XGBoost Store Documentation

Overview

The `xgboost-store.ts` is the central state management module for the XGBoost Simulation Dashboard, built using Zustand. It serves as the single source of truth for mill operation monitoring and simulation.

Key Responsibilities

1. **State Management**: Manages mill parameters, predictions, and UI states
2. **Data Flow**: Handles real-time data fetching and processing
3. **Simulation**: Enables what-if analysis with adjustable parameters
4. **Prediction**: Interfaces with ML models for operational insights
5. **Integration**: Connects with OPC UA data sources and ML APIs

Technical Stack

- **Framework**: Next.js with TypeScript
- **State Management**: Zustand with devtools
- **Data Fetching**: Custom API client (`mlApiClient`)
- **Styling**: Tailwind CSS
- **Visualization**: Recharts
- **API**: Custom ML endpoints for predictions

Table of Contents

Core Concepts

State Management Architecture

The store is built around several core concepts that work together to provide a robust solution for mill operation monitoring and simulation:

1. **Parameter Management**
    - Tracks all mill parameters (Ore, WaterMill, etc.) with their current values
    - Maintains historical trends for each parameter
    - Handles unit conversions and value formatting
    - Enforces parameter bounds to ensure values stay within operational limits
2. **Simulation Mode**
    - Toggle between real-time and simulation modes
    - Maintains separate slider values for simulation
    - Preserves real-time data while in simulation mode
    - Supports what-if analysis with adjustable parameters
3. **Prediction System**
    - Interfaces with ML models for operational predictions
    - Handles model loading and switching
    - Manages prediction requests and responses
    - Supports dynamic model features and metadata
4. **Real-time Data**
    - Fetches live data from mill systems
    - Updates parameters at regular intervals
    - Handles connection states and errors
    - Maintains data consistency across UI components

Core Concepts

Dual Data Source Architecture

The store implements a sophisticated dual data source pattern that enables seamless switching between real-time and simulation modes:

```
// Simplified example of the dual data source pattern
const getParameterValue = (parameter: Parameter, state: XgboostState) => {
  return state.isSimulationMode
    ? state.sliderValues[parameter.id] ?? parameter.value
    : parameter.value;
};
```

Real-time Mode:

- Connects to the mill's data acquisition system
- Fetches live process values (PV) at regular intervals
- Updates the UI in real-time with current process conditions
- Used for monitoring and operational decision-making

Simulation Mode:

- Uses user-adjustable slider values for what-if analysis
- Enables testing different scenarios without affecting live processes
- Maintains separate state for slider values to preserve real-time data
- Allows for predictive analysis and optimization

State Management with Zustand

The store leverages Zustand's powerful capabilities:

```
// Store creation with middleware
const useXgboostStore = create<XgboostState>()(
  devtools((set, get) => ({
    // State and actions...
  }))
);
```

Key Features:

- **Type Safety**: Full TypeScript support for all state and actions
- **Middleware**: Extensible with devtools and persistence
- **Reactivity**: Components only re-render when their subscribed state changes
- **Simplicity**: No need for context providers or wrapping components

State Structure

Main State Interface

The `XgboostState` interface defines the complete shape of the store's state. Here's a detailed breakdown of each property:

```
interface XgboostState {
  // Core Parameters
  parameters: Parameter[];          // Array of all process parameters with current values and trends
  parameterBounds: ParameterBounds;  // Min/max bounds for parameters

  // Simulation State
  sliderValues: Record<string, number>;  // User-adjusted values in simulation mode
  isSimulationMode: boolean;        // Whether simulation mode is active

  // Process Values
  currentTarget: number | null;     // Current target setpoint (SP)
  currentPV: number | null;         // Current process value (PV)
  targetData: TargetData[];         // Historical data for trending and analysis

  // Model Configuration
  modelName: string;                // Currently selected model name
  availableModels: string[];        // List of available model names
  modelFeatures: string[] | null;   // Features used by the current model
  modelTarget: string | null;       // Target variable name for the model
  lastTrained: string | null;       // Timestamp of last model training

  // Real-time Data
  currentMill: number;              // Currently selected mill (1-12)
  dataUpdateInterval: NodeJS.Timeout | null;  // Handle for the data update interval

  // Actions
  updateParameter: (id: string, value: number) => void;
  updateSliderValue: (id: string, value: number) => void;
  setSimulationMode: (isSimulation: boolean) => void;
  setPredictedTarget: (target: number | null) => void;
  addTargetDataPoint: (dataPoint: TargetData) => void;
  updateSimulatedPV: () => void;
  startSimulation: () => void;
  stopSimulation: () => void;
  setModelName: (modelName: string) => void;
  setAvailableModels: (models: string[]) => void;
  setModelMetadata: (features: string[], target: string, lastTrained: string) => void;
  setCurrentMill: (millNumber: number) => void;
  fetchRealTimeData: () => Promise<void>;
  updateParameterFromRealData: (featureName: string, value: number, timestamp: number) => void;
```

```
  resetFeatures: () => void;
  predictWithCurrentValues: () => Promise<void>;
}
```

Parameter Interface

```
interface Parameter {
  id: string;                    // Unique identifier (e.g., "Ore", "WaterMill")
  name: string;                  // Display name (e.g., "Ore Feed")
  unit: string;                  // Measurement unit (e.g., "t/h")
  value: number;                 // Current value
  trend: Array<{                 // Historical trend data
    timestamp: number;           // Unix timestamp
    value: number;               // Value at this timestamp
  }>;
  color: string;                 // Color for UI representation
  icon: string;                  // Icon for UI representation
}

// Parameter bounds configuration
type ParameterBounds = {
  [key: string]: [number, number]  // [min, max] bounds for each parameter
};
```

