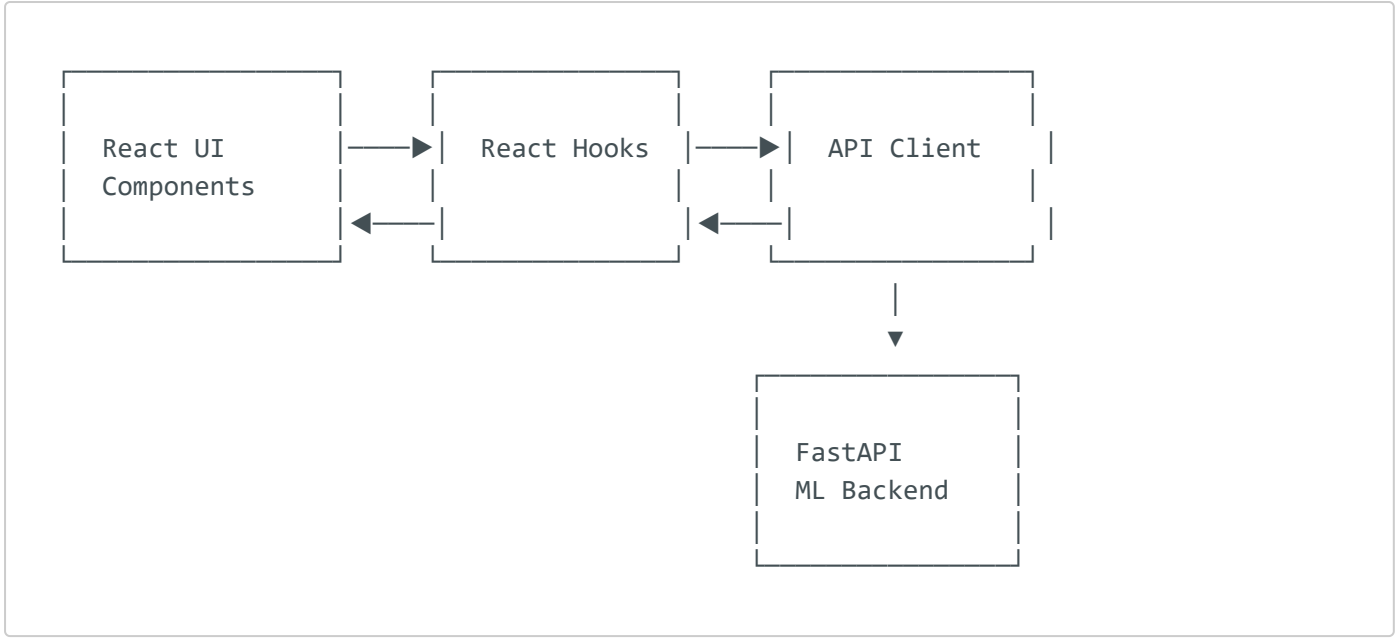
Mills-AI Model Training System

This document provides a comprehensive explanation of the Model Training UI system, including its architecture, component structure, data flow, API integration, and state management.

System Architecture Overview

The Model Training system is structured as a modern React application with the following key components:

1. **UI Components**: A hierarchical structure of React components that handle user interactions
2. **Custom Hooks**: React hooks for state management and API communication
3. **API Client**: Axios-based client for communicating with the backend ML service
4. **Backend Integration**: Connection to a FastAPI-based ML backend for training XGBoost models



Component Structure

1. Page Component

The entry point is `model-training/page.tsx`, which serves as a container for the entire training interface:

```
// src/app/mills-ai/model-training/page.tsx
"use client"

import { ModelTrainingDashboard } from "../components/model-training-dashboard"

export default function ModelTrainingPage() {
  return (
    <div className="min-h-screen bg-gradient-to-br from-slate-50 to-slate-100
dark:from-slate-900 dark:to-slate-800 p-4">
      <div className="max-w-[1600px] mx-auto">
        <div className="mb-6">
          <h1 className="text-3xl font-bold text-slate-900 dark:text-slate-
100">XGBoost Model Training</h1>
          <p className="text-slate-600 dark:text-slate-400 mt-2">
            Configure features, targets, and train your process optimization model
          </p>
        </div>
        <ModelTrainingDashboard />
      </div>
    </div>
  )
}
```

