Offline Kiosk Documentation

Complete documentation for the Offline Kiosk system - A React-based offline-first kiosk for laundry machines

? Overview

Project overview, architecture, and quick start guide

Project Overview

Offline Kiosk - Complete Documentation

> Comprehensive documentation for the Offline Kiosk system

Generated: 2026-05-06T07:42:12.006Z

Project Overview

A React-based offline-first kiosk system for laundry machines with Node.js backend, MQTT integration, payment processing, and WiFi management.

Architecture

System Components

1. **Frontend (React)** - User interface kiosk application
2. **Backend (Node.js)** - API server and business logic
3. **Database (PostgreSQL)** - Data persistence
4. **MQTT Broker (EMQX)** - Real-time messaging
5. **Deployment Service** - Automated deployment management
6. **WiFi Manager** - Network connectivity management

Documentation Modules

- Node.js Backend Server
- Automated Deployment Service
- WiFi Management Scripts
- Infrastructure and Docker Configuration

Quick Start

Prerequisites

- Docker and Docker Compose
- Node.js 20+
- Git

Installation

```
# Clone the repository
git clone <repository-url>
cd offline-kiosk

# Start all services
docker-compose up -d
```

Environment Variables

Backend Server

- `NODE_ENV` - Environment (production/development)
- `DB_HOST` - PostgreSQL host
- `DB_PORT` - PostgreSQL port
- `DB_NAME` - Database name
- `DB_USERNAME` - Database username
- `DB_PASSWORD` - Database password
- `MQTT_BROKER_URL` - MQTT broker URL
- `MQTT_USERNAME` - MQTT username
- `MQTT_PASSWORD` - MQTT password

Contributing

This documentation is automatically generated from source code comments. To update the documentation:

1. Update comments in the source code
2. Push changes to the repository
3. CI/CD pipeline will automatically update BookStack

? Frontend Documentation

React application, components, hooks, and services

Frontend Architecture

React Frontend Application

> Auto-generated documentation for the client module

Last Updated: 2026-05-06T07:42:11.908Z

Table of Contents

Overview

React Frontend Application

File Structure

```
client/src/utils/machineUtils.js
client/src/services/qrPaymentService.js
client/src/services/pricingService.js
client/src/services/machineService.js
client/src/pages/index.js
client/src/hooks/useWifiManager.js
```

```
client/src/hooks/useTheme.js
client/src/hooks/useReconfigureSocket.js
client/src/hooks/usePaymentTracking.js
client/src/hooks/useNetworkStatus.js
client/src/hooks/useMaintenanceSocket.js
client/src/hooks/useMachineSocket.js
client/src/hooks/useKioskSecurity.js
client/src/hooks/useIpAddress.js
client/src/hooks/useIdleTimeout.js
client/src/hooks/useCoinAcceptorStatus.js
client/src/context/createThemeContext.js
client/src/config/api.js
client/src/components/index.js
client/src/main.jsx
client/src/App.jsx
client/src/pages/WelcomePage.jsx
client/src/pages/PaymentTimeoutPage.jsx
client/src/pages/PaymentSuccessPage.jsx
client/src/pages/MaintenancePage.jsx
client/src/pages/LaundryMachinesPricingPage.jsx
client/src/pages/LaundryMachinesPage.jsx
client/src/pages/ConfigurePage.jsx
client/src/context/ThemeContext.jsx
client/src/components/Header.jsx
client/src/components/Footer.jsx
client/src/components/ui/PricingOptions.jsx
client/src/components/ui/PaymentSummary.jsx
client/src/components/ui/NumericKeyboard.jsx
client/src/components/ui/MachineCard.jsx
client/src/components/ui/LoadingSpinner.jsx
client/src/components/ui/LoadingPage.jsx
client/src/components/ui/InternalKeyboard.jsx
client/src/components/ui/ErrorPage.jsx
client/src/components/ui/DuitNowBorder.jsx
```

Components/Modules

client/src

Component: main

Component: App

Hooks Used: useEffect, useNavigate, useLocation, useKioskSecurity, useIdleTimeout, useMaintenanceSocket, useReconfigureSocket, useTheme

client/src/components

client/src/components/Header.jsx

Header Component Global header for the kiosk application. Displays branding, network status, and coin acceptor status. Features: - Branded logo and title - Real-time network status indicator - Real-time coin acceptor status indicator - Optional configure button (maintenance mode) - Theme support (laundro/cuci) - Responsive design

Parameters:

- `props` (`Object`): - Component props
- `props.showConfigureButton` (`boolean`): - Show configure button for maintenance
- `props.theme` (`string`): - Theme override (laundro/cuci/green)

Example:

```
// Standard usage: <Header />
```

```
// With configure button and custom theme: <Header showConfigureButton={true} theme="green" />
```

