Technical Design Documentation

Vishan Summinga-Sonagadu

Table of Contents

Introduction

Welcome to the ReM's documentation. This document provides an in-depth guide to understanding, setting up, and using the IR control system built using an ESP32, React for the frontend, Express.js for the backend and and MongoDB as the database management system (DBMS).

Project Overview

Project Description

This project allows users to remotely control IR appliances using an ESP32 microcontroller, with each appliance requiring its own ESP32 to control it. The system includes a web application that enables multiple users to create accounts, add their own appliances, and control them from a central interface. The project supports automation through API integration with services like the Shortcuts app on iPhone and Siri for voice control.

Features

- **Security Features**: The project ensures secure communication and data storage using HTTPS* and JWT tokens for authentication.
- **Scalability**: Designed to handle multiple users and devices efficiently, with plans for future scalability to accommodate more appliances and users.
- **User Experience**: A user-friendly interface built with React, providing easy navigation and intuitive controls for managing appliances.
- **Real-time Updates**: The system uses regular HTTP requests, with each microcontroller polling the server every 250ms** to provide real-time updates and control.
- **Integration**: Potential for integration with other smart home systems and IoT devices to create a seamless smart home experience.
- **Customizability**: Users can customize control settings and add new types of appliances, enhancing the flexibility of the system.
- **API for External Projects**: Users can utilize the provided API to integrate and control their added devices from their own projects, enabling a wide range of custom automation and control scenarios.
- **Maintenance and Support**: Includes detailed documentation for system setup, maintenance, and troubleshooting.

** For HTTPS, a valid SSL/TLS certificate is required to encrypt the data transfer. This involves obtaining a certificate from a trusted Certificate Authority (CA) and configuring the server to use the certificate and private key.*

*** The polling interval is set to 250ms by default, but this value can be modified in the firmware. Increasing the polling frequency will generate more frequent requests, which can increase the load on the network and server. It is important to ensure that the network infrastructure can handle this increased traffic to prevent potential performance degradation.*

Hardware

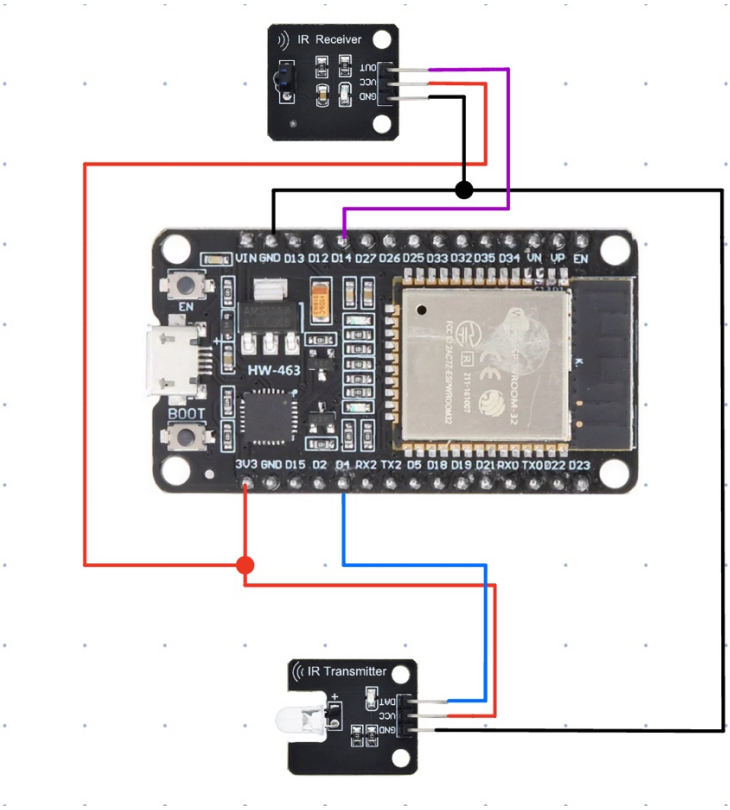
ESP32

The ESP32 microcontroller is at the core of the IR control system. It handles the communication between the IR sensors and the server, sending data about the appliance's status and receiving commands to control the appliance.

IR LED (Transmitter) and Receiver

IR LED (Transmitter) and receiver are used to detect and transmit infrared signals, enabling the control of various IR appliances.

Hardware Connections



The IR configurations in the firmware specify the GPIO pins used for the IR LED and the IR receiver:

- **IR LED (Transmitter)**: Connected to GPIO pin 4.
- **IR Receiver**: Connected to GPIO pin 14.

Software

Firmware

The firmware for the ESP32 is written in Arduino C and is located in the `*ESP32/firmware/firmware.ino*` file. It handles the main logic for sending and receiving IR signals. The following libraries are used:

- *WiFi.h* for managing WiFi connections,
- *HTTPClient.h* for making HTTP requests,
- *ArduinoJson.h* for parsing and creating JSON data,
- *IRremoteESP8266.h*, *IRsend.h*, *IRrecv.h*, and *IRutils.h* for sending and receiving IR signals.

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <IRremoteESP8266.h>
#include <IRsend.h>
#include <IRrecv.h>
#include <IRutils.h>
```

Key Functions

The firmware includes two critical functions that manage the sending and receiving of IR signals: *sendReceivedIRSignal* and *sendReceivedIRCode*.

sendReceivedIRSignal

```
void sendReceivedIRSignal(int protocol, uint32_t value, uint16_t bits);
```

The sendReceivedIRSignal function is responsible for transmitting the IR signal that the user wants to send. This signal can be initiated from the virtual remote on the client-side application or through the API. When a command to send an IR signal is received from the server, this function is called. It takes three parameters: the protocol, the value, and the number of bits of the IR signal. Depending on the protocol specified, the function sends the IR signal using the appropriate method. If the protocol is not recognized, it logs an “Unknown protocol” message.

Here is an overview of its purpose and usage:

- **Purpose**: To transmit IR signals based on user commands received from the server.
- **Usage**: The function is triggered when a user interacts with the virtual remote or sends a command via the API, which instructs the ESP32 to send a specific IR signal.

The function processes the command as follows:

1. The protocol type, signal value, and bit length are extracted from the server's response.
2. The function uses a switch-case structure to handle different protocols like SYMPHONY and NEC.
3. Based on the protocol, the corresponding send function is called to transmit the IR signal.

sendReceivedIRCode

```
void sendReceivedIRCode(decode_type_t protocol, uint32_t value, uint16_t bits);
```

The sendReceivedIRCode function handles sending the IR code received from the IR receiver to the server. This function is primarily used during the setup process, specifically during button mapping when the user presses a button on their physical remote. The function captures the protocol, value, and bit length of the received IR signal and sends this information to the server using an HTTP POST request.

Here is an overview of its purpose and usage:

- **Purpose**: To relay the IR signal received from a physical remote control to the server, typically for button mapping during setup.
- **Usage**: This function is invoked when an IR signal is detected by the IR receiver, capturing the signal details and transmitting them to the server for processing.

The function works as follows:

1. It checks if the ESP32 is connected to WiFi.
2. If connected, it constructs the server URL for sending the IR signal data.
3. The function prepares the JSON payload with the protocol, value, and bits.
4. It sends the HTTP POST request to the server with the JSON payload.
5. Based on the HTTP response code, it logs success or error messages.

These functions are integral to the system, enabling both the transmission of user-initiated IR commands and the reception of signals from physical remotes for setup purposes. By managing these interactions, the firmware ensures seamless control and configuration of IR appliances through the ESP32 microcontroller.

Protocol

Polling

The ESP32 microcontroller polls the server every 250ms to communicate its state and receive commands. This continuous polling mechanism ensures that the ESP32 is always in sync with the server, whether it is ready to send an IR signal or receive one during button mapping. This regular communication also helps determine if the appliance is online or not. The server dictates the state of the ESP32 by sending specific commands.

The ESP32 sends the following JSON message to the server to indicate it is polling:

{ "state": "send", "message": "polling" }

States

The ESP32 can be in one of two states: *send* or *receive*.

The server sets the ESP32 to the send state using a JSON message like this:

{ "time": <Unix Time in seconds>, "message": { "command": "send", "protocol": -1 } }

The server sets the ESP32 to the receive state using a JSON message like this:

{ "time": <Unix Time in seconds>, "message": { "command": "receive" } }

Send State

When the ESP32 is in the send state, it is ready to send any command received from the server. This state is typically used during normal operation when the user interacts with the virtual remote or sends commands via the API.
In this state, if a button is pressed on the virtual remote or a command is sent via the API, the server responds with a message instructing the ESP32 to send the IR signal. The response from the server looks like this:

{ "command": "send", "signal": { "value": 3458, "bits": 12, "protocol": 76 } }

Receive State

When the ESP32 is in the receive state, it is ready to receive an IR signal. This state is used during the button mapping process, where the user needs to press a button on their physical remote to map it to a button on the virtual remote.
During this state, when a button is pressed on the physical remote, the ESP32 captures the IR signal and sends it to the server with the following message:

```
{ "protocol": 76, "value": 3457, "bits": 12 }
```

Transitioning Between States

- **Entering Receive State**: The ESP32 transitions to the receive state during the button mapping process, allowing it to capture signals from the physical remote.

- **Exiting Receive State**: After the button mapping is completed, the ESP32 returns to the send state, either by server command or automatically after 10 minutes.

Additionally, the ESP32 is programmed to automatically return to the send state after 10 minutes if the receive state was initiated and no further instructions were received. This is specified in the firmware with the following line:

```
const unsigned long receiveDuration = 10 * 60 * 1000; // 10 minutes in milliseconds
```

For detailed information on how the server handles the sending and receiving of IR signals, please refer to the IR_Appliance Class section. This section provides comprehensive details on the server's role in managing the IR appliance's commands and state transitions.

Server

The server is built using Express.js and MongoDB and is responsible for handling API requests, managing user authentication, and controlling the appliances. The server directory structure is as follows:

Server Structure

The server is organized into several key components:

1. **IR_Appliance Class**: Manages the state and behavior of IR appliances, including sending and receiving IR signals.
2. **Controllers**: Handle the logic for different routes and manage the interaction between the server and the database.
3. **Models**: Define the schemas for MongoDB collections.
4. **Routes**: Define the endpoints for API requests.
5. **Middleware**: Manage authentication and other request pre-processing tasks.

In-Memory Store

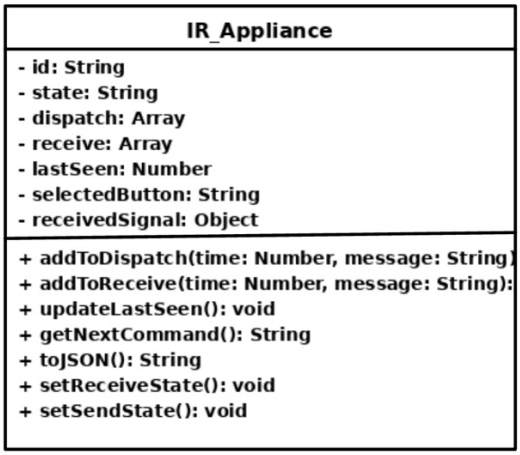
The appliances object is an in-memory store (hash map) that stores appliances that are currently online. This allows for quick access and management of appliances that are actively communicating with the server.