2. Main Dashboard Component

The `ModelTrainingDashboard` component (`components/model-training-dashboard.tsx`) is the core component that:

- Manages the overall state of the training system
- Coordinates between feature configuration and result display
- Handles the training process

```
// src/app/mills-ai/components/model-training-dashboard.tsx (simplified)
export function ModelTrainingDashboard() {
  const [parameters, setParameters] = useState<ModelParameter[]>(initialParameters)
  const [isTraining, setIsTraining] = useState(false)
  const [trainingResults, setTrainingResults] = useState<TrainingResults | null>
(null)
  const { trainModel, isLoading, progress: trainingProgress, error: trainingError }
= useModelTraining()

  const handleParameterUpdate = (updatedParameter: ModelParameter) => {
    setParameters((prev) => prev.map((p) => (p.id === updatedParameter.id ?
updatedParameter : p)))
  }

  const handleTrainModel = async () => {
    try {
      setIsTraining(true)
      setTrainingResults(null)

      // Call the API to train the model
      const results = await trainModel(parameters)

      // Update UI with results
      setTrainingResults(results)
    } catch (err) {
      console.error('Error during model training:', err)
    } finally {
      setIsTraining(false)
    }
  }

  // Component rendering with FeatureTargetConfiguration and ModelTrainingResults
  return (
    // UI Layout with Cards, Buttons, and child components...
  )
}
```

3. Feature & Target Configuration Component

The `FeatureTargetConfiguration` component (`components/feature-target-configuration.tsx`) allows users to:

- Select features for training
- Choose a target variable
- Configure parameter bounds
- Toggle feature/target status

```
// src/app/mills-ai/components/feature-target-configuration.tsx (simplified)
export function FeatureTargetConfiguration({
  parameters,
  onParameterUpdate
}: FeatureTargetConfigurationProps) {

  // Group parameters by type (feature, target, disabled)
  const features = parameters.filter((p) => p.type === "feature")
  const targets = parameters.filter((p) => p.type === "target")
  const disabled = parameters.filter((p) => p.type === "disabled")

  const handleParameterTypeChange = (parameter: ModelParameter, type:
ModelParameterType) => {
    // Ensure only one target can be selected
    if (type === "target") {
      onParameterUpdate({
        ...parameter,
        type,
        enabled: true,
      })

      // Disable other targets
      targets.forEach((t) => {
        if (t.id !== parameter.id) {
          onParameterUpdate({
            ...t,
            type: "disabled",
            enabled: false,
          })
        }
      })
    } else {
      onParameterUpdate({
        ...parameter,
        type,
        enabled: type === "feature",
      })
    }
  }

  // Component rendering with tabs for Features, Targets, and Disabled parameters
  return (
    // UI Layout with parameter cards, switches, and controls...
```

```
  )
}
```

4. Training Results Component

The `ModelTrainingResults` component (`components/model-training-results.tsx`) visualizes:

- Training metrics (MAE, MSE, RMSE, R2, MAPE)
- Feature importance
- Training/validation loss curves

```
// src/app/mills-ai/components/model-training-results.tsx (simplified)
export function ModelTrainingResults({ results }: ModelTrainingResultsProps) {
  if (!results) return null

  return (
    <div className="space-y-6">
      {/* Performance metrics badges */}
      <div className="flex flex-wrap gap-3">
        <MetricBadge label="MAE" value={results.mae} />
        <MetricBadge label="MSE" value={results.mse} />
        <MetricBadge label="RMSE" value={results.rmse} />
        <MetricBadge label="R²" value={results.r2} />
        <MetricBadge label="MAPE" value={`${results.mape.toFixed(2)}%`} />
        <MetricBadge label="Training Time" value=
{`${results.trainingTime.toFixed(2)}s`} />
      </div>

      {/* Feature importance chart */}
      <div className="space-y-2">
        <h3 className="text-lg font-medium text-slate-900 dark:text-slate-
100">Feature Importance</h3>
        <div className="bg-white dark:bg-slate-800 rounded-md p-4">
          <ResponsiveContainer width="100%" height={300}>
            <BarChart data={results.featureImportance} layout="vertical">
              {/* Chart components */}
            </BarChart>
          </ResponsiveContainer>
        </div>
      </div>

      {/* Validation curves chart */}
      <div className="space-y-2">
        <h3 className="text-lg font-medium text-slate-900 dark:text-slate-
100">Validation Curves</h3>
        <div className="bg-white dark:bg-slate-800 rounded-md p-4">
          <ResponsiveContainer width="100%" height={300}>
            <LineChart data={results.validationCurve}>
              {/* Chart components */}
```

```
            </LineChart>
          </ResponsiveContainer>
        </div>
      </div>
    </div>
  )
}
```

