XGBoost Store Documentation

Overview

The `xgboost-store.ts` is a state management module built with Zustand that handles the core logic for the XGBoost Simulation Dashboard. It manages the application state, including real-time data fetching, simulation mode, and prediction functionality.

Table of Contents

Core Concepts

Dual Data Source Architecture

The store implements a dual data source pattern:

- **Real-time Mode**: Uses live process values (PV) from the mill's data acquisition system
- **Simulation Mode**: Uses user-adjustable slider values for what-if analysis

State Management

- Uses Zustand for state management with TypeScript support

- Implements a unidirectional data flow pattern
- Maintains a clean separation between state and UI components

State Structure

Main State Interface

```
interface XgboostState {
  // Parameters and their current values
  parameters: Parameter[]
  parameterBounds: ParameterBounds

  // Slider values (separate from PV values)
  sliderValues: Record<string, number>

  // Mode control
  isSimulationMode: boolean

  // Target and prediction data
  currentTarget: number | null
  currentPV: number | null
  targetData: TargetData[]

  // Model configuration
  modelName: string
  availableModels: string[]
  modelFeatures: string[] | null
  modelTarget: string | null
  lastTrained: string | null

  // Real-time data settings
  currentMill: number
  dataUpdateInterval: NodeJS.Timeout | null

  // Actions and methods...
}
```

Parameter Structure

```
interface Parameter {
  id: string
  name: string
  unit: string
  value: number
  trend: Array<{ timestamp: number; value: number }>
```

```
    color: string
    icon: string
  }
```

Target Data Structure

```
  interface TargetData {
    timestamp: number
    value: number
    target: number
    pv: number
    sp?: number | null  // Setpoint value
  }
```

Data Flow

Initialization

1. Store is created with default values
2. Model metadata is loaded
3. Real-time data updates begin if in real-time mode

Real-time Data Flow

1. `startRealTimeUpdates()` is called
2. Sets up an interval to fetch data every 30 seconds
3. `fetchRealTimeData()` is called on each interval
4. Data is processed and stored in the state
5. UI components re-render with updated values

Prediction Flow

1. User interacts with sliders or real-time data updates
2. `predictWithCurrentValues()` is called
3. Feature data is collected from appropriate source (PVs or sliders)

4. API call to `/api/v1/ml/predict` is made
5. Response updates the target value and trend data

Key Functions

`fetchRealTimeData()`

- Fetches current values for all model features
- Updates parameter trends
- Triggers predictions when new data is available

`predictWithCurrentValues()`

- Collects feature values from current state
- Makes prediction API call
- Updates target value and trend data
- Handles both simulation and real-time modes

`updateParameterFromRealData()`

- Updates parameter values from real-time data
- Preserves user-set values in simulation mode
- Maintains trend history

`resetFeatures()`

- Resets all parameters to their default values
- Updates trend data with new values
- Preserves simulation/real-time mode

Zustand Integration

The store uses several Zustand features:

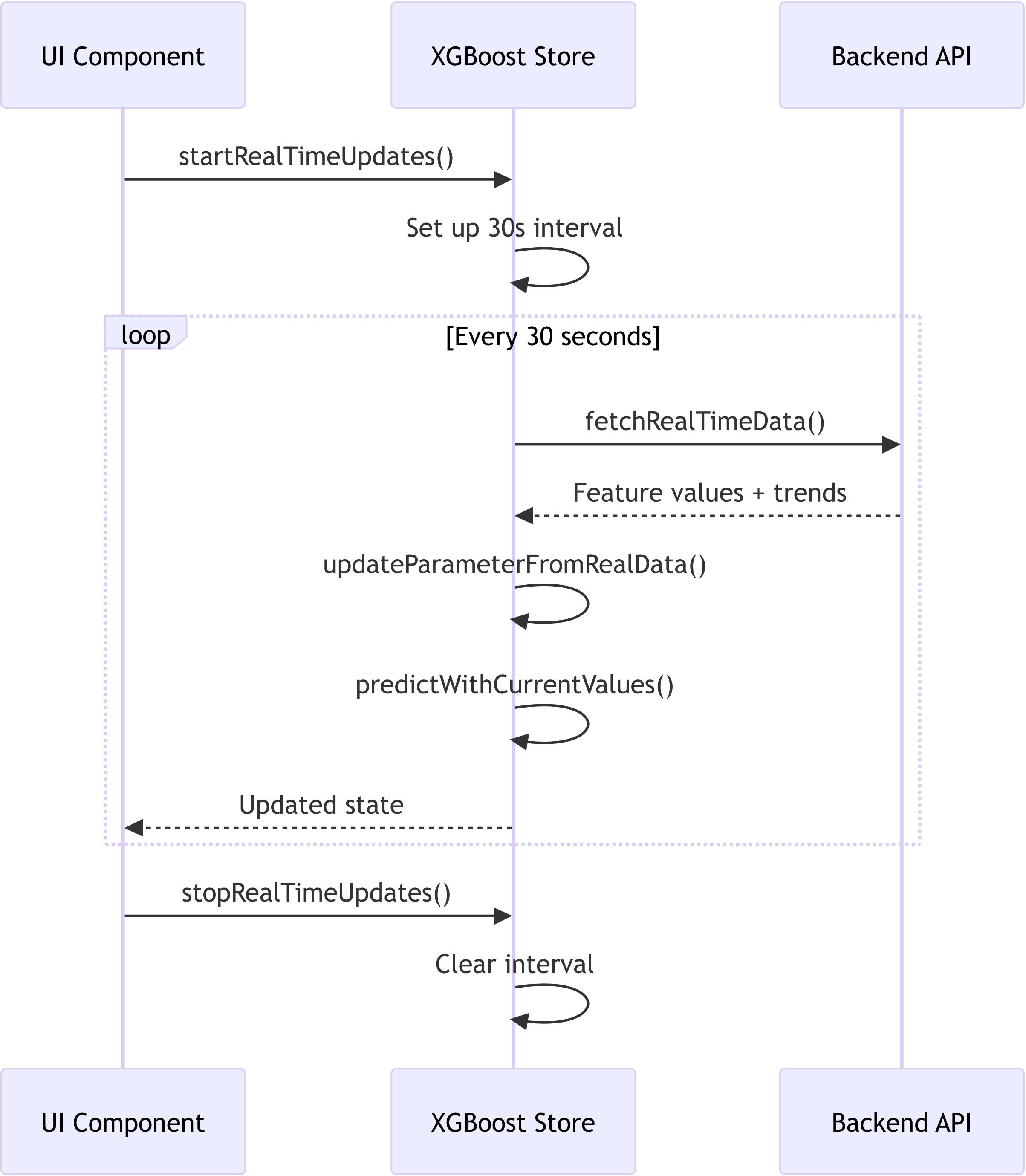
Middleware

- `devtools`: Enables Redux DevTools integration
- `persist`: (Commented out) Can be used for state persistence

State Updates

- Uses Zustand's `set` function for immutable updates
- Batches multiple updates for performance
- Implements selectors for optimized re-renders

Real-time Data Flow



Simulation Mode

Key Behaviors

- Toggled via `setSimulationMode()`
- In simulation mode:
  - Slider values are used for predictions
  - Real-time data continues to update trends
  - User adjustments are preserved
- In real-time mode:

  - Process values (PVs) are used for predictions
  - Sliders are updated to reflect current PVs

Implementation Details

- `isSimulationMode` state controls data source
- `sliderValues` object stores user adjustments
- `updateParameterFromRealData` respects simulation mode

Error Handling

API Errors

- Failed API calls are caught and logged
- State remains consistent on errors
- User receives feedback via console warnings

Data Validation

- Validates model features before prediction
- Handles missing or invalid data gracefully
- TypeScript ensures type safety

Performance Considerations

Optimization Techniques

- Batched state updates
- Debounced predictions in simulation mode
- Limited trend history (last 50 points)
- Selective re-renders with Zustand selectors

Memory Management

- Automatic cleanup of intervals
- Limited history size for trend data
- Efficient data structures for state

Best Practices

1. **State Updates**

    - Use the `set` function for all state updates
    - Batch related updates together
    - Keep state updates pure

2. **Data Fetching**

    - Handle loading and error states
    - Clean up async operations
    - Debounce rapid updates

3. **Testing**

    - Test state transitions
    - Mock API responses
    - Verify error handling

Troubleshooting

Common Issues

1. **Missing Data**

    - Verify model features are loaded
    - Check API connectivity
    - Validate tag mappings

2. **Stale State**

    - Ensure proper cleanup of intervals
    - Check for race conditions

    - Verify Zustand middleware setup

3. **Performance Problems**

    - Limit re-renders with selectors
    - Optimize data structures
    - Profile component updates

Future Improvements

1. **State Persistence**

    - Enable Zustand persistence
    - Add versioning for state migrations

2. **Enhanced Error Handling**

    - User-facing error messages
    - Retry mechanisms
    - Fallback behaviors

3. **Performance Optimizations**

    - Virtualized lists for trend data
    - WebSocket for real-time updates
    - Selective data subscriptions