```
const appliances = {};
```

This object keeps track of all the appliances that are currently online, using their IDs as keys and the corresponding IR_Appliance instances as values. It is essential for managing the state and behavior of each appliance in real-time.

IR_Appliance Class

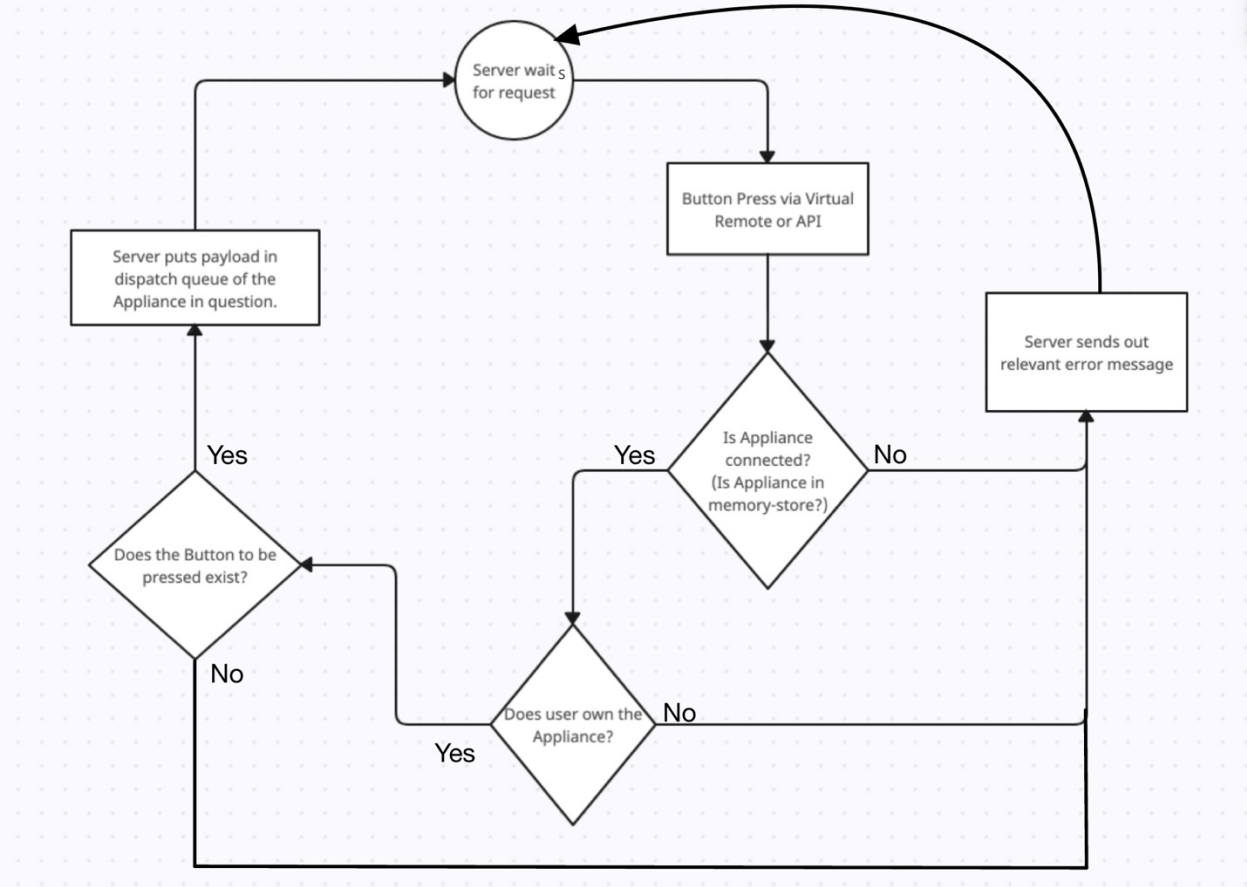
The IR_Appliance class is at the core of managing the state and behavior of the IR appliances. It includes methods for setting the state of the ESP32 devices, handling IR signal sending and receiving, and adding commands to the dispatch queue. This class ensures that the ESP32 devices are in the correct state based on the operations being performed and manages the communication between the devices and the server.



Key Methods

- **addToDispatch(time, message)**: Adds a message to the dispatch queue to be sent to the ESP32.
- **addToReceive(time, message)**: Adds a received message to the receive queue, typically used for capturing IR signals during button mapping.
- **updateLastSeen()**: Updates the last seen timestamp of the appliance.
- **getNextCommand()**: Retrieves the next command from the dispatch queue.
- **toJSON()**: Converts the appliance's state to a JSON object.
- **setReceiveState()**: Sets the appliance state to receive and adds a corresponding message to the dispatch queue.
- **setSendState()**: Sets the appliance state to send and adds a corresponding message to the dispatch queue.

What happens during a Button press?



1. **Server Waits for Request**:
   - The server is in a state of waiting for incoming requests from users.

2. **Button Press via Virtual Remote or API**:
   - A request is initiated when a user presses a button on the virtual remote or via an API call.

3. **Is Appliance Connected? (Is Appliance in memory-store?)**:
   - The server checks if the appliance is currently connected by verifying its presence in the in-memory store (appliances object).
   - **Yes**: If the appliance is connected, the flow continues to the next step.
   - **No**: If the appliance is not connected, the server sends out a relevant error message indicating the appliance is not connected.

4. **Does User Own the Appliance?**:
   - The server verifies if the user initiating the request owns the appliance.
   - **Yes**: If the user owns the appliance, the flow continues to the next step.

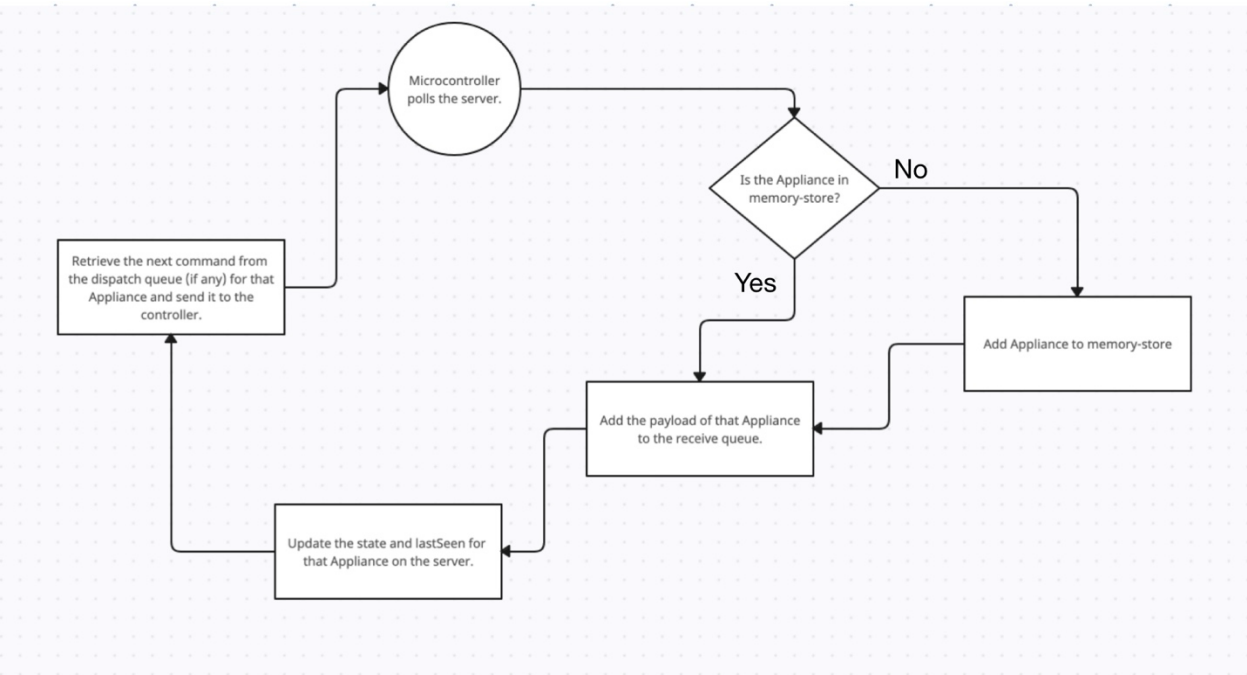
- **No**: If the user does not own the appliance, the server sends out a relevant error message indicating that the user does not have permission to control the appliance.

5. **Does the Button to be Pressed Exist?**:
   - The server checks if the button that the user intends to press exists in the appliance's configuration.
   - **Yes**: If the button exists, the server proceeds to the next step.
   - **No**: If the button does not exist, the server sends out a relevant error message indicating that the button is not configured for the appliance.

6. **Server Puts Payload in Dispatch Queue of the Appliance in Question**:
   - The server places the request payload into the dispatch queue of the specific appliance. This payload includes the necessary command and signal information for the ESP32 to process.

7. **End of Flow**:
   - The server waits for further requests and the cycle continues.

What happens when the ESP32 poll?



1. **Microcontroller polls the server**: The microcontroller sends a request to the server at regular intervals to check for updates or new commands.

2. **Is the Appliance in memory-store?**: The server checks if the appliance (identified by the microcontroller's request) is already stored in memory.
    - **No**:
        - **Add Appliance to memory-store**: If the appliance is not found in the memory store, it is added to it.
        - **Proceed to the next step**: After adding the appliance to the memory store, continue to the next step.

    - **Yes**:
        - **Add the payload of that Appliance to the receive queue**: If the appliance is already in the memory store, its payload is added to the receive queue.

3. **Retrieve the next command from the dispatch queue (if any) for that Appliance and send it to the controller**: The server checks if there are any commands in the dispatch queue for the appliance. If there are, the next command is retrieved and sent to the microcontroller.

4. **Update the state and lastSeen for that Appliance on the server**: The server updates the state and the timestamp of the last interaction (lastSeen) for the appliance.

Routes and Controllers

The routes define the endpoints for interacting with the server. The main route files include:

- **API Routes**: Define endpoints for general API operations.
- **Appliance Routes**: Define endpoints for operations related to appliances.
- **Button Routes**: Define endpoints for button-related operations.
- **User Routes**: Define endpoints for user-related operations.

The controllers are responsible for handling the core logic of the application. They interact with the models to perform CRUD operations and manage the state of the ESP32 devices.

- **API Controllers**: Handle general API requests.
- **Appliance Controllers**: Manage operations related to appliances, including sending and receiving IR signals.
- **Button Controllers**: Handle button mapping and management.
- **User Controllers**: Manage user authentication and account-related operations.