API Integration & Data Flow

1. API Client

The `ml-client.ts` file establishes the connection to the FastAPI backend:

```
// src/app/mills-ai/api/ml-client.ts
import axios from 'axios';

// Create API client pointing to the FastAPI backend
const apiClient = axios.create({
  baseURL: 'http://localhost:8000/api/v1/ml',  // Direct connection to FastAPI ML router
  headers: {
    'Content-Type': 'application/json'
  }
});

export default apiClient;
```

2. Model Training Hook

The `useModelTraining.ts` custom hook manages:

- API request preparation
- Training progress simulation
- Error handling
- Result transformation

```
// src/app/mills-ai/hooks/useModelTraining.ts (simplified)
export function useModelTraining() {
  const [isLoading, setIsLoading] = useState(false);
  const [progress, setProgress] = useState(0);
  const [results, setResults] = useState<TrainingResults | null>(null);
```

```
  const [error, setError] = useState<string | null>(null);

  // Prepare training request payload from UI parameters
  const prepareTrainingRequest = (parameters: ModelParameter[]): TrainModelRequest
=> {
    const enabledFeatures = parameters
      .filter(p => p.type === 'feature' && p.enabled)
      .map(p => p.id);

    const targetParam = parameters.find(p => p.type === 'target' && p.enabled);

    // Default database configuration (would be replaced with real config in
production)
    return {
      db_config: {
        host: "em-m-db4.ellatzite-med.com",
        port: 5432,
        dbname: "em_pulse_data",
        user: "s.lyubenov",
        password: "tP9uB7sH7mK6zA7t"
      },
      mill_number: 8,
      start_date: "2025-06-10T06:00:00",
      end_date: "2025-07-11T22:00:00",
      features: enabledFeatures,
      target_col: targetParam?.id || 'PSI80',
      test_size: 0.2,
      params: {
        n_estimators: 300,
        learning_rate: 0.05,
        max_depth: 6,
        subsample: 0.8,
        colsample_bytree: 0.8,
        early_stopping_rounds: 30,
        objective: "reg:squarederror"
      }
    };
  };

  // Process the API response into UI-friendly format
  const processTrainingResponse = (response: TrainModelResponse): TrainingResults
=> {
    const testMetrics = response.test_metrics;

    // Extract feature importance from API response
    const featureImportance = Object.entries(response.feature_importance).map(
      ([feature, importance]) => ({
        feature,
        importance: Array.isArray(importance) ? importance[0] : importance,
      })
    ).sort((a, b) => b.importance - a.importance);

    // Generate synthetic validation curve (since API doesn't provide it)
    const validationCurve = Array.from({ length: 50 }, (_, i) => ({
      iteration: i + 1,
      trainLoss: Math.exp(-i * 0.05) * (1 + Math.random() * 0.1),
      valLoss: Math.exp(-i * 0.04) * (1.1 + Math.random() * 0.15),
```

```
    }));

    return {
      mae: testMetrics.mae,
      mse: testMetrics.mse,
      rmse: testMetrics.rmse,
      r2: testMetrics.r2,
      mape: testMetrics.mape || 5.0, // Default if not provided
      trainingTime: response.training_duration,
      featureImportance,
      validationCurve,
    };
  };

  // Main function to train the model
  const trainModel = async (parameters: ModelParameter[]) => {
    let progressInterval: NodeJS.Timeout | undefined;
    try {
      setIsLoading(true);
      setProgress(0);
      setError(null);
      setResults(null);

      // Prepare the training request
      const trainingRequest = prepareTrainingRequest(parameters);

      // Set up progress tracking (simulated since API doesn't provide real-time
progress)
      progressInterval = setInterval(() => {
        setProgress(prev => {
          if (prev >= 90) return prev; // Cap at 90% until we get actual completion
          return prev + Math.random() * 5;
        });
      }, 300);

      // Call the API endpoint
      const response = await apiClient.post<TrainModelResponse>('/train',
trainingRequest);
      clearInterval(progressInterval);
      setProgress(100);

      // Process the response
      const formattedResults = processTrainingResponse(response.data);
      setResults(formattedResults);

      return formattedResults;
    } catch (err: any) {
      if (progressInterval) clearInterval(progressInterval);
      setError(err.message || 'Failed to train model');
      console.error('Training error:', err);
      throw err;
    } finally {
      setIsLoading(false);
    }
  };

  return {
```

```
    trainModel,
    isLoading,
    progress,
    results,
    error
  };
}
```