Target Data Interface

```
interface TargetData {
  timestamp: number;             // Unix timestamp
  value: number;                 // Actual value
  target: number;                // Target value (SP)
  pv: number;                    // Process value (PV)
  sp?: number | null;            // Optional setpoint value from prediction
}
```

State Initialization

The store's initial state includes default values for all parameters and configuration. Here are some key initialization details:

1. **Parameter Bounds Initialization**

```
const initialBounds: ParameterBounds = {
  Ore: [160.0, 200.0],
  WaterMill: [5.0, 20.0],
  WaterZumpf: [160.0, 250.0],
  PressureHC: [0.3, 0.5],
  DensityHC: [1600, 1900],
  MotorAmp: [180, 220],
  Shisti: [0.05, 0.3],
  Daiki: [0.1, 0.4],
  PumpRPM: [800, 1200],
  Grano: [0.5, 5.0],
  Class_12: [20, 60]
};
```

2. **Default Parameter Values**
   - Parameters are initialized with default values that are typically the midpoint of their bounds
   - Each parameter includes metadata like units, colors, and icons for UI representation
   - Trend data is initialized as an empty array
3. **Slider Values**
   - Separate slider values are maintained for simulation mode
   - These values can be adjusted independently of the real-time values

```
// Example of parameter initialization
const initialParameters = [
  {
    id: "Ore",
    name: "Ore Feed",
    unit: "t/h",
    value: 0,
    trend: [],
    color: "#3b82f6",  // blue-500
    icon: "⛏️",
  },
  {
    id: "WaterMill",
    name: "Mill Water",
    unit: "m³/h",
    value: 0,
    trend: [],
    color: "#06b6d4",  // cyan-500
    icon: "💧",
  },
```

```
  // ... other parameters
];
```

State Management Utilities

The store includes several utility functions for managing state:

1. **Parameter Updates**:
   - `updateParameter`: Updates a parameter's value
   - `updateSliderValue`: Updates a slider value in simulation mode
   - `updateParameterFromRealData`: Updates parameter from real-time data feed
2. **Simulation Control**:
   - `setSimulationMode`: Toggles between real-time and simulation modes
   - `startSimulation`: Starts the simulation mode
   - `stopSimulation`: Stops the simulation mode
   - `updateSimulatedPV`: Updates simulated process values
3. **Model Management**:
   - `setModelName`: Sets the active model
   - `setAvailableModels`: Updates the list of available models
   - `setModelMetadata`: Configures model features and target
4. **Data Operations**:
   - `fetchRealTimeData`: Fetches real-time data from the mill
   - `predictWithCurrentValues`: Makes predictions using the current parameter values
   - `resetFeatures`: Resets all parameters to their default values

```
### Parameter Structure

```typescript
interface Parameter {
  id: string;
  name: string;
  unit: string;
```
```

```
  value: number;
  trend: Array<{ timestamp: number; value: number }>;
  color: string;
  icon: string;
}
```