Component: Header

Hooks Used: useNavigate, useNetworkStatus, useCoinAcceptorStatus, useTheme

client/src/components/Footer.jsx

Footer Component Global footer for the kiosk application. Displays copyright, branding, and support contact information. Features: - Dynamic copyright year - Support phone number from configuration - Theme support (laundro/cuci) - Responsive design

Parameters:

- `props` (`Object`): - Component props
- `props.theme` (`string`): - Theme override (laundro/cuci/green)

Example:

```
// Standard usage: <Footer />
```

```
// With theme override: <Footer theme="green" />
```

Fetch support number from configuration

Component: Footer

Hooks Used: useState, useEffect, useTheme

client/src/components/ui

Component: PricingOptions

Hooks Used: useTheme

Component: PaymentSummary

Hooks Used: useState, useEffect, useRef, useTheme

Component: NumericKeyboard

Component: MachineCard

Hooks Used: useTheme

client/src/components/ui/LoadingSpinner.jsx

LoadingSpinner Component A reusable loading spinner with customizable message

Parameters:

- `props` (`Object`): - Component props
- `props.message` (`string`): - Loading message to display
- `props.size` (`string`): - Size classes for the spinner
- `props.textSize` (`string`): - Text size class for the message
- `props.showMessage` (`boolean`): - Whether to show the loading message

Component: LoadingSpinner

Hooks Used: useTheme

Component: LoadingPage

Component: InternalKeyboard

Hooks Used: useState

Component: ErrorPage

Hooks Used: useNavigate, useTheme

Component: DuitNowBorder

client/src/config

client/src/config/api.js

API Configuration Centralized API endpoint configuration for the Offline Kiosk frontend. Handles environment-specific base URLs and provides typed endpoint constants.

Example:

```
// Using in a component: import API_ENDPOINTS from '../config/api'; const fetchMachines = async () => { const response = await fetch(API_ENDPOINTS.MACHINES); const data = await response.json(); return data; };
```

Base URL for all API requests Can be overridden via VITE_API_BASE_URL environment variable

API Endpoint Constants Provides a centralized location for all API endpoints used in the application. All endpoints are relative to the API_BASE_URL.

client/src/context

Component: ThemeContext

Hooks Used: useState, useEffect

client/src/hooks

client/src/hooks/useWifiManager.js

useWifiManager Hook Manages WiFi connectivity operations for the kiosk. Interfaces with WiFi manager API endpoints to scan, connect, and manage networks. Features: - Fetch available WiFi networks - Get currently connected network - Connect to WiFi with credentials - Forget/disconnect from networks - Loading and error states

Returns:

- `Object`: management state and functions
- `Array`: - Array of available WiFi networks
- `string|null`: - Currently connected network SSID
- `boolean`: - Loading state for operations
- `string|null`: - Error message if operation failed
- `Function`: - Fetch available networks
- `Function`: - Connect to a network
- `Function`: - Forget a saved network

Example:

```
const { wifiList, connectedSsid, loading, connectWifi, fetchWifiList } = useWifiManager();
useEffect(() => { fetchWifiList(); }, []); const handleConnect = async (ssid, password) => {
const result = await connectWifi(ssid, password); if (result.success) {
console.log('Connected!'); } };
```

Fetch available WiFi networks and current connection status

Returns:

- `Promise<void>`:

Connect to a WiFi network

Parameters:

- `ssid` (`string`): - Network SSID
- `password` (`string`): - Network password

Returns:

- `Promise<Object>`: object with success boolean

Forget (remove) a saved WiFi network

Parameters:

- `ssid` (`string`): - Network SSID to forget

Returns:

- `Promise<Object>`: object with success boolean

client/src/hooks/usePaymentTracking.js

usePaymentTracking Hook Tracks real-time payment progress via WebSocket. Monitors coin and e-payment amounts, calculates remaining balance and change. Features: - Real-time payment updates via Socket.IO - Automatic payment completion detection - Change amount calculation - Connection status tracking

Parameters:

- `targetAmount` (`number`): - Target payment amount

Returns:

- `Object`: tracking state
- `Object`: - Payment details object
- `number`: - Current coin payment amount
- `number`: - Current e-payment amount
- `number`: - Total amount paid (coin + epay)
- `number`: - Remaining amount to complete payment
- `boolean`: - Whether payment is complete
- `number`: - Change amount (overpayment)
- `boolean`: - WebSocket connection status
- `Object`: - Socket.IO client instance

Example:

```
const { paymentData, isConnected } = usePaymentTracking(5.00); useEffect(() => { if (paymentData.isPaymentComplete) { console.log('Payment complete! Change:', paymentData.changeAmount); } }, [paymentData]);
```

client/src/hooks/useNetworkStatus.js

useNetworkStatus Hook Monitors internet connectivity by pinging Google's generate_204 endpoint. Sends heartbeat status updates to backend server for kiosk monitoring. Features: - Periodic internet connectivity checks (every 5 seconds) - IP address detection - Heartbeat status reporting to backend (every minute) - Automatic retry on failure

Returns:

- `Object`: status
- `boolean`: - Whether kiosk has internet connectivity

Example:

```
const { isOnline } = useNetworkStatus(); return ( <div> Status: {isOnline ? 'Connected' : 'Offline'} </div> );
```

client/src/hooks/useMachineSocket.js

useMachineSocket Hook Establishes WebSocket connection for real-time machine status updates. Subscribes to 'machines-update' events from the server. Features: - Auto-connects to Socket.IO server on mount - Receives real-time machine status updates - Tracks connection status - Auto-cleanup on unmount

Returns:

- `Object`: state and data
- `Array`: - Array of machine objects with current status
- `boolean`: - WebSocket connection status
- `Object`: - Socket.IO client instance

Example:

```
const { machines, isConnected } = useMachineSocket(); useEffect(() => { if (machines.length > 0) { console.log('Machines updated:', machines); } }, [machines]);
```

client/src/hooks/useKioskSecurity.js

Custom hook for implementing kiosk security measures Prevents common user interactions that could compromise kiosk mode

client/src/hooks/useIpAddress.js

useIpAddress Hook Fetches and tracks the kiosk's local IP address. IP is detected by the ip-detector service running with host network. Features: - Fetch IP address on mount - Loading and error states - Retry capability via refetchIpAddress

Returns:

- `Object`: address state
- `string|null`: - Local IP address (e.g., '192.168.1.100')
- `boolean`: - Loading state
- `string|null`: - Error message if fetch failed
- `Function`: - Manual IP setter
- `Function`: - Refetch IP address

Example:

```
const { ipAddress, loading, error } = useIpAddress(); if (loading) return <Spinner />; if (error) return <div>Error: {error}</div>; return <div>Kiosk IP: {ipAddress}</div>;
```

client/src/hooks/useIdleTimeout.js

Custom hook for handling idle timeout in kiosk applications Automatically triggers a callback after a specified period of user inactivity

Parameters:

- `onIdle` (`Function`): - Callback function to execute when user becomes idle
- `timeout` (`number`): - Timeout duration in milliseconds (default: 60000ms = 1 minute)
- `enabled` (`boolean`): - Whether the idle timeout is enabled (default: true)
- `events` (`Array`): - Array of event types to listen for (default: common user interaction events)

Returns:

- `Object`: Object containing reset function and idle state

client/src/hooks/useCoinAcceptorStatus.js

useCoinAcceptorStatus Hook Monitors coin acceptor status in real-time via WebSocket. Used to determine if kiosk can accept coin payments. Features: - Real-time coin acceptor status updates - Connection status tracking - Auto-reconnect on disconnect

Returns:

- `Object`: acceptor state
- `boolean`: - Whether coin acceptor is available
- `boolean`: - WebSocket connection status

Example:

```
const { coinAcceptorStatus, isConnected } = useCoinAcceptorStatus(); if (!coinAcceptorStatus) { console.warn('Coin acceptor not available'); }
```

client/src/pages

client/src/pages/WelcomePage.jsx

WelcomePage Component The landing page of the kiosk application. Displays a welcome screen with: - Branding and welcome message - Configuration status check - Coin acceptor status validation - Navigation to machine selection Features: - Automatic redirect to configuration if not configured - Disabled state when coin acceptor is offline - Themed design with gradient backgrounds - Responsive layout for different screen sizes

Example:

```
// Used in App.jsx routing: <Route path="/" element={<WelcomePage />} />
```

Check configuration status on component mount Redirects to configuration page if kiosk is not configured Maintenance mode is handled globally in App.jsx

Handle start button click Only navigates to machines page if coin acceptor is online

Component: WelcomePage

Hooks Used: useEffect, useNavigate, useTheme, useCoinAcceptorStatus

Component: PaymentTimeoutPage

Hooks Used: useEffect, useState, useNavigate, useLocation

client/src/pages/PaymentSuccessPage.jsx

PaymentSuccessPage Component Displays payment success confirmation after a successful transaction. Shows transaction details, machine information, and next steps. Features: - Success animation - Transaction summary - Change/balance display (if applicable) - Phone number points notification (if provided) - Auto-redirect to home after 10 seconds - Manual navigation options

Example:

```
// Rendered via React Router with transaction state: navigate('/payment-success', { state: { machine, pricing, transaction, machineResponse, duration, currentPrice, isDryer, change, phoneNumber } })
```

Auto-redirect to home after 10 seconds

Navigate to home page

Navigate to machines page

Redirect if required state is missing

Component: PaymentSuccessPage

Hooks Used: useEffect, useNavigate, useLocation

Component: MaintenancePage

Hooks Used: useEffect, useNavigate, useMaintenanceSocket, useTheme

client/src/pages/LaundryMachinesPricingPage.jsx

LaundryMachinesPricingPage Component Handles pricing selection and payment for a chosen machine. Supports both washer (fixed pricing) and dryer (time-based pricing) workflows. Features: - Dynamic pricing calculation based on machine type - Dryer duration selection (initial time + incremental additions) - Coin acceptor integration for payments - QR payment support (if enabled) - Real-time payment tracking via WebSocket - Network status monitoring - Change tracking (coin balance)

Example:

```
// Rendered via React Router with machine state: <Route path="/pricing"
element={<LaundryMachinesPricingPage />} /> // Navigate with machine data:
navigate('/pricing', { state: { machine } })
```

Load pricing data for the selected machine Fetches pricing configurations from API based on machine type and weight. Initializes default duration for dryers.

