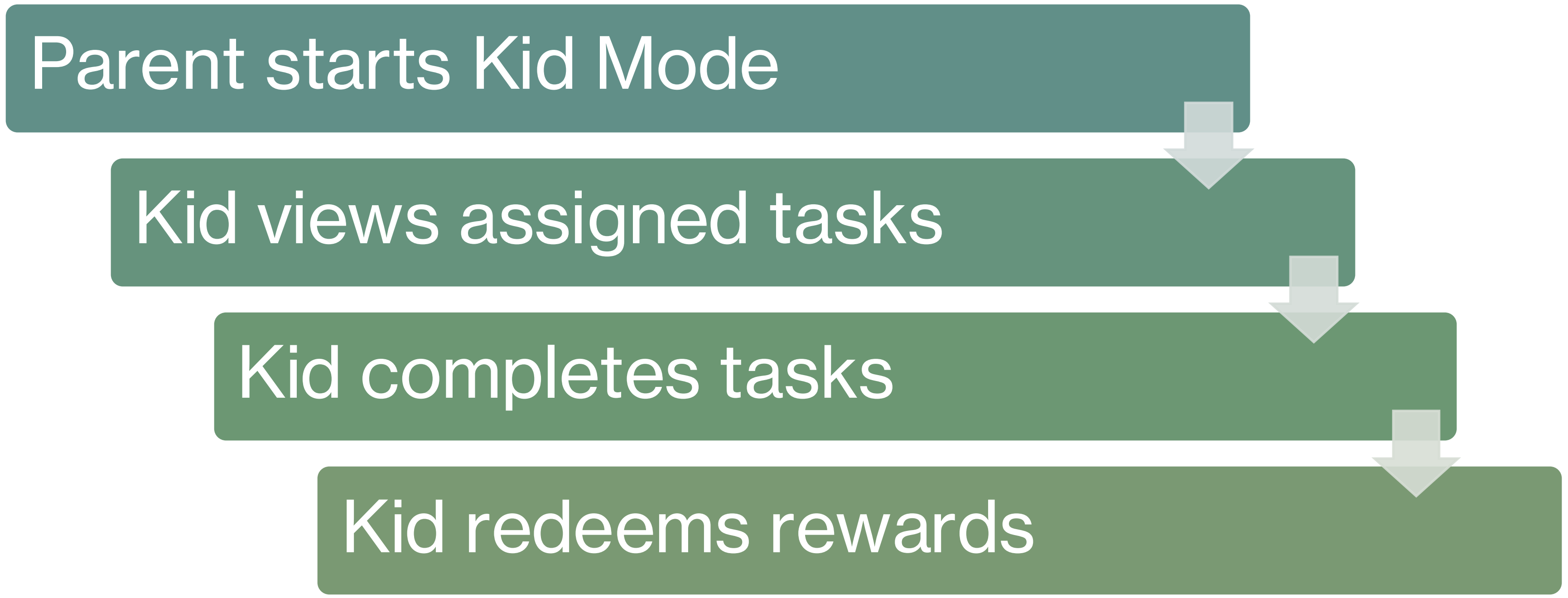
Client-Side Route Guarding

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

Parent User Flow



Kid User Flow



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

Tradeoffs

.NET 8 Minimal API vs MVC Controllers

Why I chose it: Minimal APIs reduce boilerplate and make endpoints easy to read and reason about. They are well suited for focused APIs and let me keep the HTTP surface clean.

Upside: Faster development, clearer routing, less ceremony.

Tradeoff: As features grow, structure can degrade if not enforced.

How I mitigate: Feature-based folders, endpoint grouping, DTO boundaries, and clear separation of concerns.

Tradeoffs

JWT Authentication (stateless) vs Server Sessions

Why I chose it: JWT works naturally with a SPA and API architecture and scales well because the server remains stateless.
Upside: No server-side session storage, easier horizontal scaling, clean API consumption.
Tradeoff: Token revocation is harder and token handling must be secure.
How I mitigate: Short-lived tokens, strict authorization checks, and careful client-side storage practices.

Tradeoffs

Role-based access (Parent vs Kid) vs Separate Applications

Why I chose it: A single application with role-based behavior keeps the system simpler and avoids duplicated logic.
Upside: Shared data model, shared infrastructure, consistent user experience.
Tradeoff: Increased authorization complexity and risk of exposing the wrong data.
How I mitigate: Claims-based role checks, server-side ownership enforcement, and UI gating backed by backend validation.

Tradeoffs

Entity Framework Core (ORM) vs Raw SQL

Why I chose it: EF Core improves productivity and maintainability while still allowing efficient queries.
Upside: Strong domain modeling, relationship handling, migrations, async LINQ queries.
Tradeoff: Poorly shaped queries can cause performance issues.
How I mitigate: DTO projection, disciplined use of includes, async queries, and reviewing generated SQL when needed.

Tradeoffs

SQLite (local) and PostgreSQL (production) vs One Database Everywhere

Why I chose it: SQLite keeps local development lightweight while PostgreSQL provides production-grade reliability and scalability.

Upside: Easy local setup with realistic production behavior.

Tradeoff: Minor differences in SQL behavior across providers.

How I mitigate: Keep queries portable and avoid provider-specific features unless required.

Tradeoffs

Frontend Service Layer vs Calling Axios Directly in Components

Why I chose it: Centralizing API logic improves consistency and maintainability.
Upside: Less duplication, consistent error handling, easier refactors.
Tradeoff: Slightly more upfront structure.
How I mitigate: Keep the service layer thin and focused on data access only.

Tradeoffs

Separate Tasks and Rewards Entities vs One Combined Model

Why I chose it: Tasks represent work, while rewards represent incentives. Separating them keeps responsibilities clear.

Upside: Cleaner domain model and easier future expansion.

Tradeoff: More tables and endpoints.

How I mitigate: Simple DTOs and well-defined relationships.

Tradeoffs

Optimistic Concurrency vs Last-Write-Wins

Why I chose it: Simpler concurrency handling fits the project’s scope while still supporting clean state transitions.
Upside: Easier implementation.
Tradeoff: Risk of overwriting changes.
How I mitigate: Timestamp tracking and designing actions like task completion as explicit state transitions.

Tradeoffs

Server-side Ownership Enforcement vs Client-side Filtering

Why I chose it: Security must be enforced on the backend, not trusted to the client.
Upside: Prevents unauthorized access even if the client is manipulated.
Tradeoff: More backend logic and consistent query patterns required.
How I mitigate: Always apply user and role scoping before filtering and paging.

Tradeoffs

Simple Cloud Deployment vs Full Cloud-native Infrastructure

Why I chose it: A simpler deployment pipeline allows faster iteration and stable demos.
Upside: Consistent environments and easy deployment.
Tradeoff: Less control and potential cold starts.
How I mitigate: Health checks, client-side retries, and environment-based configuration.

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) =>"