API

Routes (/api/v1)

The routes define the endpoints for interacting with the server. The main routes are defined in the apiRoutes.js file.

1. **POST /press/:apiKey/:remoteIndex**:

- **Description**: Handles button press requests via the API.
- **Controller**: apiController.pressButton
- **Parameters**:
    - apiKey: The API key of the appliance.
    - remoteIndex: The index of the button on the remote.
- **Authentication**: None required.

2. **GET /APIKey/:id**:

- **Description**: Retrieves the API key of an appliance.
- **Controller**: apiController.getAPIKey
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication (authMiddleware).

3. **GET /newKey/:id**:

- **Description**: Generates a new API key for an appliance.
- **Controller**: apiController.newKey
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication (authMiddleware).

Controllers

1. **pressButton**:

This controller handles button press requests by validating the appliance and the button, then adding the command to the dispatch queue. It performs the following steps:

i. Validates the apiKey and checks if the appliance exists in the in-memory store.
ii. If the appliance is connected, it retrieves the button configuration from the database.
iii. If the button exists, it constructs a signal and adds the command to the appliance's dispatch queue.
iv. Returns appropriate responses based on the validation and execution results.

2. **getAPIKey**:

This controller retrieves the API key of an appliance owned by the authenticated user. It performs the following steps:

i. Validates the ownership of the appliance using the ownsAppliance function.
ii. Retrieves the appliance from the database.
iii. Returns the API key of the appliance if it exists and the user owns it.

3. **newKey**:

This controller generates a new API key for an appliance owned by the authenticated user. It performs the following steps:

i. Validates the ownership of the appliance using the ownsAppliance function.
ii. Generates a new API key using the generateUrlSafeKey utility.
iii. Updates the appliance in the database with the new API key.
iv. Returns a success message upon successful update.

Appliance

Routes (/api/appliance)

The routes define the endpoints for interacting with the server. The main routes are defined in the applianceRoutes.js file.

1. **POST /:id/poll**:
- **Description**: Handles polling requests from the ESP32 devices.
- **Controller**: applianceController.poll
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: None required.

2. **POST /receive/:id**:
- **Description**: Handles receiving IR codes from the ESP32 devices.
- **Controller**: applianceController.receive
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: None required.

3. **GET /list**:
- **Description**: Retrieves a list of appliances owned by the authenticated user.
- **Controller**: applianceController.getAppliances
- **Authentication**: Requires user authentication (authMiddleware).

4. **POST /add**:
- **Description**: Adds a new appliance for the authenticated user.
- **Controller**: applianceController.addAppliance
- **Authentication**: Requires user authentication (authMiddleware).

5. **POST /:id/mapButton**:
- **Description**: Maps a button to an appliance.
- **Controller**: applianceController.mapButton
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication (authMiddleware).

6. **GET /:id/getCommand/:remoteIndex**:

- **Description**: Retrieves the command associated with a button on the remote.
- **Controller**: applianceController.getCommand
- **Parameters**:
    - id: The ID of the appliance.
    - remoteIndex: The index of the button on the remote.
- **Authentication**: Requires user authentication (authMiddleware).

7. **DELETE /**:

- **Description**: Deletes an appliance and its associated buttons.
- **Controller**: applianceController.deleteAppliance
- **Authentication**: Requires user authentication (authMiddleware).

8. **POST /setSendState/:id**:

- **Description**: Sets the appliance to the send state.
- **Controller**: applianceController.setSendState
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication (authMiddleware).

Controllers

The controllers handle the core logic for the appliance-related operations, interacting with the models to perform CRUD operations and manage the state of the ESP32 devices.

1. **poll**:
    i. Checks if the appliance exists in the in-memory store. If not, it creates a new instance.
    ii. Updates the appliance's state and last seen time.
    iii. Adds the received message to the appliance's queue.
    iv. Retrieves the next command for the appliance and sends it as a response.

2. **receive**:
    i. Logs the received IR code details (protocol, value, bits).
    ii. If the appliance is in the receive state, it adds the received IR code to the appliance's queue.
    iii. Returns a success message as a response.

3. **addAppliance**:
   i. Validates the appliance ID and checks if the appliance is connected.
   ii. Checks if the appliance is already added or owned by another user.
   iii. Generates a new API key and saves the appliance to the database.
   iv. Returns a success message upon successful addition.

4. **getAppliances**:
   i. Queries the database for appliances owned by the user.
   ii. Includes the last seen time if the appliance is currently connected.