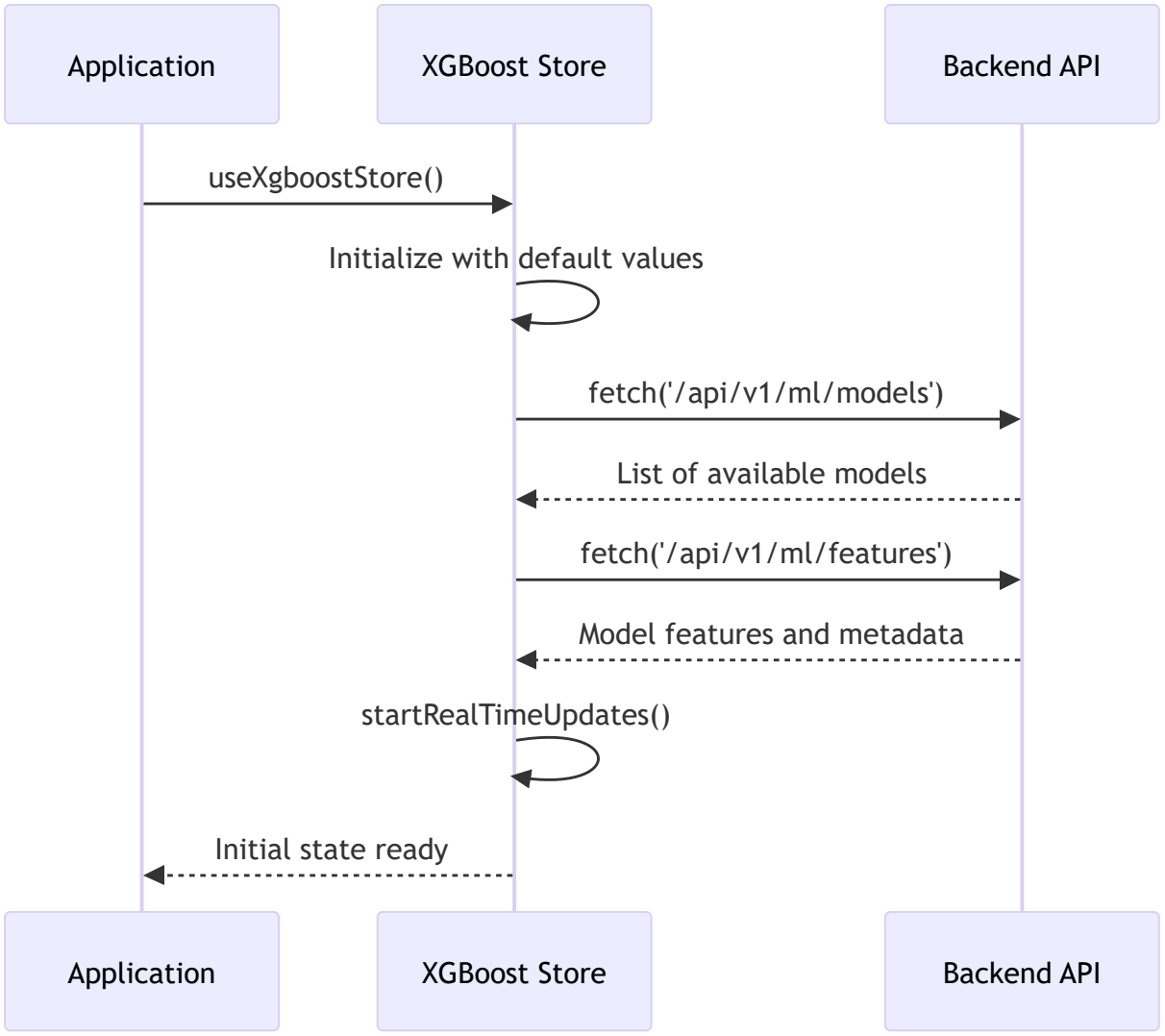
Target Data Structure

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

Data Flow

Initialization Flow

When the application starts, the store initializes with this sequence:



Real-time Data Flow

The real-time data flow is the backbone of the application's live monitoring capabilities:

1. **Initialization**:

```
// In a React component:
useEffect(() => {
  startRealTimeUpdates();
  return () => stopRealTimeUpdates();
}, [startRealTimeUpdates]);
```

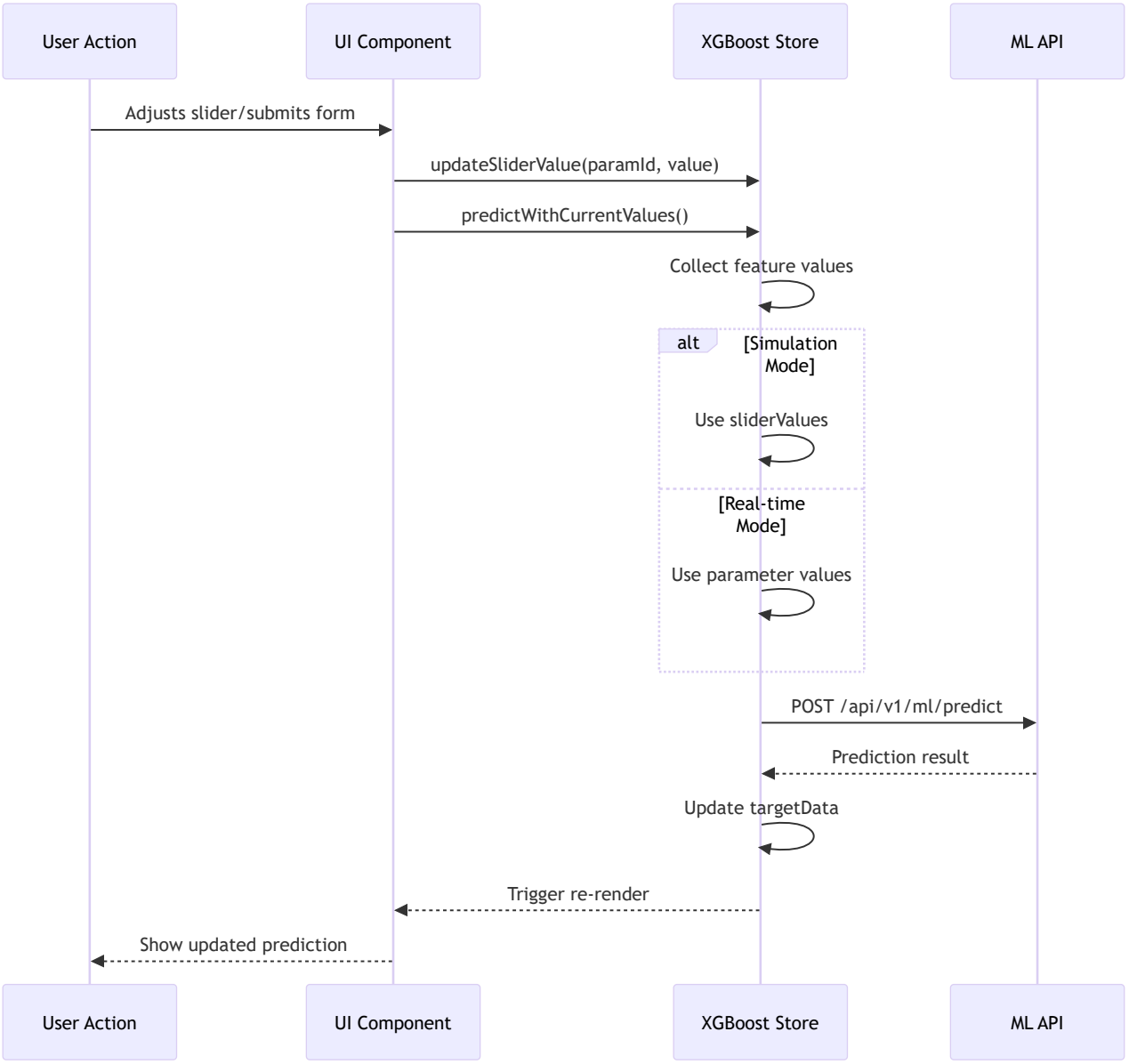
2. **Update Cycle**:

   - `startRealTimeUpdates()` sets up a 30-second interval
   - On each interval, `fetchRealTimeData()` is called
   - For each parameter, the store fetches current values and trends
   - State is updated with new values

- UI components re-render with updated data

Prediction Flow

The prediction flow is triggered by user interactions or real-time updates:



Key Functions

Data Management

`fetchRealTimeData()`

Asynchronously fetches and updates real-time data for all parameters.

```
fetchRealTimeData: () => Promise<void>
```

Features:

