Technical Documentation - Dynamic Portfolio Dashboard

1. Technical Challenges & Solutions

1.1 API Integration Challenges

Challenge: Limited API Access

- Yahoo Finance lacks an official public API
- Google Finance data is not directly accessible
- Rate limits on financial data APIs

Solution:

```
private RATE_LIMIT_WINDOW = 60 * 1000; // 1 minute
private RATE_LIMIT_COUNT = 50; // max 50 requests per minute
private requestTimestamps: number[] = [];

private enforceRateLimit() {
  const now = Date.now();
  this.requestTimestamps = this.requestTimestamps.filter(
    (ts) => now - ts < this.RATE_LIMIT_WINDOW
  );

  if (this.requestTimestamps.length >= this.RATE_LIMIT_COUNT) {
    throw new Error("Rate limit exceeded");
  }

  this.requestTimestamps.push(now);
}
```

1.2 Real-time Data Updates

Challenge:

- Frequent API calls could hit rate limits
- Maintaining data freshness while being API efficient

- Managing state updates without performance impact

Solution:

- Implemented a 15-minute refresh interval
- Used React's `useEffect` for cleanup
- Added loading states and error handling

```
const REFRESH_INTERVAL = 900000; // 15 minutes

React.useEffect(() => {
  fetchPortfolio();
  const interval = setInterval(fetchPortfolio, REFRESH_INTERVAL);
  return () => clearInterval(interval);
}, []);
```

1.3 Data Transformation & State Management

Challenge:

- Complex data structure from multiple sources
- Need to calculate derived values (gains/losses)
- Maintaining data consistency

Solution:

- Created type-safe interfaces
- Implemented adapter pattern for data transformation
- Centralized data fetching logic

```
export type Stock = {
  name: string;
  purchasePrice: number;
  quantity: number;
  investment: number;
  portfolioWeight: number;
  exchange: "NSE" | "BSE";
```

```
  cmp: number;
  presentValue: number;
  gainLoss: number;
  peRatio: number;
  latestEarnings: string;
  sector: string;
};
```

1.4 Performance Optimization

Challenge:

- Large dataset rendering
- Multiple concurrent API calls
- Client-side calculations

Solution:

- Implemented virtualization for table rendering
- Used `Promise.all` for parallel API calls
- Memoized expensive calculations

```
const stockPromises = stockData.map((stock) =>
  this.fetchStockDataFromApi(stock.stockId)
);

const stockResults = await Promise.all(stockPromises);
```

1.5 Error Handling & Reliability

Challenge:

- API failures
- Data inconsistencies
- Network issues

Solution:

- Implemented comprehensive error boundaries
- Added retry mechanisms
- Graceful fallbacks for missing data

```
try {
  const result = await fetchPortfolio(stocks);
  setData(result);
  setError(null);
} catch (err: any) {
  setError(err.message || "Unknown error");
  // Implement retry logic here
}
```

1.6 Dark Mode Implementation

Challenge:

- Consistent theming across components
- Smooth transitions
- System preference detection

Solution:

- Used `next-themes` for theme management
- Implemented CSS variables for dynamic colors
- Added transition utilities

```
:root {
  --background: #ffffff;
  --foreground: #171717;
}

.dark {
  --background: #0a0a0a;
  --foreground: #ededed;
}
```

---

2. Architecture Decisions

2.1 Technology Stack Selection

- **Next.js**: For server-side rendering and API routes
- **TypeScript**: For type safety and better developer experience
- **TailwindCSS**: For efficient styling and dark mode support
- **Supabase**: For database and authentication

2.2 Project Structure

```
src/
├── app/          # Next.js app directory
├── components/   # Reusable UI components
├── lib/          # Utility functions
├── types/        # TypeScript definitions
└── backend/      # Backend services
    ├── service/  # Business logic
    └── type/     # Backend types
```

---

3. Future Improvements

1. **Caching Layer**
    - Implement Redis for API response caching
    - Add service worker for offline support
2. **Performance**
    - Add code splitting for large components
    - Implement progressive loading for historical data
3. **Features**

    - Add portfolio analytics and insights
    - Implement real-time WebSocket updates
    - Add export functionality for reports
4. **Testing**
    - Add unit tests for critical functions
    - Implement E2E testing with Cypress
    - Add performance benchmarking

---

4. Lessons Learned

1. **API Integration**
    - Always implement rate limiting
    - Use caching strategies
    - Have fallback data sources
2. **State Management**
    - Keep state updates predictable
    - Implement proper error boundaries
    - Use appropriate loading states
3. **Performance**
    - Monitor and optimize re-renders
    - Implement proper memoization
    - Use appropriate data structures
4. **TypeScript**
    - Strict type checking saves debugging time

- Interface-first development helps planning
- Proper type definitions improve maintenance