Handle dryer duration change Updates both selected and custom duration states.

Parameters:

- `d` (`number`): - New duration in minutes

Increment dryer duration Adds runtime increment based on step value (from outlet settings). Respects maximum price/duration limits.

Component: LaundryMachinesPricingPage

Hooks Used: useState, useEffect, useCallback, useNavigate, useLocation, usePaymentTracking, useNetworkStatus, useTheme

client/src/pages/LaundryMachinesPage.jsx

LaundryMachinesPage Component Displays all available laundry machines (washers and dryers) in the outlet. Shows machine status, availability, and allows users to select a machine for payment. Features: - Real-time machine status updates via WebSocket - Visual indicators for machine state (online/offline, running/idle, maintenance) - Machine selection for payment flow - WiFi signal strength display - Automatic status refresh

Example:

```
// Rendered via React Router: <Route path="/machines" element={<LaundryMachinesPage />} />
```

Get status badge CSS class based on machine state

Parameters:

- `machine` (`Object`): - Machine object

Returns:

- `string`: CSS classes for status badge

Load machines from API Fetches initial machine data from the server. Called on component mount.

Update machines when socket receives new data Real-time synchronization of machine status via WebSocket.

Handle machine selection Navigates to pricing page with selected machine data.

Parameters:

- `machine` (`Object`): - Selected machine object

Navigate back to welcome page

Component: LaundryMachinesPage

Hooks Used: useState, useEffect, useNavigate, useMachineSocket, useTheme

Component: ConfigurePage

Hooks Used: useState, useEffect, useRef, useNavigate, useIpAddress, useWifiManager, useNetworkStatus

client/src/services

client/src/services/machineService.js

Machine Service Service layer for interacting with laundry machine-related API endpoints. Handles fetching machine data, status updates, and error handling.

Fetch all available laundry machines from the backend Retrieves a list of all laundry machines configured in the outlet, including their status, type, and availability.

Returns:

- `Promise<Object>`: data object
- `Promise<Object>`: - Operation success status

- `Promise<Array>`: - Array of machine objects
- `Promise<string>`: - Machine ID
- `Promise<string>`: - Machine display name
- `Promise<string>`: - Machine type (washer/dryer)
- `Promise<boolean>`: - Machine active status
- `Promise<boolean>`: - Machine availability

Example:

```
// Fetch machines in a component: import { fetchMachines } from '../services/machineService'; const loadMachines = async () => { try { const data = await fetchMachines(); if (data.success) { console.log('Machines:', data.machines); } } catch (error) { console.error('Failed to load machines:', error.message); } };
```

Machine Service Object Exported service object containing all machine-related API functions. Can be imported as a default export for easier mocking in tests.

client/src/utils

?? Backend Documentation

Node.js server, API endpoints, and database models

Backend Architecture

Node.js Backend Server

> Auto-generated documentation for the server module

Last Updated: 2026-05-06T07:42:11.952Z

Table of Contents

Overview

Node.js Backend Server

File Structure

```
server/server.js
server/utils/winston.js
server/utils/wifiSignal.js
server/utils/logCollection.js
server/utils/dockerRestart.js
server/utils/deployment.js
```

```
server/utils/clearDatabase.js
server/services/transactionSyncScheduler.js
server/services/syncScheduler.js
server/services/sendKioskTransaction.js
server/routes/index.js
server/models/paymentCredentials.js
server/models/Value.js
server/models/Transaction.js
server/models/Pricing.js
server/models/Outlet.js
server/models/Device.js
server/models/Config.js
server/middlewares/auth.js
server/configs/socket.js
server/configs/sequelize.js
server/configs/mqtt.js
server/configs/database.js
server/routes/api/wifimanager.routes.js
server/routes/api/value.routes.js
server/routes/api/sync.routes.js
server/routes/api/qrPayment.routes.js
server/routes/api/pricing.routes.js
server/routes/api/payment.routes.js
server/routes/api/external.routes.js
server/routes/api/device.routes.js
server/routes/api/configure.routes.js
server/routes/api/auth.routes.js
server/controller/api/wifimanager.controller.js
server/controller/api/value.controller.js
server/controller/api/sync.controller.js
server/controller/api/qrPayment.controller.js
server/controller/api/pricing.controller.js
server/controller/api/payment.controller.js
server/controller/api/external.controller.js
server/controller/api/device.controller.js
server/controller/api/configure.controller.js
server/controller/api/auth.controller.js
```