   iii. Returns the list of appliances as a response.

5. **deleteAppliance**:
   i. Validates the appliance ID and ownership.
   ii. Deletes the appliance from the database.
   iii. Deletes all buttons associated with the appliance.
   iv. Returns a success message upon successful deletion.

6. **setSendState**:
   i. Validates ownership of the appliance.
   ii. Checks if the appliance is connected.
   iii. Updates the appliance's state to send.
   iv. Returns a success message as a response.

Button

Routes (/api/button)

The routes define the endpoints for interacting with the server. The main routes are defined in the buttonRoutes.js file.

1. **POST /select**:

- **Description**: Selects a button for mapping.
- **Controller**: buttonController.selectButton
- **Authentication**: Requires user authentication.

2. **POST /unselect/:id**:

- **Description**: Unselects a button.
- **Controller**: buttonController.unselectButton
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication.

3. **GET /signalStatus/:id**:

- **Description**: Gets the status of the signal received for a button.
- **Controller**: buttonController.getSignalStatus
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication.

4. **POST /save/:id**:

- **Description**: Saves a mapped button.
- **Controller**: buttonController.saveButton
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication.

5. **GET /list/:id**:

- **Description**: Retrieves a list of buttons mapped to an appliance.
- **Controller**: buttonController.getButtonList
- **Parameters**:
    - id: The ID of the appliance.
- **Authentication**: Requires user authentication.

6. **POST /click**:

- **Description**: Simulates a button click.
- **Controller**: buttonController.clickButton
- **Authentication**: Requires user authentication.

7. **DELETE /:applianceId/:remoteIndex**:

- **Description**: Deletes a button from an appliance.
- **Controller**: buttonController.deleteButton
- **Parameters**:
    - applianceId: The ID of the appliance.
    - remoteIndex: The index of the button on the remote.
- **Authentication**: Requires user authentication.

Controllers

The controllers handle the core logic for button-related operations, interacting with the models to perform CRUD operations and manage the state of the ESP32 devices.

1. **selectButton**:
    i. Validates ownership of the appliance.
    ii. Checks if the appliance is connected.
    iii. Sets the selected button and clears any previous received signal.
    iv. Puts the appliance into receive state.
    v. Returns a success message upon successful selection.

2. **unselectButton**:
    i. Validates ownership of the appliance.
    ii. Checks if the appliance is connected.
    iii. Clears the selected button and received signal.
    iv. Puts the appliance into send state.
    v. Returns a success message upon successful unselection.

3. **getSignalStatus**:
    i. Validates ownership of the appliance.
    ii. Checks if the appliance is connected.
    iii. Returns the status of the received signal.
    iv. Returns “Wait” if no signal is received.
    v. Returns “Try again” if the received signal is invalid.
    vi. Returns “OK” if a valid signal is received.

4. **saveButton**:
    i. Validates ownership of the appliance.
    ii. Checks if a valid signal and selected button are present.
    iii. Saves the button configuration to the database.

iv. Updates an existing button if one with the same remoteIndex exists.
v. Creates a new button if none exists with the same remoteIndex.
vi. Returns a success message upon successful save.

5. **getButtonList**:
   i. Validates ownership of the appliance.
   ii. Queries the database for buttons mapped to the appliance.
   iii. Returns the list of buttons or an appropriate message if no buttons are found.

6. **clickButton**:
   i. Validates ownership of the appliance.
   ii. Checks if the appliance is connected.
   iii. Retrieves the button configuration from the database.
   iv. Adds the command to the appliance's dispatch queue.
   v. Returns a success message upon successful command dispatch.

7. **deleteButton**:
   i. Validates ownership of the appliance.
   ii. Deletes the button from the database based on applianceId and remoteIndex.
   iii. Returns a success message upon successful deletion.

User

Routes (/api/user)

The routes define the endpoints for interacting with the server. The main routes are defined in the userRoutes.js file.

1. **POST /register**:
- **Description**: Registers a new user.
- **Controller**: userController.registerUser
- **Authentication**: None required.

2. **POST /login**:
- **Description**: Logs in an existing user.
- **Controller**: userController.loginUser
- **Authentication**: None required.

3. **DELETE /**:

- **Description**: Deletes the currently authenticated user.
- **Controller**: userController.deleteUser
- **Authentication**: Requires user authentication (authMiddleware).

4. **POST /changePassword**:

- **Description**: Changes the password of the currently authenticated user.
- **Controller**: userController.changePassword
- **Authentication**: Requires user authentication (authMiddleware).

Controllers

The controllers handle the core logic for user-related operations, interacting with the models to perform CRUD operations and manage user authentication.

1. **registerUser**:
   i. Extracts the username and password from the request body.
   ii. Creates a new User instance with the provided username and password.
   iii. Saves the user to the database.
   iv. Returns a success message upon successful registration.
   v. Handles and returns an error message if the registration fails.

2. **loginUser**:
   i. Extracts the username and password from the request body.
   ii. Finds a user with the provided username in the database.
   iii. If the user is not found, returns an error message indicating invalid credentials.
   iv. Compares the provided password with the stored password using the comparePassword method.
   v. If the password does not match, returns an error message indicating invalid credentials.
   vi. If the password matches, generates a JWT token with the user's ID and username, and sets an expiration time.
   vii. Returns the generated token in the response.
   viii. Handles and returns an error message if the login fails.

3. **deleteUser**:
   i. Deletes the currently authenticated user from the database.
   ii. Finds and deletes all appliances owned by the user.
   iii. Deletes all buttons associated with the user's appliances.
   iv. Returns a success message upon successful deletion.
   v. Handles and returns an error message if the deletion fails.

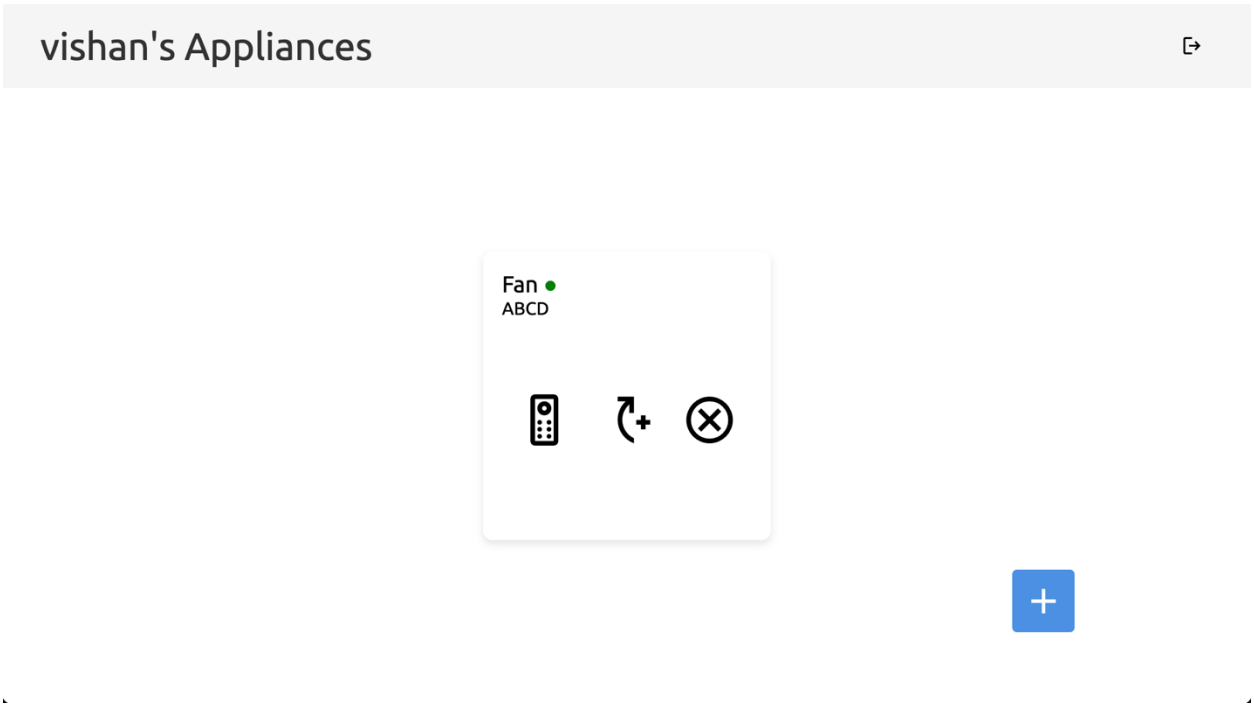
4. **changePassword**:
   i. Extracts the current password, new password, and revokeAPIKeys flag from the request body.
   ii. Finds the currently authenticated user in the database.
   iii. If the user is not found, returns an error message indicating the user is not found.
   iv. Compares the current password with the stored password using the comparePassword method.
   v. If the password does not match, returns an error message indicating invalid password.
   vi. If the password matches, updates the user's password and increments the loginTokenVersion.
   vii. Saves the updated user to the database.
   viii. If revokeAPIKeys is true, generates new API keys for all appliances owned by the user.
   ix. Returns a success message upon successful password change.
   x. Handles and returns an error message if the password change fails.

Frontend

The frontend is built using React and includes various components for the user interface.

Components Overview

Protected



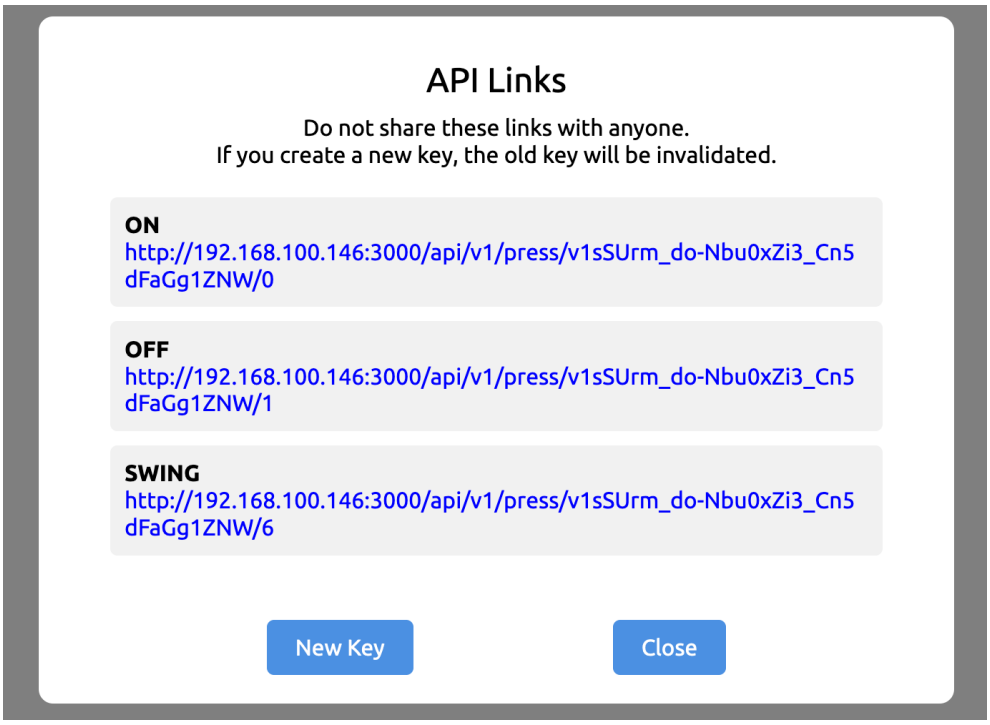
Card

- **Green Indicator**: Indicates that the appliance is online.
- **Red Indicator**: Indicates that the appliance is offline.

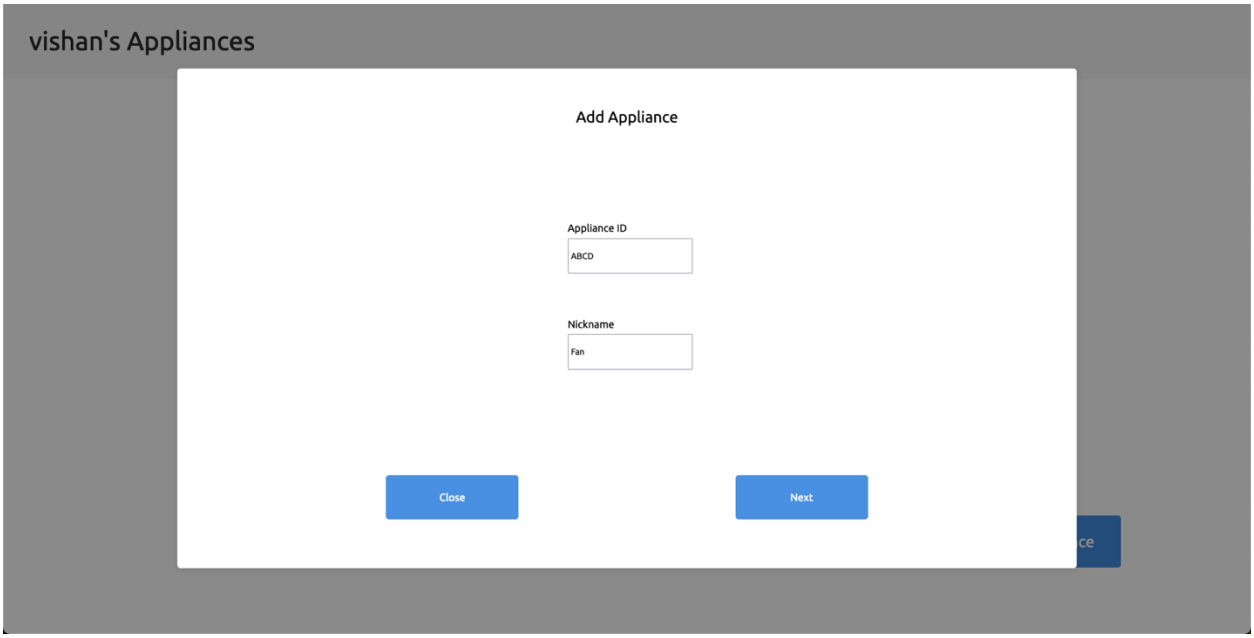
- **First Button (left to right)**: Accesses the remote-control interface.
- **Second Button**: Allows for remapping, deleting, or mapping new buttons.
- **Last Button**: Deletes the appliance.

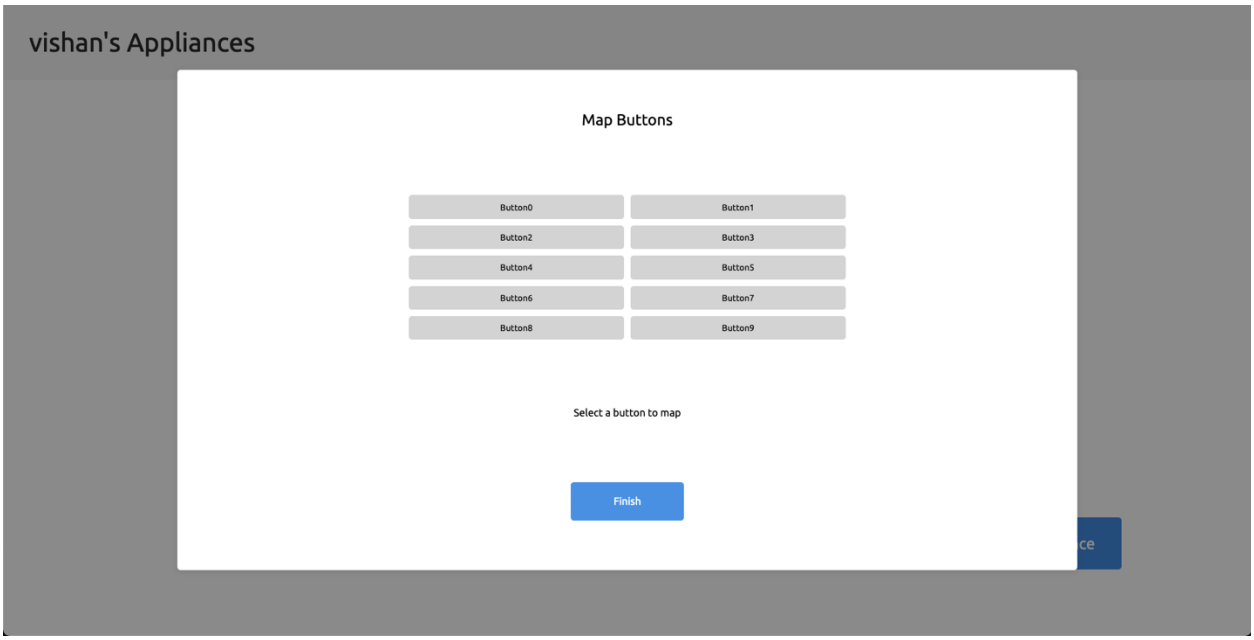
API_Links

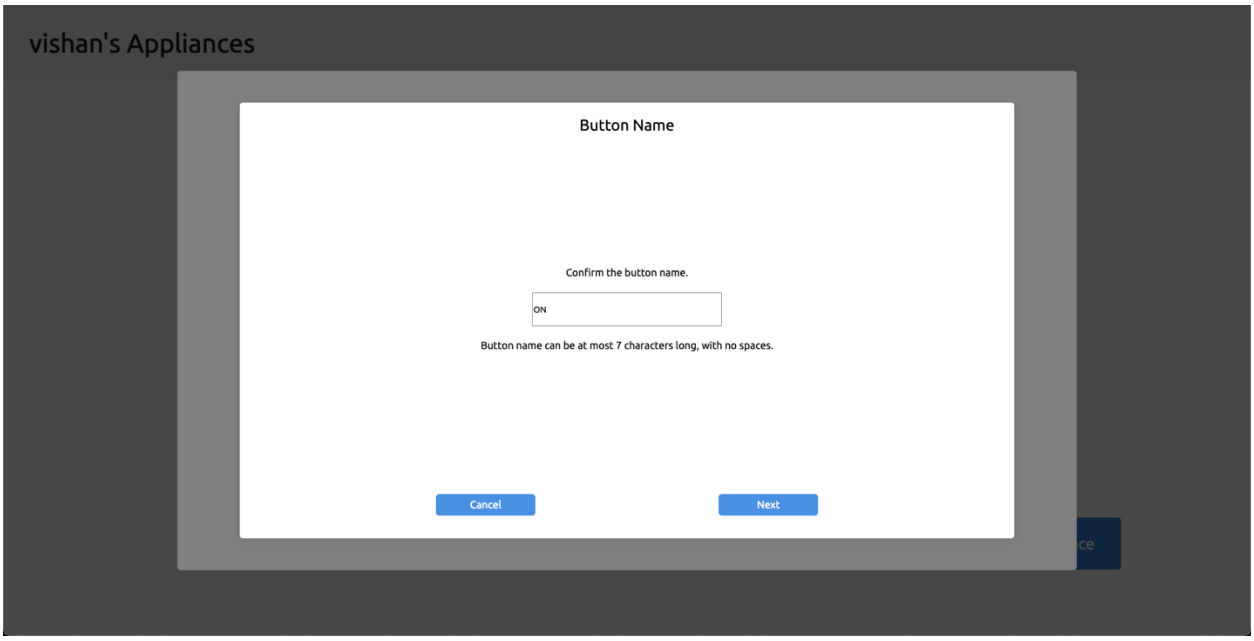
This menu can be accessed by clicking on the appliance name on its Card.

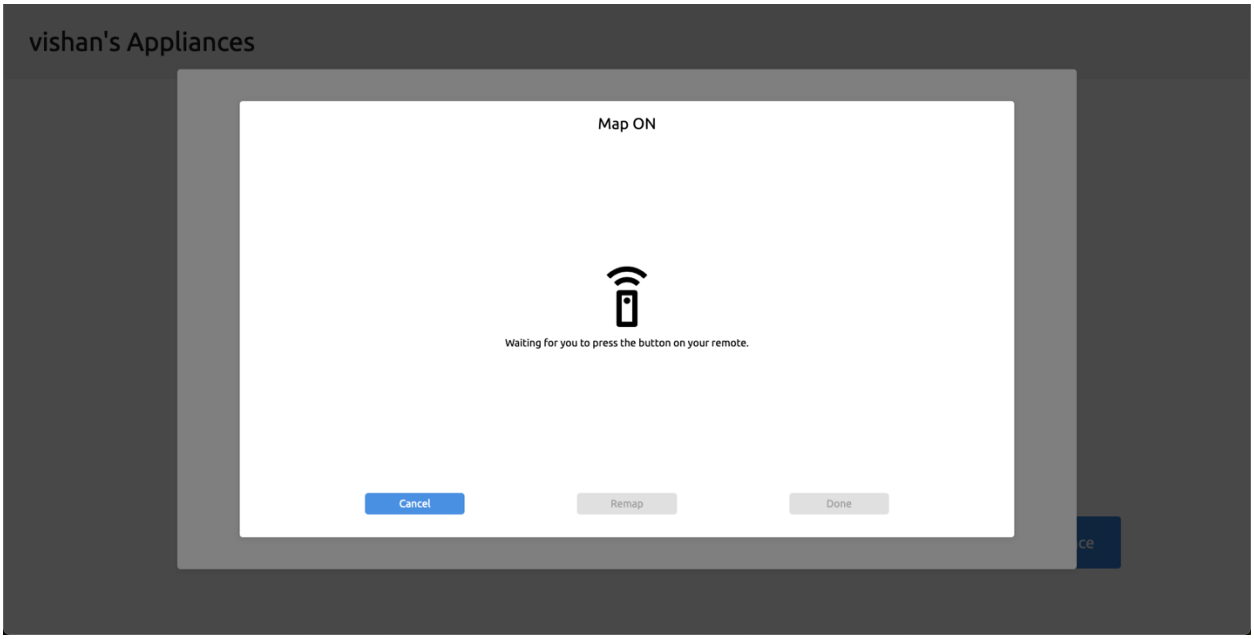


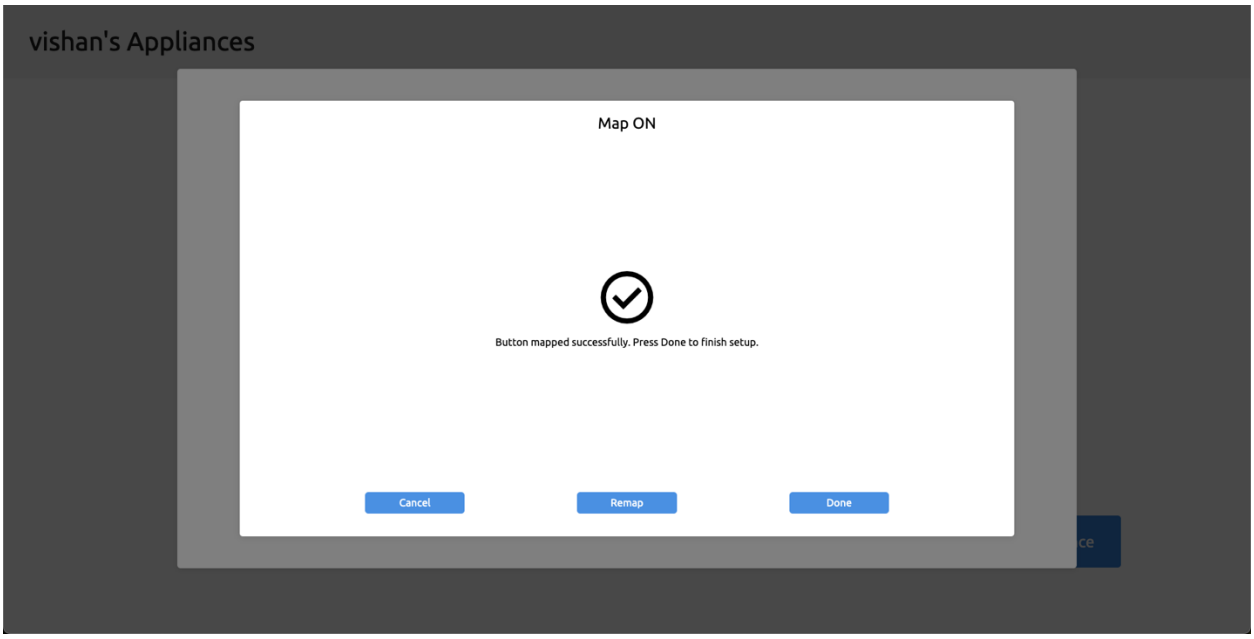
SetupWizard



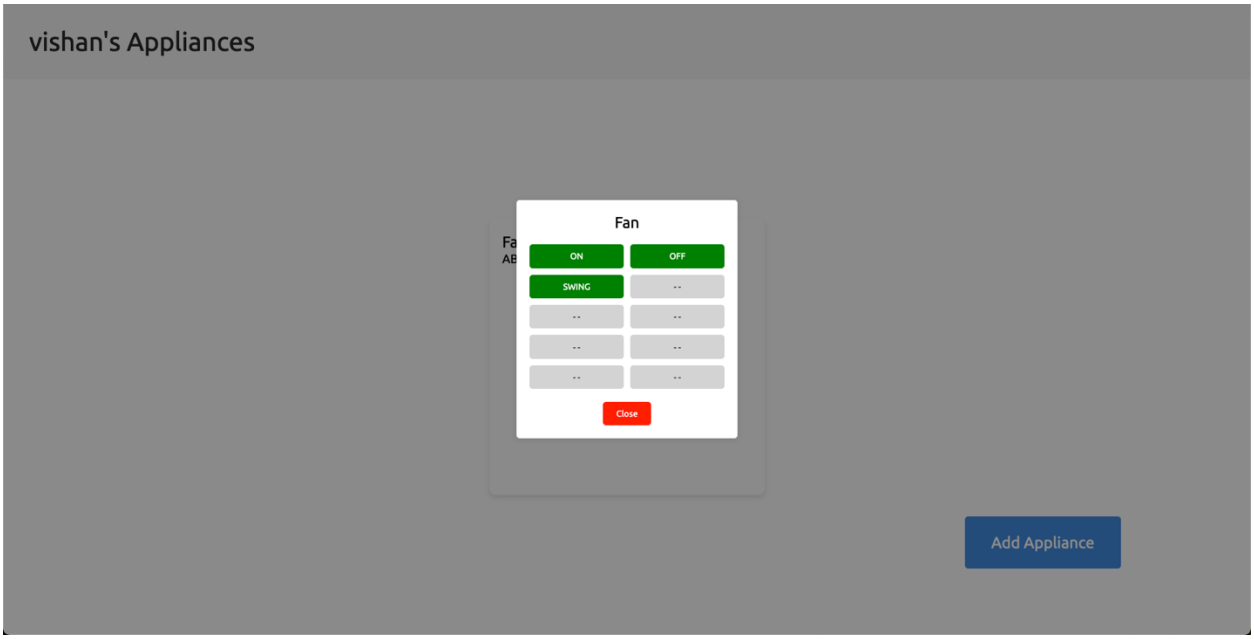



vishan's Appliances
Button Name
Confirm the button name.
ON
Button name can be at most 7 characters long, with no spaces.
Cancel
Next


vishan's Appliances
Map ON
Waiting for you to press the button on your remote.
Cancel
Remap
Done



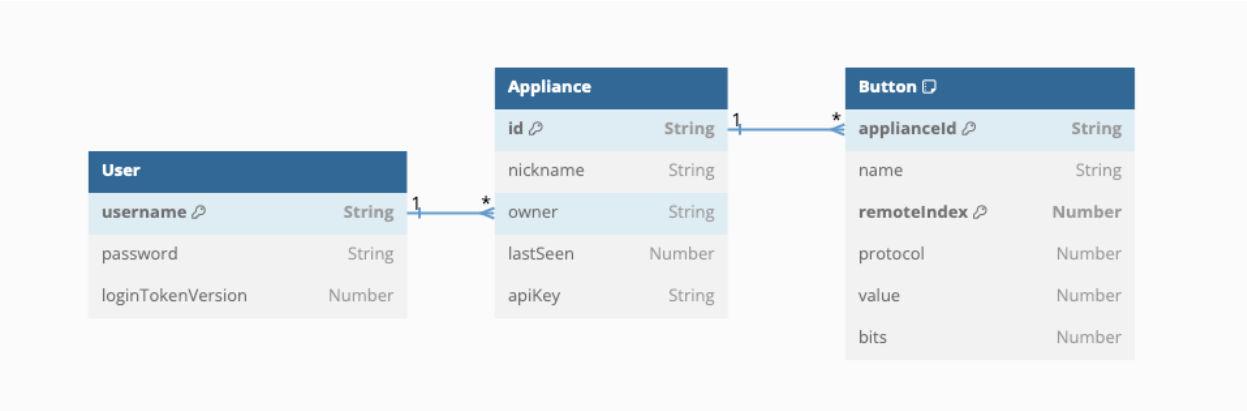
VirtualRemote



Database Schema

This section outlines the database schema used in the IR control system, detailing the structures for users, appliances, and buttons. Each schema is defined using Mongoose, a MongoDB object modeling tool.

Entity-Relationship Diagram



User Schema

The User schema defines the structure for storing user information in the database. It includes fields for the username and password, along with methods for hashing passwords and comparing entered passwords with the stored hashed passwords.

Schema Definition

- username (String, required, unique, trimmed): The user's unique username.
- password (String, required): The user's password, which is hashed before storage.
- **loginTokenVersion** (Number, default: 0): Version number for login tokens to handle token invalidation on password change

Features

- **Pre-save Hook**: Hashes the password before saving the user to ensure security.
- **Password Comparison Method**: Compares the entered password with the hashed password in the database.

Appliance Schema

The Appliance schema defines the structure for storing information about appliances. It includes fields for the appliance ID, nickname, owner, last seen timestamp, and API key.

Schema Definition

- id (String, required, unique): The unique identifier for the appliance.
- nickname (String, optional): The nickname for the appliance.
- **Validator**: Ensures the nickname is not an empty string.
- owner (String, optional): The username of the appliance's owner.
- lastSeen (Number, optional): The last seen timestamp of the appliance.
- apiKey (String, optional): The API key for the appliance.

Button Schema

The Button schema defines the structure for storing button mappings associated with appliances. It includes fields for the appliance ID, button name, remote index, protocol, value, and bit length.

Schema Definition

- applianceId (String, required, indexed): The ID of the appliance the button is associated with.
- name (String, required): The name of the button.