- Fetches current values for all parameters in parallel
- Updates parameter trends with timestamped values
- Handles mill-specific tag IDs and data normalization
- Manages error states and automatic retries
- Supports dynamic model features and metadata

Flow:

1. Gets current store state including mill number and parameters
2. For each parameter, fetches the latest value and trend data
3. Updates the store with new values while preserving historical data
4. Handles errors and connection states gracefully

updateParameterFromRealData

Updates a specific parameter with real-time data while maintaining historical trends.

```
updateParameterFromRealData: (
  featureName: string,
  value: number,
  timestamp: number,
  trend?: Array<{ timestamp: number; value: number }>
) => void
```

Behavior:

- Updates the parameter's current value
- Maintains a rolling window of the last 50 data points
- Ensures values stay within defined parameter bounds
- Handles special cases for different parameter types

predictWithCurrentValues

Makes predictions using the current parameter values and active model.

```
predictWithCurrentValues: () => Promise<void>
```

Flow:

1. Collects current parameter values (from either real-time or simulation)
2. Validates input values against parameter bounds
3. Sends request to ML API with current state
4. Updates UI with prediction results and confidence intervals
5. Handles loading states and errors

Parameters:

- Uses current store state including:
  - `parameters`: Current parameter values
  - `isSimulationMode`: Whether in simulation mode
  - `sliderValues`: Current slider values (in simulation mode)
  - `modelName`: Name of the active ML model
  - `currentMill`: ID of the current mill

Error Handling:

- Validates required parameters
- Handles API errors gracefully
- Provides user feedback for error conditions

`updateParameter`

Updates a parameter's value and maintains its trend history.

```
updateParameter: (id: string, value: number) => void
```

Parameters:

- `id`: The ID of the parameter to update (e.g., "Ore", "WaterMill")
- `value`: The new value for the parameter

Behavior:

- Updates the specified parameter's current value
- Maintains a rolling window of the last 50 historical values
- Automatically timestamps each update
- Triggers UI updates for any components using this parameter

`updateSliderValue`

Updates the value of a slider in simulation mode without affecting real-time values.

```
updateSliderValue: (id: string, value: number) => void
```

Parameters:

- `id`: The ID of the slider/parameter to update
- `value`: The new slider value

Behavior:

- Only updates the slider's visual representation
- Doesn't affect real-time parameter values
- Used exclusively in simulation mode
- Maintains separate state from real-time parameter values

`setSimulationMode`

Controls whether the application is in simulation mode or real-time mode.

```
setSimulationMode: (isSimulation: boolean) => void
```

Parameters:

- `isSimulation`: Boolean indicating whether to enable simulation mode

Behavior:

- Toggles between real-time data and simulation data views
- Preserves current parameter states when switching modes

- Enables/disables real-time data fetching when toggled

setPredictedTarget

Updates the current target value for the prediction system.

```
setPredictedTarget: (target: number) => void
```

Parameters:

- `target`: The new target value to predict

Behavior:

- Updates the current target value in the store
- Triggers UI updates for components displaying the target
- Used to set the reference value for prediction comparisons

addTargetDataPoint

Adds a new data point to the target data history.

```
addTargetDataPoint: (dataPoint: Omit<TargetData, 'pv'>) => void
```

Parameters:

- `dataPoint`: Object containing timestamp, value, and target information

Behavior:

- Adds a new data point to the target data array
- Automatically generates a simulated PV value around 50
- Maintains a rolling window of the last 50 data points
- Updates the current PV value in the store

updateSimulatedPV

Updates the simulated process value (PV) with random variation.

```
updateSimulatedPV: () => void
```

Behavior:

- Only runs when simulation is active
- Generates a new PV value with controlled random variation (±1 from current PV)
- Keeps PV values within the 45-55 range
- Updates the target data array with the new PV value
- Maintains a rolling window of the last 50 data points
- Only updates when simulation is active

`startSimulation` and `stopSimulation`

Controls the active state of the simulation.

```
startSimulation: () => void
stopSimulation: () => void
```

Behavior:

- `startSimulation`: Activates the simulation mode and begins periodic updates
- `stopSimulation`: Deactivates the simulation mode and stops periodic updates
- Both functions update the `simulationActive` state flag
- Used to control the simulation lifecycle and UI state

`setModelName` and `setAvailableModels`

Manages model selection and availability.

```
setModelName: (modelName: string) => void
setAvailableModels: (models: string[]) => void
```

Parameters:

- `modelName`: The name/ID of the model to use for predictions
- `models`: Array of available model names/IDs

Behavior:

- `setModelName`: Updates the active model for predictions
- `setAvailableModels`: Updates the list of available models in the system
- Triggers UI updates to reflect model changes
- May trigger metadata loading for the selected model

`setModelMetadata`

Configures the model's metadata including features and target variable.

```
setModelMetadata: (
  features: string[],
  target: string,
  lastTrained: string | null
) => void
```

Parameters:

- `features`: Array of feature names used by the model
- `target`: The name of the target variable
- `lastTrained`: ISO timestamp of when the model was last trained

Behavior:

- Updates the model's feature set and target variable
- Automatically adds any missing parameters to the store
- Initializes new parameters with sensible defaults
- Updates the UI to reflect available features and their current values
- Maintains existing parameter values when possible

`setCurrentMill`

Updates the currently selected mill in the application.

```
setCurrentMill: (millNumber: number) => void
```

Parameters:

- `millNumber`: The numeric ID of the mill to select

Behavior:

- Updates the current mill in the application state
- Triggers a refresh of real-time data for the new mill
- May affect which tag IDs are used for data fetching
- Updates any mill-specific UI components

`fetchRealTimeData`

Fetches real-time data for all model features from the data source.

```
fetchRealTimeData: () => Promise<void>
```

Behavior:

- Fetches current values for all model features
- Retrieves historical trend data for each parameter
- Handles feature name mapping between model and data source
- Updates the store with fresh parameter values and trends
- Includes error handling and logging for debugging
- Skips fetching if no model features are loaded
- Maintains data consistency during updates

`updateParameterFromRealData`

Updates parameter values with real-time data while respecting simulation mode.

```
updateParameterFromRealData: (
  featureName: string,
  value: number,
  timestamp: number,
```

```
  trend?: Array<{ timestamp: number; value: number }>
) => void
```

Parameters:

- `featureName`: The ID of the parameter to update (e.g., "Ore", "WaterMill")
- `value`: The new value for the parameter
- `timestamp`: Unix timestamp of the data point
- `trend`: Optional array of historical values (defaults to empty array)

Behavior:

- Only updates parameters that exist in the store
- Maintains separate values for simulation and real-time modes
- Preserves historical trend data (keeps last 50 data points)
- Handles parameter bounds validation
- Respects simulation mode (won't update actual values in simulation mode)
- Updates parameter trends in both real-time and simulation modes
  newParameters[paramIndex] = updatedParam;

  return { parameters: newParameters }; }); };

````
### `resetFeatures()`

Resets all parameters to their default values while preserving the current mode:

```typescript
const resetFeatures = () => {
  set((state) => {
    // Reset parameters to their default values (middle of min/max range)
    const updatedParameters = state.parameters.map((param) => {
      const bounds = state.parameterBounds[param.id];
      const defaultValue = bounds ? (bounds.min + bounds.max) / 2 : 0;

      return {
        ...param,
        value: defaultValue,
        // Preserve trend data
        trend:
          param.trend.length > 0
            ? [...param.trend]
            : [{ timestamp: Date.now(), value: defaultValue }],
````
````

```
      };
    });

    // Reset slider values to match new parameter values
    const updatedSliderValues = { ...state.sliderValues };
    updatedParameters.forEach((param) => {
      updatedSliderValues[param.id] = param.value;
    });

    return {
      parameters: updatedParameters,
      sliderValues: updatedSliderValues,
    };
  });
};
```

`startRealTimeUpdates()` and `stopRealTimeUpdates()`

Manages the real-time update interval:

```
const startRealTimeUpdates = () => {
  // Clear any existing interval
  const state = get();
  if (state.dataUpdateInterval) {
    clearInterval(state.dataUpdateInterval);
  }

  // Initial data fetch
  fetchRealTimeData().catch(console.error);

  // Set up new interval (30 seconds)
  const interval = setInterval(() => {
    fetchRealTimeData().catch(console.error);
  }, 30000);

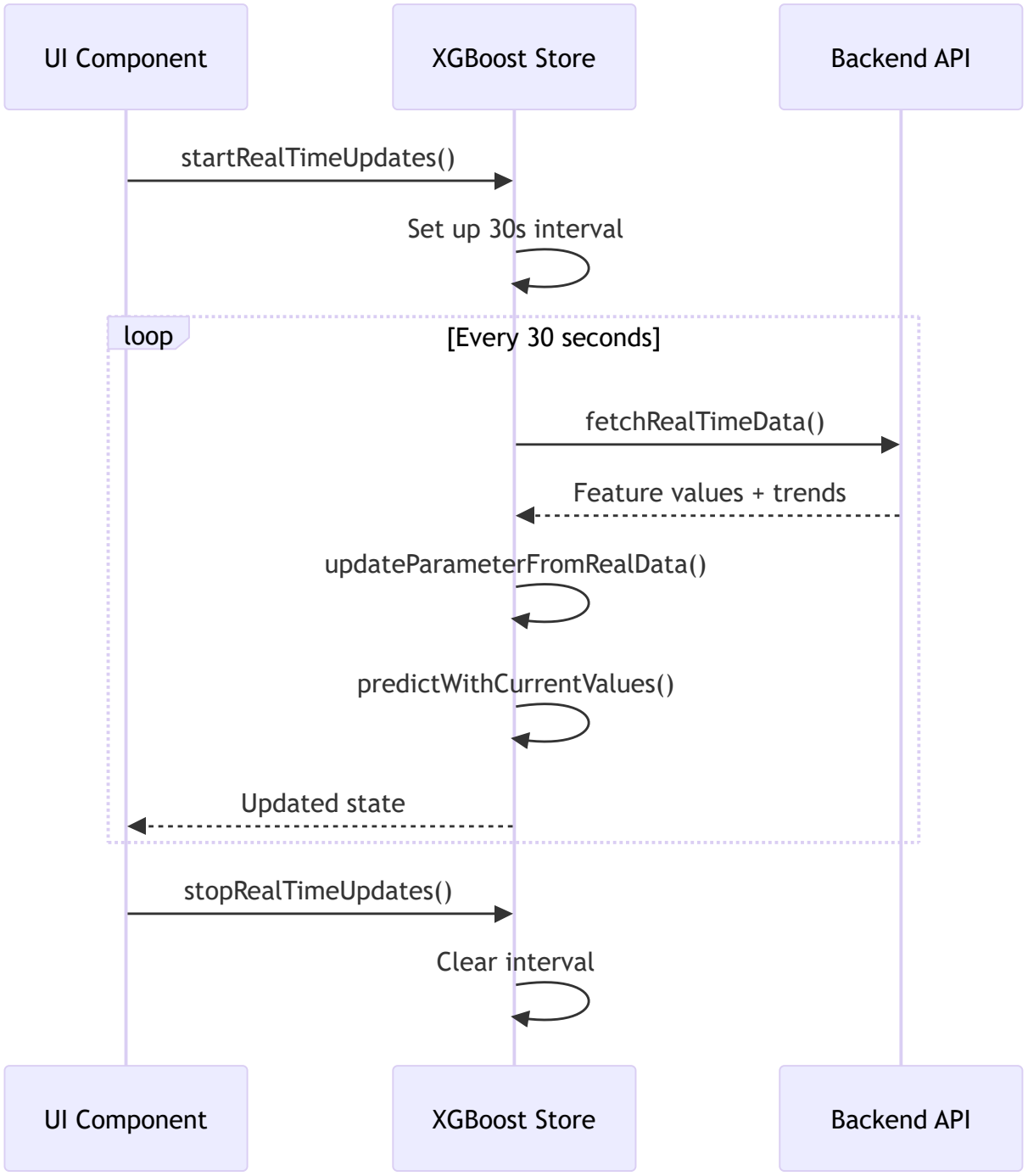
  // Store interval ID in state
  set({ dataUpdateInterval: interval });
};

const stopRealTimeUpdates = () => {
  const state = get();
  if (state.dataUpdateInterval) {
    clearInterval(state.dataUpdateInterval);
    set({ dataUpdateInterval: null });
  }
   // In component:
   const currentMill = useXgboostStore((state) => state.currentMill);

   // Memoized selector
   const getParameter = (id: string) =>
```

```
useXgboostStore(
  useCallback((state) => state.parameters.find((p) => p.id === id), [id])
);
```