Components/Modules

server

server/server.js

Offline Kiosk Backend Server Main server application for the Offline Kiosk system. Handles: - RESTful API endpoints for kiosk operations - Real-time WebSocket communication with clients - MQTT integration for machine communication - Payment processing and transaction management - Device status monitoring and updates - WiFi and network management - Automatic synchronization with external services

Socket.IO server instance for real-time communication Configured with CORS to allow connections from any origin

File paths for shared data between containers These files are mounted as volumes from host network detection services

Middleware Configuration

API Routes All API endpoints are prefixed with /api

WebSocket Connection Handler Handles new client connections and sends initial state data: - Current IP address from host network - WiFi connection status - Coin acceptor device status

Parameters:

- `socket` (`Socket`): - Socket.IO socket instance

server/configs

server/controller/api

server/controller/api/wifimanager.controller.js

WiFi Manager Controller Manages WiFi connectivity for the kiosk system. Interfaces with the wifi-manager Docker container to: - Scan for available networks - Connect to WiFi networks - Disconnect from networks - Monitor connection status

Path to WiFi list file (shared volume with wifi-manager container)

Path to current WiFi connection file

Get list of available WiFi networks Retrieves scanned WiFi networks from the wifi-manager service. Networks are sorted by signal strength (strongest first).

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Array`: of WiFi network objects
- `Object`: - Network name
- `number`: - Signal strength (0-100)
- `string`: - Security type (WPA2, Open, etc.)

Example:

```
// Response: [ { "ssid": "MyNetwork", "signal": 85, "security": "WPA2" }, { "ssid": "GuestWiFi", "signal": 60, "security": "Open" } ]
```

Get currently connected WiFi network Returns the SSID of the currently connected WiFi network. Returns null if not connected.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Object`: WiFi status
- `string|null`: - Connected network name or null

Example:

```
// Connected: { "ssid": "MyNetwork" } // Not connected: { "ssid": null }
```

Connect to a WiFi network Attempts to connect to the specified WiFi network with provided credentials. Executes wifi-connect.sh script in the wifi-manager container.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.ssid` (`string`): - Network SSID to connect to

- `req.body.password` (`string`): - Network password
- `res` (`Object`): - Express response object

Returns:

- `Object`: result

Example:

```
// Request: POST /api/wifi-manager/connect { "ssid": "MyNetwork", "password": "mypassword123" } // Success: { "success": true, "message": "Connected successfully" } // Error: { "success": false, "message": "Incorrect password" }
```

Forget (disconnect from) a WiFi network Removes the specified WiFi network from saved connections.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.ssid` (`string`): - Network SSID to forget
- `res` (`Object`): - Express response object

Returns:

- `Object`: result

Example:

```
// Request: POST /api/wifi-manager/disconnect { "ssid": "MyNetwork" } // Response: { "success": true, "message": "Network forgotten" }
```

server/controller/api/sync.controller.js

Sync Controller Handles synchronization of data between the kiosk and external core systems. Responsible for: - Syncing outlet information from core system - Syncing pricing data - Syncing device configurations - Syncing payment credentials - Ensuring data consistency across systems

Core system base URL for synchronization

Synchronize outlet data from core system Fetches outlet configuration from the core system and updates local database. Handles both new outlets and updates to existing outlets.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.outletId` (`string`): - Unique identifier of the outlet to sync
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: result with statistics

Example:

```
// Request: POST /api/sync/outlet { "outletId": "outlet_123" } // Response: { "success": true, "stats": { "updated": 1, "added": 0, "errors": 0 } }
```

server/controller/api/qrPayment.controller.js

QR Payment Controller Handles QR code generation and payment processing via external payment gateway. Supports e-payment methods like DuitNow QR for cashless transactions. Features: - QR code generation for specific amounts - Payment status tracking - Integration with RHB DuitNow payment gateway - Transaction ID generation and management

External payment API URL

Generate QR code for payment Creates a QR code for the specified amount and machine. Integrates with external payment gateway to generate payment QR codes.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.machineId` (`string`): - Machine identifier
- `req.body.amount` (`number`): - Payment amount in smallest currency unit (cents)
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: code data (base64 image)

Example:

```
// Request: POST /api/qr-payment/create { "machineId": "W001", "amount": 500 } // Response: { "success": true, "qrCode": "...", "transactionId": "tx-1234567890-12345" }
```

server/controller/api/pricing.controller.js

Pricing Controller Manages pricing information for laundry machines. Retrieves pricing configurations for different machine types and cycles.