- remoteIndex (Number, required, indexed): The index of the button on the remote.
- protocol (Number, required): The protocol of the button's IR signal.
- value (Number, required): The value of the button's IR signal.
- bits (Number, required): The bit length of the button's IR signal.

Note

There is no upper limit on the number of buttons an appliance can have within the server or database. However, a practical limit of 10 buttons is imposed on the client-side interface. This restriction helps maintain a user-friendly experience without affecting the system's overall capability to handle a larger number of buttons.

Authentication

Middleware

The authentication middleware (authMiddleware.js) is responsible for verifying the authenticity of JSON Web Tokens (JWT) provided in the request headers. It ensures that only authenticated users can access protected routes. The middleware also checks the token version to handle token invalidation when a user's password is changed.

Functionality

1. **Token Extraction**:
- The middleware extracts the JWT from the Authorization header of the incoming request. The expected format is Bearer <token>.

2. **Token Verification**:
- The extracted token is verified using the secret key. The secret key can be configured through an environment variable (JWT_SECRET) or a default value.

3. **User Validation**:
- The middleware decodes the token to retrieve the username and login token version.
- It queries the database to find the user by the username extracted from the token.

4. **Token Version Check**:
- The middleware compares the loginTokenVersion from the token with the loginTokenVersion stored in the user's record. This ensures the token is still valid, especially after password changes which would increment the token version and invalidate older tokens.

5. **Request Handling**:
- If the token is valid and the user exists, the decoded token information is attached to the req.user object.
- The request is then passed to the next middleware or route handler.
- If the token is invalid or the user is not found, a 401 Unauthorized response is sent.

User Authentication Flow

The authentication flow ensures that only authenticated users can access protected resources. It involves registering a user, logging in, and verifying the JWT for subsequent requests.

1. **User Registration**:
- The user submits a registration request with a username and password.
- The server hashes the password and saves the user information in the database.

2. **User Login**:
- The user submits a login request with a username and password.
- The server verifies the credentials:
- If valid, the server generates a JWT containing the user's ID, username, and loginTokenVersion.
- The JWT is sent to the client with an expiration time.

3. **Accessing Protected Routes**:
- The client includes the JWT in the Authorization header of requests to protected routes (Bearer <token>).
- The authentication middleware:
- Extracts and verifies the token.
- Decodes the token to retrieve user information.
- Checks the loginTokenVersion to ensure the token is not outdated.
- Attaches the user information to the request object and passes it to the next handler.

4. **Token Invalidation**:
- When a user changes their password, the loginTokenVersion is incremented.
- Any JWTs issued before the password change will be invalidated as the token version will no longer match.

Environment Variables

.env file

1. **PORT**:

- **Description**: Specifies the port number on which your server will listen for incoming requests.
- **Default**: PORT=3000

2. **DB_USERNAME**:

- **Description**: The username used to authenticate with your database.

3. **DB_PASSWORD**:

- **Description**: The password used to authenticate with your database.

4. **CLEANUP_INTERVAL**:

- **Description**: The interval in milliseconds at which the server will clean up the in-memory store, removing offline appliances. This helps to keep the in-memory store clean and free of stale data.
- **Default**: CLEANUP_INTERVAL=30000 (which is 30 seconds)

5. **JWT_SECRET**:

- **Description**: A secret key used for signing and verifying JSON Web Tokens (JWT) for authentication and authorization purposes.

6. **JWT_EXPIRATION:**

- **Description**: The number of days before the authentication token of each client expires.
- **Default**: JWT_EXPIRATION=7d

7. **API_KEY_LENGTH**:

- **Description**: Specifies the length of the API keys that are generated for appliances. This determines how many characters long each API key will be.
- **Default**: API_KEY_LENGTH=32

Deployment

Local Deployment

1. Clone the repository.
2. Install dependencies: `npm install`.
3. Configure environment variables.
4. Make sure you have a MongoDB cluster already set up. You can use a local MongoDB instance or a cloud service like MongoDB Atlas. The connection to the database is made in the *server.cjs* file.
5. Set the correct API_BASE_URL constant in src/constants.js
6. Start the server: `npm start`.

Usage

Assuming that you've already created and logged into your account.

Adding Appliance

1. Click on the '+' button.
2. Follow the setup wizard.

Notes:

- Ensure that the protocol for the appliance is added to the switch case in the firmware.
- When the ESP32 is in receiving mode, the ESP32's LED will blink slowly, indicating it is in receiving mode for mapping.

Current Switch Case Implementation