Real-time Data Flow

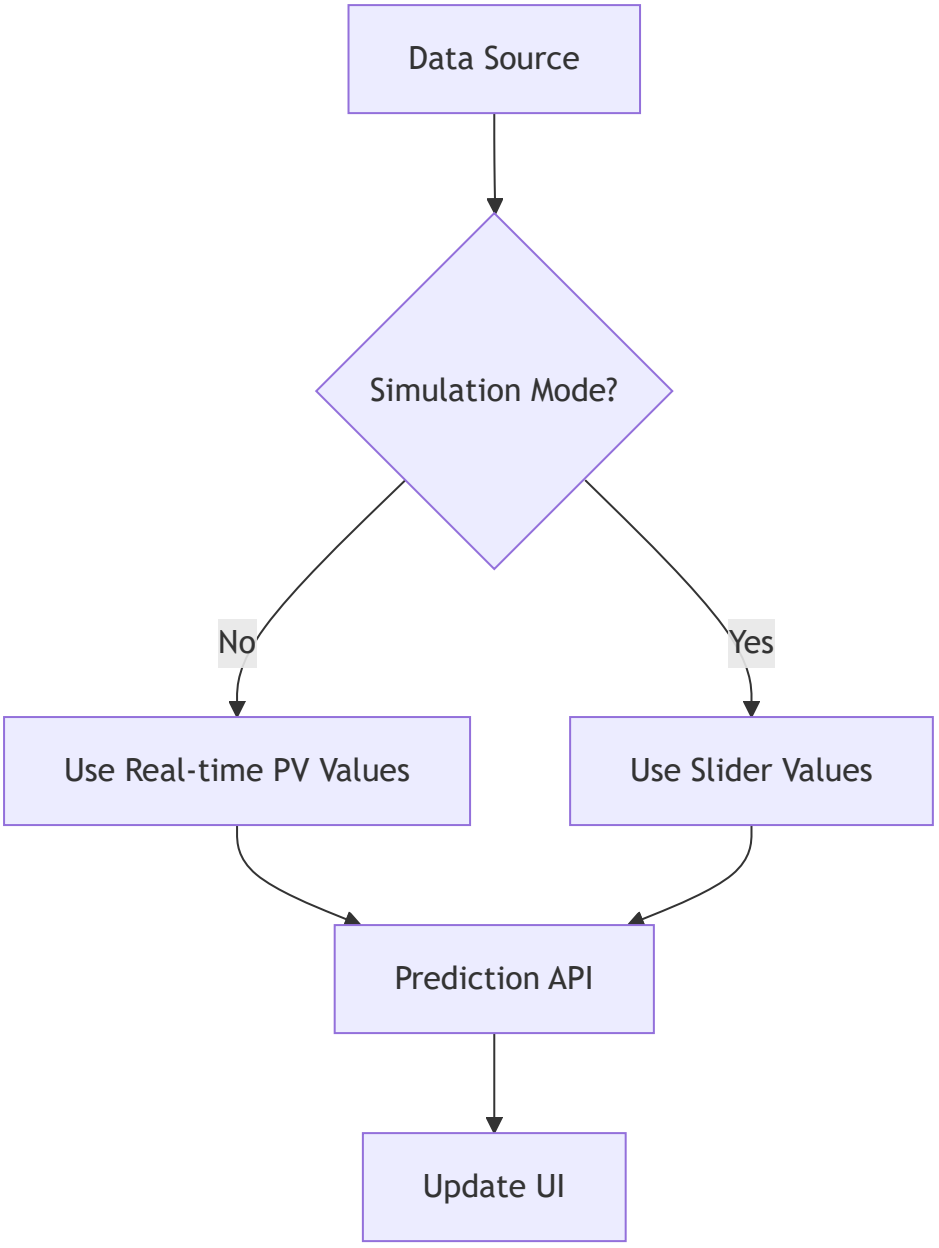


Simulation Mode

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

Architecture Overview



Key Components

1. **State Management**

```
interface XgboostState {
  // Core state
  isSimulationMode: boolean;
  sliderValues: Record<string, number>;
  parameters: Parameter[];

  // Actions
  setSimulationMode: (isSimulation: boolean) => void;
  updateSliderValue: (id: string, value: number) => void;
  // ... other actions
}
```

2. **Mode Switching**

```
// Toggle between real-time and simulation modes
const toggleSimulationMode = () => {
  useXgboostStore.setState(prev => ({
    isSimulationMode: !prev.isSimulationMode
  }));
};
```

Data Flow

1. **Real-time Mode**
   - Fetches live process values from OPC UA server
   - Updates parameter trends with real-time data
   - Uses actual PV values for predictions
2. **Simulation Mode**
   - Uses user-adjustable slider values
   - Preserves real-time data in read-only format
   - Enables what-if analysis without affecting live process

Implementation Details

- `isSimulationMode` state controls data source
- `sliderValues` object stores user adjustments

- `updateParameterFromRealData` respects simulation mode

Error Handling

Robust error handling is implemented throughout the XGBoost store to ensure application stability and provide a good user experience. The error handling strategy is designed to be both developer-friendly and user-centric.