Get all pricing configurations Retrieves all pricing data including machine types, cycle types, and associated outlet information.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Promise<Array>`: of pricing objects with outlet information

Example:

```
// Response: [ { "id": 1, "machine_type": "washer", "cycle_type": "normal", "price": 500, "duration": 30, "outlet": { "id": "outlet_123", "name": "Downtown Laundry" } } ]
```

server/controller/api/payment.controller.js

Payment Controller Handles all payment-related operations for the kiosk system including: - Processing cash and e-payment transactions - Validating payment amounts - Communicating with laundry machines via MQTT - Recording transactions in the database - Sending transaction data to external systems

Process a payment transaction Validates payment amount, creates transaction record, and triggers machine operation. Supports both coin and e-payment methods.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.machineId` (`string`): - Unique identifier of the laundry machine
- `req.body.machineType` (`string`): - Type of machine (washer/dryer)
- `req.body.pricingId` (`string`): - Pricing plan identifier
- `req.body.amount` (`number`): - Required payment amount in cents/smallest currency unit
- `req.body.outletName` (`string`): - Name of the outlet/location
- `req.body.outletId` (`string`): - Unique identifier of the outlet
- `req.body.number_phone` (`string`): - Optional customer phone number
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: result with status and details

Example:

```
// Request body example: { "machineId": "W001", "machineType": "washer", "pricingId":
"price_123", "amount": 500, "outletName": "Downtown Laundry", "outletId": "outlet_001",
"number_phone": "+1234567890" } // Success response: { "success": true, "transactionId":
"txn_456", "machineId": "W001", "amount": 500, "change": 0 }
```

server/controller/api/device.controller.js

Device Controller Manages device-related operations including: - Retrieving local IP address from host network detector - Fetching all devices (machines) with outlet information - Getting machine-specific data for UI display - Updating kiosk online/offline status - Heartbeat monitoring

Get local IP address from host network detector Reads the IP address detected by the ip-detector service running with host network. This provides the actual host machine's IP address for kiosk identification.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: address object

Example:

```
// Response: { "ipAddress": "192.168.1.100" }
```

Get all devices with outlet information Retrieves complete list of all devices (machines) including associated outlet data.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Promise<Array>`: of device objects with outlet information

Get all machines for UI display Retrieves machines with only essential attributes needed for kiosk UI. Returns machines sorted by machine code for consistent display order.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Promise<Array>`: of machine objects with selected attributes
- `Promise<Object>`: - Machine ID
- `Promise<string>`: - Machine code (W001, D001, etc.)
- `Promise<string>`: - Type (washer/dryer)
- `Promise<boolean>`: - Online status
- `Promise<string>`: - Status (running/idle)
- `Promise<boolean>`: - Locked status
- `Promise<string>`: - Display name

Update kiosk online/offline status Receives heartbeat signals from kiosk to track online status. Can optionally forward status to external monitoring systems.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.ipAddress` (`string`): - Kiosk IP address
- `req.body.status` (`string`): - Status (online/offline)
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: update confirmation

Example:

```
// Request: POST /api/device/update-kiosk-status { "ipAddress": "192.168.1.100", "status": "online" }
```

server/controller/api/configure.controller.js

Configure Controller Manages kiosk configuration including: - Initial kiosk setup and configuration - Retrieving configuration data - Maintenance mode management - MQTT reconfiguration and clustering - Database clearing for re-configuration - WebSocket broadcasting for real-time updates

Socket.IO instance for broadcasting configuration changes

Set Socket.IO instance Allows the main server to inject the Socket.IO instance for real-time communication.

Parameters:

- `socketIO` (`Object`): - Socket.IO server instance

Get current kiosk configuration Retrieves the current kiosk configuration including outlet ID, kiosk ID, maintenance status, and branding information.

Parameters:

- `req` (`Object`): - Express request object
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: data or unconfigured status

Example:

```
// Response when configured: { "success": true, "configured": true, "data": { "outletId": "outlet_123", "kioskId": "kiosk_001", "outletName": "Downtown Laundry", "maintenance": false, "brand": "laundro" } }
```

Configure kiosk Performs initial configuration or reconfiguration of the kiosk. Triggers synchronization of outlet data, devices, and pricing. Optionally handles EMQX clustering for secondary nodes.

Parameters:

- `req` (`Object`): - Express request object
- `req.body` (`Object`): - Request body
- `req.body.outletId` (`string`): - Unique outlet identifier (required)
- `req.body.kioskId` (`string`): - Unique kiosk identifier (required)
- `req.body.outletName` (`string`): - Outlet display name (required)
- `req.body.ipAddress` (`string`): - Kiosk IP address (required)
- `req.body.isSecondary` (`boolean`): - Whether this is a secondary MQTT node
- `req.body.primaryIp` (`string`): - Primary node IP (required if isSecondary is true)
- `req.body.supportNumber` (`string`): - Support contact number
- `req.body.brand` (`string`): - Brand theme (laundro/cuci)
- `res` (`Object`): - Express response object

Returns:

- `Promise<Object>`: result

Example:

```
// Request: POST /api/configure { "outletId": "outlet_123", "kioskId": "kiosk_001",
"outletName": "Downtown Laundry", "ipAddress": "192.168.1.100", "brand": "laundro" }
```

server/middlewares

server/models

server/models/Transaction.js

Transaction Model Records all payment transactions made at the kiosk. Stores both coin and e-payment transaction data for audit and reconciliation.