```
switch (protocol) {
  case SYMPHONY:
    irsend.sendSymphony(value, bits);
    break;

  case NEC:
    irsend.sendNEC(value, bits);
    break;

  // Add more cases here for additional protocols
}
```

Supported Protocols with Indices

- **UNKNOWN**
  - **Index**: -1
  - **Function**: N/A
- **RC5**
  - **Index**: 1
  - **Function**: irsend.sendRC5(value, bits);
- **RC6**
  - **Index**: 2
  - **Function**: irsend.sendRC6(value, bits);
- **NEC**
  - **Index**: 3
  - **Function**: irsend.sendNEC(value, bits);
- **SONY**
  - **Index**: 4
  - **Function**: irsend.sendSony(value, bits);
- **PANASONIC**
  - **Index**: 5
  - **Function**: irsend.sendPanasonic(value, bits);
- **JVC**
  - **Index**: 6
  - **Function**: irsend.sendJVC(value, bits);
- **SAMSUNG**
  - **Index**: 7
  - **Function**: irsend.sendSAMSUNG(value, bits);
- **WHYNTER**
  - **Index**: 8
  - **Function**: irsend.sendWhynter(value, bits);
- **AIWA_RC_T501**
  - **Index**: 9
  - **Function**: irsend.sendAiwaRCT501(value, bits);
- **LG**
  - **Index**: 10
  - **Function**: irsend.sendLG(value, bits);
- **SANYO**
  - **Index**: 11
  - **Function**: irsend.sendSanyoLC7461(value, bits);
- **MITSUBISHI**
  - **Index**: 12
  - **Function**: irsend.sendMitsubishi(value, bits);

- **DISH**
  - **Index**: 13
  - **Function**: irsend.sendDISH(value, bits);
- **SHARP**
  - **Index**: 14
  - **Function**: irsend.sendSharp(value, bits);
- **COOLIX**
  - **Index**: 15
  - **Function**: irsend.sendCOOLIX(value, bits);
- **DAIKIN**
  - **Index**: 16
  - **Function**: irsend.sendDaikin(value, bits);
- **DENON**
  - **Index**: 17
  - **Function**: irsend.sendDenon(value, bits);
- **KELVINATOR**
  - **Index**: 18
  - **Function**: irsend.sendKelvinator(value, bits);
- **SHERWOOD**
  - **Index**: 19
  - **Function**: irsend.sendSherwood(value, bits);
- **MITSUBISHI_AC**
  - **Index**: 20
  - **Function**: irsend.sendMitsubishiAC(value, bits);
- **RCMM**
  - **Index**: 21
  - **Function**: irsend.sendRCMM(value, bits);
- **SANYO_LC7461**
  - **Index**: 22
  - **Function**: irsend.sendSanyoLC7461(value, bits);
- **RC5X**
  - **Index**: 23
  - **Function**: irsend.sendRC5X(value, bits);
- **GREE**
  - **Index**: 24
  - **Function**: irsend.sendGree(value, bits);
- **PRONTO**
  - **Index**: 25
  - **Function**: irsend.sendPronto(value, bits);
- **NEC_LIKE**
  - **Index**: 26
  - **Function**: irsend.sendNEC(value, bits); // Assuming similar to NEC

- **ARGO**

    - **Index**: 27
    - **Function**: irsend.sendArgo(value, bits);

- **TROTEC**
    - **Index**: 28
    - **Function**: irsend.sendTrotec(value, bits);
- **NIKAI**
    - **Index**: 29
    - **Function**: irsend.sendNikai(value, bits);
- **RAW**
    - **Index**: 30
    - **Function**: irsend.sendRaw(value, bits);
- **GLOBALCACHE**
    - **Index**: 31
    - **Function**: irsend.sendGC(value, bits);
- **TOSHIBA_AC**
    - **Index**: 32
    - **Function**: irsend.sendToshibaAC(value, bits);
- **FUJITSU_AC**
    - **Index**: 33
    - **Function**: irsend.sendFujitsuAC(value, bits);
- **MIDEA**
    - **Index**: 34
    - **Function**: irsend.sendMidea(value, bits);
- **MAGIQUEST**
    - **Index**: 35
    - **Function**: irsend.sendMagiQuest(value, bits);
- **LASERTAG**
    - **Index**: 36
    - **Function**: irsend.sendLasertag(value, bits);
- **CARRIER_AC**
    - **Index**: 37
    - **Function**: irsend.sendCarrierAC(value, bits);
- **HAIER_AC**
    - **Index**: 38
    - **Function**: irsend.sendHaierAC(value, bits);
- **MITSUBISHI2**
    - **Index**: 39
    - **Function**: irsend.sendMitsubishi2(value, bits);
- **HITACHI_AC**
    - **Index**: 40
    - **Function**: irsend.sendHitachiAC(value, bits);
- **HITACHI_AC1**

    - **Index**: 41
    - **Function**: irsend.sendHitachiAC1(value, bits);

- **HITACHI_AC2**
    - **Index**: 42
    - **Function**: irsend.sendHitachiAC2(value, bits);
- **GICABLE**
    - **Index**: 43
    - **Function**: irsend.sendGICable(value, bits);
- **HAIER_AC_YRW02**
    - **Index**: 44
    - **Function**: irsend.sendHaierACYRW02(value, bits);
- **WHIRLPOOL_AC**
    - **Index**: 45
    - **Function**: irsend.sendWhirlpoolAC(value, bits);
- **SAMSUNG_AC**
    - **Index**: 46
    - **Function**: irsend.sendSamsungAC(value, bits);
- **LUTRON**
    - **Index**: 47
    - **Function**: irsend.sendLutron(value, bits);
- **ELECTRA_AC**
    - **Index**: 48
    - **Function**: irsend.sendElectraAC(value, bits);
- **PANASONIC_AC**
    - **Index**: 49
    - **Function**: irsend.sendPanasonicAC(value, bits);
- **PIONEER**
    - **Index**: 50
    - **Function**: irsend.sendPioneer(value, bits);
- **LG2**
    - **Index**: 51
    - **Function**: irsend.sendLG2(value, bits);
- **MWM**
    - **Index**: 52
    - **Function**: irsend.sendMWM(value, bits);
- **DAIKIN2**
    - **Index**: 53
    - **Function**: irsend.sendDaikin2(value, bits);
- **VESTEL_AC**
    - **Index**: 54
    - **Function**: irsend.sendVestelAc(value, bits);
- **TECO**

    - **Index**: 55
    - **Function**: irsend.sendTeco(value, bits);

- **SAMSUNG36**
    - **Index**: 56
    - **Function**: irsend.sendSamsung36(value, bits);
- **TCL112AC**
    - **Index**: 57
    - **Function**: irsend.sendTcl112Ac(value, bits);
- **LEGOPF**
    - **Index**: 58
    - **Function**: irsend.sendLegoPf(value, bits);
- **MITSUBISHI_HEAVY_88**
    - **Index**: 59
    - **Function**: irsend.sendMitsubishiHeavy88(value, bits);
- **MITSUBISHI_HEAVY_152**
    - **Index**: 60
    - **Function**: irsend.sendMitsubishiHeavy152(value, bits);
- **DAIKIN216**
    - **Index**: 61
    - **Function**: irsend.sendDaikin216(value, bits);
- **SHARP_AC**
    - **Index**: 62
    - **Function**: irsend.sendSharpAc(value, bits);
- **GOODWEATHER**
    - **Index**: 63
    - **Function**: irsend.sendGoodweather(value, bits);
- **INAX**
    - **Index**: 64
    - **Function**: irsend.sendInax(value, bits);
- **DAIKIN160**
    - **Index**: 65
    - **Function**: irsend.sendDaikin160(value, bits);
- **NEOCLIMA**
    - **Index**: 66
    - **Function**: irsend.sendNeoclima(value, bits);
- **DAIKIN176**
    - **Index**: 67
    - **Function**: irsend.sendDaikin176(value, bits);
- **DAIKIN128**
    - **Index**: 68
    - **Function**: irsend.sendDaikin128(value, bits);
- **AMCOR**

    - **Index**: 69
    - **Function**: irsend.sendAmcor(value, bits);

- **DAIKIN152**
    - **Index**: 70
    - **Function**: irsend.sendDaikin152(value, bits);
- **MITSUBISHI136**
    - **Index**: 71
    - **Function**: irsend.sendMitsubishi136(value, bits);
- **MITSUBISHI112**
    - **Index**: 72
    - **Function**: irsend.sendMitsubishi112(value, bits);
- **HITACHI_AC424**
    - **Index**: 73
    - **Function**: irsend.sendHitachiAc424(value, bits);
- **SONY_38K**
    - **Index**: 74
    - **Function**: irsend.sendSony38(value, bits);
- **EPSON**
    - **Index**: 75
    - **Function**: irsend.sendEpson(value, bits);
