PiSync - Admin Panel

A clean, minimalist Flutter web application for managing and synchronizing Raspberry Pi devices.

Project Overview

PiSync is a responsive Flutter application designed with a clean, monochromatic aesthetic. The application features a comprehensive device management system with authentication, real-time sync status, and error logging.

Design Principles

- **Clean, Minimalist UI**: Black and white color scheme with strategic color accents
- **Responsive Layout**: Adapts to different screen sizes using Wrap widgets
- **Performance Optimized**: Following Flutter web best practices for loading speed

Project Structure

```
lib/
  ├── app/            # App configuration and initialization
  │   ├── bootstrap.dart    # App startup configuration
  │   ├── app.dart          # Main app widget
  │   ├── theme.dart        # App theme configuration
  │   └── routes/           # Navigation and routing
  │
  ├── logic/          # Business logic layer
  │   ├── apis/           # API clients for backend communication
  │   ├── models/         # Data models
  │   ├── repositories/   # Repository implementations
  │   └── utils/          # Utility classes and mixins
  │
  ├── ui/             # User interface components
  |   └── screens/        # Application screens
  |       ├── auth/       # Authentication screens
  |       ├── dashboard/  # Dashboard views
  |       └── errors_log_screen.dart
  |
  └──main.dart  # root of project
```

Architecture Overview

Repository Pattern

The application uses a repository pattern to abstract data sources and handle caching:

1. **Repositories** (`AuthRepository`, `DevicesRepository`)

   - Extend `CachedState` for persistent state management
   - Mix in `ErrorHandlingAndRetryMixin` for error handling and retry logic
   - Provide a clean API for the UI layer

2. **Implementation Flow**:

   ```
   UI Layer (BLoC) -> Repository -> API Client -> Backend Server
   ```

3. **State Management**:

    - Repositories use HydratedBloc for persisting state
    - Cache expiration and refresh mechanisms
    - Retry logic for handling network failures

Example from `DevicesRepositoryImpl`:

```
@override
FutureOr<List<Device>> getAllDevices() async {
  // Check auth status
  var jwtToken = AuthRepository.currentUser?.jwtToken;
  if (jwtToken == null) {
    throw Exception('User is not logged in');
  }

  return handleErrorsAndRetry(() async {
    // Call API and process response
    final response = await devicesApi.getAllDevices(jwtToken);
    List<Device> devices = [];
    // Transform response to model objects
    // ...
    // Update local cache
    emit(devices);
    return devices;
  });
}
```

BLoC Pattern Implementation

The application uses the BLoC (Business Logic Component) pattern for state management:

1. **Events**: Simple, immutable objects that represent user actions or system events

    - Example: `RefreshDeviceEvent`, `LoadPageEvent`, `SyncSingleDeviceEvent`

2. **State**: Immutable objects representing UI state

    - Example: `DeviceState` with status, devices list, pagination info

3. **BLoC**: Handles events, updates state, and communicates with repositories

    - Example: `DeviceBloc` processes sync requests and refreshes

Example BLoC event handler:

```
Future<void> _handleSyncDevice(
  SyncSingleDeviceEvent event,
  Emitter<DeviceState> emit,
) async {
  // Update device status to syncing
  emit(updatedState);

  try {
    // Call repository
    var device = await devicesRepos.syncDeviceById(event.deviceId);
    // Update state with success
    emit(successState);
  } catch (e) {
    // Handle error and update state
    emit(errorState);
  }
}
```

API Integration

The API layer handles communication with the backend:

1. **API Clients** (`AuthApi`, `DevicesApi`)
   - Encapsulate HTTP requests and response parsing
   - Handle authentication headers
   - Convert between JSON and model objects
2. **Endpoints**:
   - Authentication: `/pisync/auth/login`, `/pisync/auth/register`
   - Devices: `/pisync/devices`, `/pisync/devices/:id`
3. **Pagination and Filtering**:

   ```
   Future<Map<String, dynamic>> getAllDevices(
     String token, {
     int page = 1,
     int limit = 10,
     String? syncStatusCode,
     String? sortBy,
     String? order,
   }) async {
     // Build query parameters
     final queryParams = <String, String>{};
     // Add pagination and filters
     // Make HTTP request
     // Parse response
   }
   ```

UI Layer

The UI is built with responsive design in mind:

1. **Adaptive Layout**:
   - Uses Wrap widgets for responsive content
   - Adapts to different screen sizes with conditional layouts
   - Consistent spacing and alignment
2. **State Consumption**:
   - Uses BlocBuilder for reactive UI updates
   - Uses BlocListener for side effects (like showing error messages)

Example adaptive layout from `DashboardScreen`:

```
width > 800
  ? Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: children,
    )
  : Column(children: children)
```

Getting Started

1. Clone the repository
2. Install dependencies:

   ```
   flutter pub get
   ```

3. Run the application:

   ```
   flutter run -d chrome
   ```

Best Practices Followed

- **Clean Architecture**: Separation of concerns with logic/UI layers
- **State Management**: BLoC pattern for predictable state flow
- **Error Handling**: Centralized error processing with retry mechanisms
- **Responsive Design**: Adaptive layouts using Wrap and conditional widgets
- **Performance**: Following Flutter web performance best practices

Pi Device Sync Management API

Node.js backend server built with TypeScript and Express to manage device synchronization.