Error Types and Handling

1. **API Request Failures**
    - Network connectivity issues
    - Server errors (5xx)
    - Timeout handling
    - Authentication/authorization errors
2. **Data Validation**
    - Type checking for all parameters
    - Range validation for numerical values
    - Required field validation
    - Data format verification
3. **State Consistency**
    - Race condition prevention
    - Transactional state updates
    - Rollback on failure

Implementation Details

1. API Error Handling

```
// In xgboost-store.ts
const fetchRealTimeData = async () => {
  const state = get();
  const { currentMill, parameters } = state;
```

```
  try {
    // Process each parameter with error boundaries
    await Promise.all(
      parameters.map(async (param) => {
        try {
          const tagId = getTagId(param.id, currentMill);
          if (!tagId) return;

          const value = await fetchTagValue(tagId);
          const trend = await fetchTagTrend(tagId, '1h');

          updateParameterFromRealData(param.id, value, Date.now(), trend);
        } catch (error) {
          console.error(`Error updating parameter ${param.id}:`, error);
          // Continue with other parameters even if one fails
        }
      })
    );

    // Update last successful fetch time
    set({ lastFetch: Date.now() });

  } catch (error) {
    console.error('Fatal error in fetchRealTimeData:', error);
    // Consider setting a global error state
  }
};
```

2. Error Boundaries in React Components

```
// ErrorBoundary.tsx
import { Component, ErrorInfo, ReactNode } from 'react';

interface Props {
  children: ReactNode;
  fallback?: ReactNode;
}

interface State {
  hasError: boolean;
  error: Error | null;
}

export class ErrorBoundary extends Component<Props, State> {
  public state: State = {
    hasError: false,
    error: null
  };

  public static getDerivedStateFromError(error: Error): State {
    return { hasError: true, error };
  }
```

```
  public componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    console.error('Uncaught error:', error, errorInfo);
    // Log to error reporting service
  }

  public render() {
    if (this.state.hasError) {
      return this.props.fallback || (
        <div className="error-boundary">
          <h2>Something went wrong</h2>
          <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
          </details>
          <button onClick={() => window.location.reload()}>
            Reload Application
          </button>
        </div>
      );
    }

    return this.props.children;
  }
}
```

3. Error Recovery Strategies

- **Automatic Retries**:

```
const fetchWithRetry = async <T,>(
  fn: () => Promise<T>,
  retries = 3,
  delay = 1000
): Promise<T> => {
  try {
    return await fn();
  } catch (error) {
    if (retries === 0) throw error;
    await new Promise(resolve => setTimeout(resolve, delay));
    return fetchWithRetry(fn, retries - 1, delay * 2);
  }
};
```

```
try {
  return await fn();
} catch (error) {
  lastError = error;
  if (i < maxRetries - 1) {
    await new Promise(resolve => setTimeout(resolve, delay * (i + 1)));
```

```
    }
  }
```

}

throw lastError!; };

```

```

2. **Fallback Values**

```
const getParameterValue = (id: string): number => {
  try {
    const param = get().parameters.find(p => p.id === id);
    if (!param) throw new Error(`Parameter ${id} not found`);
    return param.value;
  } catch (error) {
    console.error('Error getting parameter value:', error);
    return getDefaultParameterValue(id);
  }
};
```

Error Logging

1. **Client-Side Logging**

```
const logErrorToService = (
  error: Error,
  context: Record<string, any> = {}
) => {
  const errorInfo = {
    timestamp: new Date().toISOString(),
    message: error.message,
    stack: error.stack,
    context: {
      ...context,
      userAgent: navigator.userAgent,
      url: window.location.href,
      state: getRelevantState()
    }
  };

  // Send to error tracking service
  fetch('/api/log-error', {
    method: 'POST',
```

```
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(errorInfo)
  }).catch(console.error);

  console.error('Logged error:', errorInfo);
};
```

2. **Error Boundaries with Context**

```
const ErrorBoundaryWithContext: React.FC<ErrorBoundaryProps> = ({
  children,
  context = {}
}) => {
  return (
    <ErrorBoundary
      onError={(error, errorInfo) => {
        logErrorToService(error, { ...context, errorInfo });
      }}
    >
      {children}
    </ErrorBoundary>
  );
};
```

User Feedback

1. **Error Toast Notifications**

```
const showErrorToast = (message: string, options = {}) => {
  toast.error(message, {
    position: 'top-right',
    autoClose: 5000,
    hideProgressBar: false,
    closeOnClick: true,
    pauseOnHover: true,
    draggable: true,
    ...options
  });
};

// Usage
try {
  await someAsyncOperation();
} catch (error) {
  showErrorToast('Operation failed. Please try again.');
  throw error; // Re-throw for error boundaries
}
```

2. **Error Recovery UI**

```
const DataDisplay = () => {
  const { data, error, isLoading, retry } = useData();

  if (isLoading) return <LoadingSpinner />;
  if (error) {
    return (
      <div className="error-state">
        <Alert variant="error">
          Failed to load data: {error.message}
        </Alert>
        <Button onClick={retry}>
          Retry
        </Button>
      </div>
    );
  }

  return <DataVisualization data={data} />;
};
```

Best Practices

1. **Error Boundaries**
   - Wrap component trees with error boundaries
   - Provide helpful error UIs
   - Log errors to monitoring services
2. **Graceful Degradation**
   - Show fallback UIs when features fail
   - Disable non-critical features that depend on failed operations
   - Provide clear recovery paths
3. **Monitoring and Alerting**
   - Track error rates and patterns
   - Set up alerts for critical errors
   - Monitor user impact
4. **Testing**
   - Test error scenarios

    - Verify error boundaries
    - Check error recovery flows

- TypeScript ensures type safety

Performance Considerations

Optimizing performance is critical for a responsive user experience, especially when dealing with real-time data and complex state management. The XGBoost store implements several performance optimizations to ensure smooth operation.