- **SYMPHONY**
    - **Index**: 76
    - **Function**: irsend.sendSymphony(value, bits);
- **HITACHI_AC3**
    - **Index**: 77
    - **Function**: irsend.sendHitachiAc3(value, bits);
- **DAIKIN64**
    - **Index**: 78
    - **Function**: irsend.sendDaikin64(value, bits);
- **AIRWELL**
    - **Index**: 79
    - **Function**: irsend.sendAirwell(value, bits);
- **DELONGHI_AC**
    - **Index**: 80
    - **Function**: irsend.sendDelonghiAc(value, bits);
- **DOSHISHA**
    - **Index**: 81
    - **Function**: irsend.sendDoshisha(value, bits);
- **MULTIBRACKETS**
    - **Index**: 82
    - **Function**: irsend.sendMultibrackets(value, bits);
- **CARRIER_AC40**

    - **Index**: 83
    - **Function**: irsend.sendCarrierAC40(value, bits);

- **CARRIER_AC64**
    - **Index**: 84
    - **Function**: irsend.sendCarrierAC64(value, bits);
- **HITACHI_AC344**
    - **Index**: 85
    - **Function**: irsend.sendHitachiAc344(value, bits);
- **CORONA_AC**
    - **Index**: 86
    - **Function**: irsend.sendCoronaAc(value, bits);
- **MIDEA24**
    - **Index**: 87
    - **Function**: irsend.sendMidea24(value, bits);
- **ZEPEAL**
    - **Index**: 88
    - **Function**: irsend.sendZepeal(value, bits);
- **SANYO_AC**
    - **Index**: 89
    - **Function**: irsend.sendSanyoAc(value, bits);
- **VOLTAS**
    - **Index**: 90
    - **Function**: irsend.sendVoltas(value, bits);
- **METZ**
    - **Index**: 91
    - **Function**: irsend.sendMetz(value, bits);
- **TRANSCOLD**
    - **Index**: 92
    - **Function**: irsend.sendTranscold(value, bits);
- **TECHNIBEL_AC**
    - **Index**: 93
    - **Function**: irsend.sendTechnibelAc(value, bits);
- **MIRAGE**
    - **Index**: 94
    - **Function**: irsend.sendMirage(value, bits);
- **ELITESCREENS**
    - **Index**: 95
    - **Function**: irsend.sendElitescreens(value, bits);
- **PANASONIC_AC32**
    - **Index**: 96
    - **Function**: irsend.sendPanasonicAC32(value, bits);
- **MILESTAG2**

    - **Index**: 97
    - **Function**: irsend.sendMilestag2(value, bits);

- **ECOCLIM**
    - **Index**: 98
    - **Function**: irsend.sendEcoclim(value, bits);
- **XMP**
    - **Index**: 99
    - **Function**: irsend.sendXmp(value, bits);
- **TRUMA**
    - **Index**: 100
    - **Function**: irsend.sendTruma(value, bits);
- **HAIER_AC176**
    - **Index**: 101
    - **Function**: irsend.sendHaierAC176(value, bits);
- **TEKNOPOINT**
    - **Index**: 102
    - **Function**: irsend.sendTeknopoint(value, bits);
- **KELON**
    - **Index**: 103
    - **Function**: irsend.sendKelon(value, bits);
- **TROTEC_3550**
    - **Index**: 104
    - **Function**: irsend.sendTrotec3550(value, bits);
- **SANYO_AC88**
    - **Index**: 105
    - **Function**: irsend.sendSanyoAc88(value, bits);
- **BOSE**
    - **Index**: 106
    - **Function**: irsend.sendBose(value, bits);
- **ARRIS**
    - **Index**: 107
    - **Function**: irsend.sendArris(value, bits);
- **RHOSS**
    - **Index**: 108
    - **Function**: irsend.sendRhoss(value, bits);
- **AIRTON**
    - **Index**: 109
    - **Function**: irsend.sendAirton(value, bits);
- **COOLIX48**
    - **Index**: 110
    - **Function**: irsend.sendCoolix48(value, bits);
- **HITACHI_AC264**

    - **Index**: 111
    - **Function**: irsend.sendHitachiAc264(value, bits);

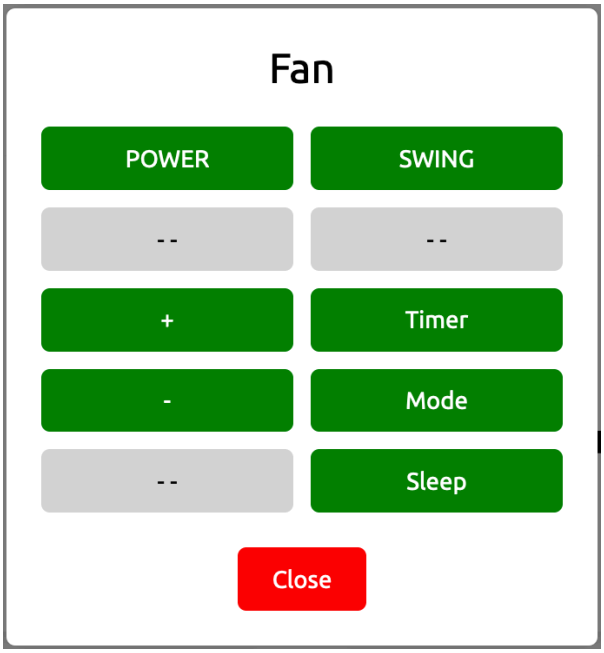
- **KELON168**
    - **Index**: 112
    - **Function**: irsend.sendKelon168(value, bits);
- **HITACHI_AC296**
    - **Index**: 113
    - **Function**: irsend.sendHitachiAc296(value, bits);
- **DAIKIN200**
    - **Index**: 114
    - **Function**: irsend.sendDaikin200(value, bits);
- **HAIER_AC160**
    - **Index**: 115
    - **Function**: irsend.sendHaierAC160(value, bits);
- **CARRIER_AC128**
    - **Index**: 116
    - **Function**: irsend.sendCarrierAC128(value, bits);
- **TOTO**
    - **Index**: 117
    - **Function**: irsend.sendToto(value, bits);
- **CLIMABUTLER**
    - **Index**: 118
    - **Function**: irsend.sendClimaButler(value, bits);
- **TCL96AC**
    - **Index**: 119
    - **Function**: irsend.sendTcl96Ac(value, bits);
- **BOSCH144**
    - **Index**: 120
    - **Function**: irsend.sendBosch144(value, bits);
- **SANYO_AC152**
    - **Index**: 121
    - **Function**: irsend.sendSanyoAc152(value, bits);
- **DAIKIN312**
    - **Index**: 122
    - **Function**: irsend.sendDaikin312(value, bits);
- **GORENJE**
    - **Index**: 123
    - **Function**: irsend.sendGorenje(value, bits);
- **WOWWEE**
    - **Index**: 124
    - **Function**: irsend.sendWowwee(value, bits);
- **CARRIER_AC84**

- **Index**: 125
- **Function**: irsend.sendCarrierAC84(value, bits);

- **YORK**
  - **Index**: 126
  - **Function**: irsend.sendYork(value, bits);

Controlling Appliances

1. Click on the Remote button (first button, starting from left to right) for the appliance you want to control.
2. Click on the desired button on the virtual remote.



Automating Tasks with Shortcuts

1. Open the Shortcuts app on your iPhone.
2. Create a new shortcut.
3. Use the API to integrate the control of appliances. Click on the appliance name to see the API routes available to you. You should make POST request.

How to change your password

1. **Access Your Account Settings:** Click on your username located at the upper left corner of the screen to open the Account Settings menu.
2. **Initiate the Password Change:** In the Account Settings menu, click on the “Change Password” button.
3. **Enter and Confirm New Password:** Enter your new password in the provided fields and confirm it.
4. **Revoke API Keys (Optional):** If you want to revoke all API keys associated with your account, check the box labeled “Revoke all API keys.” This will disable all existing keys for your devices.
5. **Finalize the Change:** Click on the “Change Password” button to save your new password.

How to delete your account

1. **Access Your Account Settings:** Click on your username located at the upper left corner of the screen to open the Account Settings menu.
2. **Initiate the Account Deletion:** In the Account Settings menu, click on the “Delete Account” button.
3. **Confirm Account Deletion:** A confirmation prompt will appear. Confirm your decision by clicking the “Confirm Delete” button.
4. **Complete the Deletion:** Your account will be permanently deleted, and all associated data will be removed.

Troubleshooting

ESP32’s light keeps blinking b quickly.

The ESP32’s light blinking quickly indicates that the microcontroller is attempting to connect to your Wi-Fi network. If the blinking continues for more than a few minutes, it likely means that the Wi-Fi credentials stored in the firmware are incorrect. Please double-check the Wi-Fi settings, including the SSID and password, and update them if necessary.

Conclusion

Future Enhancements

- Switch to web sockets to support a higher number of appliances and reduce latency

Last revised in July 2024.