Example:

```
// Create a transaction record: await Transaction.create({ coin_inserted: 5.00,
epayment_amount: 0, machine_id: 'W001', transactionId: 'tx-1234567890', outletId: 'outlet_123'
});
```

```
// Find transactions for a specific machine: const txns = await Transaction.findAll({ where: {
machine_id: 'W001' }, order: [['timestamp', 'DESC']] });
```

server/models/Pricing.js

Pricing Model Stores pricing configurations for different machine types and capacities. Supports both fixed pricing and dynamic pricing models.

Example:

```
// Create washer pricing (fixed): await Pricing.create({ weight: 10, machine_type: 'washer',
amount: 5.00, outletId: 'outlet_123', mode: 'weight-based' });
```

```
// Create dryer pricing (time-based): await Pricing.create({ weight: 10, machine_type:
'dryer', minprice: 1.00, maxprice: 10.00, initialtime: 15, runtime: 5, outletId: 'outlet_123',
mode: 'time-based' });
```

server/models/Device.js

Device Model Represents a laundry machine (washer or dryer) in the kiosk system. Stores machine configuration, status, and operational data.

Example:

```
// Create a new device: await Device.create({ machine_code: 'W001', machine_type: 'washer', active: true, outletId: 'outlet_123' });
```

```
// Find all active machines: const machines = await Device.findAll({ where: { active: true } });
```

server/models/Config.js

Config Model Stores kiosk system configuration including: - Outlet and kiosk identifiers - Network settings - MQTT clustering configuration - Maintenance mode status - Branding/theme settings

Example:

```
// Create initial configuration: await Config.create({ outletId: 'outlet_123', kioskId: 'kiosk_001', outletName: 'Downtown Laundry', ipAddress: '192.168.1.100', brand: 'laundro' });
```

```
// Check if configured: const config = await Config.findOne(); if (config) { console.log('Kiosk is configured'); }
```

server/routes

server/routes/api

API Endpoints:

- `GET` /list
- `GET` /current
- `POST` /connect
- `POST` /forget

API Endpoints:

- `GET` /current
- `PUT` /update
- `POST` /reset

API Endpoints:

- `POST` /outlet
- `POST` /device
- `POST` /pricing
- `POST` /credentials
- `POST` /all

API Endpoints:

- `POST` /generate

API Endpoints:

- `POST` /getPricing

API Endpoints:

- `POST` /process
- `GET` /status/:transactionId
- `POST` /payment
- `POST` /acknowledge

API Endpoints:

- `POST` /getAllMachines
- `GET` /local-ip
- `POST` /update-kiosk-status

API Endpoints:

- `GET` /
- `POST` /
- `POST` /getOutletId
- `POST` /machinevalid
- `POST` /reset

API Endpoints:

- `POST` /login

server/services

server/services/transactionSyncScheduler.js

Transaction Sync Scheduler Service Manages periodic synchronization of transaction data to core system (Laundro). Uploads pending transactions at regular intervals and removes successfully

synced ones. Features: - Scheduled sync every 30 minutes - Manual sync trigger - Batch transaction upload - Automatic transaction cleanup after successful sync - Error handling and retry logic - Detailed logging

Sync all unsent transactions to Laundro core system Fetches all pending transactions from local database and uploads them to core system. Deletes successfully synced transactions from local database.

Returns:

- `Promise<Object>`: summary with statistics
- `boolean`: - Overall operation success
- `number`: - Total transactions processed
- `number`: - Successfully synced count
- `number`: - Failed sync count
- `Array`: - Detailed results per transaction

Example:

```
const result = await syncUnsentTransactions(); console.log(`Synced ${result.successCount} of ${result.totalTransactions} transactions`);
```

Schedule transaction sync to run every 2 hours

Run transaction sync on server startup

server/services/syncScheduler.js

Sync Scheduler Service Manages scheduled synchronization of data from core system to kiosk. Runs periodic sync operations via node-cron to keep local data up-to-date. Features: - Scheduled sync every 2 hours - Startup sync on server initialization - Manual sync trigger capability - Syncs outlets, devices, pricing, payment credentials - Error handling and logging

Run all synchronization operations Executes complete data sync from core system including: - Outlet configuration - Device list - Pricing data - Payment credentials

Returns:

- `Promise<Object>`: sync results

Example:

```
// Manual trigger: await runAllSyncOperations();
```

server/utils

server/utils/wifiSignal.js

Get WiFi signal strength and raw dBm Returns { percentage, dbm } or { percentage: 0, dbm: null } if not available Reads from shared volume written by wifi-detector service

server/utils/logCollection.js

Collect logs from all Docker containers and zip them

Returns:

- `Promise<string>`: to the zip file

Create a zip file from a directory

Parameters:

- `sourceDir` (`string`): - Directory to zip
- `outputPath` (`string`): - Output zip file path

Returns:

- `Promise<void>`:

Send logs to Laundro API

Parameters:

- `zipPath` (`string`): - Path to the zip file

Returns:

- `Promise<Object>`: response

Clean up old log zip files

Parameters:

- `zipPath` (`string`): - Path to the zip file to delete

Main function to collect logs and send to Laundro

Returns:

- `Promise<Object>`: of the operation

server/utils/dockerRestart.js

Call Docker API via Unix socket

Parameters:

- `path` (`string`): - API path
- `method` (`string`): - HTTP method

Restart EMQX container using Docker API

Execute command in container using Docker API

Parameters:

- `containerName` (`string`): - Container name
- `cmd` (`array`): - Command array

Join EMQX cluster by connecting to primary node

Parameters:

- `primaryIp` (`string`): - IP address of the primary EMQX node

Perform the actual cluster join operation

server/utils/deployment.js

Execute shell command with Docker socket access

Parameters:

- `command` (`string`): - Command to execute

Returns:

- `Promise<{stdout: string, stderr: string}>`:

Call Docker API via Unix socket

Parameters:

- `path` (`string`): - API path
- `method` (`string`): - HTTP method

- `body` (`object`): - Request body (optional)

Returns:

- `Promise<object>`:

Get Docker Compose project name from environment or default

Returns:

- `string`:

Pull images for all services in docker-compose

Returns:

- `Promise<object>`:

Build EMQX image using docker exec

Returns:

- `Promise<object>`:

Pulls latest image first (or builds for local images), then starts services one by one Order: postgres -> ip-detector -> emqx -> client -> offline-collector Note: server is excluded because this script runs inside the server container

Returns:

- `Promise<object>`:

Restart all containers in the compose project

Returns:

- `Promise<object>`:

Handle deployment automation via MQTT Executes docker compose pull, build emqx, and up -d

Returns:

- `Promise<object>`: result

API Documentation

Base URL

`http://localhost:5000/api`

Authentication

Some endpoints require authentication. Include the auth token in the request headers.

? Services Documentation

Supporting services and automation

Deployment Service

Automated Deployment Service

> Auto-generated documentation for the deployment module

Last Updated: 2026-05-06T07:42:11.999Z

Table of Contents

Overview

Automated Deployment Service

File Structure

```
deployment-service/logger.js
deployment-service/index.js
deployment-service/deployment.js
```

Components/Modules

deployment-service

deployment-service/index.js

Deployment Service Automated deployment service that listens for deployment triggers via MQTT and handles Docker container updates for the kiosk system. Features: - MQTT-based deployment triggering - Automated Docker image pulling and container restart - Deployment status reporting - Error handling and logging - Reconnection logic for MQTT broker Environment Variables: - MQTT_BROKER_URL: MQTT broker URL (default: mqtt://emqx) - MQTT_USERNAME: MQTT username (default: admin) - MQTT_PASSWORD: MQTT password - MQTT_PORT: MQTT port (default: 1883) - COMPOSE_PROJECT_NAME: Docker Compose project name MQTT Topics: - Subscribe: kiosk/deployment/trigger - Receives deployment requests - Publish: kiosk/deployment/status - Sends deployment status updates

Example:

```
// Trigger deployment via MQTT: mqtt publish -h emqx -t kiosk/deployment/trigger \ -m
'{"service":"kiosk-server","action":"update"}'
```

MQTT Broker Configuration Configures connection parameters for the MQTT broker

MQTT Client Instance Maintains connection to the MQTT broker for deployment triggers

MQTT Connection Event Handler Subscribes to deployment topics when connected

MQTT Message Event Handler Processes incoming deployment requests

Parameters:

- `topic` (`string`): - MQTT topic that received the message
- `message` (`Buffer`): - Message payload

deployment-service/deployment.js

Execute shell command

Parameters:

- `command` (`string`): - Command to execute

Returns:

- `Promise<{stdout: string, stderr: string}>`:

Get Docker Compose project name from environment or default

Returns:

- `string`:

Pulls/builds all images first, then starts all services at once

Returns:

- `Promise<object>`:

Handle deployment automation via MQTT Executes docker compose pull, build emqx, and up -d

Returns:

- `Promise<object>`: result

WiFi Manager

WiFi Management Scripts

> Auto-generated documentation for the wifi module

Last Updated: 2026-05-06T07:42:12.002Z

Table of Contents

Overview

WiFi Management Scripts

File Structure

```
wifi-manager/wifi-forget.sh
wifi-manager/wifi-connect.sh
wifi-manager/manage-wifi.sh
```

Components/Modules

wifi-manager

?? Infrastructure

Docker configuration, deployment, and system architecture

Infrastructure

Infrastructure & Docker

Infrastructure and Docker Configuration

> Auto-generated documentation for the docker module

Last Updated: 2026-05-06T07:42:12.005Z

Table of Contents

Overview

Infrastructure and Docker Configuration

File Structure

```
docker-compose.yml
README.md
```

Components/Modules

.