State Management Optimizations

1. **Selective State Subscriptions**

```
// Component only re-renders when specific parameter changes
const oreValue = useXgboostStore(useCallback(
  state => state.parameters.find(p => p.id === 'Ore')?.value,
  []
));

// Memoize complex selectors
const getFilteredParameters = useMemo(
  () => useXgboostStore(state =>
    state.parameters.filter(p => p.value > 0)
  ),
  []
);
```

2. **Batched Updates**

```
// Multiple updates in a single state transition
const updateMultipleParameters = (updates: Array<{id: string, value: number}>)
=> {
```typescript
// Keep only the last 50 data points
const addDataPoint = (newPoint) => {
  set(state => ({
    trendData: [...state.trendData.slice(-49), newPoint]
  }));
};
```
```

3. **Efficient Data Structures**

```
// Use Maps for fast lookups
const parameterMap = useMemo(
  () => new Map(parameters.map(p => [p.id, p])),
  [parameters]
);

// Use Sets for membership checks
const activeParameterIds = useMemo(
  () => new Set(parameters.filter(p => p.isActive).map(p => p.id)),
  [parameters]
);
```

Rendering Optimizations

1. **Memoization**

```
// Memoize expensive calculations
const processedData = useMemo(() => {
  return parameters.map(processParameter);
}, [parameters]);

// Memoize components
const ParameterList = React.memo(({ parameters }) => (
  <div>
    {parameters.map(param => (
      <ParameterItem key={param.id} parameter={param} />
    ))}
  </div>
));
```

2. **Virtualized Lists**

```
// For long lists, use virtualization
import { FixedSizeList as List } from 'react-window';

const VirtualizedList = ({ items }) => (
  <List
    height={400}
    itemCount={items.length}
    itemSize={50}
    width="100%"
  >
    {({ index, style }) => (
      <div style={style}>
        {items[index].name}
```

```
      </div>
    )}
  </List>
);
```

Network Optimizations

1. **Request Deduplication**

```
// Cache in-flight requests
const requestCache = new Map();

async function fetchWithCache(url) {
  if (requestCache.has(url)) {
    return requestCache.get(url);
  }

  const promise = fetch(url).then(res => res.json());
  requestCache.set(url, promise);

  try {
    return await promise;
  } finally {
    requestCache.delete(url);
  }
}
```

2. **Request Prioritization**

```
// Critical data first
const fetchCriticalData = async () => {
  const [userPrefs, initialData] = await Promise.all([
    fetchUserPreferences(),
    fetchInitialData(),
  ]);

  // Load non-critical data after initial render
  requestIdleCallback(() => {
    fetchSecondaryData();
  });

  return { userPrefs, initialData };
};
```

Memory Management

1. **Cleanup Effects**

```
useEffect(() => {
  const interval = setInterval(updateData, 30000);

  // Cleanup function
  return () => {
    clearInterval(interval);
  };
}, []);
```

2. **Event Listener Optimization**

```
useEffect(() => {
  const handleResize = debounce(() => {
    setDimensions({
      width: window.innerWidth,
      height: window.innerHeight,
    });
  }, 250);

  window.addEventListener('resize', handleResize);
  return () => window.removeEventListener('resize', handleResize);
}, []);
```

Performance Monitoring

1. **React DevTools Profiler**

```
import { Profiler } from 'react';

const onRender = (id, phase, actualDuration) => {
  if (actualDuration > 100) {
    console.warn(`Slow render (${actualDuration}ms) in ${id}`);
  }
};

<Profiler id="Dashboard" onRender={onRender}>
  <Dashboard />
</Profiler>
```

2. **Performance Metrics**

```
// Measure critical operations
const measure = (label, fn) => {
  performance.mark(`${label}-start`);
  const result = fn();
  performance.mark(`${label}-end`);

  performance.measure(
    label,
    `${label}-start`,
    `${label}-end`
  );

  const measures = performance.getEntriesByName(label);
  console.log(`${label} took ${measures[0].duration}ms`);

  return result;
};
```

Best Practices

1. **Code Splitting**

```
// Lazy load non-critical components
const HeavyComponent = React.lazy(() => import('./HeavyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <HeavyComponent />
    </Suspense>
  );
}
```

2. **Web Workers**

```
// Offload heavy computations to a web worker
const worker = new Worker('worker.js');

worker.postMessage({ type: 'COMPUTE', data: largeDataSet });

worker.onmessage = (event) => {
  if (event.data.type === 'RESULT') {
    setResult(event.data.result);
  }
};
```

3. **React.memo and useMemo**

```
// Only re-render when props change
const ExpensiveComponent = React.memo(({ data }) => {
  // Component implementation
}, (prevProps, nextProps) => {
  // Custom comparison function
  return prevProps.data.id === nextProps.data.id;
});

// Memoize expensive calculations
const processedData = useMemo(() => {
  return processLargeDataset(data);
}, [data]);
```

4. **Avoid Inline Functions**

```
// Bad: Creates new function on every render
<button onClick={() => handleClick(id)}>Click me</button>

// Good: Memoize callback
const handleClick = useCallback((id) => {
  // Handle click
}, []);

<button onClick={handleClick}>Click me</button>
```

These performance optimizations ensure that the XGBoost dashboard remains responsive and efficient, even when dealing with large datasets and frequent updates.

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