Data Flow Sequence

The Model Training system follows this sequence during operation:

1. **User Interaction**:
   - User selects features and targets in the UI
   - User configures parameter bounds
   - User clicks "Train Model" button
2. **Training Request**:
   - `handleTrainModel` in ModelTrainingDashboard is triggered
   - `trainModel` from useModelTraining hook is called with parameters
   - Progress simulation begins (simulated 0-90%)
   - API request is prepared with the correct schema
3. **API Communication**:
   - Request is sent to `http://localhost:8000/api/v1/ml/train`
   - FastAPI backend processes the request
   - XGBoost model is trained on the server side
4. **Response Handling**:
   - API response is received with metrics and model information
   - Progress is set to 100%
   - Response is processed into UI-friendly format
   - Results are stored in state
5. **Results Visualization**:
   - ModelTrainingResults component displays metrics
   - Feature importance is visualized in bar charts

- Synthetic validation curves are displayed

API Contract

Training Request Schema


```
{
  "db_config": {
    "host": "em-m-db4.ellatzite-med.com",
    "port": 5432,
    "dbname": "em_pulse_data",
    "user": "s.lyubenov",
    "password": "tP9uB7sH7mK6zA7t"
  },
  "mill_number": 8,
  "start_date": "2025-06-10T06:00:00",
  "end_date": "2025-07-11T22:00:00",
  "features": [
    "Ore",
    "WaterMill",
    "WaterZumpf",
    "PressureHC",
    "DensityHC",
    "MotorAmp"
  ],
  "target_col": "PSI80",
  "test_size": 0.2,
  "params": {
    "n_estimators": 300,
    "learning_rate": 0.05,
    "max_depth": 6,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "early_stopping_rounds": 30,
    "objective": "reg:squarederror"
  }
}
```


Training Response Schema


```
{
  "model_id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "train_metrics": {
    "mae": 1.2,
    "mse": 3.4,
    "rmse": 1.8,
```

```
    "r2": 0.85
  },
  "test_metrics": {
    "mae": 1.5,
    "mse": 4.2,
    "rmse": 2.1,
    "r2": 0.82
  },
  "feature_importance": {
    "Ore": [0.23],
    "WaterMill": [0.18],
    "WaterZumpf": [0.15],
    "PressureHC": [0.12],
    "DensityHC": [0.19],
    "MotorAmp": [0.13]
  },
  "training_duration": 12.5,
  "best_iteration": 178,
  "best_score": 4.2
}
```


Future Enhancements

1. **Settings Persistence**:
    - Add endpoints for saving/loading parameter configurations
    - Create database schema for storing user preferences
    - Add UI controls for managing saved configurations
2. **Real-time Progress Updates**:
    - Implement server-sent events or WebSockets for real progress tracking
    - Replace simulated progress with actual training progress
3. **Training History**:
    - Enhance API to return actual validation curves
    - Store historical training runs for comparison
4. **Enhanced Security**:
    - Move database credentials to backend environment variables
    - Implement proper authentication for API requests

Error Handling

The system includes multiple layers of error handling:

1. **API Call Errors**:
    - Try/catch blocks in hook functions
    - Error state management in useModelTraining
    - Visual error display in the UI
2. **Validation Errors**:
    - Input validation before API calls
    - Feature/target selection validation
3. **UI Feedback**:
    - Loading indicators during API calls
    - Progress bar for long-running operations
    - Error messages for failed requests

Conclusion

The Mills-AI Model Training system provides a comprehensive interface for training XGBoost models against mill process data. The modular design separates concerns between UI components, data management, and API communication, making the system extensible and maintainable.

The integration with the FastAPI backend demonstrates a clean separation between frontend and backend concerns while providing a responsive user experience with simulated progress tracking and rich visualization of training results.