Overview

This API provides endpoints

- authenticating pisync admin users
- managing devices
- tracking synchronization statuses and error logs.
- It includes hardcoded mock data.

Features

- User authentication with JWT
- Device management
- Device synchronization
- Error logging and tracking
- CORS-enabled for cross-origin requests

Prerequisites

- Node.js (v14 or higher)
- npm or yarn

Getting Started

Installation

1. Clone the repository

2. Install dependencies
3. Ready to Run

```
npm install

npm run dev # Development Mode

npm run build #### Production Mode

npm start
```

The server will run at http://localhost:3000 by default.

API Endpoints

Authentication

- **POST /pisync/auth/register**: Register a new user

- **POST /pisync/auth/login**: Login with existing credentials

  ```
  # Same for both register & login
  {
    "username": "your_username",
    "password": "your_password"
  }
  ```

Devices (Requires Authentication)

- **GET /pisync/devices**: Get all devices
- **GET /pisync/devices/:id**: Trigger synchronization for a specific device

Authentication

All device endpoints require a valid JWT token. Include the token in the Authorization header:

```
Authorization: Bearer YOUR_JWT_TOKEN
```

😄 Why are you waiting for just run it...

Pi Device Sync Management API – Final Docs

This API helps you manage and sync your devices effortlessly. Below is the full breakdown – endpoints, request-response format, error handling, and all that jazz you already know.

Base URL

All endpoints start with:

```
/pisync
```

□ Authentication

We use **JWT tokens** for secure access. For protected routes:

```
Authorization: Bearer <your_jwt_token>
```

Token validity = **1 hour**.

□□ Auth Endpoints

POST `/auth/register`

Registers a new user.

Body:

```
{
  "username": "john_doe",
  "password": "securePassword123"
}
```

Success:

```
{
  "message": "User registered successfully.",
  "token": "<jwt_token_here>"
}
```

Errors:

- 400: `{ "message": "Username already exists" }`
- 500: `{ "message": "Server error during registration" }`

POST `/auth/login`

Logs in a user.

Body:

```
{
  "username": "john_doe",
  "password": "securePassword123"
}
```

Success:

```
{
  "message": "Login successful.",
  "token": "<jwt_token_here>"
}
```

Errors:

- 401: `{ "message": "Invalid username or password." }`
- 500: `{ "message": "Server error during login" }`

□ Device Endpoints

> All device-related endpoints **require auth**.

GET `/devices`

Retrieves the device list with filtering, pagination, and sorting.

Query Params:

Param	Type	Description	Default
`sync_status_code`	Number or String	Filter by sync status. Use `200` for success or `!200` for failures.	All devices
`page`	Number	Page number for pagination.	1
`limit`	Number	Number of items per page.	10
`sort_by`	String	Sort field. Options: `last_sync_at`, `last_attempt_at`.	`last_sync_at`
`order`	String	`asc` for ascending, `desc` for descending.	`desc`

Example Requests:

```
GET /devices
GET /devices?sync_status_code=200
GET /devices?sync_status_code=!200&page=2&limit=5&sort_by=last_attempt_at&order=asc
```

Success Response:

```
{
  "total": 5,
  "page": 1,
  "limit": 10,
  "total_pages": 1,
  "devices": [
    {
      "device_id": "PBX00121",
      "name": "Raspberry Pi 4B",
      "type": "Pi4",
      "last_sync_at": "2025-04-23T14:30:25.000Z",
      "sync_status_code": 200,
      "error_message": null,
      "last_attempt_at": "2025-04-23T14:30:25.000Z"
    }
  ]
}
```

Errors:

- 401: Missing or invalid token
- 500: `{ "message": "Server error while fetching devices" }`

GET `/devices/:id`

Triggers a sync for the device with the given ID.

URL Param: `id` (number or string depending on your system)

Success:

```
{
  "message": "Device synced successfully",
  "device": {
    "device_id": 101,
    "name": "Raspberry Pi 4B",
    "type": "Pi4",
    "last_sync_at": "2025-04-28T10:30:25.000Z",
    "sync_status_code": 200,
    "error_message": null,
    "last_attempt_at": "2025-04-28T10:30:25.000Z"
  }
}
```

Errors:

- 400: `Sync failed: Connection Timeout`
- 404: `Device with ID 123 not found` or `Sync failed: Server Not Reachable`
- 500: `Sync failed: Unknown Sync Error`
- 401: Missing or invalid token

⚙ Sync Status Codes

Code	Meaning	Possible Errors
200	Success	–
400	Bad Request	"Connection Timeout", "Authentication Failure"
404	Not Found	"Server Not Reachable", "Device not found"
500	Server Error	"Unknown Sync Error", "Server Not Reachable"

□ Data Models

`User`

```
interface User {
  id: number;
  username: string;
  password: string; // Hashed only, never returned
}
```

`Device`

```
interface Device {
  device_id: number | string;
  name: string;
  type: string;
  last_sync_at: string | null;
  sync_status_code: number | null;
  error_message: string | null;
  last_attempt_at: string | null;
}
```